The Transformer Architecture

CS 780/880 Natural Language Processing Lecture 18

Samuel Carton, University of New Hampshire

Last lecture

Sequence-to-sequence models

- Main application: translation

Attention

- Improves performance of sequence-to-sequence models
- Improves interpretability of classifiers

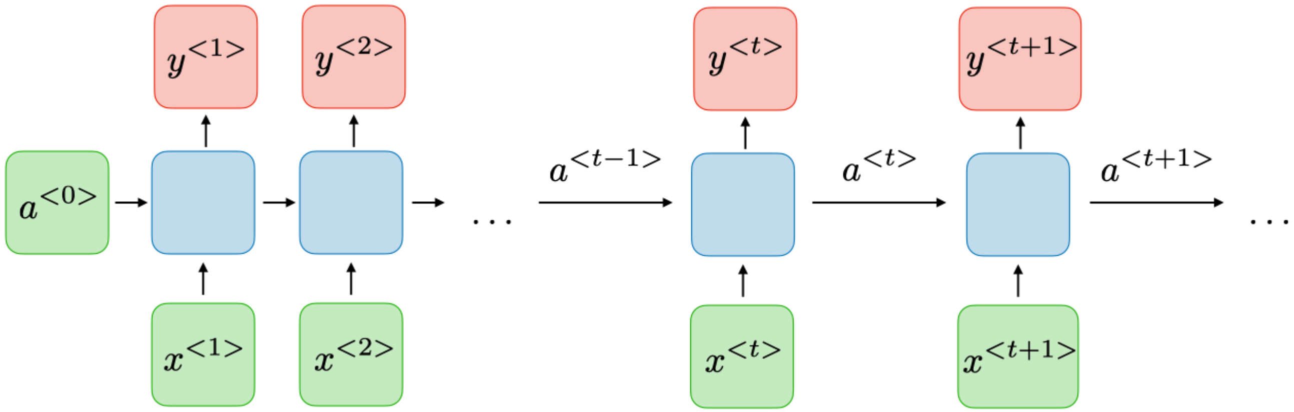
Model saving/loading

The main problem with RNNs

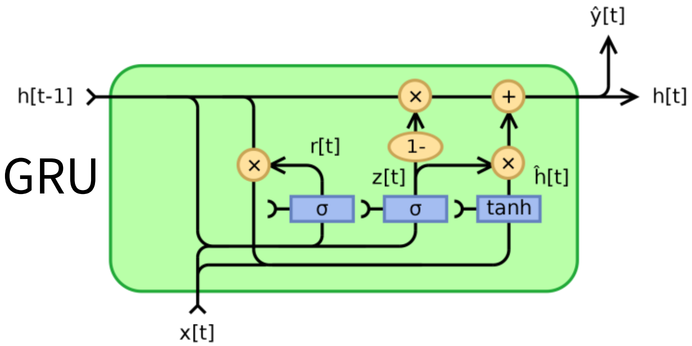
Because of **vanishing gradients**, it is hard for RNNs to learn to remember information in early timesteps that is needed for later timesteps (i.e. **long-term dependencies**)

This leads to **catastrophic forgetting**

RNNs are also not very parallelizable

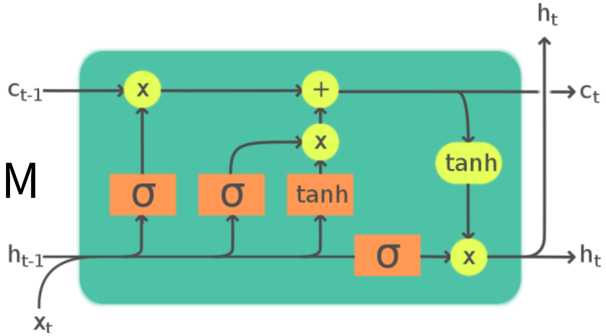


GRU



https://en.wikipedia.org/wiki/Gated_recurrent_unit

LSTM



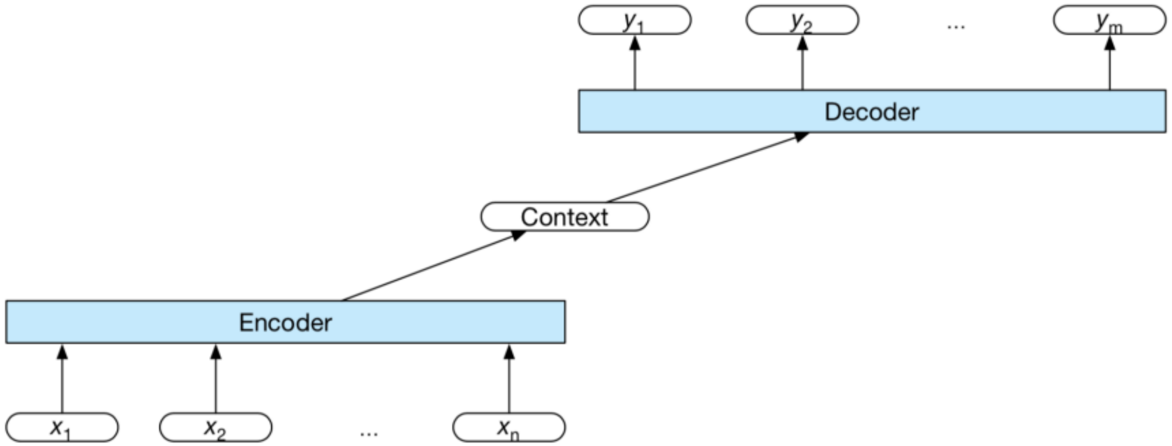
https://en.wikipedia.org/wiki/Long_short-term_memory

Solution: attention?

With sequence-to-sequence models, we discovered that **attention** is a valuable mechanism for augmenting the final context vector

So what if we just want to encode one sequence?

Neural machine translation by jointly learning to align and translate
D Bahdanau, K Cho, Y Bengio - arXiv preprint arXiv:1409.0473, 2014 - arxiv.org
Neural machine translation is a recently proposed approach to machine translation. Unlike the traditional statistical machine translation, the neural machine translation aims at building a single neural network that can be jointly tuned to maximize the translation performance. The models proposed recently for neural machine translation often belong to a family of encoder-decoders and consists of an encoder that encodes a source sentence into a fixed-length vector from which a decoder generates a translation. In this paper, we conjecture that …
Save Cite Cited by 28013 Related articles All 28 versions

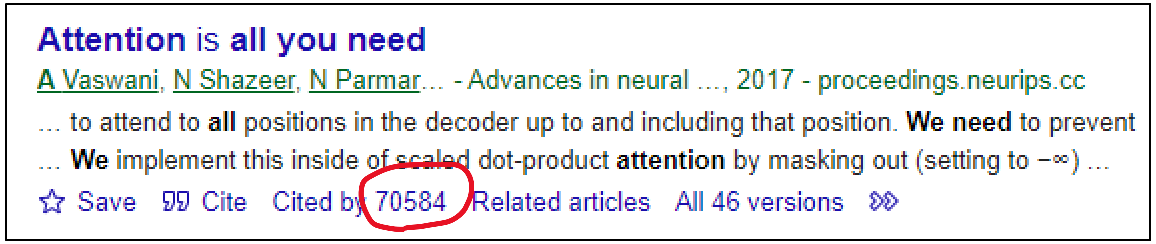


Transformers

The **Transformer** is a **non-recurrent** architecture that uses **self-attention** to represent relationships between words in the **same sequence**

- As opposed to between words in the input and output sequence
- Although, transformers are also be used in sequence-to-sequence models (and actually do both kinds of attention)

Invented in Attention is All You Need (Vaswani et al., 2017)



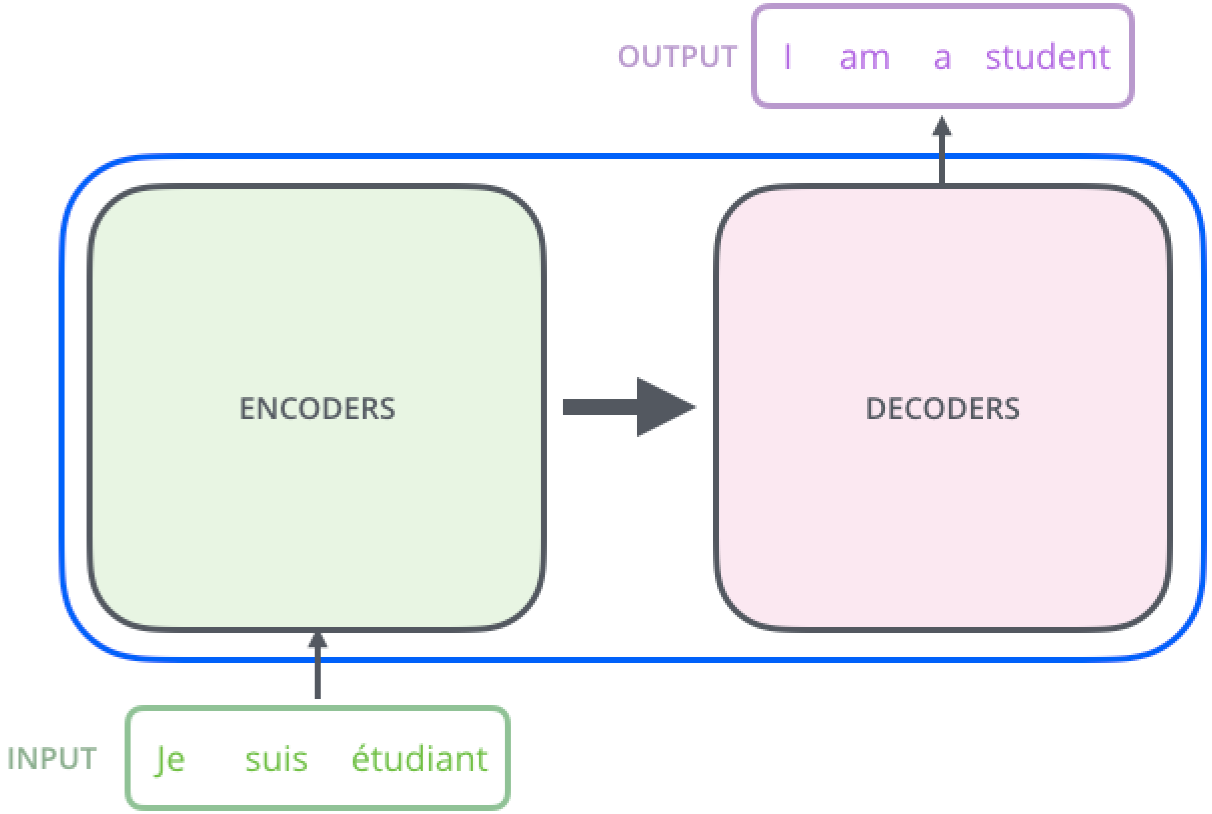
Transformers

Content for this lecture drawn largely from The Illustrated Transformer: https://jalammar.github.io/illustrated-transformer/

- Super duper popular breakdown
- Not a Medium article, but a good model of how influential these breakdowns can be (hint hint)

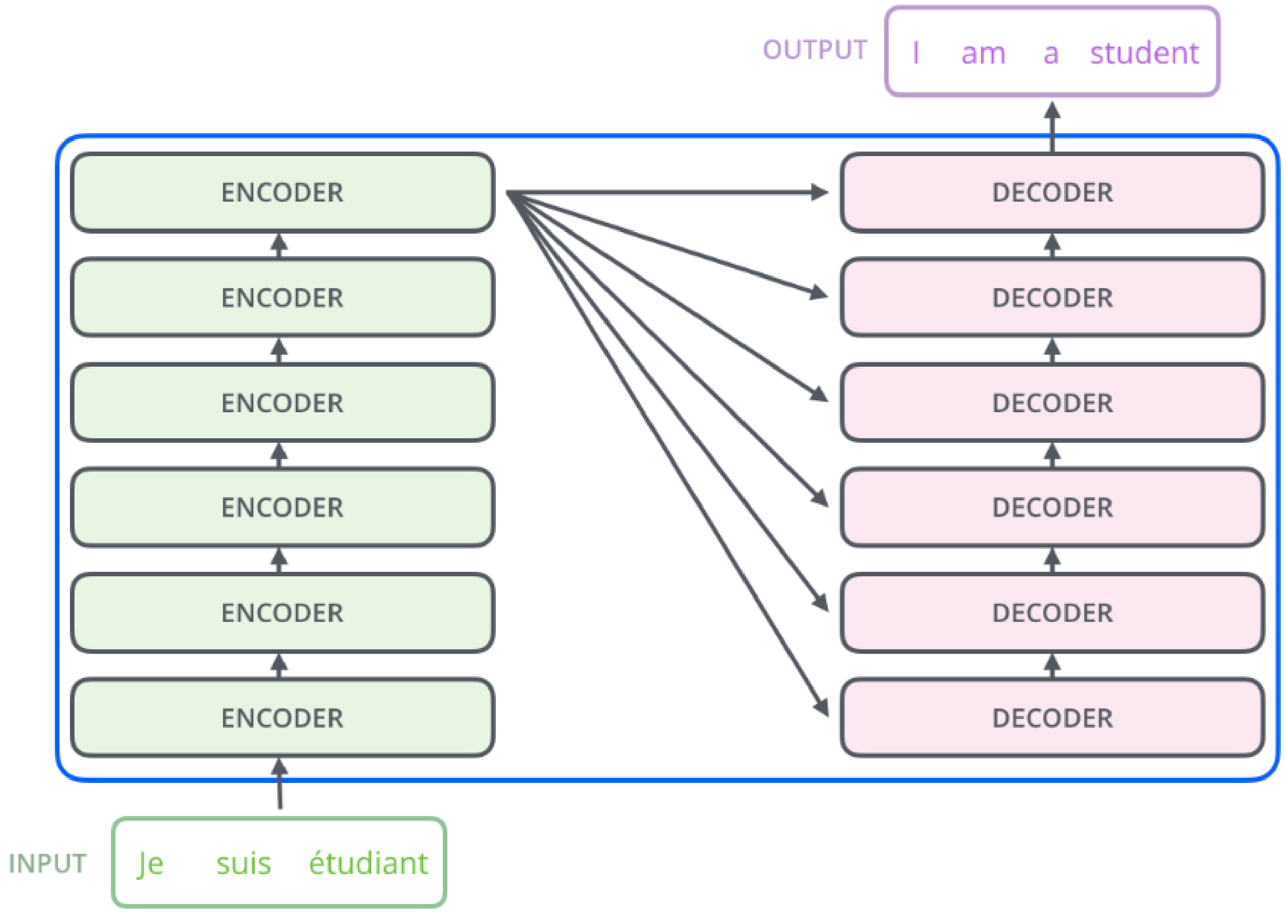
Encoder and decoder

The full transformer model includes an **encoder** (which encodes the text into a vector) and a **decoder** (which converts the encoded vector back into a text)



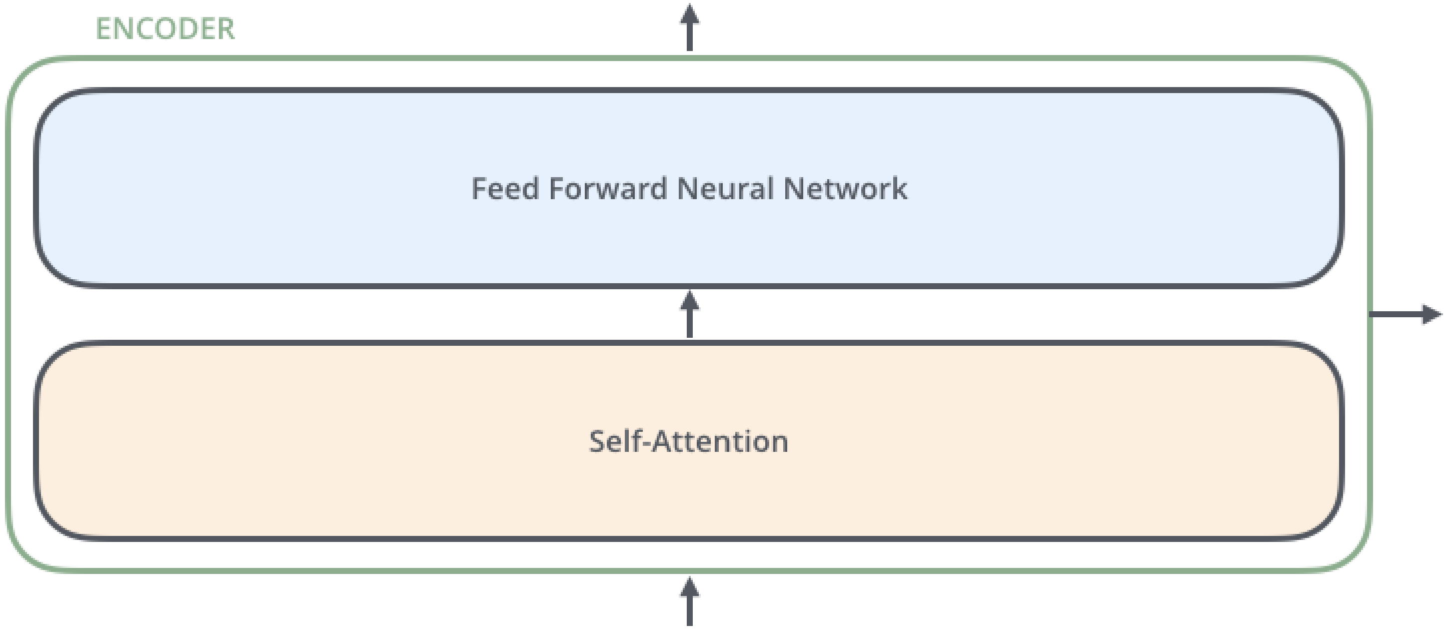
Encoder and decoder

Each component is actually a stack of repeated encoder or decoder layers



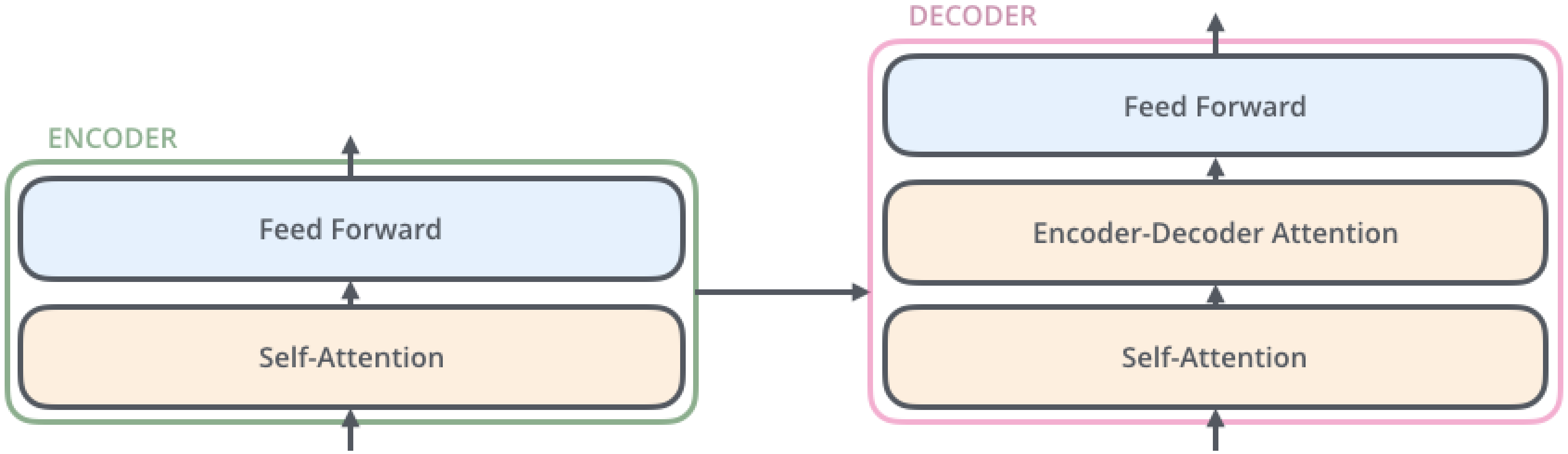
Encoder

The encoder consists of a “self-attention” layer, followed by a feedforward layer

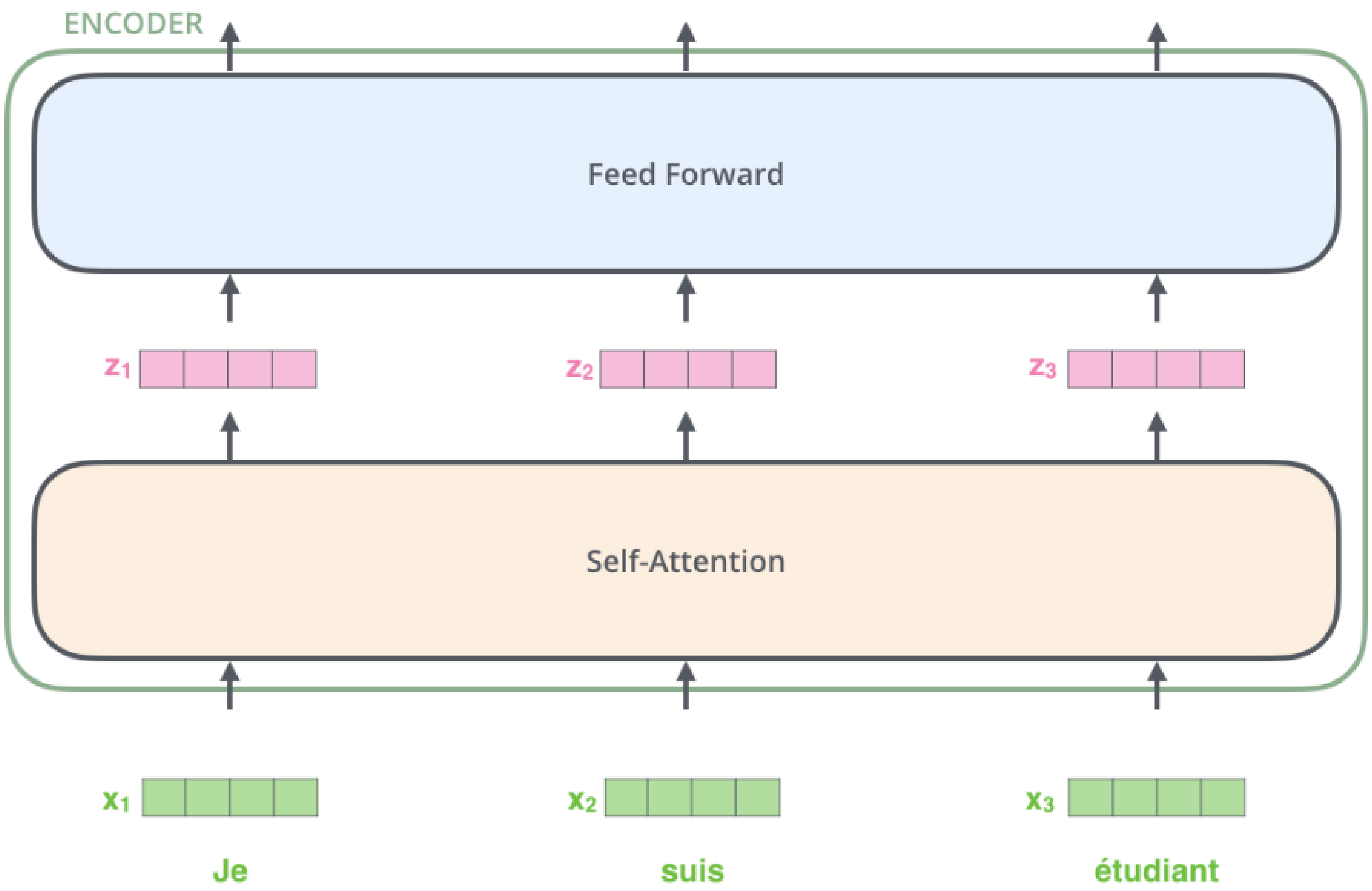


Decoder

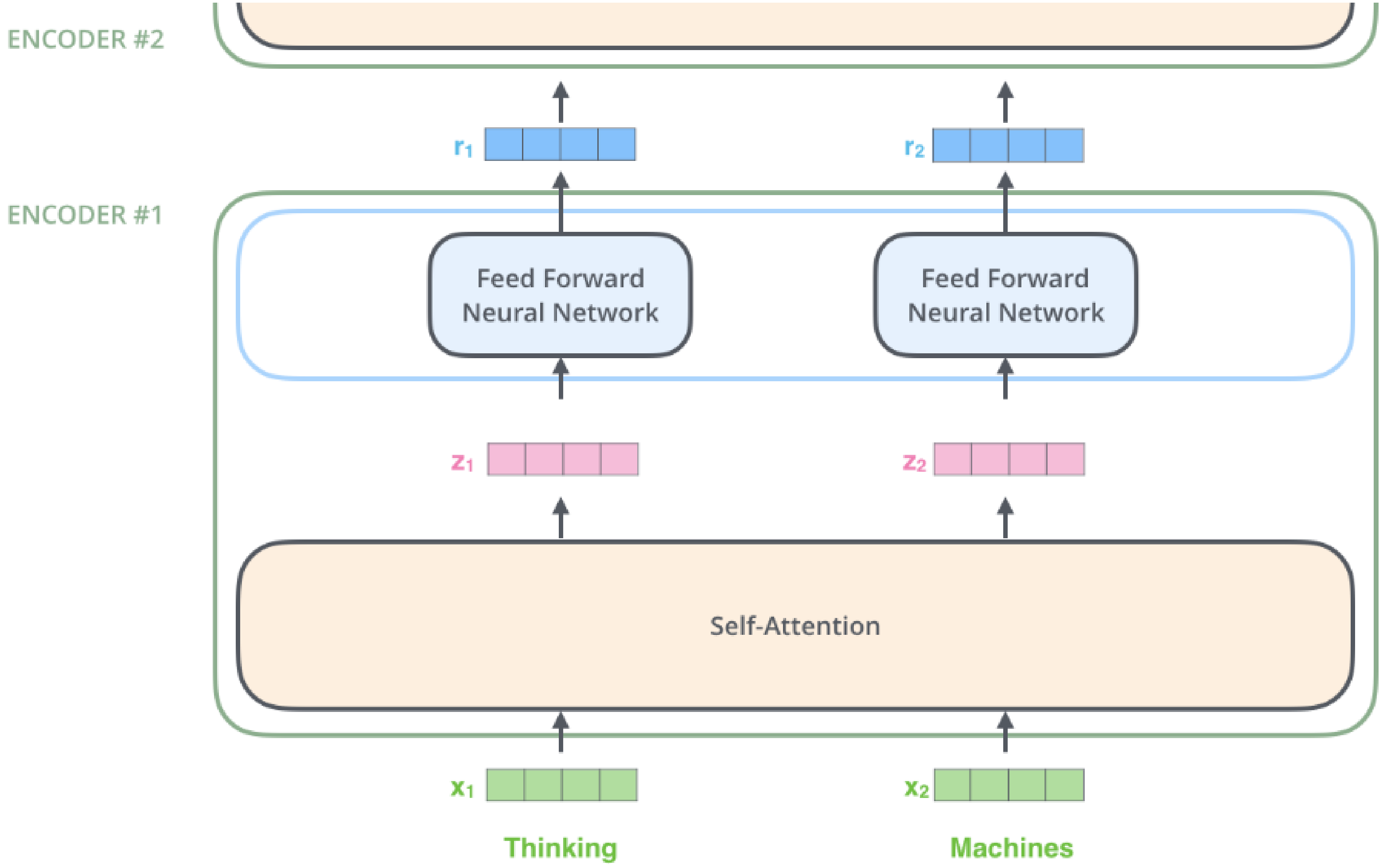
And then the decoder consists of both of these elements, plus an additional layer that learns to attend to the output from the encoder.



Encoder detail



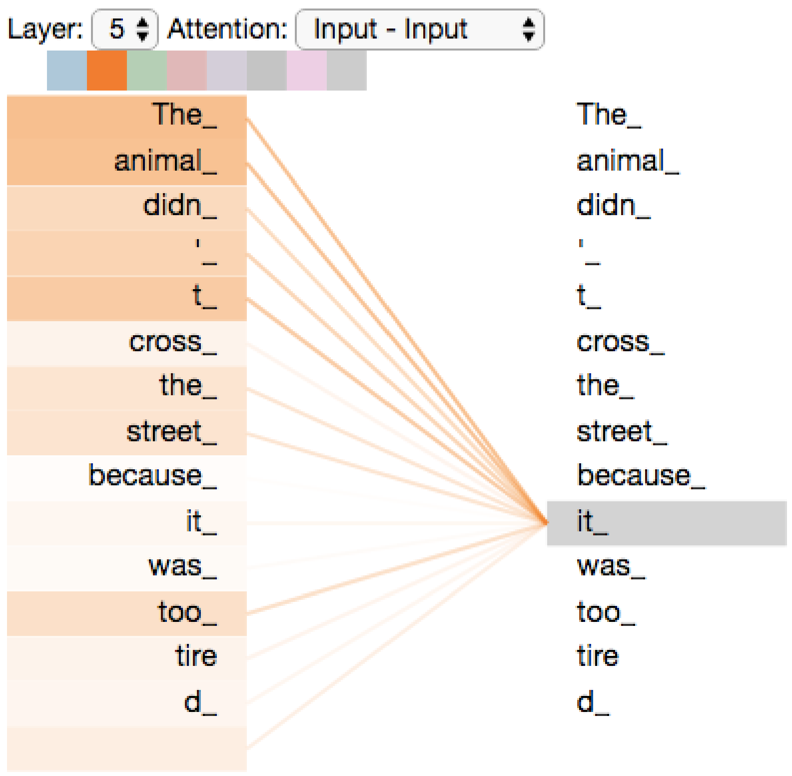
Encoder detail



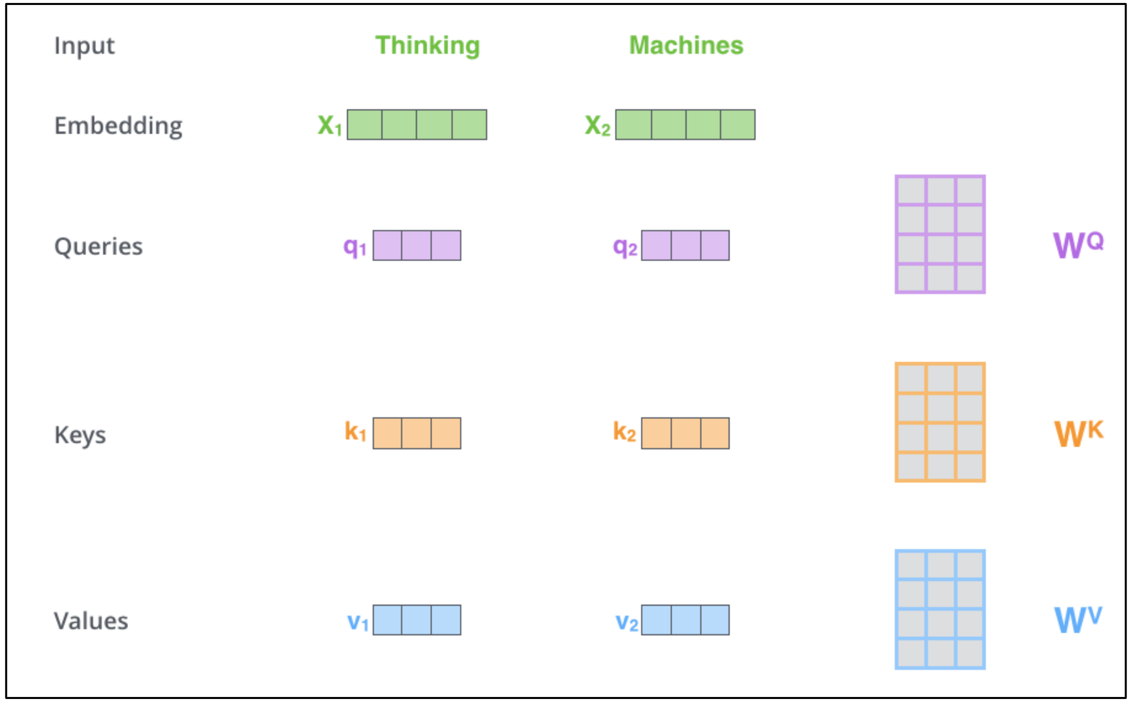
Self-attention

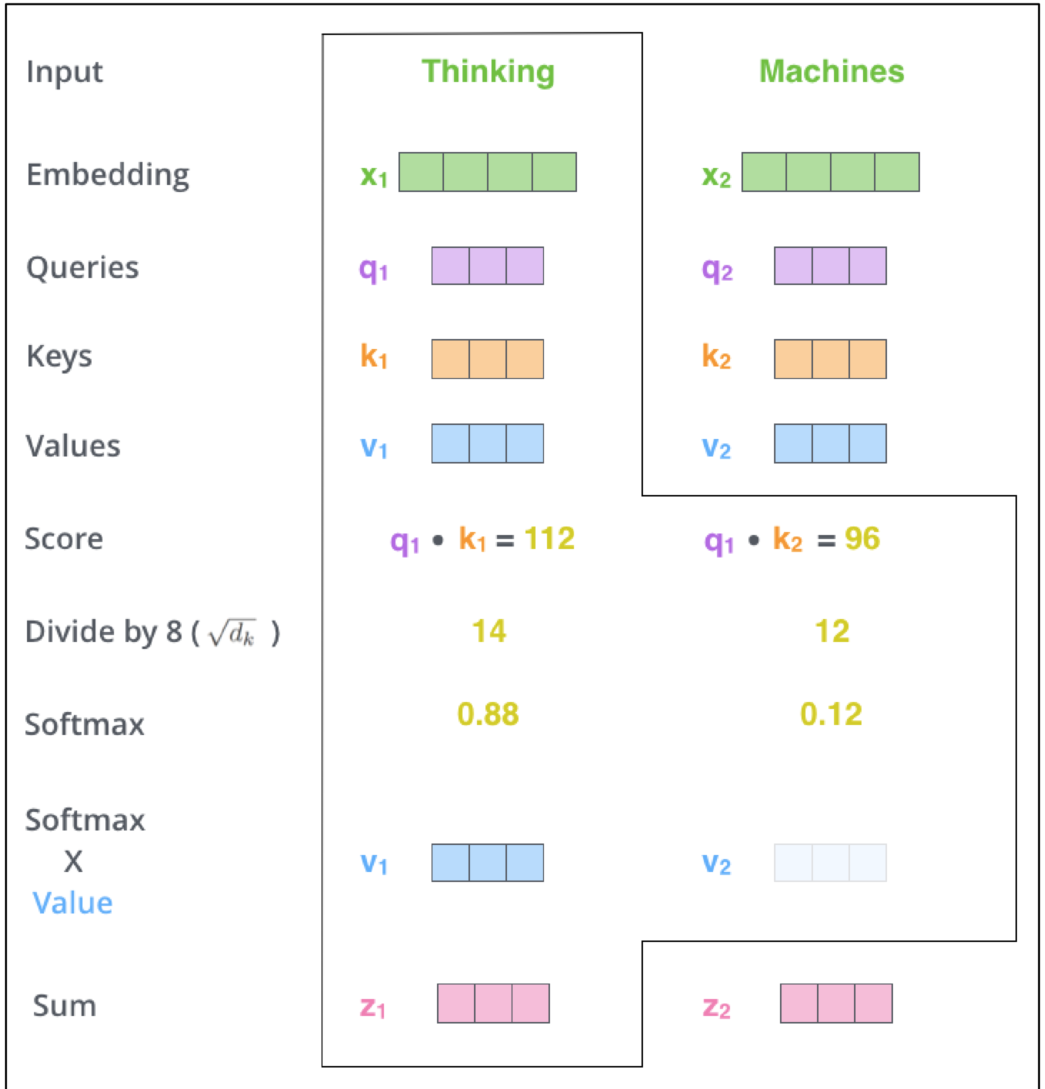
Basic idea: The model will learn an attention weight from each word w_i to each word w_j, representing how important w_j is for understanding the meaning of w_i

In this example, in order to understand "it", we really need to understand "the" and "animal", since that is what "it" is referring to

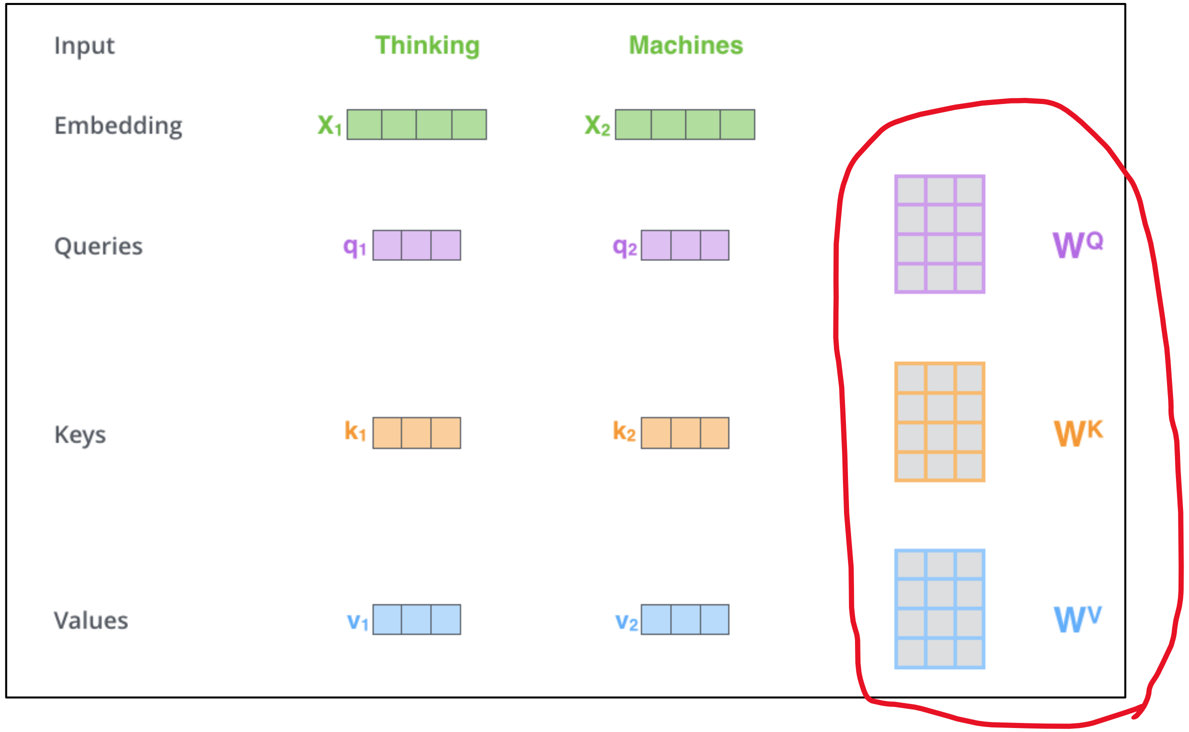


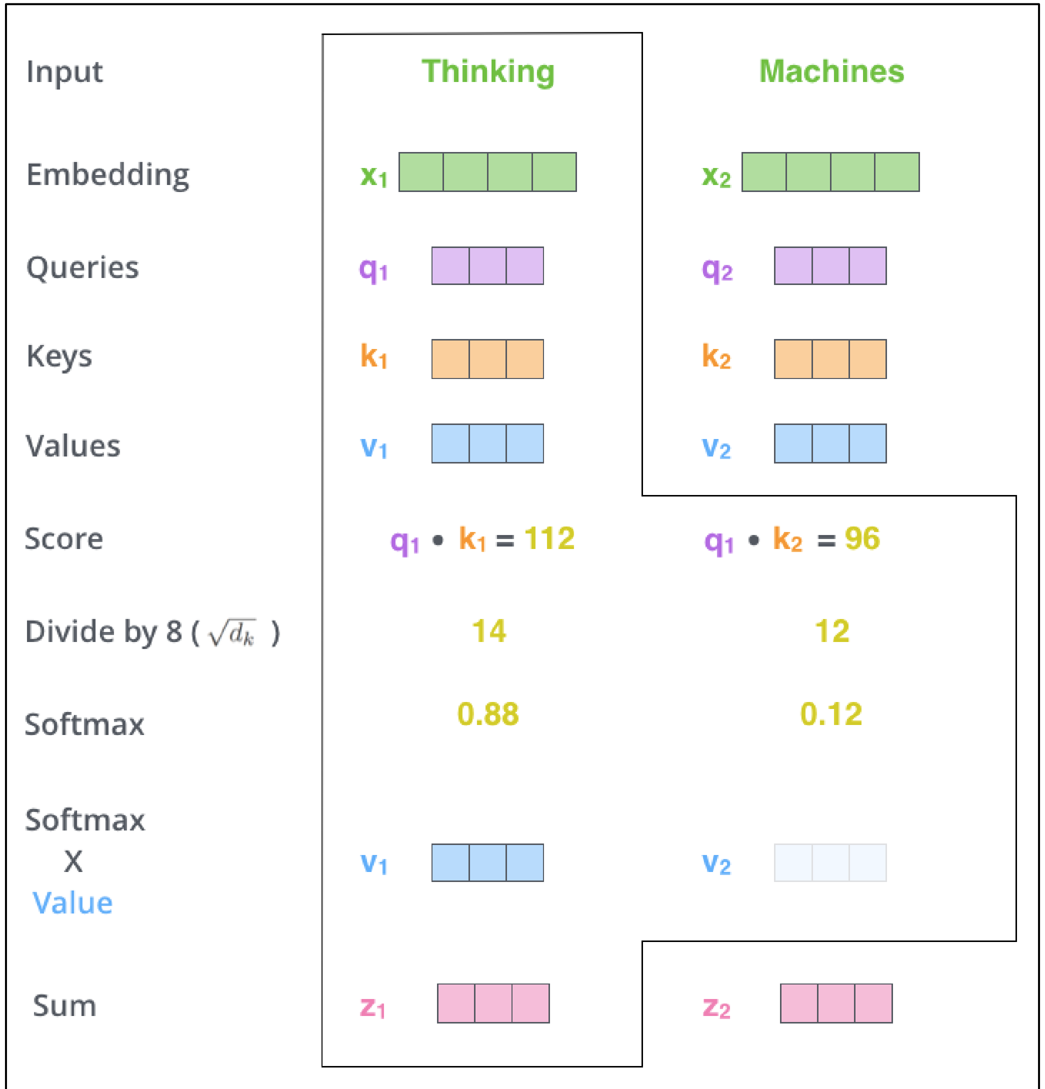
Self-attention detail



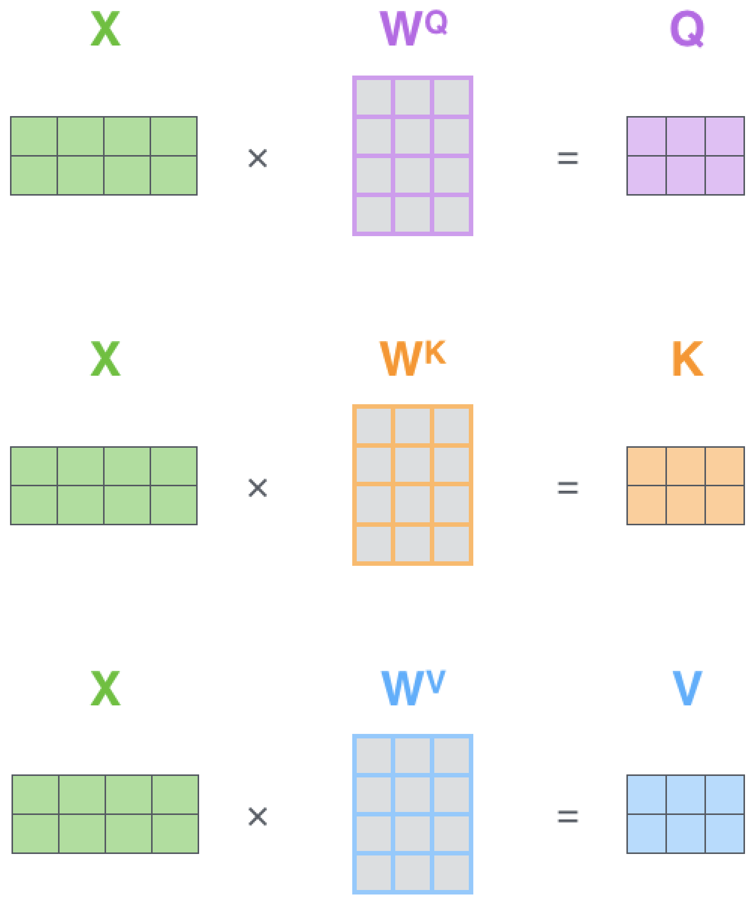


Self-attention detail

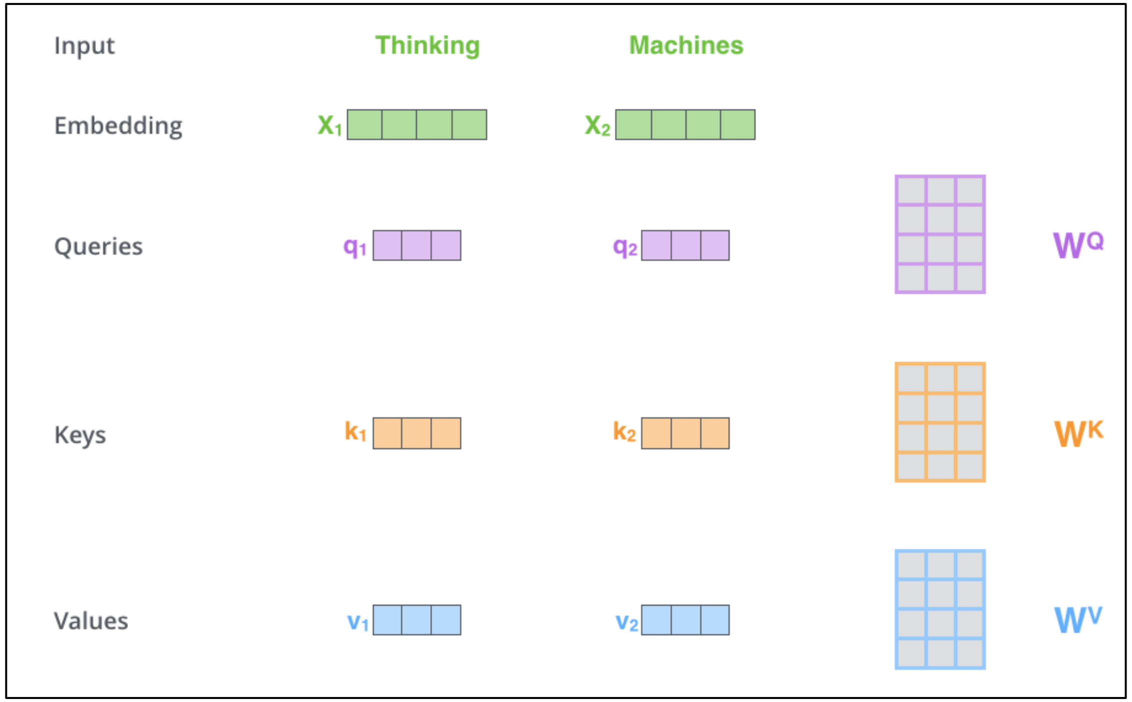


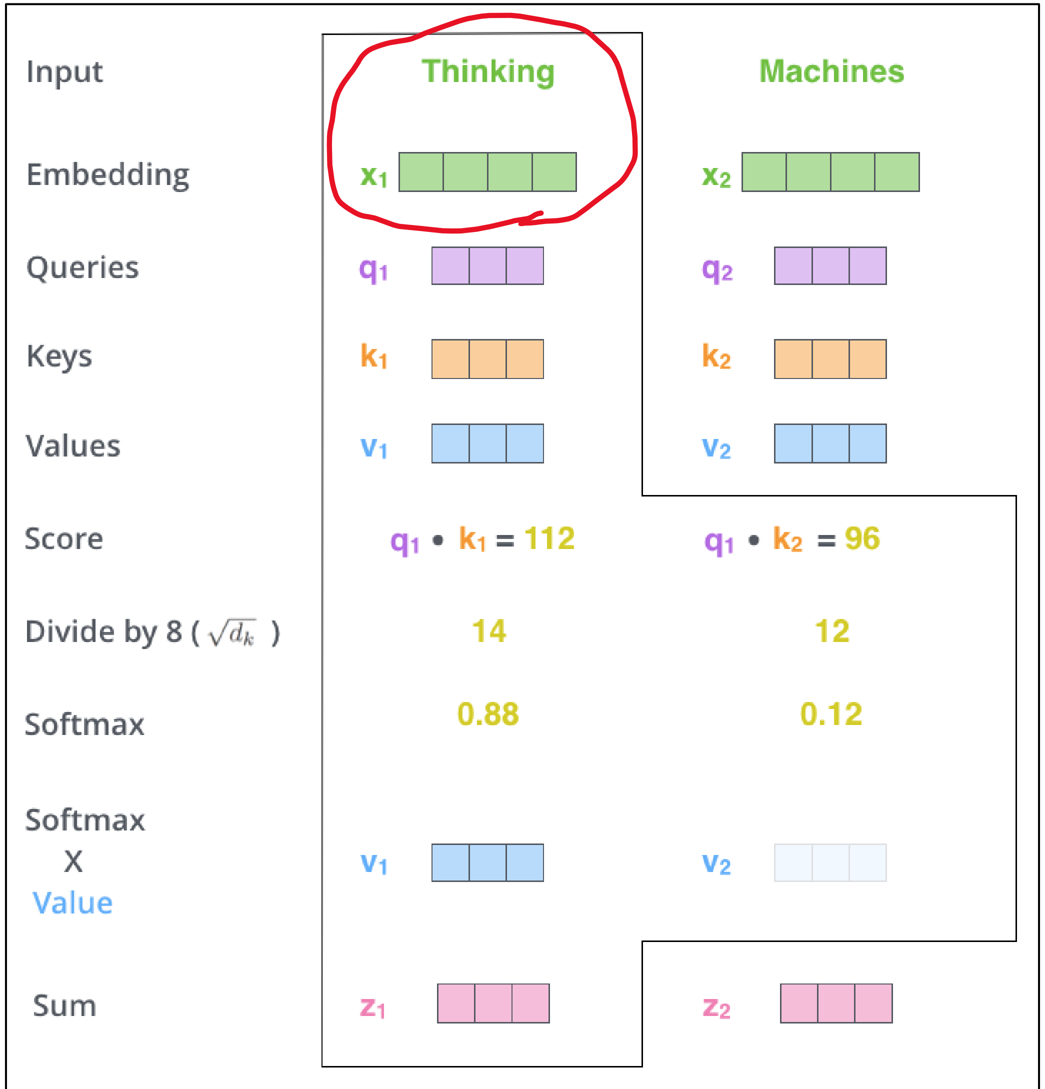


Self-attention detail

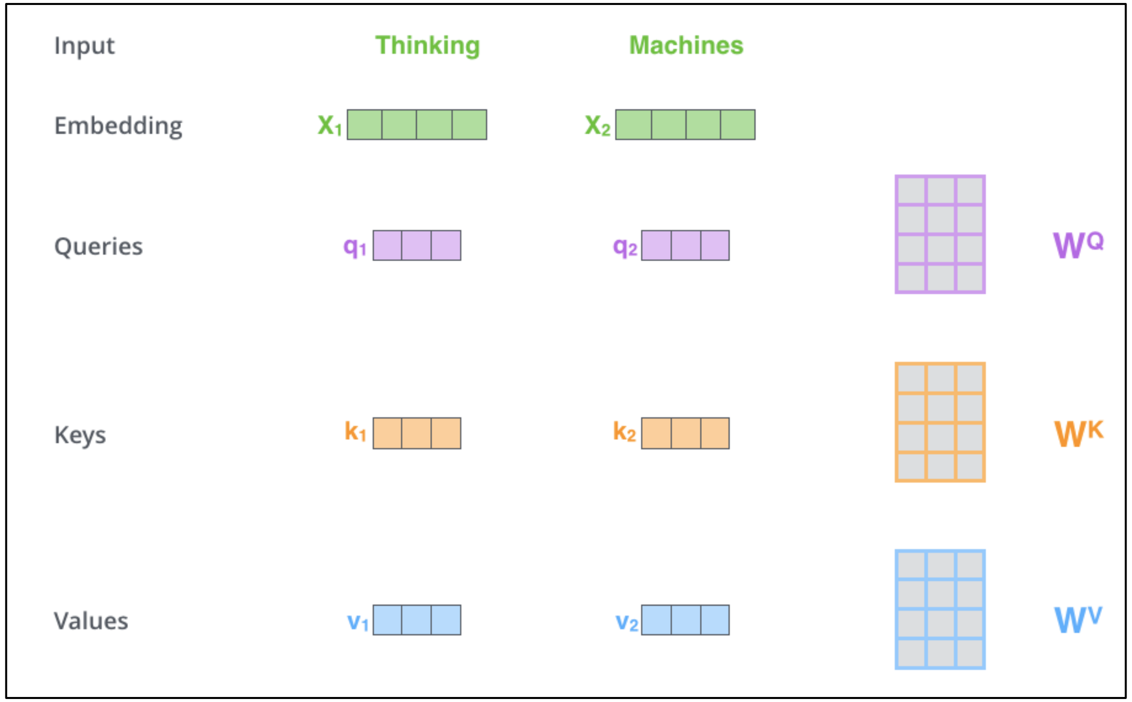


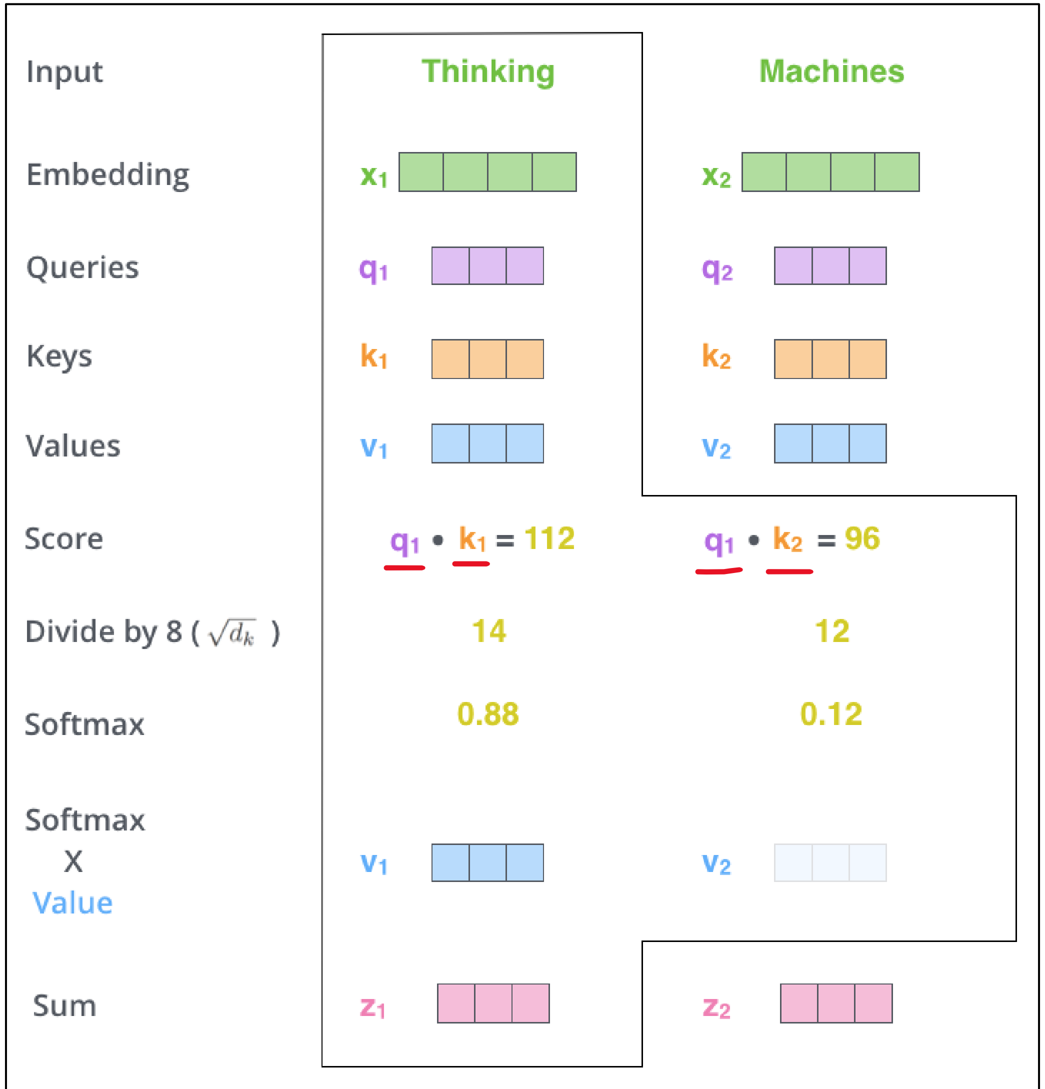
Self-attention detail



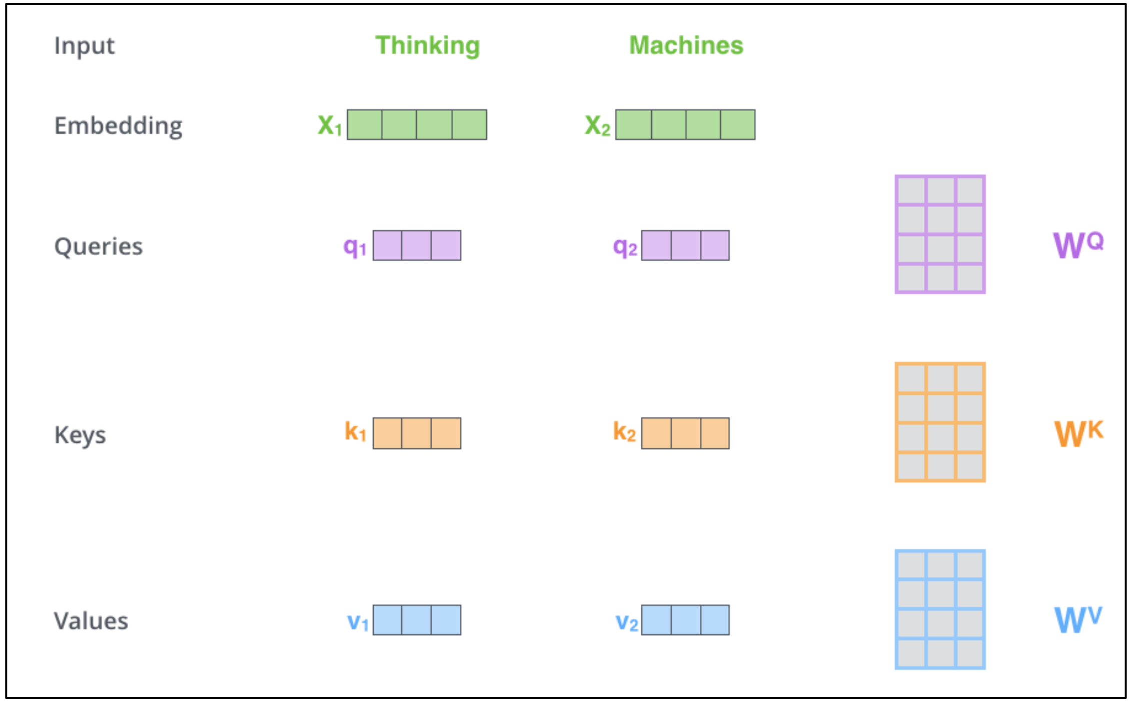


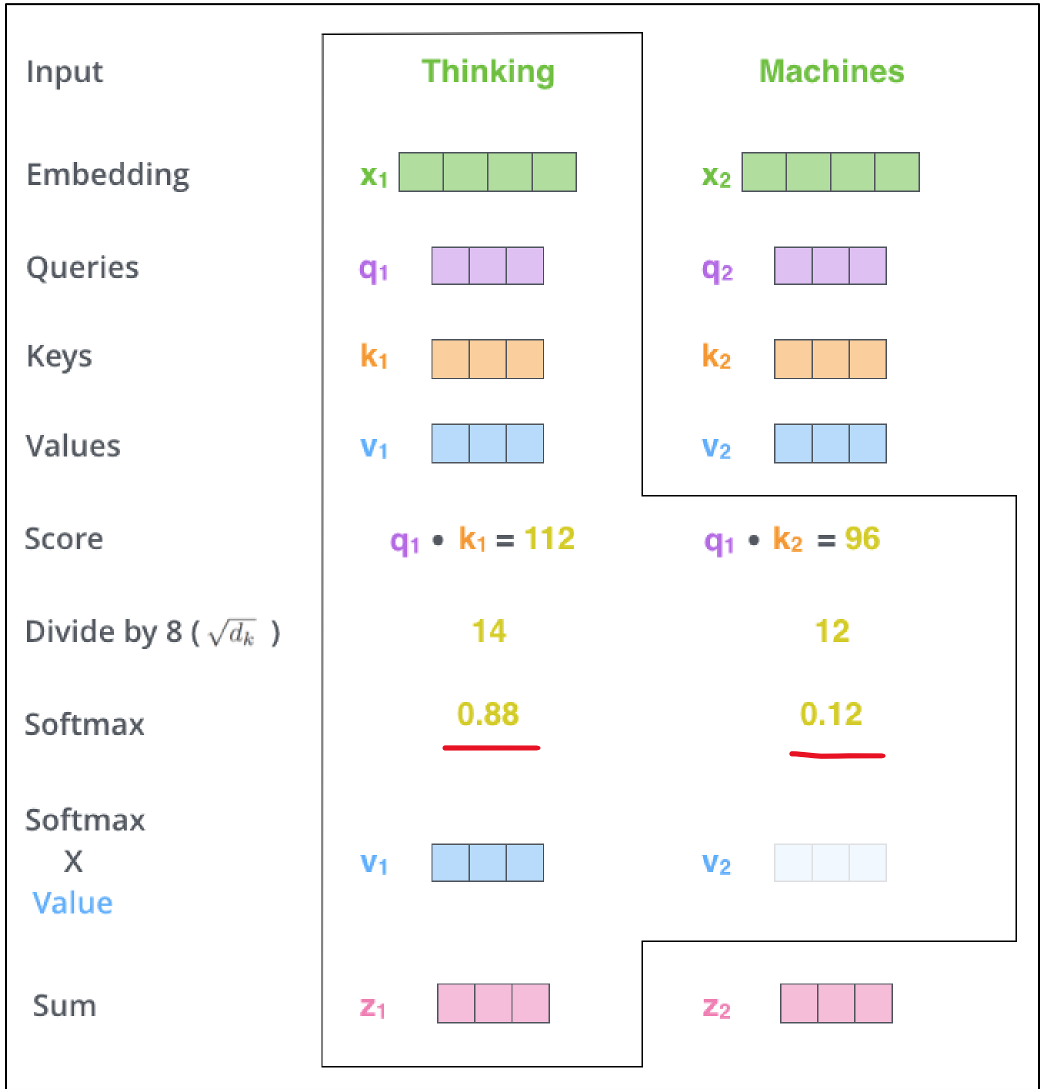
Self-attention detail



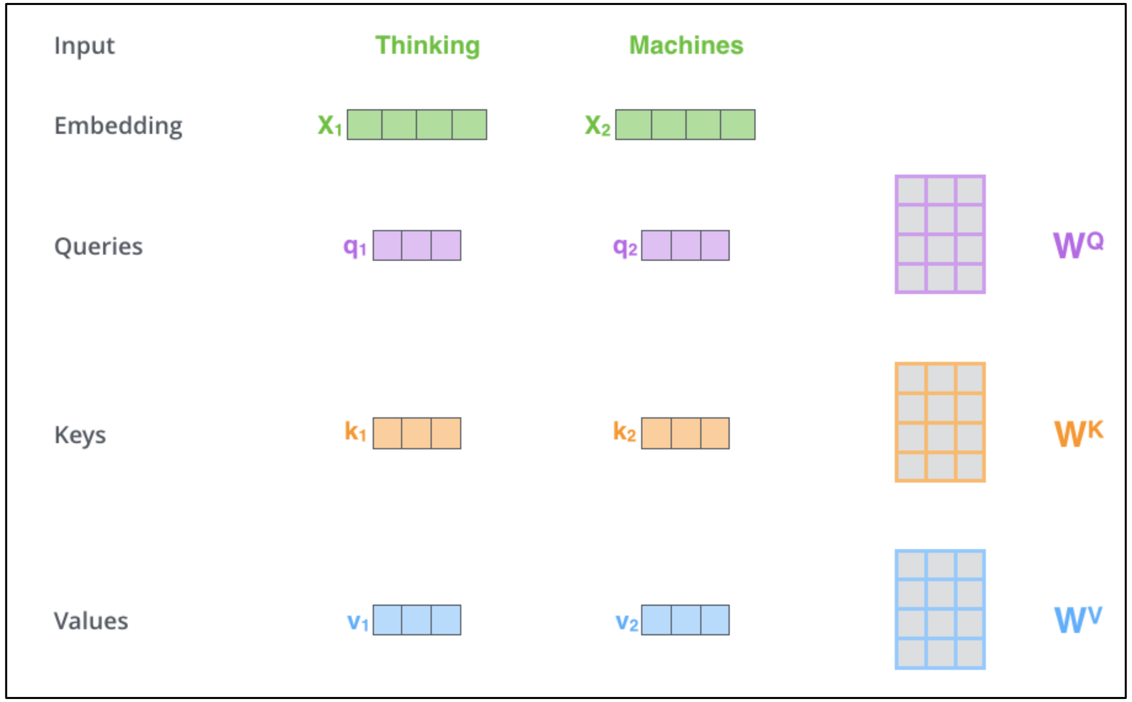


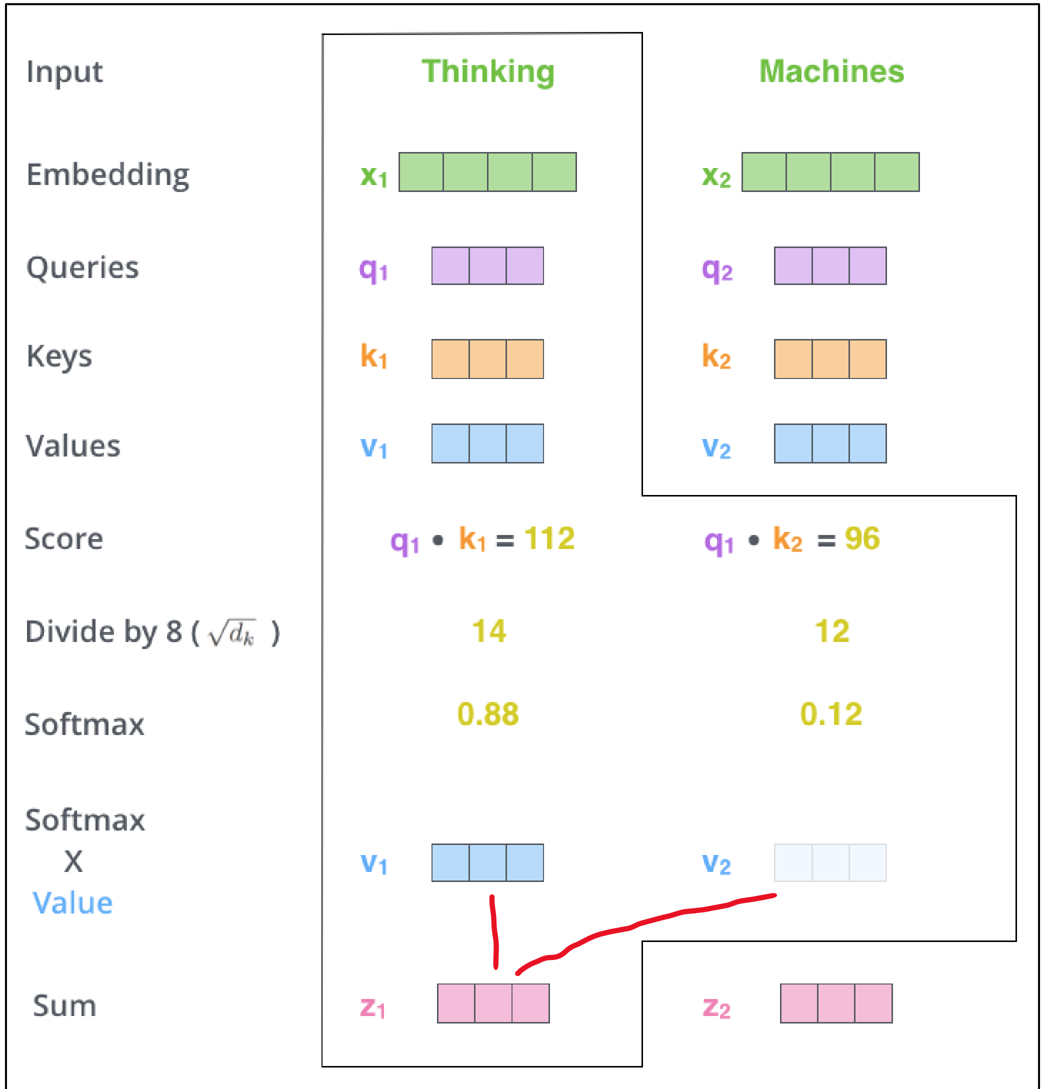
Self-attention detail



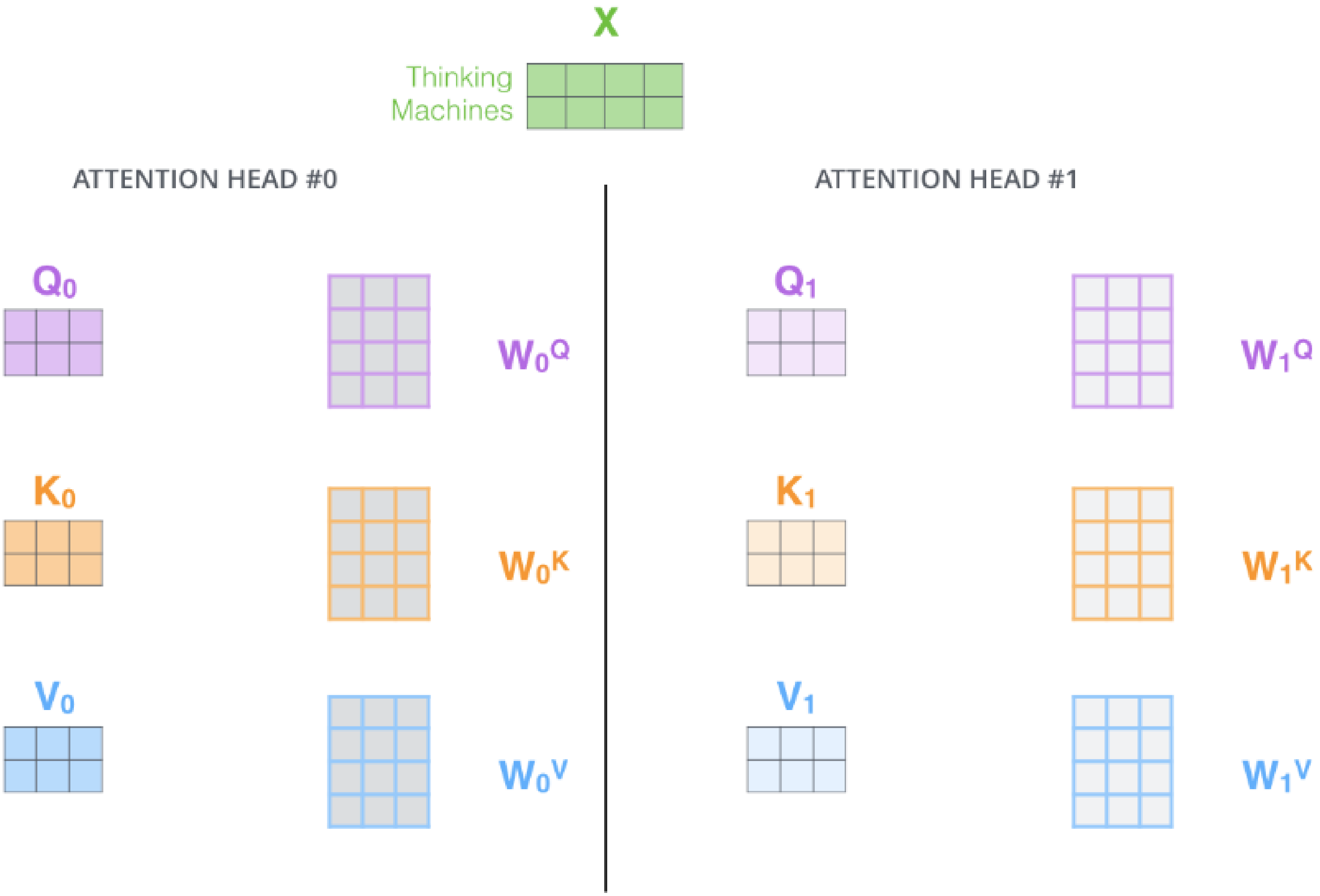


Self-attention detail

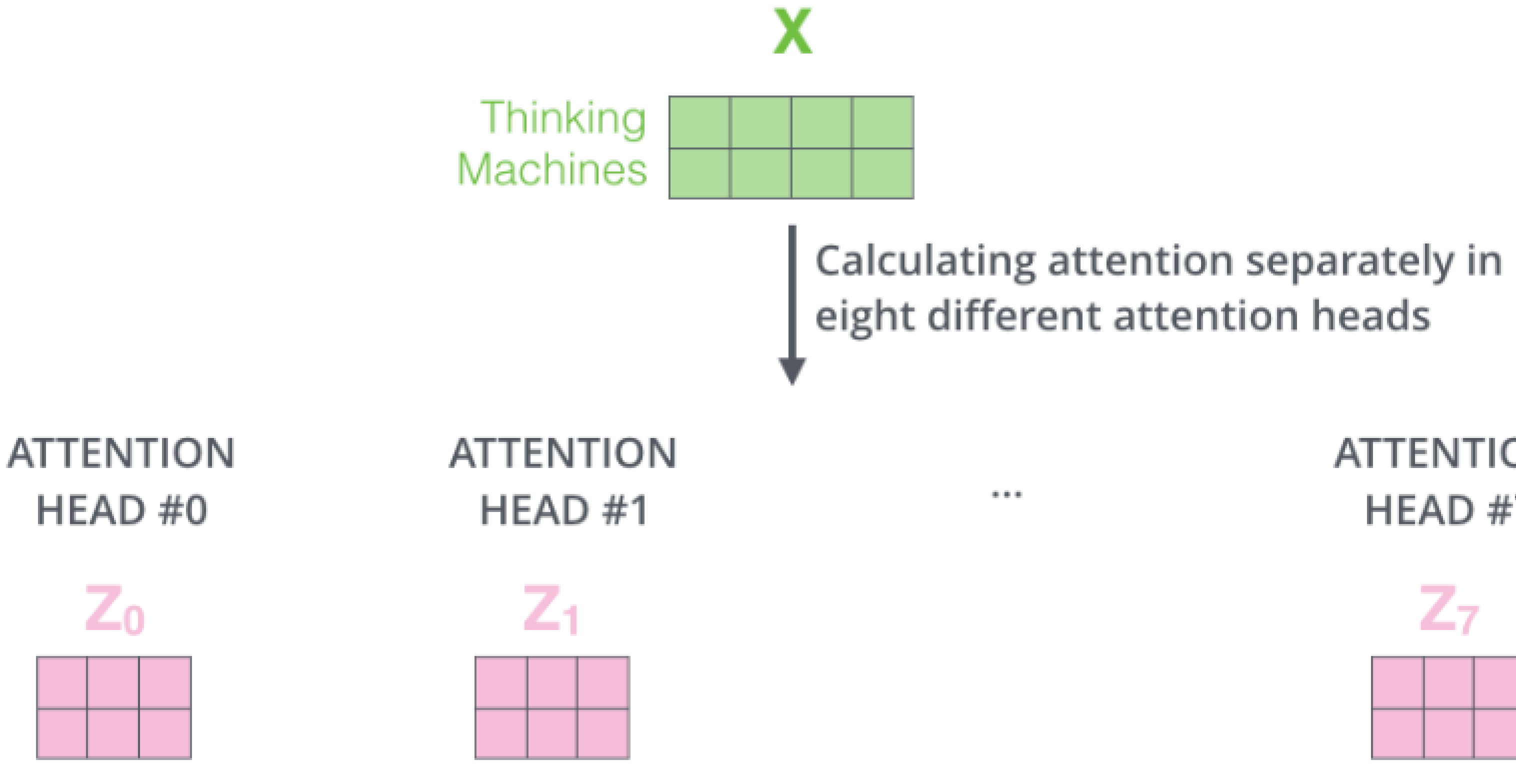




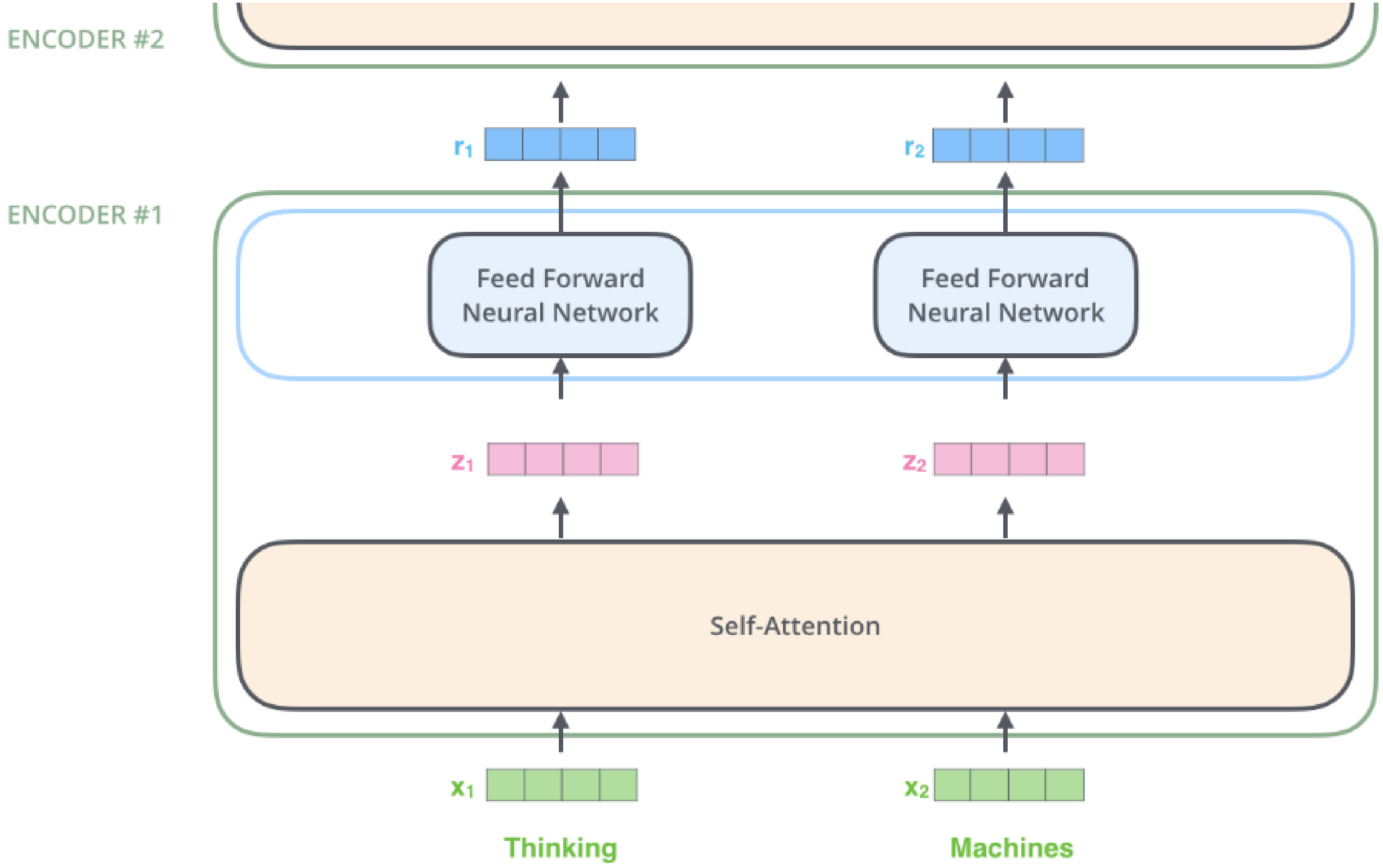
Multi-headed attention



Multi-headed attention

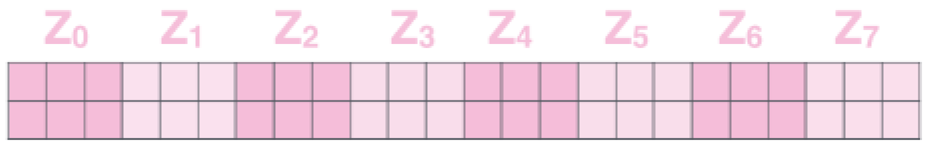


Encoder detail



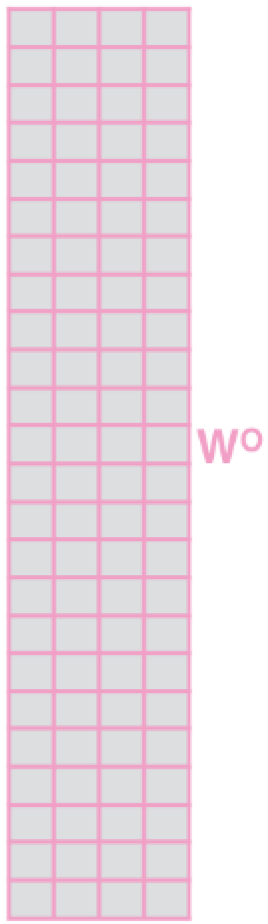
Multi-headed attention

1) Concatenate all the attention heads



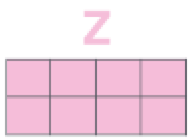
2) Multiply with a weight matrix W^O that was trained jointly with the model

X

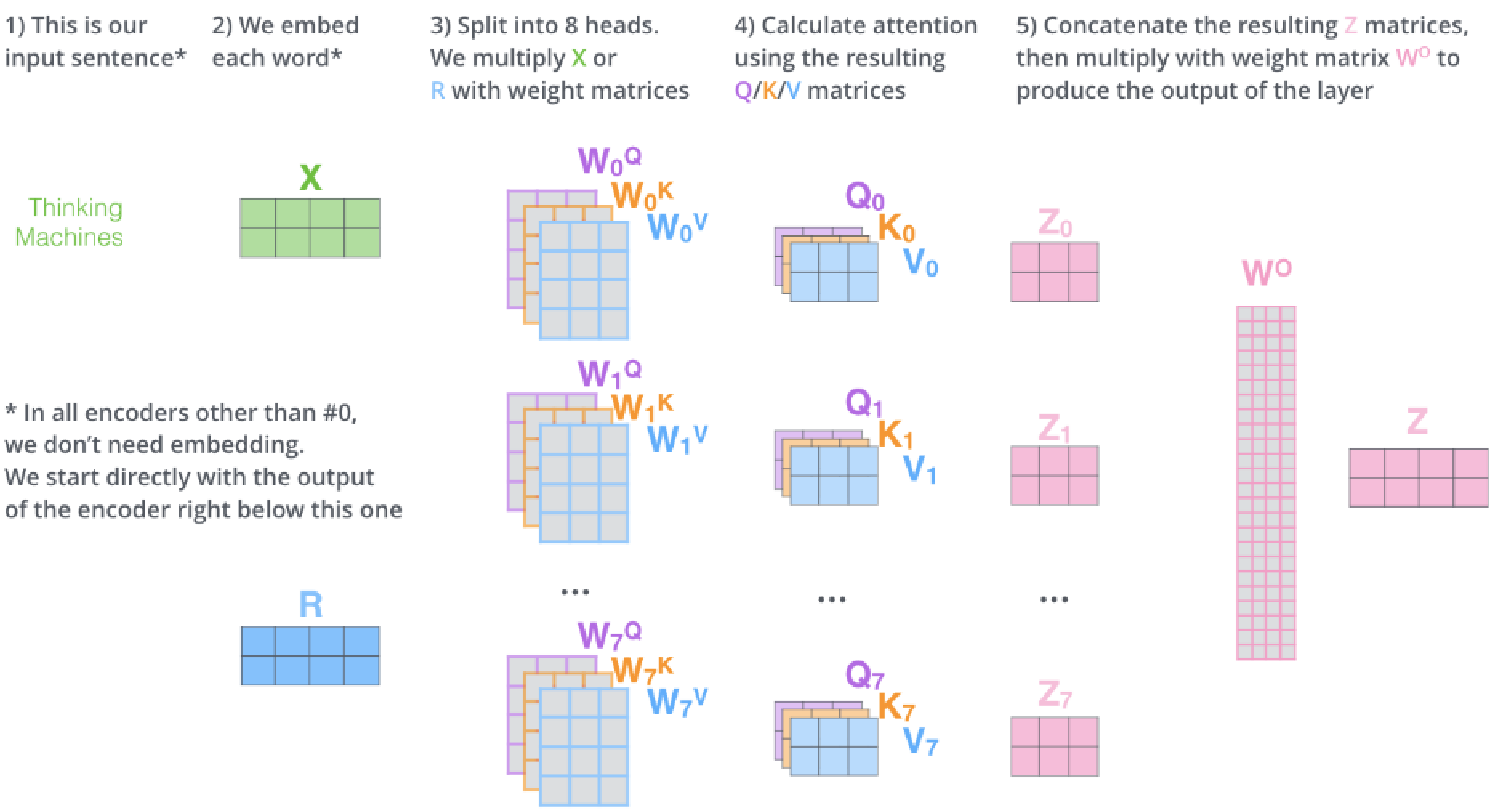


3) The result would be the Z matrix that captures information from all the attention heads. We can send this forward to the FFNN

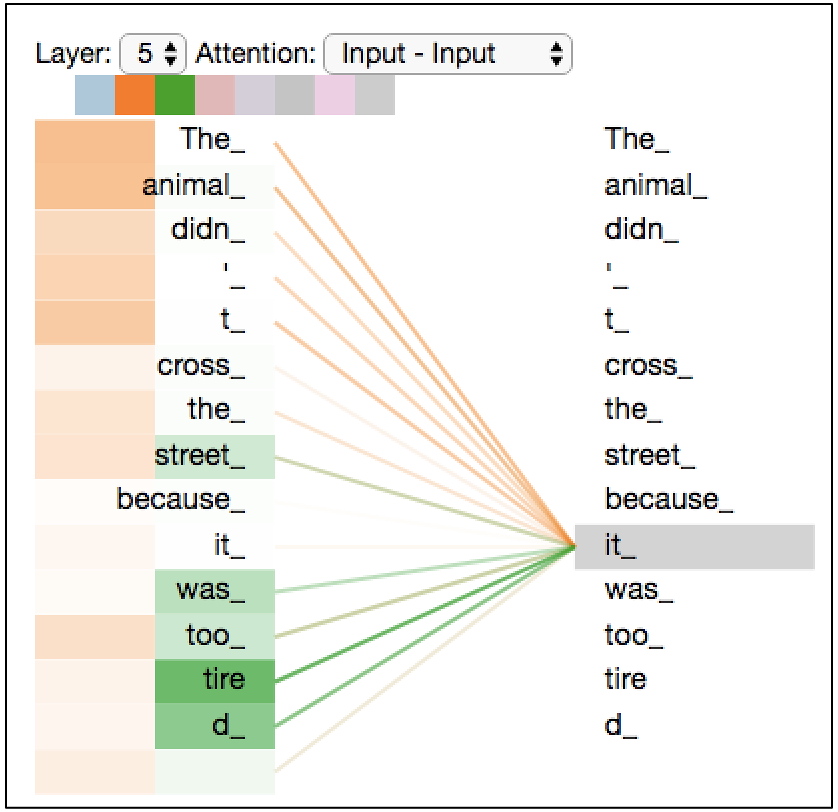
=

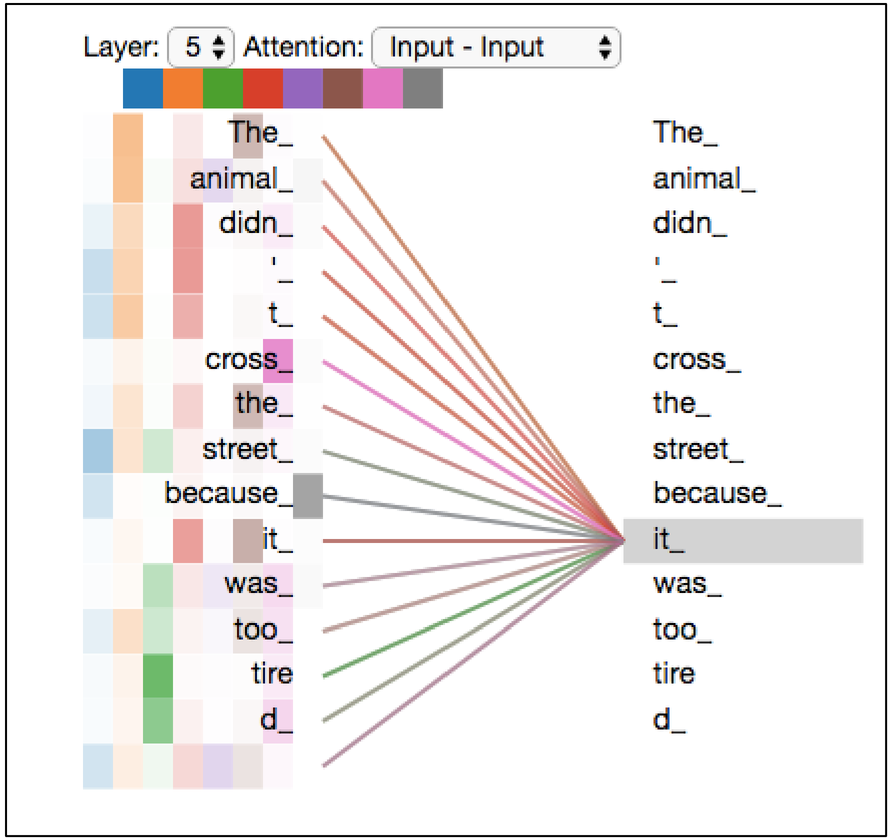


All self-attention steps

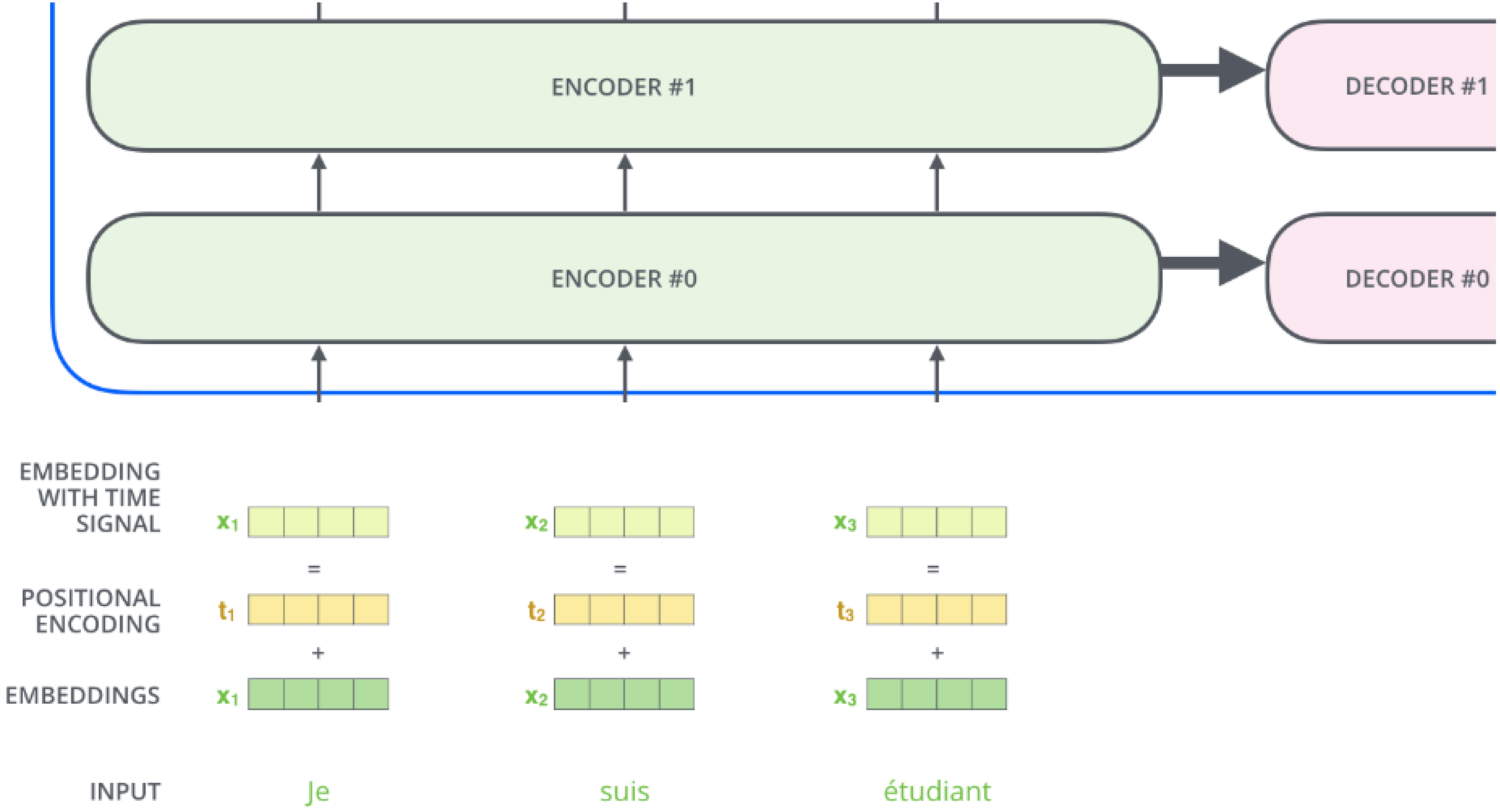


Multi-headed attention

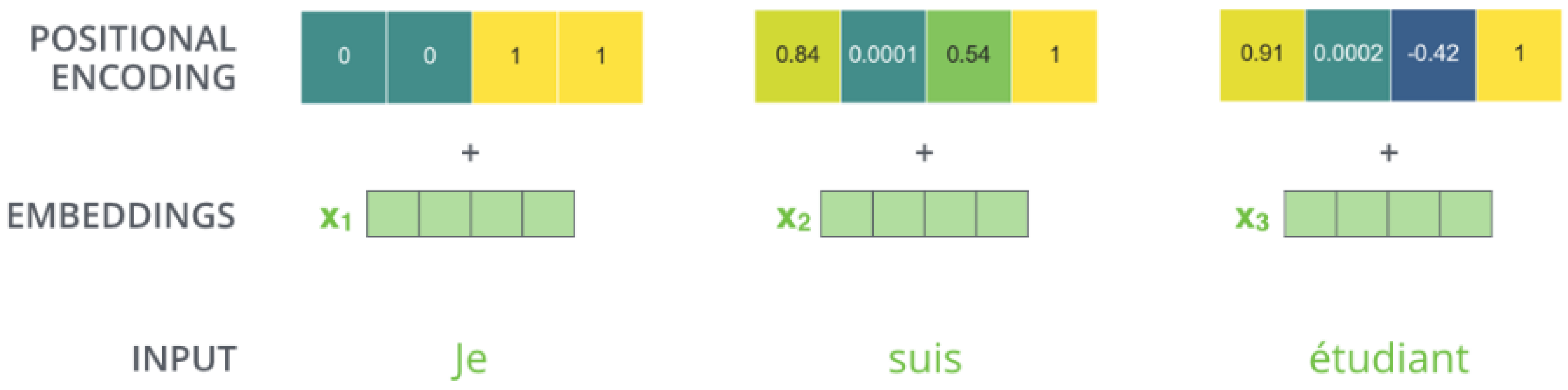




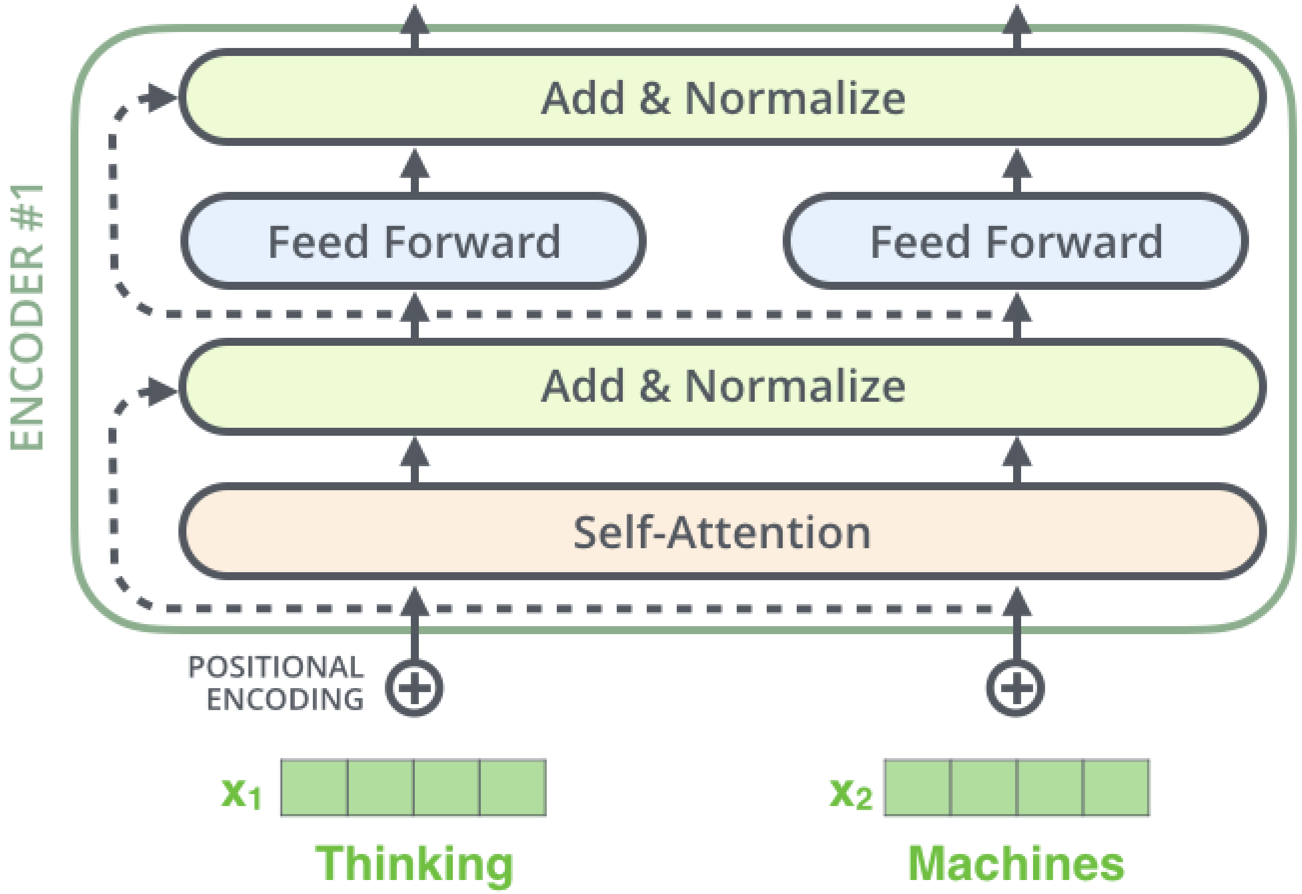
Positional embeddings



Positional embeddings



Residuals



Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
ENCODER #1
POSITIONAL
ENCODING
x1
x2
Thinking
Machines

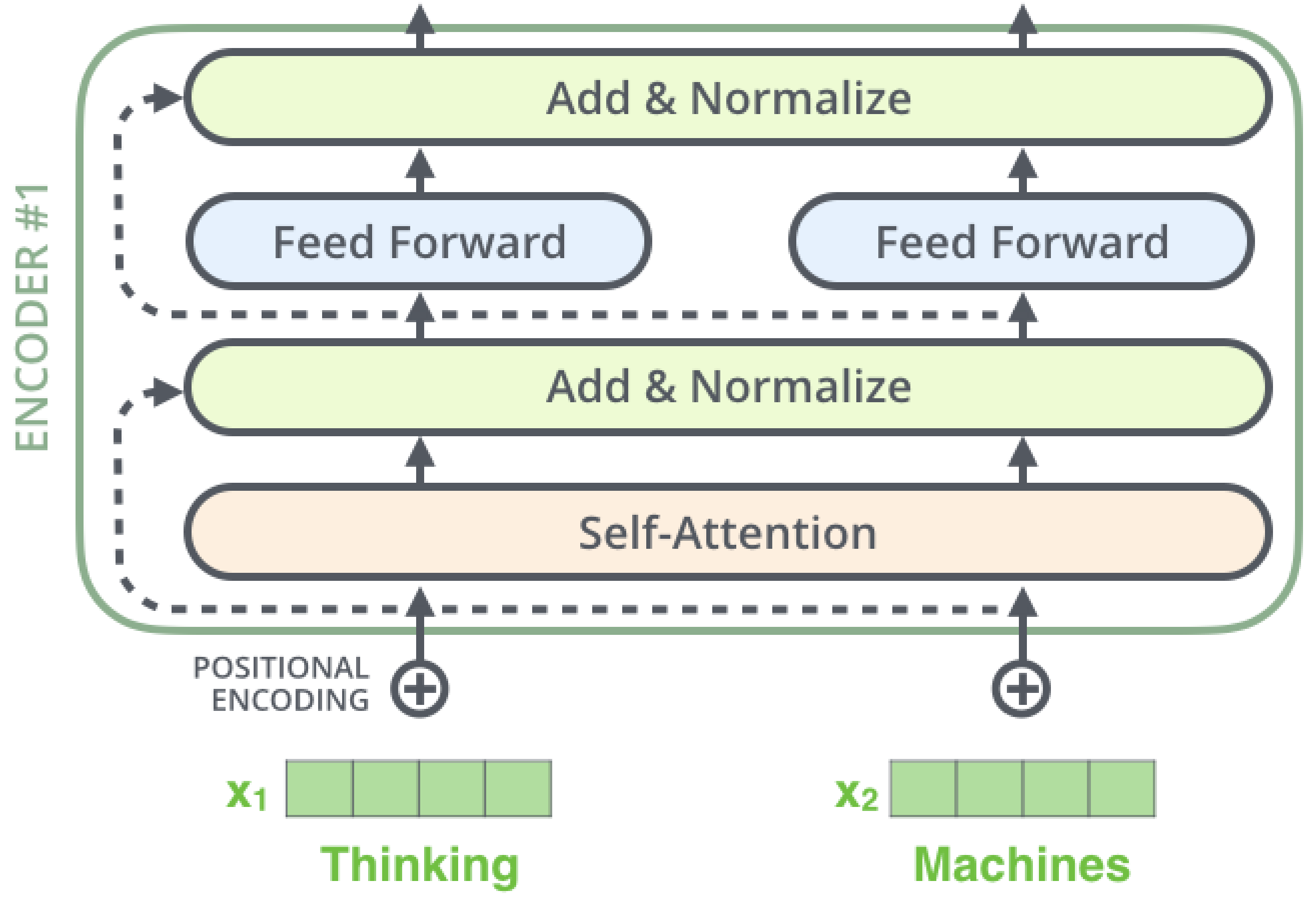
Layer normalization

Layer normalization is a training trick where you take the output from a neural net layer and statistically **normalize** it so that it has a mean value of 0 and a variance of 1

- This turns out to improve training speed and consistency.
- It's kind of just one of those handy tricks that people have discovered to generally improve deep learning, similar to dropout and L2 regularization.

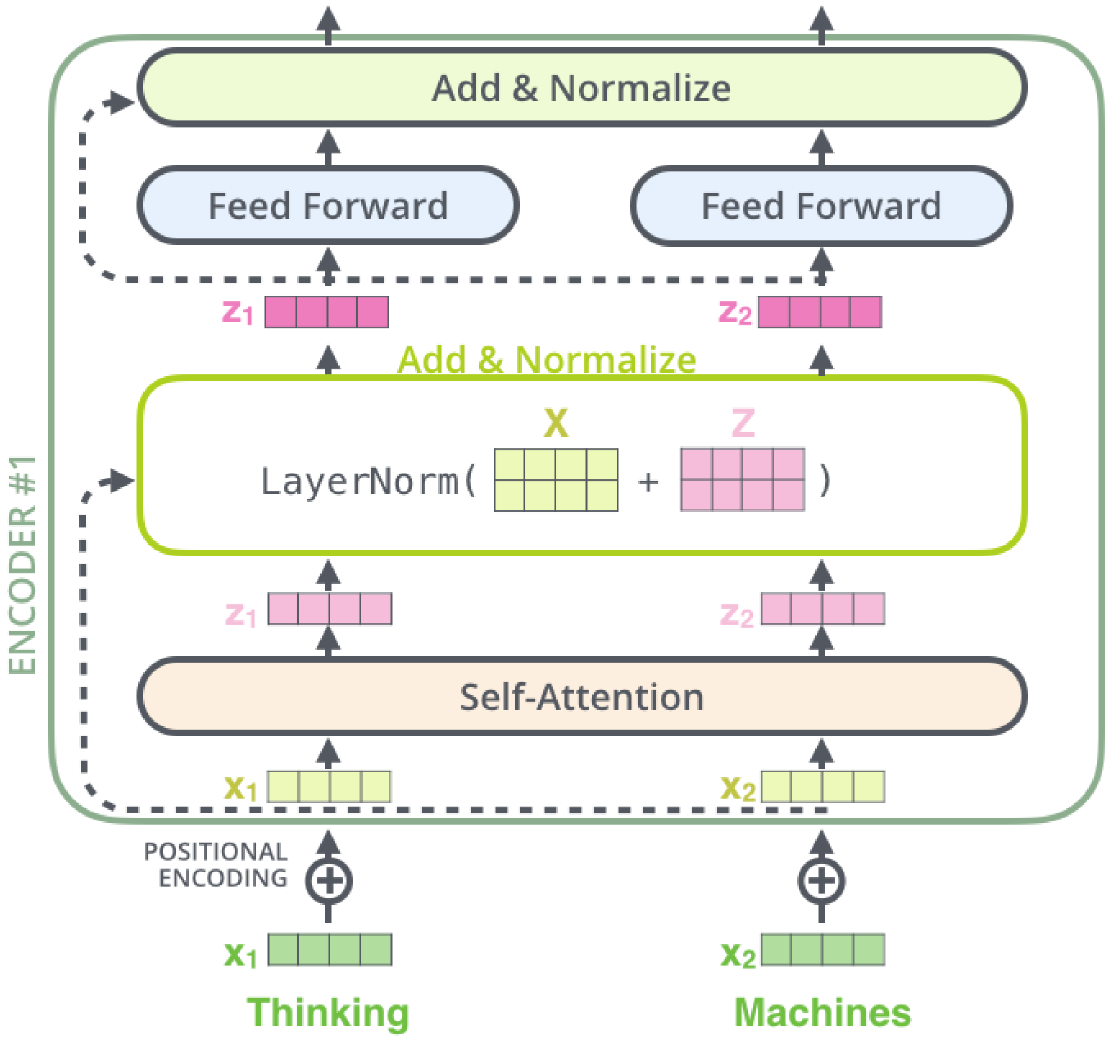
Layer normalization
JL Ba, JR Kiros, GE Hinton - arXiv preprint arXiv:1607.06450, 2016 - arxiv.org
… , we transpose batch **normalization** into **layer normalization** by computing the mean and variance used for **normalization** from all of the summed inputs to the neurons in a **layer** on a …
Save Cite Cited by 7452 Related articles All 6 versions

Residuals



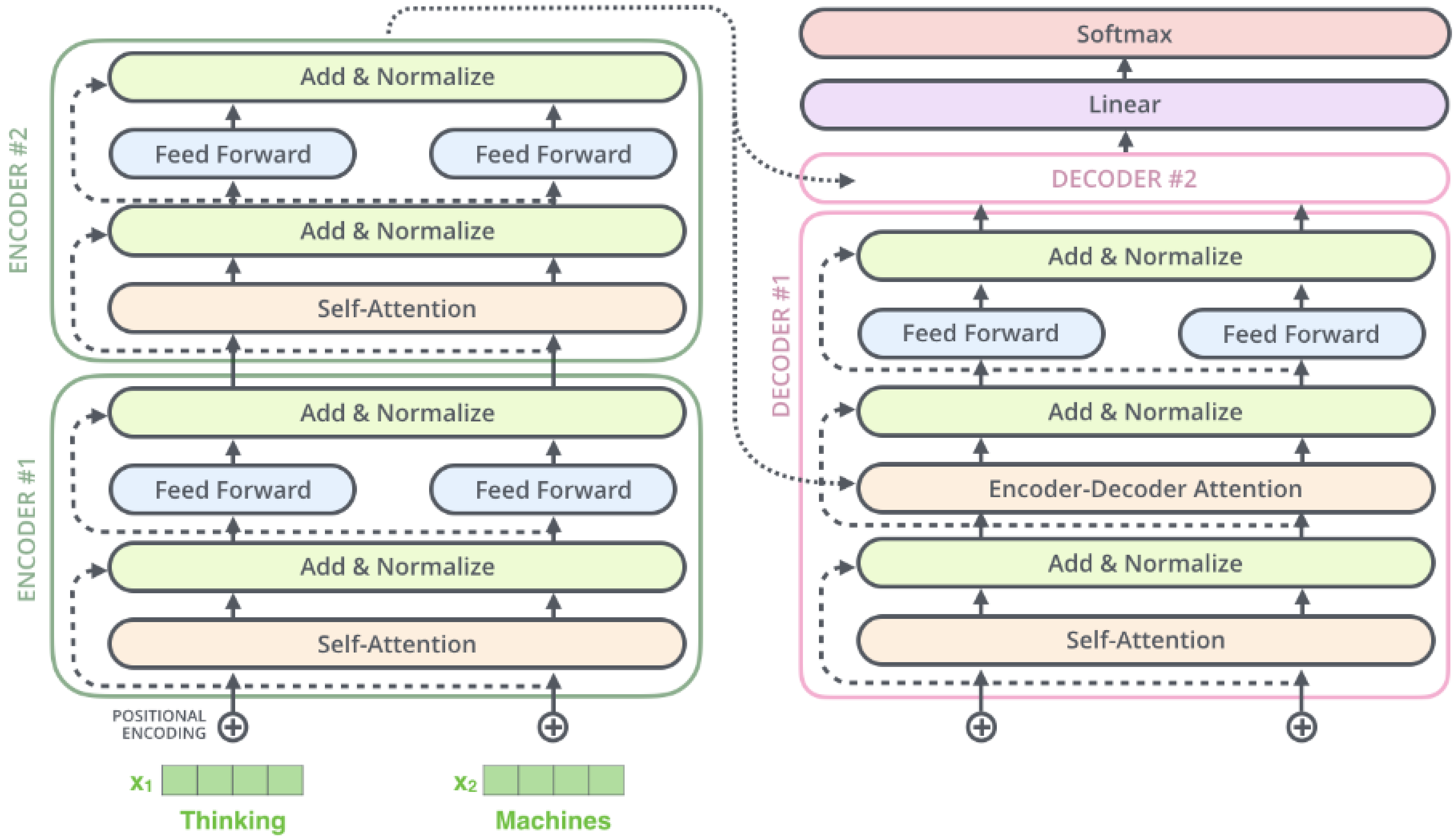
ENCODER #1
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
POSITIONAL
ENCODING
x1
x2
Thinking
Machines

Residuals

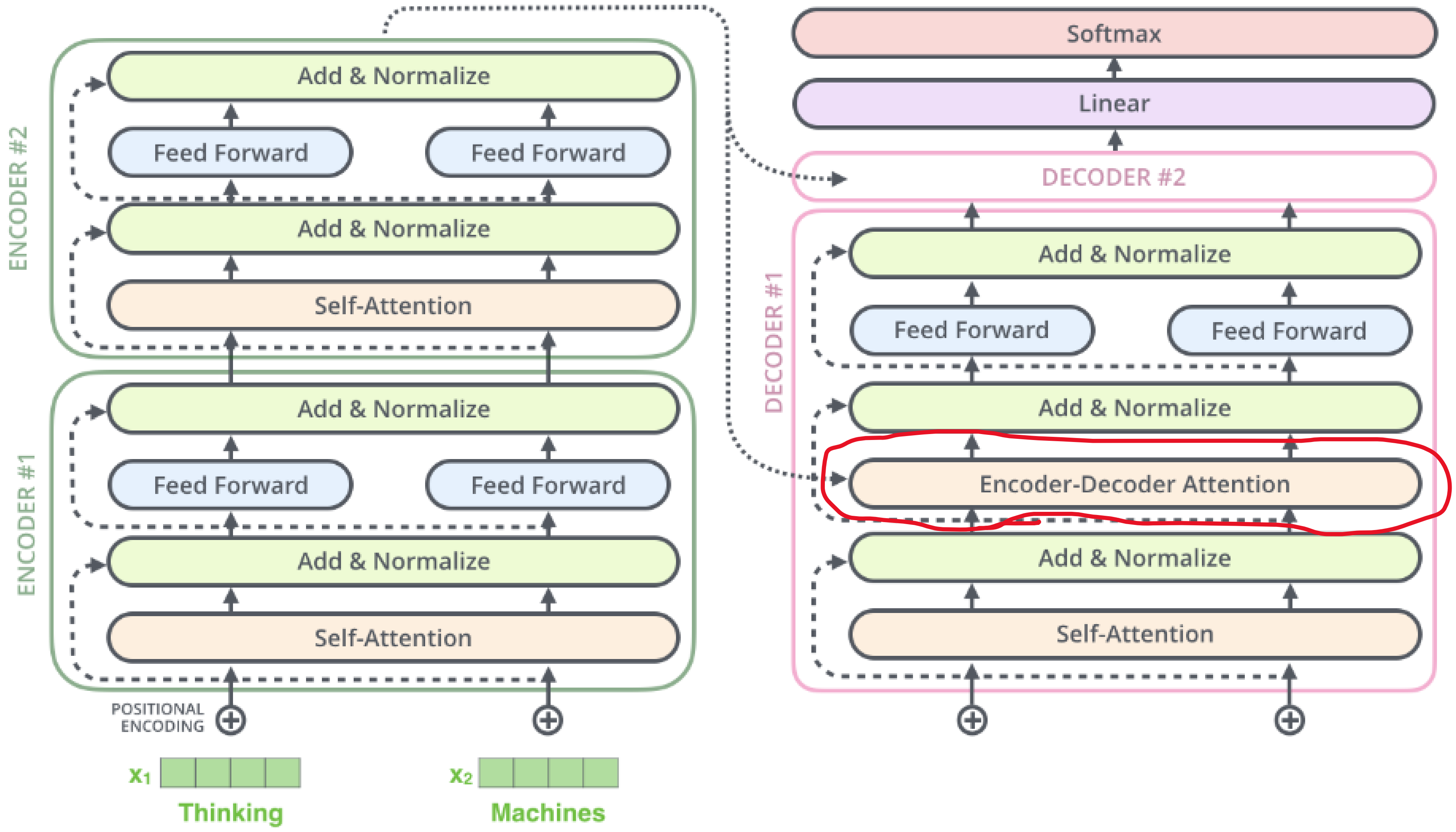


Add & Normalize
Feed Forward
Feed Forward
z1
z2
Add & Normalize
X
Z
LayerNorm(+)
z1
z2
Self-Attention
x1
x2
ENCODER #1
POSITIONAL
ENCODING
x1
x2
Thinking
Machines

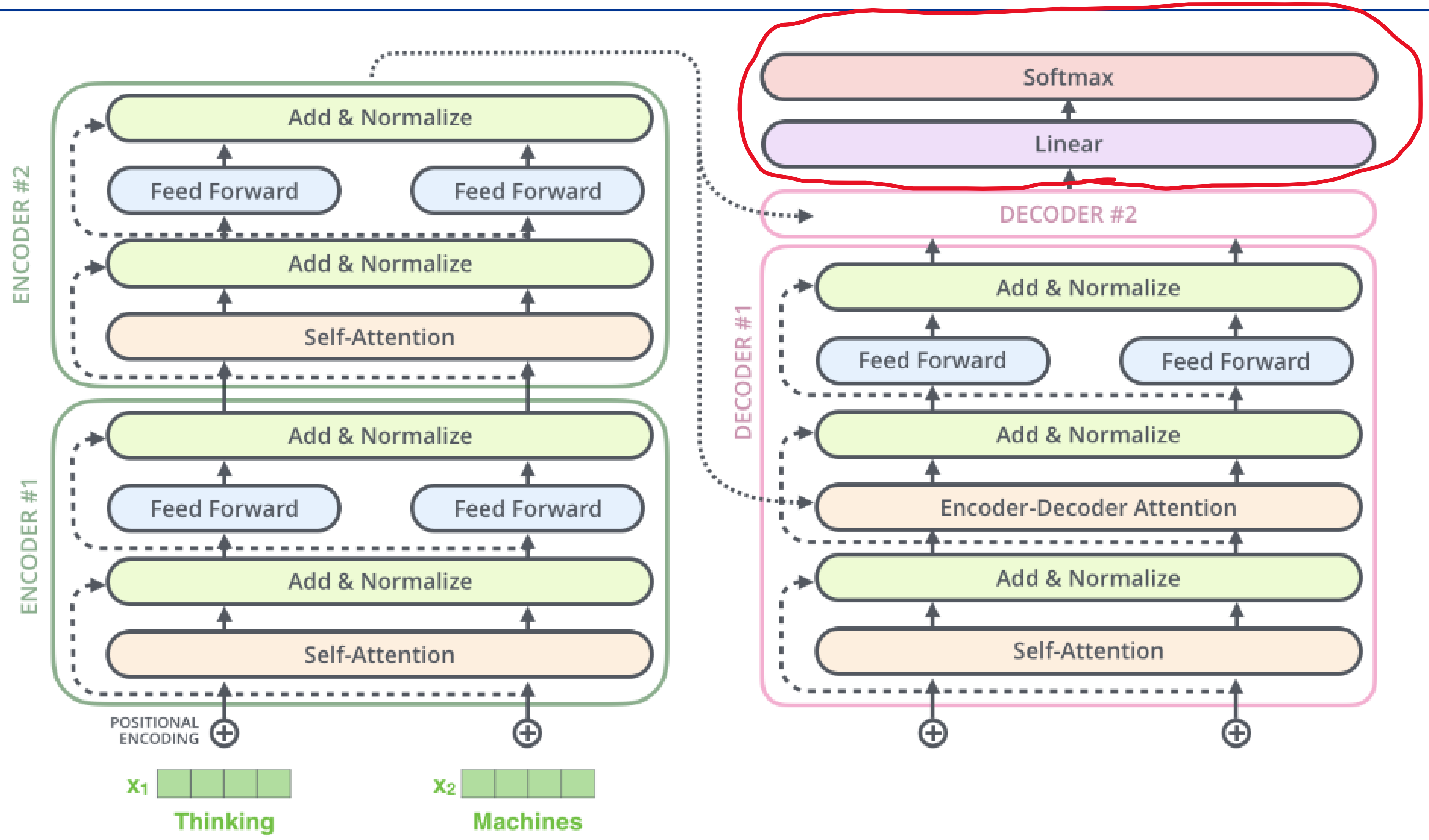
Encoder-decoder



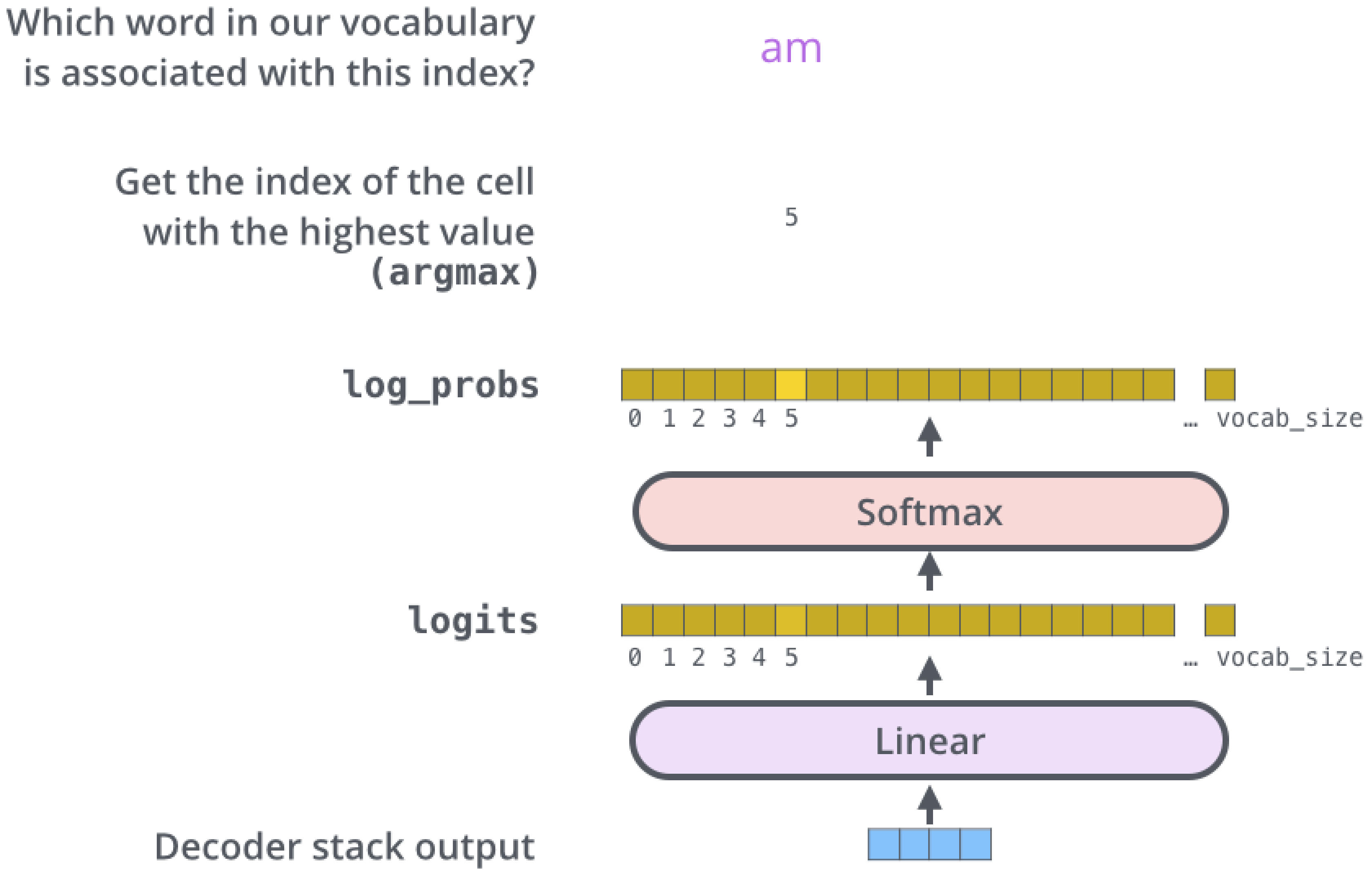
Encoder-decoder



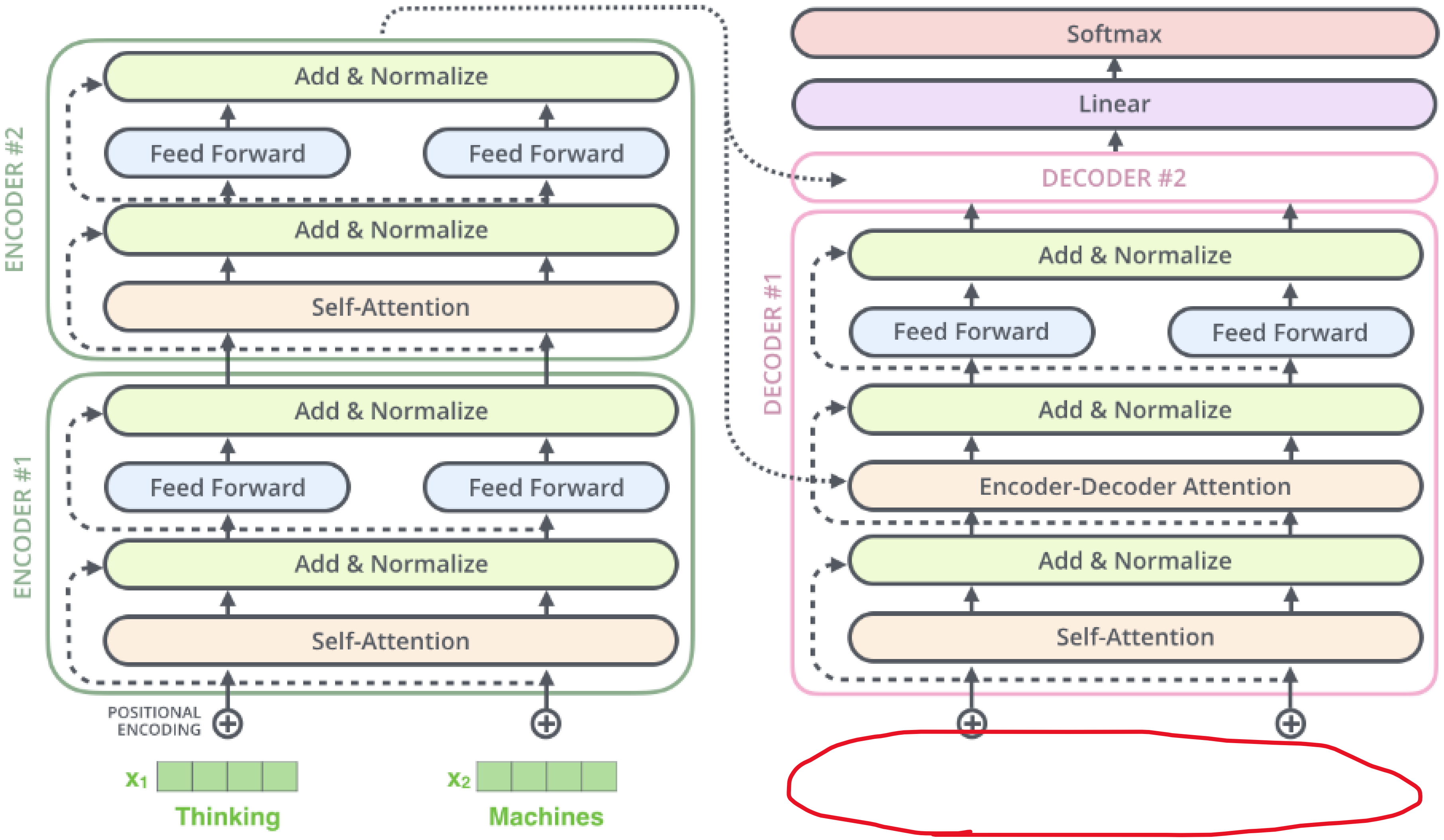
Encoder-decoder



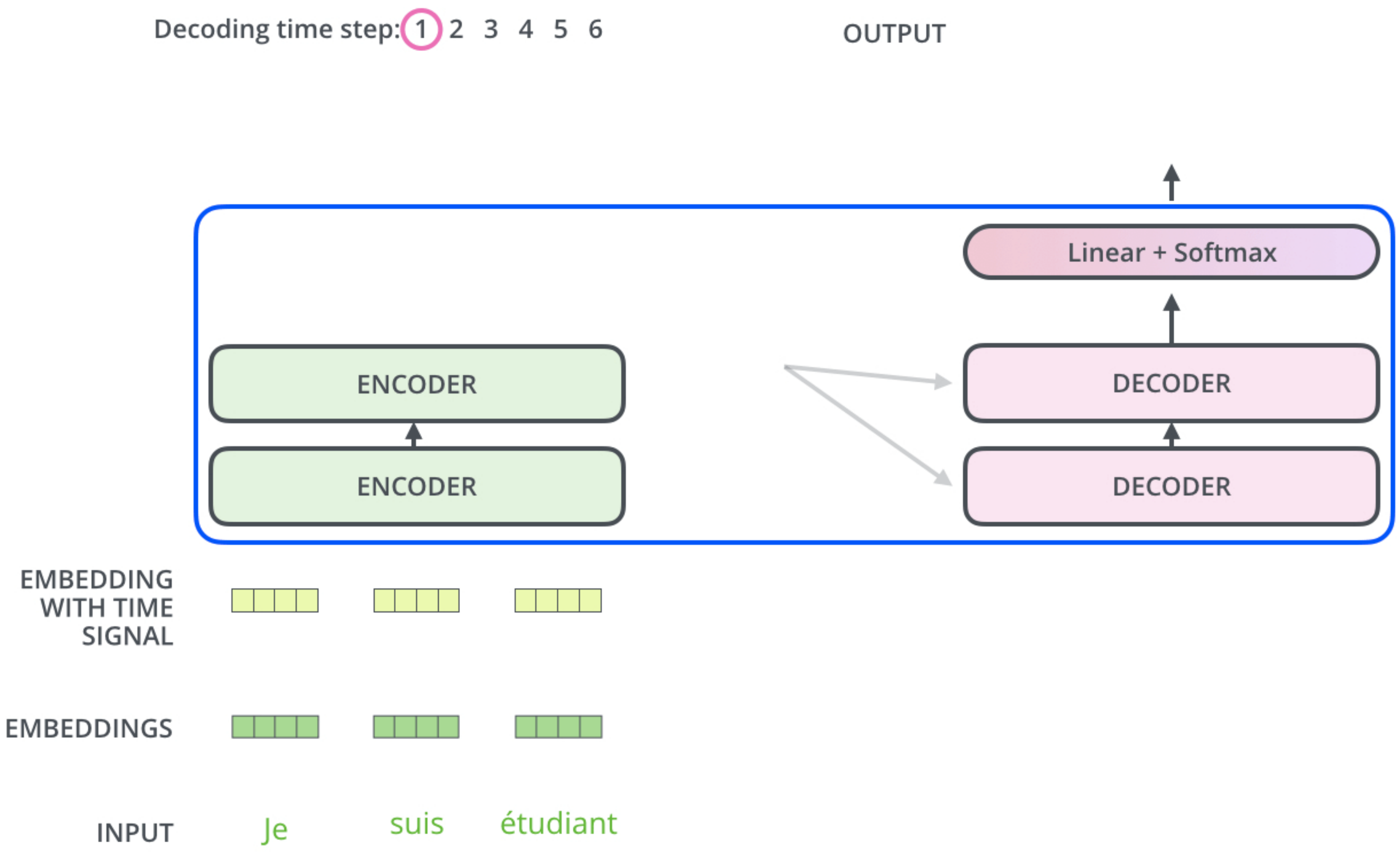
Decoder output layer



Encoder-decoder

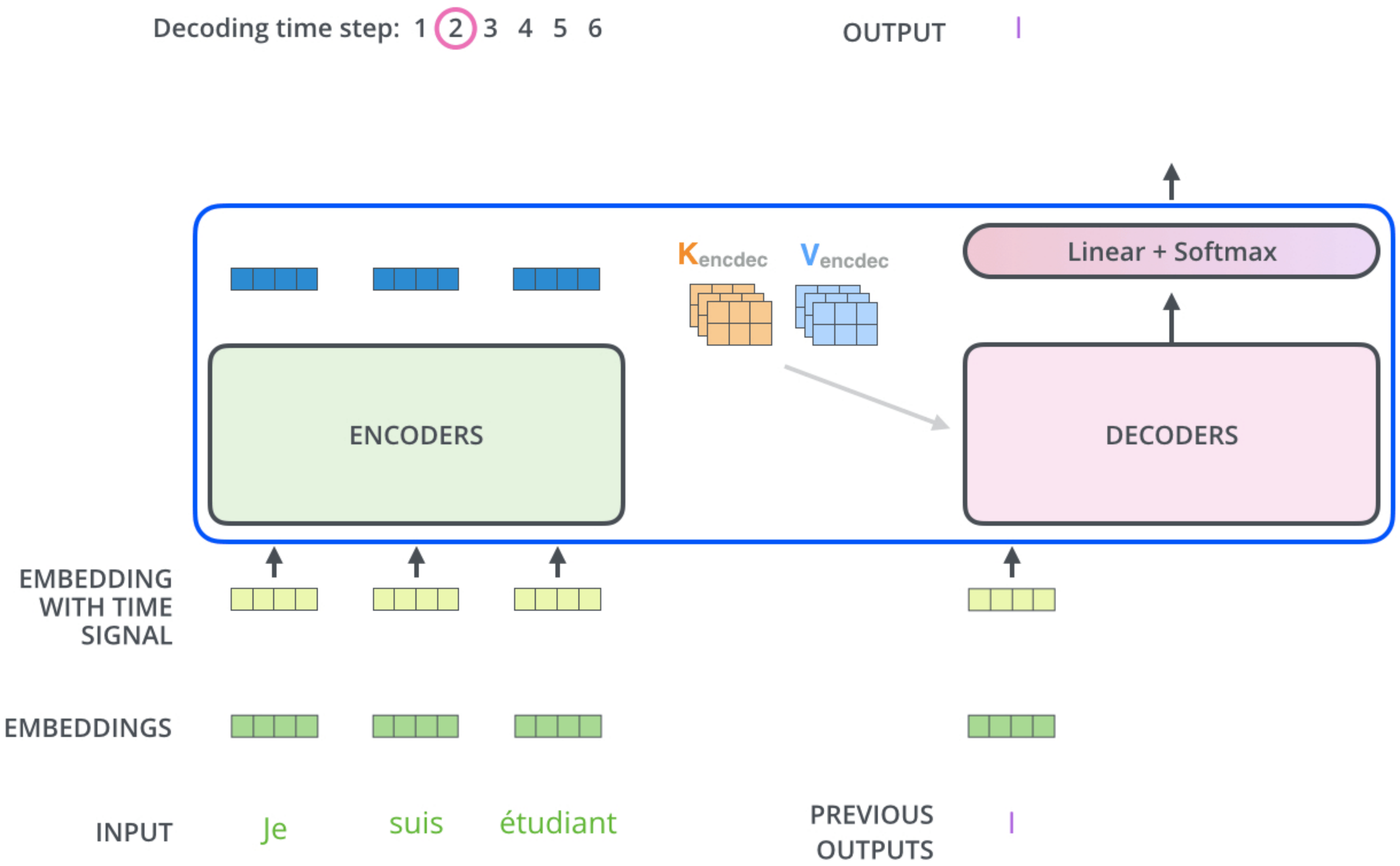


Decoder



Decoding time step: 1 2 3 4 5 6
OUTPUT
Linear + Softmax
ENCODER
ENCODER
DECODER
DECODER
EMBEDDING WITH TIME SIGNAL
EMBEDDINGS
INPUT
Je
suis
étudiant

Decoder



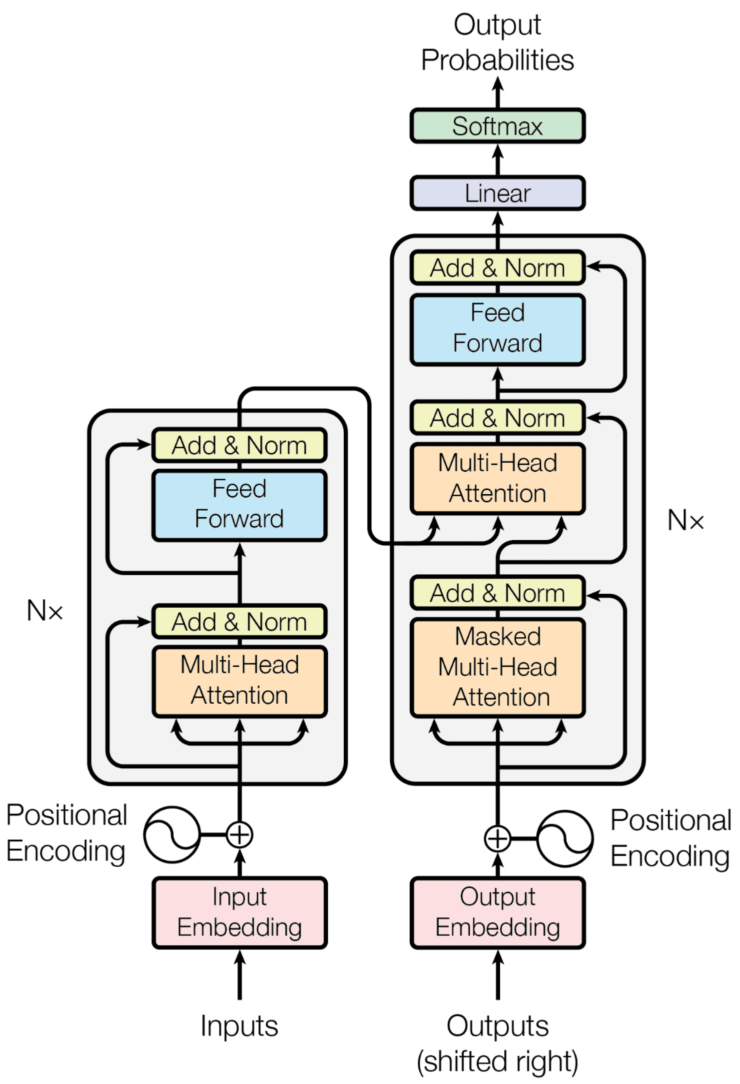
Decoding time step: 1 2 3 4 5 6
OUTPUT
I
Kencdec
Vencdec
Linear + Softmax
ENCODERS
DECODERS
EMBEDDING WITH TIME SIGNAL
EMBEDDINGS
INPUT
Je
suis
étudiant
PREVIOUS OUTPUTS
I

Transformer

Many components!

- Self-attention (NxN)
- Multiple self-attention heads per layer
- Multiple self-attention layers
- Encoder + decoder

Worth it?



Reading SST-2

```
1 display(dev_df)
```

	sentence	label
0	it 's a charming and often affecting journey .	1
1	unflinchingly bleak and desperate	0
2	allows us to hope that nolan is poised to emba...	1
3	the acting , costumes , music , cinematography...	1
4	it 's slow -- very , very slow .	0
...	...	...
867	has all the depth of a wading pool .	0
868	a movie with a real anarchic flair .	1
869	a subject like this should inspire reaction in...	0
870	... is an arthritic attempt at directing by ca...	0
871	looking aristocratic , luminous yet careworn i...	1

872 rows × 2 columns

Installing Transformers

```
!pip install Transformers
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting Transformers
  Downloading transformers-4.27.4-py3-none-any.whl (6.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.8/6.8 MB 94.6 MB/s eta 0:00:00
Collecting tokenizers!=0.11.3,<0.14,>=0.11.1
  Downloading tokenizers-0.13.3-cp39-cp39-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (7.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.8/7.8 MB 98.2 MB/s eta 0:00:00
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.9/dist-packages (from Transformers) (23.0)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.9/dist-packages (from Transformers) (6.0)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.9/dist-packages (from Transformers) (4.65.0)
Requirement already satisfied: requests in /usr/local/lib/python3.9/dist-packages (from Transformers) (2.27.1)
Requirement already satisfied: filelock in /usr/local/lib/python3.9/dist-packages (from Transformers) (3.10.7)
Collecting huggingface-hub<1.0,>=0.11.0
  Downloading huggingface_hub-0.13.4-py3-none-any.whl (200 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 200.1/200.1 KB 26.7 MB/s eta 0:00:00
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.9/dist-packages (from Transformers) (2022.10.31)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.9/dist-packages (from Transformers) (1.22.4)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.9/dist-packages (from huggingface-hub<1.0,>=0.11.0->Transformers)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.9/dist-packages (from requests->Transformers) (3.4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist-packages (from requests->Transformers) (2022.12.7)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.9/dist-packages (from requests->Transformers) (1.26.15)
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3.9/dist-packages (from requests->Transformers) (2.0.12)
Installing collected packages: tokenizers, huggingface-hub, Transformers
Successfully installed Transformers-4.27.4 huggingface-hub-0.13.4 tokenizers-0.13.3
```

Transformer tokenizer

```python
from transformers import BertTokenizerFast

# This command goes out onto the Hugging Face website and downloads the tokenizer
# associated with the pretrained bert-base-uncased model

# We'll talk later about how this pretraining works, but the long story short is
# that this thing will do all the preprocessing we need for us.
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
```

```
Downloading (…)okenizer_config.json: 100% 28.0/28.0 [00:00<00:00, 698B/s]
Downloading (…)solve/main/vocab.txt: 100% 232k/232k [00:00<00:00, 550kB/s]
Downloading (…)/main/tokenizer.json: 100% 466k/466k [00:00<00:00, 1.09MB/s]
loading file vocab.txt from cache at /root/.cache/huggingface/hub/models--bert-base-uncased/snapshots/0a6aa9128b6194f4f3c4db429b6cb4891cdb421b/vocab.txt
loading file tokenizer.json from cache at /root/.cache/huggingface/hub/models--bert-base-uncased/snapshots/0a6aa9128b6194f4f3c4db429b6cb4891cdb421b/toke
loading file added_tokens.json from cache at None
loading file special_tokens_map.json from cache at None
loading file tokenizer_config.json from cache at /root/.cache/huggingface/hub/models--bert-base-uncased/snapshots/0a6aa9128b6194f4f3c4db429b6cb4891cdb42
Downloading (…)lve/main/config.json: 100% 570/570 [00:00<00:00, 15.9kB/s]
loading configuration file config.json from cache at /root/.cache/huggingface/hub/models--bert-base-uncased/snapshots/0a6aa9128b6194f4f3c4db429b6cb4891c
```

Transformer tokenizer

```
# These tokenizers are very simple to use
tokenized = tokenizer.encode_plus('The tokenizer has lots of functionality.')
from pprint import pprint
pprint(tokenized)
```

```
{'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 'input_ids': [101, 1996, 19204, 17629, 2038, 7167, 1997, 15380, 1012, 102],
 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]}
```

```
# By default it returns these things as lists
pprint({key:type(value) for key, value in tokenized.items()})
```

```
{'attention_mask': <class 'list'>,
 'input_ids': <class 'list'>,
 'token_type_ids': <class 'list'>}
```

```
# But you can tell it to return PyTorch tensors instead
tokenized_pt = tokenizer.encode_plus('The tokenizer has lots of functionality.', return_tensors='pt')
from pprint import pprint
pprint(tokenized_pt)
```

```
{'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]),
 'input_ids': tensor([[  101,  1996, 19204, 17629,  2038,  7167,  1997, 15380,  1012,   102]]),
 'token_type_ids': tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])}
```

Transformer tokenizer

```
# One thing to note is that transformer-based models operate on wordpieces, not words
# Also note how it inserts a [CLS] token at the beginning and a [SEP] token at the end
print(tokenizer.convert_ids_to_tokens(tokenized['input_ids']))
```

```
['[CLS]', 'the', 'token', '##izer', 'has', 'lots', 'of', 'functionality', '.', '[SEP]']
```

Transformer tokenizer

```
# If we give it a list of texts, it will return a batch of results (and do padding!)
texts = ['This is the first sentence.',
         'This may be the second sentence, I really do not know.',
         'I never learned to count.']

tokenizeds = tokenizer.batch_encode_plus(texts, return_tensors='pt', padding=True)
pprint(tokenizeds)
```

```
{'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]]),
 'input_ids': tensor([[ 101, 2023, 2003, 1996, 2034, 6251, 1012,  102,    0,    0,    0,    0,
            0,    0,    0],
        [ 101, 2023, 2089, 2022, 1996, 2117, 6251, 1010, 1045, 2428, 2079, 2025,
         2113, 1012,  102],
        [ 101, 1045, 2196, 4342, 2000, 4175, 1012,  102,    0,    0,    0,    0,
            0,    0,    0]]),
 'token_type_ids': tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])}
```

```
# The default behavior is to pad sequences out to the max sequence length in the batch
print(tokenizer.convert_ids_to_tokens(tokenizeds['input_ids'][0]))
```

```
['[CLS]', 'this', 'is', 'the', 'first', 'sentence', '.', '[SEP]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]', '[PAD]']
```

Dataset

```python
class SST2TransformerDataset(Dataset):
  def __init__(self,
               labels=None,
               texts=None):

    self.y = torch.tensor(labels,dtype=torch.int64)
    self.texts = texts

  def __len__(self):
    return self.y.shape[0]

  def __getitem__(self, idx):
    rdict = {
      'y': self.y[idx],
      'text': self.texts[idx]
    }
    return rdict
```

```python
train_dataset = SST2TransformerDataset(train_df['label'], train_df['sentence'])
dev_dataset = SST2TransformerDataset(dev_df['label'], dev_df['sentence'])

print(train_dataset[0])

```

```
{'y': tensor(0), 'text': 'hide new secretions from the parental units '}
```

DataLoader

```
def SST2_transformer_collate(batch:List[Dict[str, Union[torch.Tensor,str]]]):
  y_batch = torch.tensor([example['y'] for example in batch])
  tokenized_batch = tokenizer.batch_encode_plus([example['text'] for example in batch],
                                                return_tensors='pt', padding=True)

  return {
      'y':y_batch,
      'input_ids':tokenized_batch['input_ids'],
      'attention_mask':tokenized_batch['attention_mask']
  }

batch_size = 10
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, collate_fn = SST2_transformer_collate, shuffle=True)
dev_dataloader = DataLoader(dev_dataset, batch_size=batch_size, collate_fn = SST2_transformer_collate, shuffle=False)
```

Pretrained transformers

```python
from transformers import BertModel
# Like the tokenizer, we can just download one of these from Hugging Face
bert = BertModel.from_pretrained('bert-base-uncased')
```

```
loading configuration file config.json from cache at /root/.cache/huggingface/hub/models--bert-base-uncased/snapshots/0a6aa9128b6194f4f3c4db429b6cb4891c
Model config BertConfig {
  "architectures": [
    "BertForMaskedLM"
  ],
  "attention_probs_dropout_prob": 0.1,
  "classifier_dropout": null,
  "gradient_checkpointing": false,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "model_type": "bert",
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  "pad_token_id": 0,
  "position_embedding_type": "absolute",
  "transformers_version": "4.27.4",
  "type_vocab_size": 2,
  "use_cache": true,
  "vocab_size": 30522
}
```

Downloading pytorch_model.bin: 100% 440M/440M [00:02<00:00, 276MB/s]

Pretrained Transformers

```python
# And then using them is very easy:
bert_result = bert(input_ids = first_train_batch['input_ids'],
                   attention_mask = first_train_batch['attention_mask']) #This is how we tell it where the masking is

# Like the LSTM returning both the intermediate output values and the final hidden state,
# The BERT model returns the last hidden state (for each input), and the final pooler output
pprint({key:value.shape for key, value in bert_result.items()})
```

```
{'last_hidden_state': torch.Size([10, 20, 768]),
 'pooler_output': torch.Size([10, 768])}
```

Transformer-using model

```python
class BertClassifier(pl.LightningModule):
  def __init__(self,
               learning_rate:float,
               num_classes:int,
               freeze_bert:bool=False,
               **kwargs):
    super().__init__(**kwargs)

    # Like with the LSTM, we'll define a central BERT we're gonna use
    # Again, this will download this from Hugging Face in the background
    self.bert = BertModel.from_pretrained('bert-base-uncased')

    # If we want to speed up training, we can freeze the BERT module and train
    # just the output layer
    if freeze_bert:
      for param in self.bert.parameters():
        param.requires_grad = False

    # Then the only other thing we need is an output layer, whose input size will
    # be the BERT's output size (768), which can can find as follows:
    self.output_layer = torch.nn.Linear(self.bert.config.hidden_size, num_classes)

    self.learning_rate = learning_rate
    self.train_accuracy = Accuracy(task='multiclass', num_classes=num_classes)
    self.val_accuracy = Accuracy(task='multiclass', num_classes=num_classes)
```

Transformer-using model

```python
def forward(self, y:torch.Tensor, input_ids:torch.Tensor, attention_mask:torch.Tensor):
    # And then the forward function is pretty simple--way simpler than with the LSTM
    bert_result = self.bert(input_ids=input_ids,
                            attention_mask=attention_mask) # this is how we tell the BERT where the padding is
    # Typically we just use the pooler output for classification
    cls_output = bert_result['pooler_output']

    py_logits = self.output_layer(cls_output)
    py = torch.argmax(py_logits, dim=1)
    loss = torch.nn.functional.cross_entropy(py_logits, y, reduction='mean')
    return {'py':py,
            'loss':loss}
```

Transformer-using model

```python
bert_model = BertClassifier(learning_rate=2e-5, #if we were fine-tu
                            num_classes=2)
print('Model:')
print(bert_model)
```

```
BertClassifier(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
```

Training

```python
from pytorch_lightning import Trainer
from pytorch_lightning.callbacks.progress import TQDMProgressBar

# And then training is easy with our old friend PyTorch Lightning
bert_trainer = Trainer(
    accelerator="auto",
    devices=1 if torch.cuda.is_available() else None,
    max_epochs=1,
    callbacks=[TQDMProgressBar(refresh_rate=20)],
    val_check_interval = 0.2,
    )
```

Training

```
bert_trainer.fit(model=bert_model,
                 train_dataloaders=train_dataloader,
                 val_dataloaders=dev_dataloader)
```

```
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
INFO:pytorch_lightning.callbacks.model_summary:
  | Name           | Type               | Params
------------------------------------------------------
0 | bert           | BertModel          | 109 M
1 | output_layer   | Linear             | 1.5 K
2 | train_accuracy | MulticlassAccuracy | 0
3 | val_accuracy   | MulticlassAccuracy | 0
------------------------------------------------------
109 M     Trainable params
0         Non-trainable params
109 M     Total params
437.935   Total estimated model params size (MB)
Validation accuracy: tensor(0.5000, device='cuda:0')
Epoch 0: 100%  7175/7175 [10:42<00:00, 11.16it/s, loss=0.141, v_num=4]
Validation accuracy: tensor(0.9094, device='cuda:0')
Validation accuracy: tensor(0.9071, device='cuda:0')
Validation accuracy: tensor(0.9220, device='cuda:0')
Validation accuracy: tensor(0.9174, device='cuda:0')
Validation accuracy: tensor(0.9209, device='cuda:0')
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped: `max_epochs=1` reached.
Training accuracy: tensor(0.9238, device='cuda:0')
```