Data 188: Introduction to Deep Learning

Transformers (Part 2)

Speaker: Eric Kim
Lecture 17 (Week 11)
2026-04-02, Spring 2026. UC Berkeley.

Rev02: 2026-04-05 3:45 PM PST
- Clarify right-shift decoder input. We don't pass <END> to decoder input.

Announcements

- Midterm grades out!
  - Regrade requests via Gradescope
    - Last call: Wednesday April 1st, 11:59 PM PST
- HW3 released: "Intro to Pytorch"
  - Tip: the remaining homework assignments in this course will not use Needle (ie your previous HW0 -> HW2).

Today's lecture

Transformers (Part 2!)

Encoders: classification techniques

Decoders

- Cross attention
- Masked self-attention ("causal self attention")

Sequence-to-sequence tasks

- Ex: Machine translation

(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**

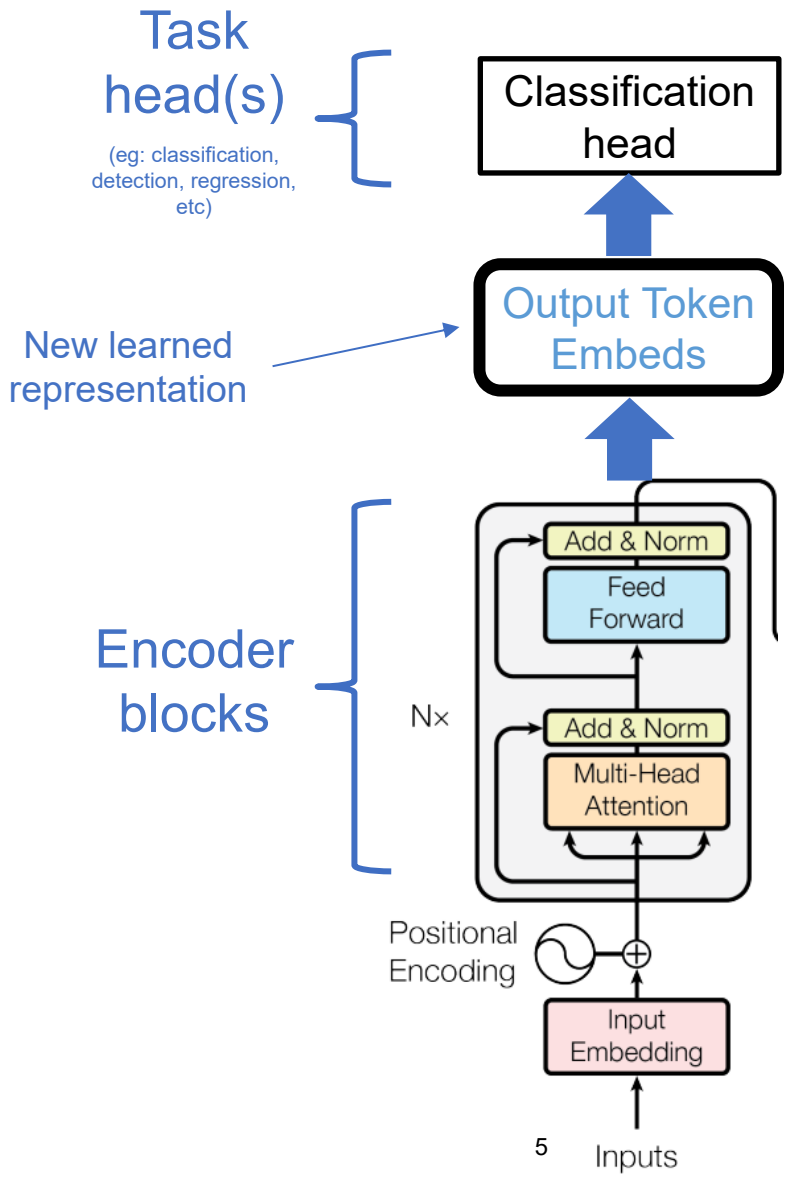
Code | Blame | 6300 lines (5389 loc) · 231 KB ·

```
5878  def multi_head_attention_forward(
6230      # adjust dropout probability
6231      if not training:
6232          dropout_p = 0.0
6233
6234      #
6235      # (deep breath) calculate attention and out projection
6236      #
6237
6238      if need_weights:
6239          B, Nt, E = q.shape
6240          q_scaled = q * math.sqrt(1.0 / float(E))
6241
6242          assert not (
6243              is_causal and attn_mask is None
6244          ), "FIXME: is_causal not implemented for need_weights"
6245
6246          if attn_mask is not None:
6247              attn_output_weights = torch.baddbmm(
6248                  attn_mask, q_scaled, k.transpose(-2, -1)
6249              )
```

Encoder: Classification?

Recall: a Transformer encoder performs the following:

- Input: sequence X (shape=[batchsize, n, d])
- Output: representation Y (shape=[batchsize, n, d])
- Where Y is a learned transformation of X, via multi-head self attention (MHA), MLP's
- 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

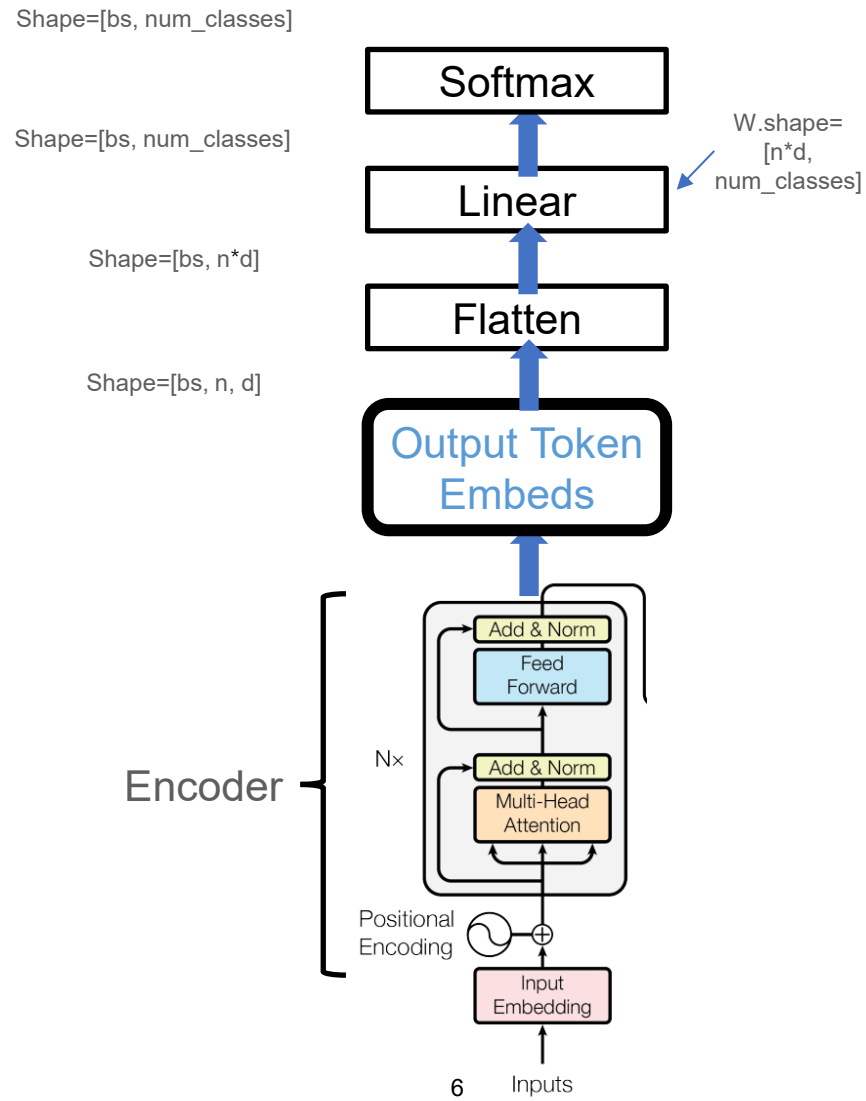
Proposal: flatten the Y from [batchsize, n, d] to [batchsize, n*d], and add a Linear(in=n*d, out=num_classes) layer after the Encoder output.

Question: what are the pros/cons of this?

Pro: Simple

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

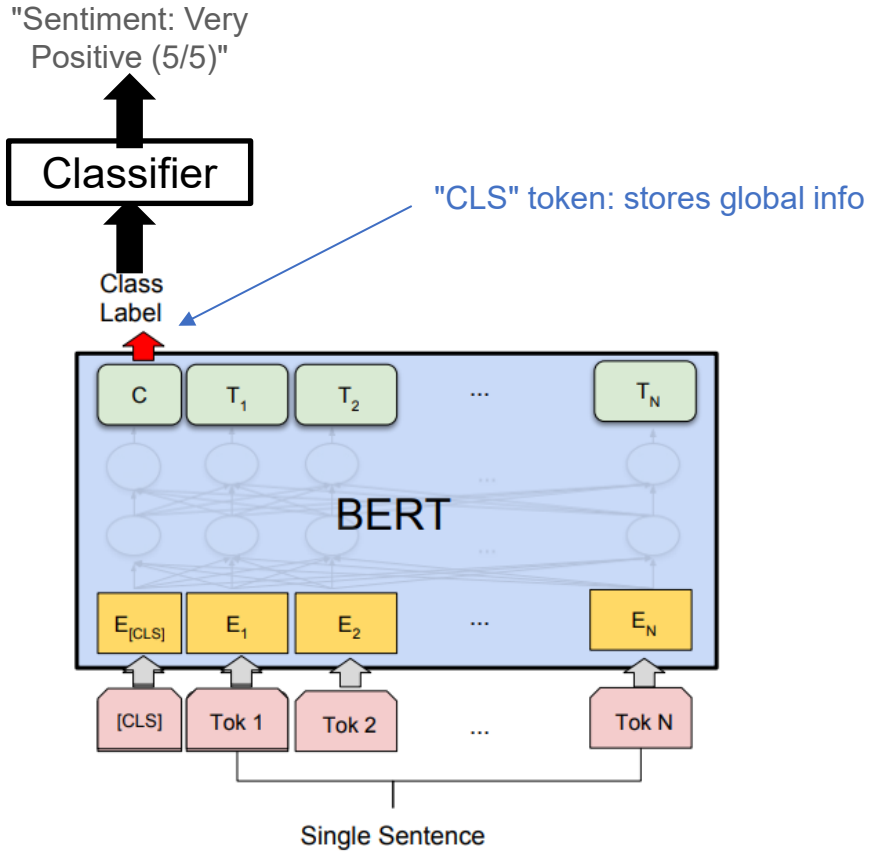
Can be computationally expensive: for long sequence lengths and large number of target classes, the Linear layer can become too large



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

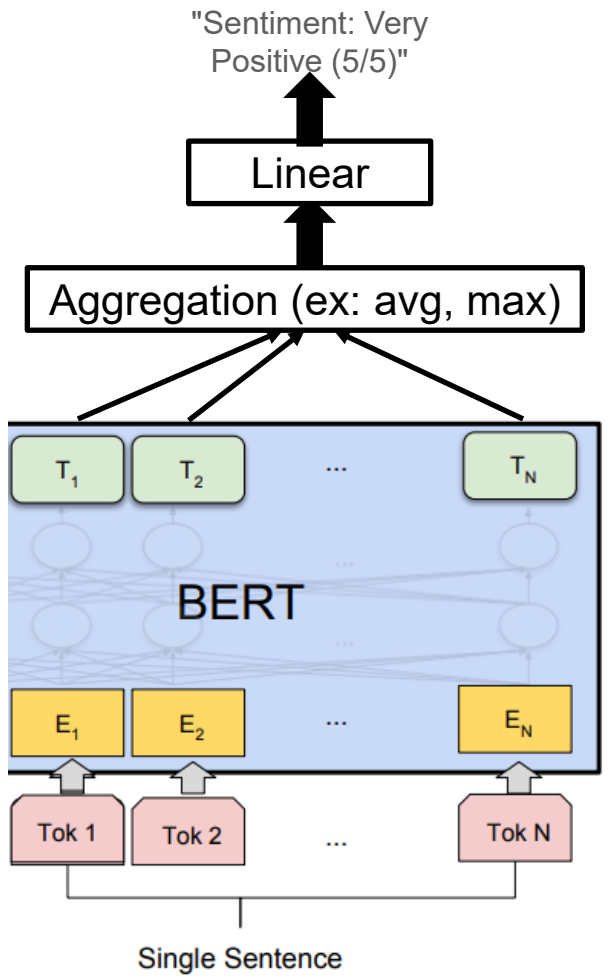
Classification approach 2: Token aggregation

Key idea: aggregate the `n` output tokens into a single output embedding, then add your classifier on top of this

- Ex: average, max

After aggregation: [bs, d]

Before aggregation: [bs, n, d]



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

Answer: [d, num_classes]

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

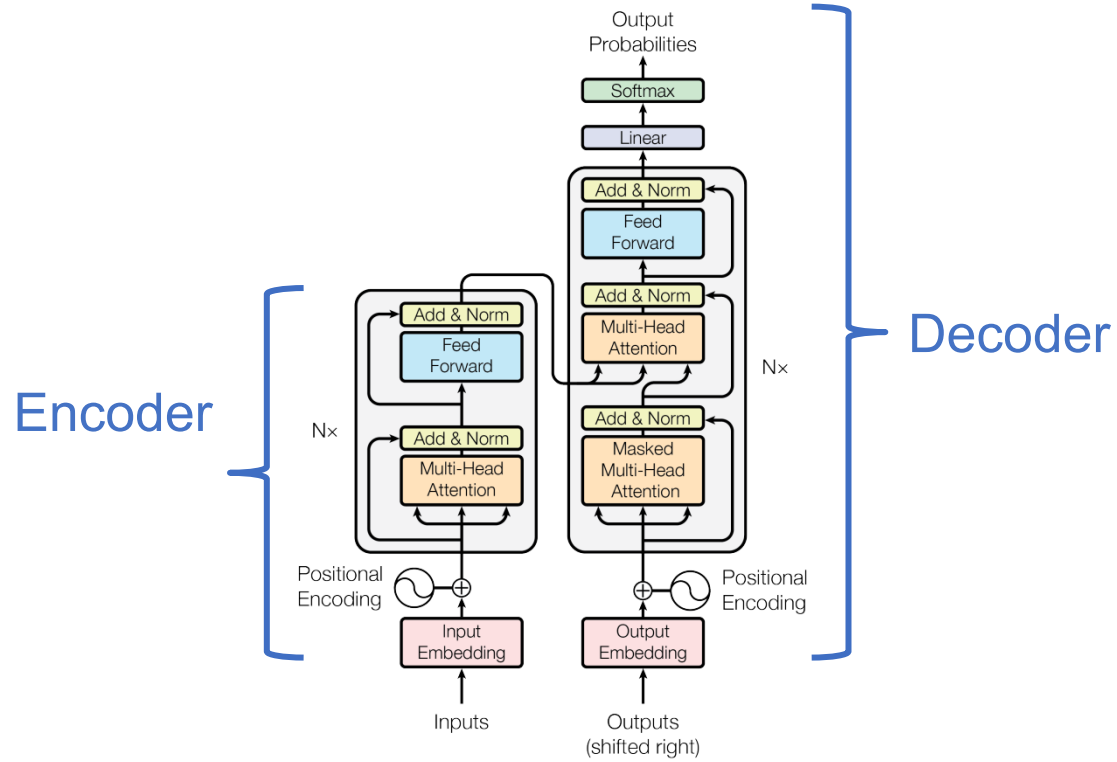


Figure 1: The Transformer - model architecture.

Cross attention

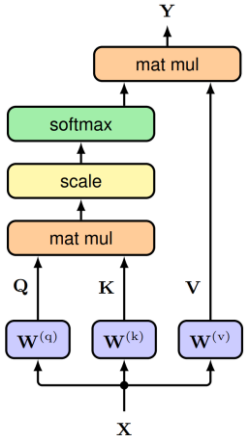
Recall: in the Encoder's multi-head attention (MHA), we had only one input sequence, aka **"Self attention"**

"Cross attention": MHA but with **two** different input sequences: X_{tgt}, X_{src}

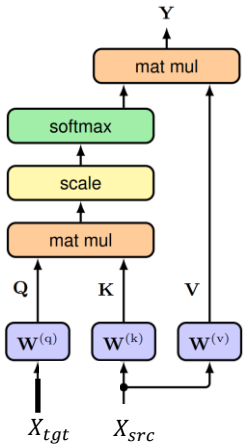
Ex: consider English->French translation, where we have a target sequence X_{tgt} (French) and an input sequence X_{src} (English)

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

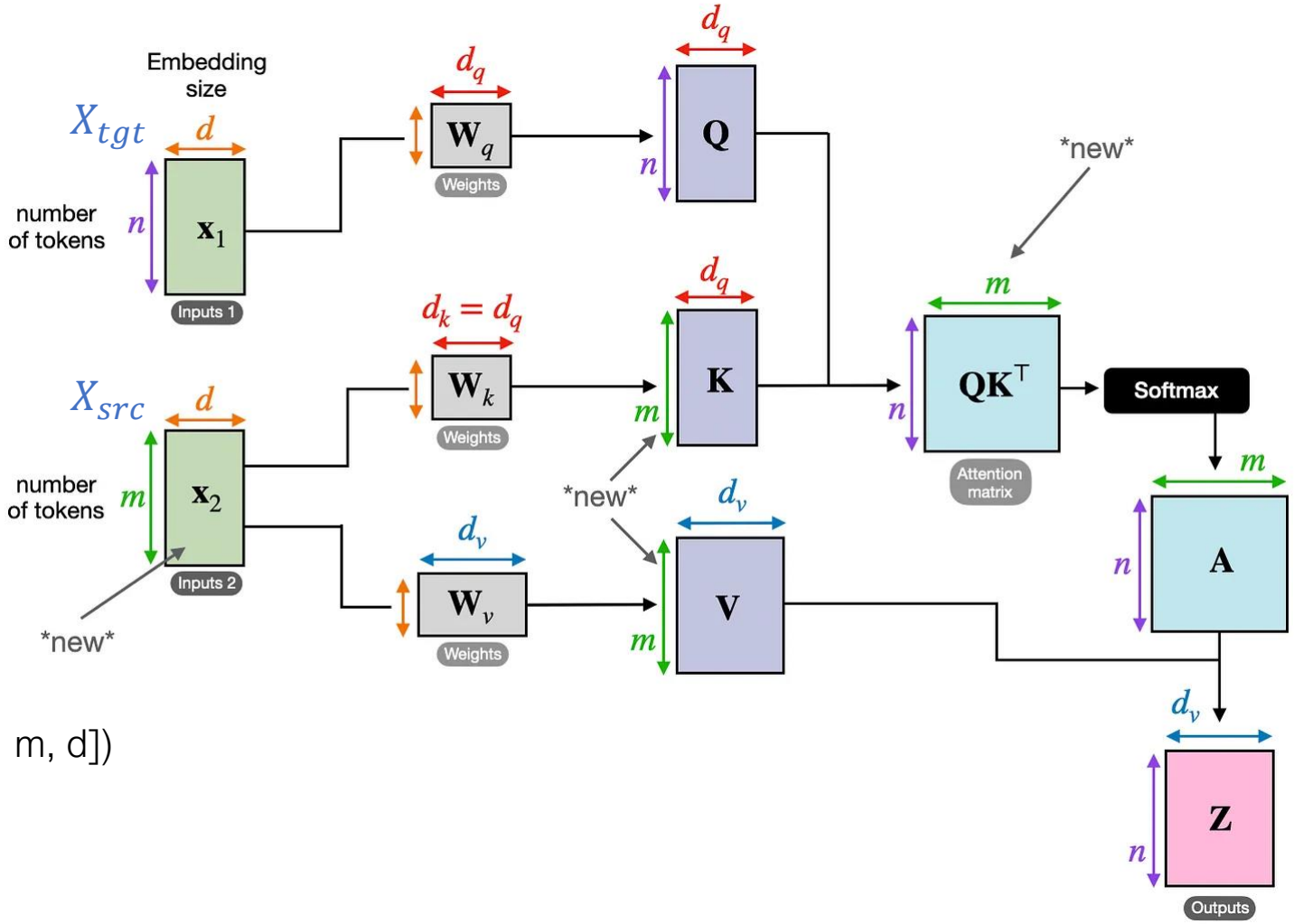


Cross-attention

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

Multi-head cross attention: equations

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

Let `h` be number of heads.

$$Q_h = X_{tgt} W_h^q$$
$$K_h = X_{src} W_h^k$$
$$V_h = X_{src} W_h^v$$

W_h^q, W_h^k, W_h^v shape=[d, d_h]
Q_h shape=[n, d_h]
K_h, V_h shape=[m, d_h]

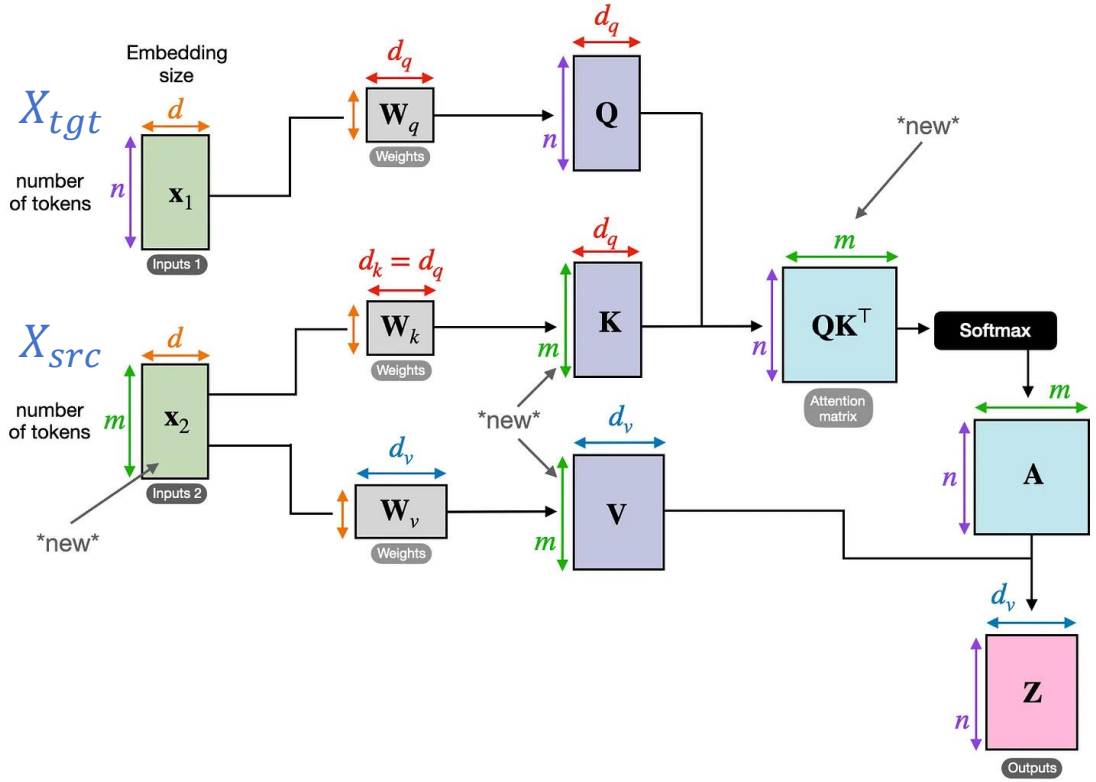
$$d_h = floor(\frac{d}{h})$$

H shape=[n, d_h]

$$H_h = attention(Q_{,h}\ K_h, V_h) = softmax\left(\frac{Q_h K_h^T}{\sqrt{d_h}}\right) V_h$$

$$\mathrm{Y} = Concat(H_1, \ldots, H_h) W^o$$

W^o shape=[d, d]
Y.shape=[n, d]



Can do **multi-head** cross-attention in the same way as self-attention: split up Q, K, V in the `d` dimension. Assumes that both X_{tgt} and X_{src} have the same embed dim!

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

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)

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}	"I"	"ate"	"breakfast"	"already"
"j'ai"	0.8	0.2	0.1	0.0
"déjà"	0.1	0.2	0.1	0.6
"pris"	0.1	0.7	0.2	0.1
"le"	0.0	0.2	0.8	0.0
"petit"	0.0	0.1	0.9	0.0
"déjeuner"	0.0	0.1	0.9	0.0

"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"]
=> [TOKEN_START, "hello", "there", 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 (ie text never seen before in training)
- <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)
```

```
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])
```

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

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

dim_embed: 768

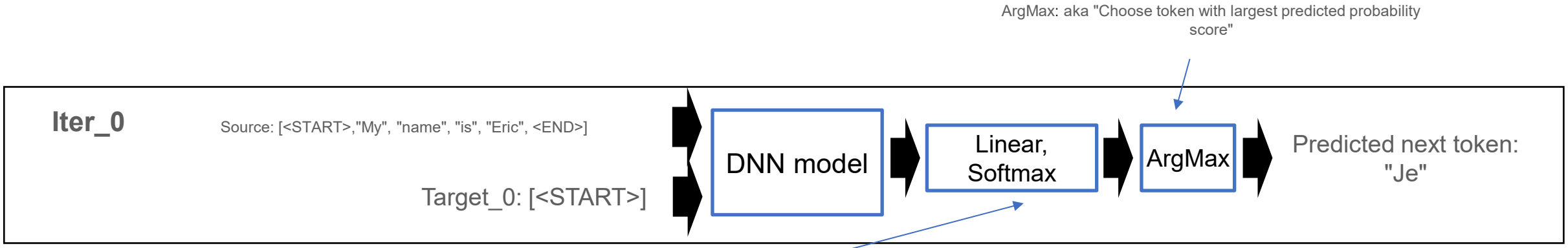
https://huggingface.co/google-bert/bert-base-uncased

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



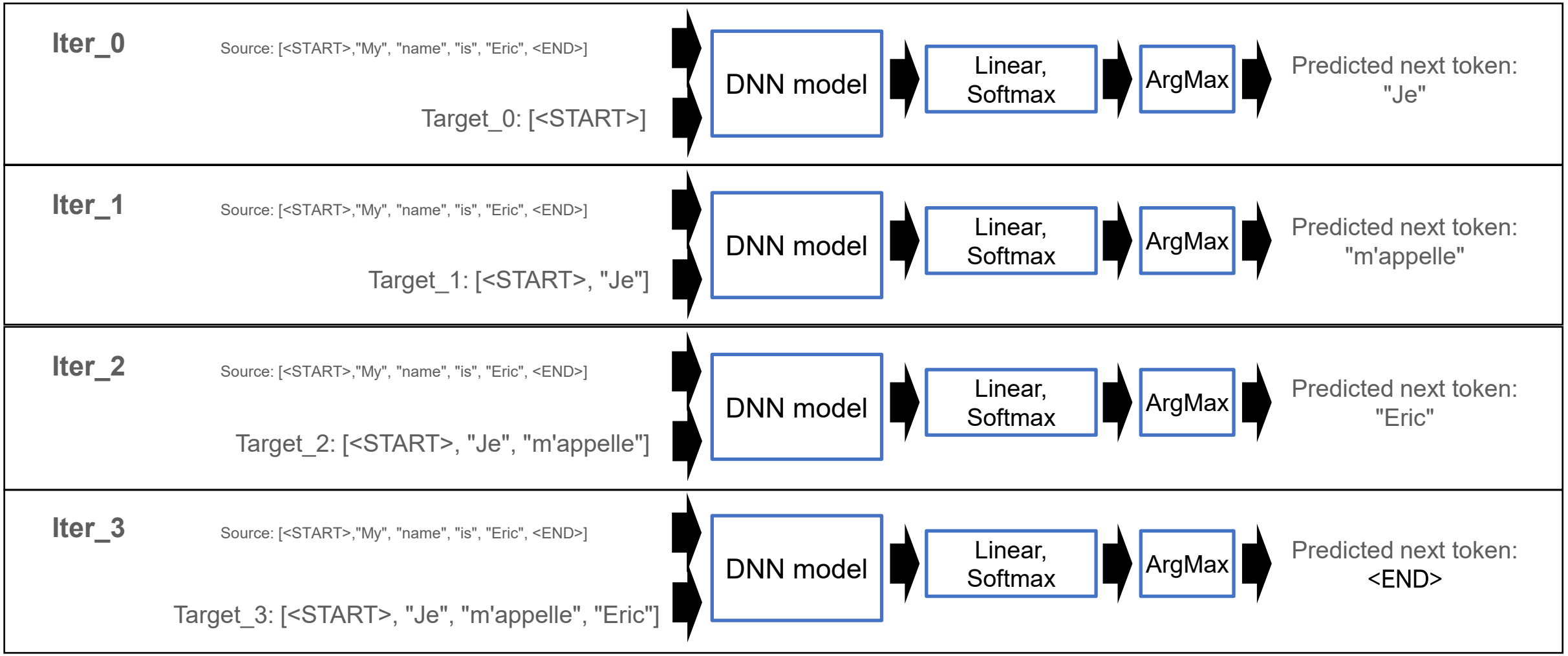
Note: "Linear, Softmax" produces a probability score vector over the tokenizer vocabulary size. Ex: Bert tokenizer has ~30k tokens in its vocabulary [link]

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
...
Token29999 ("!"): 0.0
```

Translation: next token prediction (autoregressive inference)

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

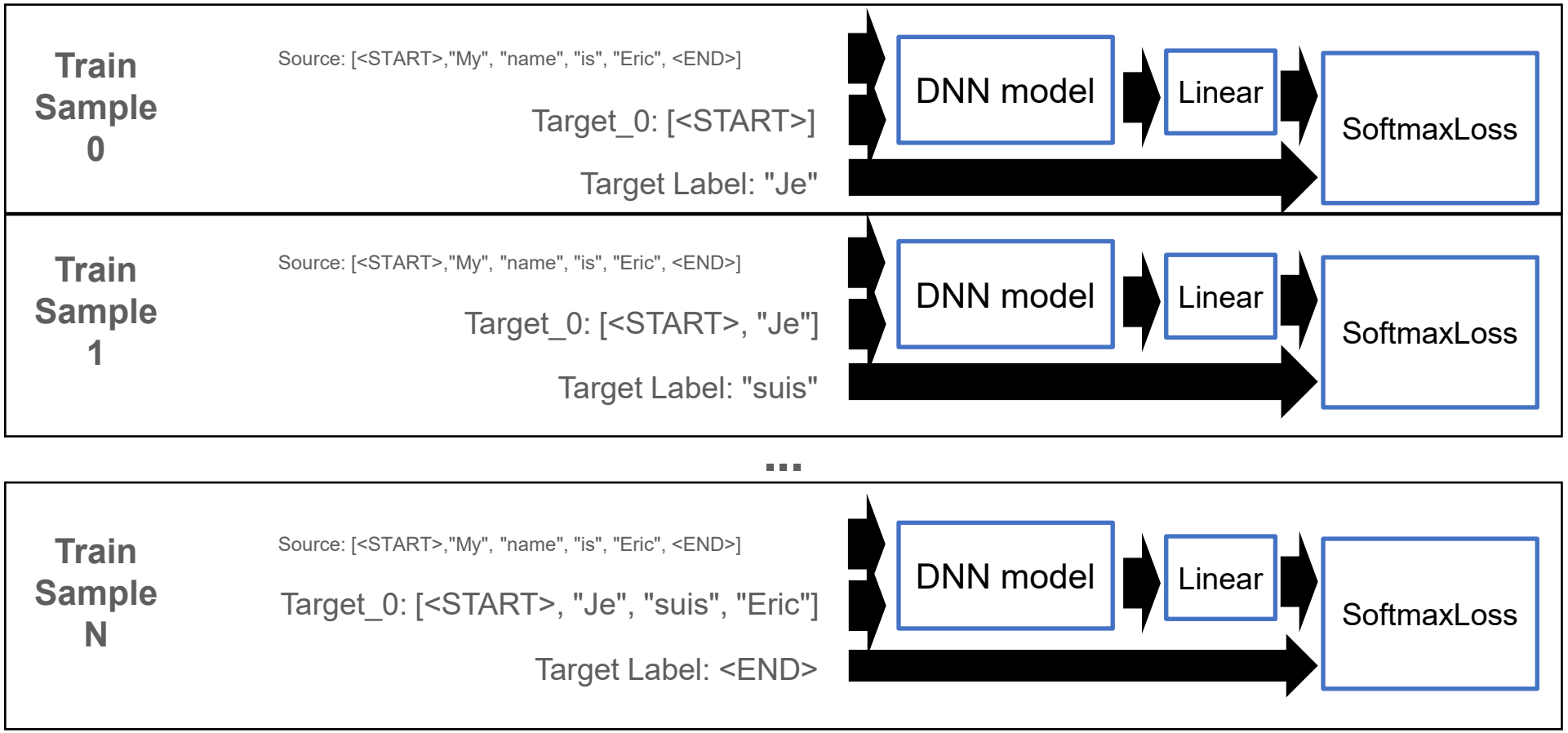


Output: [<START>, "Je", "m'appelle", "Eric", <END>]

Translation loss

How to build a training loss out of this idea?

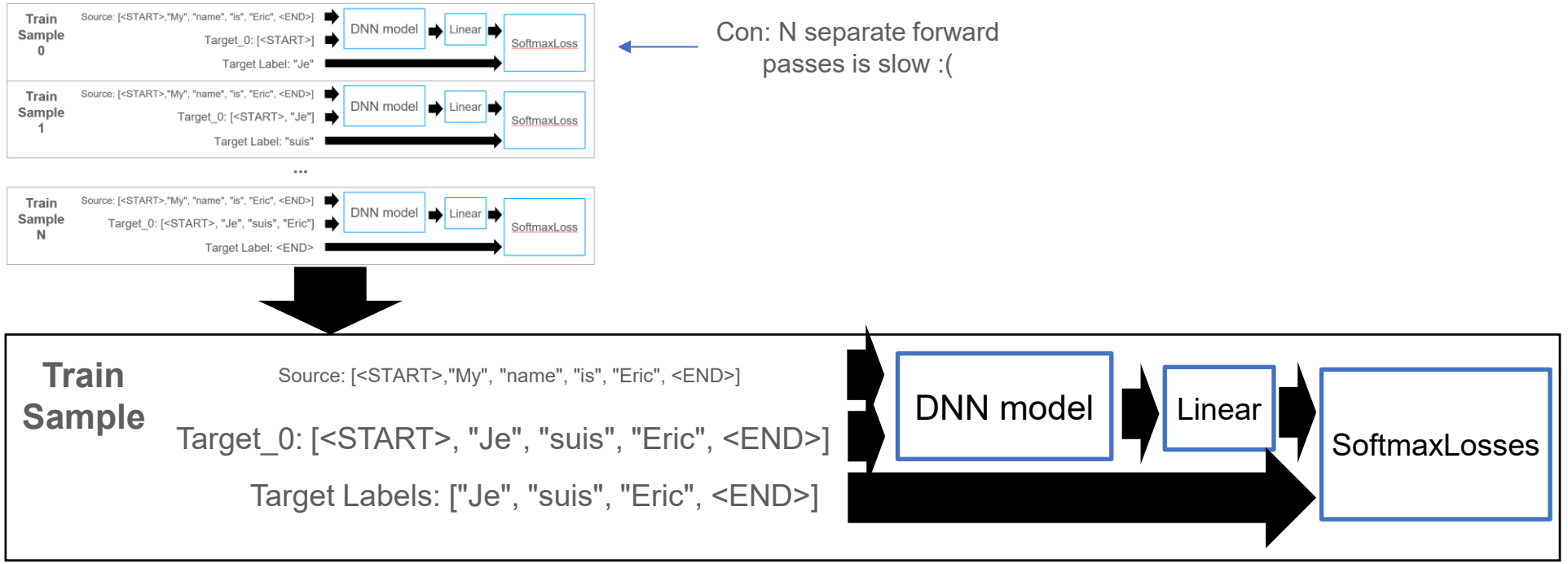
Answer: all-possible next-token prediction tasks (classification loss)!



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

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

ind=0, target label: 42

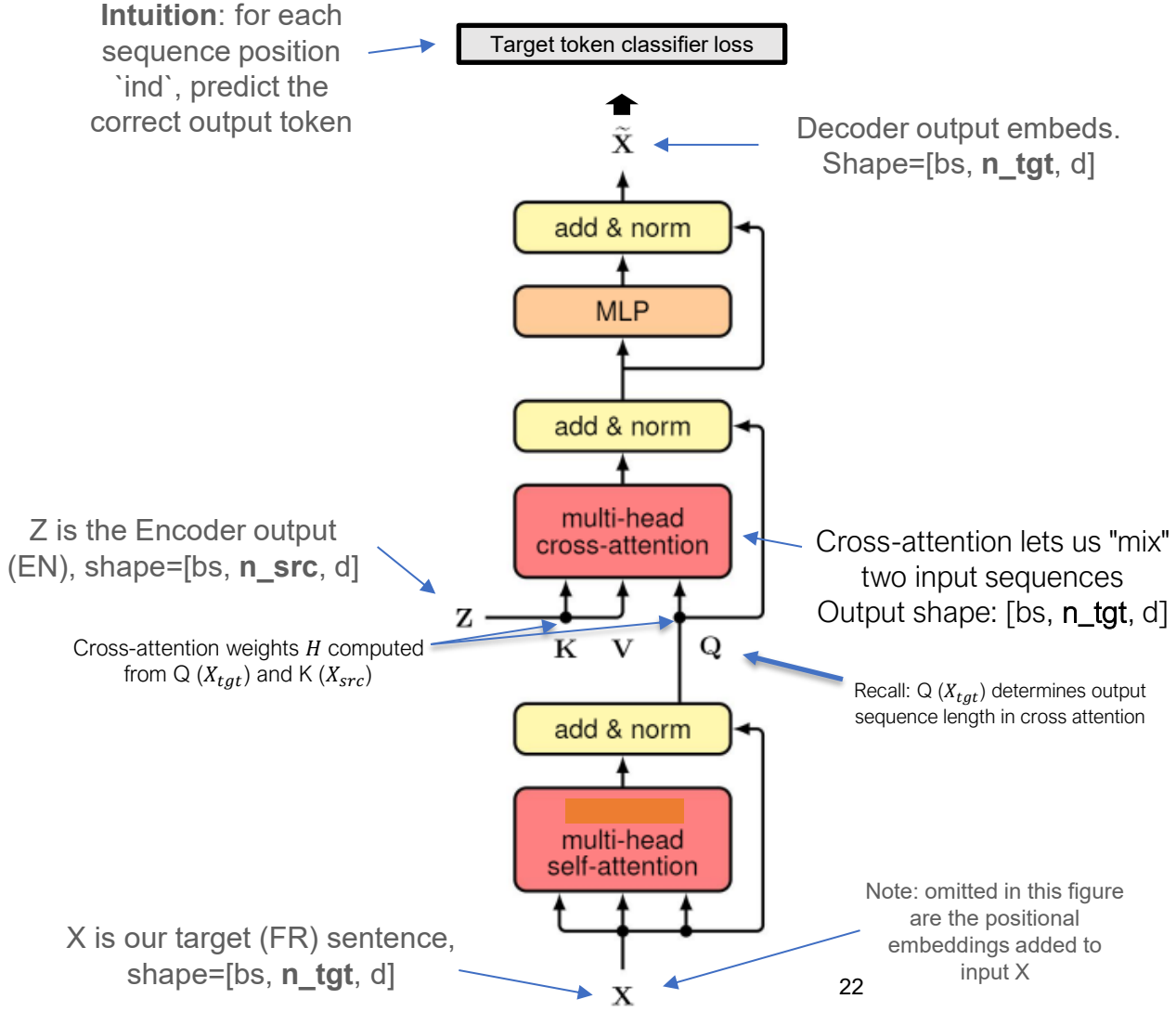
```
# Token, predicted_probability
Token0 ("eau"): 0.01
Token1 ("boulangerie"): 0.02
...
Token9001 ("suis"): 0.75
...
```

ind=1, target label: 9001

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)



Translation loss: Issue?

```
# 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
...
```

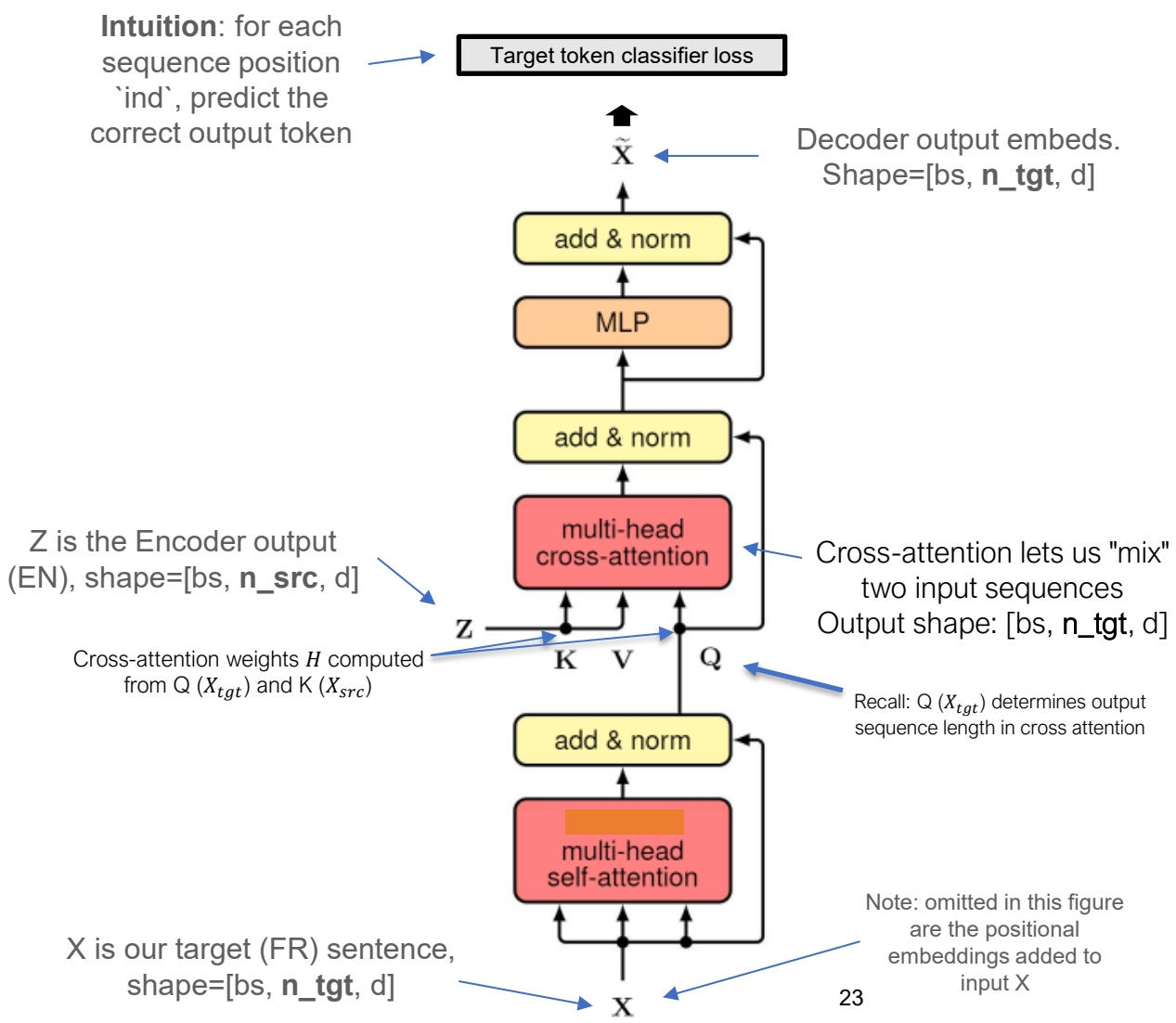
ind=1, target label: 9001

Question: from a modeling perspective, why might this setup be suboptimal?

- Hint: information leakage

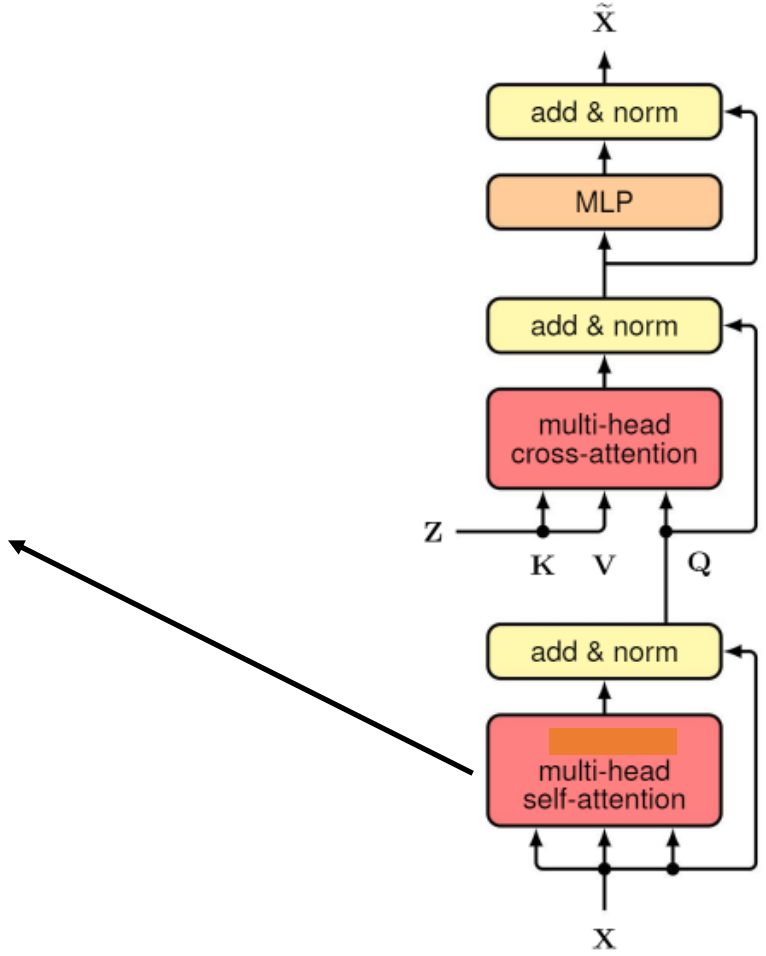
Answer: decoder can "cheat" and use information from later in the sequence when predicting the current token!

- Ex: when predicting the "suis" token, we don't want it to look at the full ground truth sentence "je suis Eric" to trivially copy the answer.
- A prediction for sequence position `ind` should only use information before `ind` ("causality")



Attention scores (no causal mask)

	<START>	"je"	"suis"	"Eric"
<START>	0.8	0.2	0.1	0.1
"je"	0.0	0.1	0.9	0.3
"suis"	0.1	0.2	0.1	0.6
"Eric"	0.1	0.7	0.2	0.1

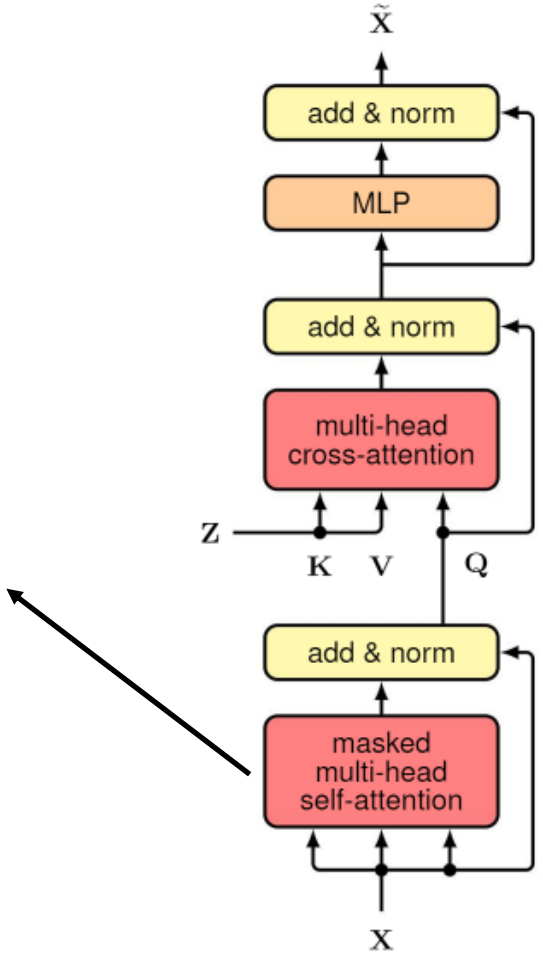


Issue: when predicting the first token "je", the decoder can utilize information from the rest of the sequence, which can result in cheating (ex: trivial copy).
Solution: apply a "look ahead" mask to the decoder's self-attention weights

Solution: attention scores (with causal mask)

	<START>	"je"	"suis"	"Eric"
<START>	0.8	-inf	-inf	-inf
"je"	0.0	0.1	-inf	-inf
"suis"	0.1	0.2	0.1	-inf
"Eric"	0.1	0.7	0.2	0.1

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



Masked Attention scores (post softmax)

Note: softmax values here aren't accurate, just for illustration

	<START>	"je"	"suis"	"Eric"
<START>	0.8	-inf	-inf	-inf
"je"	0.0	0.1	-inf	-inf
"suis"	0.1	0.2	0.1	-inf
"Eric"	0.1	0.7	0.2	0.1

Softmax (along rows)

	<START>	"je"	"suis"	"Eric"
<START>	1.0	0	0	0
"je"	0.1	0.9	0	0
"suis"	0.1	0.8	0.1	0
"Eric"	0.1	0.7	0.2	0.1

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

Now, the decoder can't "cheat" by looking ahead.

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"

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

Ex: for n_tgt=4, d=2

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

H V Y

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

Note that the 1st row of Y only depends on the first row of V, the 2nd row of Y only depends on the first two rows of V, etc.
Thus, we achieved our goal: the Y 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:

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

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

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!

```
# 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
...
```

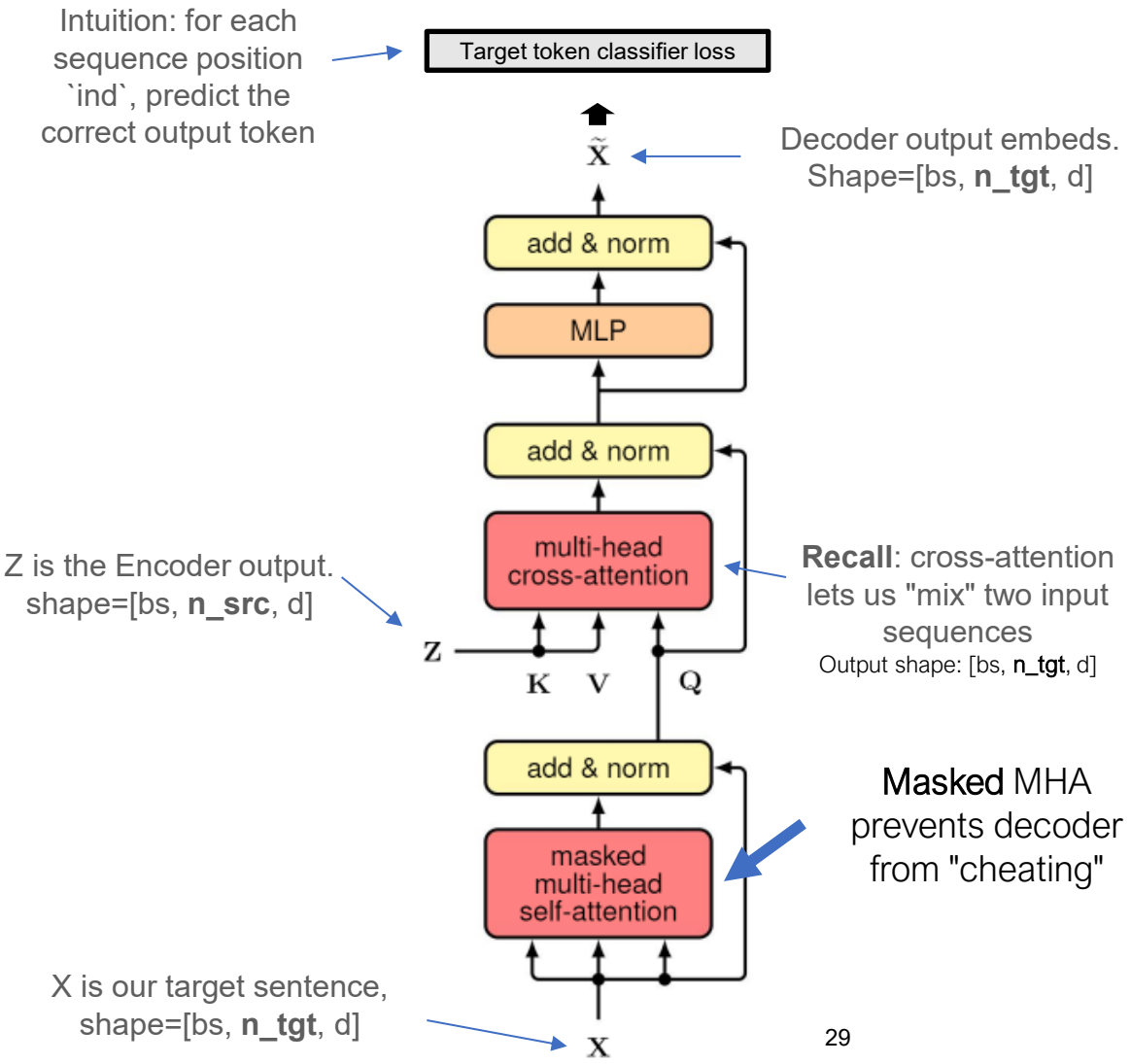
ind=1, target label: 9001

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



Translation loss: next token prediction

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

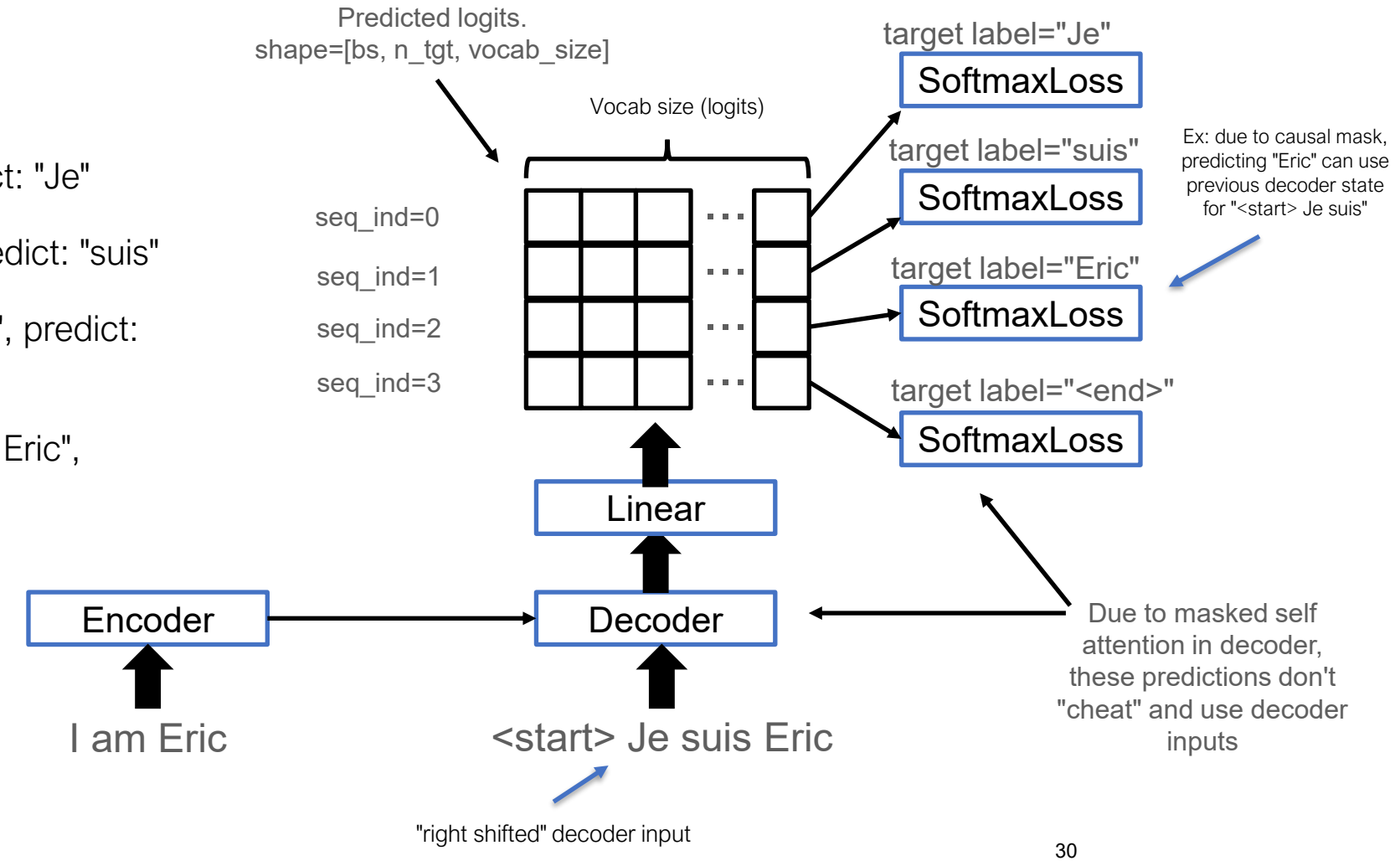
src="I am Eric"

Task1: Given src and tgt="<START>", predict: "Je"

Task2: Given src and tgt="<START> Je", predict: "suis"

Task3: Given src and tgt="<START> Je suis", predict: "Eric"

Task4: Given src and tgt="<START> Je suis Eric", predict: "<END>"

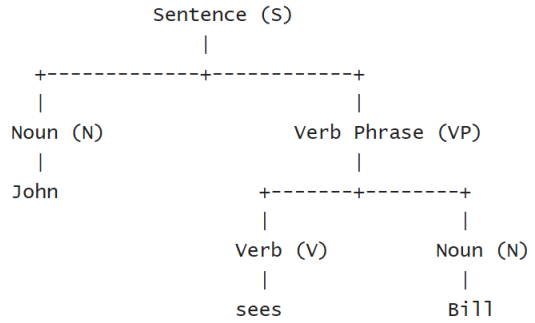


Encoder-Decoder models

The "Attention Is All You Need" paper [link]. Sequence-to-sequence model.

Tasks

- English->German, English->French translation
- "English constituency parsing"
  - Aka: Parse a sentence into a subject/verb/noun parse tree



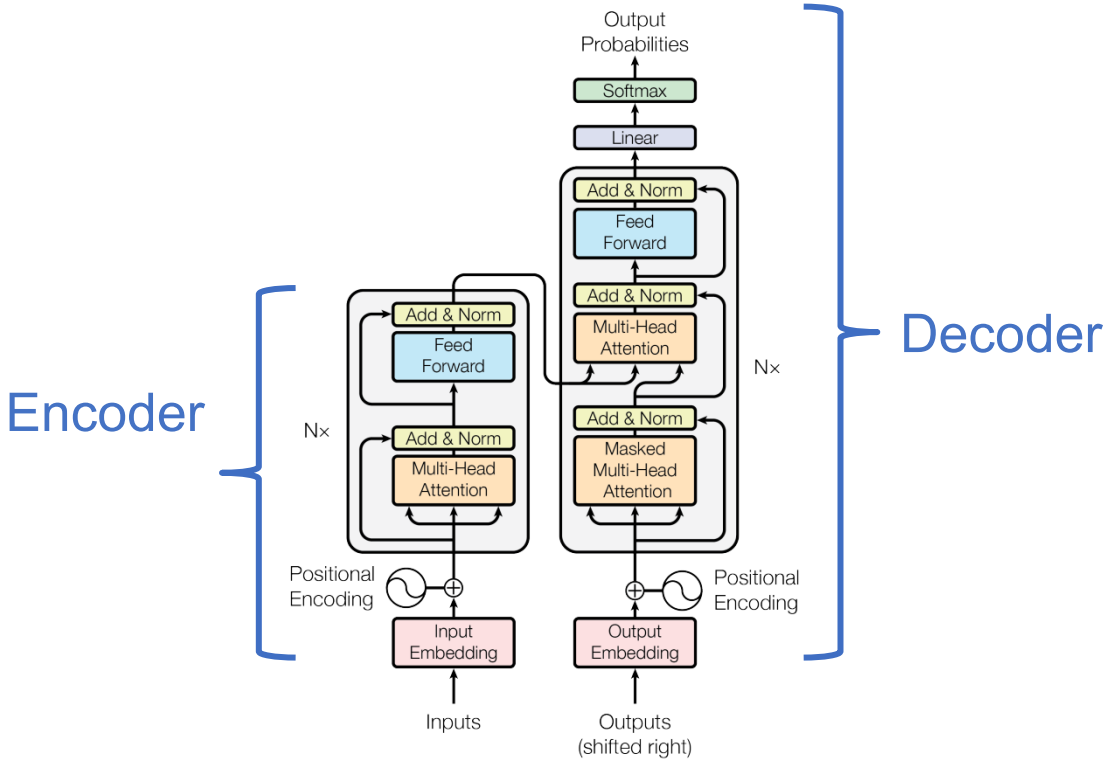


Figure 1: The Transformer - model architecture.

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

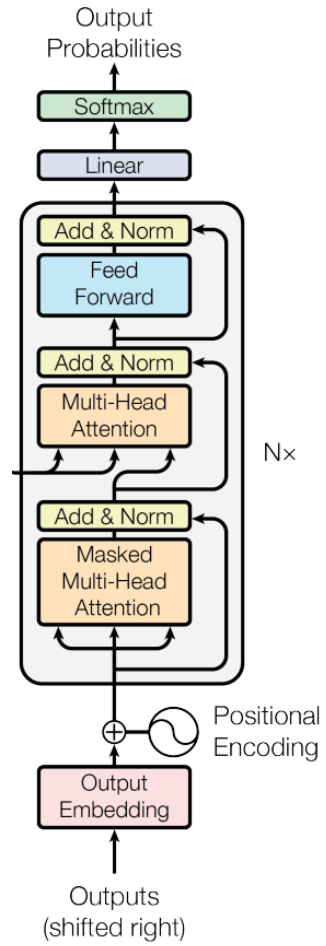
Decoder-only models

Decoder-only models are popular for generative text models, like "chatbot" LLMs (ChatGPT, etc)

Task: next-token prediction

Training data: scrape the Internet for text (ex: Wikipedia, Reddit, etc)

No need for encoder, since for generative text there is only one sequence: no need to do cross-attention between a source sequence and target sequence.



Summary: Transformer model types

Encoder only. Useful for learning a good token representation useful for downstream tasks, like classification (eg BERT).

Encoder+Decoder. Useful for sequence-to-sequence tasks, such as machine translation (ex: "English -> French").

Decoder only. Useful for generative tasks, like text generation (eg ChatGPT).

Many more fun topics!

Inference improvements for generative tasks

- Beam search
- Inject randomness into generation via sampling (rather than deterministic argmax)

More natural language processing (NLP) applications

- Generative text models (aka Chat-GPT)
- Pretraining/training/fine-tuning strategies