Kempner INSTITUTE

HARVARD UNIVERSITY

Building a Transformer from Scratch

November 12, 2024

Objectives

By the end of this workshop, you will be able to:

- Explain the components of a standard decoder-only transformer
- Code and train a language model from scratch

Agenda

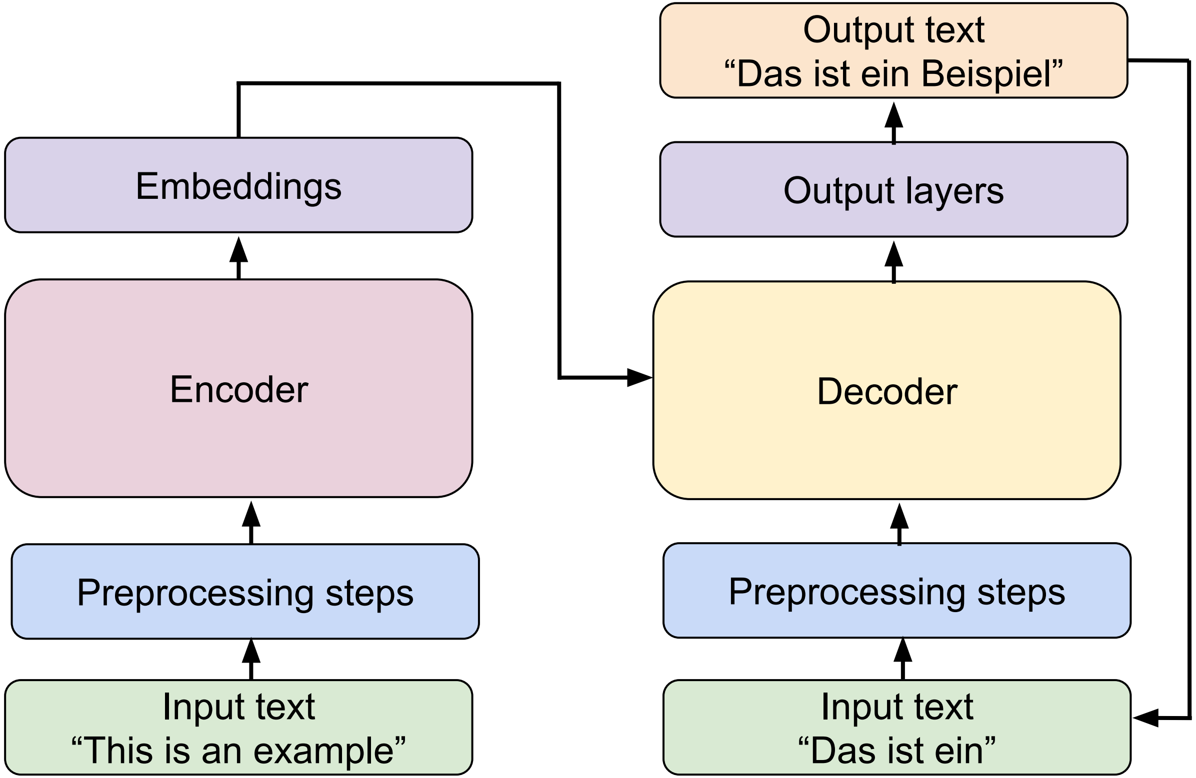
1 Introduction to Transformers

2 Building a Transformer
- Tokenization
- Embeddings
- Attention
- Full Decoder

3 Training a Transformer

Original Transformer Architecture (Simplified)

Understanding of text



Generation of text

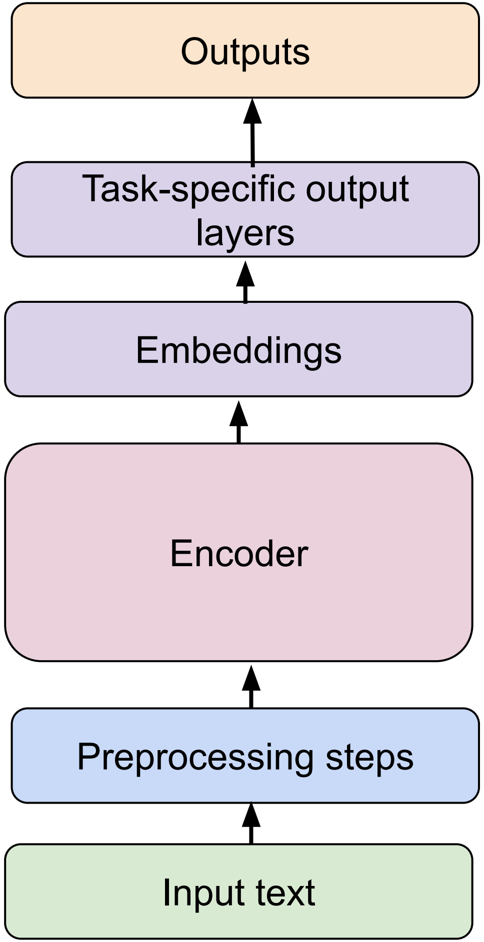
Figure modified from Build a Large Language Model, Raschka, architecture introduced in Vaswani et al, 2017

Do you think an encoder-only, decoder-only, or encoder-decoder transformer architecture would be most suitable for each task?

Task	Example	Suitable Transformer Architecture
Translation	“Rewrite this book in French”	Encoder-Decoder
Sentiment Analysis	“I would marry this vacuum if I could” -> Predict whether positive, negative, or neutral	
Creative Writing	“Tell me a story about a rabbit and a bear being friends”	
Summarization	“Give me a recap on this paper I didn’t have time to read”	

Do you think an encoder-only, decoder-only, or encoder-decoder transformer architecture would be most suitable for each task?

Task	Example	Suitable Transformer Architecture
Translation	“Rewrite this book in French”	Encoder-Decoder
Sentiment Analysis	“I would marry this vacuum if I could” -> Predict whether positive, negative, or neutral	Encoder-only
Creative Writing	“Tell me a story about a rabbit and a bear being friends”	Decoder-only
Summarization	“Give me a recap on this paper I didn’t have time to read”	Encoder-Decoder

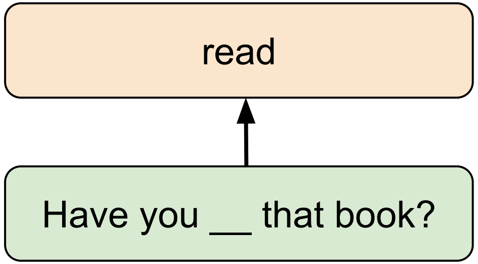


Encoder-only Transformers

Bidirectional understanding of text, good for tasks such as text classification, sentiment analysis, etc

Well-known example is BERT (Bidirectional Encoder Representations from Transformers), introduced by Google in Devlin et al, 2018

BERT was trained on masked word prediction

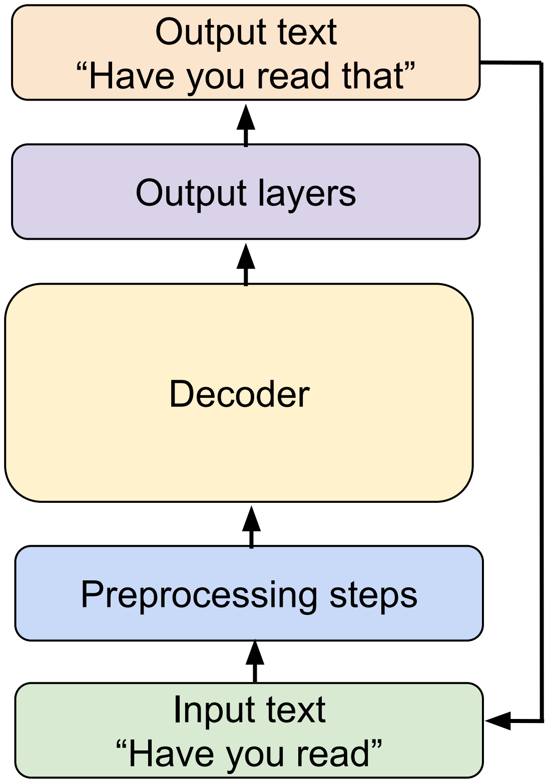


Decoder-only Transformers

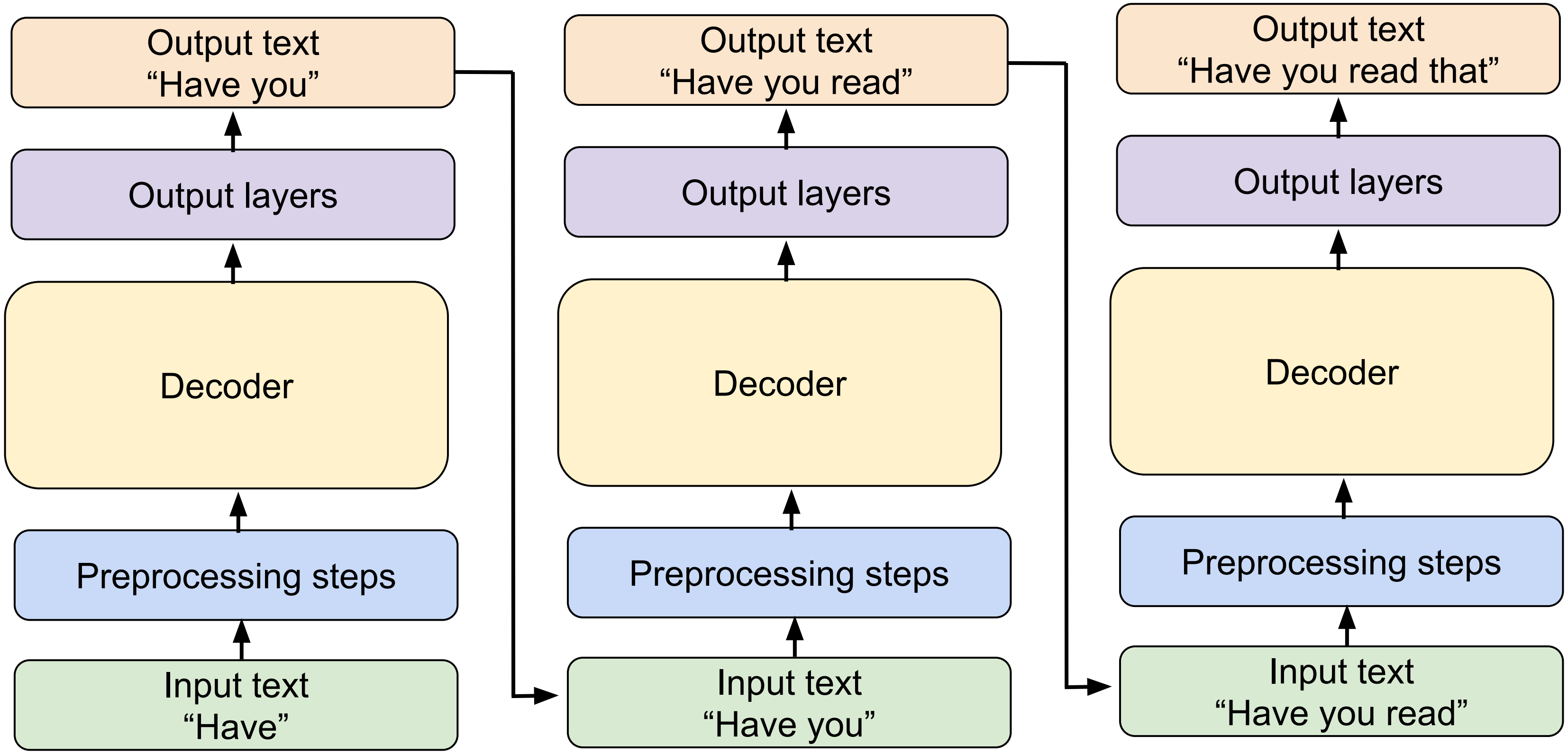
Autoregressive generation of text, good for tasks such as text generation, creative writing. Surprisingly good at translation and summarization

Well-known example is GPT models (Generative Pre-trained Transformers), introduced by OpenAI in Radford et al, 2018

GPT models are trained on next token prediction



Iterative language generation



Many Variants of Decoder-only Transformers

We're going to build a generic decoder-only transformer.

What I cannot build, I do not understand - Richard Feynman

[Minaee et al, 2024](#)

Exercise: Getting set up to build

1) Find the tutorial here

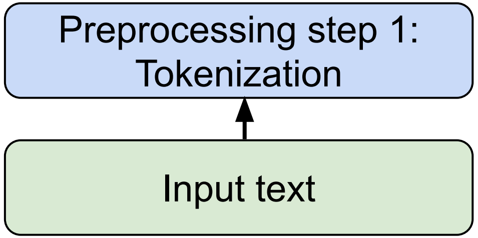
2) Open tutorial:
   a) Can launch in Colab. In this case, you need to download the data `tiny_wikipedia.txt` and upload to Colab workspace

   b) Can clone repo or download file(s) and work locally. You should be working with `transfomers_student.ipynb` file. In this case you need to have Pytorch installed.

Agenda

1. Introduction to Transformers
2. Building a Transformer
   - Tokenization
   - Embeddings
   - Attention
   - Full Decoder
3. Training a Transformer

Building a Decoder-only Transformer from Scratch



Tokenization

100	5	120	88	65	21	22

↑

My	dog	Leo	is	extremely	cute	.

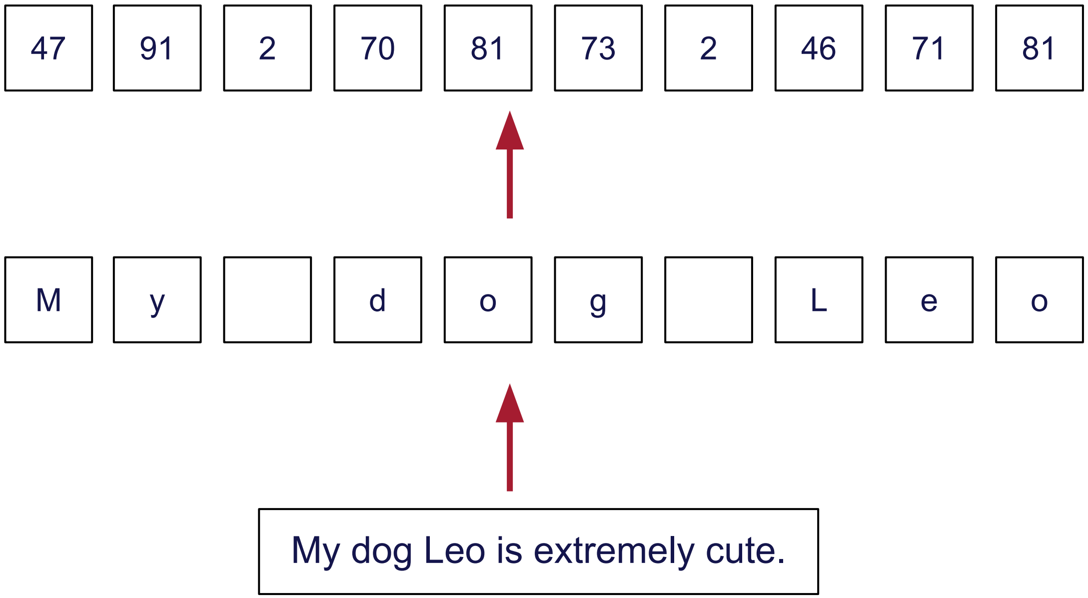
↑

My dog Leo is extremely cute.

2) Convert tokens into token ids

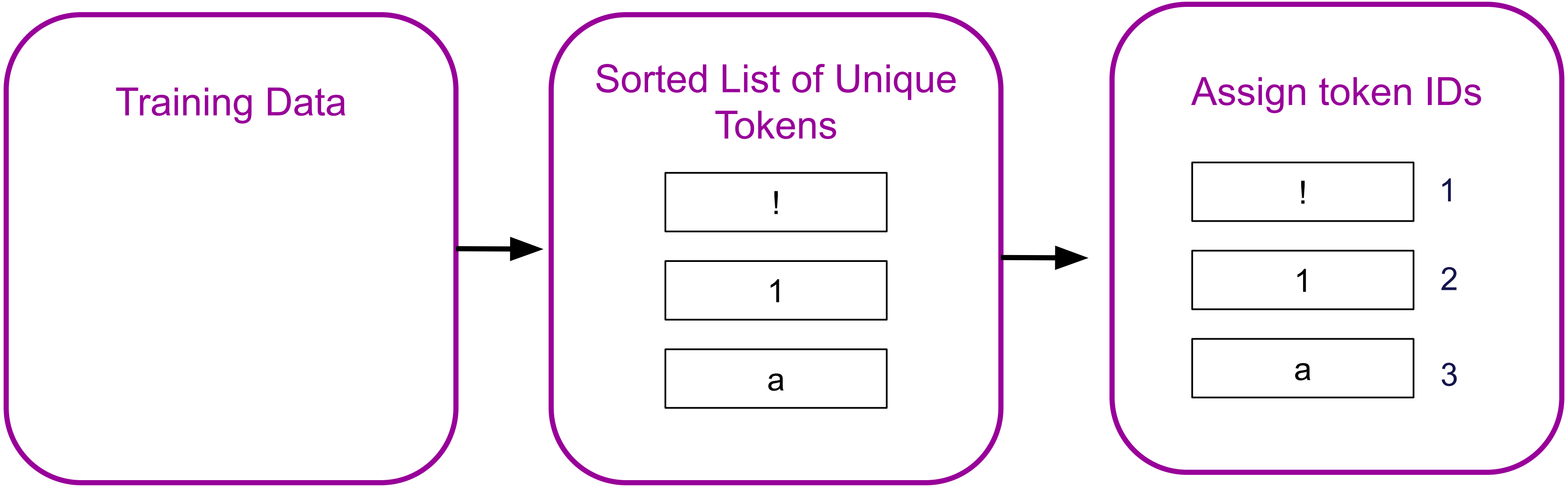
1) Convert sequence into tokens (in this case, words)

Tokenization



Tokens don’t have to be words. You could use character-based (shown to the right), subword-based, or word-based tokenization.

Implementing Tokenization



This forms your vocabulary
(set of unique tokens)

Encode: tokens -> token IDs
Decode: token IDs -> tokens

Implementing Character-based Tokenization

1) Get a sorted list of every unique character in your training data.

2) Create a dictionary that converts tokens to IDs (str_to_int) and one that converts IDs to tokens (int_to_str)

3) Implement functions encode and decode
   a) Encode should take in a string and output list of token IDs
   b) Decode should take in a list of token IDs and output a string

Considerations for type of tokenization

Vocabulary size = number of unique tokens the model recognizes
Sequence length = number of tokens in a given text sequence
Context length = maximum sequence length the model will see

- How will vocabulary size and sequence length vary between character and word-based tokenization?
- What are some potential downsides of character-based tokenization?
- What are some potential downsides of word-based tokenization?

Sub-word tokenization - best of both worlds?

Byte-Pair Encoding

Common algorithm for tokenization. Tokens include whole words, subwords, and characters

Byte-Pair Encoding Algorithm Example

1) Start with individual characters as tokens

t	h	e	t	h	e	a	t	e	r	i	s	l	a	r	g	e	r

2) Find most frequent consecutive pair

t	h	e	t	h	e	a	t	e	r	i	s	l	a	r	g	e	r

3) Merge that pair into one token

th	e	th	e	a	t	e	r	i	s	l	a	r	g	e	r

4) Find most frequent consecutive pair

th	e	th	e	a	t	e	r	i	s	l	a	r	g	e	r

5) Merge that pair into one token

the	the	a	t	e	r	i	s	l	a	r	g	e	r

6) Repeat 2 & 3 up to some specified vocab size

the	the	a	t	er	i	s	l	a	r	g	er

Byte-Pair Encoding

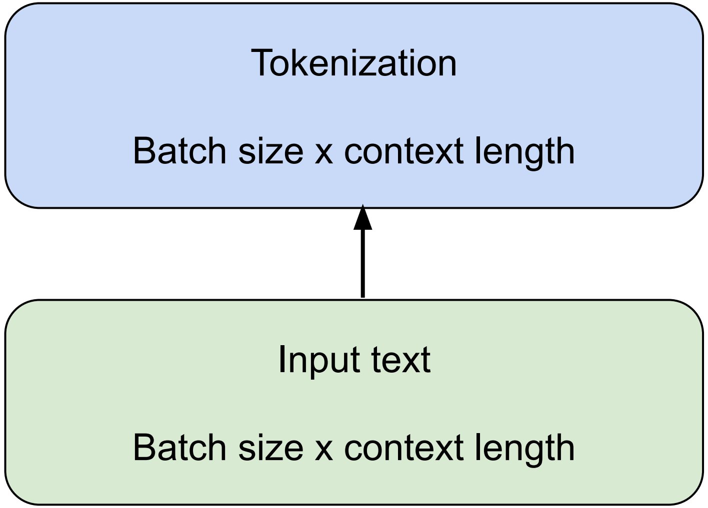
Common algorithm for tokenization. Tokens include whole words, subwords, and characters

Benefits:

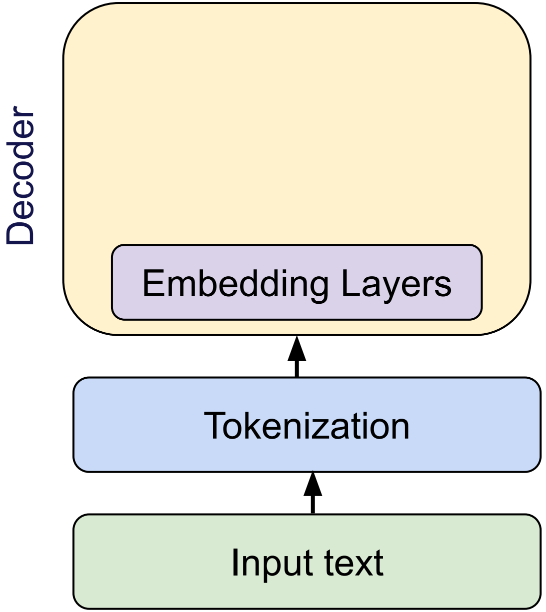
- More manageable vocabulary size than word-based (don't have to include every single word)
- More manageable sequence lengths than character-based
- Can handle new words by breaking them up into subwords

Visualize GPT 4 tokenization

Dimensions of inputs

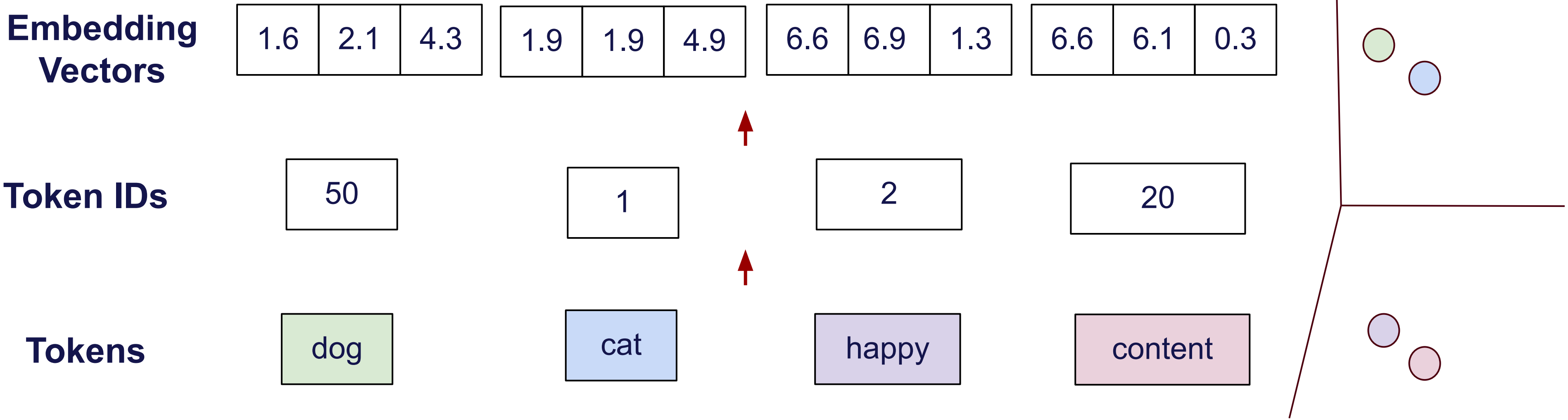


Building a Decoder-only Transformer from Scratch



Token Embedding

elements in vector = embedding dimension of the model (2048 for large GPT-3 model)



Token Embedding Matrix

Embedding Dimensions

Different tokens

4.1	1.9	2.1
5.6	6.1	8.4
1.3	2.2	5.8
3.7	1.0	7.1
2.3	10.1	3.5
2.4	0.9	7.4
4.8	2.2	1.2
9.0	2.1	2.1

Embedding Vector

1.3	2.2	5.8

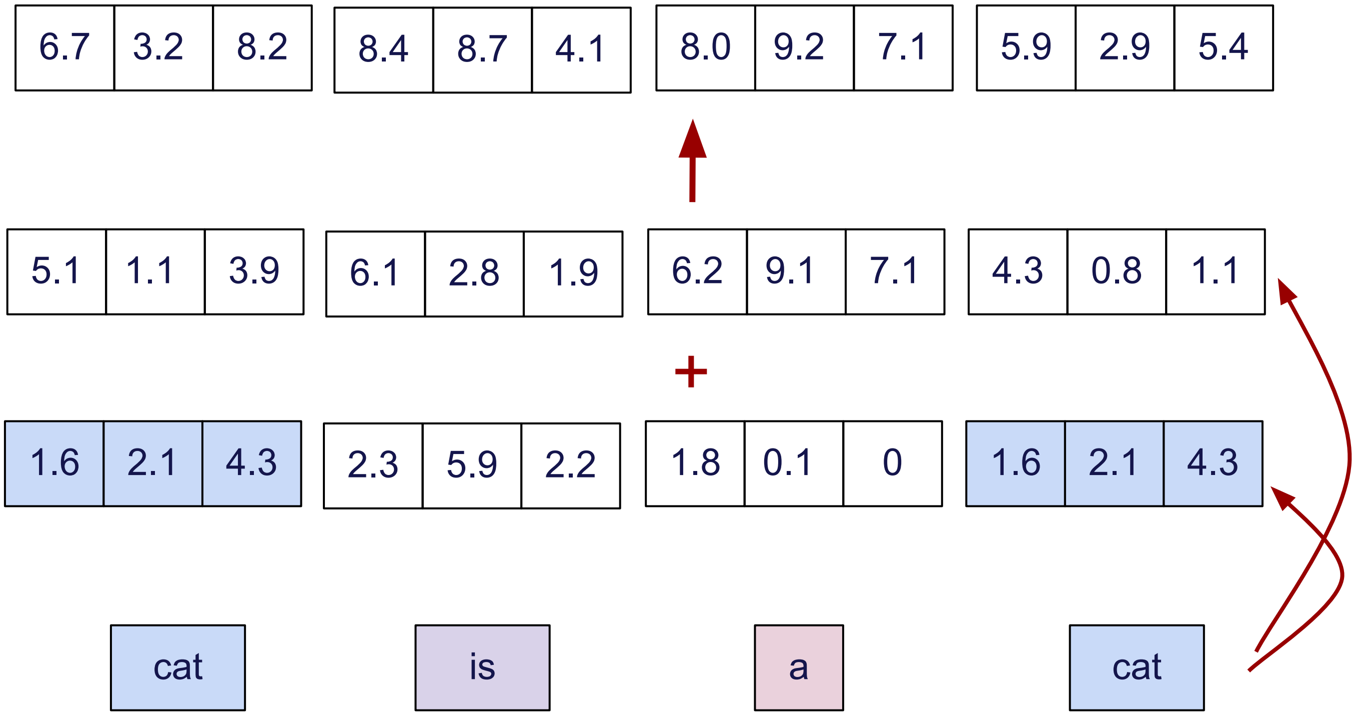
Token ID

2

You can learn the embedding matrix as part of model training

Luckily, Pytorch has an nn.Embedding class we can use

Embedding Layer



Position embeddings defined by absolute position in sequence. Do not take into account token ID. We can learn these embeddings too!

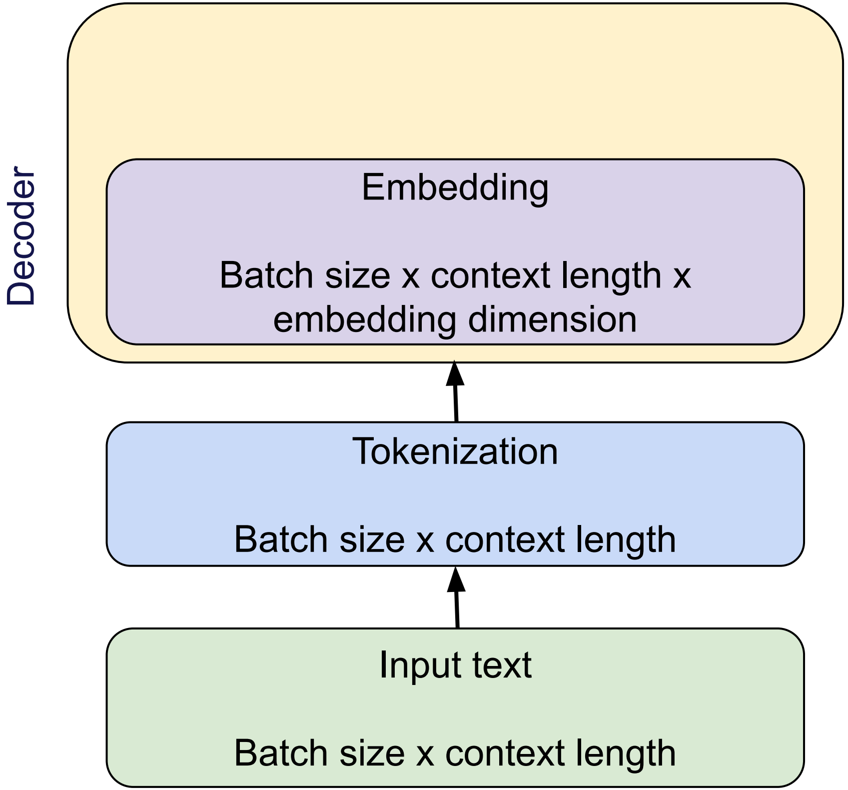
Token embeddings defined by token ID. Do not take into account position.

Exercise: Implementing Embedding Layer

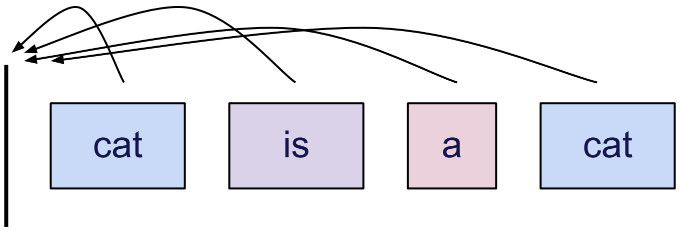
1) Complete TokenEmbeddingLayer class

2) (Advanced): Complete EmbeddingLayer class which should return the final embeddings (token + position). Getting the position embeddings may require a bit of thought first!

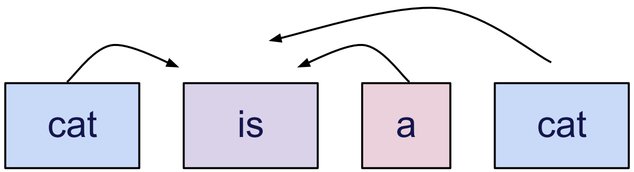
Dimensions of inputs



Absolute vs relative position embedding



Absolute position embedding: what matters is absolute distance from beginning of sequence



Relative position embedding: what matters is relative distance between tokens

Attention

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

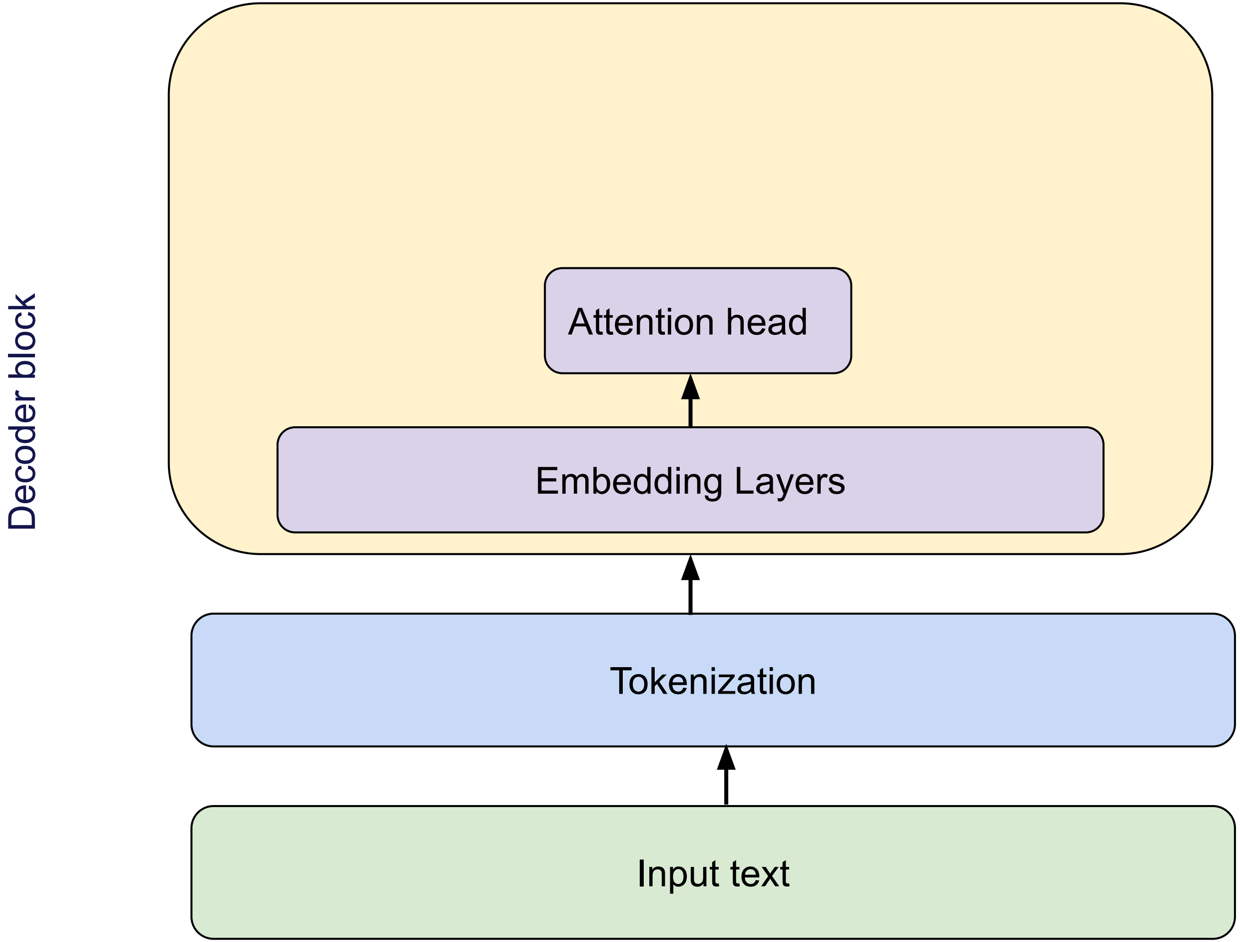
Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com


Decoder block
Attention head
Embedding Layers
Tokenization
Input text

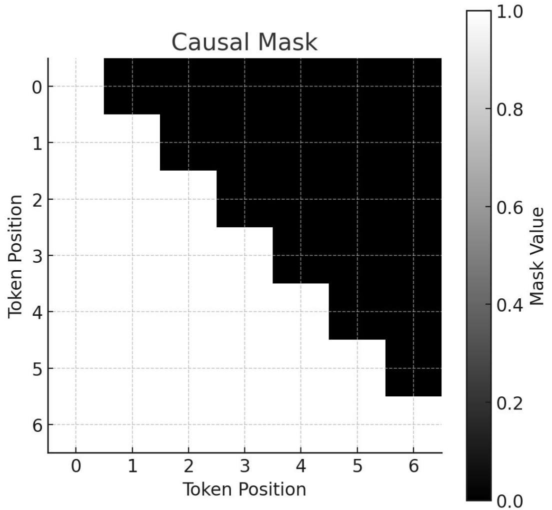
Attention formula

$$\text{attention}(K, V, Q) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

- K: keys (constructed from input) K = W_K @ x
- Q: queries (constructed from input) Q = W_Q @ x
- V: values (constructed from input) V = W_V @ x
- d_k: head dimension

The Learning Objective

- The goal will be to **predict the next token in a sequence** in an **autoregressive** way
- Let's work through a simple example: "The quick brown fox jumps over…"



Attention: drawing context-dependent correlations

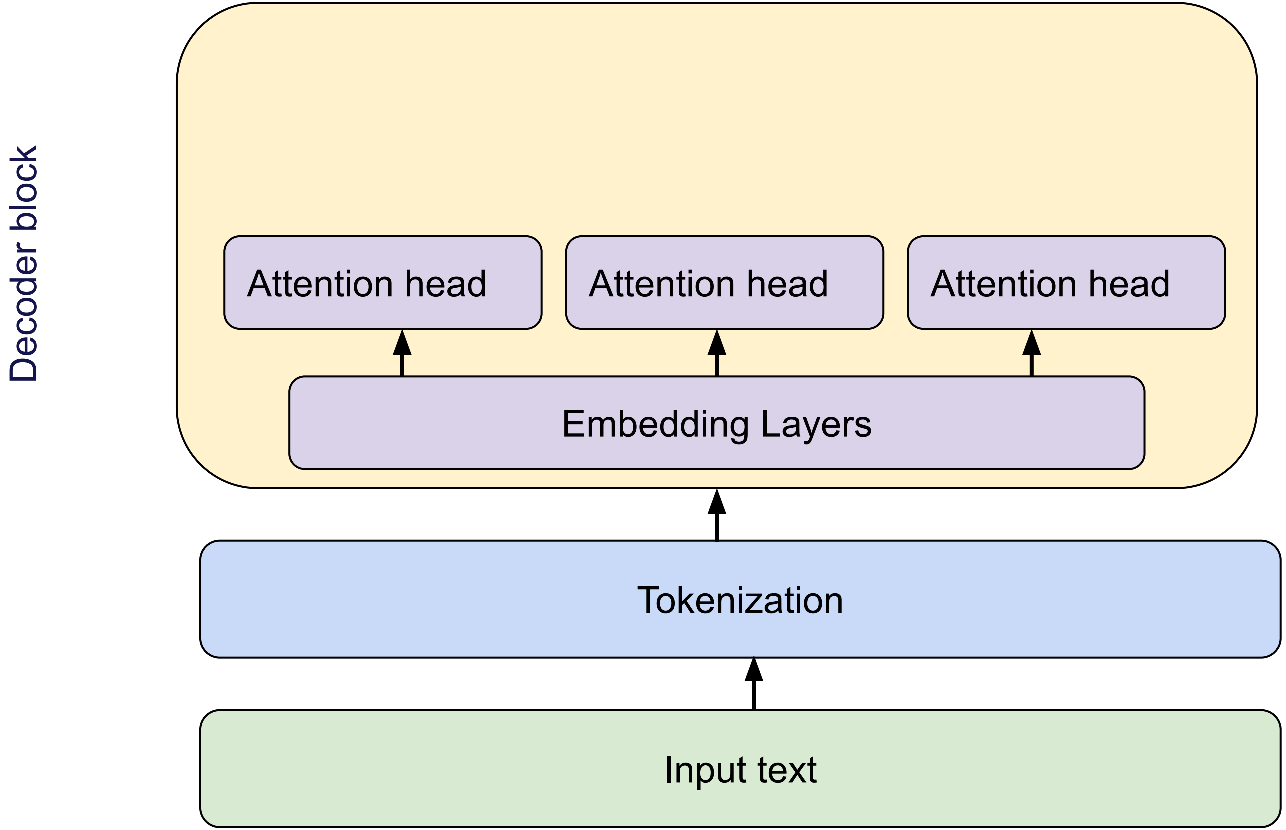
$$\text{attention}(K, V, Q) = \text{softmax}\left(c \odot \frac{QK^\top}{\sqrt{d_k}}\right) V$$

- K: keys (constructed from input) K = W_K @ x
- Q: queries (constructed from input) Q = W_Q @ x
- V: values (constructed from input) V = W_V @ x
- d_k: head dimension
- c: (causal) mask

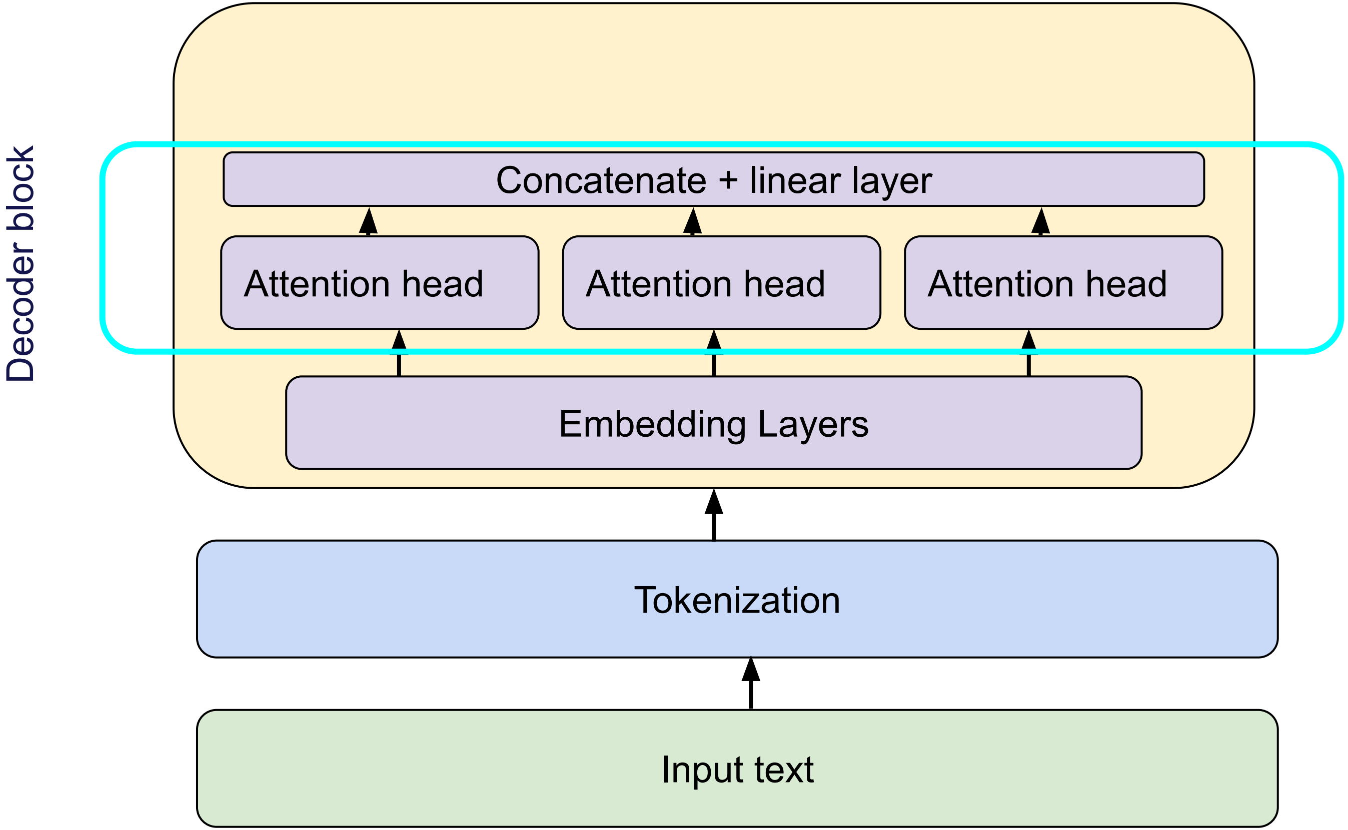
Exercise: Implementing Single Head Self Attention

Complete the "**Implementing single headed causal self attention**" exercise in the "transformer_student" Jupyter notebook.

Multi headed attention



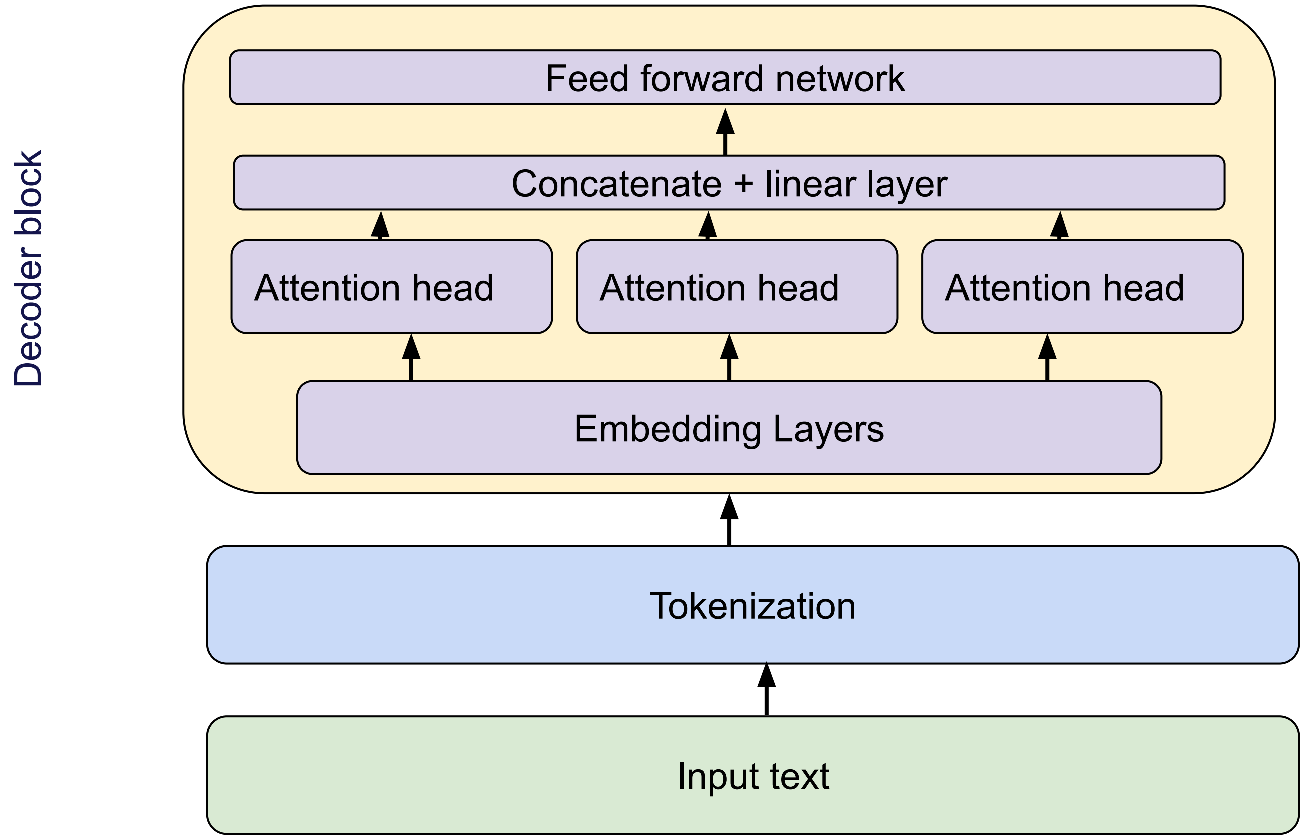
Multi headed attention



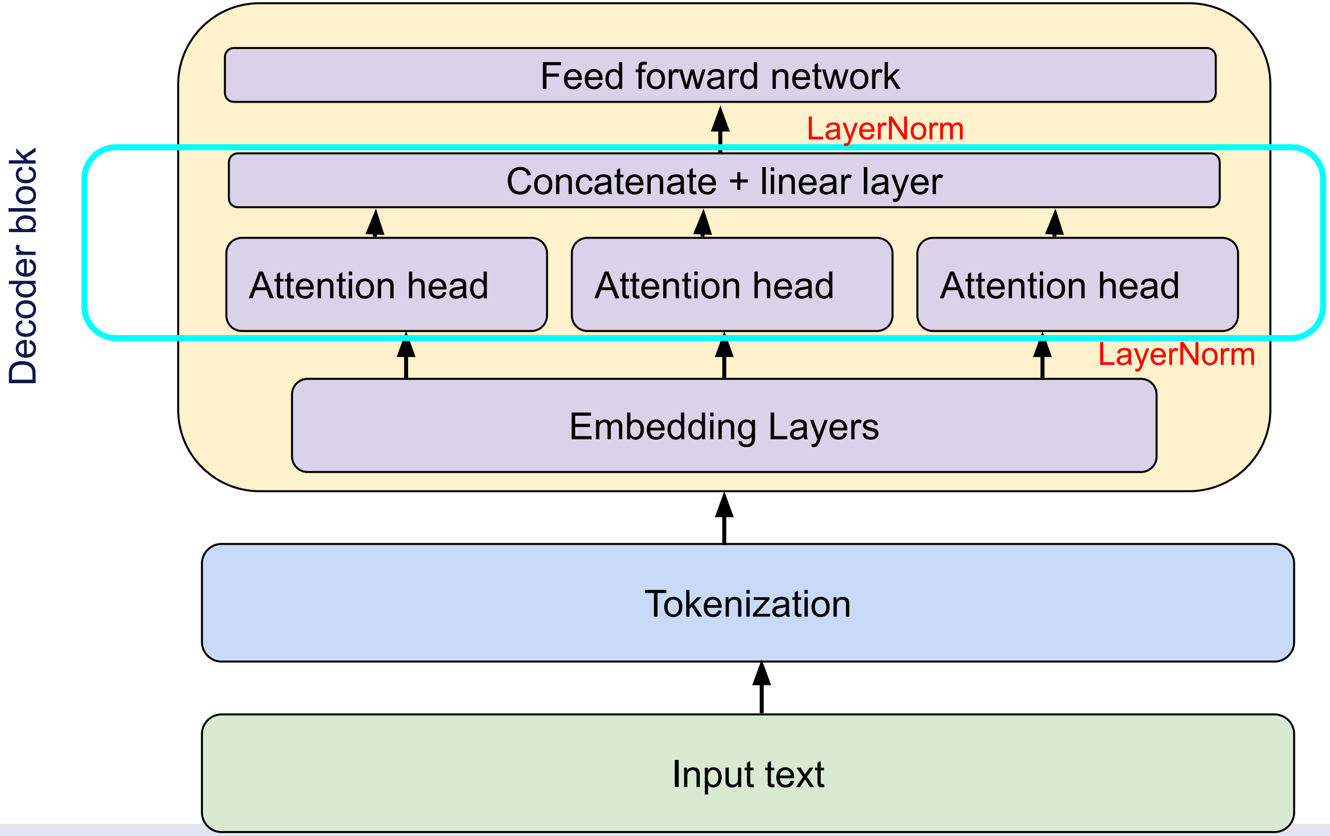
Exercise: Implementing Multi Head Self Attention

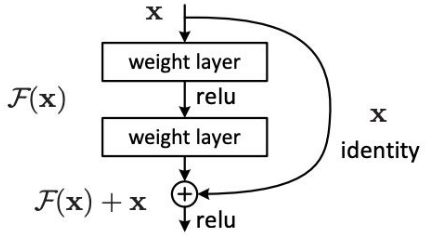
Complete the "**Implementing multi-head attention**" exercise in the "transformer_student" Jupyter notebook.

Feed forward network ("factor of 4")



Feed forward network and layer normalization





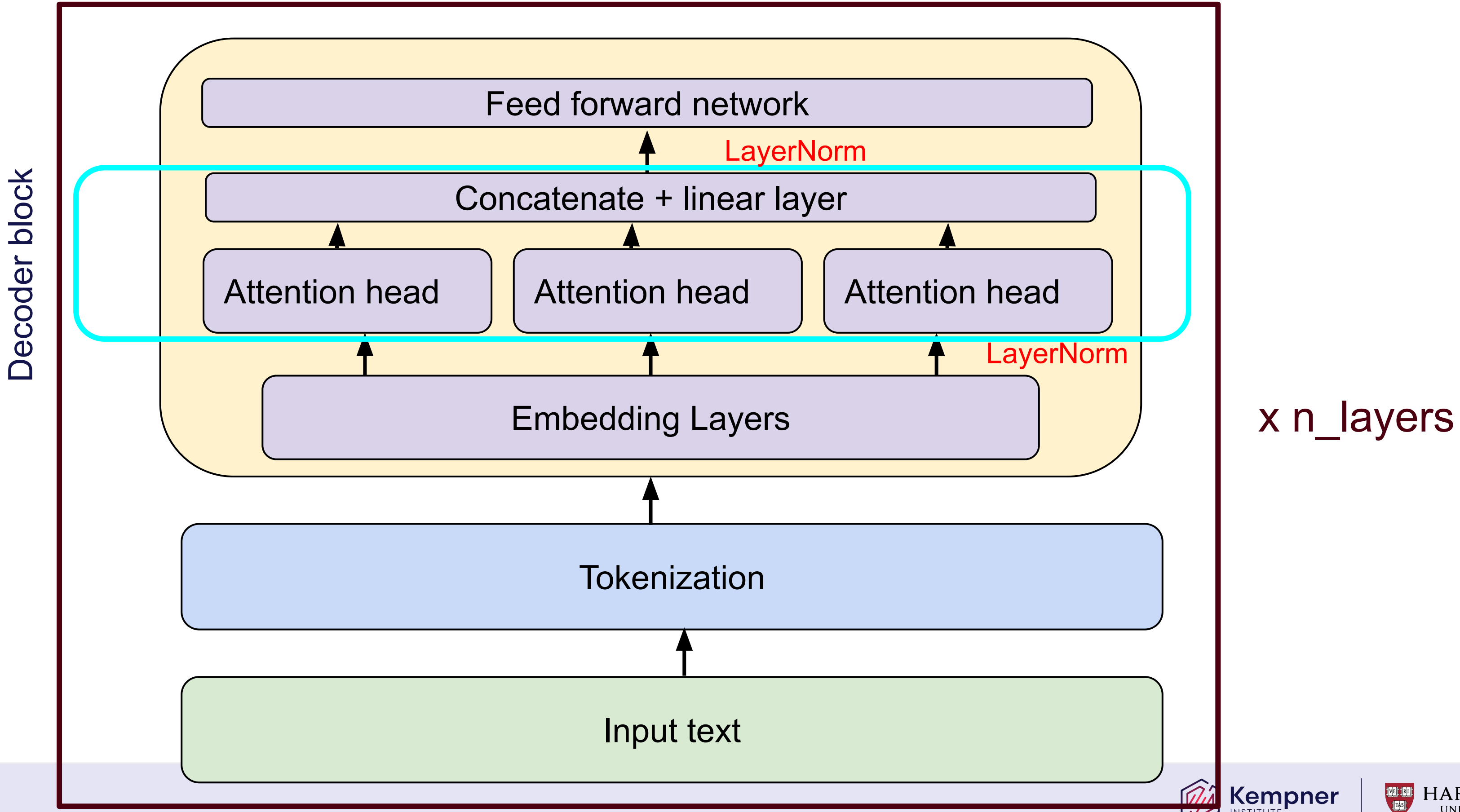
From He et al.
https://arxiv.org/abs/1512.03385

Exercise: Implement feed forward network + decoder block

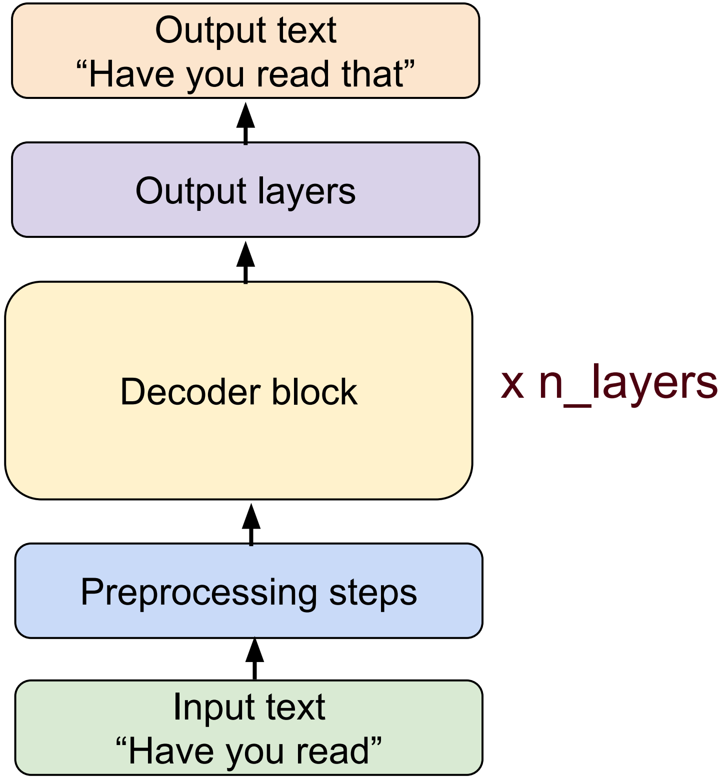
Complete the following exercises in the "transformer_student" Jupyter notebook:

1) FFN
2) Decoder Block

Putting together our transformer



Putting together our transformer



Output layers will take the result of the decoder blocks and output logits in the vocabulary for our next token prediction task

Exercise: Implementing the transformer

Complete the Decoder class in the “transformer_student” Jupyter notebook.

a) First, complete the `forward` method
b) Next, complete the `generate` method, which implements the autoregressive task

Agenda

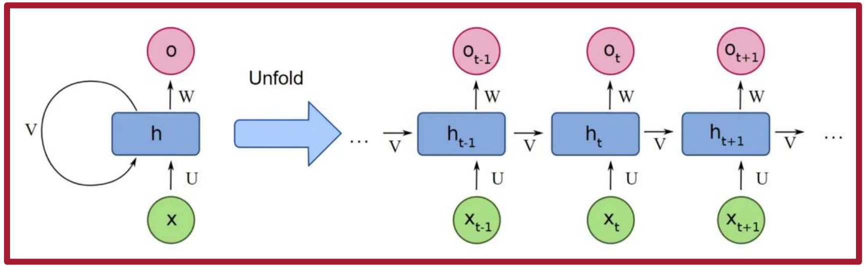
1 Introduction to Transformers

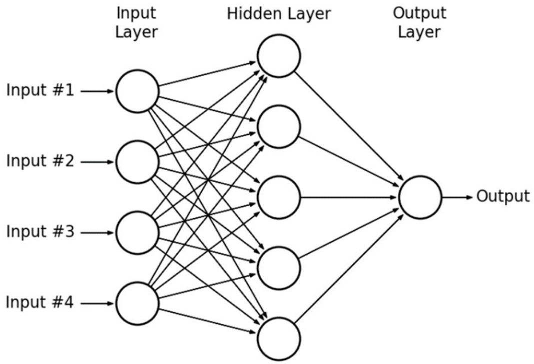
2 Building a Transformer
- Tokenization
- Embeddings
- Attention
- Full Decoder

3 Training a Transformer

Why transformers?

- Why did transformers take off? Why can't we use, e.g., MLP layers or RNNs for everything?





Brainstorm your best guess for why transformers became so popular for language modeling tasks (as opposed to other architectures)

Why transformers?

- **Computation**
  - For RNN/LSTM, the time to compute the loss is a fundamentally serial operation. For a sequence length of dimension T, this is an O(T) operation.
  - For a transformer, the serial compute is O(1) (no T dependence) while total computational complexity is O(T^2)
  - The # of parameters have no T-dependence
  - Practically very easy to parallelize
- **Inductive bias**
  - Still an open area of research, but the inductive bias of attention seems to be useful.
    - Transformers are able to create sparse features of things far apart
    - Use context very easily (e.g., recall/copy)

Modern LLMs (Llama 3, GPT4, Gemini, etc,)

- Surprisingly, not very different from the model you just wrote
- A few improvements:
  - FlashAttention (faster attention computation)
  - Better positional embeddings (RoPE, Alibi)
  - Minor changes to activation functions (SwiGLU)
  - Methods to manage attention memory consumption (multi-query and grouped-query attention)
  - **Data quality**
- However, many difficulties lie in the engineering challenges of scaling up the models to 400B+ parameters
- TMRC (Kempner LLM training codebase): https://github.com/KempnerInstitute/tmrc

Thank you