Lecture 14: Transformers (Part 4)

Data C182 (Fall 2024). Week 08. Tuesday Oct 15th, 2024

Speaker: Eric Kim

Announcements

- HW02 ("RNNS") out! Due: Sun Oct 27th 11:59PM PST
  - Please start early!

Announcements

- Midterm is coming up! [link]
  - Tuesday, October 22th 2024, 6:30 PM - 8 PM.
  - Location: ~50% in 10 Evans, ~50% in Physics 1
    - We'll send exam room assignments to students shortly
  - If you're unable to make this time, please contact us ASAP (make a private Ed post)
  - Midterm will cover everything from:
    - Lectures, discussions, HW01+HW02
  - In-person, paper + pencil exam.
  - **DSP**: if you need exam accommodations, please contact us ASAP (private post on Ed)

DSP: Midterms

- DSP students with exam accommodations: you should have received an email regarding scheduling your midterm exam. Please fill out the "V2" google form ASAP so that we can schedule your midterm

Today's lecture

- (Correction on Lecture 12: MHA)
- Transformers (Part 3!)
- Encoders: classification techniques
- Decoders
  - Cross attention
  - Masked self-attention ("causal self attention")
- Sequence-to-sequence tasks
  - Ex: Machine translation, text generation

(for fun)

- Deep in the pytorch implementation for \`torch.nn.functional.multi_head_attention_forward()\`, there is this funny comment [link]:
    - (open-source can be fun!)

pytorch / torch / nn / functional.py

Code Blame 6300 lines (5389 loc) · 231 KB

```python
def multi_head_attention_forward(
    # adjust dropout probability
    if not training:
        dropout_p = 0.0

    #
    # (deep breath) calculate attention and out projection
    #

    if need_weights:
        B, Nt, E = q.shape
        q_scaled = q * math.sqrt(1.0 / float(E))

        assert not (
            is_causal and attn_mask is None
        ), "FIXME: is_causal not implemented for need_weights"

        if attn_mask is not None:
            attn_output_weights = torch.baddbmm(
                attn_mask, q_scaled, k.transpose(-2, -1)
            )
```

(Correction from Lecture 12) (slide pdfs on website is updated)

Neat animation from: https://towardsdatascience.com/illustrated-guide-to-transformers-step-by-step-explanation-f74876522bc0

MHA (v1.5): multiple heads + split

In practice: to reduce computation costs, rather than have each self-attention module operate on the full embedding \`d\`, we **divide up the embeddings into \`h\` chunks.**

Example: for d=16 and h=2 heads,
Head0: work on first 8 embed dims: X[:, :8]
Head1: work on last 8 embed dims: X[:, 8:]

$$\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{q})}$$
$$\mathbf{K}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{k})}$$
$$\mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(\mathrm{v})}$$

X_h shape=[seq_len, d]

Q_h, K_h, V_h
Shape=[seq_len, d_h]

W_h^q, W_h^k, V_h^v
Shape=[d, d_h]

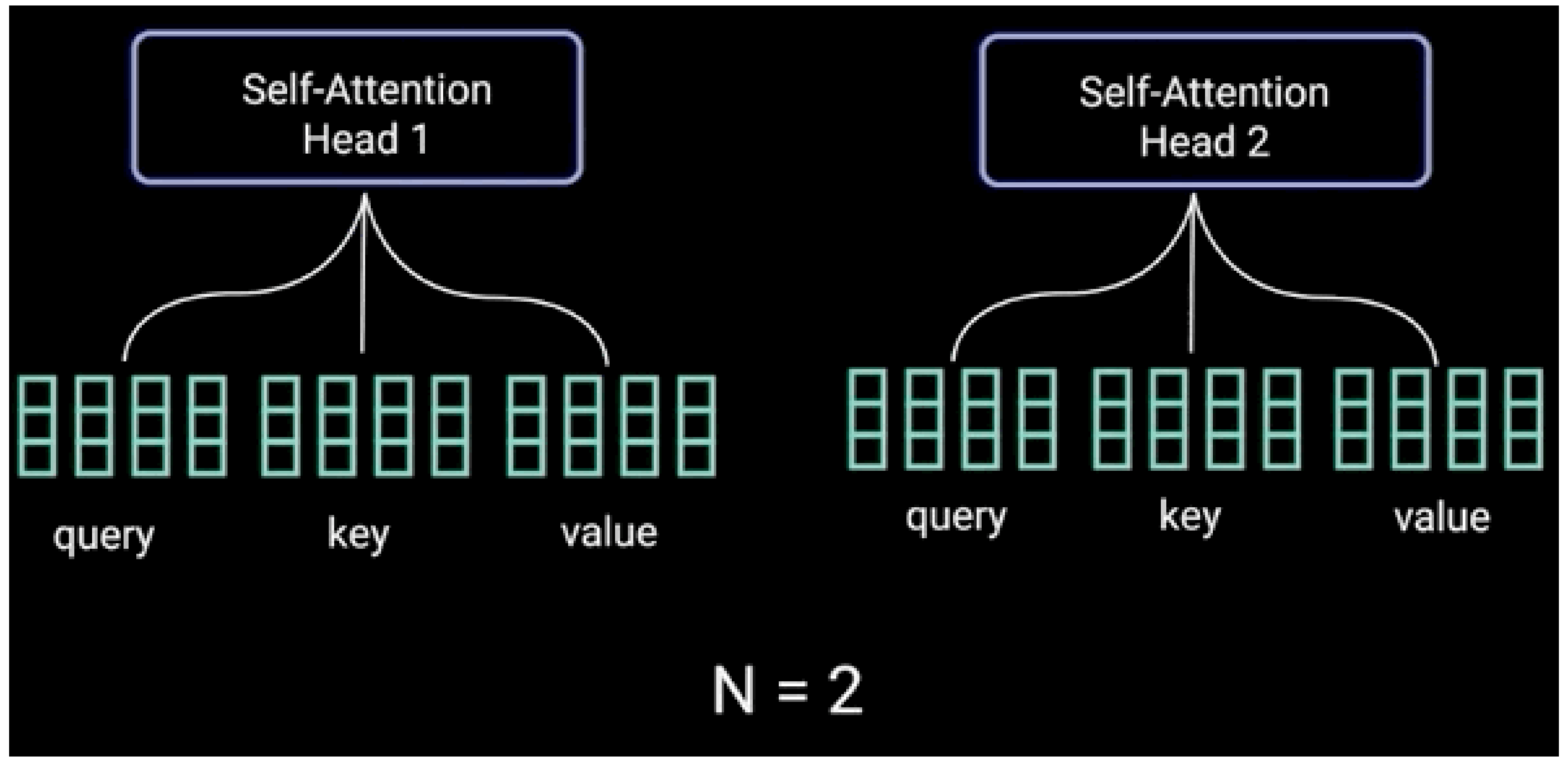
$$d_h = floor(\frac{d}{h})$$

"effective" embed dimensionality for each head

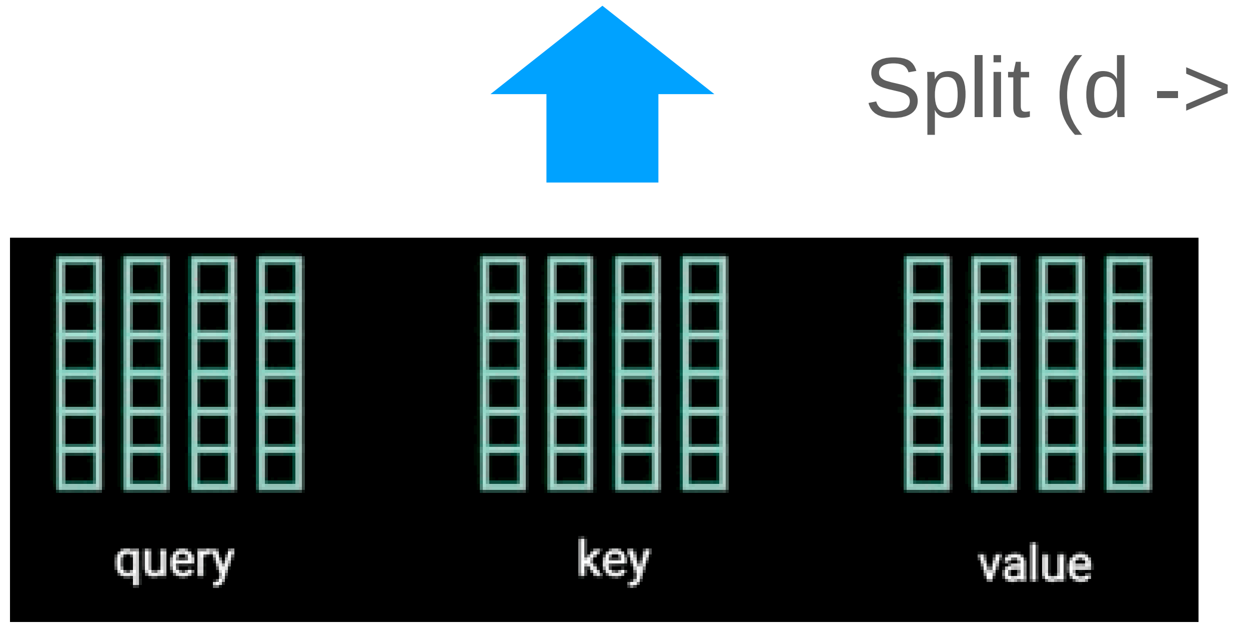
$$\mathbf{H}_h = \mathrm{Attention}(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h)$$

$$\mathbf{Y}(\mathbf{X}) = \underbrace{\mathrm{Concat}\left[\mathbf{H}_1, \ldots, \mathbf{H}_H\right]}_{\text{shape=[seq\_len, h*}d_h\text{] =[seq\_len, d]}} \mathbf{W}^{(\mathrm{o})}$$

Learned linear transform.
Shape=[d, d]



Splitting Q, K, V, N times before applying self-attention



Split (d -> d_h)

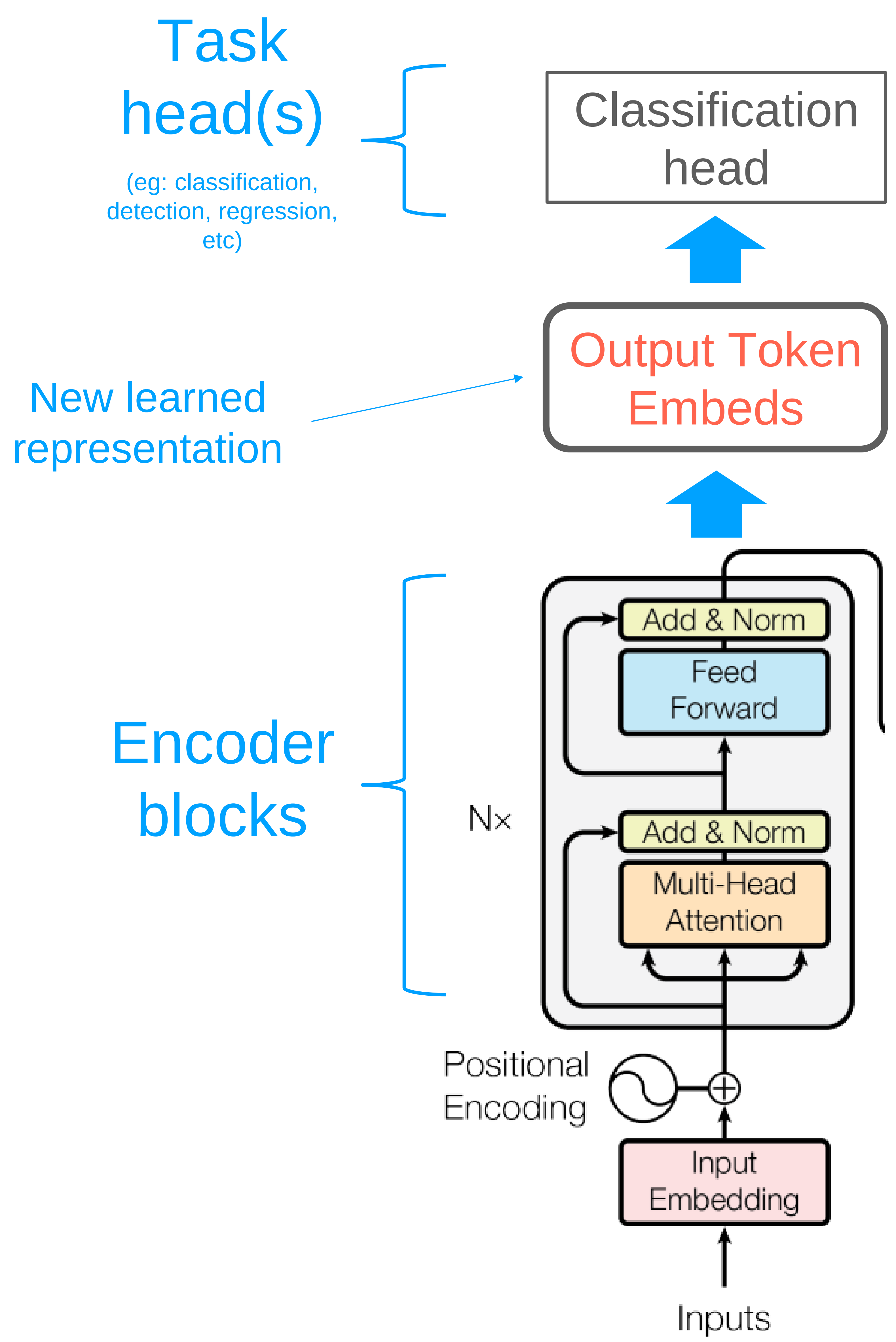
Implication: with this embedding "splitting", a MHA with h heads (operating on d//h dims) is roughly the same computation cost as a MHA with 1 head but operating on the full embedding dimensionality.

Rev02(2024-10-15) This slide originally had a mistake: it stated that we split X into \`h\` chunks for MHA. This is not true, we split Q,K,V into \`h\` chunks: Q_h, K_h, V_h

Encoder: Classification?

Recall: a Transformer encoder performs the following:

- Input: sequence X (shape=[batchsize, seq_len, dim_embed])
- Output: representation Y (shape=[batchsize, seq_len, dim_embed])
- Where Y is a learned transformation of X (eg via multi-head self attention, FFNs, etc)
- Notably, output token Y[:, ind_token, :] corresponds to input token X[:, ind_token, :]
- **Question**: how to perform classification on the output Y?



TransformerEncoder pytorch code: [link]

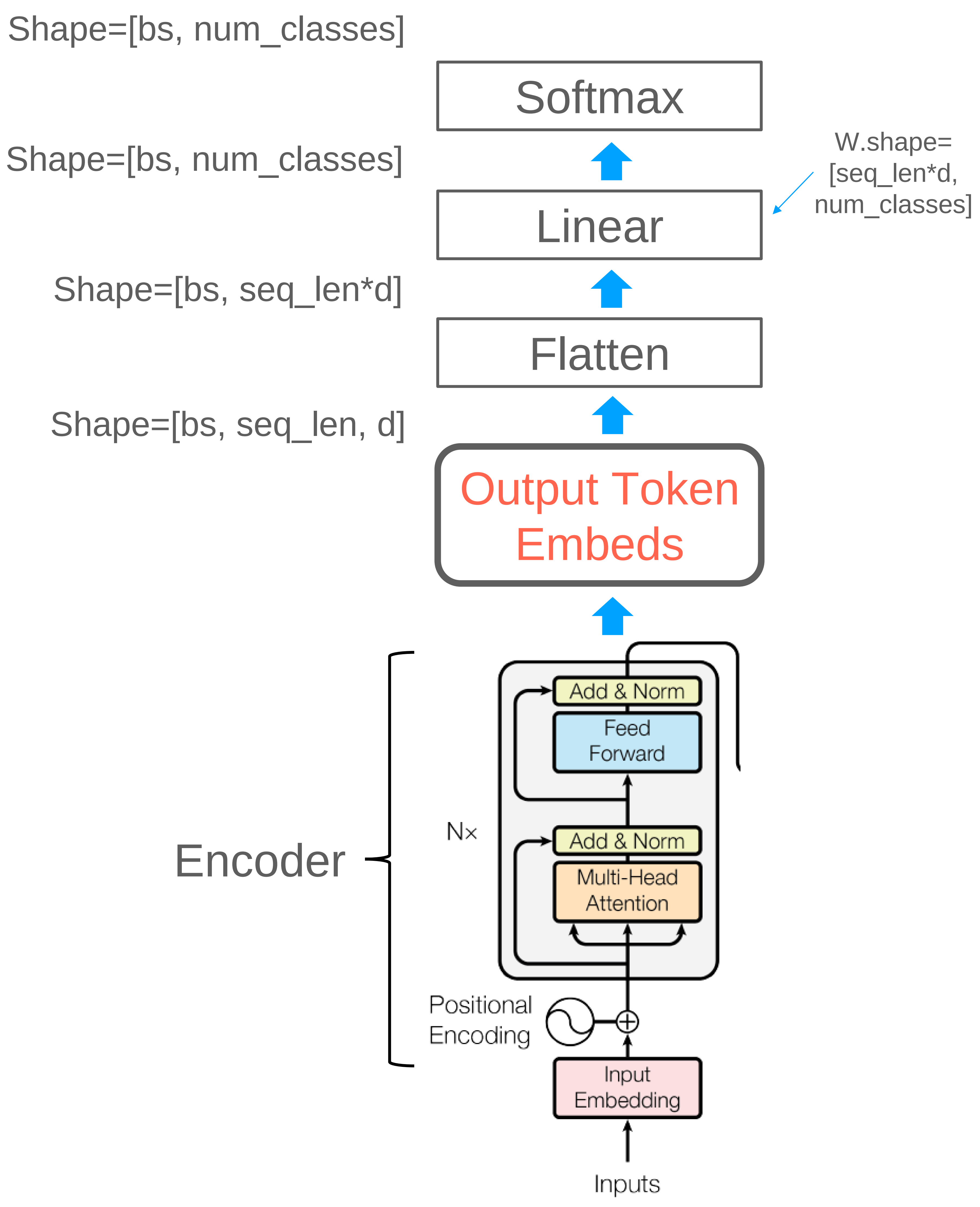
Encoder Classification V0: Naive classifier

- **Proposal**: flatten the Y from [batchsize, seq_len, dim_hidden] to [batchsize, seq_len*dim_hidden], and add a Linear(in=seq_len*dim_embed, out=num_classes) layer after the Encoder output.
- **Question**: what are the pros/cons of this?

  Pro: Simple

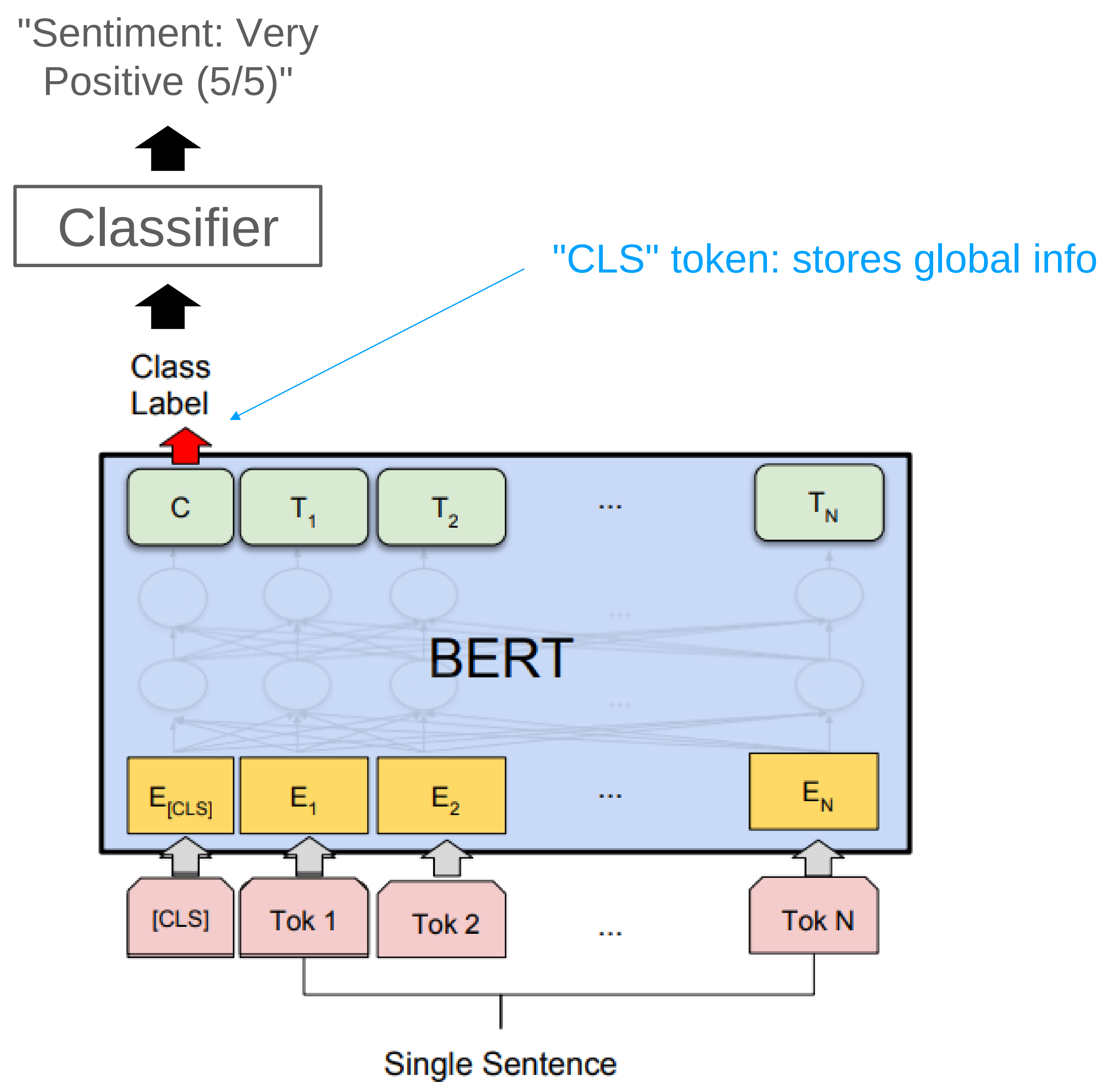
  Con:
  Hard codes the sequence length into the classifier, which means you can't easily modify the sequence length past whatever length you used during training

  Can be computationally expensive: for long sequence lengths and large number of target classes, the Linear layer can become too large



Classification approach 1: "CLS" token

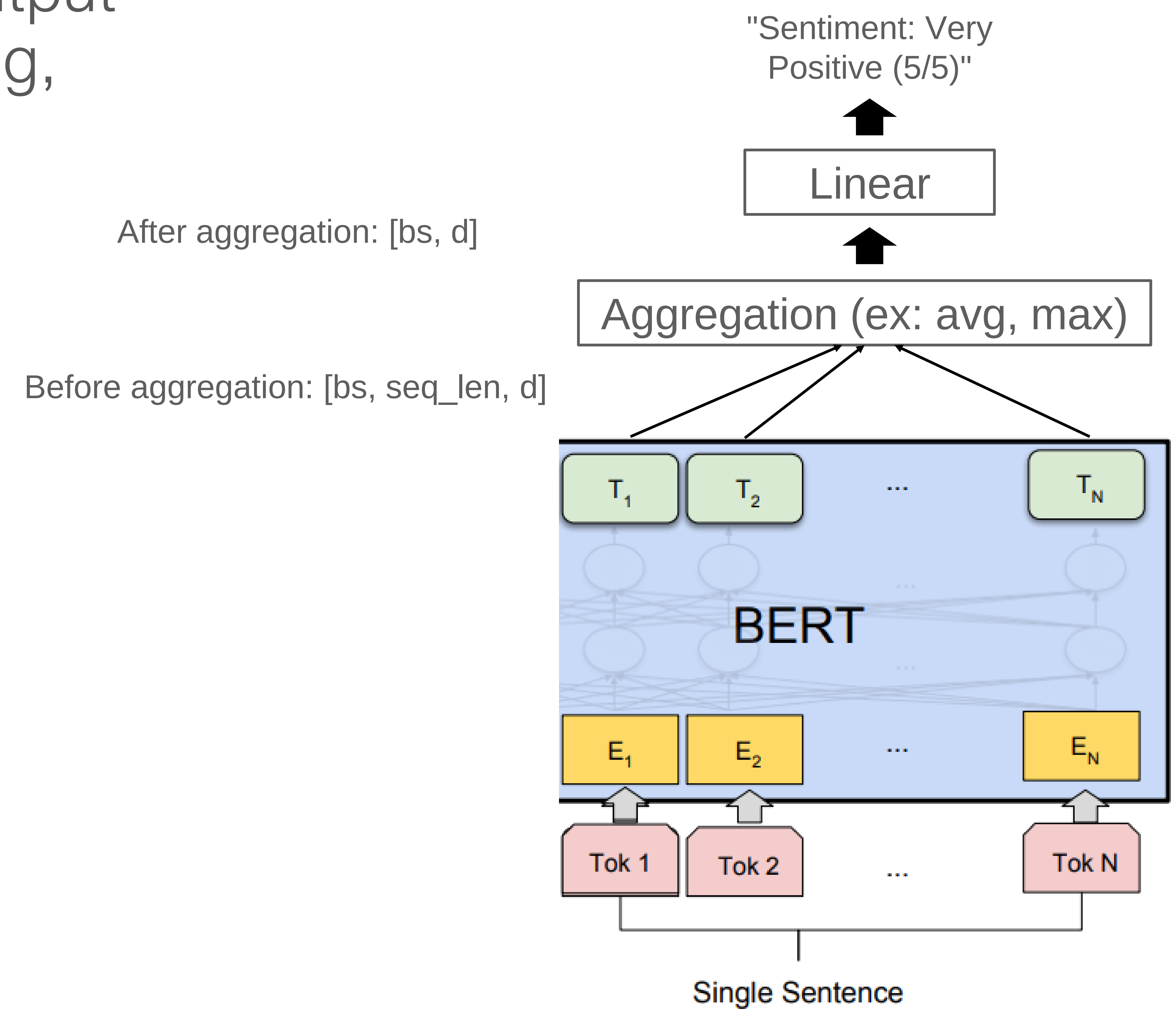
- Key idea: prepend a "**CLS**" token to the start of every sequence. Then, train a classifier on top of this CLS token embedding
    - Intuition: CLS token stores the "global" info about the sentence



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

Classification approach 2: Token aggregation

- Key idea: aggregate the `seq_len` output tokens into a single output embedding, then add your classifier on top of this
  - Ex: average, max



Question: what is the shape of the Linear layer's W weight?

Answer: [num_classes, d]

Decoder

- Useful for tasks involving token generation
  - Ex: machine translation, text summarization, question-and-answer bots, etc.
- Key concepts
  - Cross-attention
  - Masked self attention
  - Auto-regressive inference

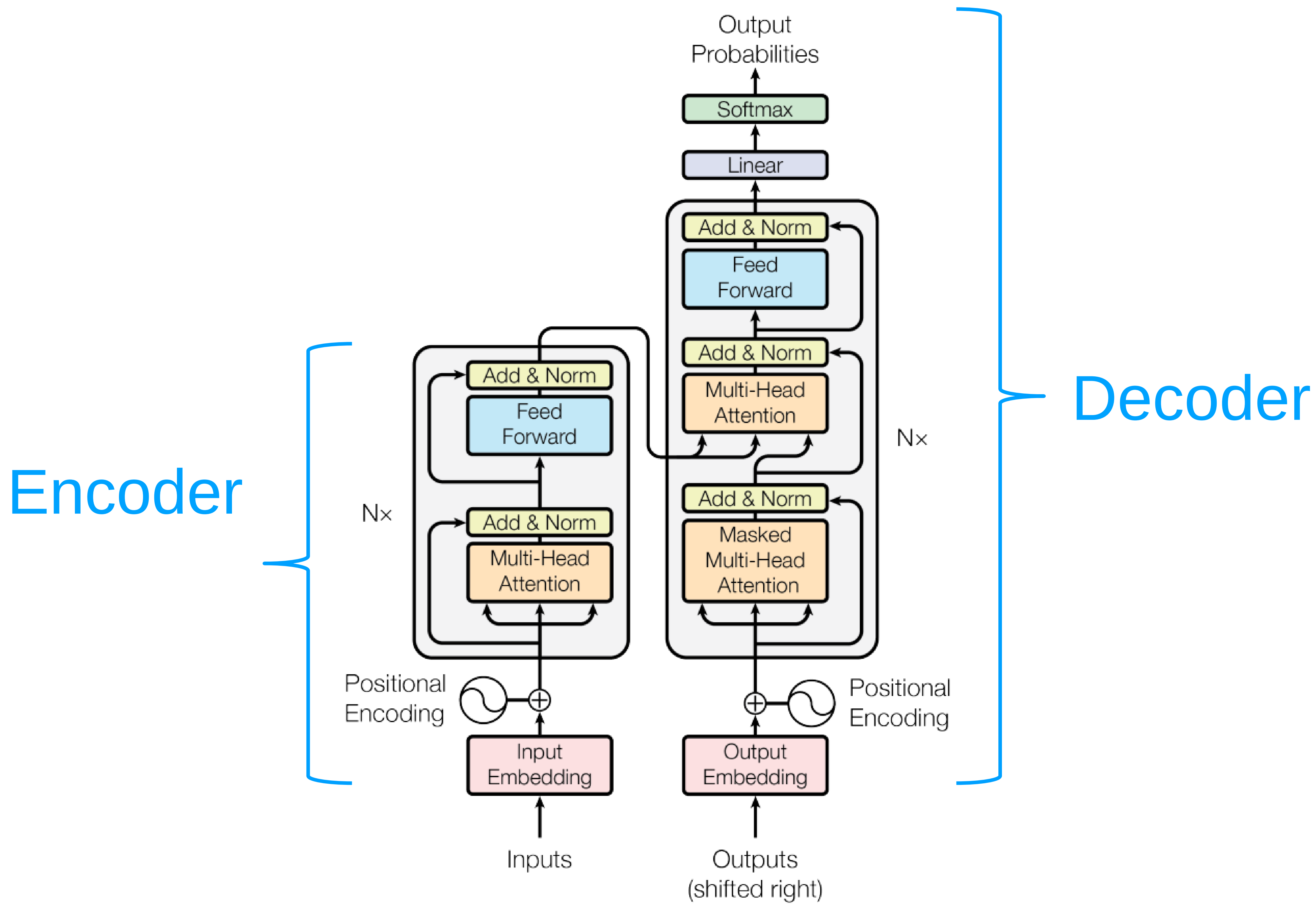
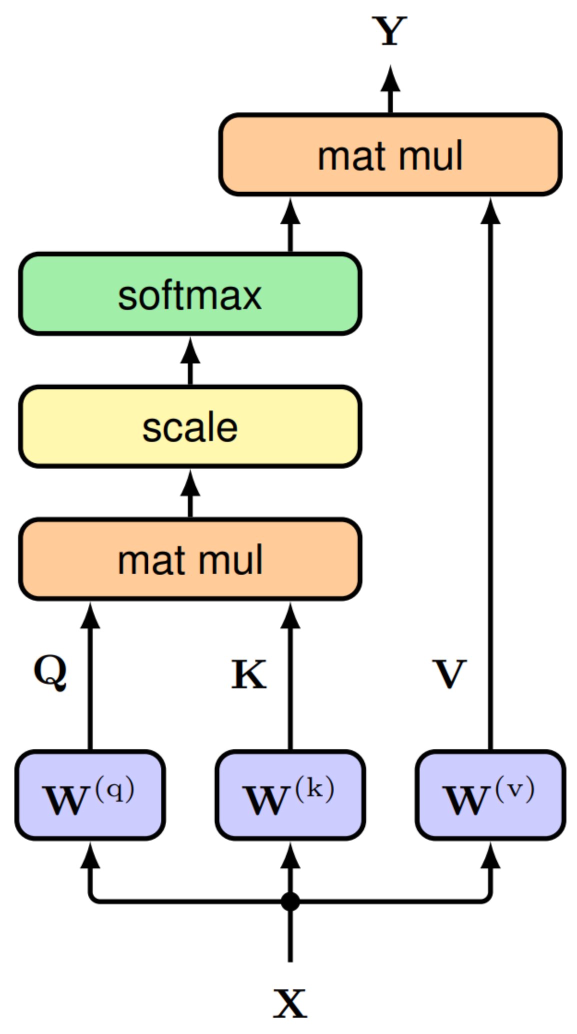


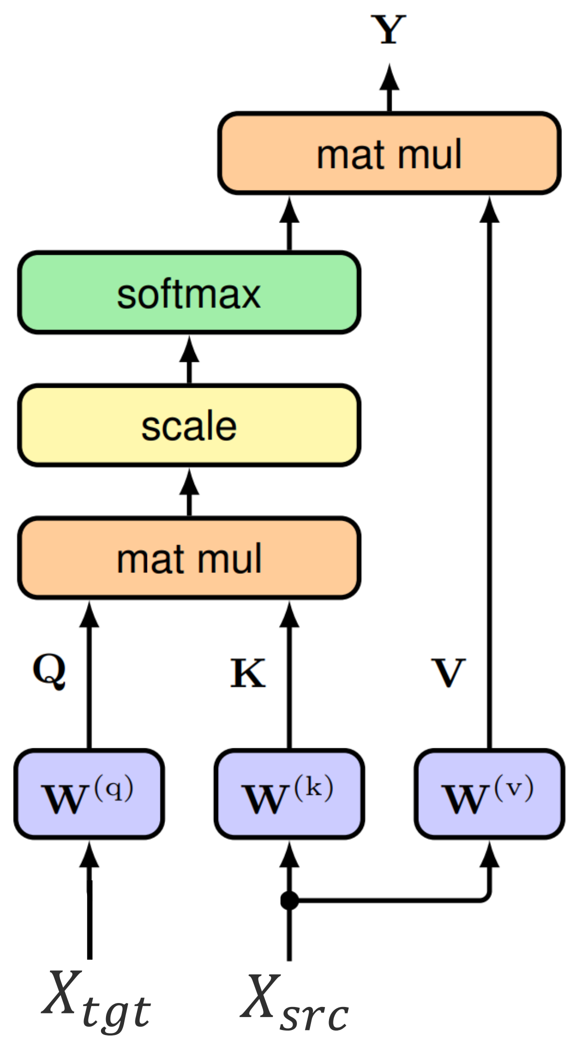
Figure 1: The Transformer - model architecture.

Cross attention

- Recall: in the Encoder's multi-head attention (MHA), we had only one input sequence, aka **"Self attention"**
- **"Cross attention"**: MHA but with **two** different input seqs.
- Tip: consider English->French translation, where we have a target sequence X_{tgt} (French) and an input sequence X_{src} (English)
- Intuition: given X_{tgt} and X_{src}, do the following
  - Cross-attention weights: determine how important tokenA from X_{src} is to tokenB from X_{tgt}
  - Attention-weighted transform: given cross-attention weights, transform X_{src}



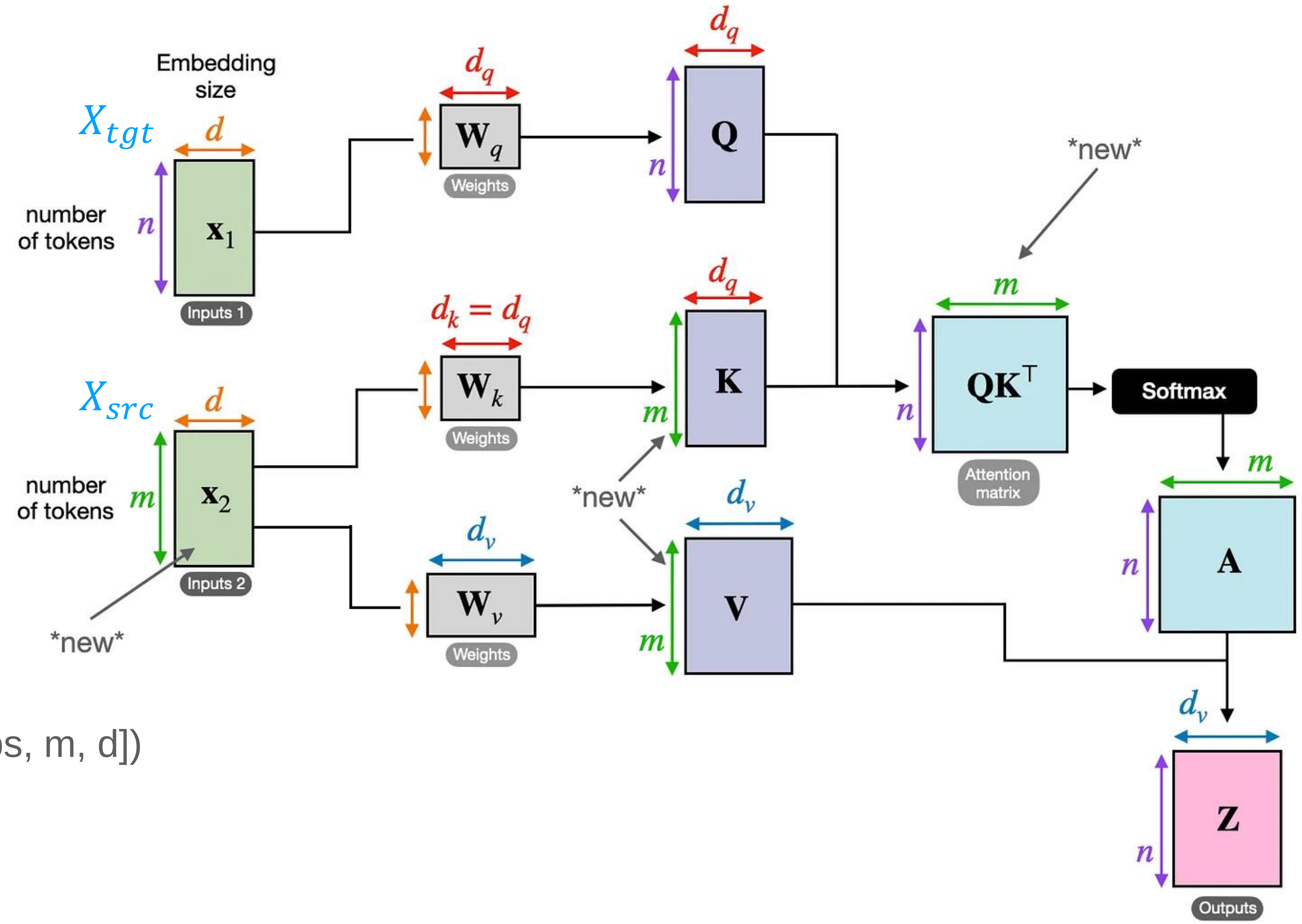
Self-attention



Cross-attention

Cross attention: seq lengths

- Note that, in this formulation, the sequence lengths are allowed to be different for X_{tgt} and X_{src}!
- Fortunately, all shapes adjust in the natural way:



Input: X_{tgt} (shape=[bs, n, d]), X_{src} (shape=[bs, m, d])
Output: Z (shape=[bs, n, d])

Important: X_{tgt} determines the output sequence length!

https://magazine.sebastianraschka.com/p/understanding-and-coding-self-attention

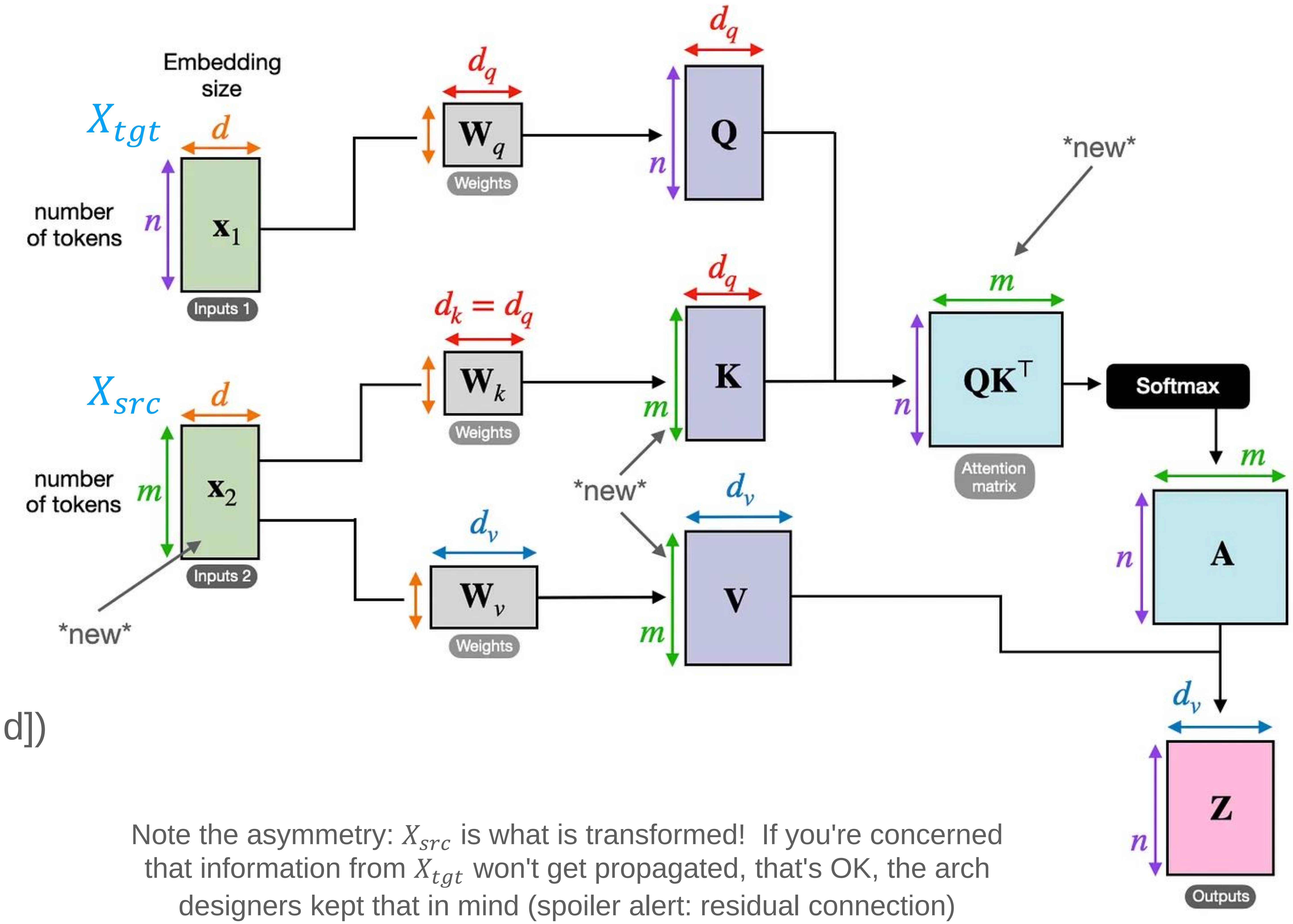
Cross attention: equations ("single head")

$$Q = X_{tgt} W_q$$
$$K = X_{src} W_k$$
$$V = X_{src} W_v$$

$$A = attention(Q, K, V) = Softmax(\frac{QK^T}{\sqrt{d}})$$

$$Out = AV$$

Can do multi-head cross-attention in the natural way, eg split up both X_{tgt} and X_{src} in the `d` dimension. Assumes that both X_{tgt} and X_{src} have the same embed dim!



Input: X_{tgt} (shape=[bs, n, d]), X_{src} (shape=[bs, m, d])
Output: Z (shape=[bs, n, d])

Note the asymmetry: X_{src} is what is transformed! If you're concerned that information from X_{tgt} won't get propagated, that's OK, the arch designers kept that in mind (spoiler alert: residual connection)

Important: X_{tgt} determines the output sequence length!

https://magazine.sebastianraschka.com/p/understanding-and-coding-self-attention

Cross-attention: attention scores

- Cross-attention scores lets us see what tokens from X_src are "relevant" to which tokens in X_tgt
- Ex: A[2, 1] = 0.7 means source token "ate" has high importance 0.7 to the target token "pris" for the machine translation task.
    - French: "pris" means "took" (aka "eat")

$$X_{src}$$

$$X_{tgt} \quad \begin{array}{r c} & \begin{array}{cccc} \text{"I"} & \text{"ate"} & \text{"breakfast"} & \text{"already"} \end{array} \\ \begin{array}{r} \text{"j'ai"} \\ \text{"déjà"} \\ \text{"pris"} \\ \text{"le"} \\ \text{"petit"} \\ \text{"déjeuner"} \end{array} & \begin{bmatrix} 0.8 & 0.2 & 0.1 & 0.0 \\ 0.1 & 0.2 & 0.1 & 0.6 \\ 0.1 & 0.7 & 0.2 & 0.1 \\ 0.0 & 0.2 & 0.8 & 0.0 \\ 0.0 & 0.1 & 0.9 & 0.0 \\ 0.0 & 0.1 & 0.9 & 0.0 \end{bmatrix} \end{array}$$

"I ate breakfast already" -> "j'ai déjà pris le petit déjeuner"

Masked self attention: motivation

- Let's consider the **machine translation problem**
- Dataset: paired sentences from source language to target language (ex: French to English)
- Task:
  - Given English text, translate it to French

Dataset rows

"I ate breakfast already"
-> "j'ai déjà pris le petit déjeuner"

"Where is the bathroom?"
-> "où sont les toilettes?"

...

Aside: tokenizers and "control characters"

- **Clever trick**: represent the start and end of a sequence via "<START>" and "<END>" tokens. These are special "control" tokens added to the tokenizer vocabulary
  - Implication: model emits <END> to signal to stop generating tokens

```
tokens = ["hello", "there"]
tokens = [TOKEN_START] + tokens + [TOKEN_END]
```

Aside: tokenizers and "control characters"

- Other common control tokens:
  - <PAD>: if you need to pad your input to a specific seq_len (ex: batching N input sentences each with different number of tokens), insert <PAD> tokens (typically right-pad)
  - <UNKNOWN>: if an unexpected input comes in (eg text never seen before in training), then represent it with this
  - <CLS>: the classification token we've seen before!
  - ...

Demo: huggingface text encoder

```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained("bert-base-uncased")
input_text = "I am Eric meow"

# Tokenizer
input_tokens = tokenizer(input_text, return_tensors='pt')
print("input_tokens: ", input_tokens)
print("input_tokens.input_ids.shape: ", input_tokens.input_ids.shape)
print("convert_ids_to_tokens: ",
tokenizer.convert_ids_to_tokens(input_tokens.input_ids[0, :]))

output = model(**input_tokens)
print("output shape: ", output.last_hidden_state.shape)
```

Note: tokenizer can break up a single word into multiple tokens!

```
input_tokens:  {'input_ids': tensor([[ 101, 1045, 2572, 4388, 2033, 5004,  102]]),
'token_type_ids': tensor([[0, 0, 0, 0, 0, 0, 0]]), 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1]])}
input_tokens.input_ids.shape:  torch.Size([1, 7])
convert_ids_to_tokens:  ['[CLS]', 'i', 'am', 'eric', 'me', '##ow', '[SEP]']

output shape:  torch.Size([1, 7, 768])
```

In this implementation, <SEP> is the "END" token

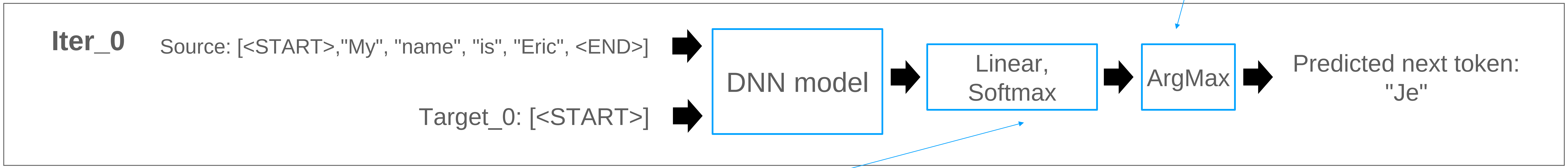
dim_embed: 768

https://huggingface.co/google-bert/bert-base-uncased

Machine translation setup

- Given a source-language sentence (EN) and a target-language sentence (FR), how to we set up the training task/loss for the translation task?
- One way: pose it as a "next token prediction" task!
  - Notably: inference is done in an iterative auto-regressive manner

ArgMax: aka "Choose token with largest predicted probability score"



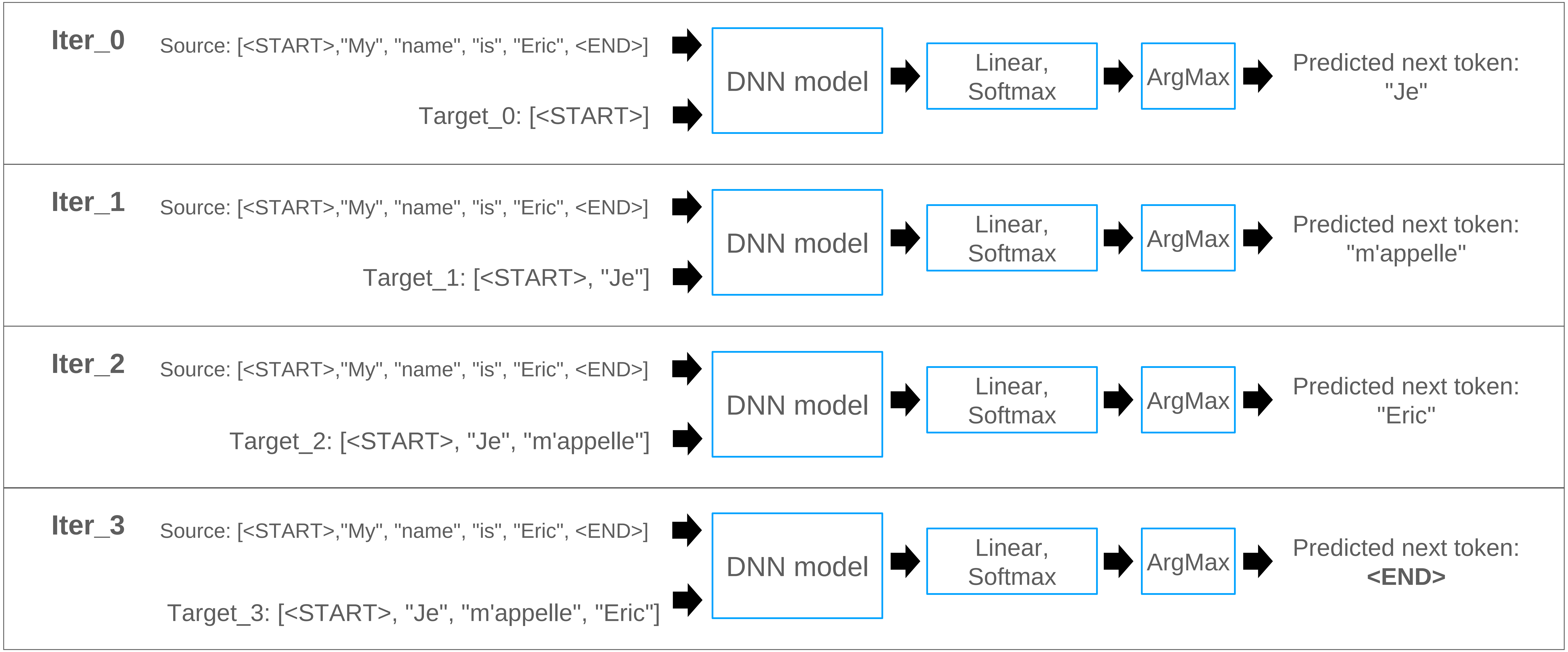
Note: "Linear, Softmax" produces a probability score vector over the tokenizer vocabulary size. Ex: Bert tokenizer has ~30k tokens in its vocabulary [link]

Softmax is done across the entire tokenizer vocabulary (ex: 30k vocab!)

```
# Token, predicted_probability
Token0 ("eau"): 0.01
Token1 ("boulangerie"): 0.02
...
Token42 ("Je"): 0.9
...
```

Translation as "next token prediction" (inference)

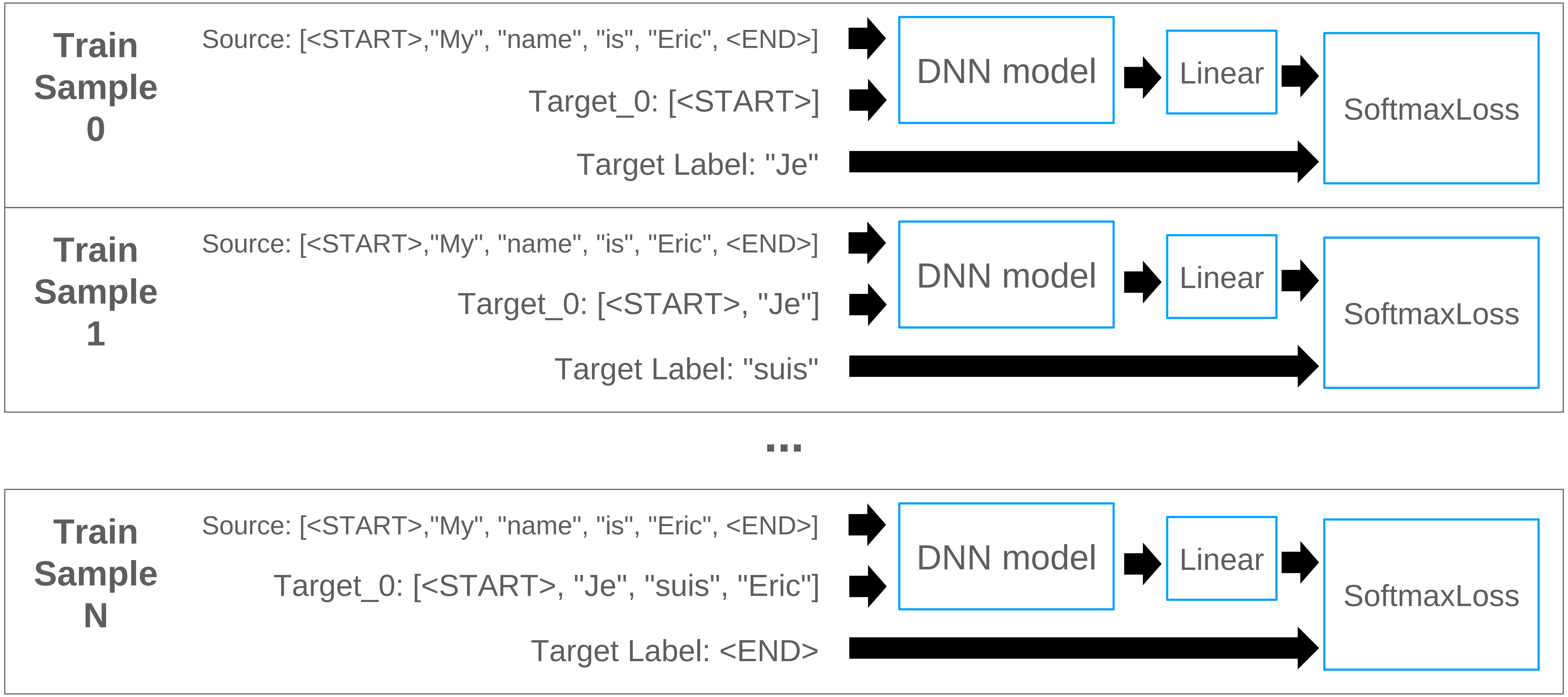
Note: "My name is Eric" -> "Je m'appelle Éric"



Output: [<START>, "Je", "m'appelle", "Eric", <END>]

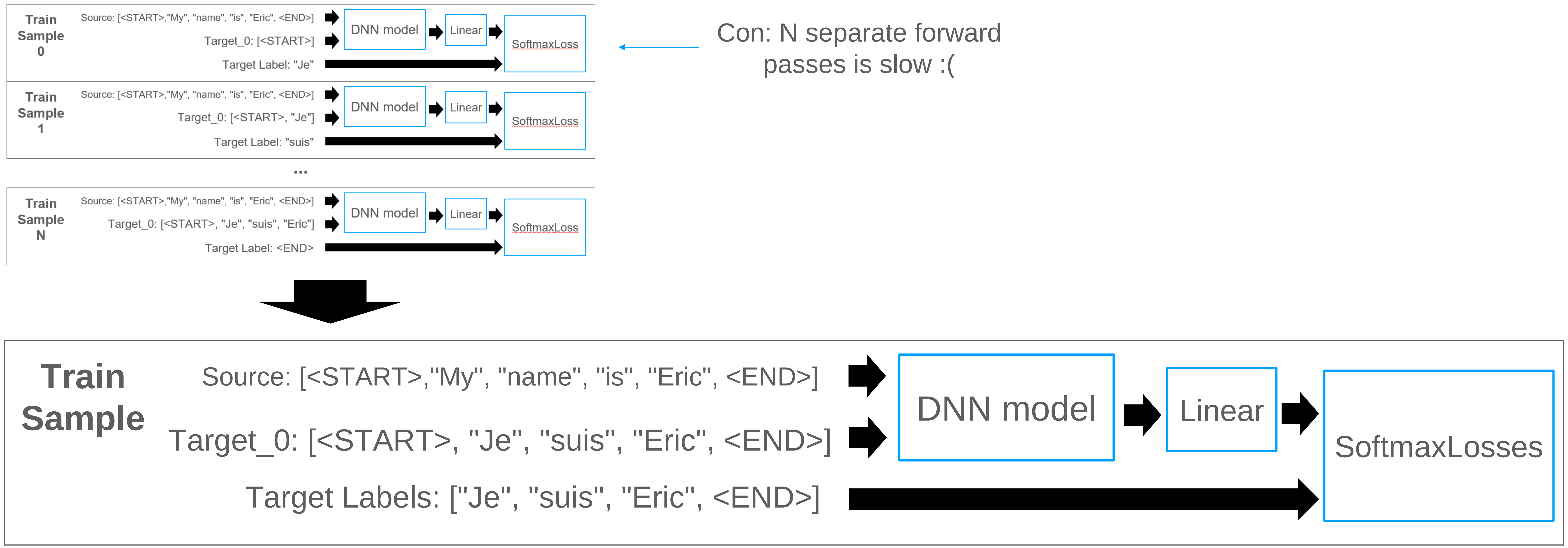
Translation loss

- How to build a training loss out of this idea?
- Answer: all-possible next-token prediction tasks (classification loss)!



Translation loss: optimization opportunity?

- In practice: for a target sequence with length N, we don't want to have to do N separate forward passes during training (lots of repeated computation!)
- Is there a way to do a single forward pass passing in the full target sequence once and getting all N prediction tasks at once?



Translation loss: Attempt 1

- Idea: let's connect our Encoder and Decoder via cross-attention
  - Encoder: given source sequence (EN), generate new **source** token embeds
  - Decoder: given target sequence (FR) and Encoder output (EN), generate new **target** token embeds
- Use cross-attention to "fuse" information from source sequence (EN) with target sequence (FR)

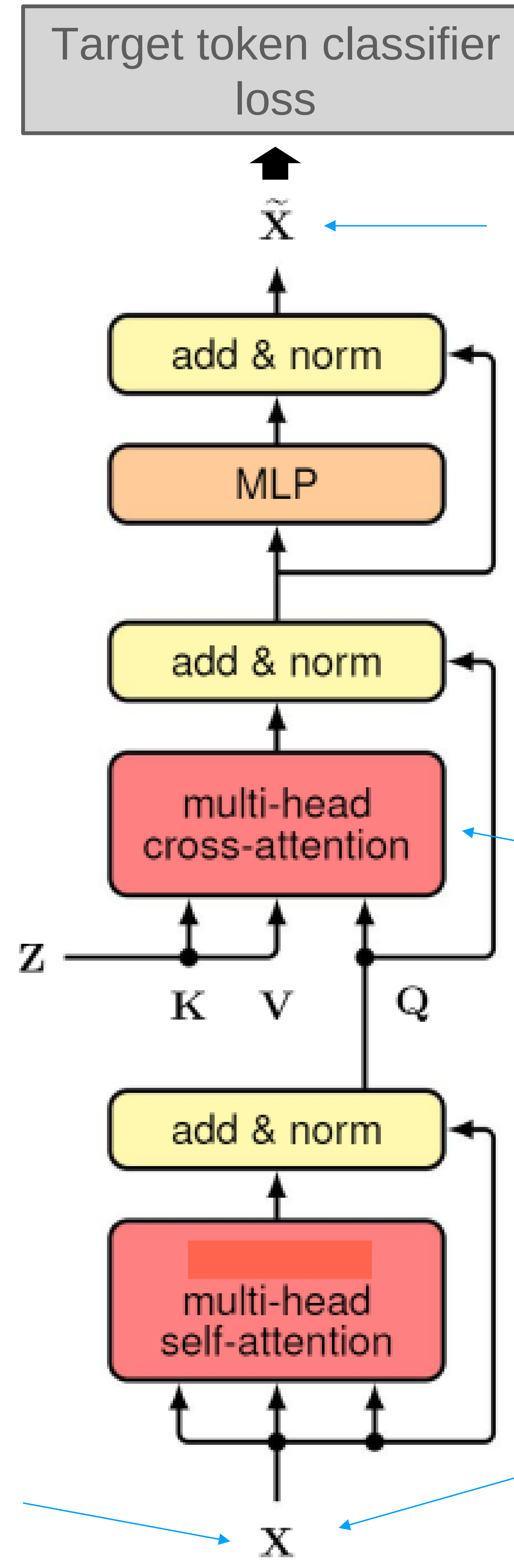
```
# Token, predicted_probability
Token0 ("eau"): 0.01
Token1 ("boulangerie"): 0.02
...
Token42 ("Je"): 0.9
...
```

ind=0, target label: 42

```
# Token, predicted_probability
Token0 ("eau"): 0.01
Token1 ("boulangerie"): 0.02
...
Token9001 ("suis"): 0.75
...
```

ind=1, target label: 9001

Intuition: for each sequence position `ind`, predict the correct output token



Decoder output embeds. Shape=[bs, **seq_len_tgt**, d]

Z is the Encoder output. shape=[batchsize, **seq_len_src**, d]

Recall: cross-attention lets us "mix" two input sequences
Output shape: [bs, **seq_len_tgt**, d]

X is our target sentence, shape=[bs, **seq_len_tgt**, d]

Note: omitted in this figure is the positional embeddings added to input X

Translation loss: Attempt 1

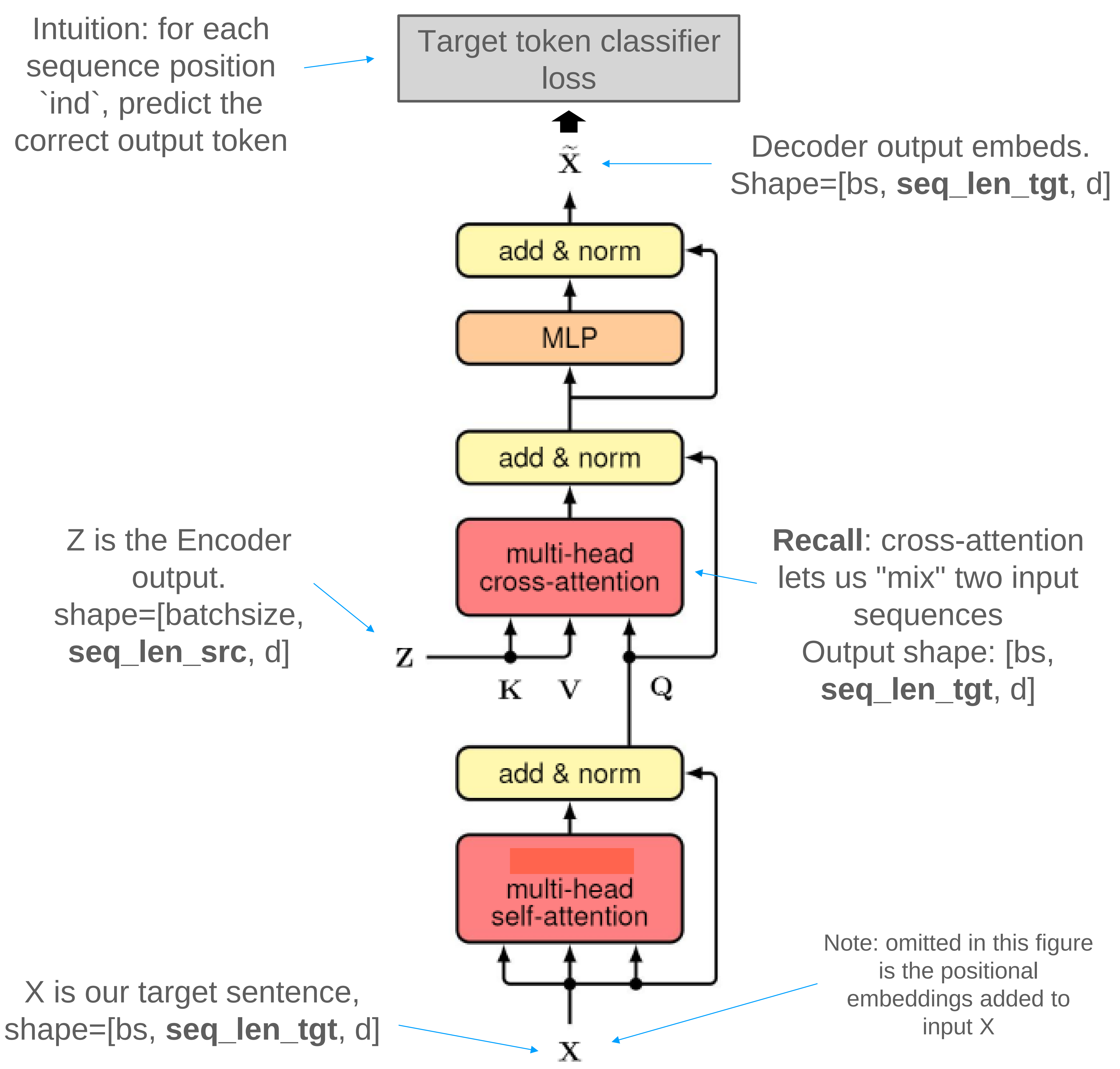
- Question: from a modeling perspective, why might this current setup be suboptimal?
  - Hint: information leakage
- **Answer**: decoder can "cheat" and use information from later in the sequence when predicting the current token!
  - Violates our desire that a prediction for sequence position `ind` should only use information before `ind` ("causality")

```
# Token, predicted_probability
Token0 ("eau"): 0.01
Token1 ("boulangerie"): 0.02
...
Token42 ("Je"): 0.9
...
```

ind=0, target label: 42

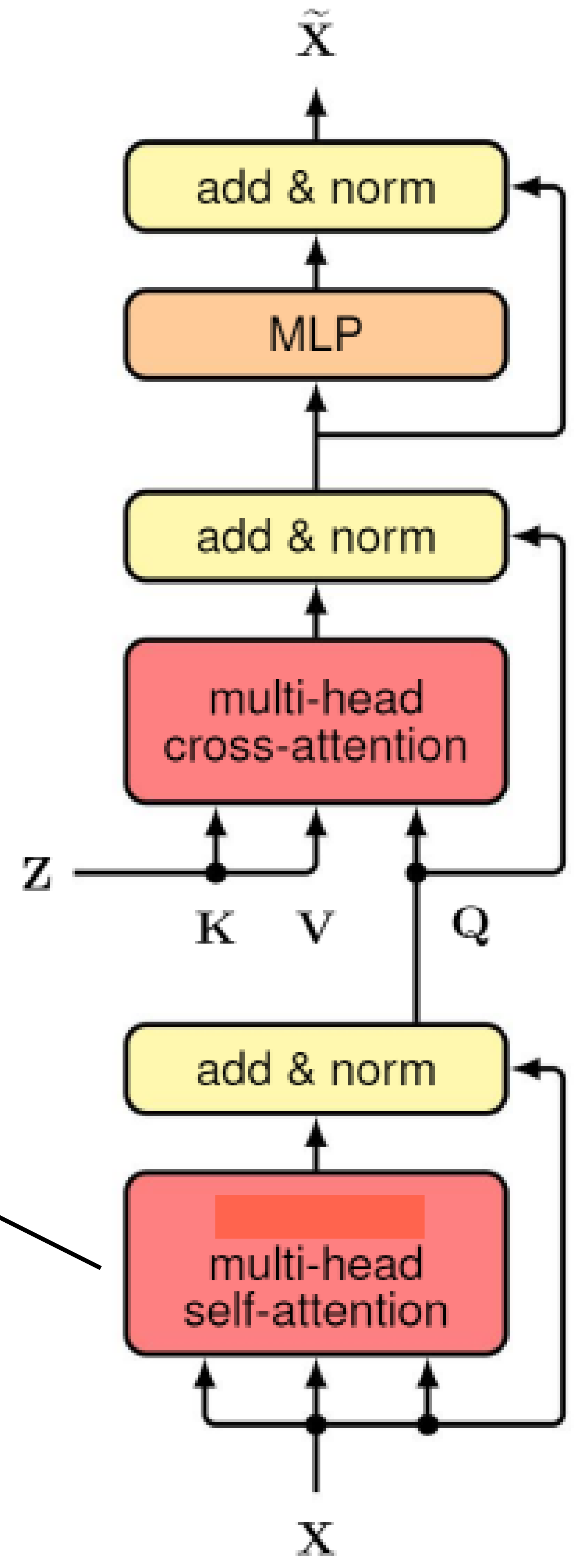
```
# Token, predicted_probability
Token0 ("eau"): 0.01
Token1 ("boulangerie"): 0.02
...
Token9001 ("suis"): 0.75
...
```

ind=1, target label: 9001



Solution: attention scores (no causal mask)

$$\begin{array}{r c}
 & \begin{array}{ccccc} \langle\text{START}\rangle & \text{"je"} & \text{"suis"} & \text{"Eric"} & \langle\text{END}\rangle \end{array} \\
\begin{array}{r} \langle\text{START}\rangle \\ \text{"je"} \\ \text{"suis"} \\ \text{"Eric"} \\ \langle\text{END}\rangle \end{array} & \begin{bmatrix} 0.8 & 0.2 & 0.1 & 0.1 & 0.2 \\ 0.0 & 0.1 & 0.9 & 0.3 & 0.1 \\ 0.1 & 0.2 & 0.1 & 0.6 & 0.1 \\ 0.1 & 0.7 & 0.2 & 0.1 & 0.2 \\ 0.0 & 0.2 & 0.8 & 0.0 & 0.9 \end{bmatrix}
\end{array}$$

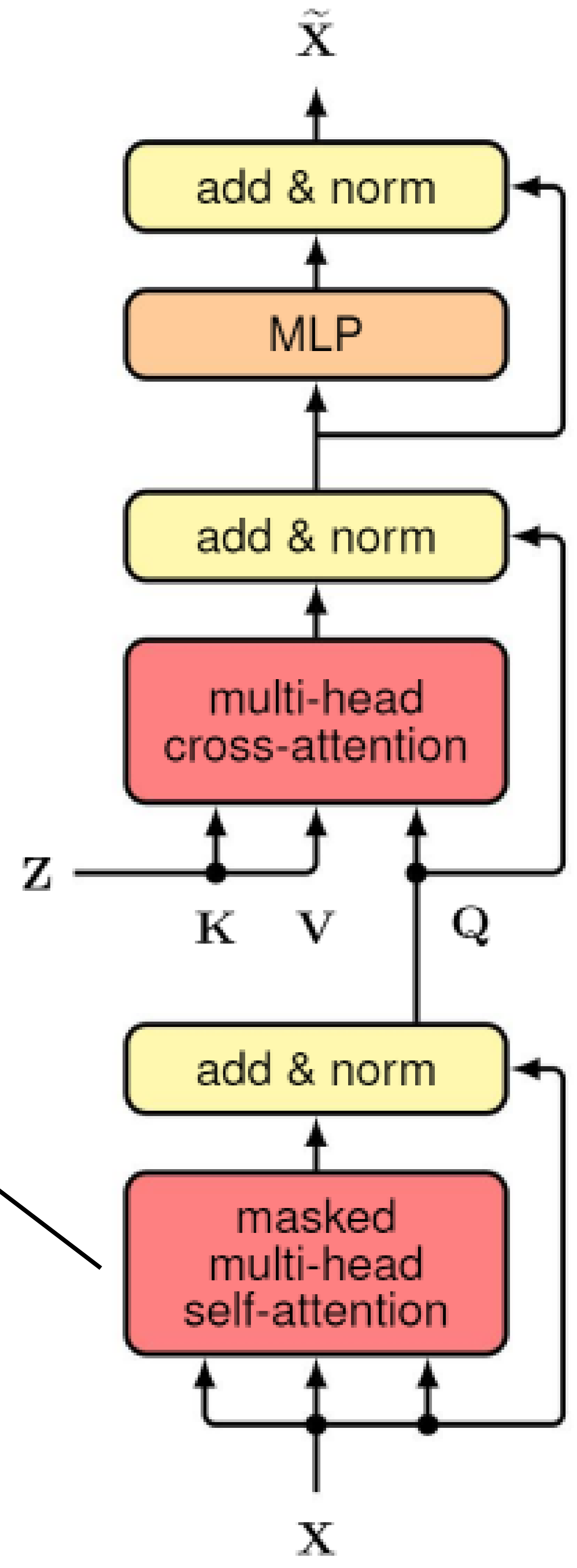


Issue: when predicting the first token "je", the decoder's can utilize information from the rest of the sequence.
Solution: apply a "look ahead" mask to the decoder's self-attention weights

Solution: attention scores (with causal mask)

$$
\begin{array}{c}
\\
\text{<START>} \\ \text{"je"} \\ \text{"suis"} \\ \text{"Eric"} \\ \text{<END>}
\end{array}
\begin{array}{c}
\begin{array}{ccccc} \text{<START>} & \text{"je"} & \text{"suis"} & \text{"Eric"} & \text{<END>} \end{array} \\
\begin{bmatrix}
0.8 & -inf & -inf & -inf & -inf \\
0.0 & 0.1 & -inf & -inf & -inf \\
0.1 & 0.2 & 0.1 & -inf & -inf \\
0.1 & 0.7 & 0.2 & 0.1 & -inf \\
0.0 & 0.2 & 0.8 & 0.0 & 0.9
\end{bmatrix}
\end{array}
$$

We apply the "look ahead" mask before the softmax(), and after the division by $\sqrt{d}$



Masked Attention scores (post softmax)

$$
\begin{array}{r c}
 & \begin{array}{ccccc} \text{<START>} & \text{"je"} & \text{"suis"} & \text{"Eric"} & \text{<END>} \end{array} \\
\begin{array}{r} \text{<START>} \\ \text{"je"} \\ \text{"suis"} \\ \text{"Eric"} \\ \text{<END>} \end{array} &
\begin{bmatrix}
0.8 & -inf & -inf & -inf & -inf \\
0.0 & 0.1 & -inf & -inf & -inf \\
0.1 & 0.2 & 0.1 & -inf & -inf \\
0.1 & 0.7 & 0.2 & 0.1 & -inf \\
0.0 & 0.2 & 0.8 & 0.0 & 0.9
\end{bmatrix}
\end{array}
$$

$$
\begin{array}{r c}
 & \begin{array}{ccccc} \text{<START>} & \text{"je"} & \text{"suis"} & \text{"Eric"} & \text{<END>} \end{array} \\
\begin{array}{r} \text{<START>} \\ \text{"je"} \\ \text{"suis"} \\ \text{"Eric"} \\ \text{<END>} \end{array} &
\begin{bmatrix}
1.0 & 0 & 0 & 0 & 0 \\
0.1 & 0.9 & 0 & 0 & 0 \\
0.1 & 0.8 & 0.1 & 0 & 0 \\
0.1 & 0.7 & 0.2 & 0.1 & 0 \\
0.0 & 0.2 & 0.3 & 0.0 & 0.5
\end{bmatrix}
\end{array}
$$

Softmax (along rows)

Note that the Softmax(-Inf) turns into 0.0 probability.

Now, the decoder can't "cheat"!

Masked Attention scores (post softmax)

Exercise: show that the output of Masked attention leads to the property that, for output token at sequence position \`ind\`, H[bs, ind, :] only includes information from the first \`ind\` tokens in V.
Aka "masked attention indeed fixes the cheating problem"

$$\boldsymbol{Q = X_{tgt} W_q}$$
$$\boldsymbol{K = X_{tgt} W_k}$$
$$\boldsymbol{V = X_{tgt} W_v}$$
$$\boldsymbol{A = mask\_attention(Q, K, V) = Softmax(mask(\frac{QK^T}{\sqrt{d}}))}$$
$$\boldsymbol{H = AV}$$

$$\begin{bmatrix} 1.0 & 0 & 0 & 0 \\ 0.1 & 0.9 & 0 & 0 \\ 0.1 & 0.8 & 0.1 & 0 \\ 0.1 & 0.7 & 0.2 & 0.1 \end{bmatrix} \begin{bmatrix} V_{00} & V_{01} \\ V_{10} & V_{11} \\ V_{20} & V_{21} \\ V_{30} & V_{31} \end{bmatrix} = \begin{bmatrix} 1.0 * V_{00} & 1.0 * V_{01} \\ 0.1 * V_{00} + 0.9 * V_{10} & 0.1 * V_{01} + 0.9 * V_{11} \\ 0.1 * V_{00} + 0.8 * V_{10} + 0.1 * V_{20} & 0.1 * V_{01} + 0.8 * V_{11} + 0.1 * V_{21} \\ 0.1 * V_{00} + 0.7 * V_{10} + 0.2 * V_{20} + 0.1 * V_{30} & 0.1 * V_{01} + 0.7 * V_{11} + 0.2 * V_{21} + 0.1 * V_{31} \end{bmatrix}$$

A V H

$$= \begin{bmatrix} A_{00} * V_{[0,:]} \\ A_{10} {*} V_{[0,:]} + A_{11} {*} V_{[1,:]} \\ A_{20} {*} V_{[0,:]} + A_{21} {*} V_{[1,:]} + A_{22} {*} V_{[2,:]} \\ A_{30} {*} V_{[0,:]} + A_{31} {*} V_{[1,:]} + A_{32} {*} V_{[2,:]} + A_{33} {*} V_{[3,:]} \end{bmatrix}$$

Note that the 1st row of H only depends on the first row of V, the 2nd row of H only depends on the first two rows of V, etc.
Thus, we achieved our goal: the H embedding at sequence position \`ind\` only relies on tokens V that precede it (causally).

Aside: Masked attention implementation

Tip: we can implement \`mask_attention()\` by adding a simple mask to the pre-softmax inputs:

$$\boldsymbol{Q} = \boldsymbol{X}_{tgt}\boldsymbol{W}_q$$
$$\boldsymbol{K} = \boldsymbol{X}_{tgt}\boldsymbol{W}_k$$
$$\boldsymbol{V} = \boldsymbol{X}_{tgt}\boldsymbol{W}_v$$
$$\boldsymbol{A} = \boldsymbol{mask\_attention}(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = \boldsymbol{Softmax}(\boldsymbol{mask}(\frac{\boldsymbol{QK}^T}{\sqrt{\boldsymbol{d}}}))$$
$$\boldsymbol{H} = \boldsymbol{AV}$$

$$\underset{\text{(along rows)}}{\text{Softmax}}\left(\underbrace{\begin{bmatrix} 0.9 & 0.2 & 0.1 & 0.1 & 0.4 \\ 0.1 & 0.2 & 0.9 & 0.1 & 0.1 \\ 0.2 & 0.3 & 0.2 & 0.7 & 0.1 \\ 0.2 & 0.8 & 0.3 & 0.2 & 0.1 \\ 0.1 & 0.3 & 0.9 & 0.1 & 1.0 \end{bmatrix}}_{\frac{QK^T}{\sqrt{d}}} + \underbrace{\begin{bmatrix} 0 & -inf & -inf & -inf & -inf \\ 0 & 0 & -inf & -inf & -inf \\ 0 & 0 & 0 & -inf & -inf \\ 0 & 0 & 0 & 0 & -inf \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}}_{\text{attention\_mask}} \right) = \underbrace{\begin{bmatrix} 1.0 & 0 & 0 & 0 & 0 \\ 0.1 & 0.9 & 0 & 0 & 0 \\ 0.1 & 0.8 & 0.1 & 0 & 0 \\ 0.1 & 0.7 & 0.2 & 0.1 & 0 \\ 0.0 & 0.2 & 0.3 & 0.0 & 0.5 \end{bmatrix}}_{\text{A}}$$

Recall: in Python (and most programming languages*), \`-Inf + <any number> = -Inf\`

*This property is defined by the IEEE floating point standard

Translation loss: Attempt 2!

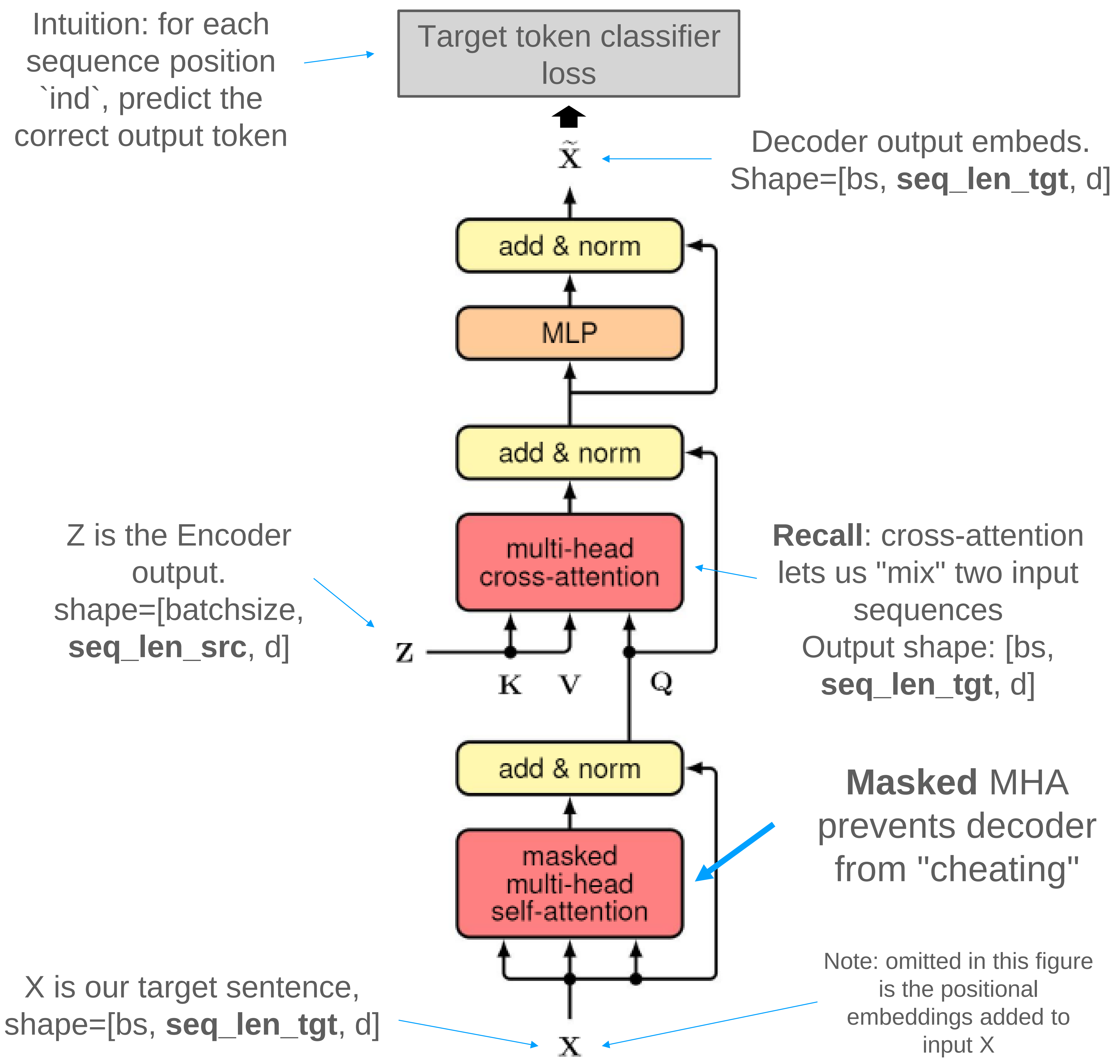
- We've (finally) arrived at a working Encoder+Decoder implementation for machine translation!
- Masked MHA: prevent information leakage ("preserve causality")
- Cross-attention: fuse information from source (EN) and target (FR) sequences
- Train task: next-token prediction task

```
# Token, predicted_probability
Token0 ("eau"): 0.01
Token1 ("boulangerie"): 0.02
...
Token42 ("Je"): 0.9
...
```

ind=0, target label: 42

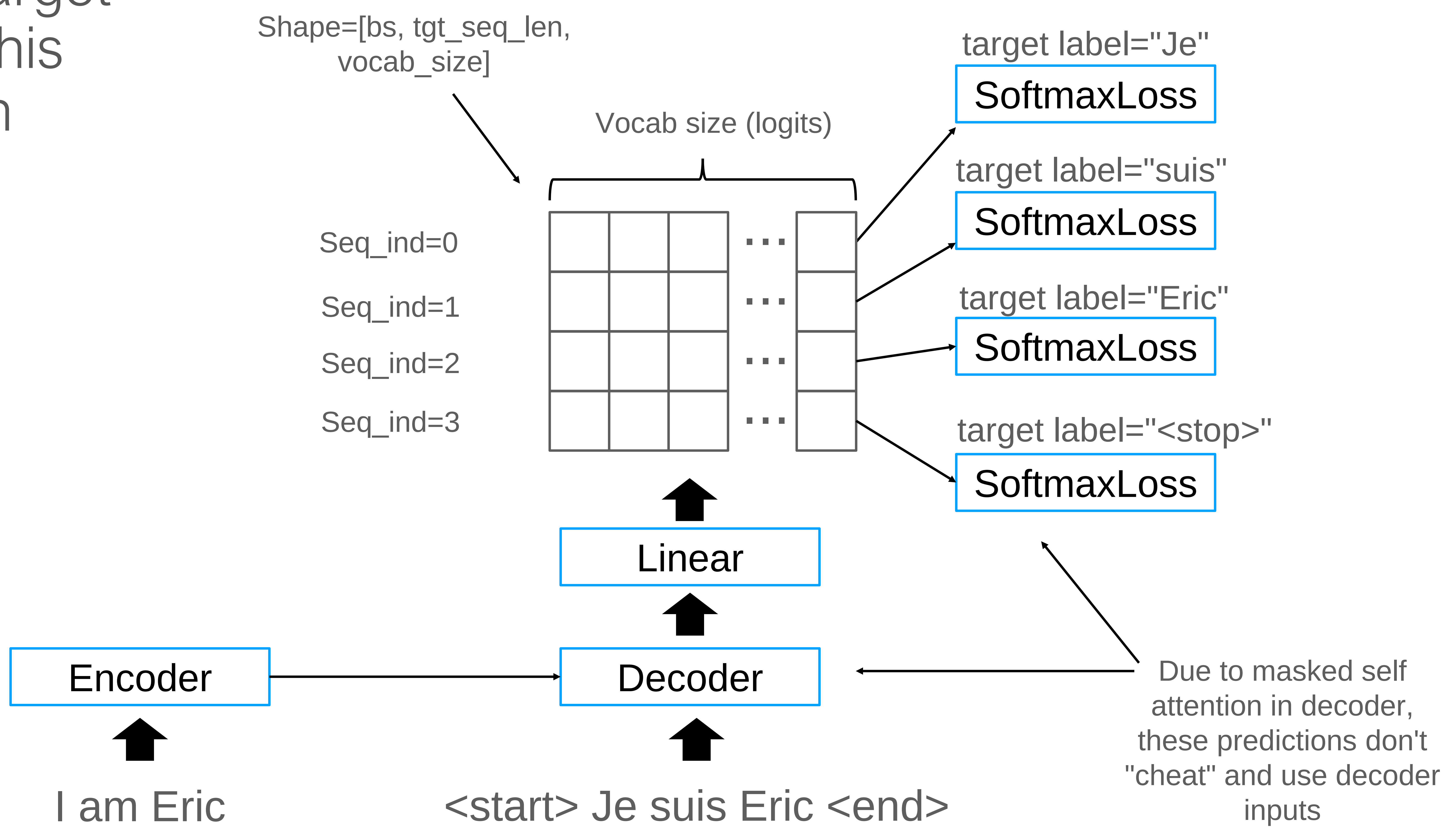
```
# Token, predicted_probability
Token0 ("eau"): 0.01
Token1 ("boulangerie"): 0.02
...
Token9001 ("suis"): 0.75
...
```

ind=1, target label: 9001



Translation loss: next token prediction

- For a given source->target dataset row, we turn this into multiple prediction tasks:



Encoder-Decoder models

- The OG "Attention Is All You Need" paper [[link]]
- Tasks
  - English->German, English->French translation
  - "English constituency parsing"
    - Aka: Parse a sentence into a subject/verb/noun parse tree

```
                 Sentence (S)
                      |
        +-------------+------------+
        |                          |
     Noun (N)               Verb Phrase (VP)
        |                          |
      John                +--------+--------+
                          |                 |
                       Verb (V)          Noun (N)
                          |                 |
                        sees              Bill
```

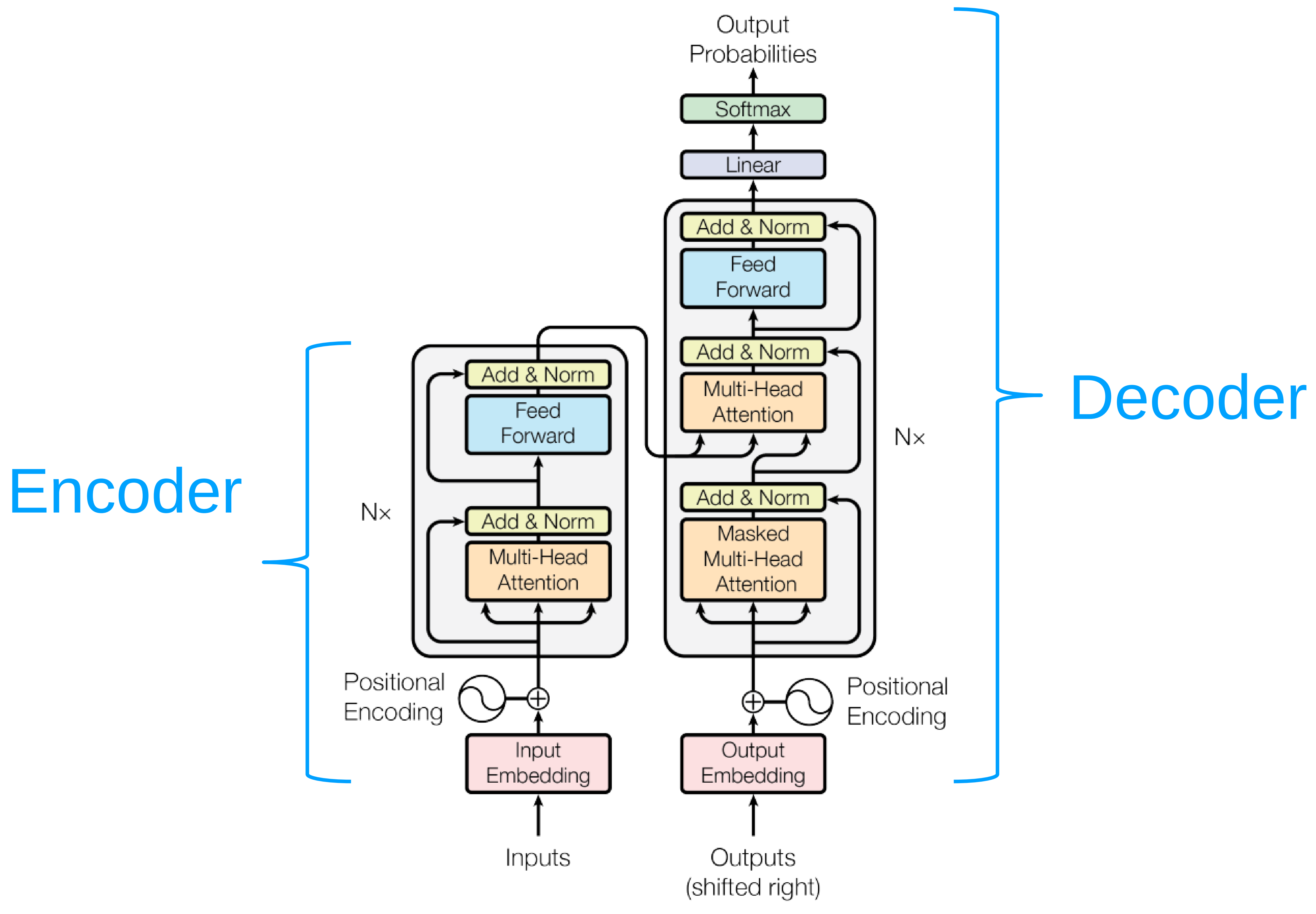


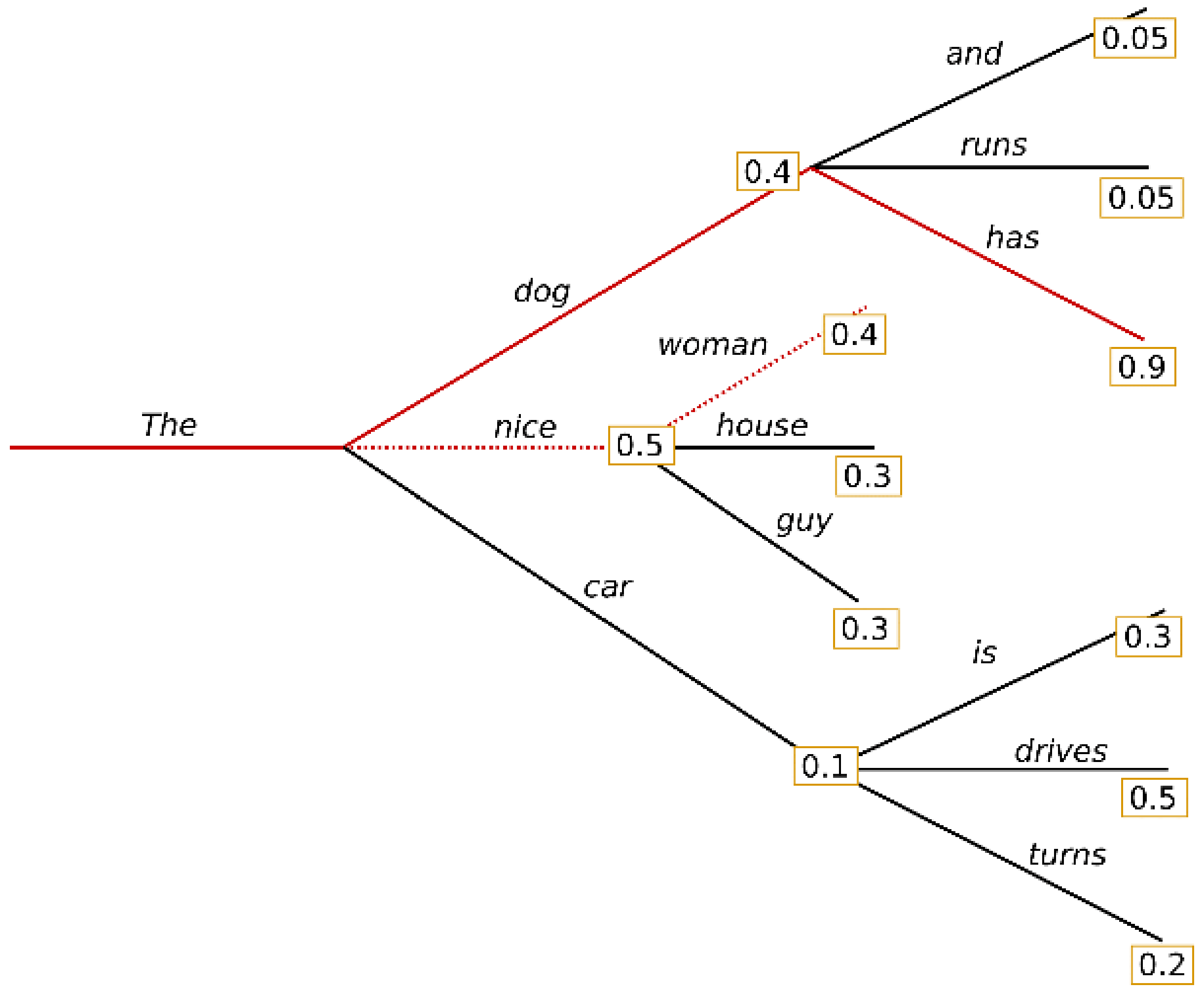
Figure 1: The Transformer - model architecture.

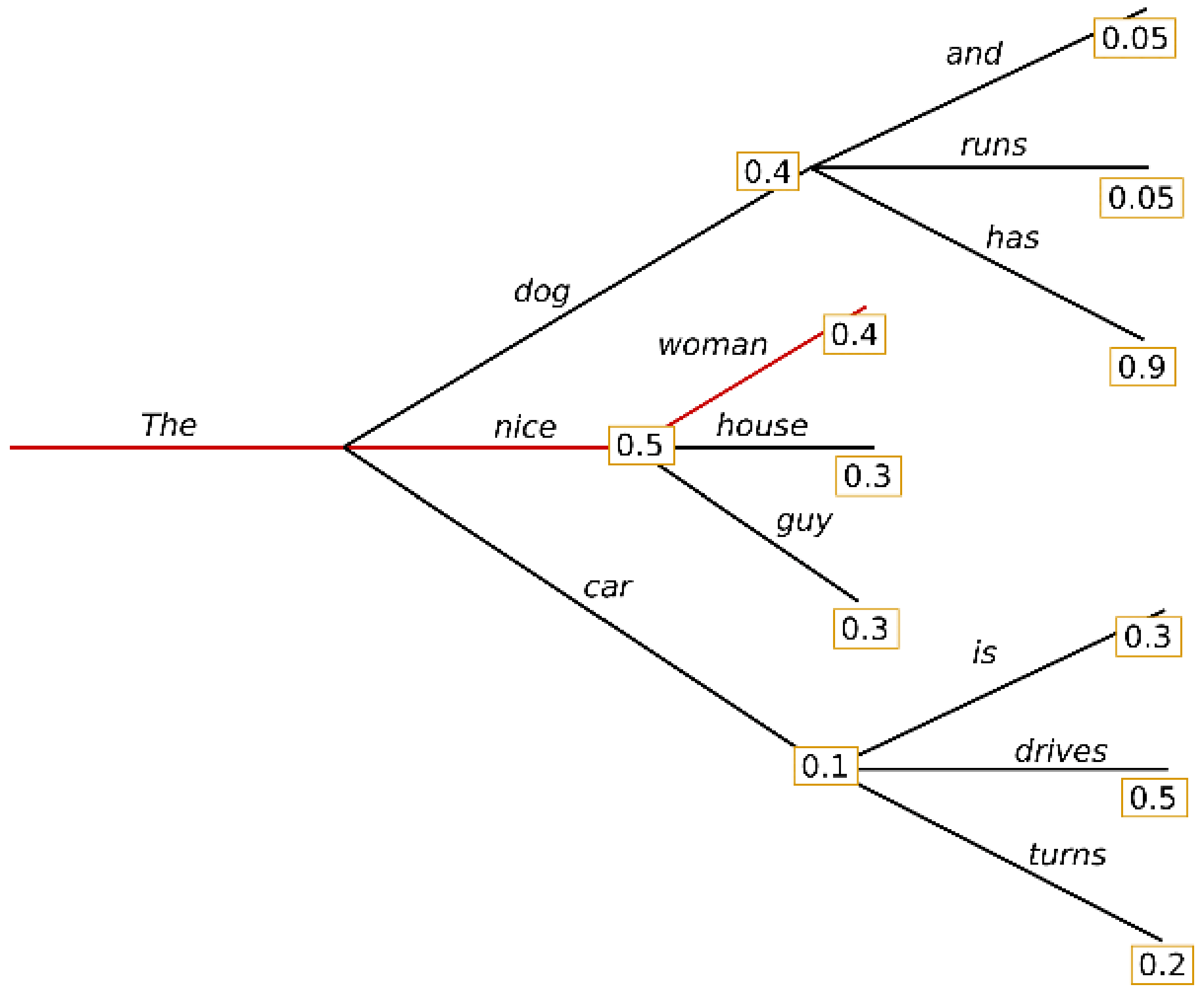
https://nlpprogress.com/english/constituency_parsing.html

Many more fun topics!

- Inference improvements for generative tasks
  - Beam search
- Decoder-only architectures
  - Ex: OpenAI's GPT models
- More natural language processing (NLP) applications
  - Generative text models (aka Chat-GPT)
  - Pretraining/training/fine-tuning strategies
- ...if we have time post-midterm, we'll revisit this!

(unused) Aside: Beam search





https://huggingface.co/blog/how-to-generate