Exercise Project 5 - Transformer Networks Study and Experimentation

In this exercise, I explored how transformer networks work, both in theory and practice. Since I couldn't run the full neural machine translation model (due to lack of GPU) at first, I started with a simpler Keras example on text classification. Then I analyzed the architecture of transformers and tried the translation example to test various linguistic inputs.

Practical experiments

1. Keras Transformer for Text Classification

ex_5/KerasText.ipynb

I used the Keras example Text Classification with Transformer to classify movie reviews as positive or negative. The model has:

- **Token & Positional Embedding layers** convert words into vectors and encode word order.
- **Multi-Head Self-Attention** helps the model focus on different parts of the input.
- **Feedforward Network** basically adds more depth and transformation.
- **GlobalAveragePooling1D** and **Dense Layers** are used for final classification purposes

After training for 2 epochs, the model reached ~90% accuracy. I found this architecture intuitive and useful for understanding basic Transformer structure.

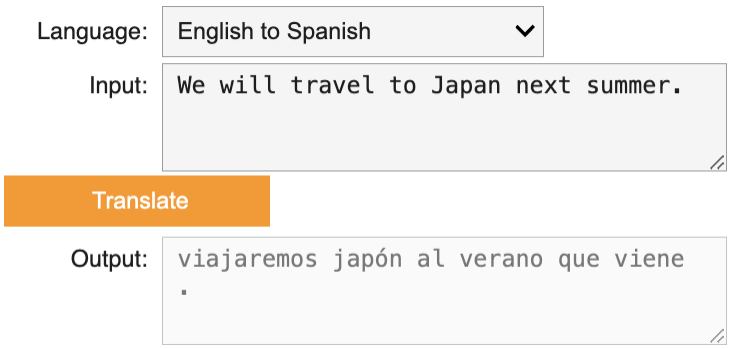
2. Keras Transformer for Translation

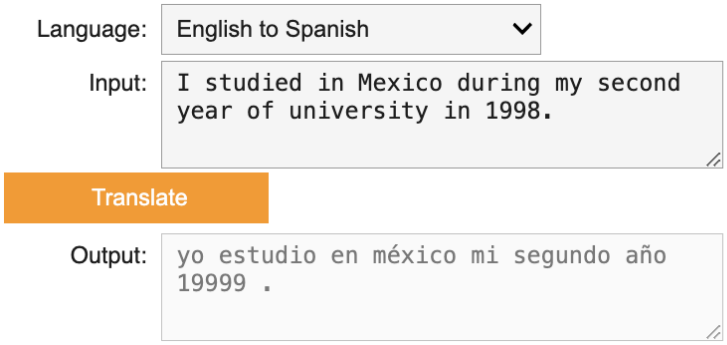
ex_5/KerasTransaltion.ipynb

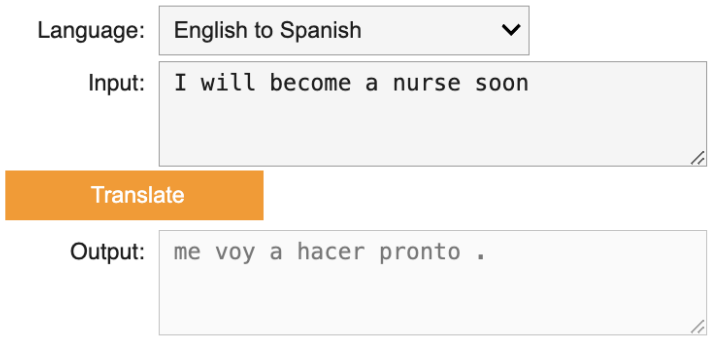
ex_5/KerasTransaltion_withGUI.ipynb

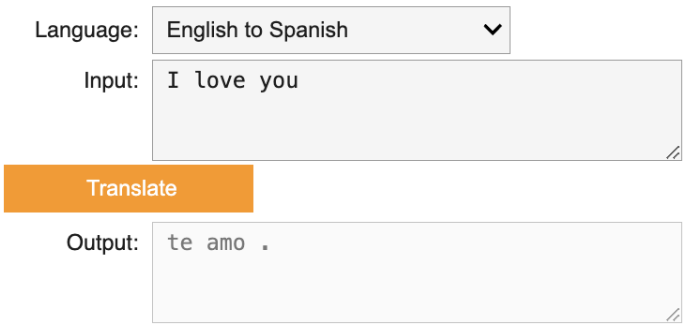
I also experimented with English-Spanish translation using Transformer. While it worked for simple inputs like “I love you → Te amo,” it really struggled with:

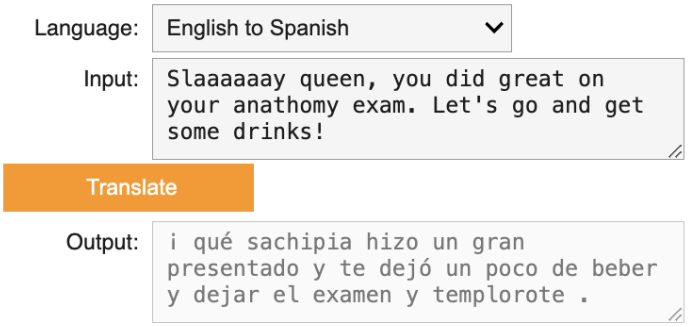
- Numbers and also years (“second year 19999”)
- Longer or slang sentences (produced nonsensical phrases)
- Tense and negation (merged two tenses into one wrong sentence)











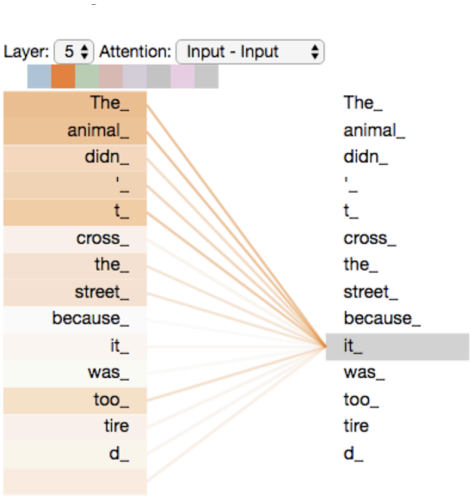
I wanted to experiment with some slang, but besides capturing some of the words in the input it did not do too well.

"What sachipia made a great presentation and left you a little to drink and leave the exam and templotote."

To be honest the output when translating longer sentences reminds me of a person with Wernicke's aphasia. It is a type of language disorder that often occurs after damage to the Wernicke's area in the brain, located in the left temporal lobe. People with Wernicke's aphasia can speak in long sentences fluently, but their words lack meaning and contain made-up words or incorrect word substitutions.

Theoretical Understanding of Transformers

1. Structure of a Transformer

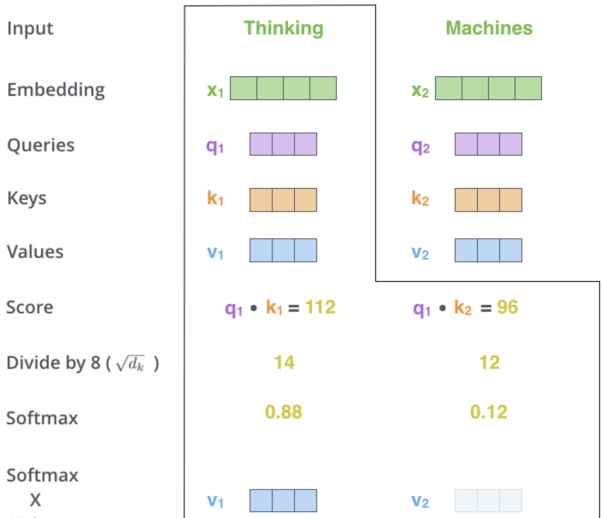


The Transformer architecture is built to handle **sequence-to-sequence tasks** like for instance translation. It takes an input sentence, processes it, and generates a corresponding output sentence. The architecture is mainly divided into two blocks:

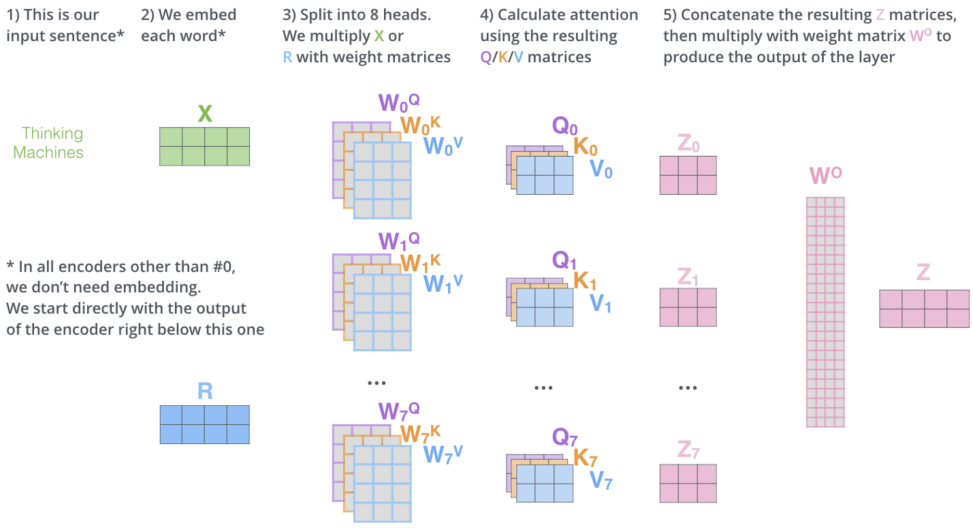
Encoder Block (1)

The **encoder** receives the input sentence (e.g., "Je suis étudiant") and turns it into a complex representation that captures not just the English meanings but also the relationships between the words.

Each encoder layer contains the following:



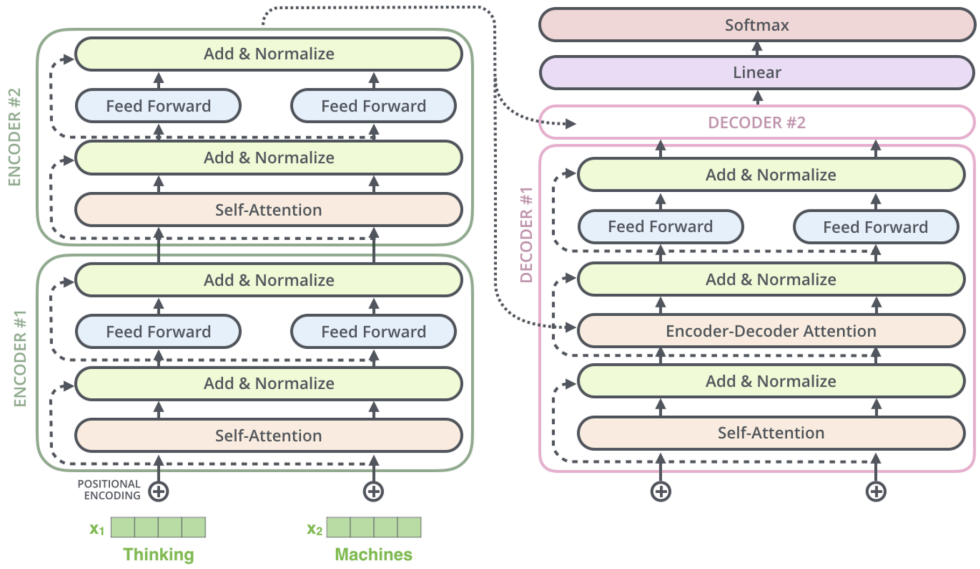
- **Input Embedding**: Converts words into numerical vectors. For example, "Je" becomes something like `[0.1, 0.2, 0.3]`. (That's how ChatGPT explained to me)
- **Positional Encoding**: Adds position info to these vectors because transformers don't naturally know the order of words.
- **Self-Attention Layer**: Helps each word "pay attention" to the other words in the sentence. "suis" will consider both "Je" and "étudiant" while forming its meaning.
- **Feedforward Layer**: A small neural network that further processes the output from the attention layer.
- **Layer Normalization + Dropout**: Helps the model train better and avoid overfitting, which I have used in previous project, so this was similar to me.



All encoder layers are stacked (often 6–12 times in real models), and the output of the final encoder is passed to the decoder.

Decoder Block (2)

The **decoder** takes the encoder's output and uses it to generate the target sentence step by step (like the translation in English: "I am a student").



Each decoder layer includes:

- The same components as the encoder: embedding, positional encoding, self-attention, feedforward, normalization, and dropout.
- **PLUS a Masked Self-Attention**, which basically prevents the model from seeing future words in the output. While generating word 1, it shouldn't know what word 2 or 3,4,5.. will be.
- **PLUS a Encoder-Decoder Attention** allows the decoder to attend to encoder outputs, so it knows which parts of the input sentence are relevant at each generation step.

2. How does Self-Attention work (With Examples, how I explained to myself)

Self-attention is the **core idea** of transformers. It allows each word to "look at" other words in the sentence and decide how important they are.

This was my favourite example:

Sentence: *"Je suis étudiant"* (I am a student in english)
Embedding size: Let's say each word becomes a 3D vector.

Step 1: Creating Q, K, V Vectors

Each word is turned into 3 vectors basically:

- **Q**: Query – what this word wants to know.
- **K**: Key – what this word offers.
- **V**: Value – the actual content carried by this word.

Example for "Je":

- Q = `[0.2, 0.3, 0.5]`
- K = `[0.1, 0.4, 0.3]`
- V = `[0.3, 0.2, 0.7]`

Step 2: Calculate Attention Scores

We compare "Je"'s Q with every other word's K using the dot product:

- `Je ↔ Je` → score = 0.29
- `Je ↔ suis` → score = 0.43
- `Je ↔ étudiant` → score = 0.67

Step 3: Apply Softmax

Convert these scores to probabilities:

```
Softmax([0.29, 0.43, 0.67]) = [0.24, 0.29, 0.47]
```

Now the model knows it should pay 24% attention to “Je” itself, 29% to “suis”, and 47% to “étudiant”.

Step 4: Weighted Sum

Next we multiply each value vector (V) by its attention weight and sum them up. This gives a **new vector** for “Je” that now contains contextual information from the full sentence.

"Je" now knows it's part of the phrase "Je suis étudiant" and its meaning is updated accordingly.

3. Decoder differences

While the encoder processes the input sentence, the **decoder** generates the output sentence word by word.

Key differences:

1. **Masked Self-Attention**:
   Unlike the encoder, the decoder isn’t allowed to see the entire output sequence. While generating the second word, it can only look at the first one, this is enforced using **masking**.
2. **Encoder-Decoder Attention**:
   Every decoder layer also has access to the encoder output. It uses this to “focus” on relevant input words. For example, while generating “student,” the decoder may focus on “étudiant.”

3. **Final Steps**:

   After passing through all decoder layers, the result goes through:

   - A **linear layer** that maps it to the size of the output vocabulary (like approx. 10,000 words)
   - A **softmax** function that picks the word with the highest probability.

Summary: How Transformers process data?

Input

1. Text is tokenized and converted into **embedding vectors**
2. **Positional encoding** is added to keep track of word order

Encoder

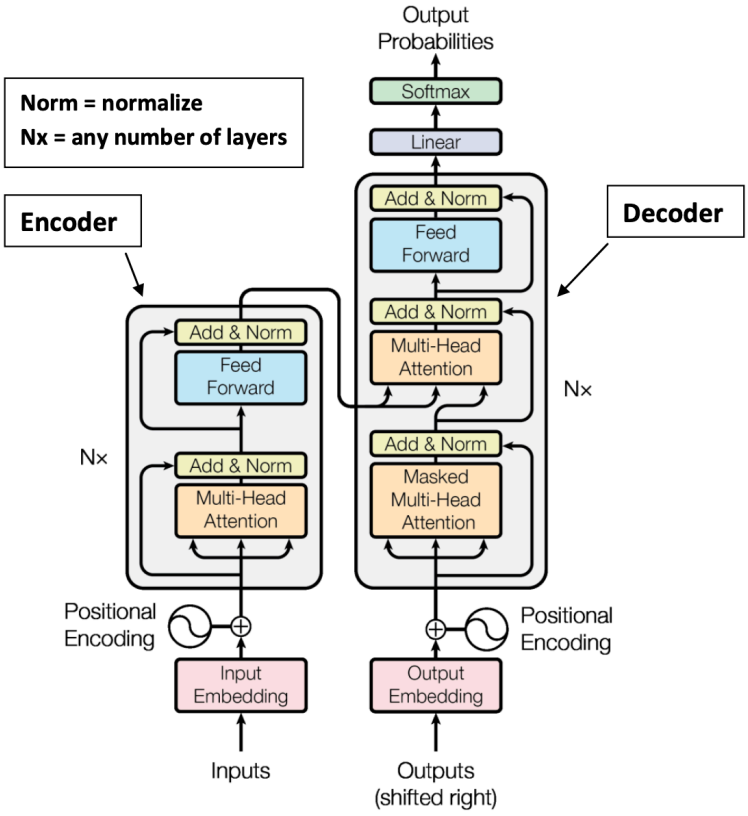
3. Each word goes through **self-attention**, deciding how much to pay attention to other words
4. **Feedforward network** processes the attention output
5. Output from all encoder layers is passed to the decoder

Decoder

6. Decoder generates output **one word at a time**, and uses
   - Its own masked self-attention and
   - Encoder output (via encoder-decoder attention)
7. Final vector is passed to a **linear layer** and then **softmax** to pick the next word

Why is it better than RNNs??

- RNNs process one word at a time, Transformers handle full sequences in parallel
- Transformers can connect “I” to “doctor” in “I want to become a doctor” even if they are far apart positionally.



- Avoid vanishing gradient problem, by keeping the training stable

Reflection on the Transformer architecture diagram

When I first saw the diagram, there were overwhelming boxes labeled "Q," "K," "V," random arrows everywhere, lots of stacks. It looked like a maze that I will never understand.

Now, I can say that a few things have cleared up. I get that the **Multi-head attention** makes sense in a way of each head learns different relationships (e.g., subject-verb, tense, position) and **Encoder and decoder are modular** and they repeat similar blocks with minor changes like masking for instance. Basically, the data flow goes from input embedding to attention to feedforward to prediction.

I can now confidently look at the transformer diagram and explain what each part does. It's not just a "black box" anymore, experimenting with Transformer code also helped me understand what they are about.