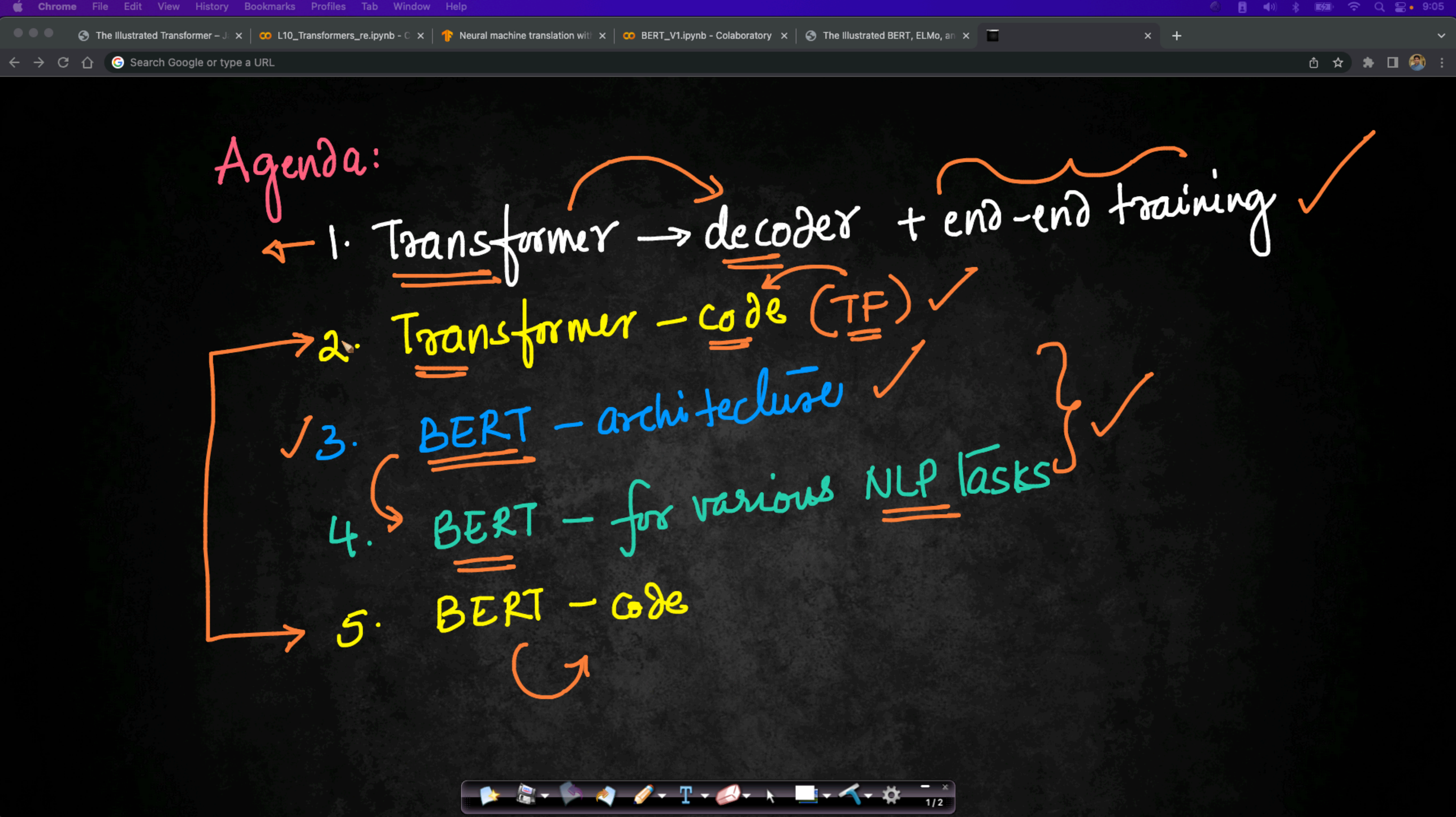
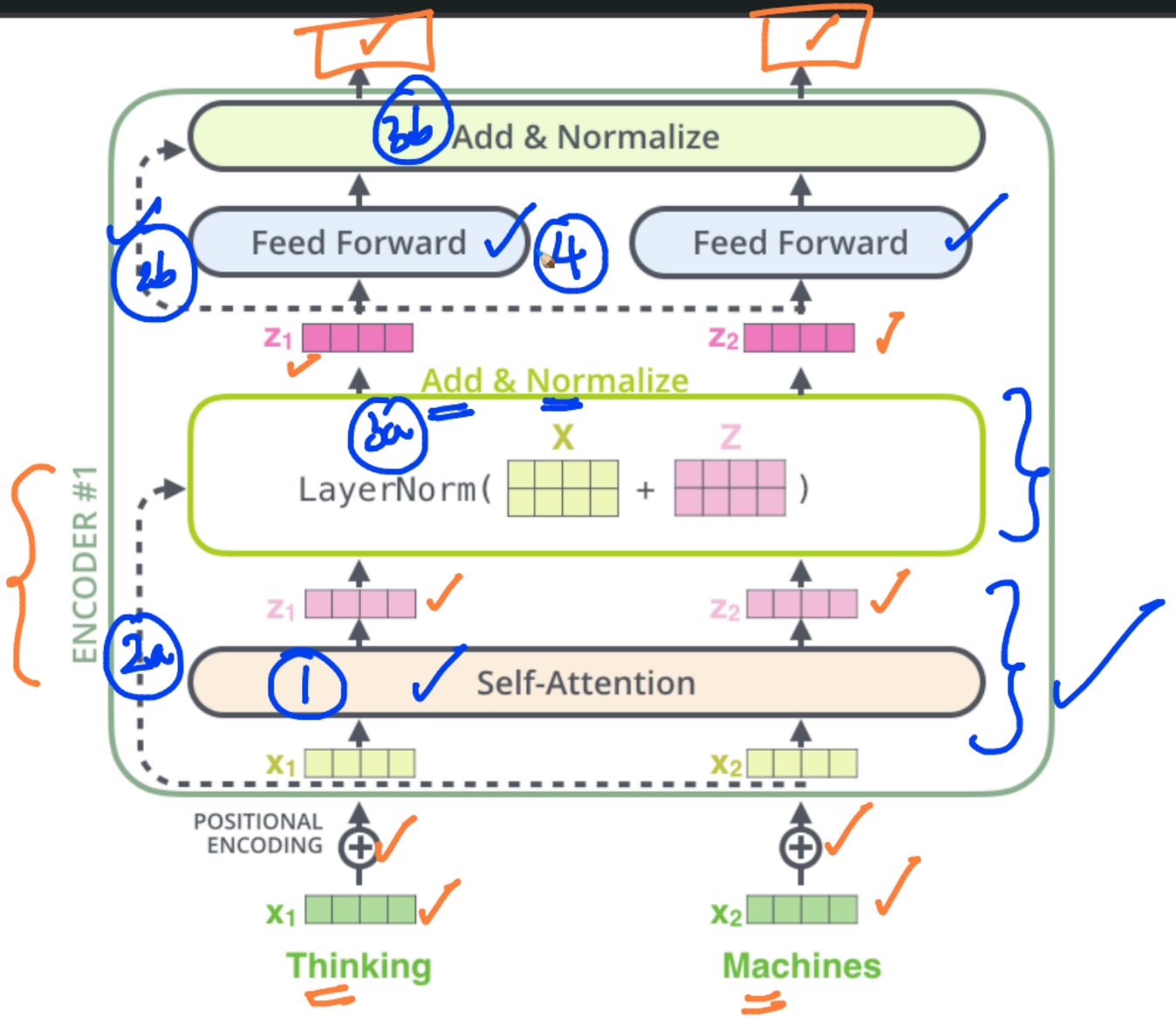
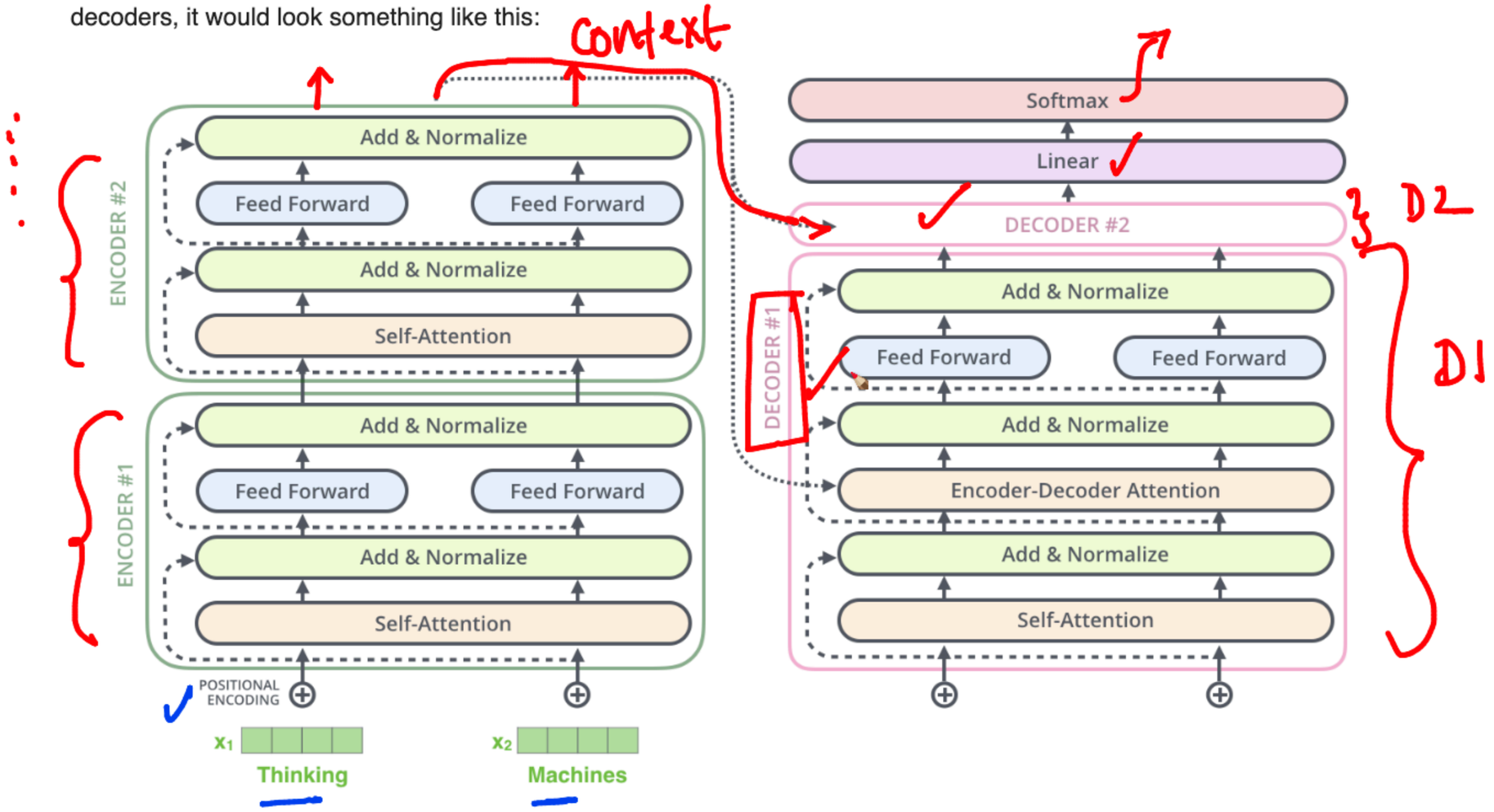

Agenda:
1. Transformer → decoder + end-end training
2. Transformer - code (TF)
3. BERT - architecture
4. BERT - for various NLP tasks
5. BERT - code



This goes for the sub-layers of the decoder as well. If we're to think of a Transformer of 2 stacked encoders and decoders, it would look something like this:

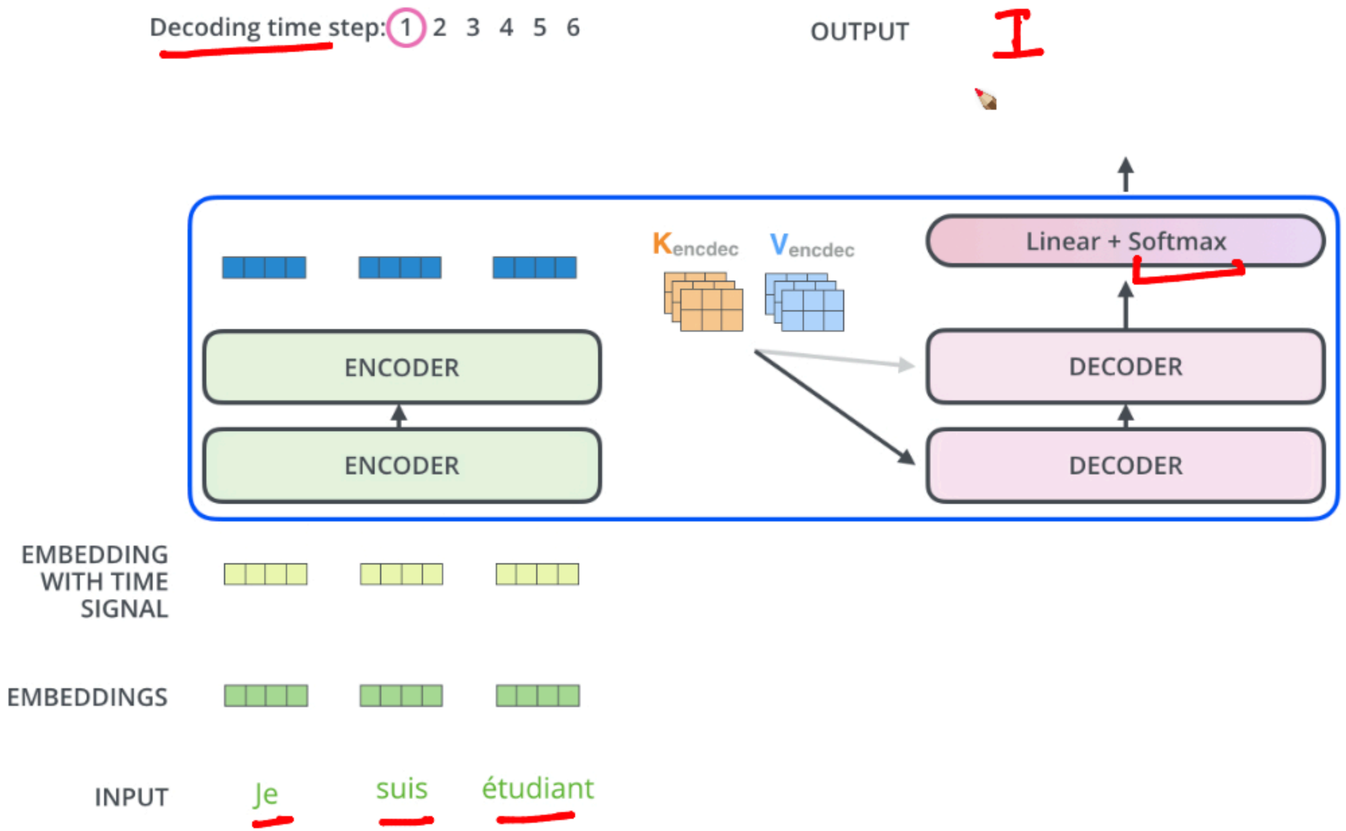
This goes for the sub-layers of the decoder as well. If we're to think of a Transformer of 2 stacked encoders and decoders, it would look something like this:



The Decoder Side

Now that we've covered the components of decoders work as well. B

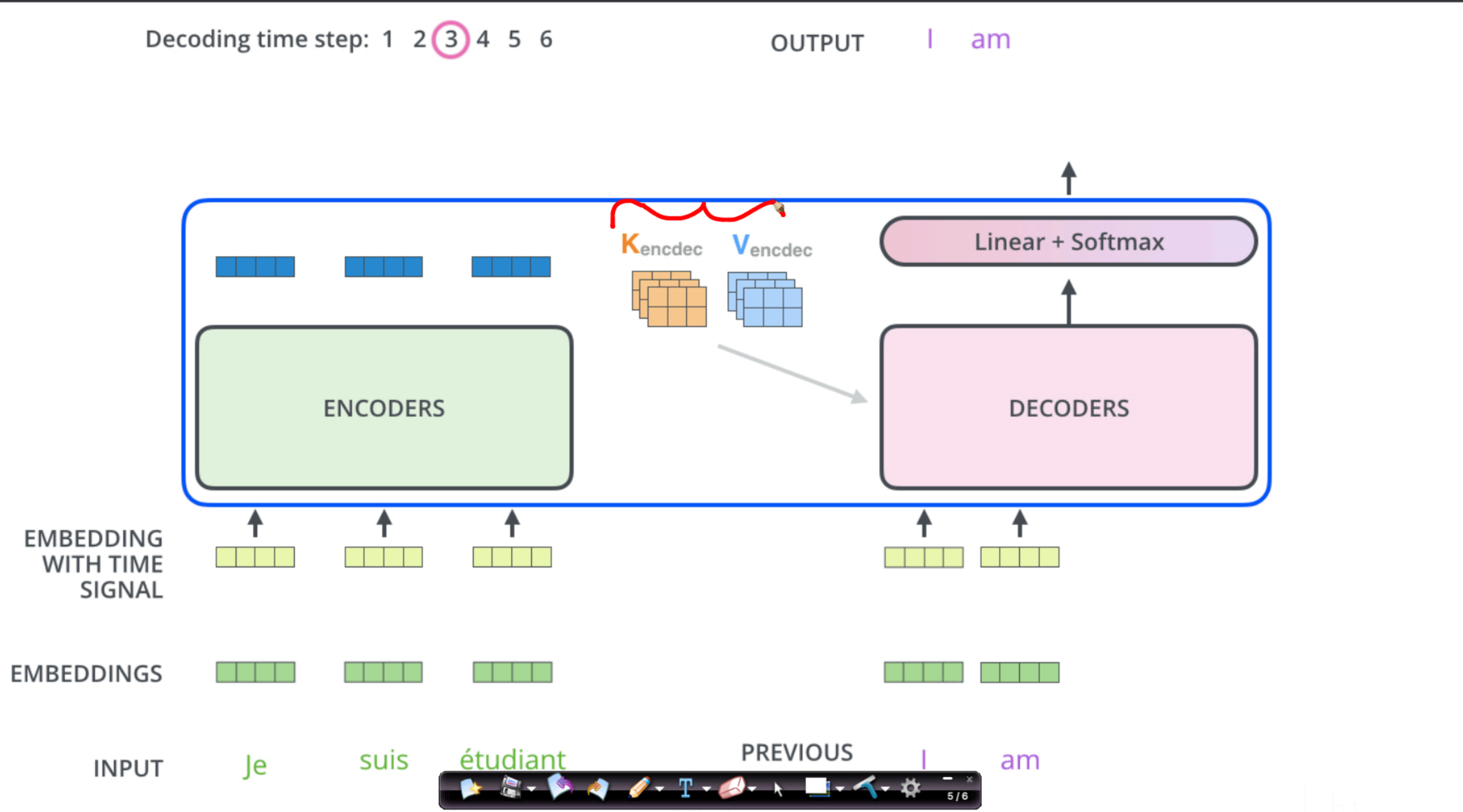
The encoder start by processing the input sequence. The output of the top encoder is then transformed into a set of attention vectors K and V. These are to be used by each decoder in its "encoder-decoder attention" layer which helps the decoder focus on appropriate places in the input sequence:

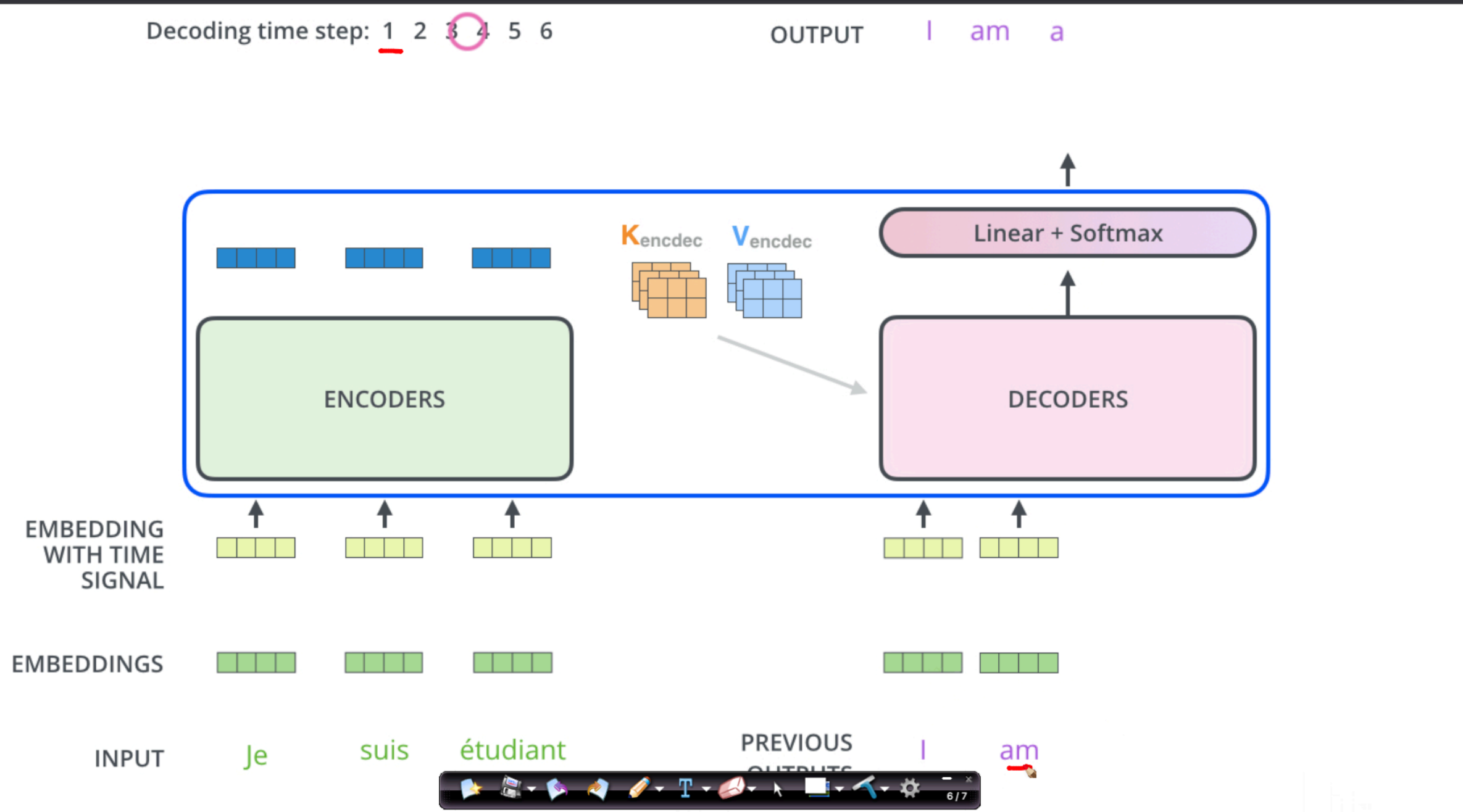


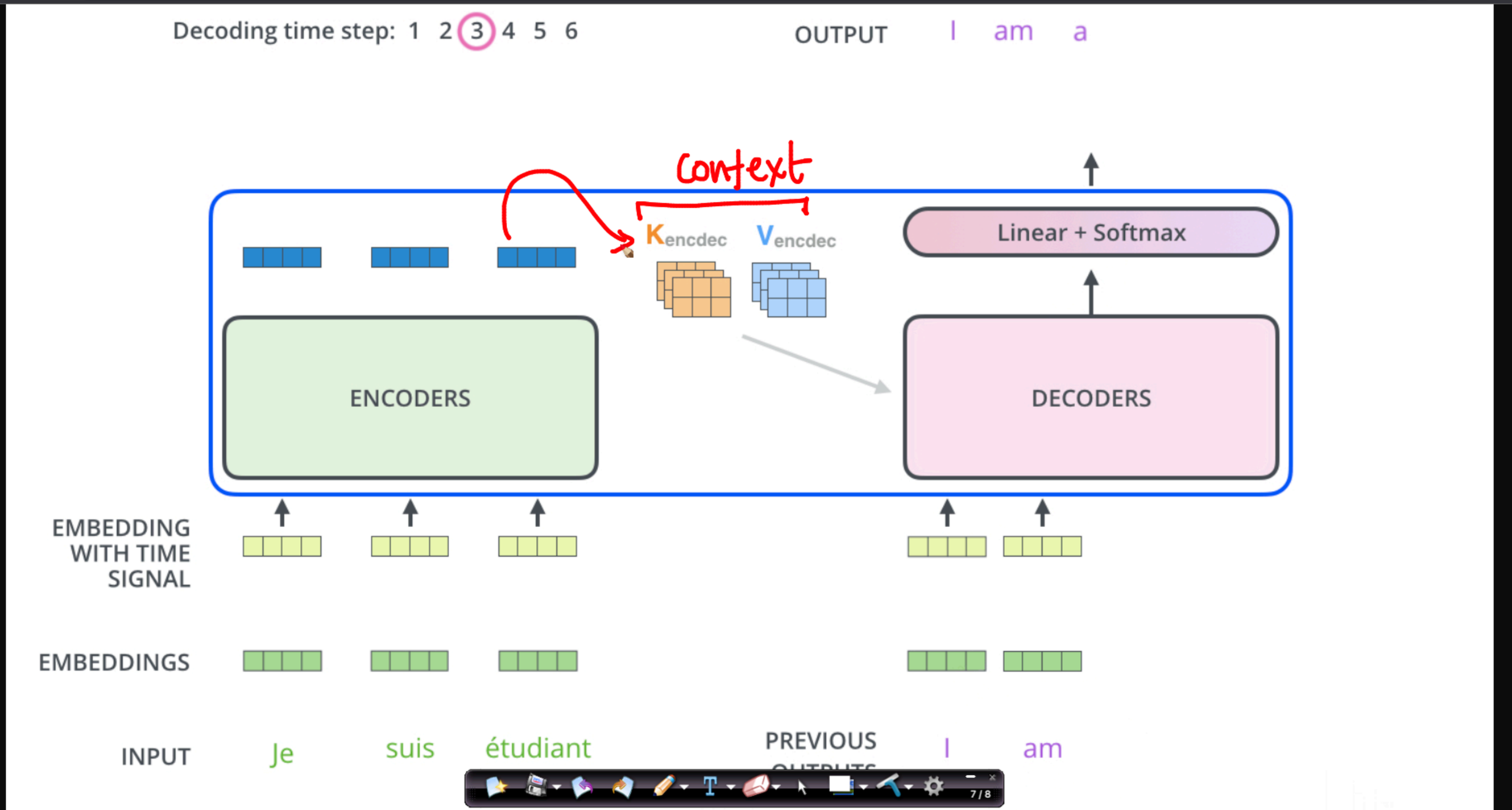
Seq – Seq

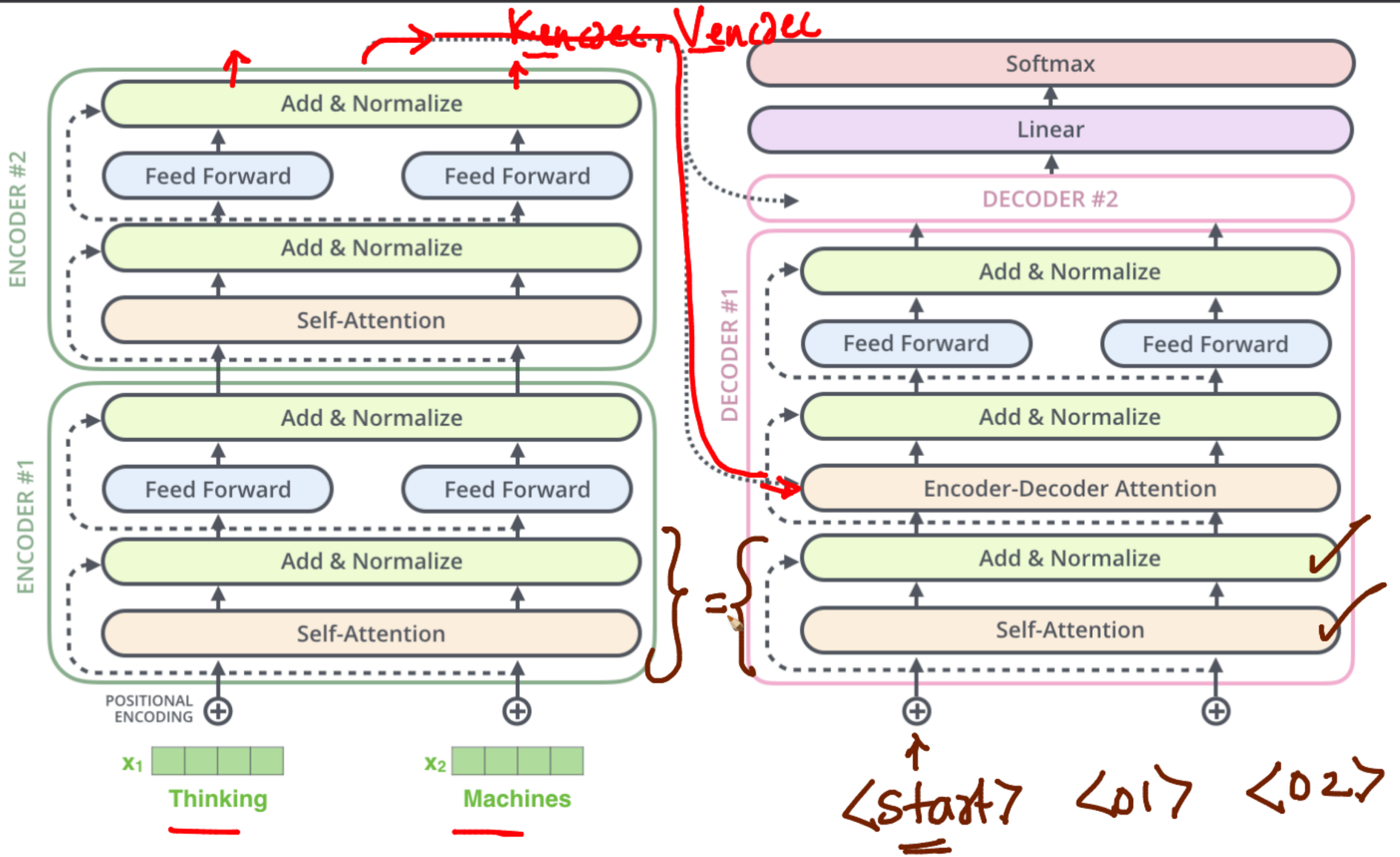
After finishing the encoding phase, we begin the decoding phase. Each step in the decoding phase outputs an element from the output sequence (the English translation sentence in this case).

The following steps repeat the process until a special symbol is reached indicating the transformer decoder has completed its output. The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding [redacted] coder inputs, we embed and





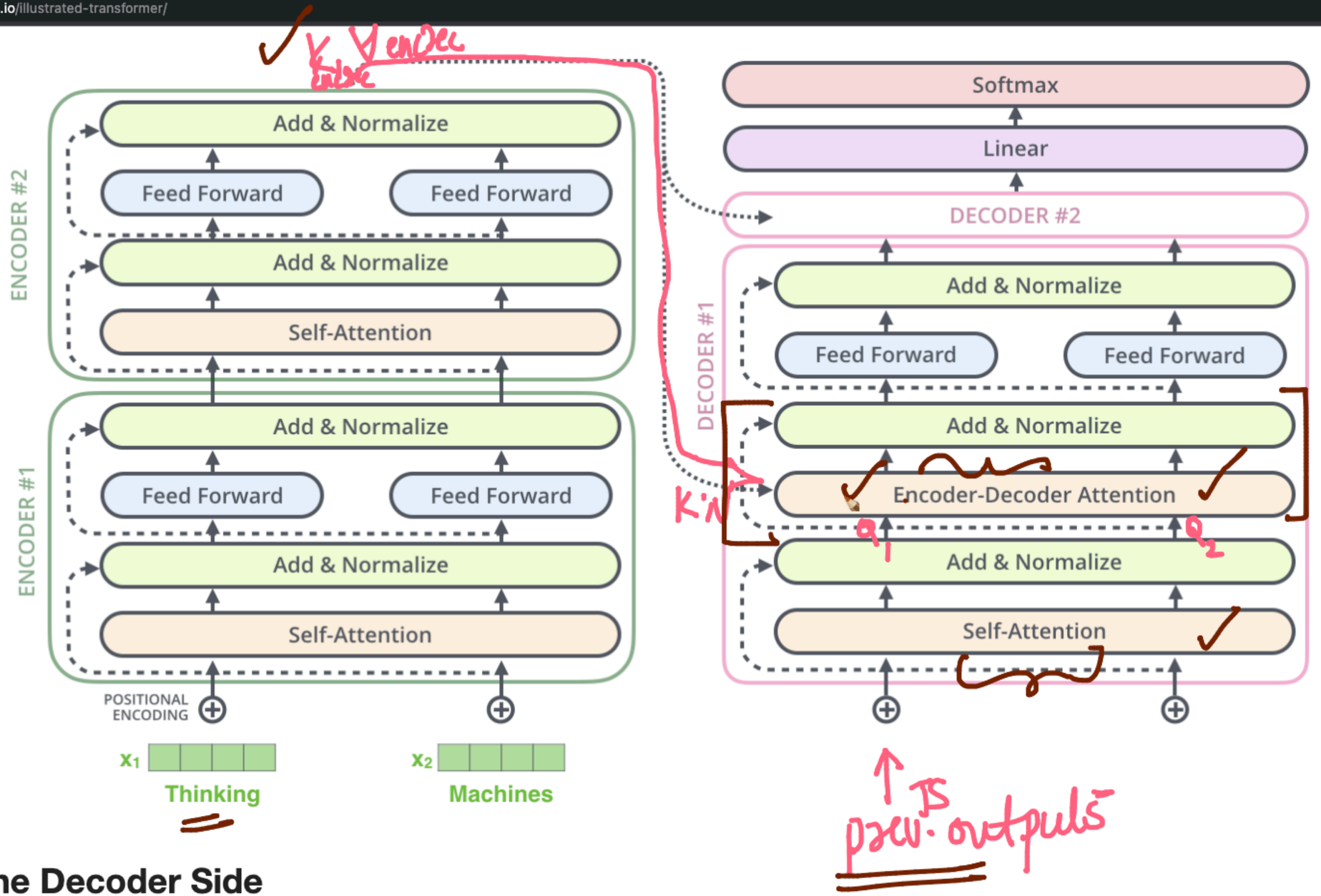




The Decoder Side

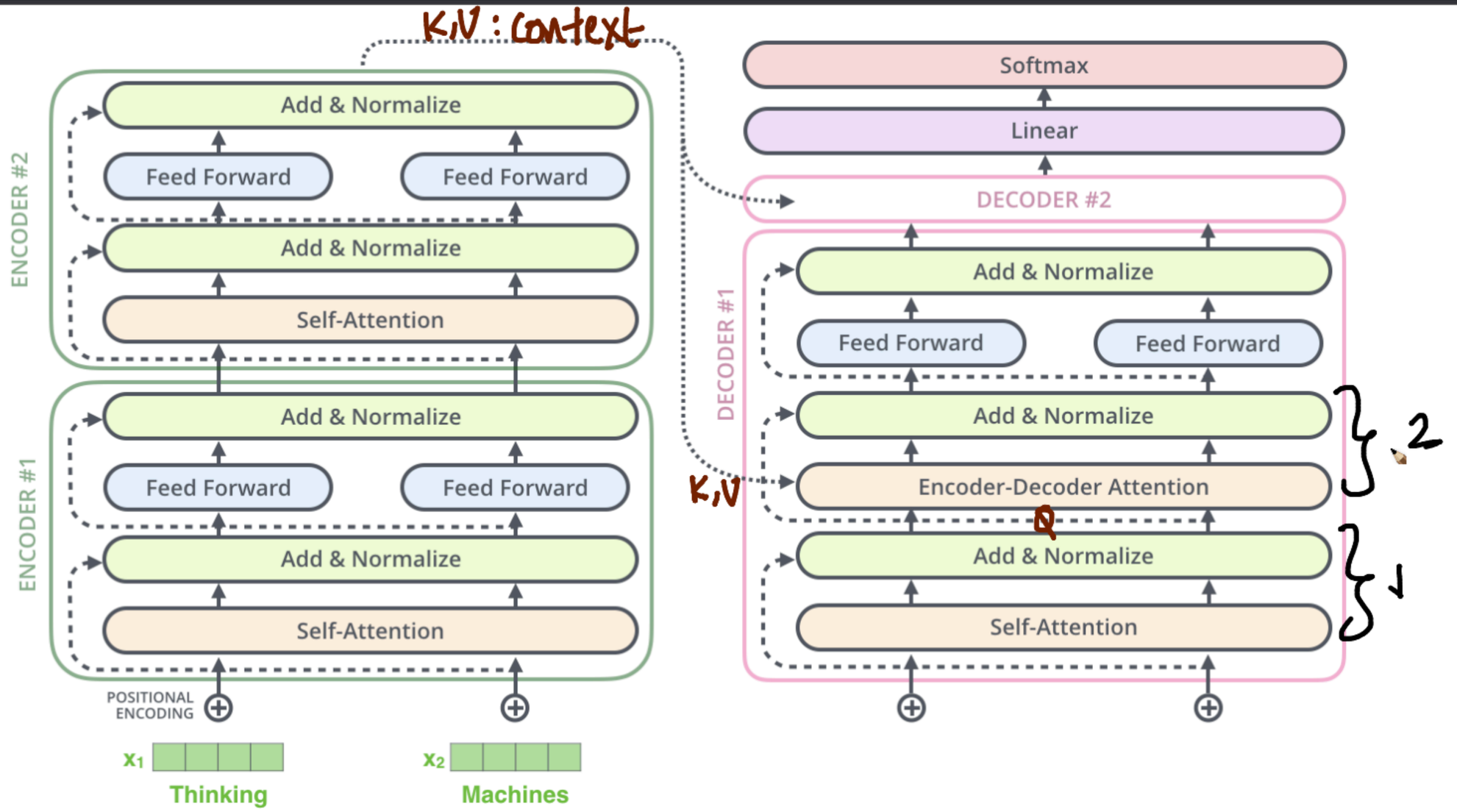
Now that we've covered most

how the components of



The Decoder Side

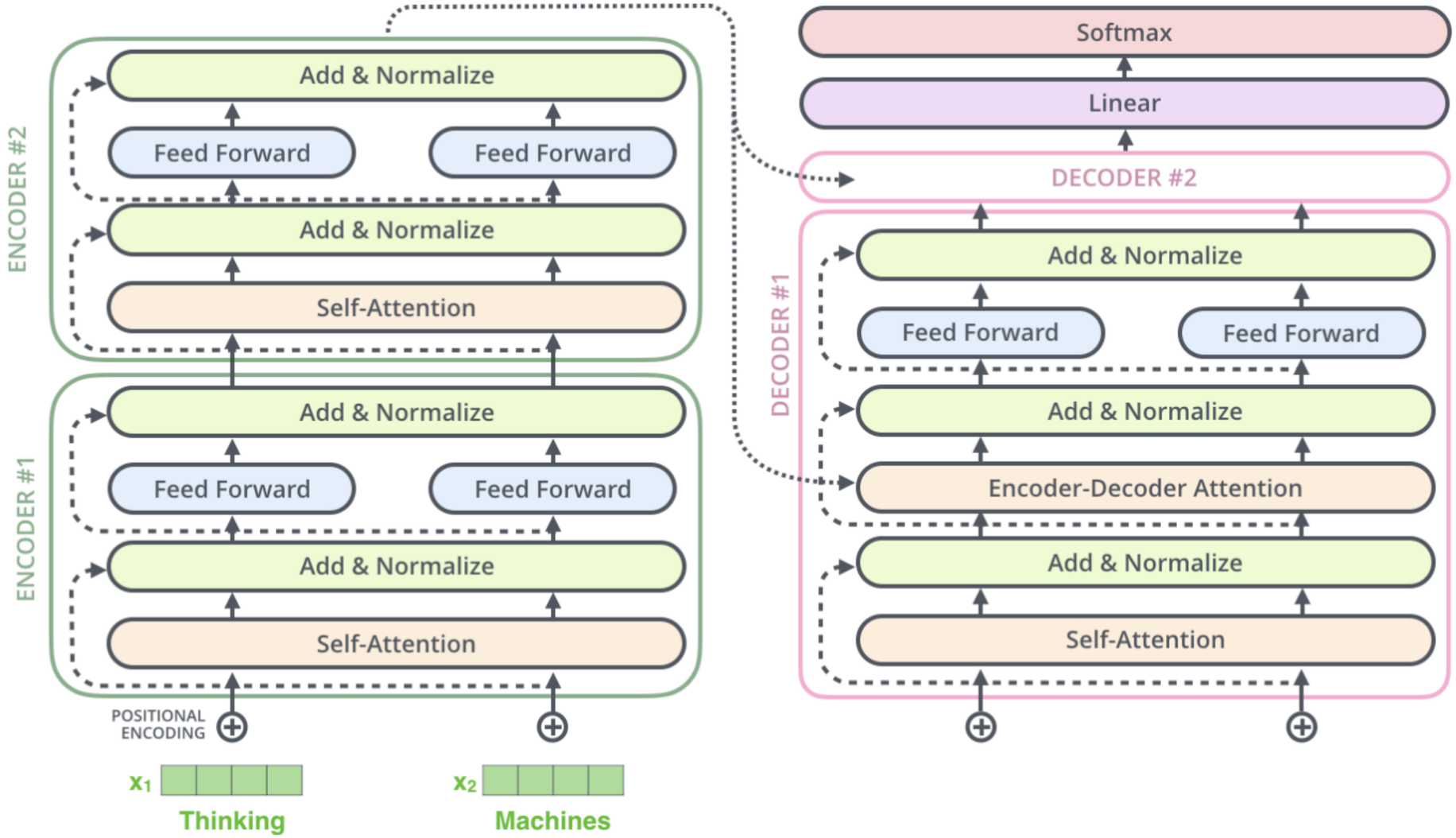
Now that we've covered most [illegible] how the components of



The Decoder Side

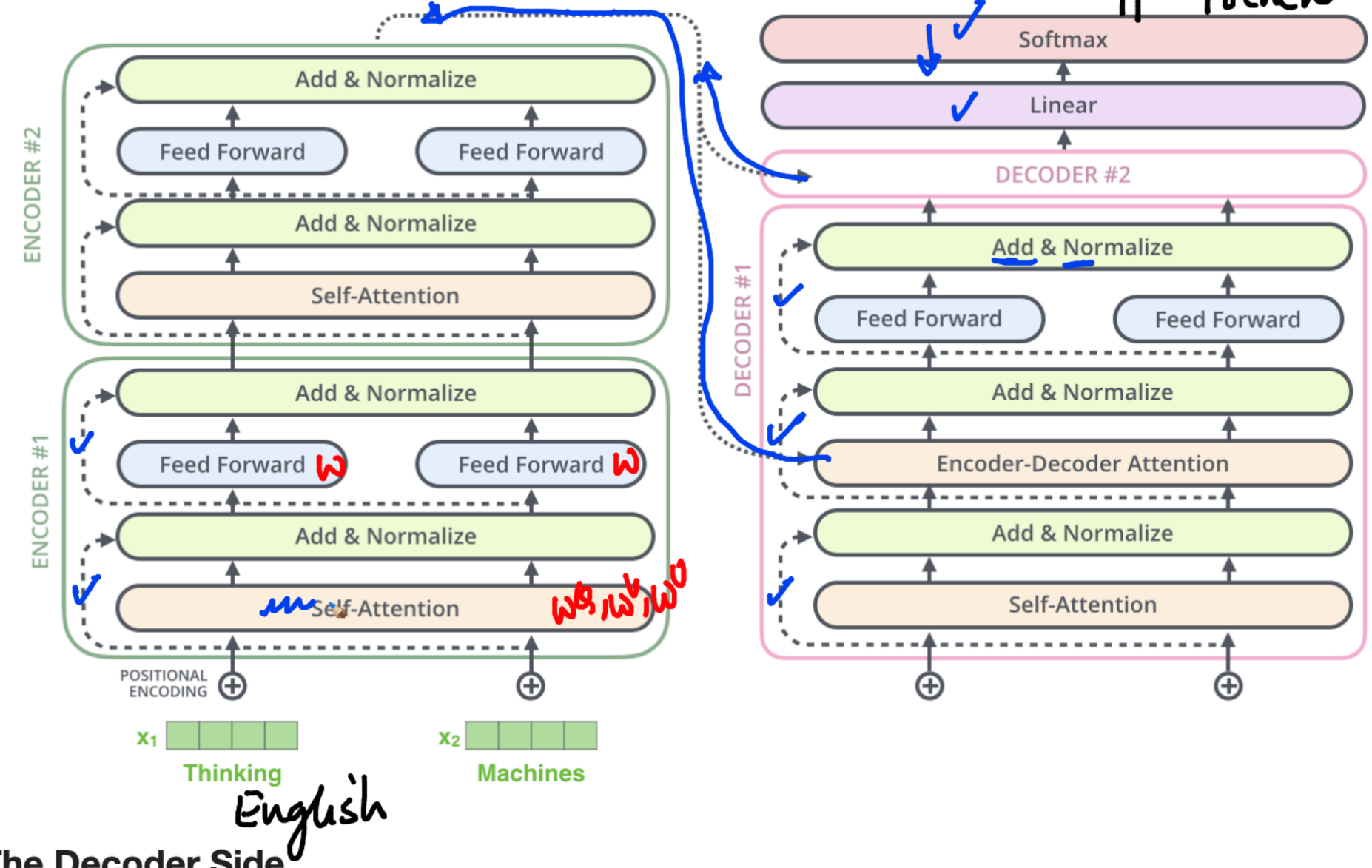
Now that we've covered most

how the components of



The Decoder Side

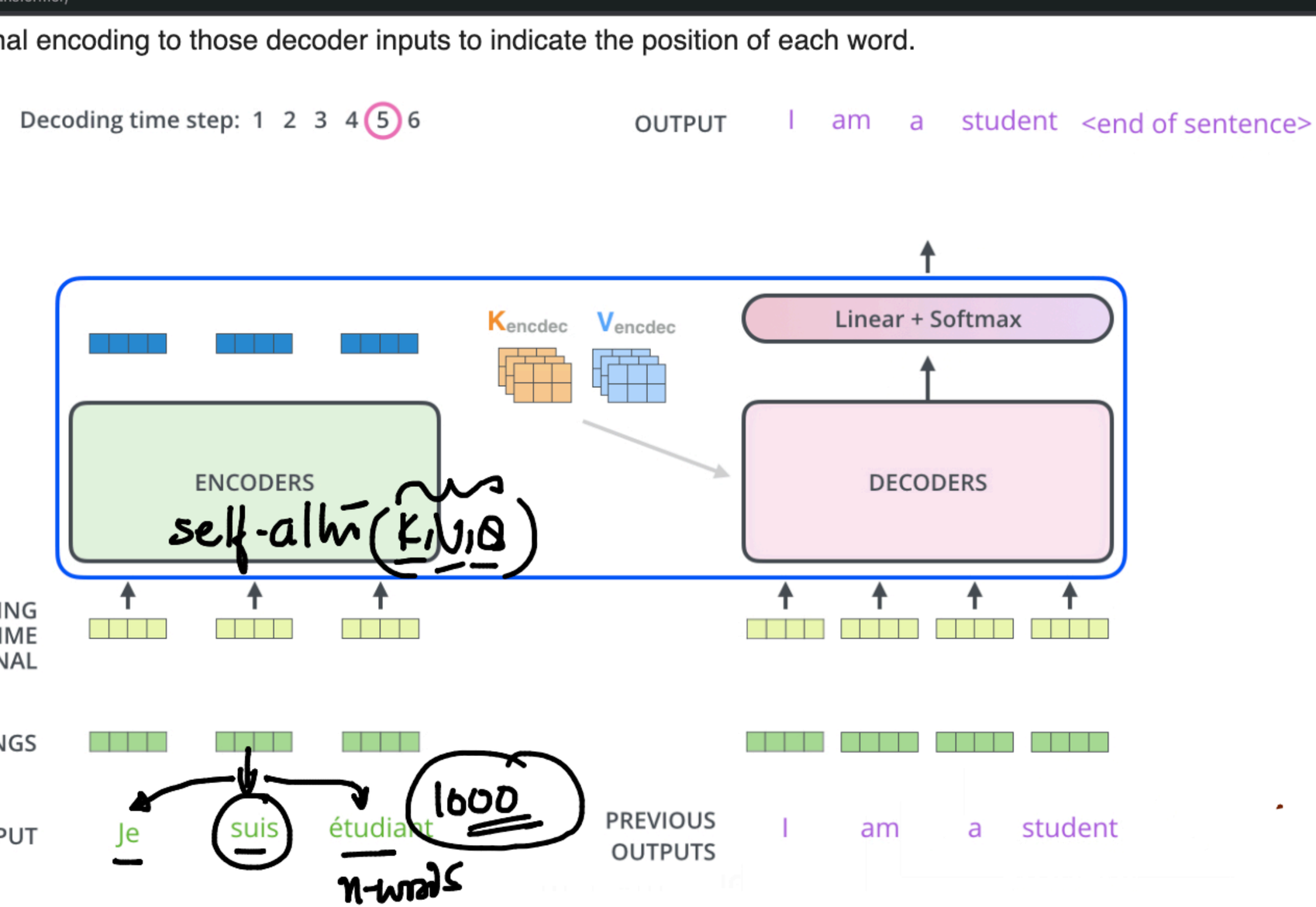
Now that we've covered most  how the components of



The Decoder Side

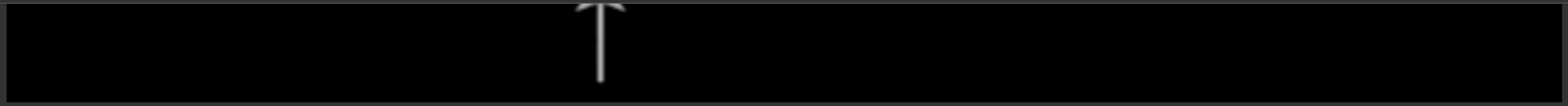
add positional encoding to those decoder inputs to indicate the position of each word.

drawback(s):
- comp. expensive (too many params)
- dec. timesteps could be many.
- O(n²) a/len



The self attention layers in the decoder operate in a slightly different way than the one in the encoder:

In the decoder, the self-attenti the output sequence. This is



Can you use this architecture for translation problem?

Lets see Time Complexity between RNN vs Attention

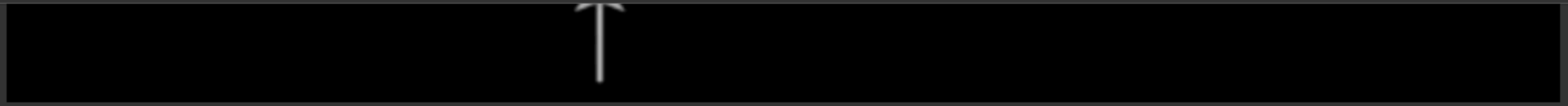
Architecture	Time Complexity per Layer	Sequential operations
RNN/LSTM	$O(n \cdot d^2)$	$O(n)$
Attention	$O(n^2 \cdot d)$	$O(1)$

- n: is the length of the sequence.
- d: is the dimension of the word representation (512, 1024 in generally)

In terms of computational complexity,

- when n<d, the attention layers are faster than the recursive layers

Quality comparison



Can you use this architecture for translation problem?

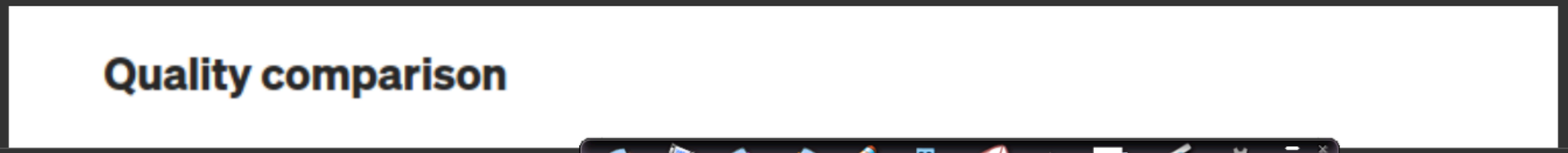
Lets see Time Complexity between RNN vs Attention

Architecture	Time Complexity per Layer	Sequential operations
RNN/LSTM	$O(n \cdot d^2)$	$O(n)$
Attention	$O(n^2 \cdot d)$	$O(1)$

- n: is the length of the sequence.
- d: is the dimension of the word representation (512, 1024 in generally)

In terms of computational complexity,

- when n<d, the attention layers are faster than the recursive layers

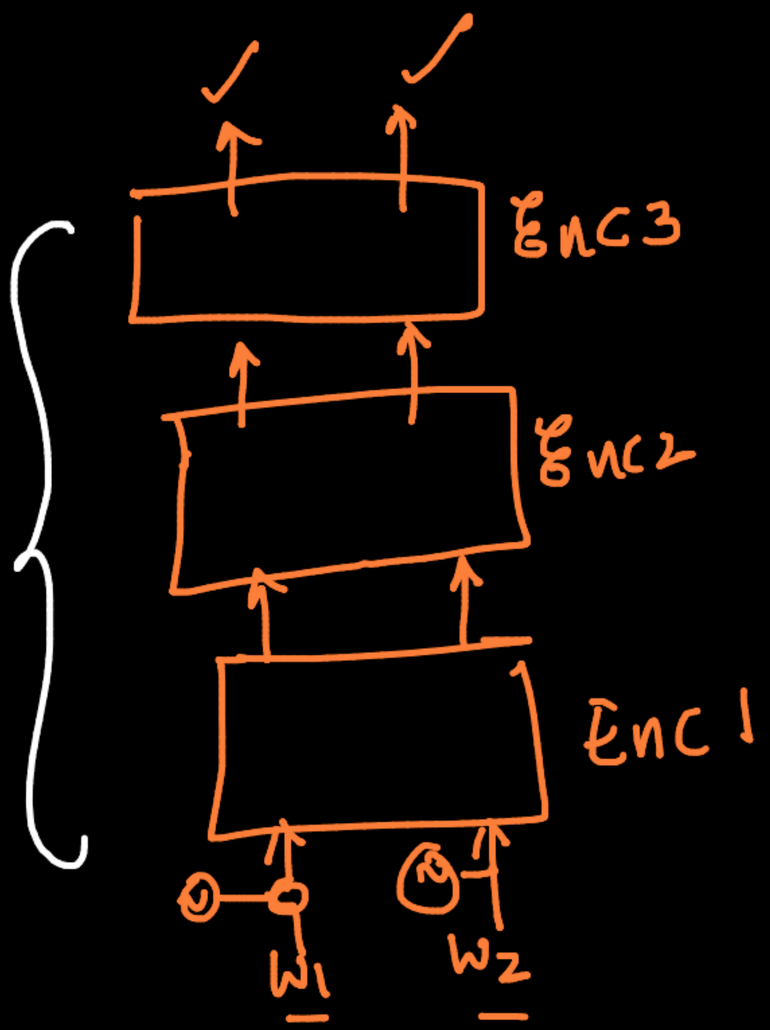


Transformer → ~~decoding~~ per time step

Encoder only ✓
↓
BERT

decoder only
↓
GPT

BERT

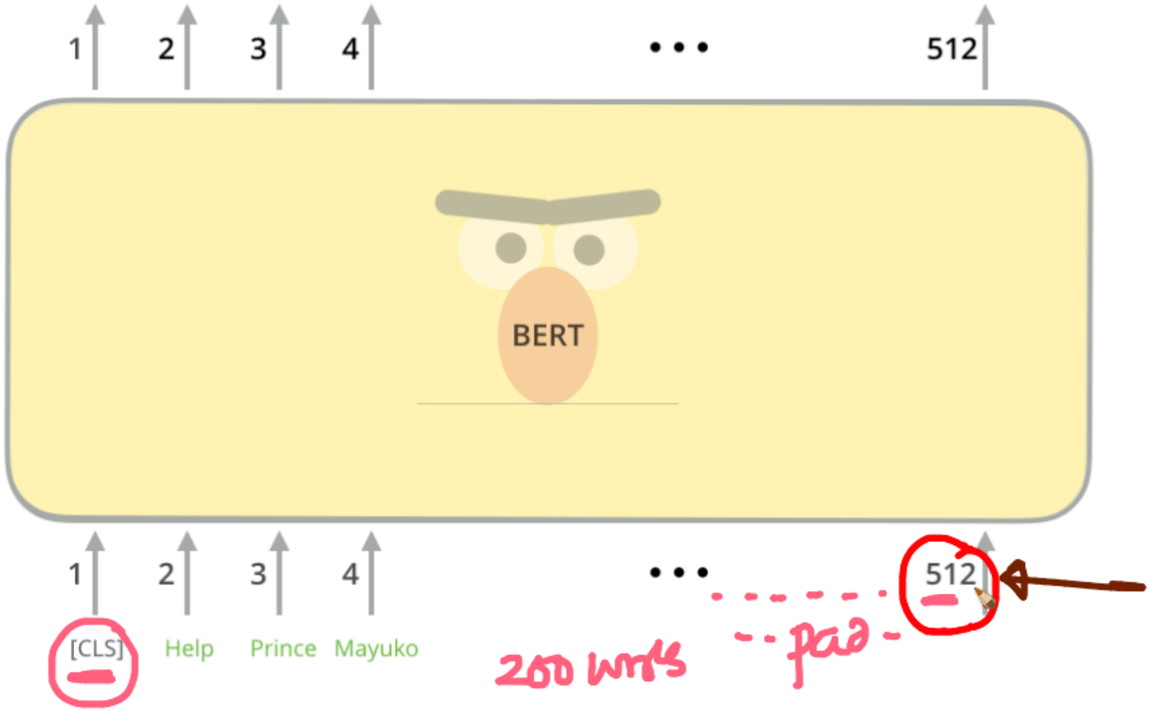


NLP:

✓ (1) Classfn (Multi-Class)

✓ (2) Seq 2 Seq

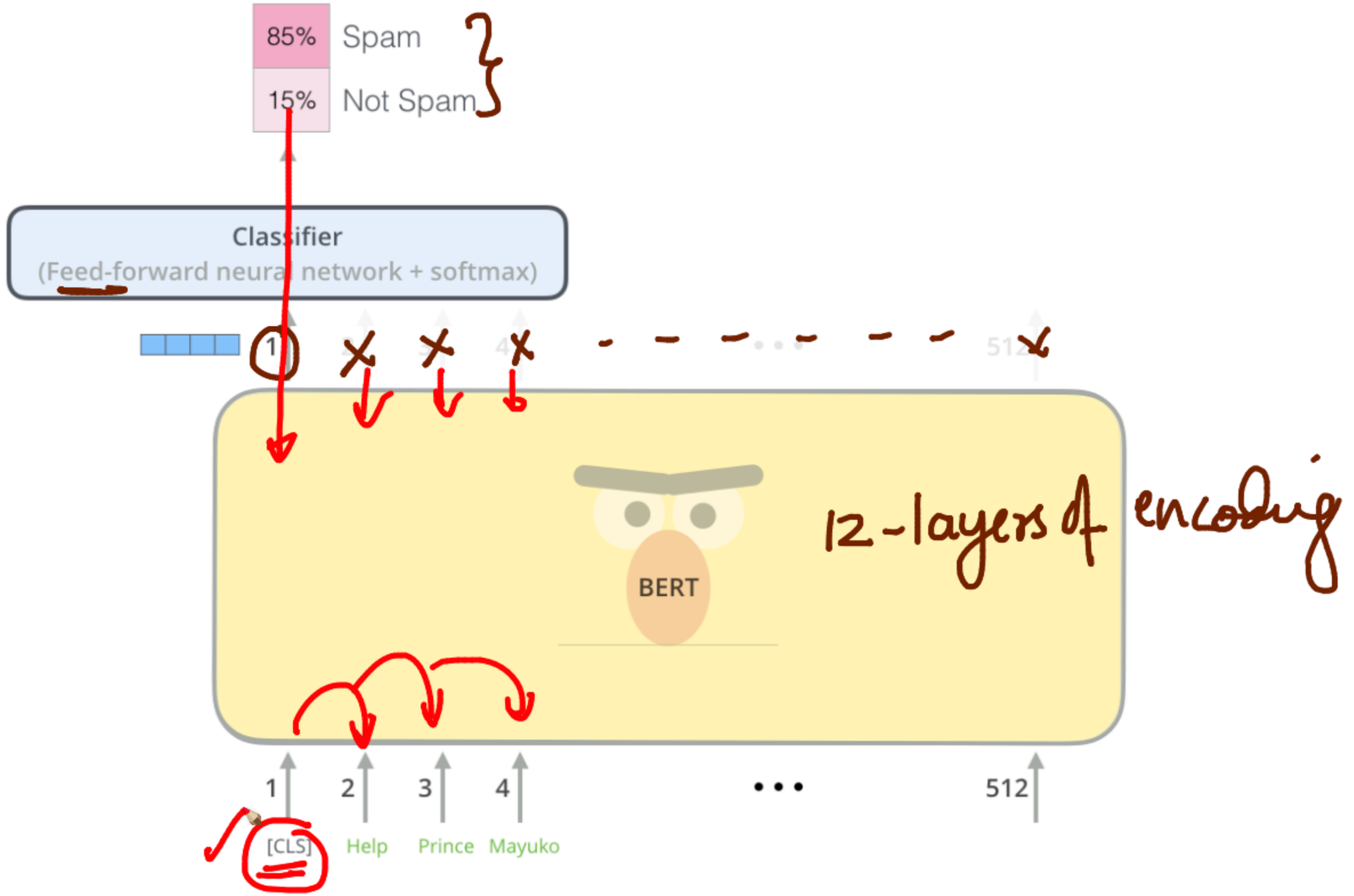
Model Inputs



The first input token is supplied with a special [CLS] token for reasons that will become apparent later on. CLS here stands for Classification.

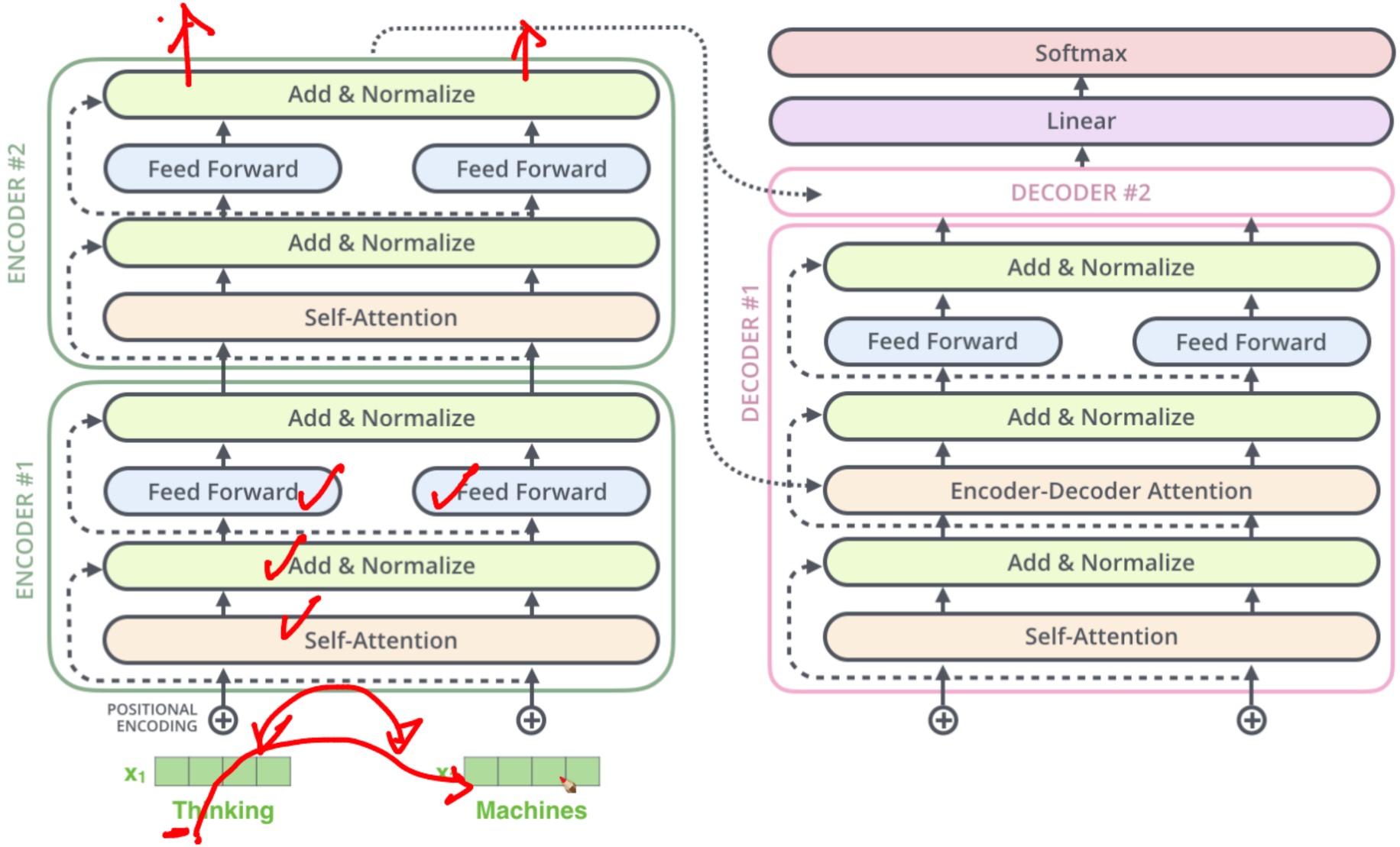
Just like the vanilla encoder of the transformer, BERT takes a sequence of words as input which keep flowing up the stack. Each layer applies self-attention, and passes its results through a feed-forward network, and then hands it off to the next encoder.

That vector can now be used as the input for a classifier of our choosing. The paper achieves great results by just using a single-layer neural network as the classifier.



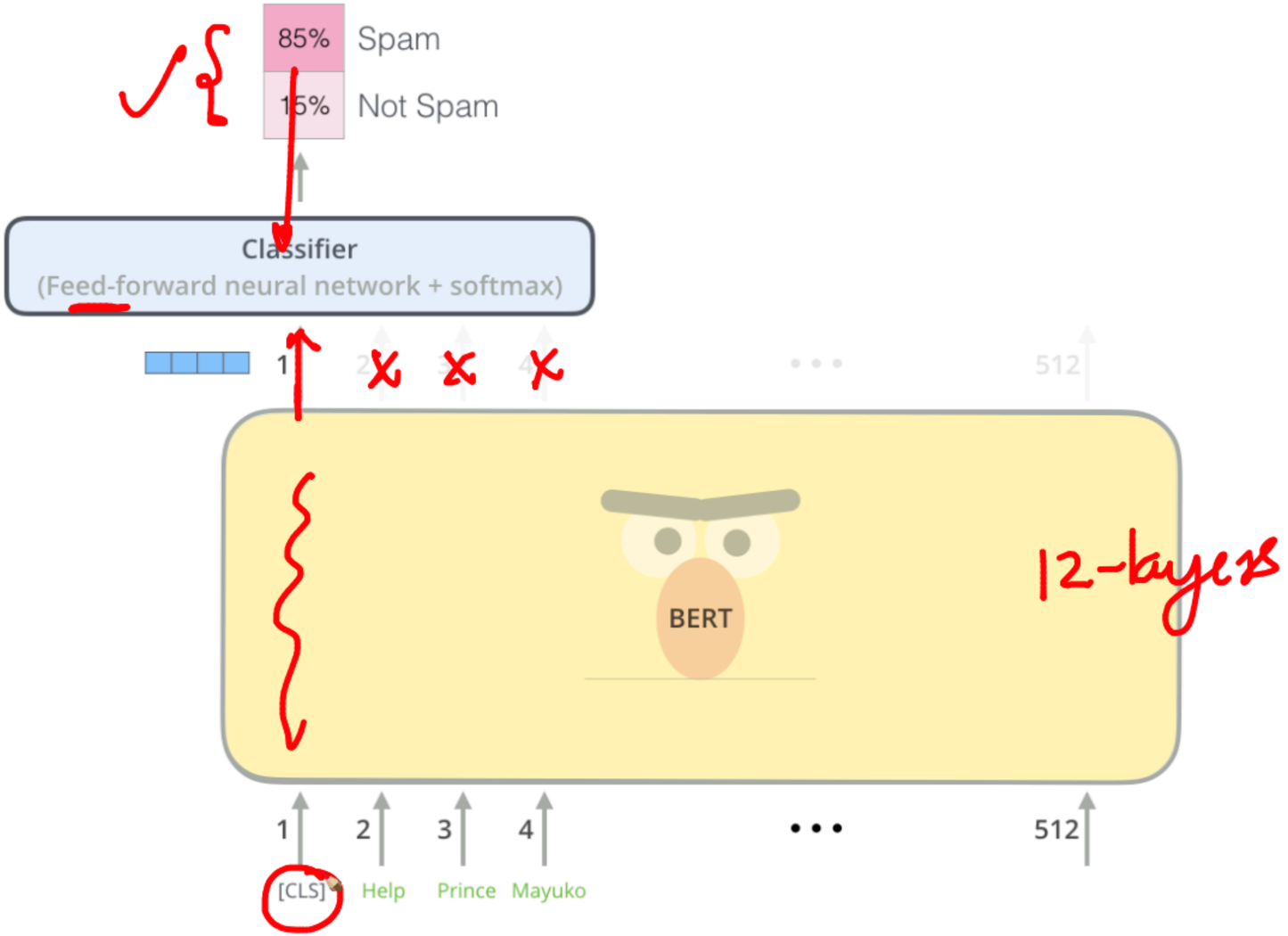
If you have more labels (for ex spam", "not spam", "social", and "promotion"), you just tweak the classifier network to have more output neurons and then pass through softmax.

This goes for the sub-layers of the decoder as well. If we're to think of a Transformer of 2 stacked encoders and decoders, it would look something like this:

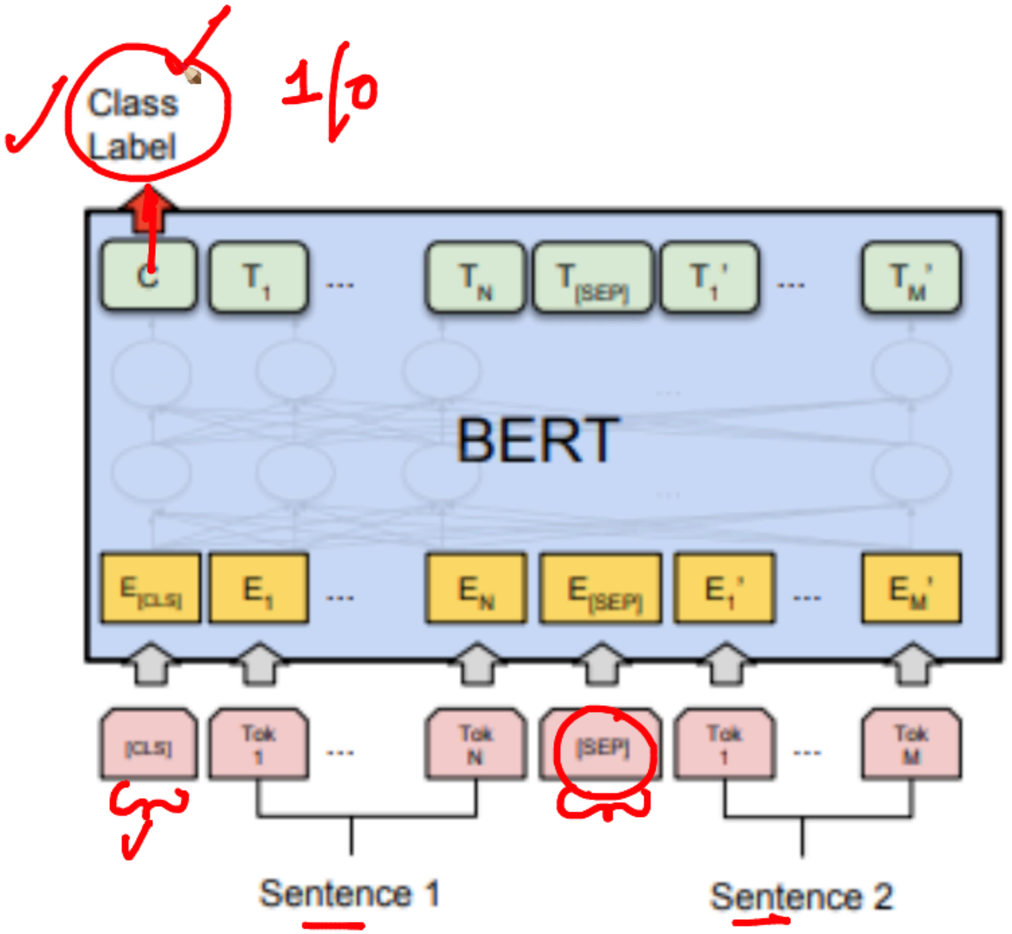


The Decoder Side

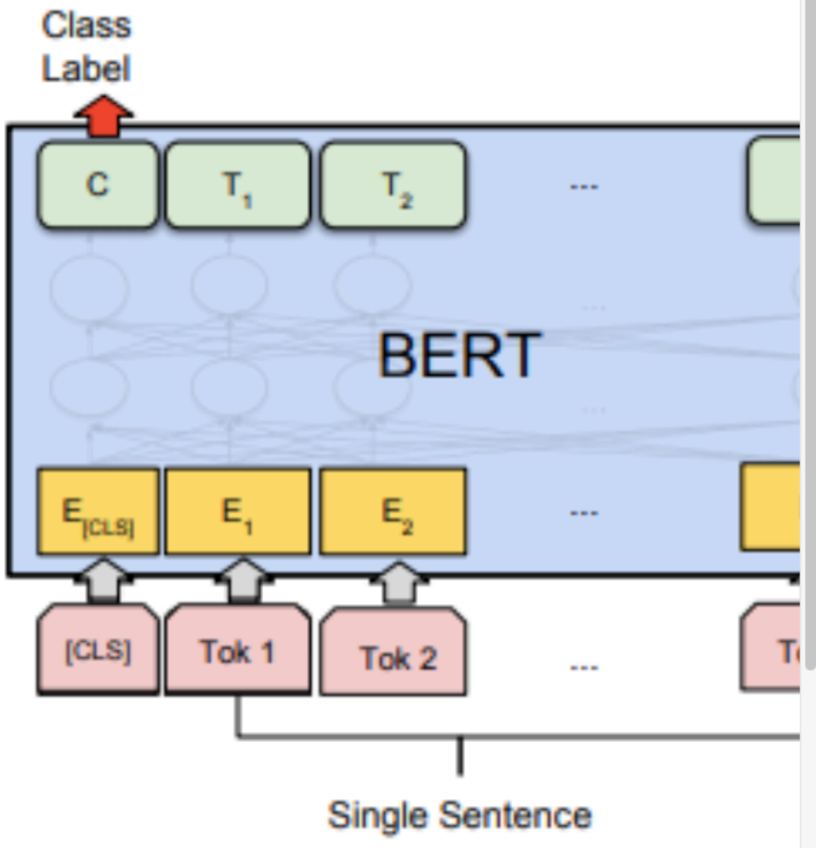
That vector can now be used as the input for a classifier of our choosing. The paper achieves great results by just using a single-layer neural network as the classifier.



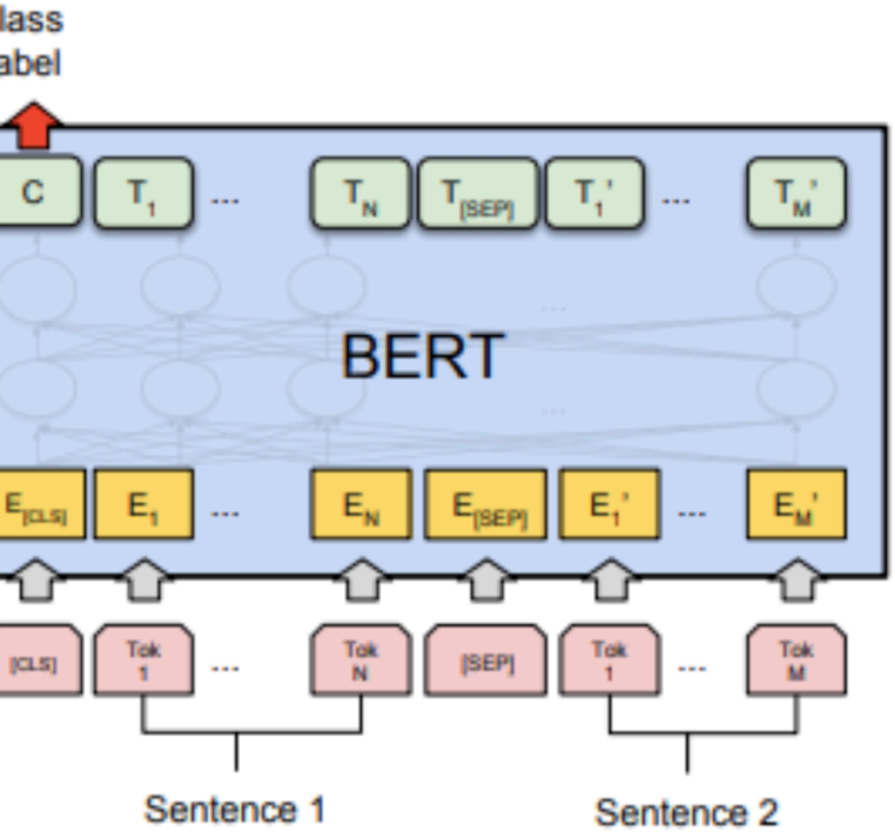
If you have more labels (for ex spam", "not spam", "social", and "promotion"), you just tweak the classifier network to have more output neurons and then pass through softmax.



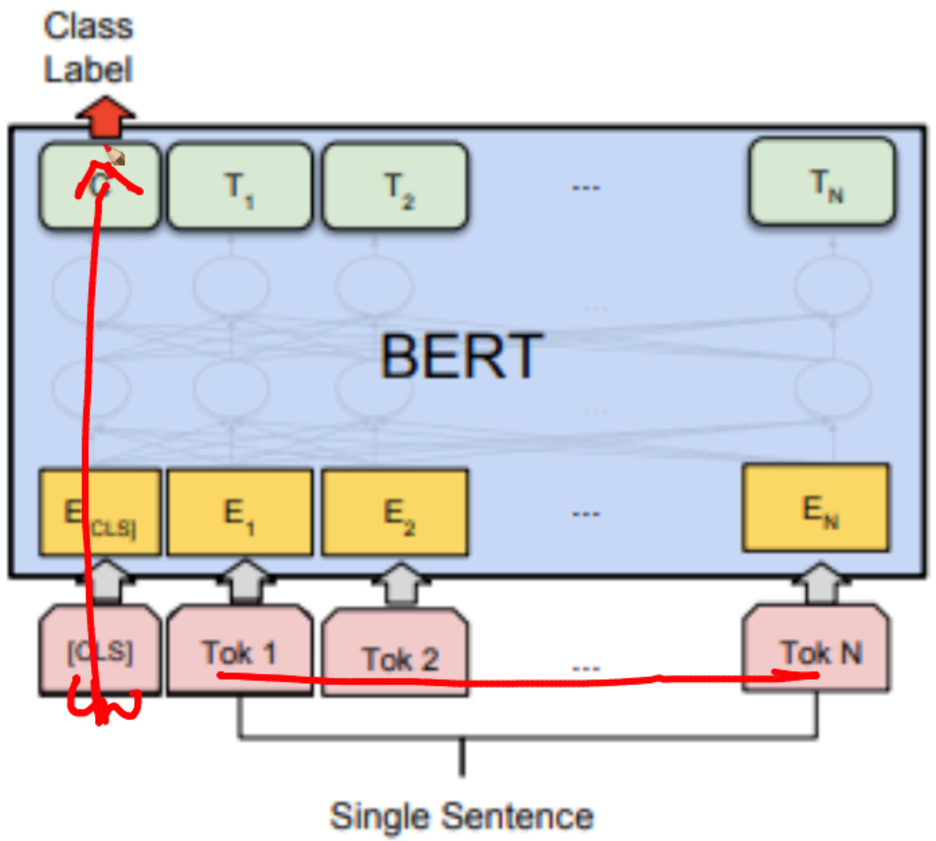
(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG



(b) Single Sentence Classification Tas
SST-2, CoLA

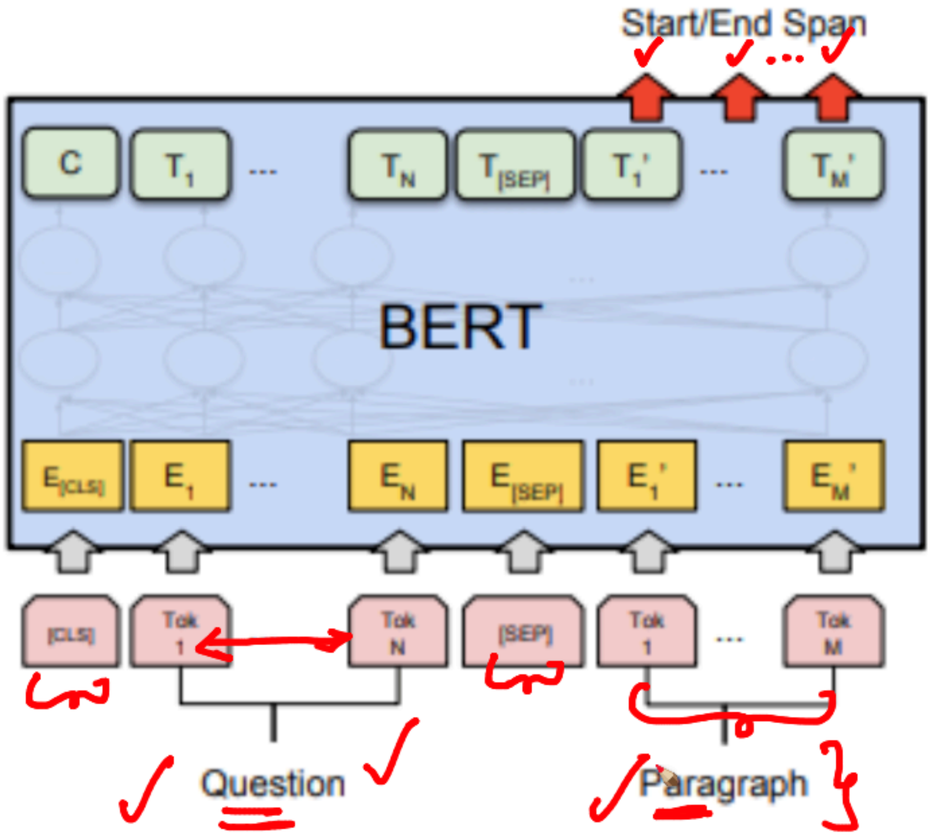


(a) Sentence Pair Classification Tasks:
MNLI, QQP, QNLI, STS-B, MRPC,
RTE, SWAG

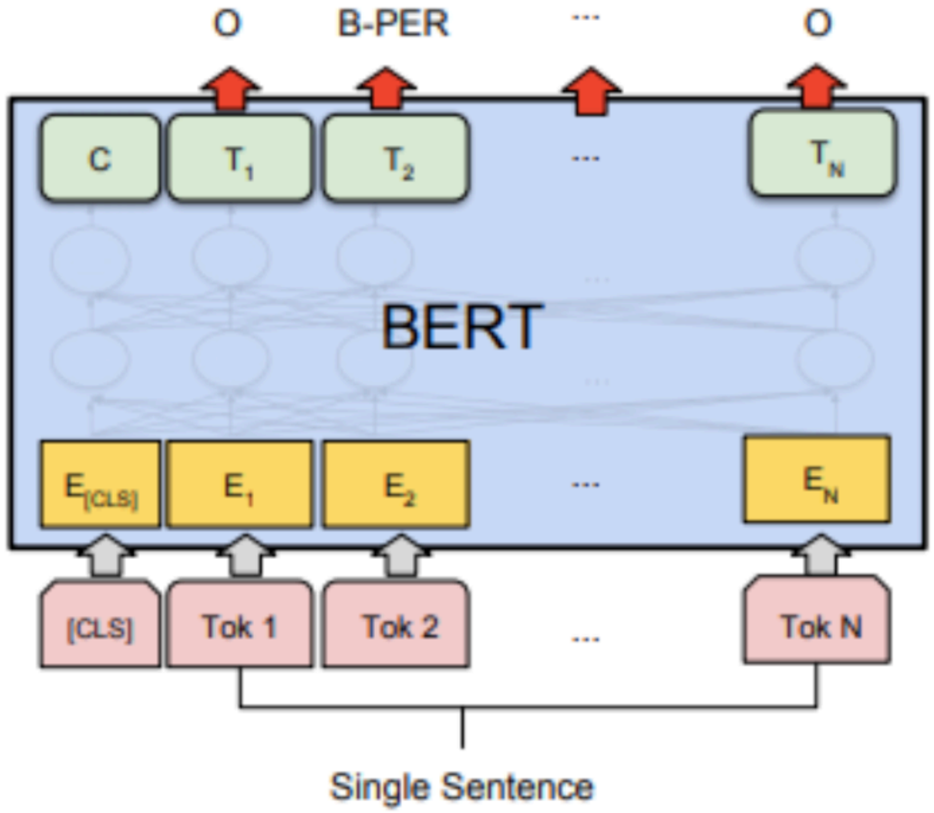


(b) Single Sentence Classification Tasks:
SST-2, CoLA

(a) Sentence Pair Classification Tasks:
MNLI, QQP, QNLI, STS-B, MRPC,
RTE, SWAG



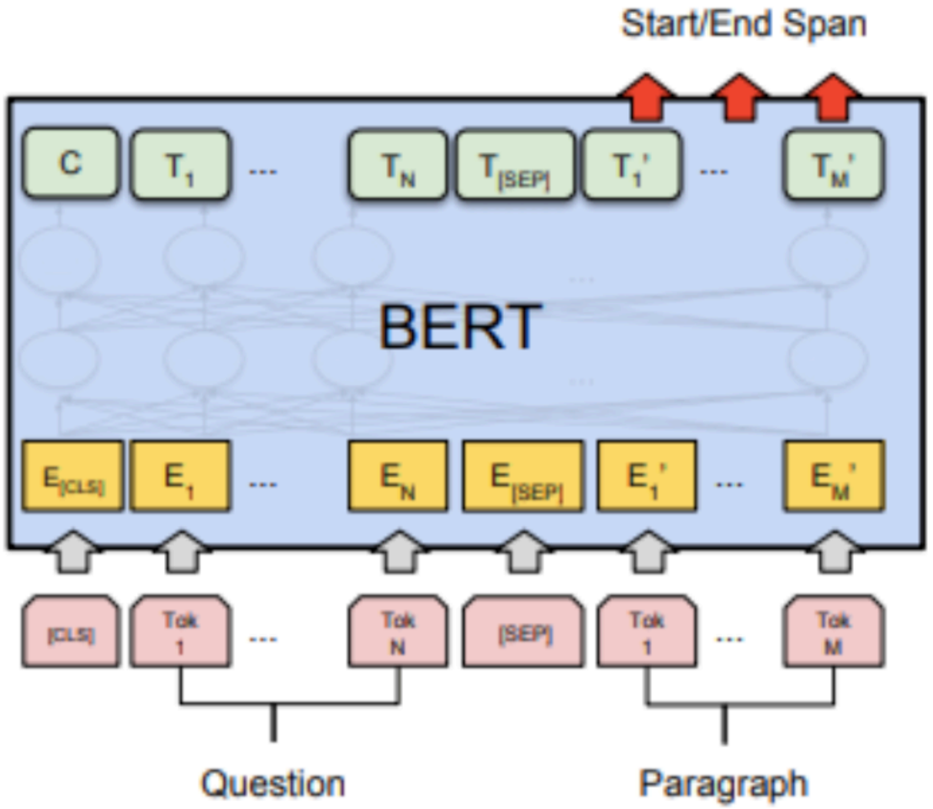
(b) Single Sentence Classification Tasks:
SST-2, CoLA



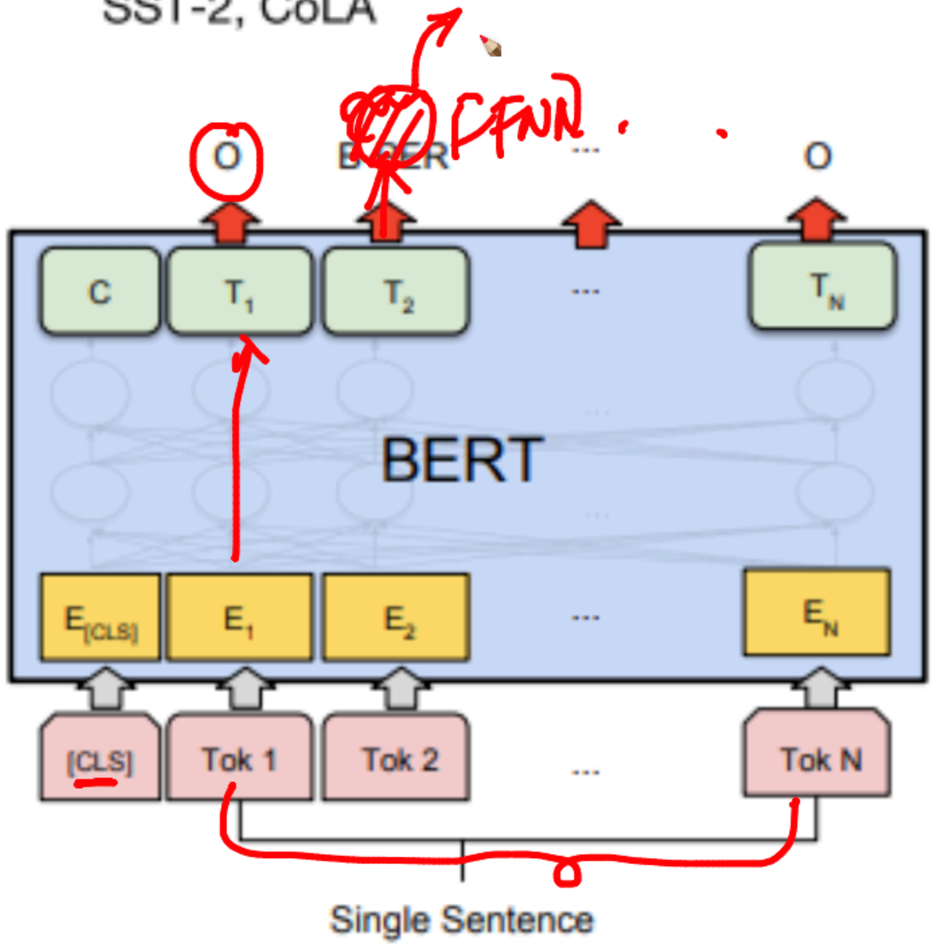
(c) Question Answering Tasks:
SQuAD v1.1

(d) Single Sentence Tagging Tasks:
CoNLL-2003 NER

(a) Sentence Pair Classification Tasks:
MNLI, QQP, QNLI, STS-B, MRPC,
RTE, SWAG

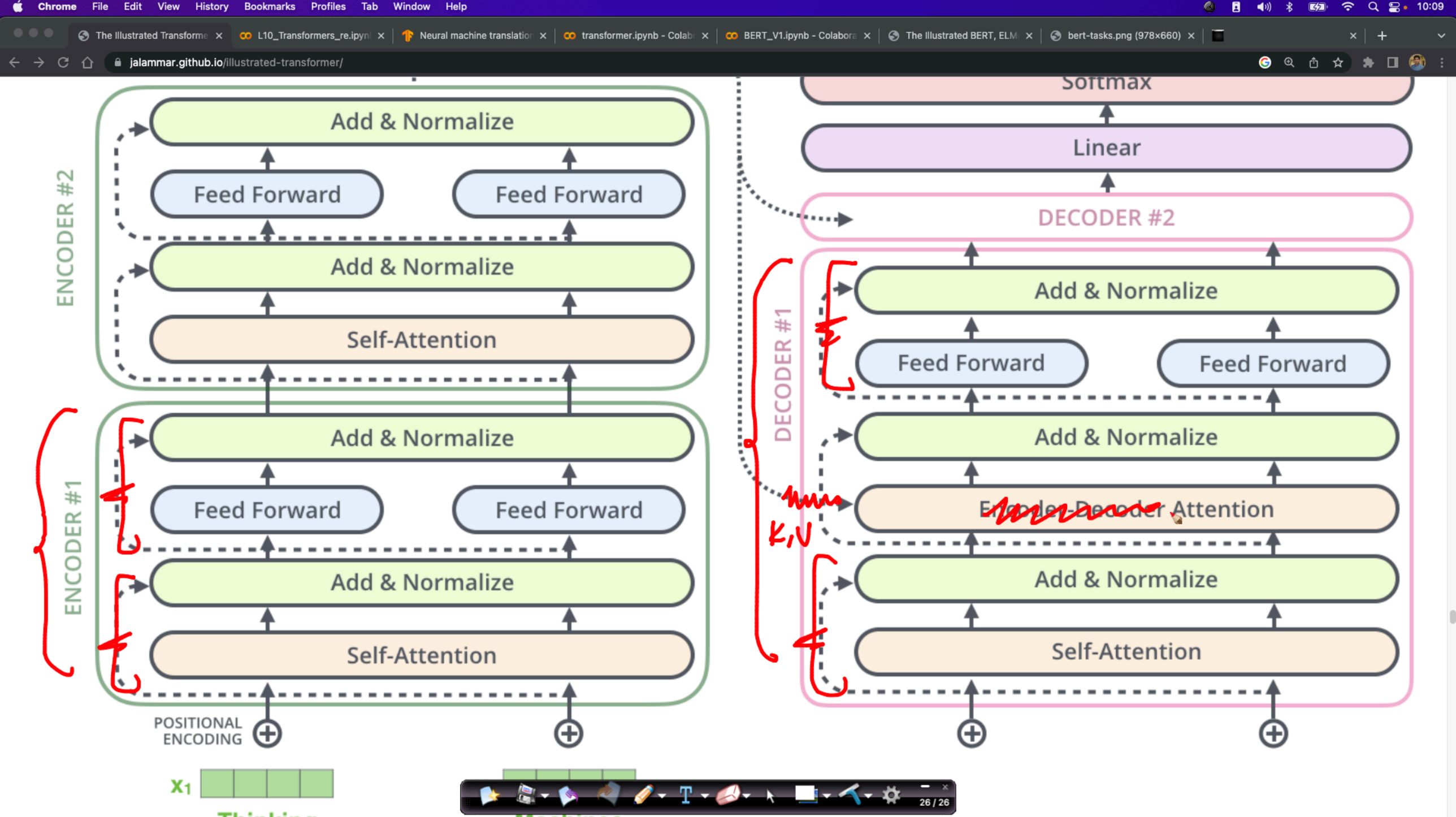


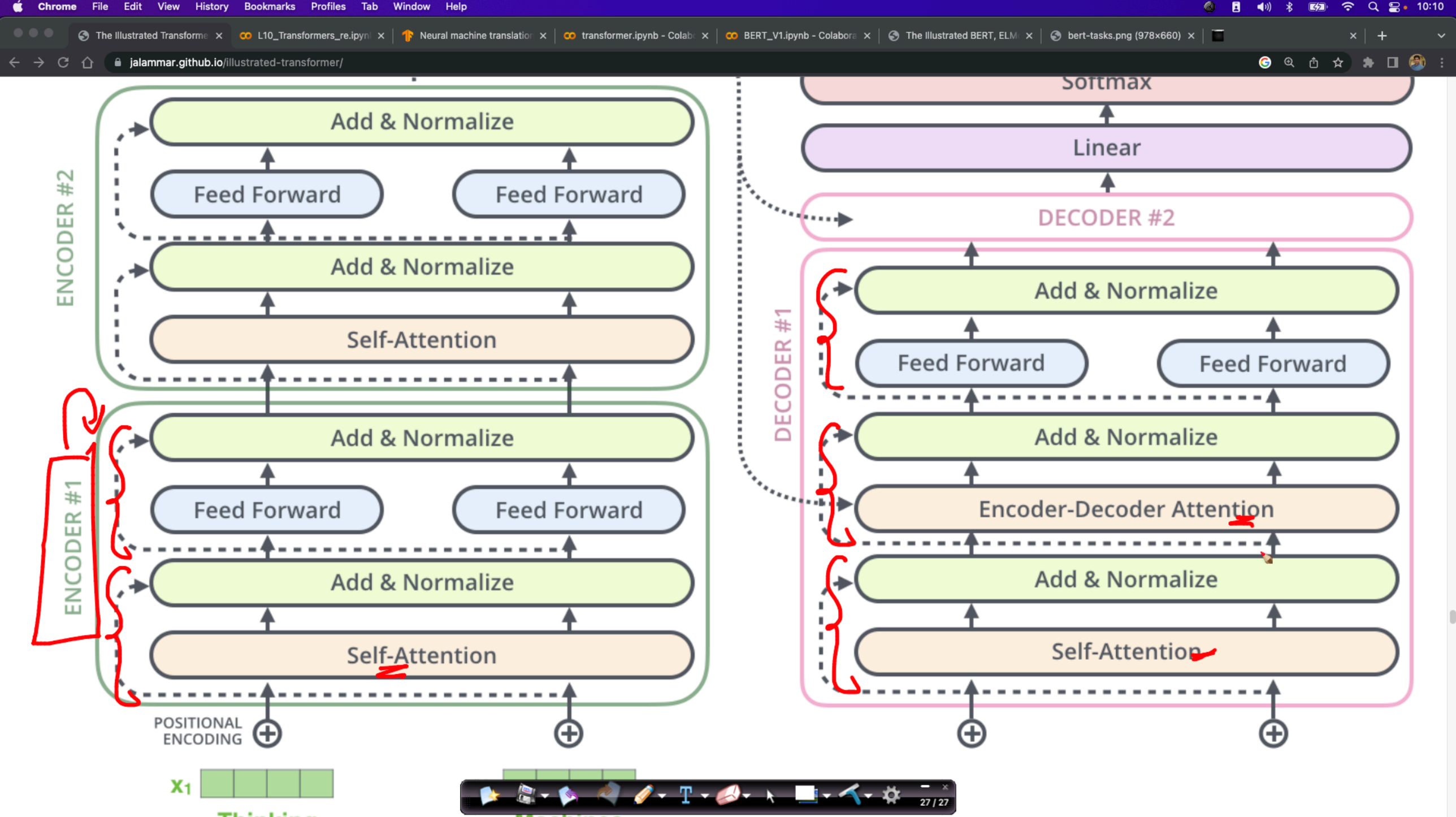
(b) Single Sentence Classification Tasks:
SST-2, CoLA



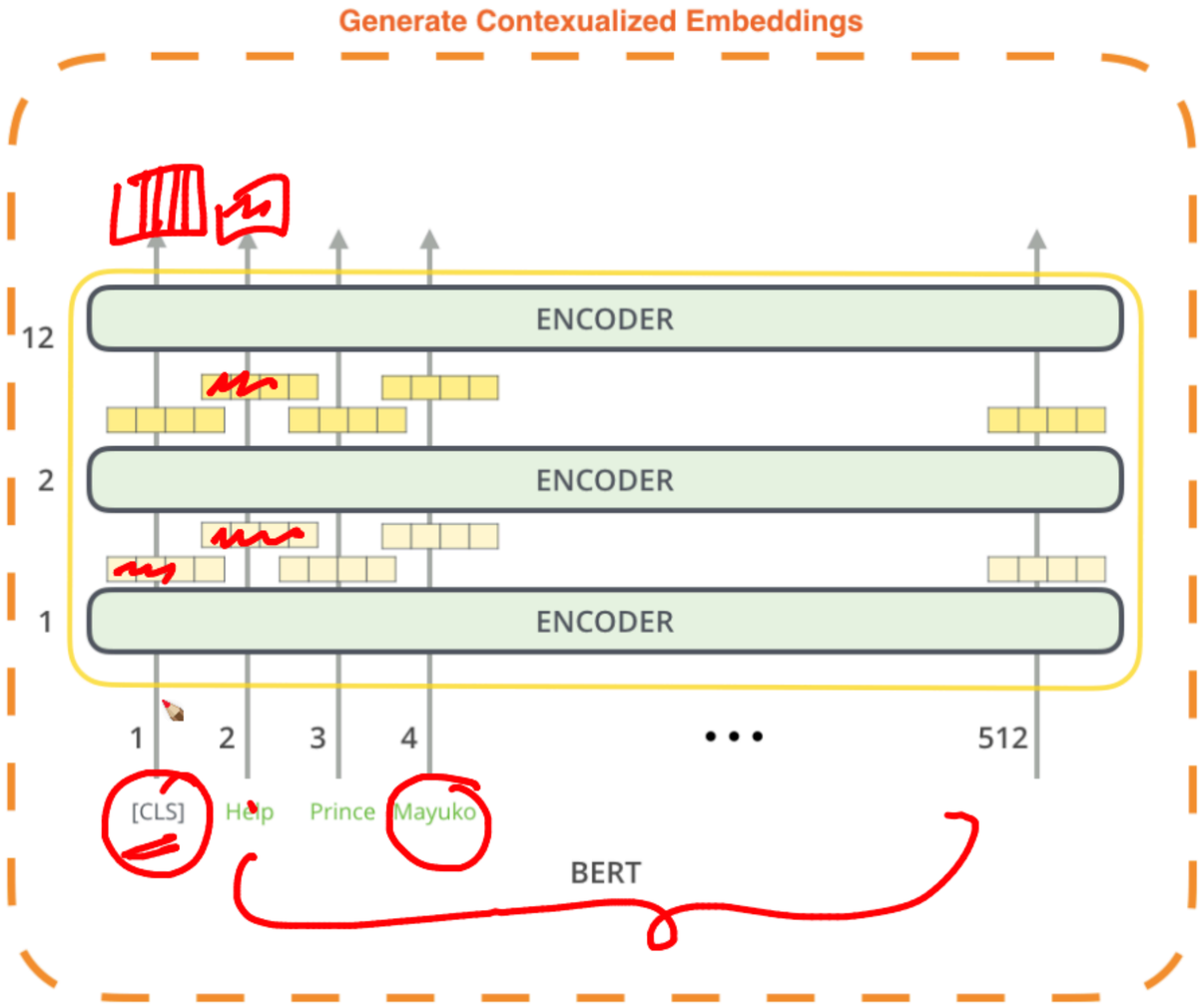
(c) Question Answering Tasks:
SQuAD v1.1

(d) Single Sentence Tagging Tasks:
CoNLL-2003 NER

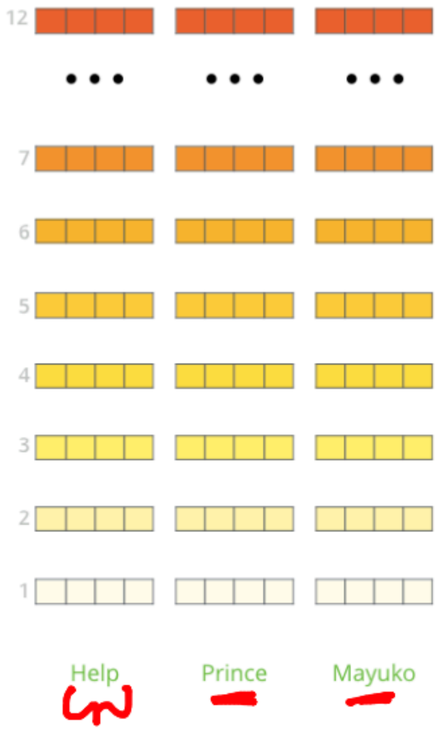

Softmax
Linear
DECODER #2
DECODER #1
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Encoder-Decoder Attention
K,V
Add & Normalize
Self-Attention
ENCODER #2
ENCODER #1
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
POSITIONAL ENCODING
x1
Thinking


Softmax
Linear
DECODER #2
DECODER #1
Add & Normalize
Feed Forward
Encoder-Decoder Attention
Self-Attention
ENCODER #2
ENCODER #1
POSITIONAL ENCODING
x1

shows yield results not far behind fine-tuning BERT on a task such as named-entity recognition.



The output of each encoder layer along each token's path can be used as a feature representing that token.

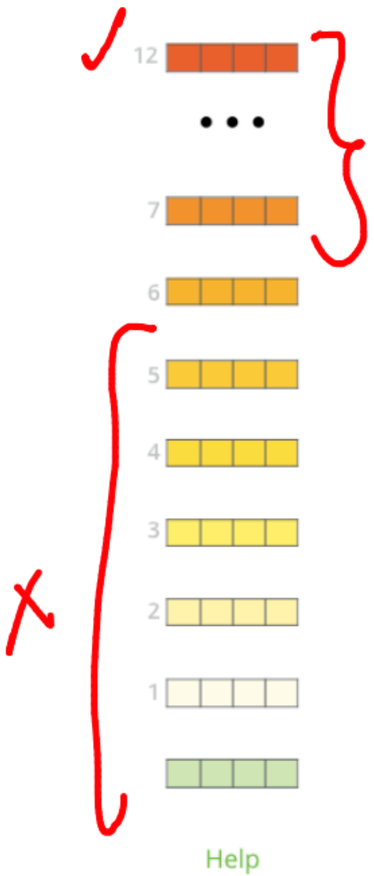


But which one should we use?

Which vector works best as a contextualized embedding? I would think it depends on the task. The paper examines six choices (Compared to the

What is the best contextualized embedding for "Help" in that context?

For named-entity recognition task CoNLL-2003 NER



		Dev F1 Score
First Layer	Embedding	91.0
Last Hidden Layer	12	94.9
Sum All 12 Layers	12 + ... + 2 + 1 =	95.5
Second-to-Last Hidden Layer	11	95.6
Sum Last Four Hidden	12 + 11 + 10 + 9 =	95.9
Concat Last Four Hidden	9 10 11 12	96.1

Take BERT out for a s

(Q)

{Train 2
10 k sentences → Classfn: 3-class clas

BERT ~ few million params
(512)

(?)
finetune a pretrained model

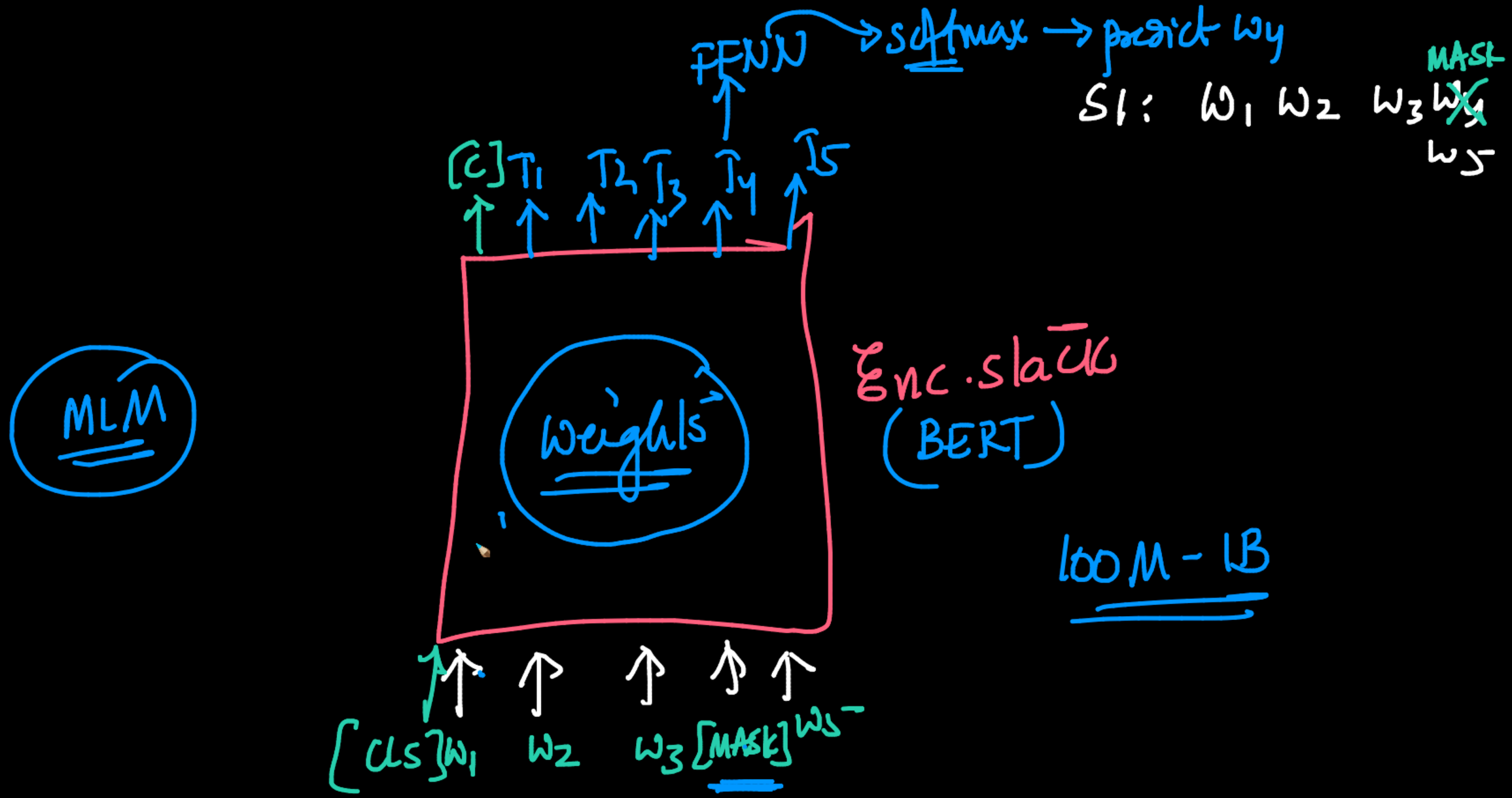
CNNs — ?
↓
ImageNet
label
10-14 MM

English- sentences → 100M ✓

w_1 w_2 w_4 w_6 w_8

BERT → Masked language models

↳ self- supervised learning



(Q)

Bert → pretrained (MLM) + finetune

@ runtime

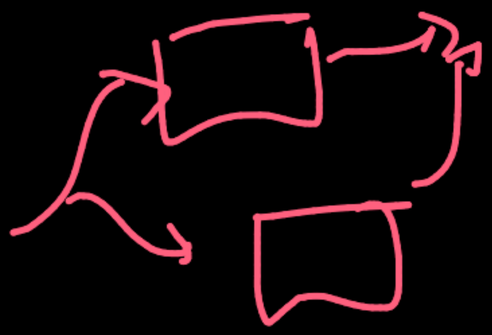
Low latency

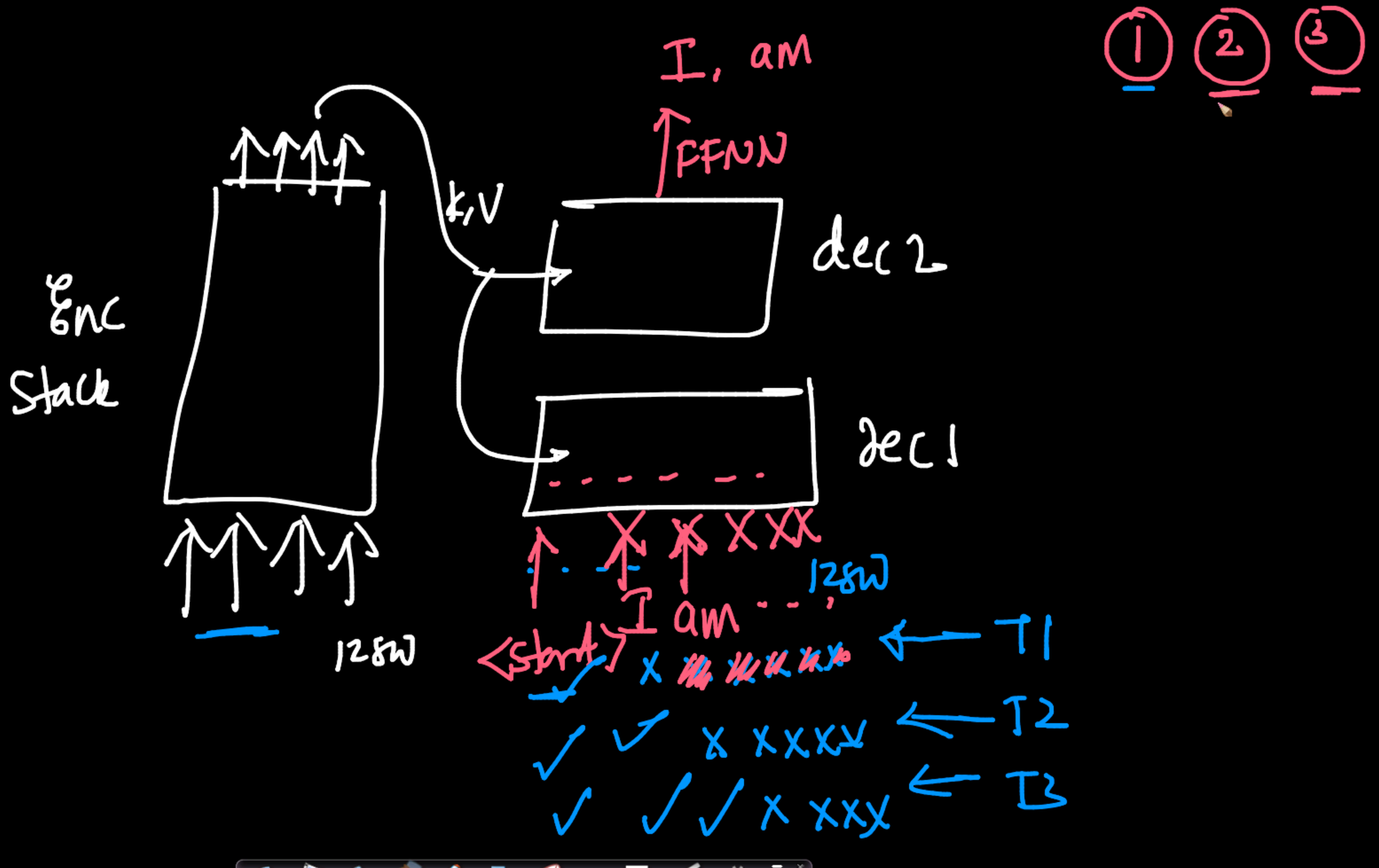
BERT

Distil BERT → ✓ which performs well

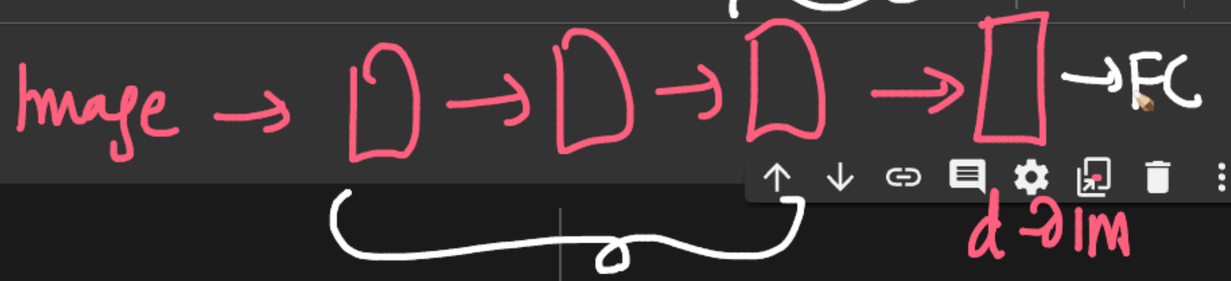
(Knowledge Distillation)

HINT: large ↔ small





- learning rate
- number of epochs

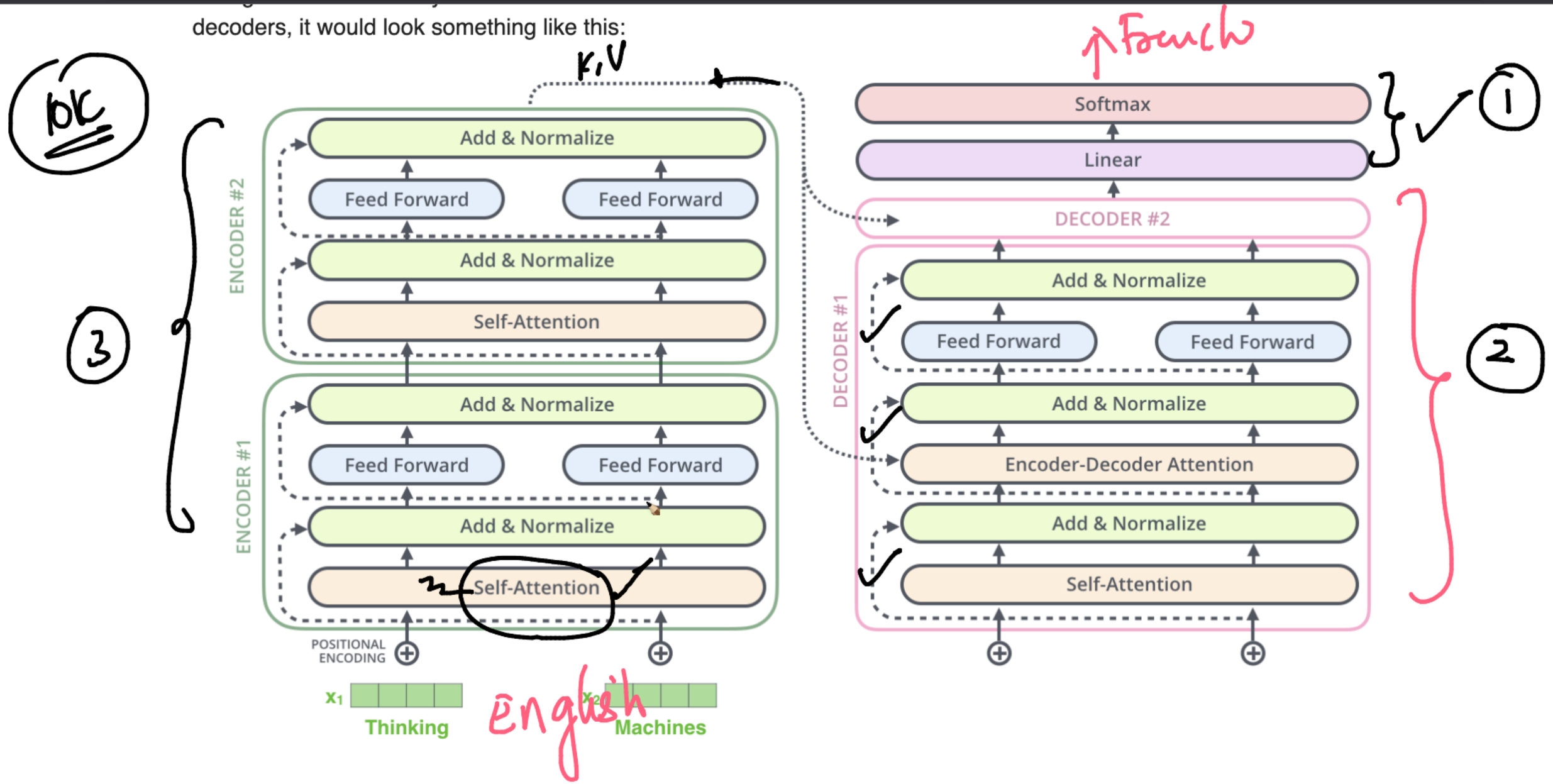



```
from transformers import create_optimizer
import tensorflow as tf

# The number of training steps is the number of samples in the dataset, divided by the batch size then multiplied
# by the total number of epochs. Note that the tf_train_dataset here is a batched tf.data.Dataset,
# not the original Hugging Face Dataset, so its len() is already num_samples // batch_size.
num_epochs = 3
num_train_steps = len(tf_train_dataset) * num_epochs

optimizer, schedule = create_optimizer(
    init_lr=5e-5,
    num_warmup_steps=0,
    num_train_steps=num_train_steps,
    weight_decay_rate=0.01,
)
model.compile(optimizer=optimizer)
```

```
No loss specified in compile() - the model's internal loss computation will be used as the loss. Don't panic - this
```

```
[ ]
```

decoders, it would look something like this:



The Decoder Side

+ Code + Text Connect

- labels

```
from transformers import DataCollatorForSeq2Seq
from transformers import TFAutoModelForSeq2SeqLM, Seq2SeqTrainingArguments, Seq2SeqTrainer

model = TFAutoModelForSeq2SeqLM.from_pretrained(model_checkpoint, from_pt=True)

data_collator = DataCollatorForSeq2Seq(tokenizer=tokenizer, model=model, return_tensors="tf")
```

```
All PyTorch model weights were used when initializing TFMarianMTModel.

All the weights of TFMarianMTModel were initialized from the PyTorch model.
If your task is similar to the task the model of the checkpoint was trained on, you can already use TFMarianMTModel
```

```
[ ] # Lets see content of data_collator

batch = data_collator([tokenized_datasets["train"][i] for i in range(1, 3)])
batch.keys()
```

```
dict_keys(['input_ids', 'attention_mask', 'labels', 'decoder_input_ids'])
```

```
# Lsts see how decoder input is mapped to id
batch["decoder_input_ids"]
```