Contextualised Meaning Representations and BERT

LT2213 Computational Semantics V21

Nikolai Ilinykh

May 10, 2021

Outline

embed-**encode**-attend-predict

embed-**encode-attend**-predict

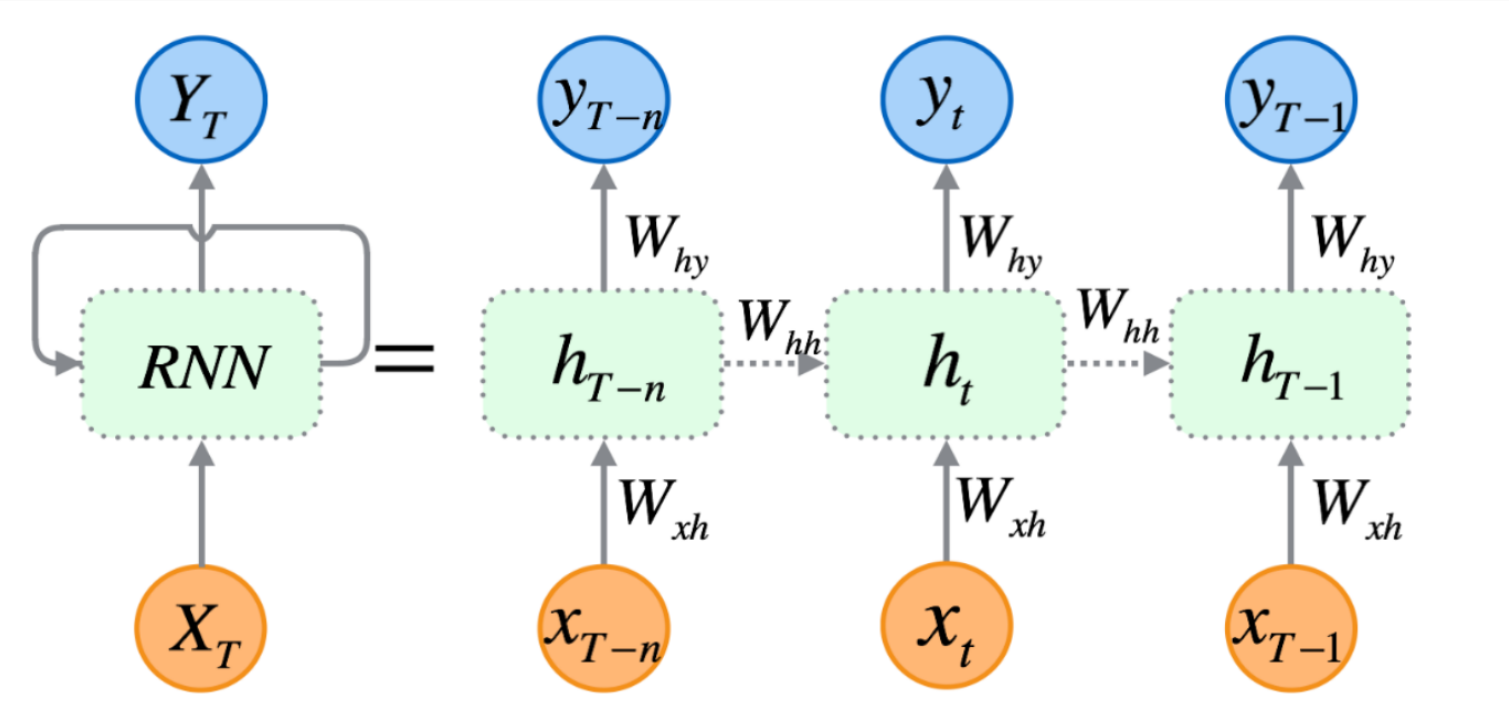
Transformers

BERT

embed-encode-attend-predict

What did we learn about RNNs/LSTMs?

- In the NLP paradigm **embed-encode-attend-predict**, RNNs/LSTMs are used to encode.
- RNNs/LSTMs are uni-directional: they take information from **the past** (previously generated words) to predict the next word (e.g., language generation task)

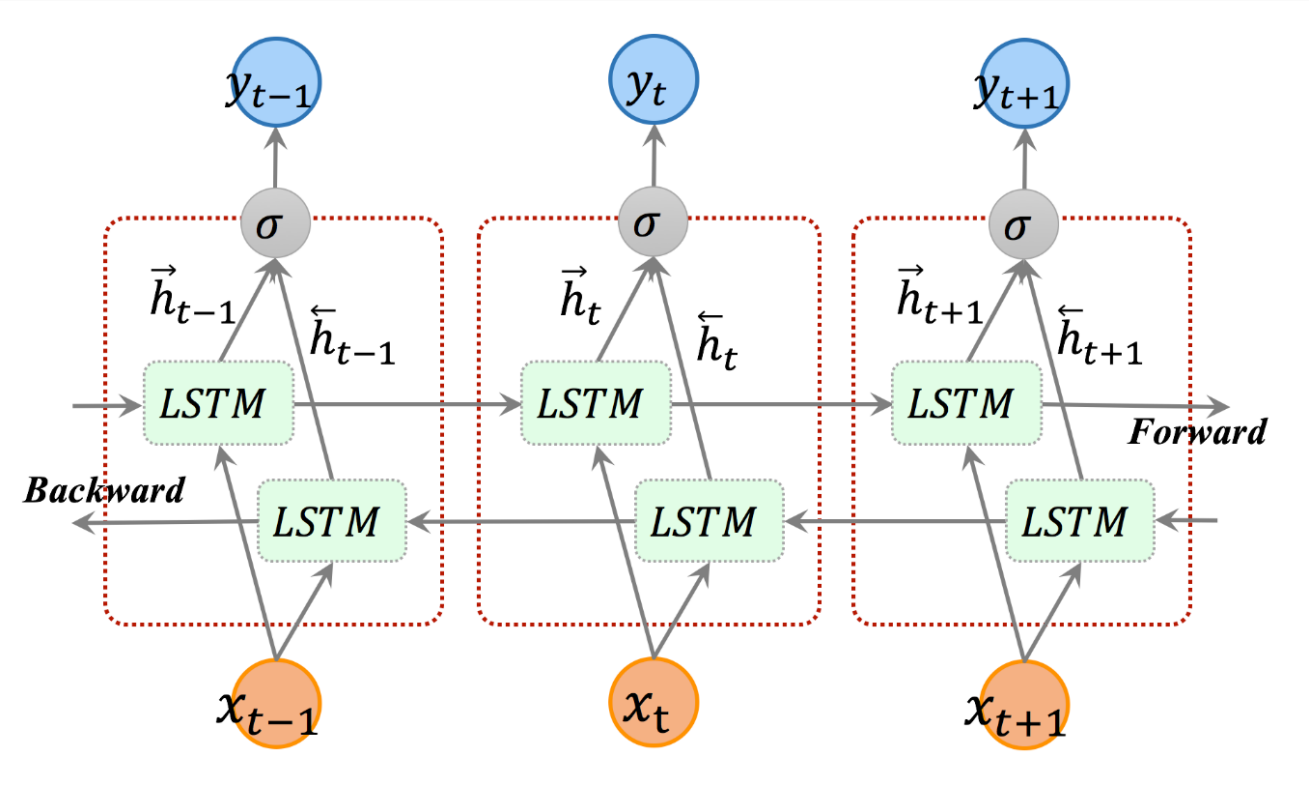


BiLSTMs vs. LSTMs

- Consider the phrase "He said, Teddy ___".
- What is the word to predict? Teddy bears? Teddy Roosevelt? Teddy Riley?
- It is difficult to predict the next word, because the context that clarifies Teddy comes later, that is, we need to look into the **future**.
- The problem is solved with Bidirectional RNNs/LSTMs: they process the sequence in both directions.

BiLSTMs are deeper and better

- Typically, there are **two separate RNNs** (forward and backward direction).
- Two hidden states are concatenated to form a single hidden state vector.
- The final hidden state is used by some sort of a decoder (e.g., textbffully connected network) and followed by **softmax**.
- Depending on the design of the network, the BiLSTM's output can be (i) the full sequence of hidden states, or (ii) the states from the last time step.



Why do we need BiLSTMs?

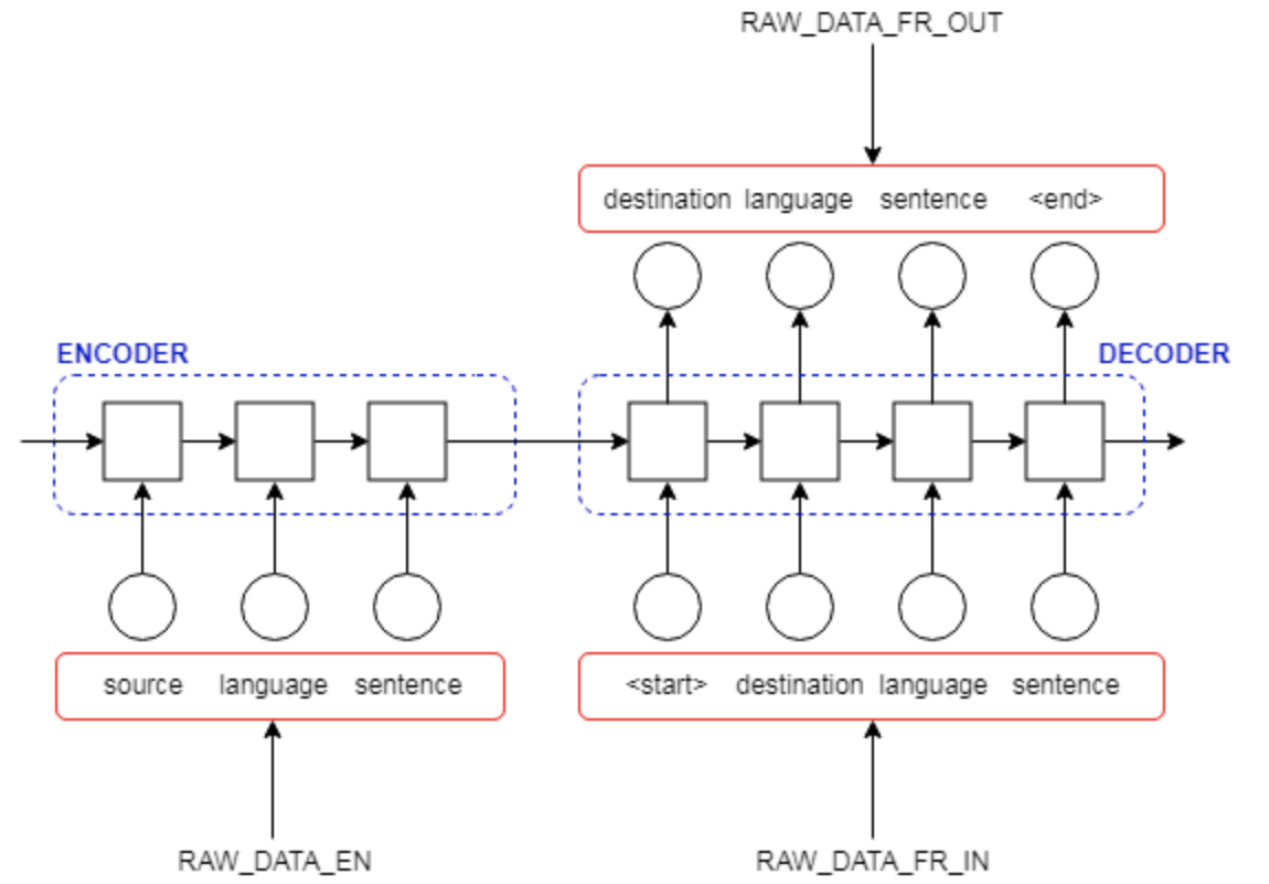
- BiLSTM are great for many NLP tasks:
  1. POS tagging
  2. Word Sense Disambiguation (WSD)
  3. Named Entity Recognition (NER)
- beyond NLP, BiLSTMs have been applied to some computer vision problems, e.g. OCR (optical character recognition)
- however, it is not natural to use BiLSTMs for language generation (which has auto-regressive, incremental nature)

BiLSTMs literature

- Schuster and Paliwal 1997: regression and classification experiments
- Graves and Schmidhuber 2005: phoneme classification in speech recognition, "humans understand sounds only after hearing future contexts"
- Wu et al. 2016: Google uses BiLSTMs for Neural Machine translation (first layer of encoder is BiLSTM)
- Peters et al. 2018: Embeddings from Language Models (ELMo)

embed-encode-attend-predict

Sutskever et al., 2014: vanilla seq2seq



Encoder

Encoder needs only sequences from the source language as inputs, its hidden states are initialised with either zeros or randomly.

```
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(input_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)

    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)
        output = embedded
        output, hidden = self.gru(output, hidden)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

Decoder

Decoder is **a language model**, which needs to predict the next word (output a probability)

```
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)
        output = self.softmax(self.out(output[0]))
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

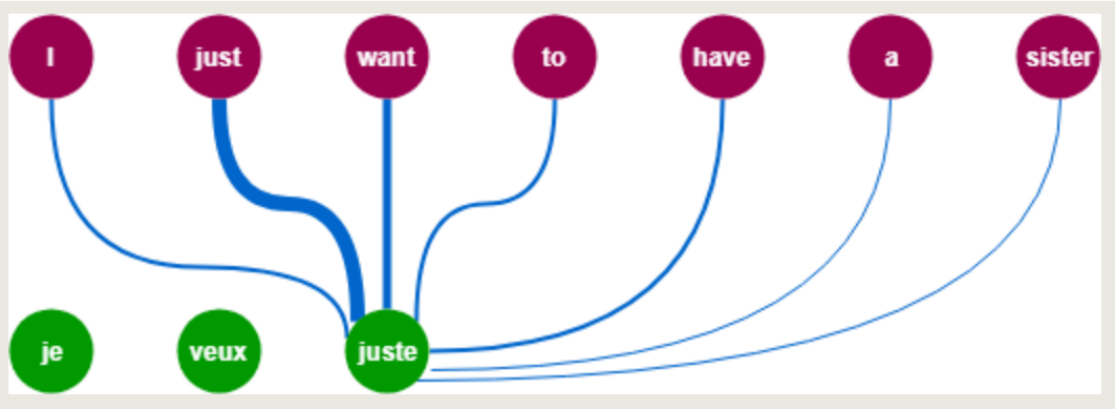
What is the problem here?

- let's translate from English to French: from
  "I just want to have a sister" to "Je veux juste avoir une soeur"

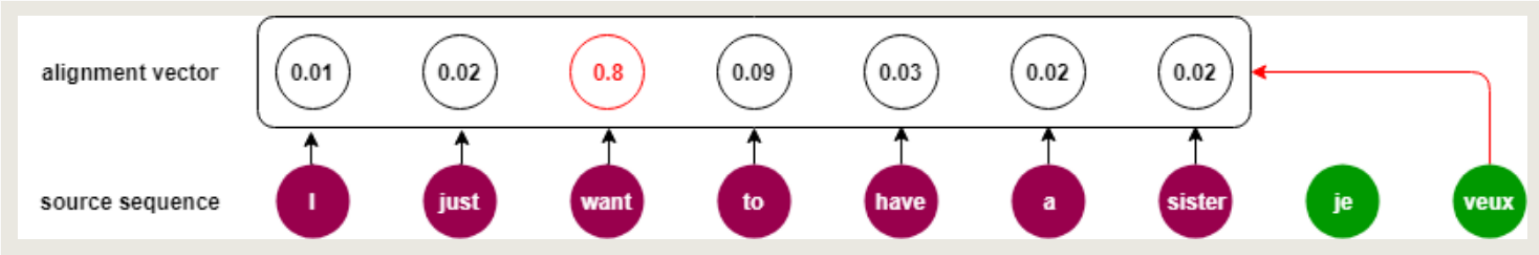
- problem 1: encoder's state is passed to the first node of the decoder
- problem 2: differences in syntax structure could give a tough time to figure things out
- How can we solve these problems? We want our decoder to have access to all elements in the encoder's output.
- This is called **attention mechanism**.

Bahdanau et al., 2015: attention for machine translation

- attention allows decoder to weight different parts of the encoder for **every time step** (every generated output).

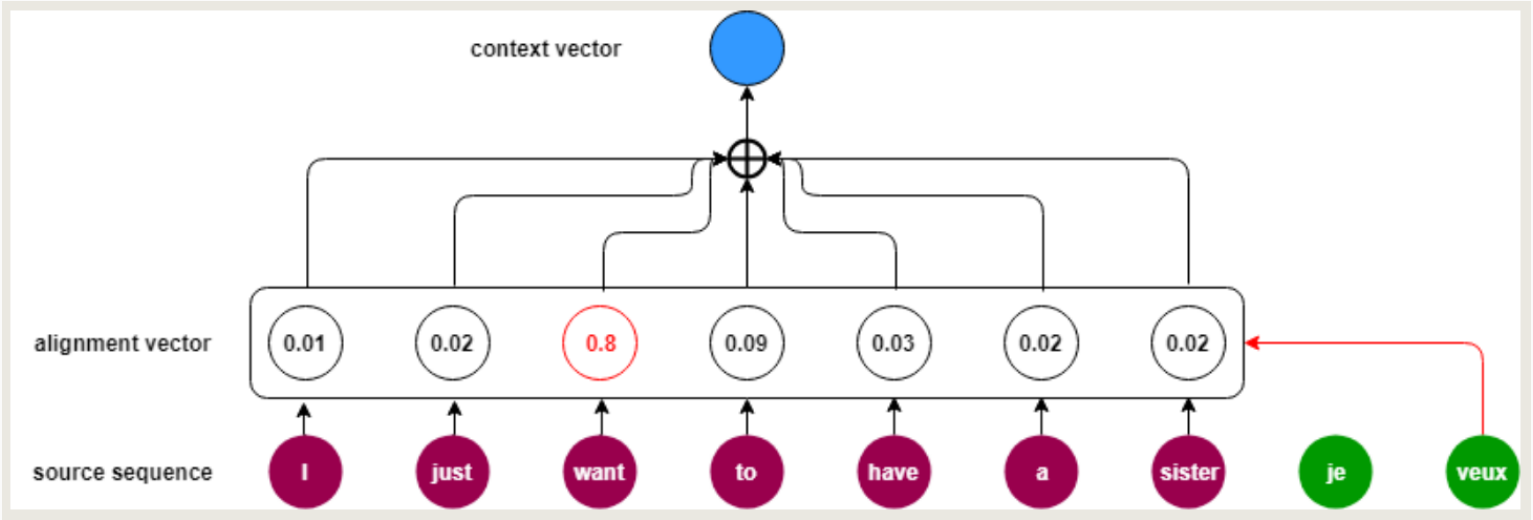


- **the alignment vector** is a vector that has the same length as the source sequence and is computed at **every time step of the decoder**. Each of its values is the score (or the probability) of the corresponding word within the source sequence



The context vector in attention

- **the context vector** is the weighted average of the encoder's output. We get it by computing the dot product of the alignment vector and the encoder's output.

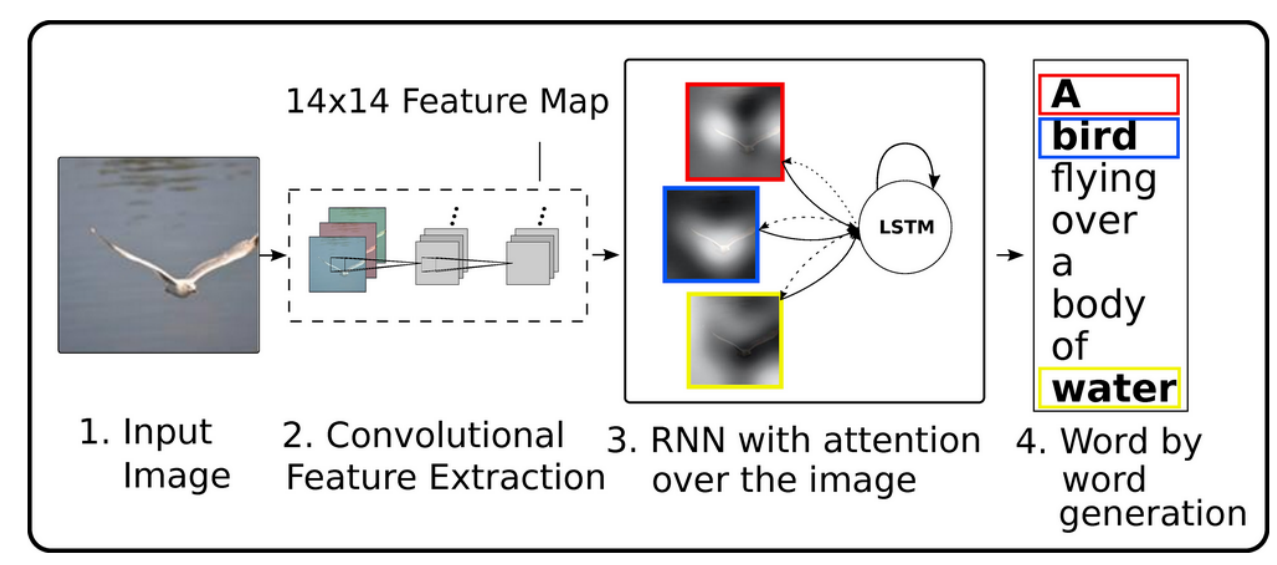


Attention in code

```
class AttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, dropout_p=0.1, max_length=MAX_LENGTH):
        super(AttnDecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.dropout_p = dropout_p
        self.max_length = max_length

        self.embedding = nn.Embedding(self.output_size, self.hidden_size)
        self.attn = nn.Linear(self.hidden_size * 2, self.max_length)
        self.attn_combine = nn.Linear(self.hidden_size * 2, self.hidden_size)
        self.dropout = nn.Dropout(self.dropout_p)
        self.gru = nn.GRU(self.hidden_size, self.hidden_size)
        self.out = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input, hidden, encoder_outputs):
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded)

        attn_weights = F.softmax(
            self.attn(torch.cat((embedded[0], hidden[0]), 1)), dim=1)
        attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                                 encoder_outputs.unsqueeze(0))

        output = torch.cat((embedded[0], attn_applied[0]), 1)
        output = self.attn_combine(output).unsqueeze(0)

        output = F.relu(output)
        output, hidden = self.gru(output, hidden)

        output = F.log_softmax(self.out(output[0]), dim=1)
        return output, hidden, attn_weights
```

Xu et al. 2016: attention in image captioning

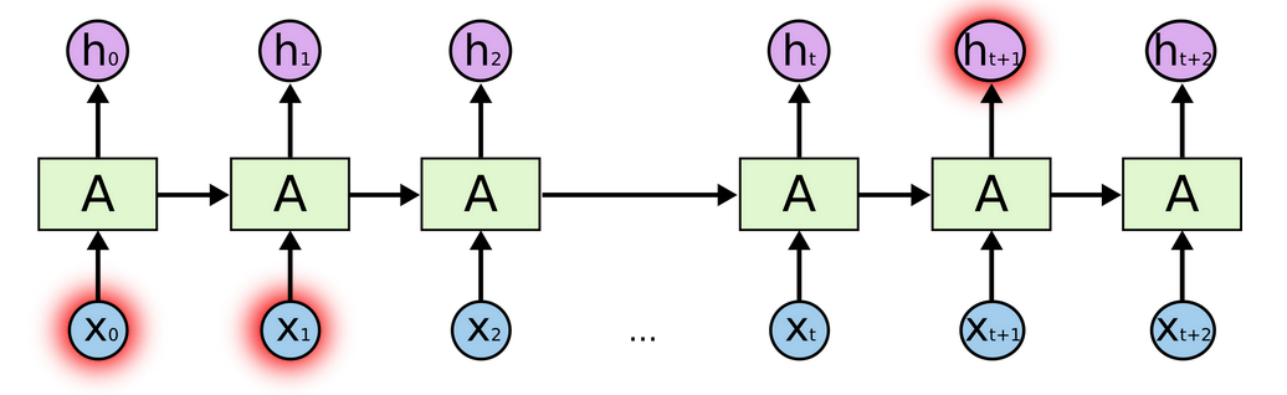
- attention has shown to be beneficial for many NLP-related tasks, e.g. image captioning, describing an image through natural language



Transformers

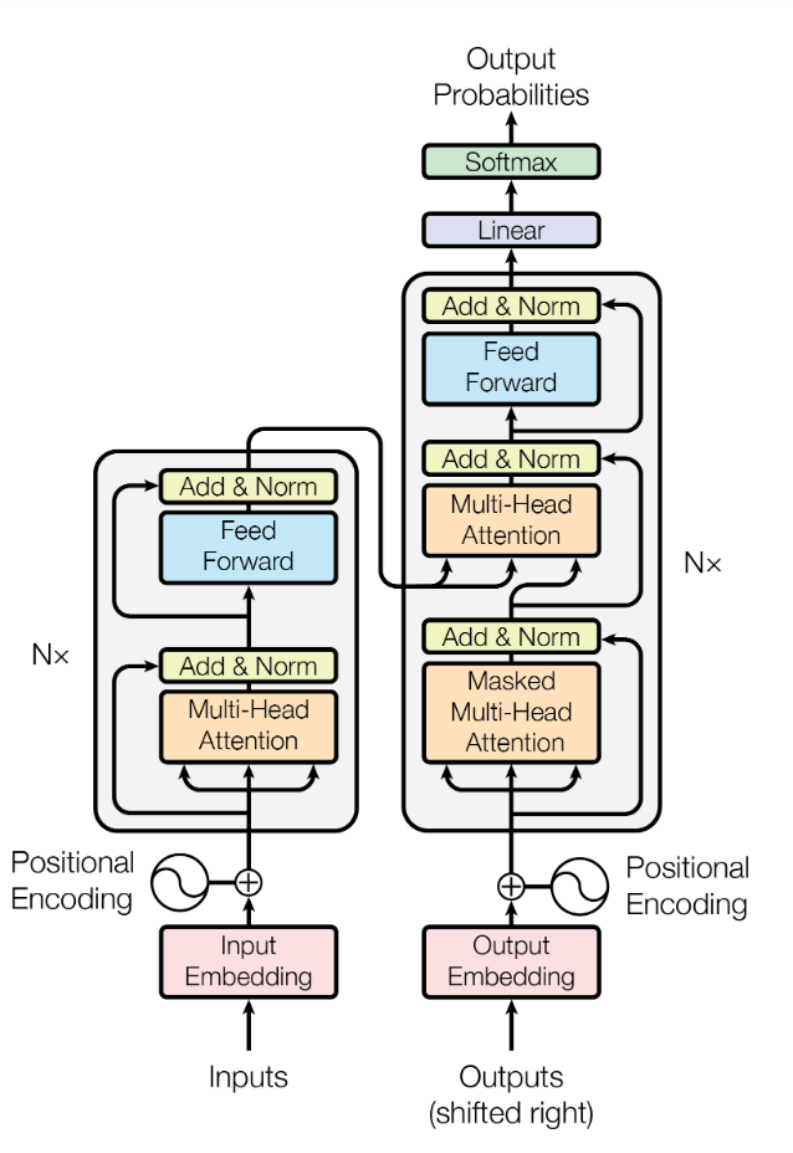
Problems with RNNs and LSTMs

- Sequential processing: sentences are processed word by word
- The encoding of a specific word is important for the next step the most, and strongly affects the next word, loosing its influence after a few time steps, e.g. there is a risk to forget past information.
- "I grew up in Sweden. ... I speak fluent *Swedish*"

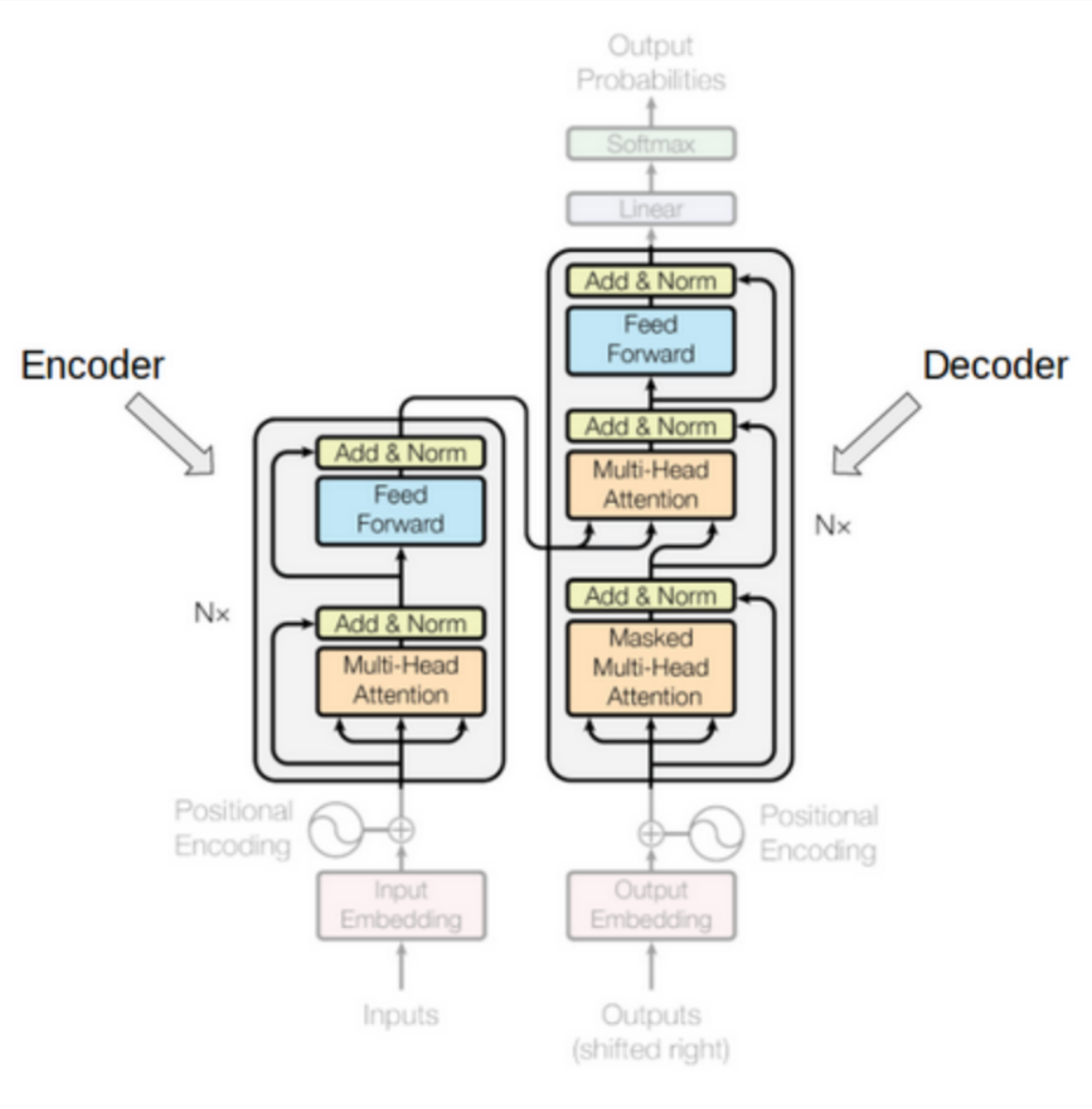


- Past information is kept through past hidden states *only*. BiLSTM is a hack to the problem, not a real solution to capture very long dependencies.

Transformer Architecture

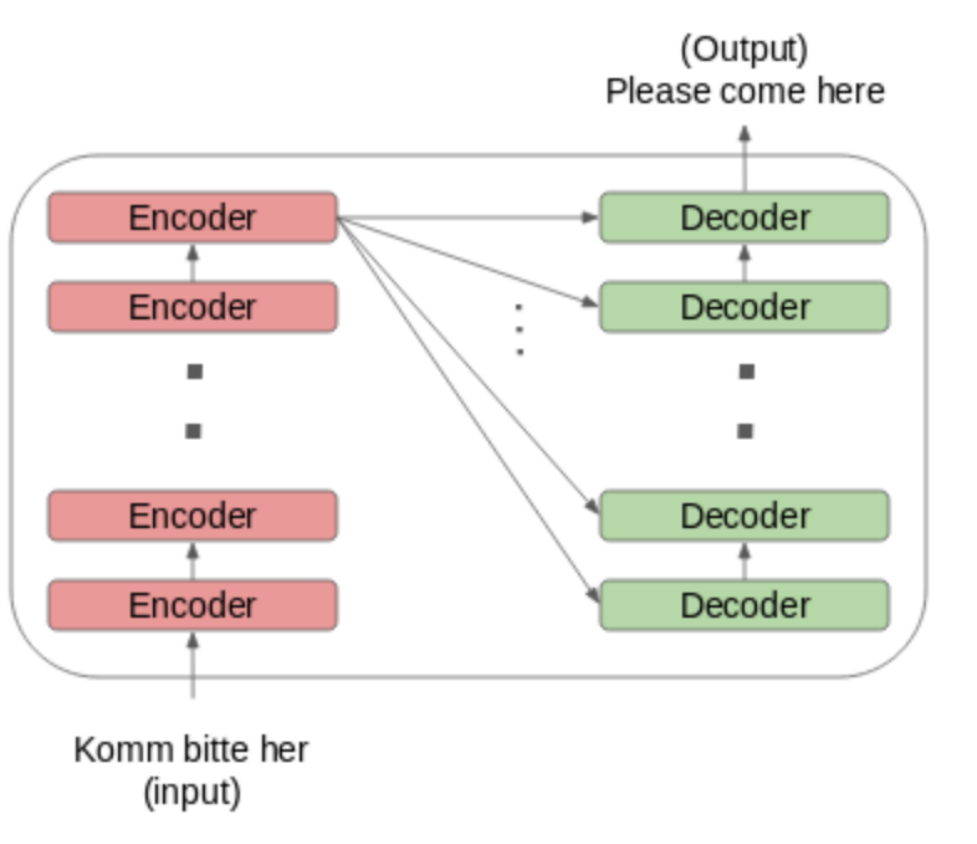


Transformer Architecture



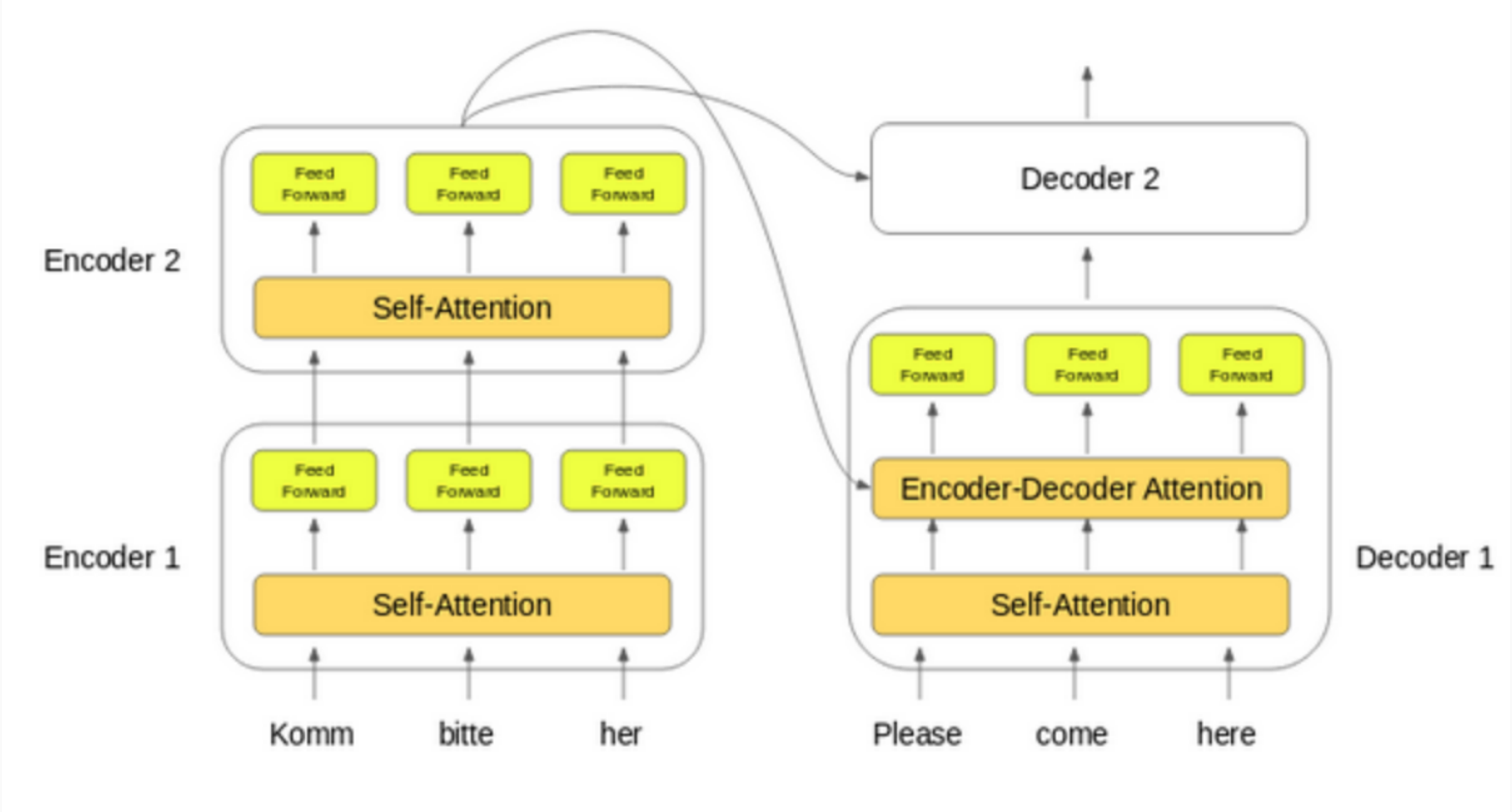
Transformer Architecture

The encoder and decoder blocks are actually multiple identical encoders and decoders stacked on top of each other. Both the encoder stack and the decoder stack have the same number of units.



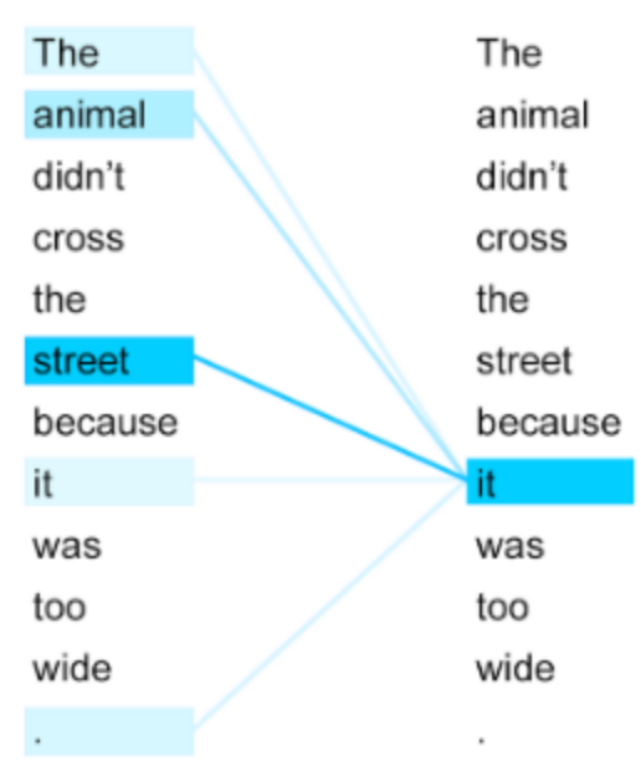
Transformer Architecture

Note that in addition to the self-attention and feed-forward layers, the decoders also have one more layer of Encoder-Decoder Attention layer. This helps the decoder focus on the appropriate parts of the input sequence.



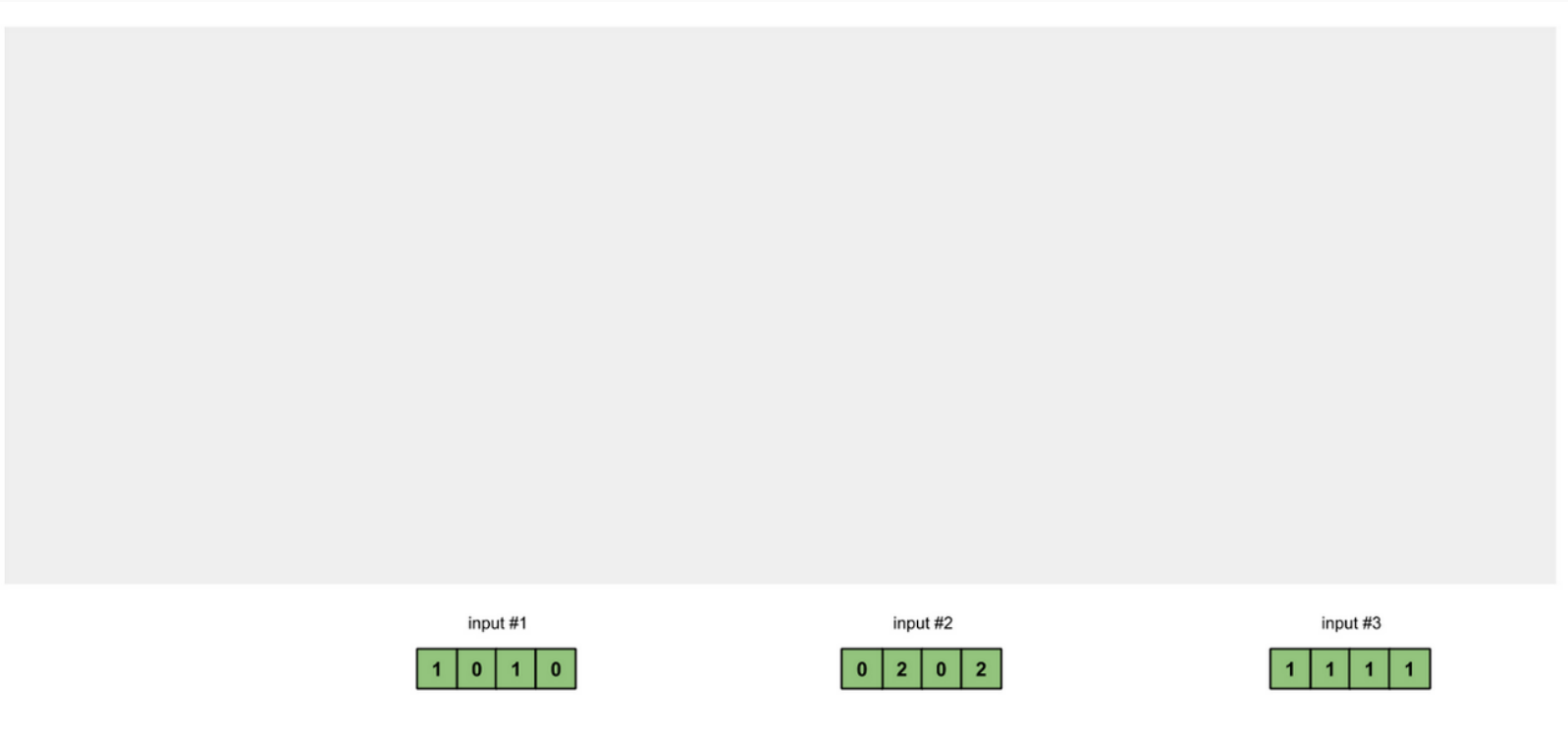
Transformer Architecture

Self-attention, sometimes called intra-attention, is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence.



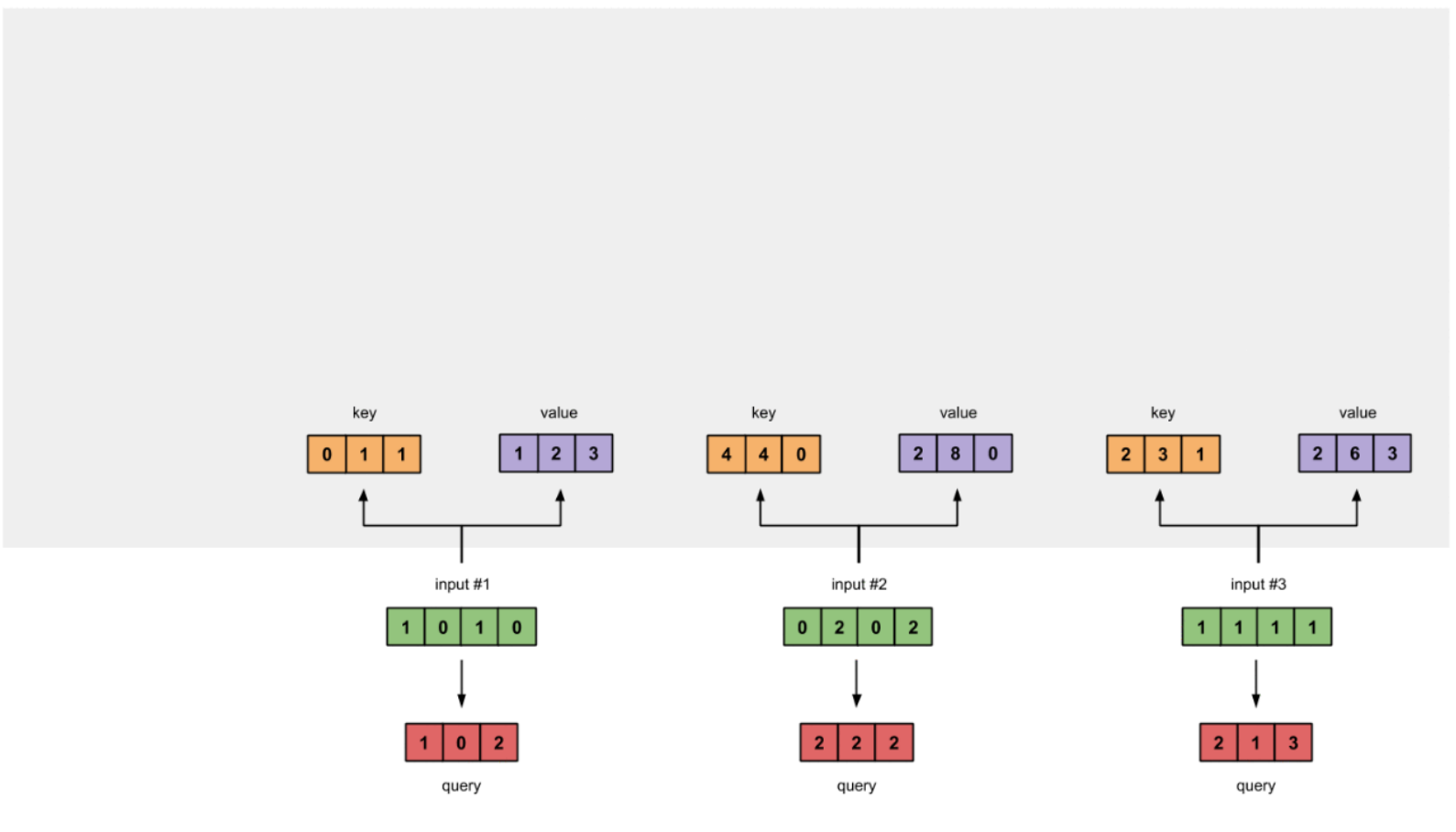
Self-Attention: prepare inputs

We start with 3 inputs (e.g., 3 words) each with the same dimension (=4).



Self-Attention: keys, queries, values

Every input must have three representations (weights): keys, queries, values. Let's say, their dimension is 3 (normally, bigger). Every input has a dimension of 4, therefore, the weights must be of shape 4 x 3.



Self-Attention: keys, queries, values

How do we get these representations?

Each input is multiplied with a set of weights for keys, a set of weights for queries, a set of weights for values. We randomly initialise these weights. For our example, let's say we initialise keys with the following numbers:

```
[[0, 0, 1],
 [1, 1, 0],
 [0, 1, 0],
 [1, 1, 0]]
```

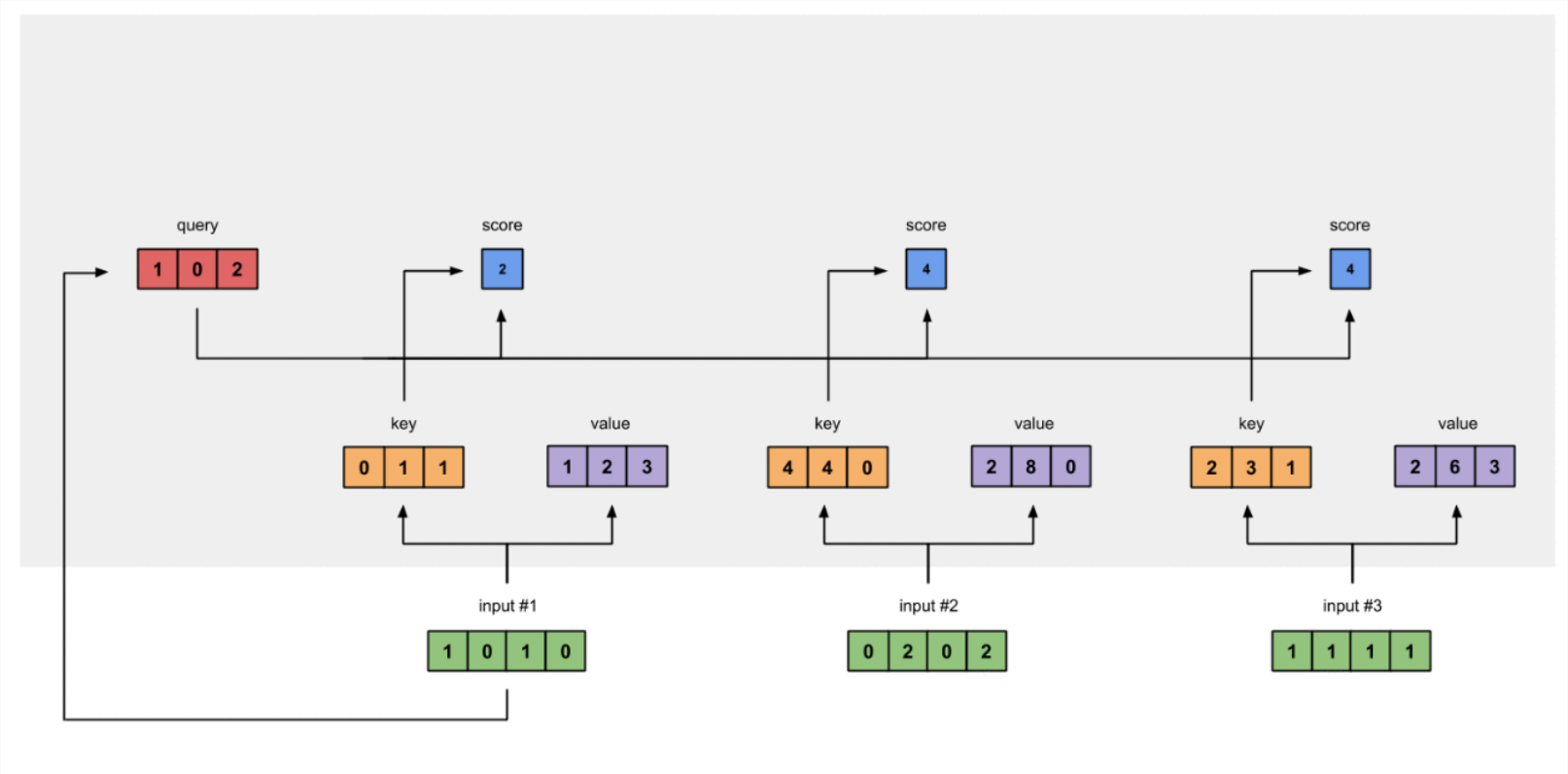
Then, we obtain a key representations for our first input.

```
                  [0, 0, 1]
[1, 0, 1, 0] x    [1, 1, 0] = [0, 1, 1]
                  [0, 1, 0]
                  [1, 1, 0]
```

We do it with keys, values, queries for each input (!)

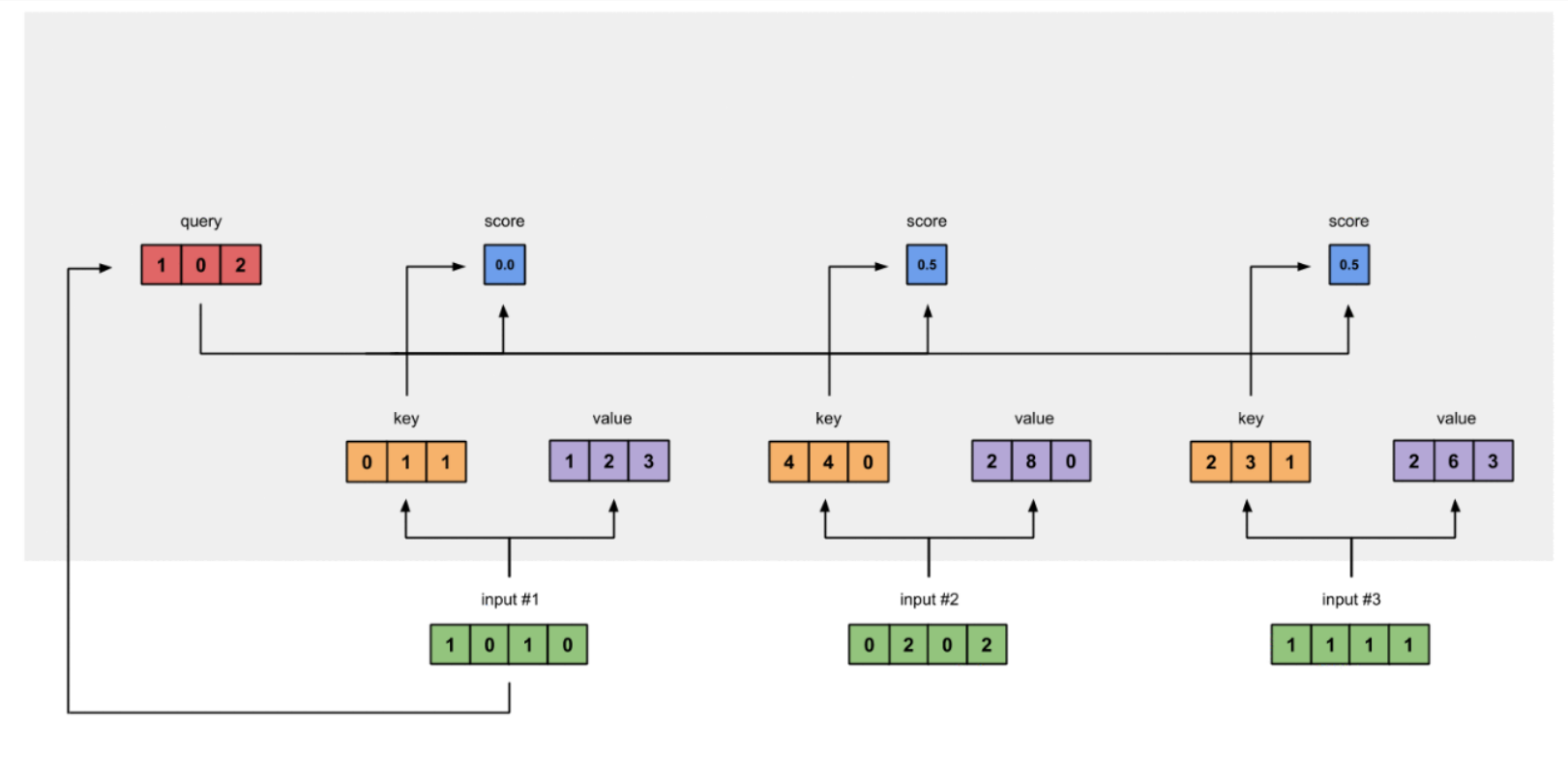
Self-Attention: calculate attention scores for each input

We take a dot product between input's query and keys. We have 3 key representations, therefore, we get 3 attention scores.



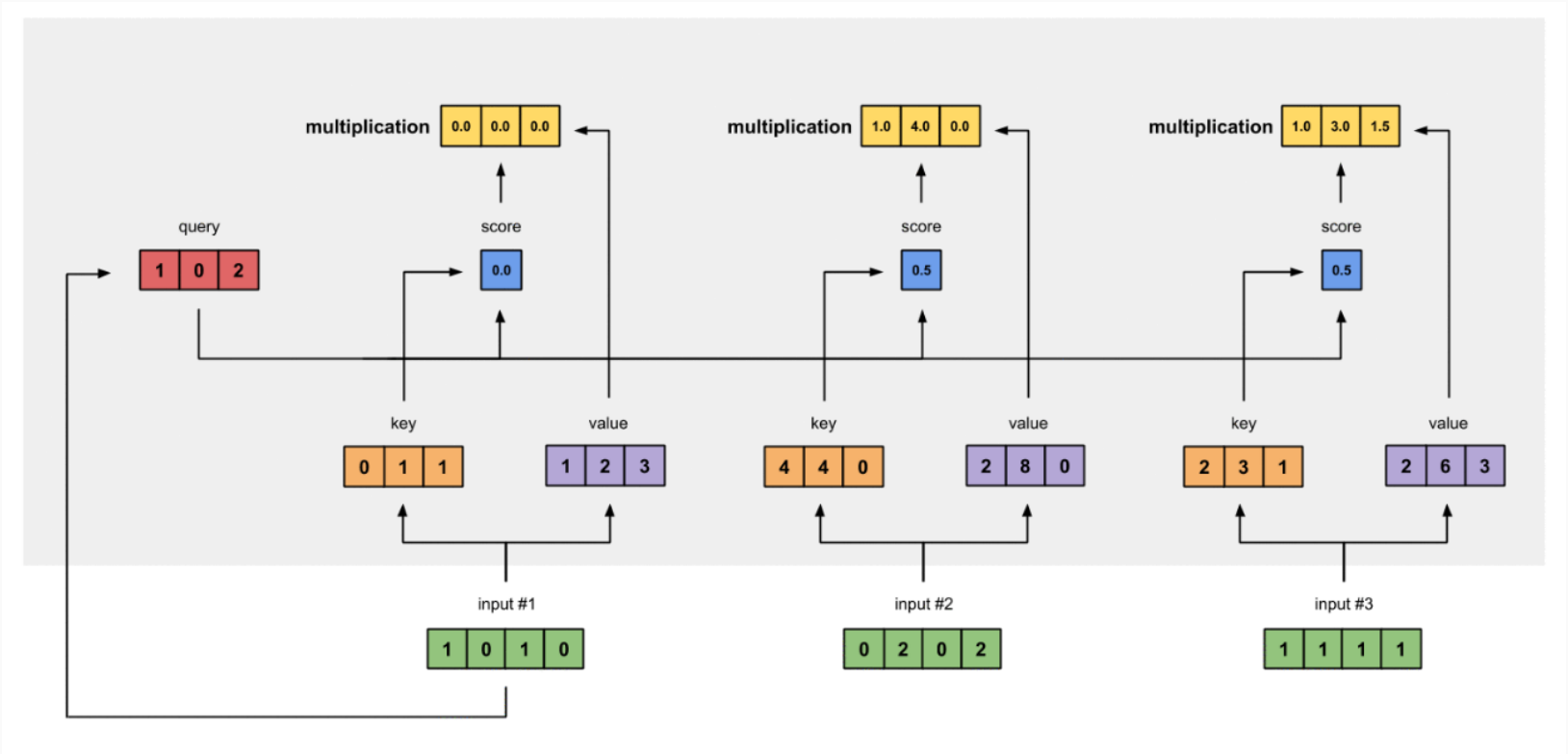
Self-Attention: softmax

Attention scores are not probabilities, we use softmax to put them in range 0-1.



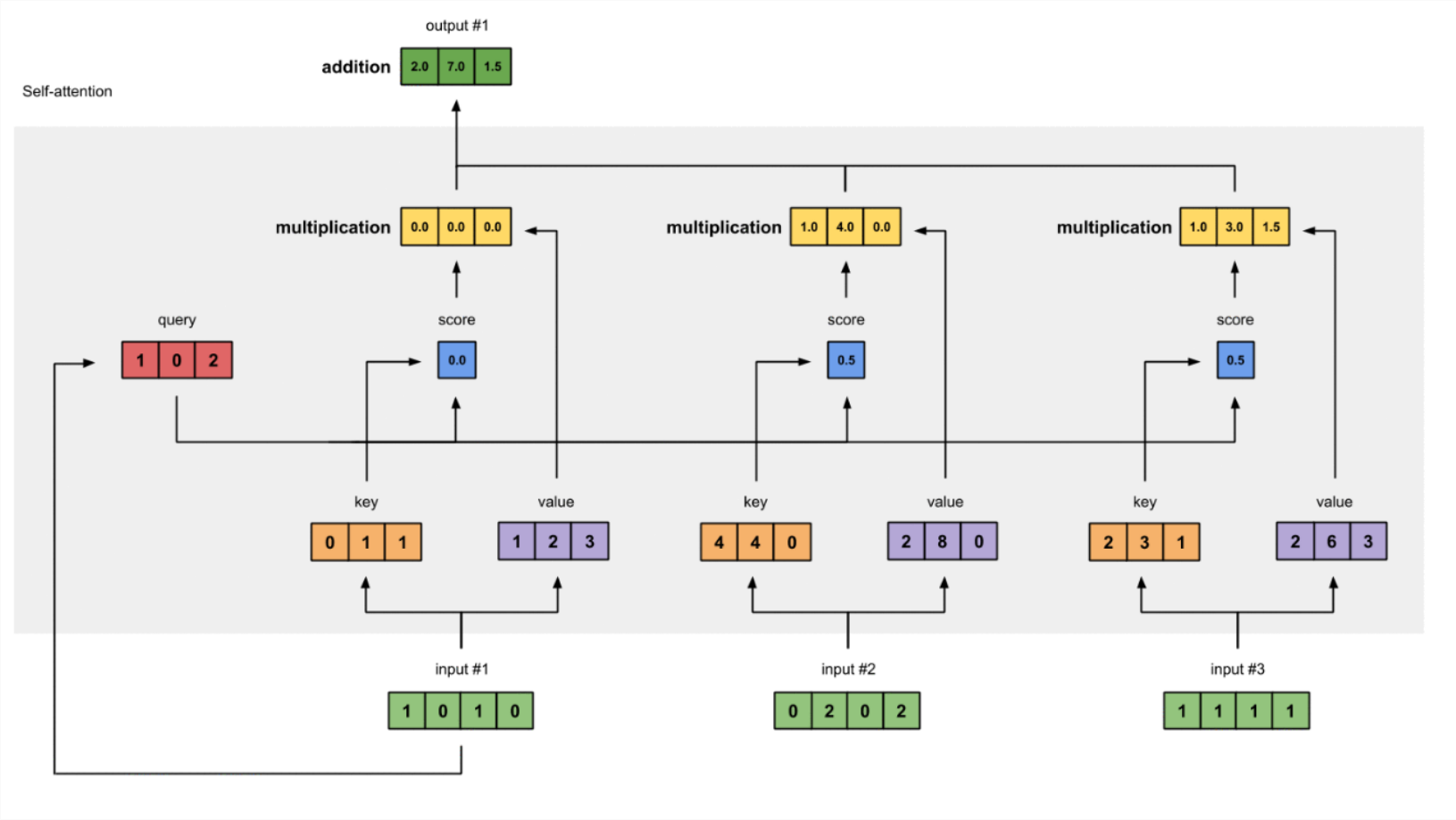
Self-Attention: multiply with values

Next, we multiply attention probability with the corresponding value representation.



Self-Attention: summing weight values element-wise

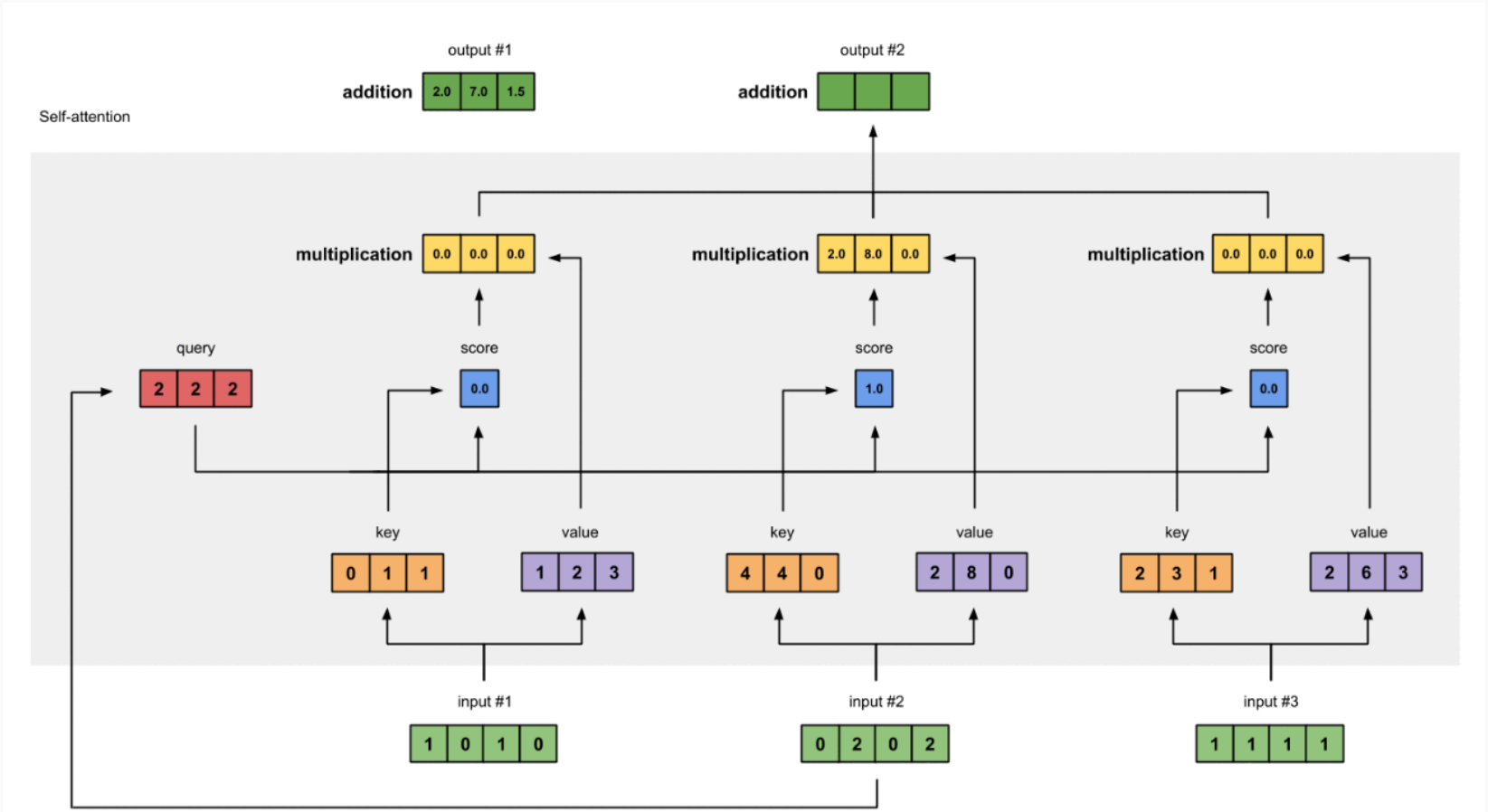
Finally, we sum *all* weighted values to get our first output. This output is constructed from the query representation 1 interacting with all other keys, including itself.



Self-Attention: repeat for all other outputs

We repeat all actions for other inputs as well.

The main idea is to learn good weights for keys, queries and values for our input sequence, which are constructed based on the other elements in this sequence.



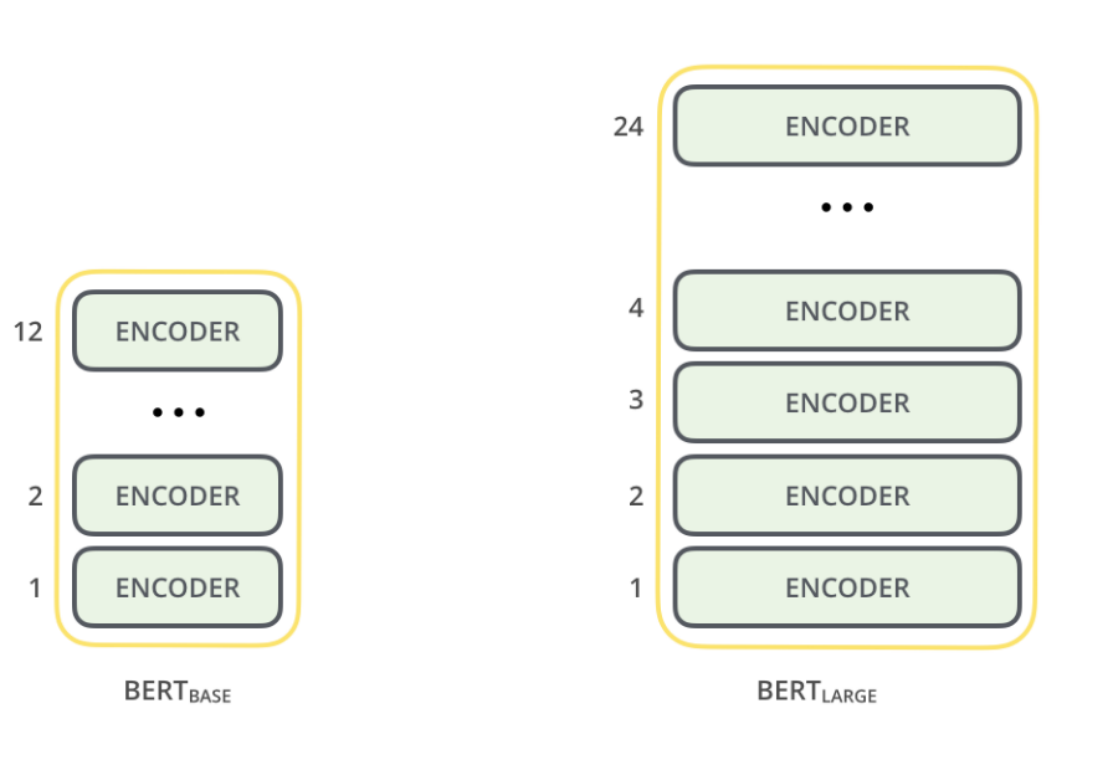
Vaswani et al. 2017: transformers and self-attention

- Transformers are better than all other architectures because:
  1. they **process sentences as a whole** (past and future), no recursion, allowing us to avoid making extra 2x computations (as in BiLSTMs)
  2. **self-attention** allows us to have direct access to all other steps in the sequence, drastically reducing information loss
  3. **positional encodings** let us learn about position of a token in a sentence, capturing more information about relations between words
  4. **multi-head attention** allows us to have many keys, queries and values, learning different aspects of meaning in text.

BERT

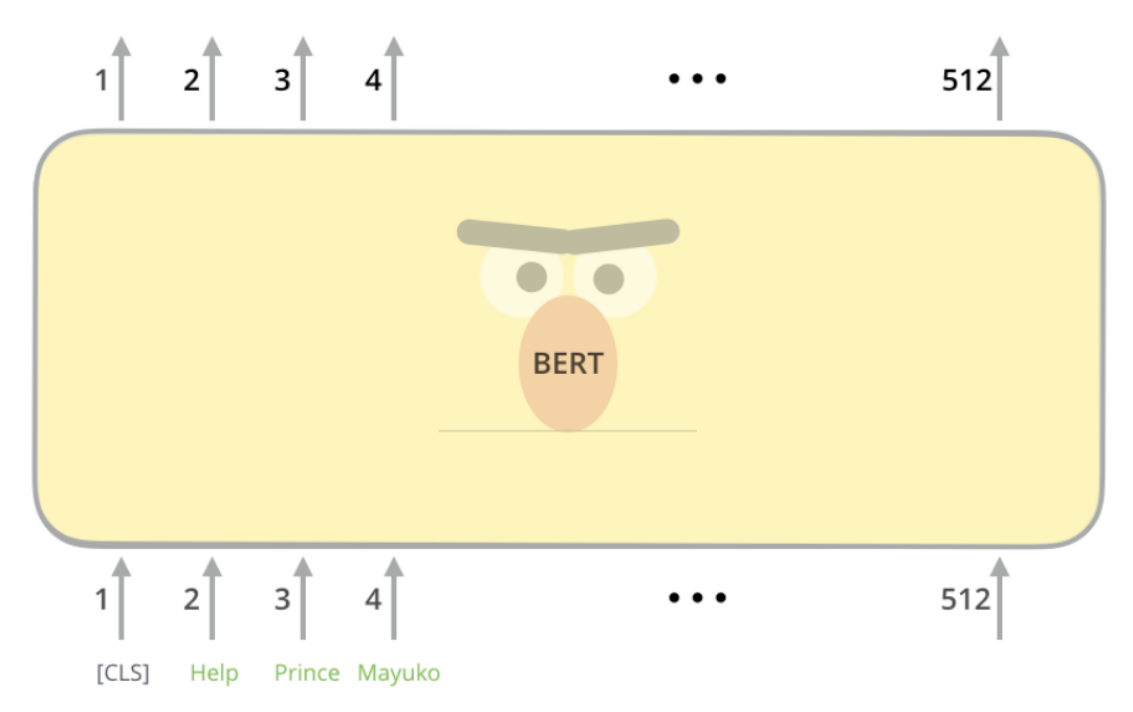
Devlin et al. 2018: deep bidirectional transformer

- BERT is basically a trained Transformer Encoder stack.



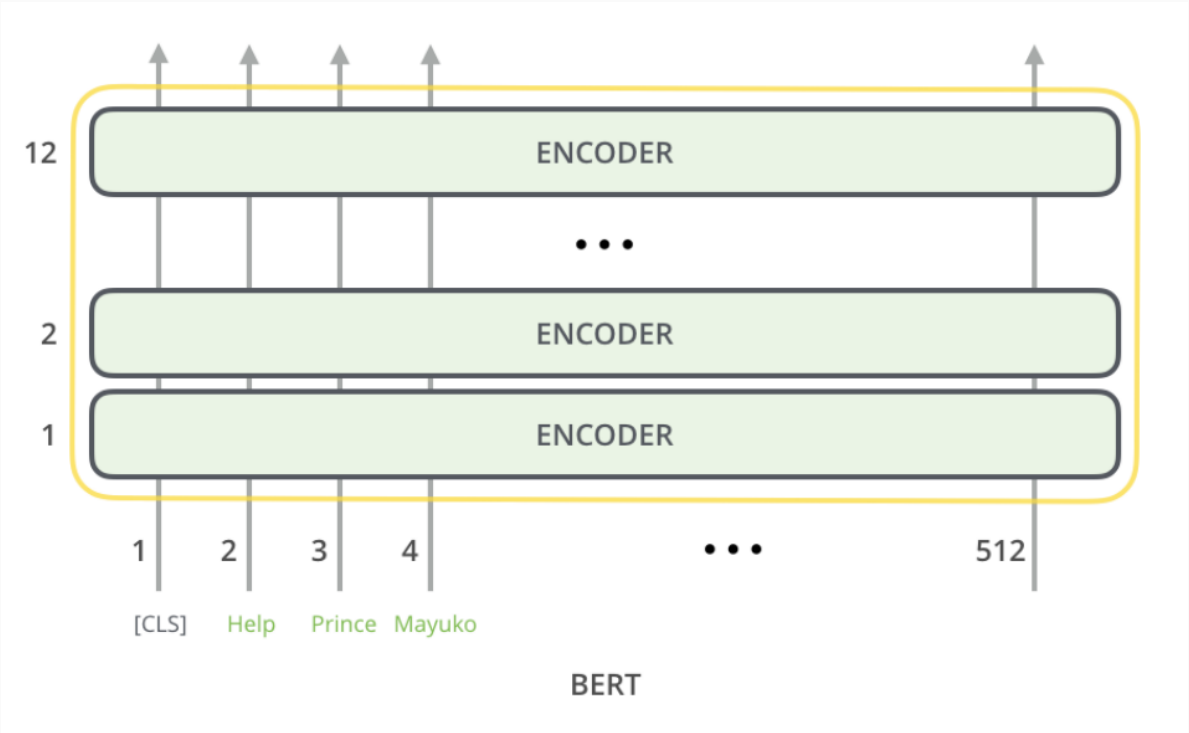
BERT

Input: Embedding of the CLS token and your whole sequence



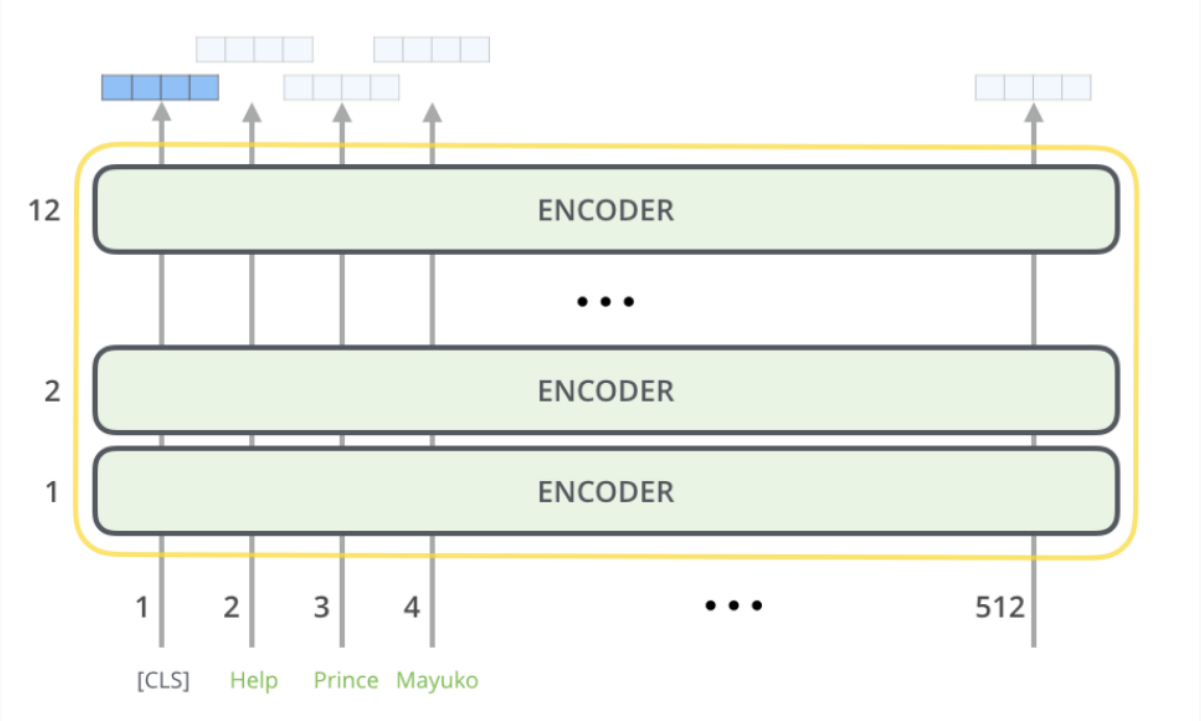
BERT

Structure: the stack of encoders, consisting of self-attention and feed-forward layers



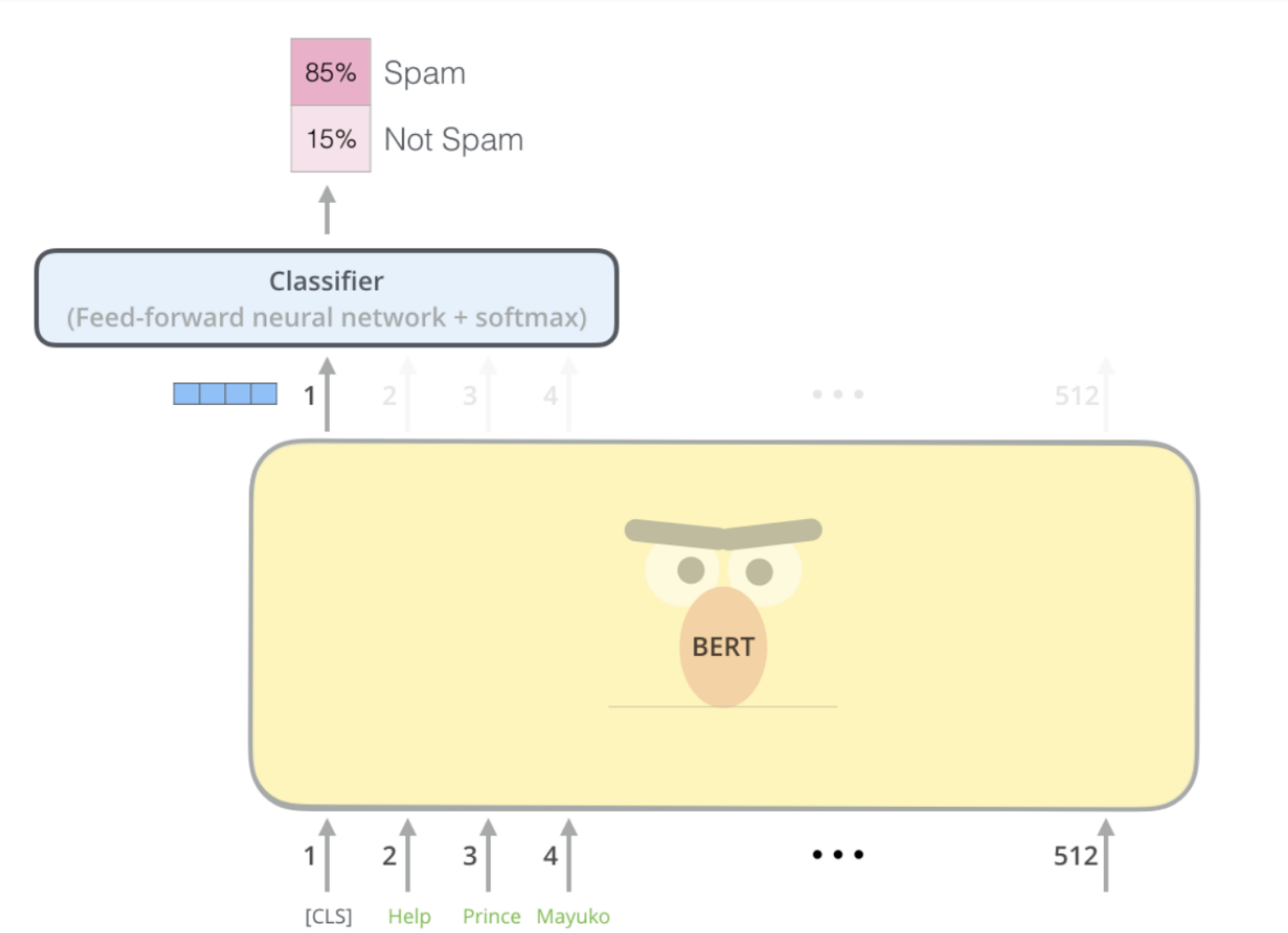
BERT

Output: a sequence of vectors for each input item, but we are using the CLS one for our task



BERT

Task: for example, spam classification



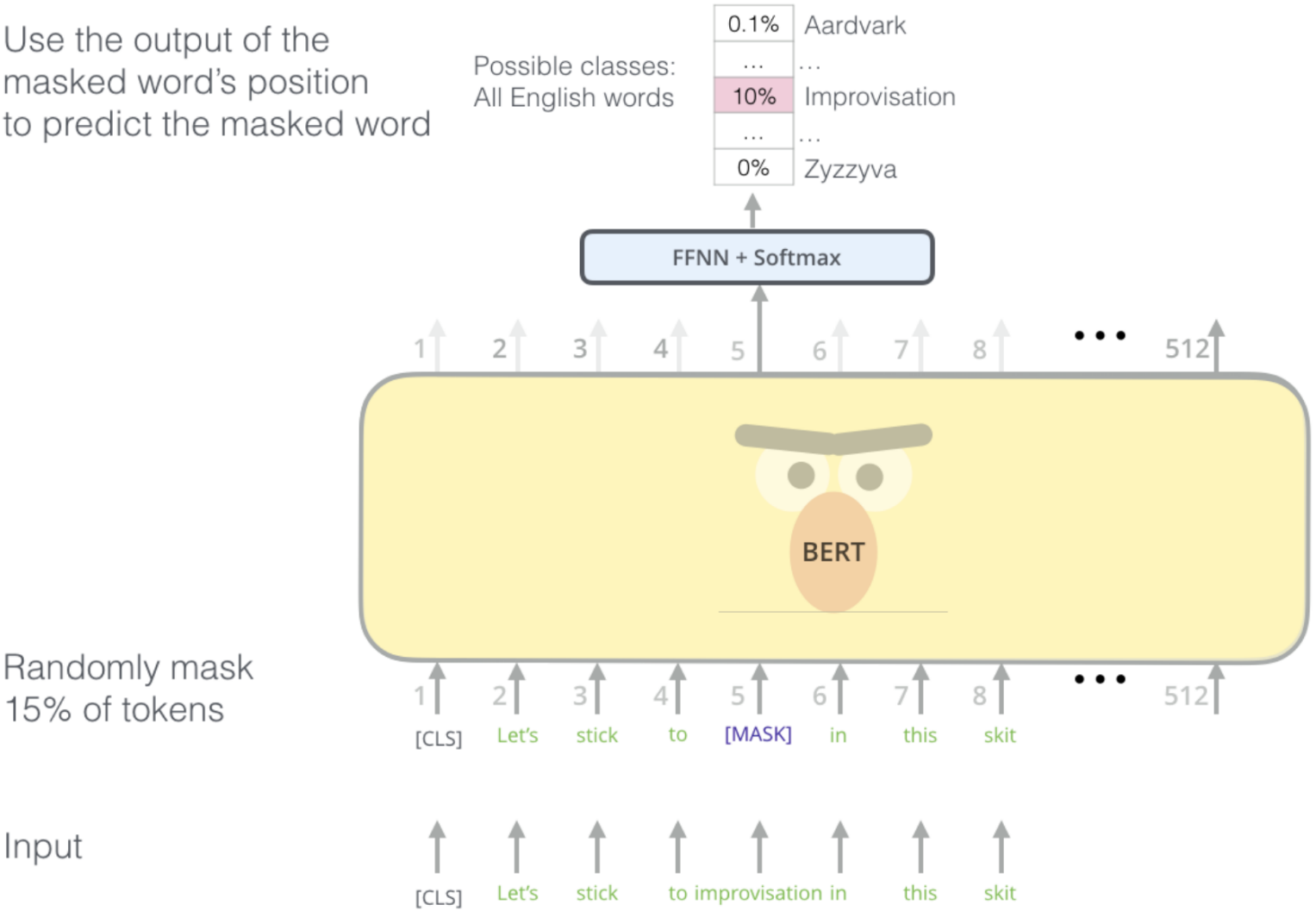
What is so special about BERT?

- Since BERT is the Transformer's Encoder, it reads the entire sequence at once. It is considered bidirectional, but we could say it is non-directional. BERT captures a deeper sense of language context and flow than single-direction language models or combined left-to-right and right-to-left training.
- But how do we train BERT, a **bidirectional** model, given that transformers are normally trained for **unidirectional** task of predicting the next word?

Training strategies for BERT

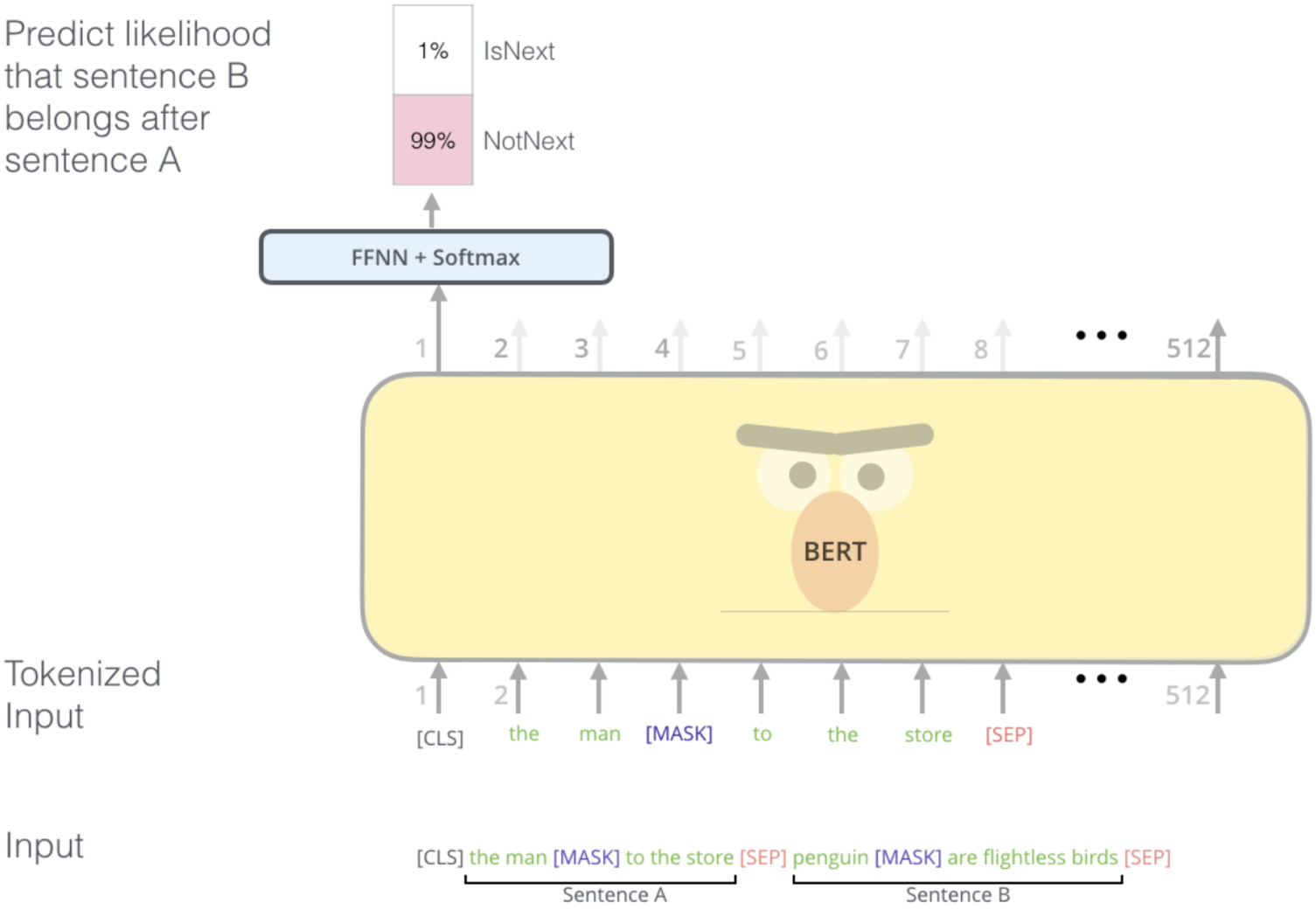
- **Masked Language Modelling**: 15% of the words in the input sequence are replaces with a [MASK] token. The task is to predict original values of the masked words, based on the context provided by non-masked words. We learn very deep word representations based on this objective.
- **Next Sentence Prediction**: based on two sentences, predict whether the second sentence in the pair is likely to be the sentence that follows the first one. We learn to better handle relationships between multiple sentences (e.g. random sentence would be disconnected from the first sentence).

Masked Language Modelling



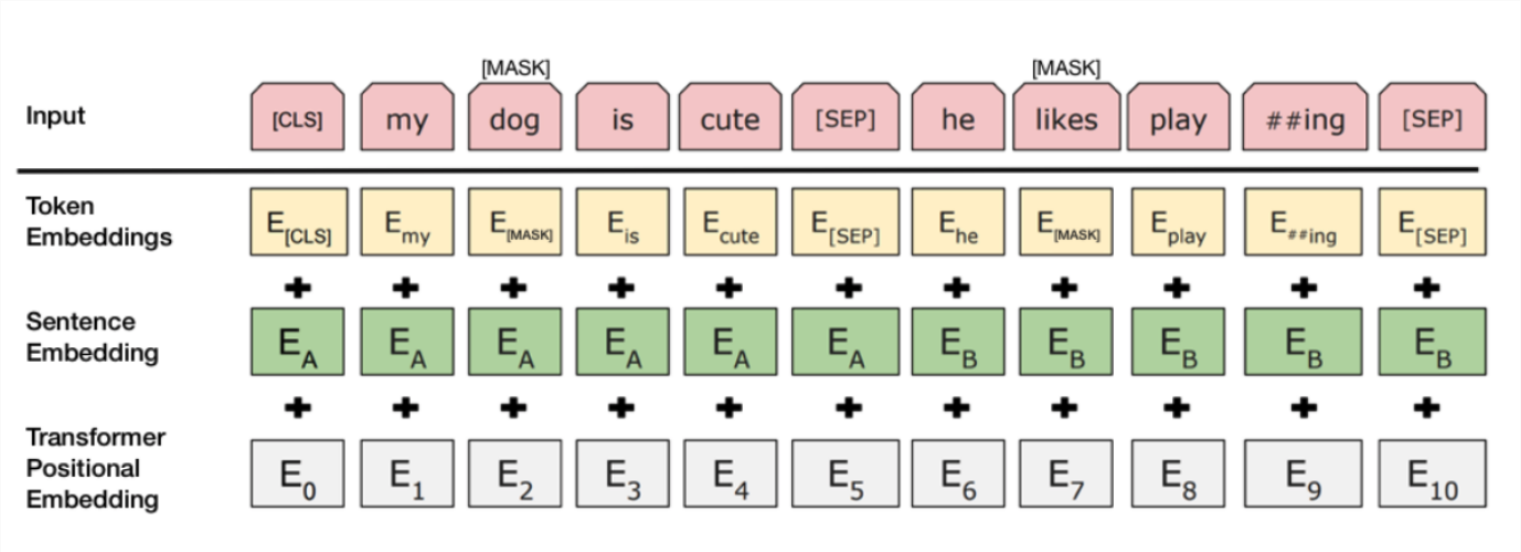
BERT's clever language modeling task masks 15% of words in the input and asks the model to predict the missing word.

Next Sentence Prediction I



The second task BERT is pre-trained on is a two-sentence classification task. The tokenization is oversimplified in this graphic as BERT actually uses WordPieces as tokens rather than words --- so some words are broken down into smaller chunks.

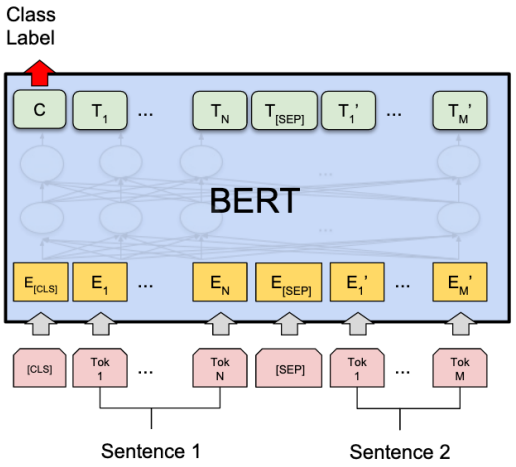
Next Sentence Prediction: input



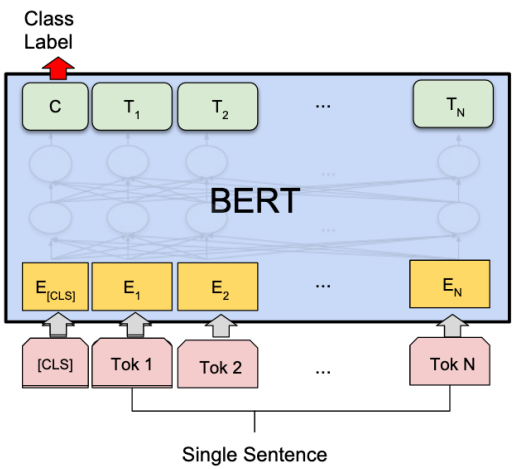
BERT: fine-tuning

- we have talked about training BERT, but, in fact, we described tasks that BERT has been *pre-trained* on
- next, we need to *fine-tune* BERT on the downstream task
- general intuition: pre-train on fundamental tasks/a lot of data to capture general patterns, then fine-tune on specific data and specific task to change model's parameters from more general to more specific for the task
- that is, we can take pre-trained BERT, add a small layer to it and fine-tune (train) this layer for our task at hand; we can also freeze/unfreeze BERT's parameters

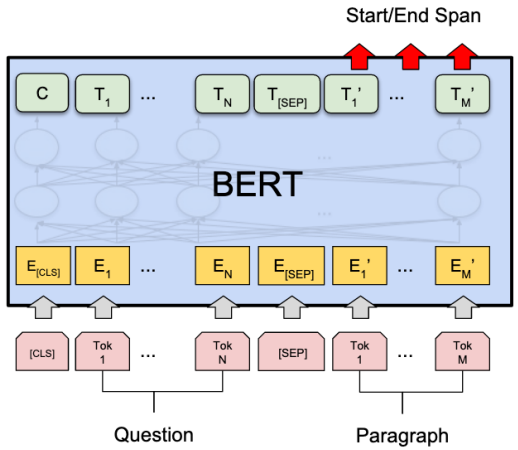
BERT for downstream tasks



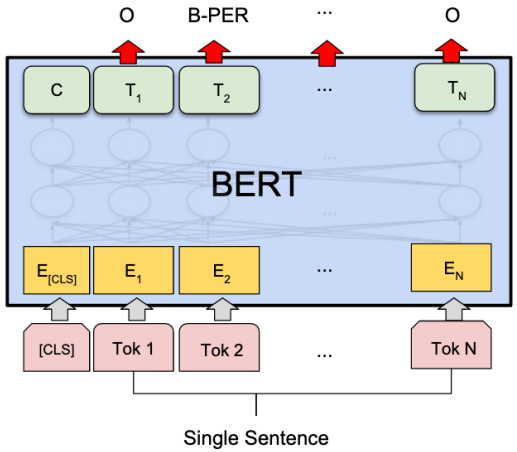
(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG



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



(c) Question Answering Tasks: SQuAD v1.1



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