# 1 Problem

We have a sequence of tokens $x_0, ..., x_{T-1}$ ($x_i \in \mathbb{R}^d$). We want a model that can understand the tokens as an entire set of objects interacting with each other. "Understand" is intentionally vague — what we're building towards will model relationships often in an uninterpretable way to us, as an intermediate inside a larger model.

(notational note: $\mathbb{R}^n$ refers to a row vector of length $n$, not a column vector; ie, shape $[n]$, not shape $[n, 1]$.)

# 2 Attention

Pack our tokens into rows of a tensor: $X$ has shape $[T, d]$. Attention takes $X$ in and outputs $X'$ of shape $[T, d_v]$, where each output $\mathbb{R}^{d_v}$ vector is a "contextually enhanced" feature vector with info from the others — the purpose of attention is to allow for information flow between tokens among the entire sequence. And when we say "information flow", that sounds vague, but what we really mean is simply that information flows from one variable $x$ to another variable $y$ iff a change in $x$ can cause a change in $y$.

Attention owns parameters $W_Q, W_K, W_V$, which are used to project tokens in $\mathbb{R}^d$ to tokens in a separate feature space.

- $W_Q$ and $W_K$ have shape $[d, d_k]$. (aka project to $\mathbb{R}^{d_k}$)
- $W_V$ has shape $[d, d_v]$. (aka project to $\mathbb{R}^{d_v}$)

The actual mechanism:

- For token $i$, we want to create a corresponding "query" $q_i$ and a "key" $k_i$ in a new feature space $\mathbb{R}^{d_k}$.
  - So let $q_i = x_i W_Q$ and $k_i = x_i W_K$. Vectorized, we use $\boxed{Q = XW_Q}$ and $\boxed{K = XW_K}$ in practice. $Q, K$ both have shape $[T, d_k]$.
- Now for token $i$, we want to see how relevant each other token $j$ is to $i$.
  - $q_i$ is the $\mathbb{R}^{d_k}$ vector that $i$ is looking for; $k_j$ is the $\mathbb{R}^{d_k}$ vector that token $j$ is offering.
  - So we score all $j \in [0, T-1]$ from $i$'s perspective, using dot product $q_i \cdot k_j$ and producing a score vector $s_i[j] = q_i \cdot k_j$ with shape $[T]$. The scores are then normalized with $s_i := \text{softmax}\left(\frac{s_i}{\sqrt{d_k}}\right)$ (divide by $\sqrt{d_k}$ to counter softmax saturation — normalize logits against expected variance of $q \cdot k$ which is $d_k$ — typical logit magnitude is one stdev $\sqrt{d_k}$).
  - So the interpretation of $s_i[j]$ becomes a proportion in $[0, 1]$ that says "proportionally, how relevant is token $j$ to token $i$, compared to all the other tokens in the sequence?"
  - Vectorized across all tokens in the sequence instead of just $i$, we have $\boxed{S = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)}$ of shape $[T, T]$ (softmax on rows, to be clear). (we use $QK^T$ to dot rows of $Q$ against rows of $K$ together for $q_i \cdot k_j$. The $i$th row of $Q$ ($q_i$) dots against all rows of $K$ ($k_j$) to produce the $i$th row of $S$ ($s_i$).)
- Then, for token $i$, we want to create a contextually enhanced feature vector $x'_i$ that encodes $x_i$ along with its relationship with the rest of the sequence.
  - $W_V$ transforms our $\mathbb{R}^d$ vectors into a $\mathbb{R}^{d_v}$ feature space with $v_i = x_i W_V$. Vectorized: $\boxed{V = XW_V}$ of shape $[T, d_v]$.
  - Then we form token $i$'s output feature vector according to how much the other tokens $j$ are relevant to $i$: $x'_i = \sum_{j=0}^{T-1} s_i[j] v_j$.
  - Vectorized, it's $\boxed{X' = SV}$ of shape $[T, d_v]$. ($SV$ takes each row in $S$ ($s_i$) and uses those to weight the rows of $V$, with $s_i[j]$ being the weight for the $j$th row of $V$.)
- In the end, we have the classic attention formula $\boxed{X' = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V}$.

Conceptual notes:

- The feature spaces $\mathbb{R}^{d_k}$ and $\mathbb{R}^{d_v}$ are not necessarily interpretable, especially true if $W_Q, W_K, W_V$ are learned and the training labels don't have a clear simple / direct relationship with these feature spaces.
- Attention may currently seem a bit too simple to create meaningfully useful contextually enhanced vectors — they're just linear combinations of $v_j$ vectors, which is the part that actually matters for expressibility. This is true, which is why we don't use attention alone; it's a central mechanism within a larger system (which will be covered in the "Transformers" section). Think of attention as the mechanism for basic info flow between tokens in the sequence, with nonlinear MLPs on the output of attention creating more complex features, which can then be fed back to attention to create further contextual connections.

- Why $QKV$? What does "what it's looking for" vs. "what it's advertising" vs. "what it offers" even mean? Sounds vague and stupid and handwavy. But these are really just simplified analogies for allowing attention to control two distinct knobs in information flow: $q_i \cdot k_j$ is *how much* info flows, $v_j$ is *what* info flows. We believe "how much" vs. "what" are two entirely different questions, so we give attention a separate tool for each of these knobs.

# 3 Multi-head attention

Currently, attention only packages a single $v_j$ for *all* receiver tokens $i$ in the sequence. This is a problem — what if one token $i_1$ cares about feature $x_1 \in \mathbb{R}^{d_v}$ and token $i_2$ cares about feature $x_2 \in \mathbb{R}^{d_v}$? $v_j$ can bundle feature $x_1$, but then token $i_2$ can't get its $x_2$ target. $v_j$ can then bundle feature $x_2$ instead, but then token $i_1$ can't get its $x_1$ target. And bundling both just introduces $v_j$'s $x_1$ noise when contributing to $i_2$ and $v_j$'s $x_2$ noise when contributing to $i_1$. The problem is simply that when we use scalar weighting, we're bundling all features together when not all receiver tokens care about the same set of features, and all compromises that exist suck. One attention function is incapable of distributing receiver-dependent info.

So relax the stress on a single attention call; run multiple attention calls on the same token sequence in parallel.

- We have $h$ "heads", each of which is just an individual attention call $X'^{(m)} = \text{Attn}_m(X)$ with its own attention parameters $W_Q^{(m)}, W_K^{(m)}, W_V^{(m)}$. The shapes are all still the same — for example, $X'^{(m)}$ is still shape $[T, d_v]$.
- Then, concatenate all these $X'^{(m)}$ together along the feature axis to get a concat tensor $C$ of shape $[T, hd_v]$.
- Feed this concat tensor $C$ through a matmul against $W_O$ (shape $[hd_v, d_{out}]$) to mix the $hd_v$ features together for each row in $C$ into an output $d_{out}$ features, outputting a final result $\boxed{\text{MHA}(X) = CW_O}$.

In practice, we usually use $d_k = d_v = \dfrac{d}{h}$ — heads are factorization, not copied cost.

Conceptual notes:

- Why not just use even more transformer blocks? Each block handles a couple of features, and the receiver tokens who don't care about those features can just wait for a future block that deals with features they care about.
  - That actually can roughly achieve the same results — not identical obviously, but there's nothing blaringly wrong with it. The only constraints here are that deeper networks are harder to train, MHA is parallelizable, and MHA is more efficient than one-block-per-feature because it's effectively $\frac{d}{h}$ entries per feature analysis in MHA but $d$ entries per feature analysis in one-block-per-feature.
- Also... as usual, we create multiple heads with the hope that the model will find that specializing them each to smaller tasks decreases loss the best, but we can't guarantee that it will, or that the way it specializes the heads is interpretable to us.

# 4 Transformers

A transformer is a bunch of transformer blocks chained in a sequence. A **transformer block** modifies a sequence of tokens to add contextual enhancement to each one. Specifically:

- Takes in a sequence of tokens $x_0, ..., x_{T-1}$ ($x_i \in \mathbb{R}^d$)
- Outputs a "contextually enhanced" sequence of tokens $x'_0, ..., x'_{T-1}$ ($x'_i \in \mathbb{R}^d$)

The sequence of events for a block:

- Let info flow across the sequence. Use multihead attention: $\boxed{X := X + \text{MHA}(\text{LN}(X))}$. Since we need to add MHA's result to $X$, we set $d_{out} = d$ for MHA.
- Mix the features nonlinearly for each $\mathbb{R}^d$ vector into something more useful for the next transformer block: $\boxed{X := X + \text{MLP}(\text{LN}(X))}$. The standard MLP is $\text{MLP}(X) = \sigma(XW_1 + b_1)W_2 + b_2$ where $W_1$ has shape $[d, d_{ff}]$ and $W_2$ has shape $[d_{ff}, d]$ — project each feature vector in $X$ to $\mathbb{R}^{d_{ff}}$ (intuition being that the expansion gives the MLP more space to compute features before compressing back), and then compress it back down to $\mathbb{R}^d$ again. Typically the nonlinearity $\sigma$ is GELU, though ReLU was the original standard and works fine too.
- *Notes:*
  - **Residual connections**: We use residual connections (ie $x := x + f(x)$ instead of $x := f(x)$) because transformers generally stack a large number of blocks in a sequence, causing the earlier blocks to have less of an effect on the model's output for the same parameter perturbations compared to the later blocks (verifiable via the long gradient product in chain rule of composed

functions $x_n = f_n(f_{n-1}(...(f_1(x_1))))$ which causes gradients to either exponentially decay or explode), under the replacement model. The residual stream instead allows each module to contribute to the output at a relatively equal degree by adding their outputs directly to the residual stream. Expanded, residual connections give us $x_n = x_1 + f_1(x_1) + ... + f_{n-1}(x_{n-1})$; say we want $\frac{\partial J}{\partial \theta_i}$. Then, since $\theta_i$ enters computation only at $x_{i+1} = x_i + f_i(x_i)$, we have $\frac{\partial J}{\partial \theta_i} = \frac{\partial J}{\partial x_{i+1}} \frac{\partial x_{i+1}}{\partial \theta_i}$, and since $\frac{\partial J}{\partial x_{i+1}} = \frac{\partial J}{\partial x_n} \cdot \frac{\partial x_n}{\partial x_{n-1}} \cdot ... \cdot \frac{\partial x_{i+2}}{\partial x_{i+1}}$, our final form of $\frac{\partial J}{\partial \theta_i}$ is $\frac{\partial J}{\partial x_n} \prod_{j=i+1}^{n-1} \left( \frac{\partial x_{j+1}}{\partial x_j} \right) \frac{\partial x_{i+1}}{\partial \theta_i}$. Since $\frac{\partial x_{j+1}}{\partial x_j} = I + J_{f_j}$, the middle product in the chain rule isn't affected by repeated multiplications of small jacobians, and thus the size of the gradient for module $i$'s parameters is now less strongly influenced by module $i$'s position in the model. **And critically**: remember that zeroed gradients are a symptom of bad info flow from earlier modules, not some kind of numerical bug we're band-aiding; we choose the residual stream to improve info flow, and gradients look normal again as a downstream consequence.

– **LayerNorm**: We use LayerNorm (denoted LN) to prevent our activations from growing too much in magnitude over time; this also does help against gradient explosion which is unaddressed by residual connections. Residual connections actually cause our $\mathbb{R}^d$ vectors to increase in magnitude on average (the vector's magnitude decreases in the small number of cases where $f_i(x)$ is actively anti-aligned with $x$), so we make sure every read of a vector in a transformer block gives us a normalized vector. Without LayerNorm, imagine each MHA or MLP call outputting a roughly similar-in-magnitude vector; we'd approximately double $X$'s magnitude upon every update which ends up growing exponentially, roughly like $O(2^{\text{num. updates}})$; LayerNorm turns it into something more linear like $O(\text{num. updates})$ in the worst case.

Conceptual notes:

- As always, remember that the learnable internals of the model (the vectors output from the first transformer block all the way up to the vectors that are input to the last transformer block, the weights, etc) are not necessarily interpretable at all — only spec'd data like the input and output vectors are. Mechanistic interpretability can discover structures post-hoc, but these are not guaranteed to exist.

# 5 Summary

A transformer takes in a sequence of tokens $X$ of shape $[T, d]$ (each row is a token), feeds $X$ through a large number of sequential chained transformer blocks, and outputs a contextually-enhanced sequence of tokens $X'$ of the same shape $[T, d]$, which are not necessarily interpretable relative to the original vocab — again, depends on where the training objective is specified, and what it's specified to be.

Transformers are typically a "good default" as an internal component of a model for data which decomposes into a collection of homogeneous elements with complex and unknown interactions with each other. They're great because they're so generalizable. But some areas where they fail:

- **Tiny datasets**: transformers generally need lots of data.
- **Exact algorithmic computation**: transformers have fixed depth, so the same number of steps occur for every problem. Though, this is a limitation which something like reasoning in LLMs overcomes, by increasing the number of transformer block ops.
- **Heterogeneous data**: aka the tokens aren't the same kind of "thing". For example, in tabular data, if your tokens are columns, then each token is a separate thing: age, income, zip code, etc. (also, if you make your tokens be rows, the rows of tabular data are typically just independent samples that don't have relationships with each other, so a transformer's ability to model complex interactions between tokens is wasted. this is why transformers suck at tabular data)