



Data Analysis Block Course: Statistics and Machine learning

Solutions: Transformers by Hand

Prof. Dr. Mikhail Mikhasenko

Assistants: Hendric Jonas, Christoph Tönnis



30 September 2026

Indico schedule

RUB Moodle

Part I - Tokens and embeddings

Exercise 1.1 - Subword tokens

1. With character-level tokens, the model has to spend capacity and computation learning spelling and morphology (how characters combine into words) before it can even begin to reason about meaning - and sequences become much longer (roughly one step per letter instead of one step per word/subword), which is expensive since attention cost grows with sequence length.
2. With word-level tokens, the vocabulary must contain every distinct word form seen during training (millions of entries). Any string not in that fixed vocabulary at training time - a typo, a rare name, a word in another language, a new coinage - has no token at all and is simply unrepresentable ("out of vocabulary") at inference time.
3. BPE is a compromise: frequent words are merged all the way into a single token (so the model doesn't have to re-derive them character by character every time, keeping sequences short like the word-level case), while rare or unusual strings are decomposed into a handful of smaller, previously-seen pieces built ultimately from raw bytes (so nothing is ever truly "unknown", like the character-level case) - every possible string can still be encoded, just sometimes using more tokens.
4. The model never sees individual letters, only token ids. Tokenization happens once, before training, and is never revisited by the model: "strawberry" may become a token sequence such as **st**, **raw**, **berry**, and the model only ever sees these three integer ids, never the raw letters "s-t-r-a-w-b-e-r-r-y" as individual symbols. Counting the letter "r" requires information (the exact character sequence) that was merged away at tokenization time; the model would have to have implicitly memorized the letter-by-letter spelling of each token id, which is a much harder and less reliable thing to represent than something it can read directly off its input, like a short script iterating over characters can.

How this holds up against current models. Neither answer should be taken as "the tokenization argument above was wrong" - here is why: - **Typos (Q2):** the tokenization mechanism itself hasn't changed - an unusual misspelling still gets split into smaller/unfamiliar subword pieces, exactly as argued above. What *has* changed is that current models are trained on vastly larger and noisier internet-scale corpora containing huge numbers of real typos in context, so the model has effectively "seen" enough misspelled variants (and the surrounding context still carries meaning) that it can usually infer intent anyway. The architecture's limitation is compensated by data and scale, not removed. - **Counting letters (Q4):** many current frontier models now answer "strawberry" correctly, but be sceptical of what that proves - this exact example went viral and is now heavily represented in training/fine-tuning

data, so a correct answer here may partly reflect memorization of this specific case rather than a general fix. The more informative mechanism is **reasoning**: a model that is prompted (or trained) to first spell a word out character by character as intermediate output effectively re-derives the letter sequence itself before counting, working around the tokenizer rather than seeing "through" it. Some newer tokenizers also split more finely, which helps somewhat. A good way to test whether this is a general fix or just memorization of a famous example: try a *less famous* word with a tricky repeated letter and see whether the same reliability holds up.

Exercise 1.2 - The embedding matrix

1. Toy example: $6 \times 4 = 24$ trainable parameters. GPT-2: $50,257 \times 768 = 38,597,376 \approx 38.6\text{M}$ parameters - about $38.6/124.4 \approx 31\%$ of GPT-2's entire 124M parameters. The embedding table alone is nearly a third of the whole model, simply because it scales with vocabulary size, not model depth.
2. At initialization (e.g. small random values), the rows of E carry no meaningful structure: semantically related words (like "cat" and "dog") sit at essentially random, unrelated positions in the embedding space, just like unrelated words. Training updates every row that appears in a training batch via the gradient of the downstream task loss; words that tend to occur in similar contexts receive similar gradient signals over many updates, so their rows are pulled toward each other. The geometry of E (which rows end up close together) is therefore *learned* from data, not designed.

Concretely bad initializations:

- **All rows equal to $\vec{0}$:** unlike the classic all-zero-weight problem in a dense layer (where every neuron sees the same input *and* the same gradient and stays identical forever), an embedding lookup only updates the row of the word that actually occurred, so rows *do* start to separate as training proceeds - this is not a permanent symmetry lock. But the first forward pass still carries **no information about which word is present**: Q, K, V are identical for every token, attention degenerates to a uniform average, and if the model uses layer normalisation, normalising an all-zero vector means dividing by a variance of zero.
- **Unrelated words initialized to point in (nearly) the same direction:** the model *starts out* believing they are interchangeable. Training can still pull them apart, but only with enough contrastive gradient signal; for rare words this signal may be weak, so the false similarity can persist for a long time, wasting model capacity undoing a wrong prior instead of learning new structure.

The common thread: initialization should be small, (pseudo-)random, and unstructured - it should not encode any prior claim about which words are related, since real relatedness must come from data via gradient descent, not from the initial values.

3. If d is too small, E does not have enough degrees of freedom to place all semantically distinct words at distinguishable positions - the model is forced to conflate words/relationships that should be kept apart, limiting what it can represent (underfitting). If d is too large, the embedding layer has many more trainable parameters (recall question 1: the count is $V \times d$, growing linearly with d) than the data may support, increasing the risk of overfitting and computational/memory cost, often with diminishing returns. In practice d is tuned as a hyperparameter, trading representational capacity against parameter count and overfitting risk.
4. Training only ever optimizes embeddings to help predict the next token given context - but the *statistics* of how words are actually used in language are systematically structured:

pairs like (king, man) and (queen, woman) tend to appear in parallel, analogous contexts (royal titles combined with gendered contexts). Gradient descent finds a geometry that efficiently captures such regularities, because encoding "gender" and "royalty" as roughly consistent directions is a compact way to explain many such parallel word pairs at once, which helps prediction generalize across all of them. Nobody hand-designed a "gender axis" - it emerges because the co-occurrence statistics of real language contain that structure, and a linear direction is often the simplest geometric object that can capture "the same kind of contextual shift" across many word pairs simultaneously. This is what nearest-neighbour clustering and analogy arithmetic expose.

5. Yes, both occurrences of "the" give the identical row $(1, 0, 0, 1)$ (or whatever E 's row 0 is), because the lookup depends only on token identity, not on position in the sentence. This alone cannot distinguish the two occurrences - that gap is exactly what positional encoding fills (Exercise 2.2).

Part II - Scaled dot-product attention by hand

Exercise 2.1 - From embeddings to queries, keys and values

quantity	shape
Q	(B, L, d_k)
K	(B, L, d_k)
V	(B, L, d_v)
$QK^\top$	(B, L, L)
A	(B, L, L)
$Z = AV$	(B, L, d_v)

1. No - the shape of Z is (B, L, d_v) ; it depends on d_v , not d_k . d_k only affects the (square) shape of the intermediate score matrix.
2. $Q = XW_Q$ with $X \in \mathbb{R}^{B \times L \times d_{\text{model}}}$ and $W_Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$ (applied identically to every token/batch element). Analogously, $K = XW_K$ with $W_K \in \mathbb{R}^{d_{\text{model}} \times d_k}$, and $V = XW_V$ with $W_V \in \mathbb{R}^{d_{\text{model}} \times d_v}$.

Exercise 2.2 - A three-token example

Step 1:

$$s_{\text{the}} = (0, 2, 0, 0) \cdot (2, 0, 0, 0) = 0$$

$$s_{\text{cat}} = (0, 2, 0, 0) \cdot (0, 2, 0, 0) = 4$$

$$s_{\text{sat}} = (0, 2, 0, 0) \cdot (0, 0, 2, 0) = 0$$

Step 2: dividing by $\sqrt{4} = 2$:

$$\tilde{s}_{\text{the}} = 0 \quad \tilde{s}_{\text{cat}} = 2 \quad \tilde{s}_{\text{sat}} = 0$$

Step 3:

i	$\tilde{s}_i$	$e^{\tilde{s}_i}$	a_i
the	0	1.00	0.107
cat	2	7.39	0.787
sat	0	1.00	0.107

Sum: $1.00 + 7.39 + 1.00 = 9.39$; weights sum to $0.107 + 0.787 + 0.107 \approx 1.00$, as required.

Step 4:

$$\begin{aligned} z_{\text{cat}} &\approx 0.107(1, 0) + 0.787(0, 1) + 0.107(1, 1) \\ &= (0.107 + 0.107, 0.787 + 0.107) \\ &\approx (0.214, 0.894) \end{aligned}$$

Step 5:

1. "cat" attends most strongly to **itself**, since $q_{\text{cat}} \cdot k_{\text{cat}} = 4$ is the largest of the three dot products - its query and key vector happen to be identical here, so it is maximally "similar to itself".
2. Yes, the weights would change. Scaling every key by c scales every score $q \cdot k$ by c as well, and softmax is **not** invariant to a uniform rescaling of its inputs (multiplying all logits by $c > 1$ sharpens the distribution towards the largest score; $c < 1$ flattens it).
3. If entries of q and k are $\mathcal{O}(1)$ and roughly independent, the dot product $q \cdot k = \sum_{m=1}^{d_k} q_m k_m$ is a sum of d_k roughly independent terms, so its variance grows linearly with d_k . Without rescaling, scores would grow in magnitude as d_k increases, pushing softmax into a saturated, near-one-hot regime with very small gradients. Dividing by $\sqrt{d_k}$ keeps the variance of the scores approximately constant regardless of d_k .
4. $q_{\text{cat}} \cdot k_{\text{mat}} = (0, 2, 0, 0) \cdot (1, 1, 1, 0) = 2$. This is smaller than $q_{\text{cat}} \cdot k_{\text{cat}} = 4$, so "cat" would still attend most strongly to itself, but the dot product with "mat" is now nonzero - unlike "the" and "sat" in the original three-token example, which both scored 0. Once q and k are learned from real text rather than handed to us, a nonzero query-key dot product like this would mean the model has learned that "cat" and "mat" are relevant to each other - e.g. because they co-occur systematically (a cat sitting *on* a mat), so attending to "mat" helps build a more informative representation of "cat" in context.

Exercise 2.3 - Why attention needs position information

1. If we permute the tokens with the same permutation π , every pairwise comparison $q_i \cdot k_j$ that existed before still exists after, just relabelled by π : the comparison between the *same two tokens* still happens, and produces the same value, only its row/column index in the score matrix changes - formally, $a_{\pi(i), \pi(j)}^{\text{new}} = a_{i,j}^{\text{old}}$.
2. No. Plain self-attention is **permutation equivariant**: permuting the input tokens permutes the output the same way, but the *content* of each token's output (given its neighbours) is unchanged. It carries no information about the original order.
3. E.g. "the dog bit the man" vs. "the man bit the dog" - identical bag of words, opposite meaning.
4. A positional encoding is added to the embedding **before** it is compared via attention, and it depends only on the position t , not on which word is there. Two identical words at different positions now receive different vectors (word embedding + different positional vector), so query/key comparisons - and hence attention output - become sensitive to where each token sits in the sequence.
5. GPT-2's position vectors are only trained for positions up to its maximum length (e.g. 1024): there is simply no learned vector for a position beyond that, so the model cannot properly process (or generalizes very poorly to) a sequence longer than what it was trained on. A fixed sinusoidal formula can be evaluated at any position, and RoPE's relative rotation can be applied for any offset, so both extrapolate more gracefully to sequence lengths unseen in training, at least in principle.

6. **(Optional/advanced)** What typically matters for how two tokens should influence each other is *how far apart* they are (e.g. "the word right before me" vs. "a word ten positions away"), not their absolute index in the document. If the attention score depends only on the relative distance $t_i - t_j$, then the same relative pattern behaves identically no matter where it occurs - at position 5 or position 50,000 - which lets patterns learned on shorter sequences transfer to longer ones. An absolute positional code, by contrast, ties learned behaviour to specific absolute indices, which do not generalize the same way to unseen lengths or offsets.

A natural worry is that, since each rotated dimension-pair is periodic, the score should occasionally "spike" back up once the relative distance passes a full wavelength - but this does not happen in practice: RoPE sums contributions from many dimension-pairs with geometrically spaced (and effectively incommensurate) frequencies, so they essentially never all realign in phase at once except at distance 0. The fast-rotating pairs only ever contribute one bounded term out of many to the sum, and the slow-rotating pairs barely complete a fraction of a full rotation within any realistic context length, so no coherent aliasing spike survives in the total.

Exercise 2.4 - Multi-head attention dimensionality

1. $d_k = 8/2 = 4$.
2. $(B, L, d_k) = (1, 5, 4)$ per head.
3. $(1, 5, 4)$ per head.
4. Concatenated: $(1, 5, 8)$.
5. After output projection: $(1, 5, 8)$ - $W_O \in \mathbb{R}^{8 \times 8}$ (in general $W_O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$), so it maps the concatenated heads' d_{model} -wide vector back to d_{model} , preserving the model dimension.
6. Concatenating h heads of width d_k must reconstruct exactly d_{model} ; if d_{model} were not divisible by h , there would be no way to split it into h equal-width heads (or one head would need a different width, breaking the uniform per-head computation).
7. With $h = 4$, $d_k = 8/4 = 2$ per head - it halves. Each head still attends over the same L keys as before (the *number* of values attended per head is unchanged), but each head represents its query/key/value comparisons in a narrower (2-dimensional instead of 4-dimensional) subspace; the total representational budget $d_{\text{model}} = 8$ is split more finely across more, narrower heads.

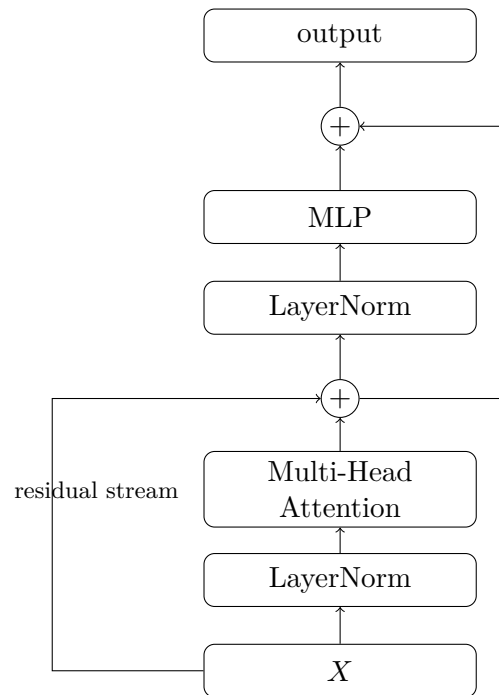
Exercise 2.5 - Induction heads (Optional/advanced)

1. Both roles require **attention**: the previous-token head must pull information from a *different* position (the one immediately before), and the induction head must search over *all* earlier positions to find a match and copy from a (generally distant, non-local) position. The MLP sub-layer acts identically and independently on each token's own vector and has no mechanism to read from any other position at all, so it structurally cannot perform either role - only attention mixes information between positions.
2. A single attention layer performs one comparison step: it reads Q/K/V computed from the current representations and produces one weighted combination. The circuit needs the previous-token head to first *write* "the token before me was A" into token B's representation, and only *then* can a later-layer induction head search for other positions whose *own* "previous token was B" signature matches, and copy what followed there. This is a composition where one head's output becomes part of the input read by another, later head - which requires two attention layers stacked in sequence; one attention operation cannot both "look one token back" and "search-and-copy from a matching earlier occurrence" at the same time.
3. No parameters are updated when this happens - it occurs within a single forward pass over the given prompt, using the model's already-trained, fixed previous-token and induction

heads to detect and exploit repetition that happens to be present in this particular input. The model's weights never change; what changes is that its existing, general-purpose circuitry finds and exploits a pattern specific to the current context. That is why it is called "in-context learning": the loss drop looks like learning, but no gradient descent or weight update is involved.

Part III - Transformer blocks, decoding, and scale

Exercise 3.1 - The transformer block and causal masking



1. The two "Add & Norm" steps are shown above: (i) X bypasses LayerNorm and Multi-Head Attention entirely, and is added directly to their output at the first "+"; (ii) the result of that first addition bypasses the second LayerNorm and the MLP, and is added directly to their output at the second "+". Both bypass paths together form the **residual stream** - the running vector each sub-layer only ever *adds* to, never replaces.
2. A residual connection means the *raw, pre-normalisation* input X is added elementwise to the sub-layer's output, where the sub-layer itself only ever sees the *normalised* input: $X' = X + \text{Sublayer}(\text{LN}(X))$. Only $\text{LN}(X)$ goes into the attention/MLP computation; the value added back is X itself, not $\text{LN}(X)$.
3. It gives gradients a direct, unimpeded path (an identity shortcut) back to earlier layers, so they do not have to pass through every nonlinear transformation at every stacked block. This is the same reasoning as for skip connections in deep ReLU networks: it mitigates vanishing gradients and makes very deep stacks trainable.
4. Setting disallowed scores to $-\infty$ before softmax guarantees that, after softmax, the corresponding weight is exactly 0 **while the remaining allowed weights still form a valid probability distribution that sums to 1** over exactly the allowed keys. If instead weights were zeroed *after* an unmasked softmax, they would no longer sum to 1 (an invalid distribution), and the masked positions would still have influenced the softmax normalisation and gradients during the forward/backward pass.
5. Bidirectional attention lets every token use both left and right context, which is exactly what is needed to understand the meaning of a whole, already-written sentence. For

left-to-right generation, however, a token must be predicted using only what came before it; allowing it to see future tokens during training would let the model trivially "cheat" by copying the answer, and that information would not exist yet at inference time.

Exercise 3.2 - The residual stream and the logit lens

1. If each block fully *replaced* $\mathbf{x}$ instead of adding to it, the representation at layer 5 would live in a completely different, layer-specific space than the representation at the final layer, with no reason to expect the *final* unembedding matrix (trained only to interpret the final layer's output) to produce anything meaningful when applied to it. Because every block only *adds* a comparatively small update to a running vector that starts as the token embedding and stays expressed in a broadly consistent basis throughout, the intermediate residual stream is already an evolving, partial approximation of the final answer, in the same coordinate system the final unembedding matrix expects - so reading it out early with that same matrix still gives a meaningful, if less refined, guess.
2. It suggests the correct prediction is not immediately available from the embedding alone - it is refined progressively as more blocks add their contribution to the residual stream, moving from a plausible-but-wrong guess (*pressure*, a generic "physics quantity") toward the correct, specific answer (*charge*) only in the later layers. Different depths seem to do different work: earlier layers narrow down a broad semantic category, later layers pin down the specific correct fact.
3. Because the MLP cannot exchange information between token positions, any "knowledge" it stores must be retrievable from a single token's own (already-attention-enriched) representation in context - e.g. factual associations of the form "given a context vector that already encodes '...electron has a negative...', output charge". It cannot itself decide *which* other tokens or context are relevant, gather information from elsewhere in the sequence, or match/copy patterns across positions - that is necessarily attention's job, since only attention mixes information across positions.

Exercise 3.3 - Counting parameters and compute

1. Attention projections: $4d^2$.
2. MLP: $2 \times d \times d_{\text{ff}}$; with $d_{\text{ff}} = 4d$: $2 \times d \times 4d = 8d^2$.
3. Total per block: $4d^2 + 8d^2 = 12d^2$.
4. $12d^2 = 12 \times 768^2 = 12 \times 589,824 = 7,077,888$ per block; for $N = 12$ blocks: $12 \times 7,077,888 = 84,934,656 \approx 84.9\text{M}$. Compared to GPT-2's reported 124.4M, about $124.4 - 84.9 \approx 39.5\text{M}$ is missing - this is essentially the token embedding matrix (38.6M, Exercise 1.2 Q1) plus the position embedding matrix ($1024 \times 768 \approx 0.8\text{M}$) plus the small remainder from biases and layer norms, none of which were counted in this block-only estimate.
5. No, the parameter count does **not** depend on L : all weight matrices (W_Q, W_K, W_V, W_O and the two MLP matrices) have fixed shapes independent of sequence length, applied/shared across all positions. The **computation and memory** do depend on L : the attention score matrix A has shape (B, h, L, L) , so both its memory footprint and the cost of computing it scale **quadratically** in L ; the MLP's cost scales linearly in L (applied independently per position).
6. **(Optional/advanced)** Once a token's key and value vectors are computed, they never change in later steps (they depend only on that token's own representation, fixed once it has been processed). If they are cached, generating the next token only requires computing one new query and comparing it against the already-known keys/values of all previous tokens - an $O(T)$ operation - instead of recomputing the entire $T \times T$ attention matrix (or recomputing every earlier token's key/value) from scratch, which would cost $O(T^2)$ overall across a whole generation. Reusing what does not change is what keeps the per-new-token cost linear in the number of tokens seen so far.

Exercise 3.4 - From logits to text (Optional/advanced)

1. As $T \rightarrow 0$, the exponent z_i/T blows up in magnitude for the largest logit relative to all others, so the softmax collapses onto the single most likely token - this is **greedy decoding** (equivalent to argmax). It can loop because greedy decoding has no randomness: once the model starts producing a phrase whose continuation makes the same phrase the argmax again (e.g. "the lunch menu today" recurring as the most likely continuation of itself), it will deterministically keep choosing that same continuation forever, with no mechanism to escape.
2. As T increases well above 1, the logits are divided by an increasingly large number before the softmax, flattening the distribution toward uniform and increasing the chance of sampling less-likely, more novel/surprising tokens. The tradeoff against $T \rightarrow 0$ is diversity/creativity vs. coherence: very high T increases the chance of incoherent, ungrammatical, or nonsensical output (as in the $T = 1.5$ example), while $T \rightarrow 0$ is maximally coherent but repetitive and deterministic.
3. When the model is very confident (mass concentrated on 1-2 tokens), top-p with a sensible p automatically keeps only those 1-2 tokens, adapting to the model's confidence, whereas a fixed top-k might force in additional, implausible tokens the model considered essentially impossible just to fill the fixed quota k . When the model is very unsure (mass spread over hundreds of tokens), top-p automatically expands the candidate set to preserve that genuine diversity, whereas a fixed top-k would arbitrarily truncate to exactly k candidates regardless of whether the true distribution needs more options. Top-p's candidate-set size adapts to context; top-k's does not.

Exercise 3.5 - Matching architectures to tasks (Optional/advanced)

task	architecture	justification
Sentiment classification of a movie review	encoder-only	Needs one representation summarising the whole (already complete) input using bidirectional context; no generation is required.
Autoregressive next-word text generation	decoder-only	Must produce tokens one at a time, each conditioned only on the tokens generated so far - exactly what causal self-attention provides.
Machine translation (English $\rightarrow$ German)	encoder-decoder	The source sentence benefits from full bidirectional encoding, while the target must be generated left-to-right, conditioned on the source via cross-attention.
Producing a single embedding vector to search a document database	encoder-only	Needs one fixed-size, context-aware representation of the full (complete) input; no autoregressive generation involved.

Summary answers

Statement	Answer
Real tokenizers use	byte-pair encoding (BPE) , trained once before the network, on subword units
An embedding matrix has shape	vocabulary size $\times$ embedding dimension
Plain self-attention, without extra input, is	permutation equivariant / order-blind
Scores are divided by	$\sqrt{d_k}$, to keep score variance roughly constant as d_k grows
Per-head key/query dimension	$d_k = d_{\text{model}}/h$
What is added to token embeddings to restore order information	positional encodings (fixed sin/cos, learned, or RoPE)
What masking technique makes a decoder generate strictly left-to-right	causal (look-ahead) masking , applied by setting disallowed scores to $-\infty$ before softmax
The residual stream is	the running per-token vector that every block <i>adds</i> to, rather than replaces
Cross-attention is used in	encoder-decoder architectures, letting the decoder attend to the full encoder output
Transformer weight-matrix parameter counts scale with	d_{model} and d_{ff} , not with sequence length L
Attention compute/memory scales with	L^2 (quadratically in sequence length)
The KV cache avoids recomputing	keys and values of earlier tokens, keeping per-new-token generation cost linear in tokens so far

Central teaching point

Attention mixes information across positions and is order-blind by itself; **positional information and causal masking** restore order and, for decoders, generation direction.

Every shape in a transformer block is fixed by three numbers - d_{model} , h (hence $d_k = d_{\text{model}}/h$) and d_{ff} - and is otherwise independent of sequence length; only the *cost* of attention grows with L , quadratically. This is the central bookkeeping students should be able to reproduce for any given configuration.

Question answers (short conceptual questions)

1. Self-attention compares queries and keys symmetrically with no notion of "where" a token sits, so permuting the input tokens simply permutes the output the same way - the computation itself carries no order information. This is fixed by (i) adding position-dependent positional encodings to the input embeddings (so token identity and position both matter to the encoder), and (ii) causal masking in the decoder, which enforces that generation happens strictly in left-to-right order.
2. Queries, keys and values play three different functional roles - "what this token is looking for" (query), "what this token offers to be matched against" (key), and "what this token

hands over once matched" (value). A single shared representation could not simultaneously be optimised for all three roles; separate learned linear projections let the model shape the same underlying embedding differently for each purpose.

3. $d_k = d_{\text{model}}/h$ grows if d_{model} increases with h fixed. A larger d_k gives each head a higher-dimensional subspace to represent its query/key/value comparisons in, i.e. more capacity per head to encode finer-grained or more complex relationships.
4. Layer normalisation computes its mean/variance over the feature dimension of a single token, independent of the batch and of other tokens, so it behaves identically regardless of batch size or sequence length and remains well-defined even for variable-length sequences or a batch size of 1. Batch normalisation's statistics are computed across the batch, which is unstable for small batches and does not transfer sensibly across positions in a sequence of varying length.
5. If a decoder used bidirectional self-attention during training, position i could directly see the token it is being trained to predict (since that token is present in the full, unmasked sequence), so the model could trivially copy it and the training loss would collapse without the model ever learning the actual predictive relationship. At inference time, however, future tokens do not exist yet, so a model trained this way would fail catastrophically compared to training.
6. Only the self-attention sub-layer mixes information *between* different token positions. The MLP is applied identically and independently to each position's own vector (point-wise), so it can transform a token's representation but cannot exchange information with any other token.
7. **(Optional/advanced)** Freezing pretrained embeddings is preferable when the downstream training set is small (too little data/signal to safely update $V \times d$ parameters without overfitting or destroying the general-purpose structure learned on a much larger pretraining corpus) or when the downstream vocabulary/domain closely matches what the embeddings were pretrained on. Fine-tuning is preferable when there is enough downstream data to update the embeddings safely and the task benefits from task-specific adjustments - e.g. words that are pretrained as generically similar but need to be pulled apart (or together) for this particular task, or domain-specific vocabulary/usage that differs from the pretraining corpus.
8. **(Optional/advanced)** $p_i \propto e^{z_i/T} = \left(e^{1/T}\right)^{z_i} = a^{z_i}$ with $a = e^{1/T}$ - so temperature sampling is exactly a softmax computed in base $a = e^{1/T}$ applied directly to the raw logits. Low temperature ($T < 1$) gives $1/T > 1$, so $a = e^{1/T} > e$: a larger base than e amplifies the relative differences between logits when exponentiated, sharpening the distribution - consistent with $T \rightarrow 0$ approaching a one-hot/argmax distribution. High temperature ($T > 1$) gives $1/T < 1$, so $a = e^{1/T}$ sits between 1 and e (approaching 1 as $T \rightarrow \infty$); a base close to 1 makes the exponential nearly flat regardless of the exponent, which flattens the distribution toward uniform - consistent with high T increasing randomness/diversity.