Course - ARENA

CHAPTER 1 - TRANSFORMER INTERPRETABILITY

CH 1.1 - TRANSFORMER FROM SCRATCH

1.1.0 - Introduction (~6 min)

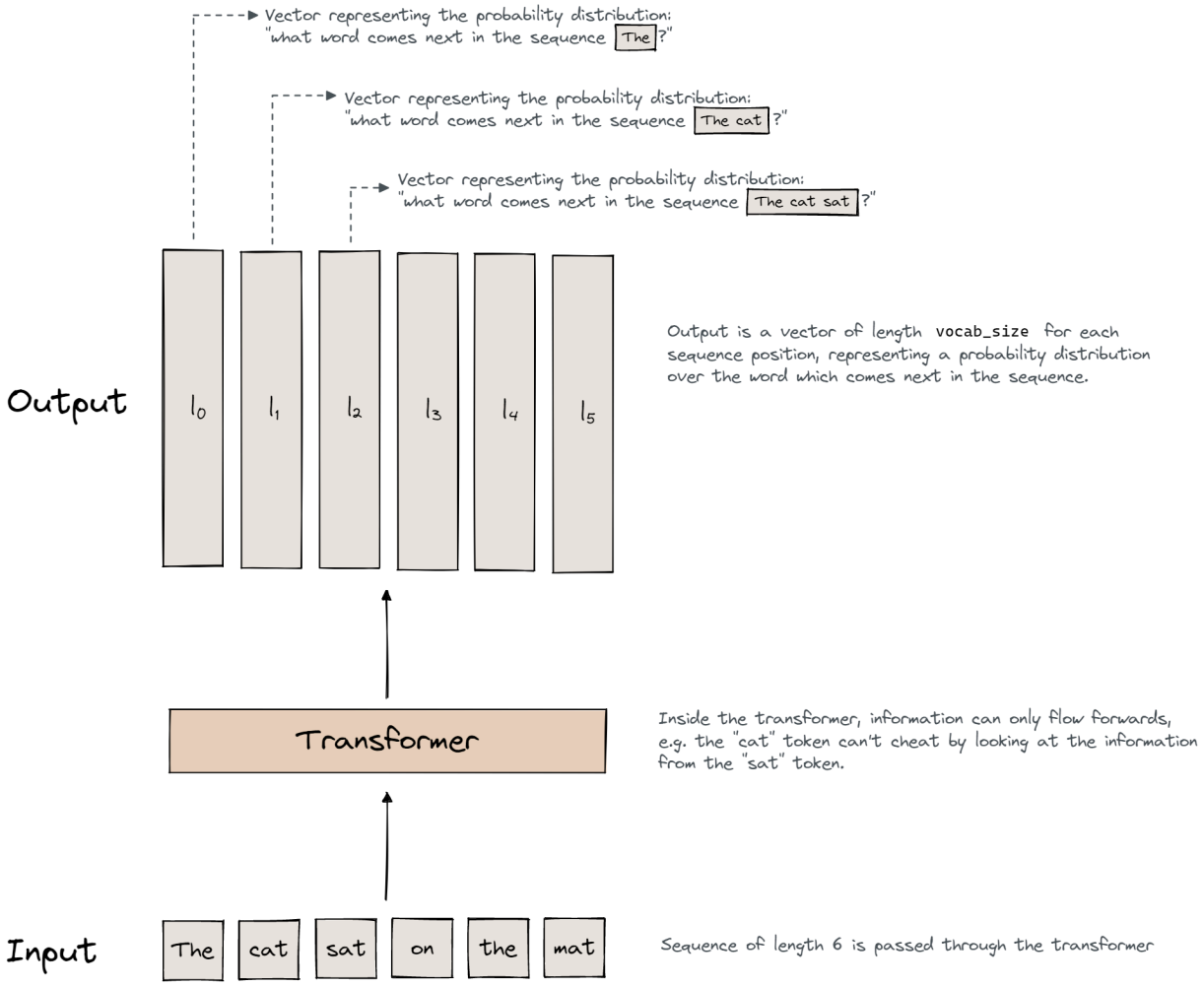
1.1.1 - Understanding the Inputs & Outputs of a Transformer (~36m)

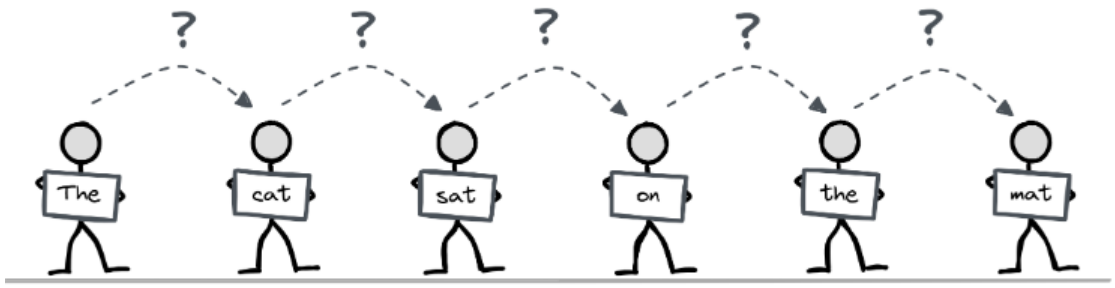
Objectives

- What is a transformer used for?
- Causal attention
- What transformers' outputs represent
- Algebra operations on tensors
- What and how tokenization
- What are logits and how to use in deriving probability distribution

How is the model trained?

- I didn't realize this but if you give 100 tokens it predicts next for each token in the series: (why?)
  - $p(x1), p(x2|x1), p(x3|x1x2), \ldots, p(xn|x1 \ldots xn-1)$
- CAUSAL ATTENTION: model only has info for certain token and all previous ones not future ones (also called autoregressive model)





Tokens - Transformer Inputs

- How convert "words" to vectors? EMBEDDING --> each element of vocabulary gets an index label (or equivalently a one-hot vector) for looking up its embedding vector

$$W_E = \begin{bmatrix} \leftarrow v_0 \rightarrow \\ \leftarrow v_1 \rightarrow \\ \vdots \\ \leftarrow v_{d_{vocab}-1} \rightarrow \end{bmatrix} \quad \text{is the embedding matrix (size } d_{vocab} \times d_{embed}),$$

$$t_i = (0, \ldots, 0, 1, 0, \ldots, 0) \quad \text{is the one-hot encoding for the } i\text{th word (length } d_{vocab})$$

- $$v_i = t_i W_E \quad \text{is the embedding vector for the } i\text{th word (length } d_{embed}).$$
- How to convert language into "sub-units (not exactly words)"
- Start with token list = (all ASCII chars), then repeat: add as a new token the most common pair of previous tokens
- In GPT <|endoftext|> is a beginning/end token (like '.') in Karpathy's course. Aka a Beginning Of Sequence BOS token, etc.
- it's automatically added so keep that in mind I might want to disable when doing some things
- BOS and EOS can't be the same if not causal because the difference then can matter, e.g. not in GPT but yes in BERT
- Initial Spaces/upper/lowercase give different words " boy" "boy" "Boy" all different
- Numbers are annoying sometimes a group is a token sometimes not
- Tokens = integers as inputs to the transformer, vector is later

Text Generation

- **STEP 1:** Convert text into tokens [batch, seq_length]
- **STEP 2:** Map tokens to logits [batch, seq_length, vocab_size], where logit[i, j, k] element is batch i, token j, next token (i.e prediction for the j+1th token) k
- **STEP 3:** Convert logit to distribution using softmax
- **STEP 4**: Map distribution to a token, here just take the most likely next one, logits[0, -1].argmax(dim=-1), argmax over the k (-1th dim), then take the one predicted to come after the last known j (-1th index) for our only batch (0th index)
- **STEP 5:** Add this token to the end of the input and re-run, keep going until you generate your answer to the end.

1.1.2 - Clean Transformer Implementation (~50m reading+~2h37m exercises+~1hr not logged = ~4h27m total)

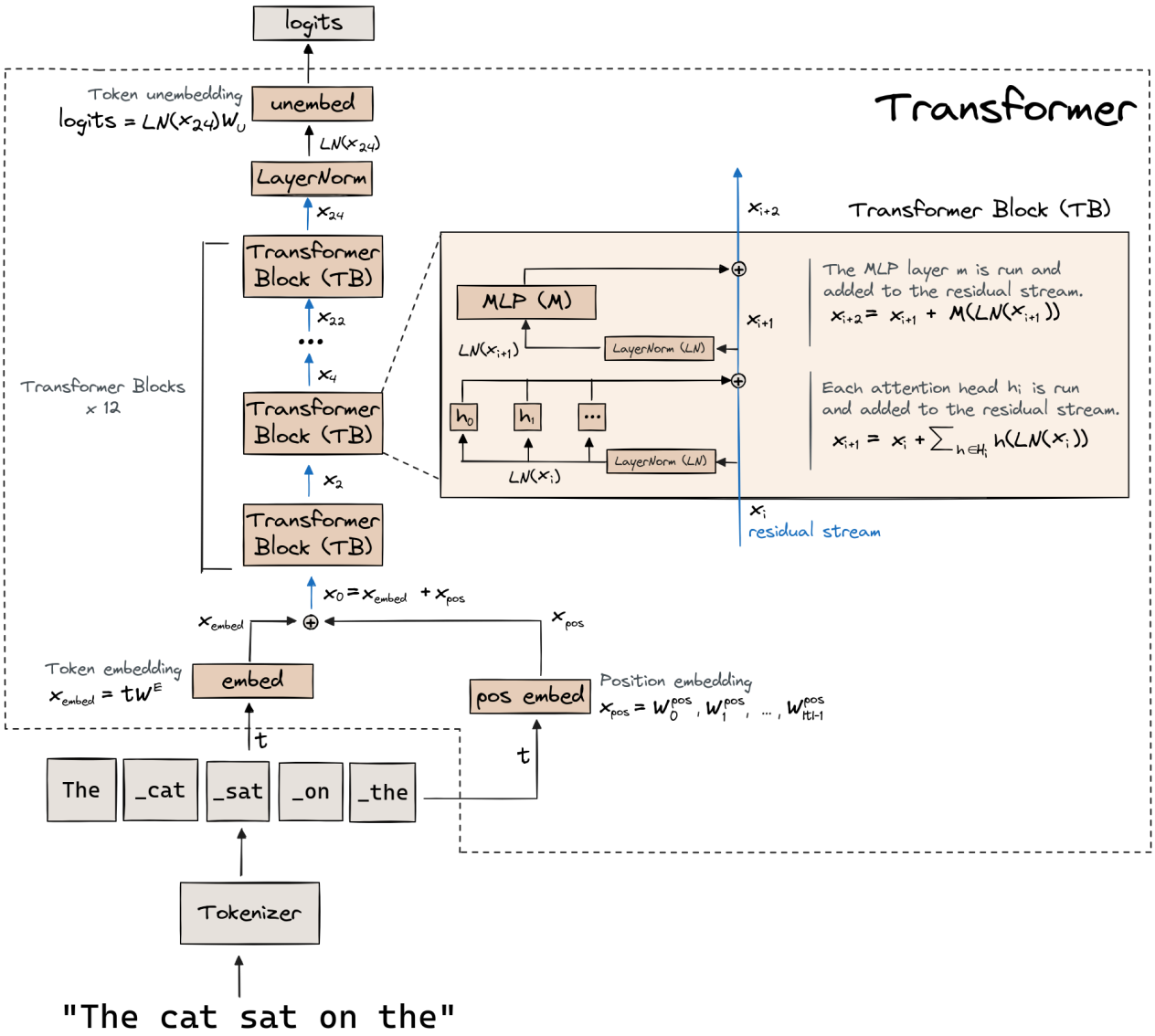
Objectives

- Understand attention heads and MLPs
- Attention heads in a single layer are independent

- Learn and implement the following transformer modules
  - LayerNorm
  - Positional Embedding
  - Attention
  - MLP
  - Embedding
  - Unembedding

High-Level Architecture

- Transformer circuits walkthrough: https://www.youtube.com/watch?v=KV5gbOmHbjU



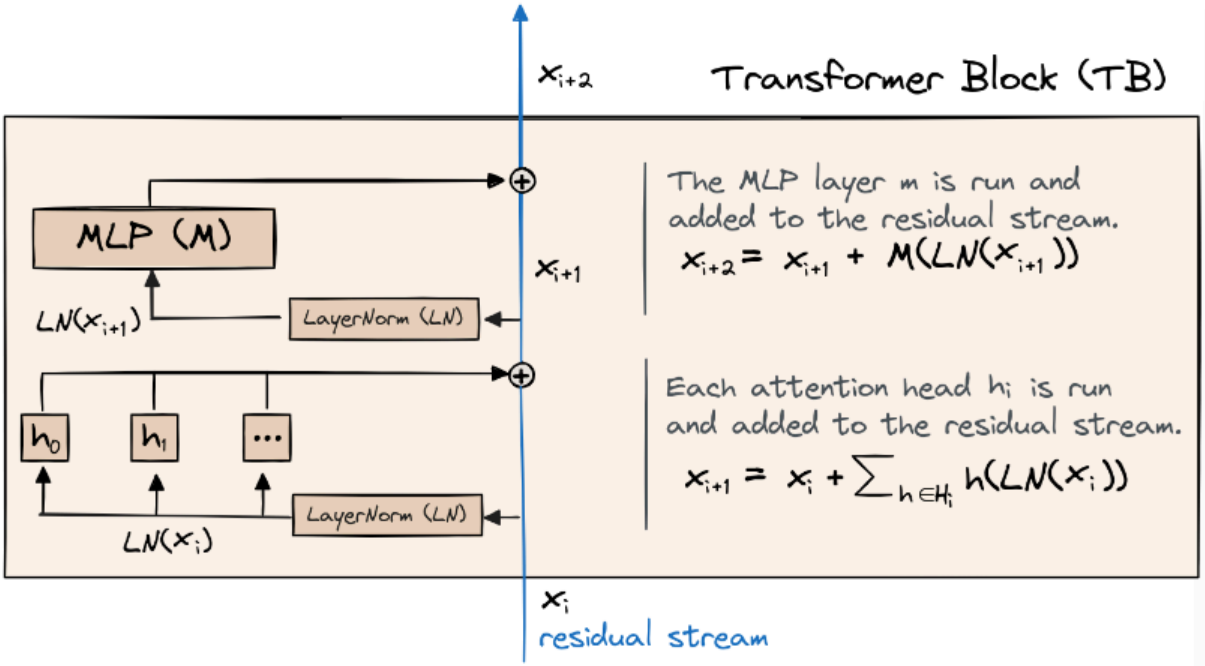
- Tokenize the input string, then input tokens t are integers, embedding is $x_{emb} = tW_{emb}$ where W here is a lookup table from tokens to their vectors
- RESIDUAL STREAM is the sum of outputs of all previous layers = input to each new layer, it is [batch, seq_length, d_model=dim of embedding vector]

- x_i is residual stream after going through i=(num transformer layers)+(num MLP layers)
- "logit lens" is idea you can look at the residual stream at different stages in the model an still get useful information, since the residual stream is additive as you go along, the earlier information is getting you partway to the final answer in a way it seems

Transformer blocks

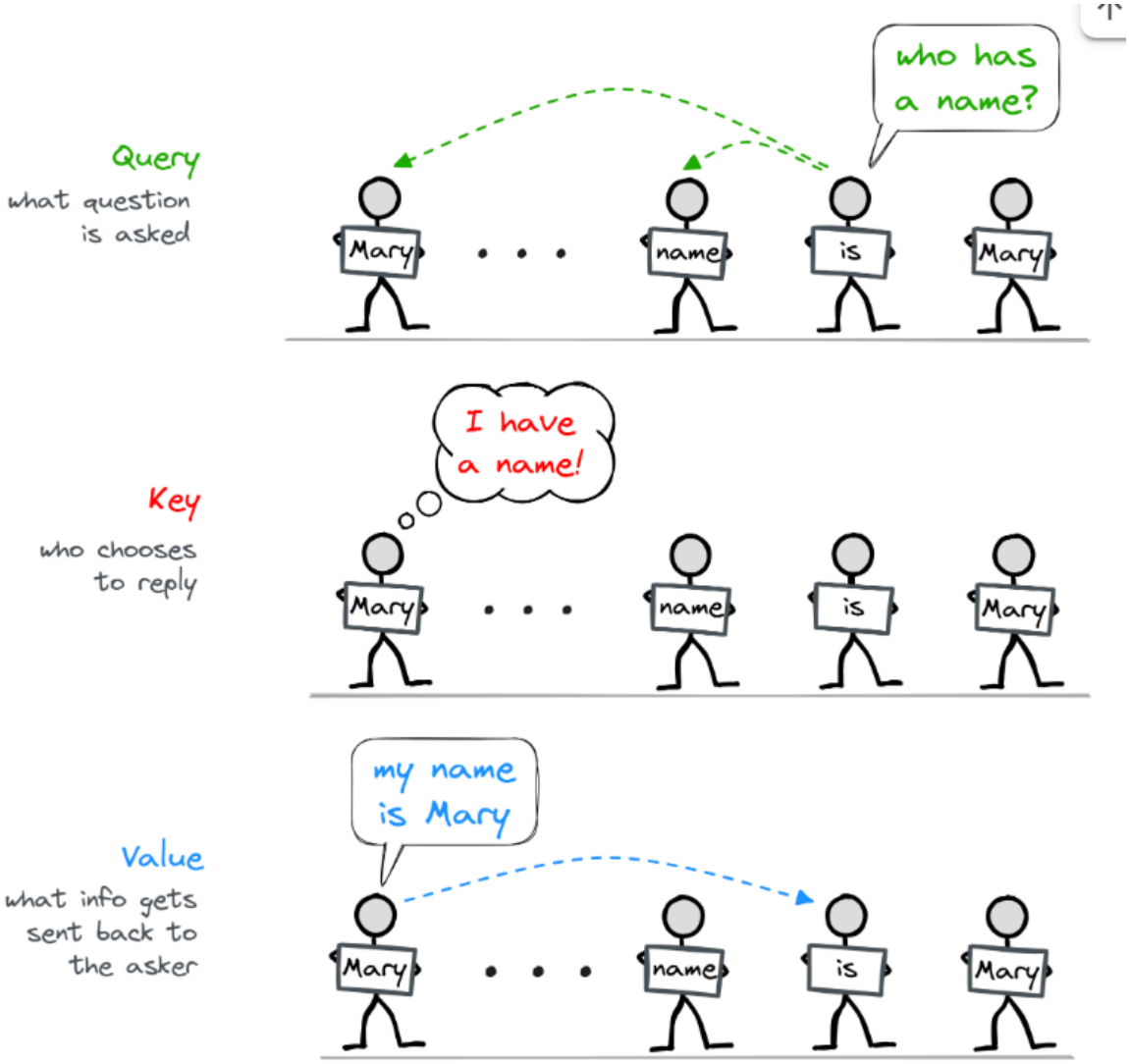
Then we have a series of `n_layers` **transformer blocks** (also sometimes called **residual blocks**).

Note - a block contains an attention layer *and* an MLP layer, but we say a transformer has k layers if it has k blocks (i.e. $2k$ total layers).

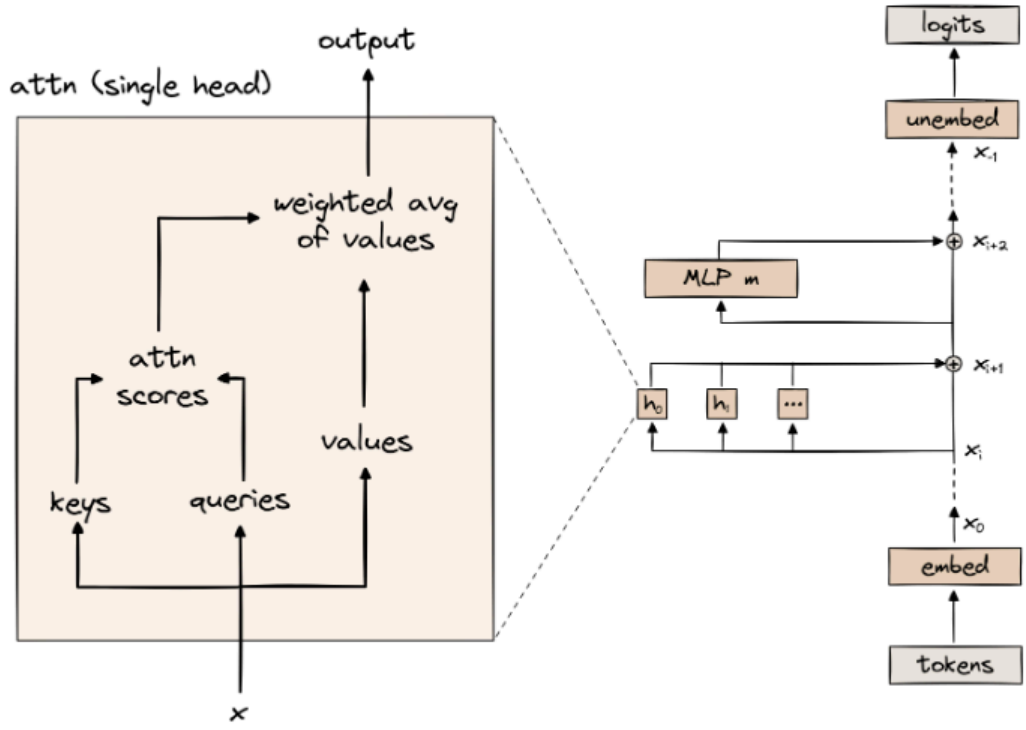


Attention

- Done in parallel on every token
- Only part that moves information between positions
- Each layer is made of independent heads
- Each HEAD produces an ATTENTION PATTERN --> prob distro of tokens weighting how much info to copy + it moves info from each source to destination token
- Heads have 3 parts:
  - Queries: questions/requests for certain info
  - Keys: whether a token has the answer to different questions
  - Values: the info that gives the actual answer to a question

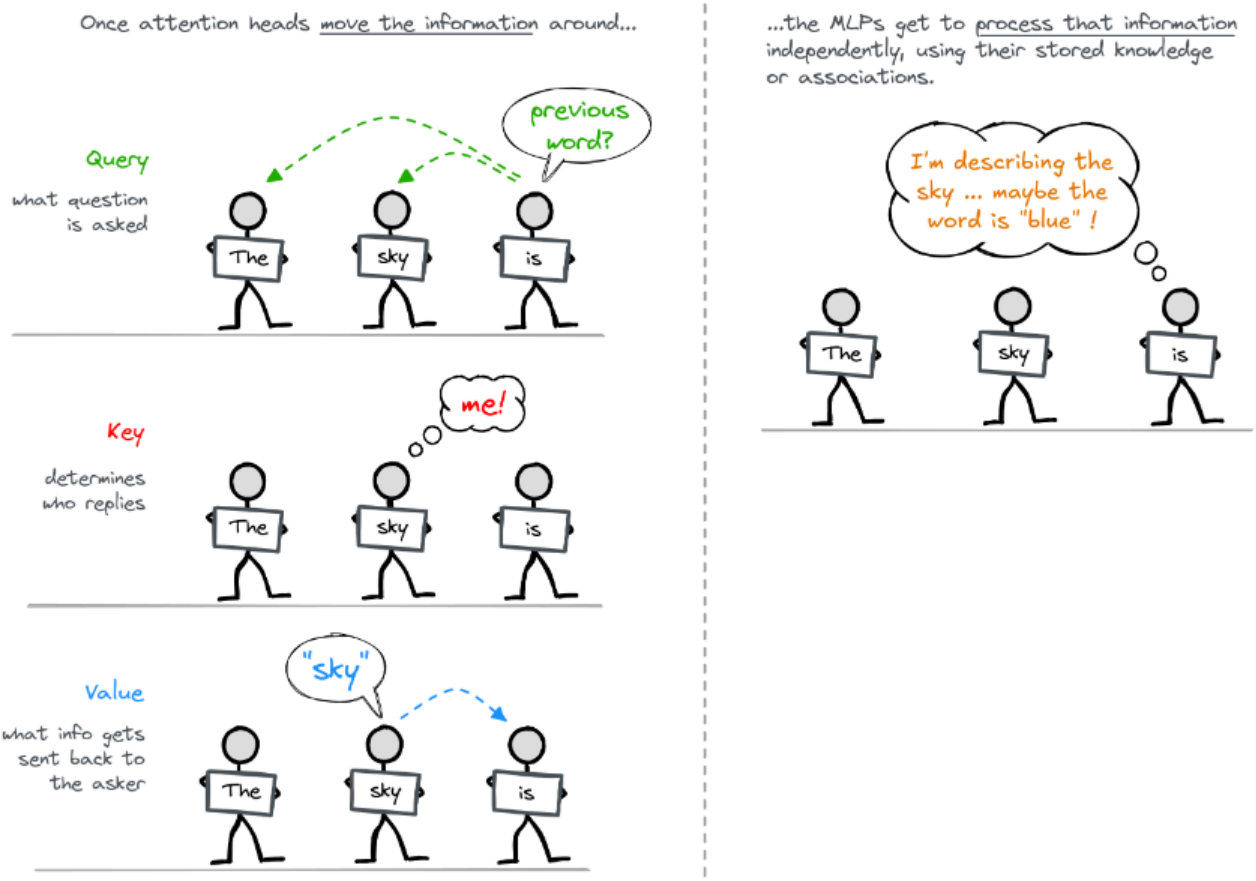


- Attention is like convolution in a CNN except convolutions take info from nearby pixels but this can have info in a more complicated way from any earlier token

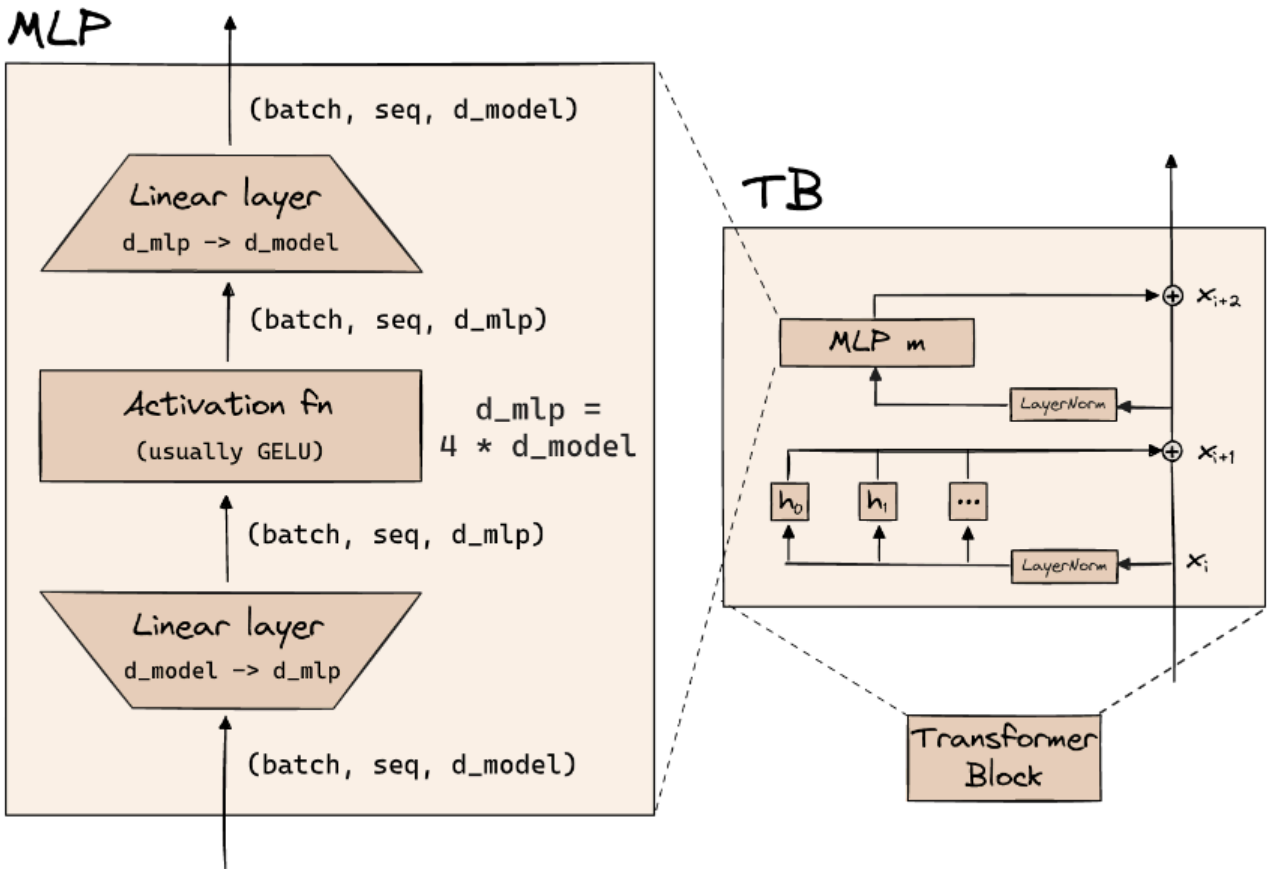


MLP

- 1 hidden layer with dimension d_mlp = 4(d_model)
- MLP doesn't move info between positions
- Imagine attention is question asking / answering / sharing info and MLP is independent thinking



- MLP is where the "memory" is stored, it takes in a vector and outputs a vector sort of like a key-value pair sort of like looking up a response to a bit of information?

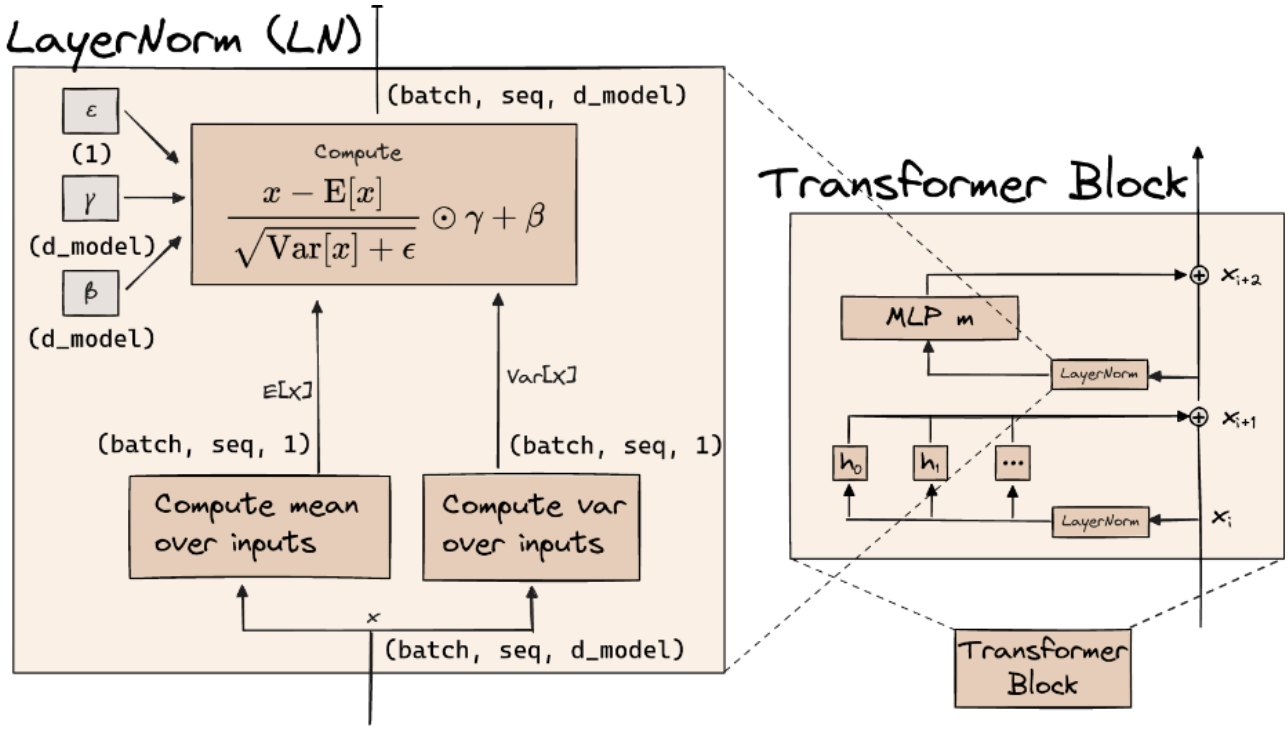


Unembedding

- Go from final residual stream into logits
- "tied embeddings" match the same weights int the initial embedding and final unembedding. This is efficient but can be a problem by incentivizing the model to lean toward acting like a bigram? not super important

LayerNorm

- Before every layer, normalize to mean zero and variance 1
- It's "almost" linear but the division breaks it -- so ~linear for small changes in a single part of the input



Positional Embeddings

- By default, attention doesn't care about how near 2 tokens are. But really, nearby tokens are usually more relevant
- Here we add in a lookup table mapping index of position of each token into the residual stream, so it's additional information that can be learned (and will likely learn to focus on nearby tokens more) - again there's an analogy to convolution

Coding Section

Parameter	Value
batch	1
position	35
d_model	768
n_heads	12
n_layers	12
d_mlp	3072 (= 4 * `d_model`)
d_head	64 (= `d_model / n_heads`)

- PARAMETERS = learned weights and biases
- ACTIVATIONS = temporary numbers, calculated during forward pass, that are functions of the input
- https://raw.githubusercontent.com/chloeli-15/ARENA_img/main/img/transformer-full-updated.png

Key

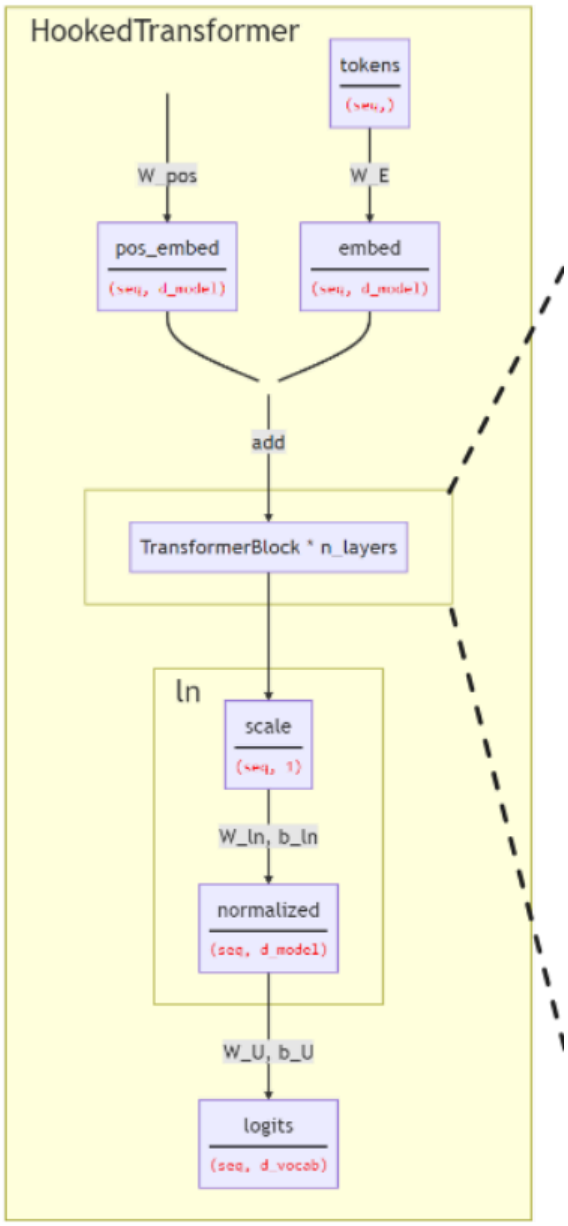
You can index from the cache using these names, followed by layer number if they are in a transformerblock. For example:

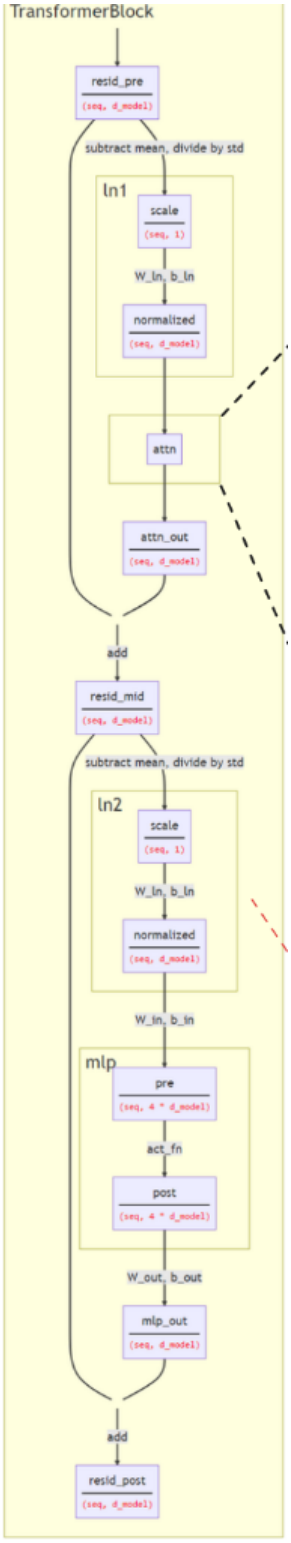
```
cache["q", 0]      -> query vectors for the 0th transformerblock
cache["pre", -1]   -> pre-act fn neuron activations for MLP in last layer
cache["embed"]     -> token embeddings
```

The only exceptions are "scale" and "normalized", because you have to specify which of the layernorms these refer to:

```
cache["normlized", 3, "ln1"]   -> 3rd layer 'normalized', pre-attention
cache["scale", 7, "ln2"]       -> 7th layer 'scale', pre-MLPs
```

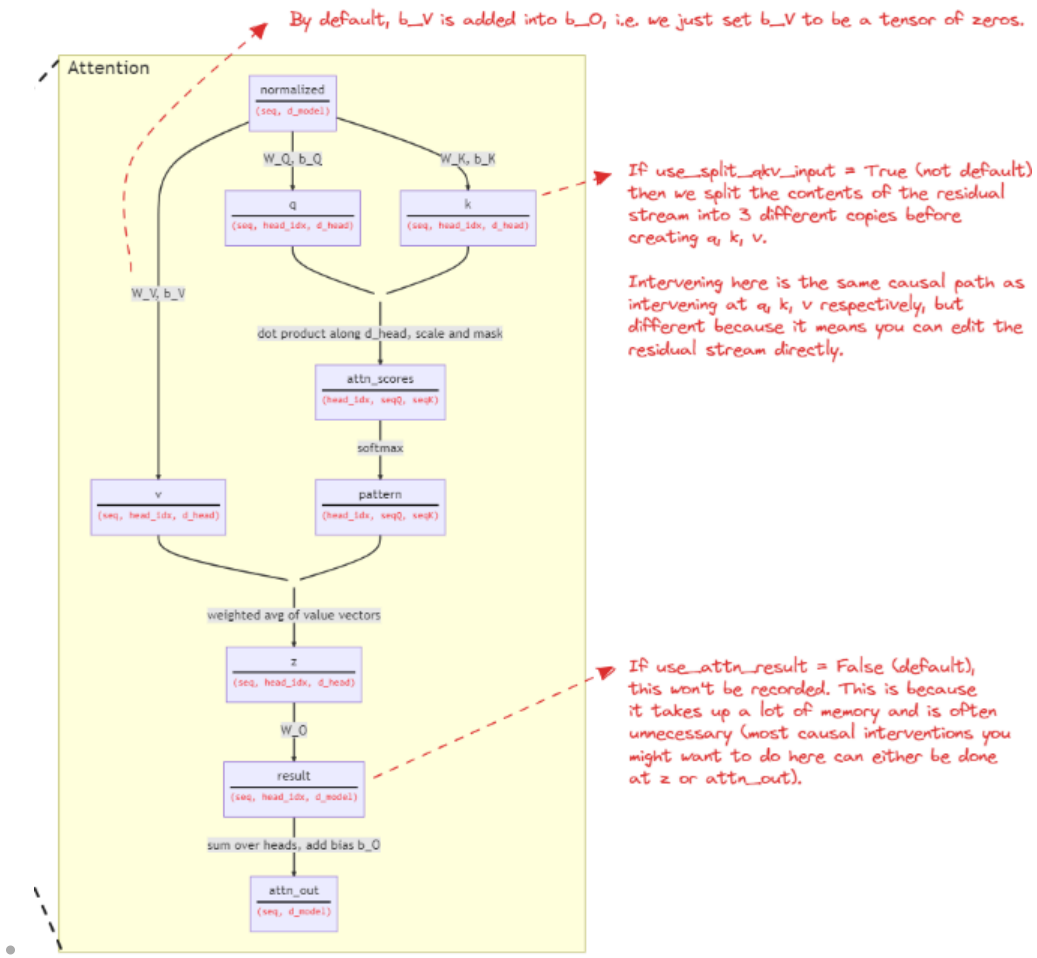
Under the hood, cache calls utils.get_act_name on the thing you index it with. This utils function can be helpful for getting hook names.


HookedTransformer
tokens
(seq,)
W_pos
W_E
pos_embed
(seq, d_model)
embed
(seq, d_model)
add
TransformerBlock * n_layers
ln
scale
(seq, 1)
W_ln, b_ln
normalized
(seq, d_model)
W_U, b_U
logits
(seq, d_vocab)



- If fold_ln = True (default), then W_ln and b_ln are folded into the weight matrices which follow them (i.e. W_in for ln2, or W_Q/K/V for ln1). This means that the layernorms only apply the nonlinear bit (centering and scaling).

  The scale parameter is the std dev of the residual stream. For example, in the case of ln2, we take the input 'resid_mid', subtract the mean, calculate 'scale' (the std dev), divide by this value, and we get 'normalized'.



Config

- Contains all the hyperparameters

Tests

- I'll have to write some as we go but there are a few provided

EXERCISE - IMPLEMENT LAYERNORM (EST. 10-15min / ACT. ~23min)

EXERCISE - IMPLEMENT EMBED (EST. 5-10min / ACT. ~7min)

- haha yes you literally use the token to index into the lookup table

EXERCISE - IMPLEMENT POSEMBED (EST. 10-15min / ACT. ~21min)

- How I originally did it passed the tests but wouldn't have worked overall
- I looked at the solution and saw it was very different. I think I understand it now but the positional embedding might be a potential weak point for me
- I think the main thing missing was the dimension! I lost the batch dimension by what I was doing.

- As I understand it the indexing was a really inefficient way - why index all 0, 1, 2, etc. when a slice will do.

EXERCISE - IMPLEMENT APPLY_CAUSAL_MASK (EST. 10-15min / ACT. ~24min)

- Think my answer was correct before checking the solution
- Only change is to make sure I use the same device
- Although it is slightly different. After thinking about it and consulting chatgpt I think they are equivalent although mine might use more memory since I am making a larger mask

EXERCISE - IMPLEMENT ATTENTION (EST. 30-45min / ACT. ~68min)

- STEP 1: For each destination token, produce probability distribution over previous tokens (including self)
  - input --> [batch, seq_pos, head_idx, d_head]
  - input --> (query, key)
  - scores = all pairs (q, k)
  - [batch, head_idx, q_pos, k_pos]
  - scale (divide by sqrt(d_head)) + mask (so causal)
  - softmax for probs (this is ATTENTION PATTERN)
- STEP 2: use attention pattern to move info from source (key) token to destination (query) token
  - input --> [batch, key_pos, head_idx, d_head]
  - input --> value
  - weighted avg of value vectors along key_pos direction, call it z
  - z =[batch, q_pos, head_idx, d_head]
  - sum over heads [batch, position, d_model]

Key for dimensions in the diagram:

b = batch dim (we assume batch = 1 in the diagram, to keep things simple)
n = num heads / head index
s = sequence length / position (s_Q and s_K are the same, but indexed to distinguish them)
e = embedding dimension (also called d_{model})
h = head size (also called d_{head}, or d_K)

We apply a final linear transformation to our value vectors, mapping them up to the right size to be added to the residual stream.

This is a linear map from size h → e_I for each head.

For each destination position, we take a weighted average of value vectors from each source position, in accordance with how much attention destination pays to source.

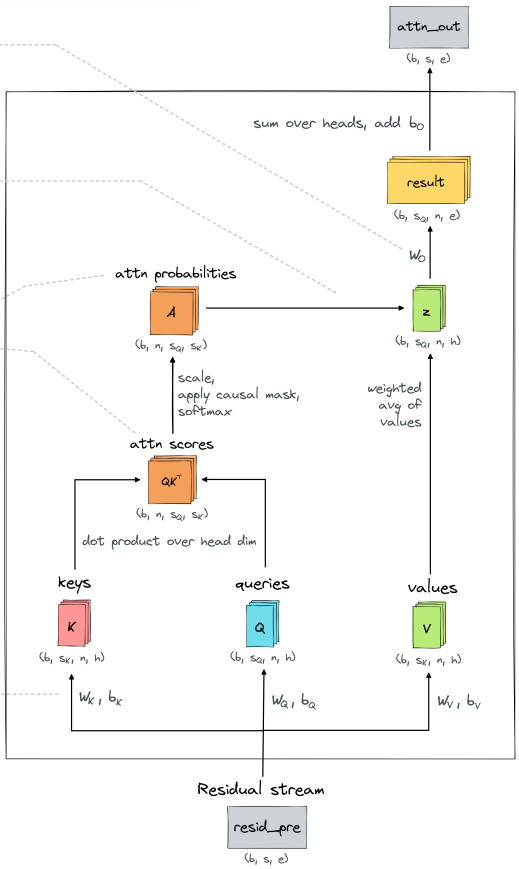
In other words, we multiply attn probs by values along the s_K dim.

We get attention scores by taking the inner product of keys and queries along the head dimension. This gives us matrices of shape (seq, seq) for each head. Then we softmax over the k-dimension to get attn probs.

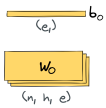
Attn probs tell us how much each query position (destination) pays attention to each key position (source), i.e. this tells us where information moves to and from.

We make sure no information flows backwards by setting the attention scores to be -1e5 wherever query posn < key posn (so the corresponding attn probs are zero).

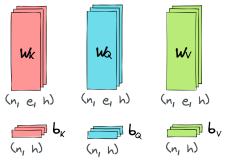
3 separate linear transformations give us keys, queries and values for each sequence position.



Projection matrices (and biases):

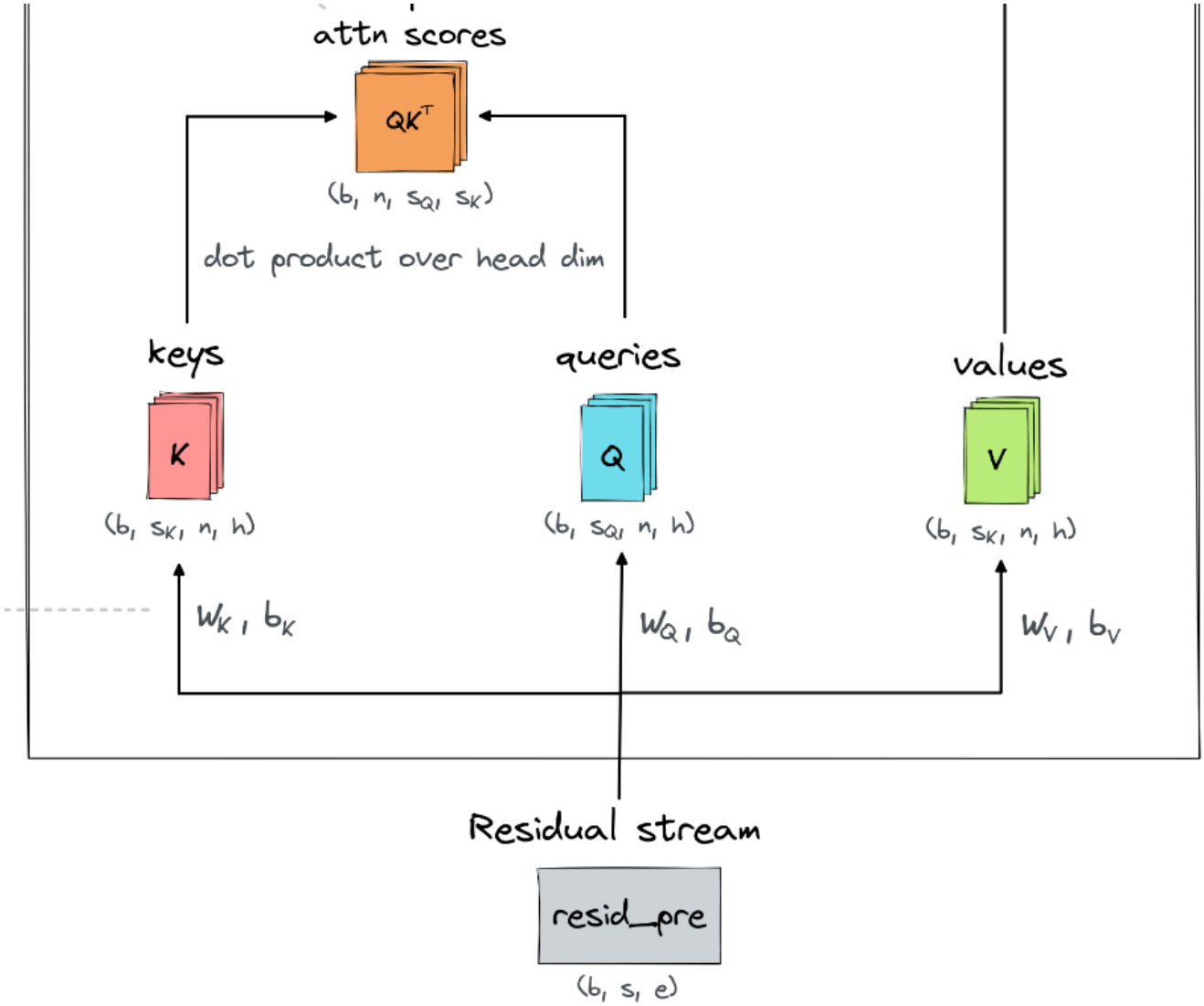


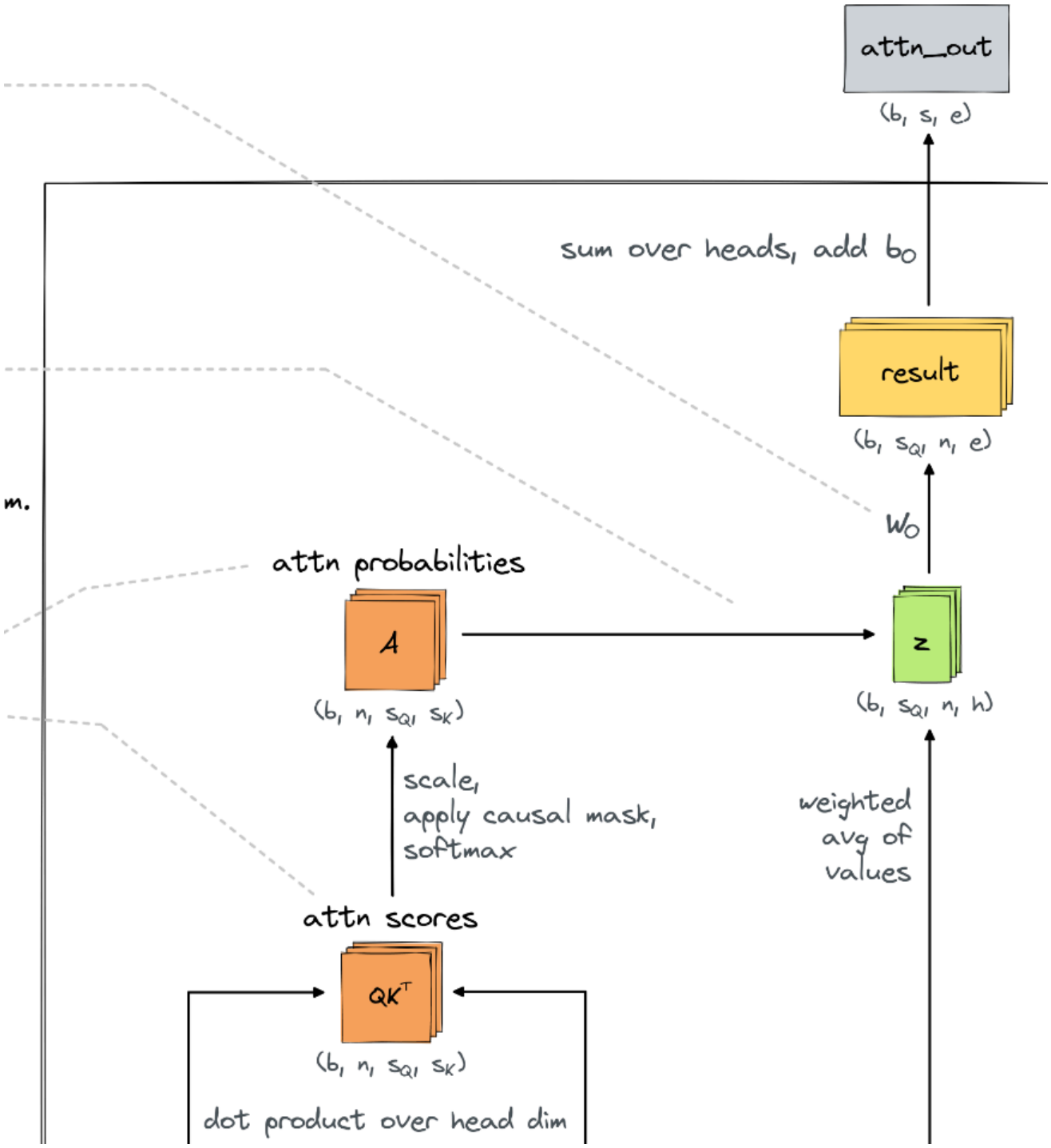
Projection matrices (and biases):



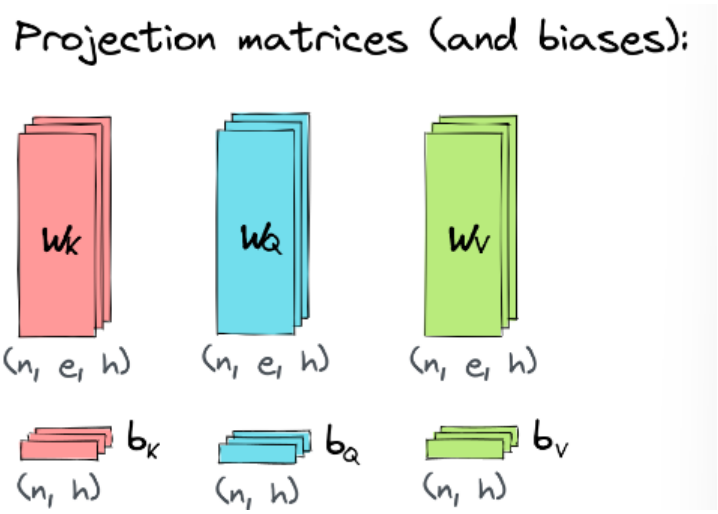
Key for dimensions in the diagram:

b = batch dim (we assume batch = 1 in the diagram, to keep things simple)
n = num heads / head index
s = sequence length / position (s_Q and s_K are the same, but indexed to distinguish them)
e = embedding dimension (also called d_{model})
h = head size (also called d_{head}, or d_K)


attn scores
QK^T
(b, n, s_Q, s_K)
dot product over head dim
keys
queries
values
K
Q
V
(b, s_K, n, h)
(b, s_Q, n, h)
(b, s_K, n, h)
W_K, b_K
W_Q, b_Q
W_V, b_V
Residual stream
resid_pre
(b, s, e)



Projection matrices (and biases):



- I'm not really understanding the insight I should be gleaning from the examples of attention patterns provided...
- WOW! Proud I was able to fully do this one. Answer key helped me find one typo where I put values instead of keys but besides that.
- Also really learned the power of einops on this one.... I was a doubter but it is really helpful and "just works"

EXERCISE - IMPLEMENT MLP (EST. 10-15min / ACT. ~4min)

- Think my way is equivalent to the einops way, if so this was very straightforward

EXERCISE - IMPLEMENT TRANSFORMERBLOCK (EST. 10-15min / ACT. ~3min)

- also seemed very easy

EXERCISE - IMPLEMENT UNEMBED (EST. 10min / ACT. ~3min)

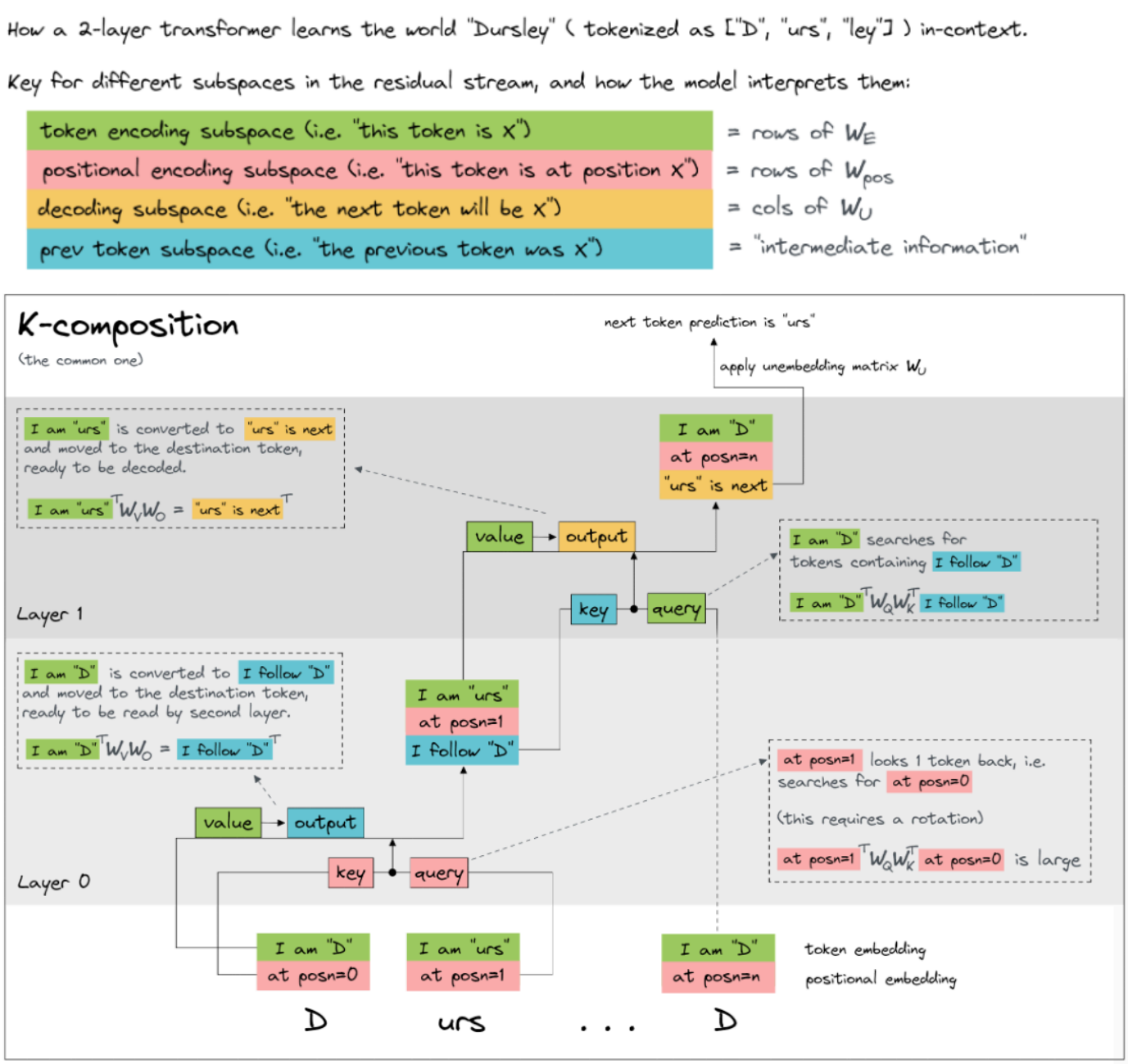
- Another straightforward one where I think einops is not necessary
- At first I thought unembed would be like embed in that you just index into it as a lookup table. But of course that's not the case esp. since the residual is a tensor of floats

EXERCISE - IMPLEMENT DEMOTRANSFORMER (EST. 10-15min / ACT. ~4min)

- Cool. Reminder that you add the embedding and positional embedding to get the first residual

CH 1.2 - Intro to Mechanistic Interpretability: TransformerLens & induction circuits

Introduction (~ 5min)



- I feel like I partly, but don't fully, understand this diagram

TransformerLens: Introduction (~20 min + ~ min exercises)

- load model with `HookedTransformer.from_pretrained(MODEL_NAME)`
- index weights like `W_Q` directly from the model via e.g. `model.blocks[0].attn.W_Q`
- model.to_str_tokens(text) are strs vs model.to_tokens(text) are ints
- Without given a BOS token the model might default to pointing to first letter (since can't all be zero everywhere since probs sum to 1)

EXERCISE - how many tokens does your model guess correctly? (EST. 10m / ACT. 12m)

- I didn't realize the return shape of to_tokens and a good reminder about prepend_bos

Caching all Activations

- INDUCTION HEADS --> attention heads that generalize in text they are seeing that if AB, then later in the text if they see A then they will predict B even if not in their training
- Cache the activations so you can study them later
- How to get attention patterns for layer 0, 2 ways:
  - `attn_patterns_from_shorthand = gpt2_cache["pattern", 0]`
  - `attn_patterns_from_full_name = gpt2_cache["blocks.0.attn.hook_pattern"]`
- utils.get_act_name() goes from short name to long one

EXERCISE - Verify Activations (EST. 10-15m / ACT. ~m)