Contextual Representations

and Transformers

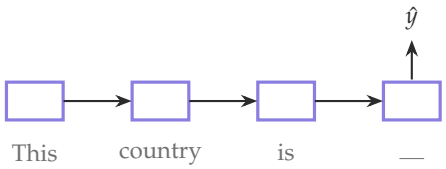
DS 4440 · LECTURE 13

1. Autoregressive generative models

Model the next token given everything before it, and a whole sequence as the product of those conditionals:

$$P(x_t \mid x_{t-1} \dots x_1), \qquad P(x) = \prod_t P(x_t \mid x_{<t}).$$

For example **language modelling**:



Language modelling is a kind of **self-supervised** learning: the labels come from the data itself, so no annotation is needed. That is what makes it usable for **pre-training**.

stage	what you need	
pre-train	unlabelled data (e.g. the internet) + self-supervised loss	
fine-tune	the pretrained net + a loss $\mathcal{L}$ and labelled (x, y)	
in-context learning	just (x, y), at inference time	(later)

2. Contextualized word vectors

Recall from the 10/5 lecture that word2vec gives an embedding *per word type*:

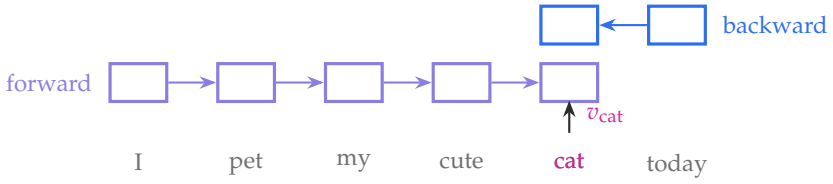
$$\text{“cat”} \xrightarrow{\text{w2v}} [\,x\ \ y\ \ z\,]$$

and that vector is **static** — always $[x\ y\ z]$, in “cute cat” and in “fat cat” alike. The word’s meaning in context is not represented at all.

> **Idea.** Train an RNN for language modelling, then extract its **contextualized embeddings**: the hidden state at a position depends on the whole sentence, so the same word in two sentences gets two different vectors.

ELMo: Embeddings from Language Models [Peters et al., 2018]

Run two LSTMs over the sentence, one in each direction:



Each direction is trained as a language model — predict the token from the context on *its* side:

$$\mathcal{L} \;=\; -\log P\big(x_t \mid \overrightarrow{h}_{t-1}\big) \;-\; \log P\big(x_t \mid \overleftarrow{h}_{t+1}\big) \qquad \text{self-supervised!}$$

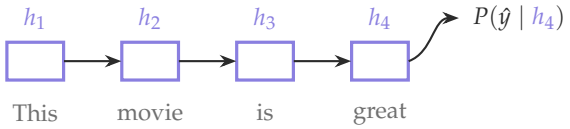
and the representation of a token is its static embedding concatenated with both directions' states at that position:

$$\text{ELMo}_{\text{cat}} \;=\; \big[\, v_{\text{cat}} \,;\, \overrightarrow{h}_t \,;\, \overleftarrow{h}_t \,\big].$$

These vectors can then be used as features for downstream tasks, or the whole thing fine-tuned. It works well. **But** ELMo is built on RNNs, so it is slow: the state at position t cannot be computed until position $t-1$ is done, so there is no way to parallelize across the sequence.

3. Attention

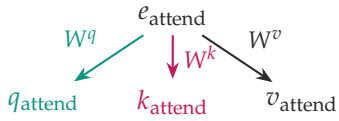
Enter **transformers**, which do away with recurrence using **self-attention**. Start with the plain version. Given hidden states $h_1 \dots h_m$ from an RNN, rather than predicting from the last one, use a *weighted combination* of all of them:



$$a_j \;=\; \underbrace{W_a}_{\langle 1\times h\rangle} \cdot \underbrace{h_j}_{\langle h\rangle}, \qquad \alpha \;=\; \text{softmax}(a), \qquad \bar{h} \;=\; \sum_{j=1}^{m} \alpha_j h_j, \qquad P(y \mid \bar{h}).$$

Generalizing: queries, keys and values

Instead of one learned vector W_a, let each position produce a **query** saying what it is looking for, a **key** advertising what it has, and a **value** carrying what it passes on. All three are linear maps of the same embedding:



$$a_j \;=\; S\big(k_j, q\big), \qquad \alpha \;=\; \text{softmax}(a), \qquad \bar{h} \;=\; \sum_j \alpha_j v_j.$$

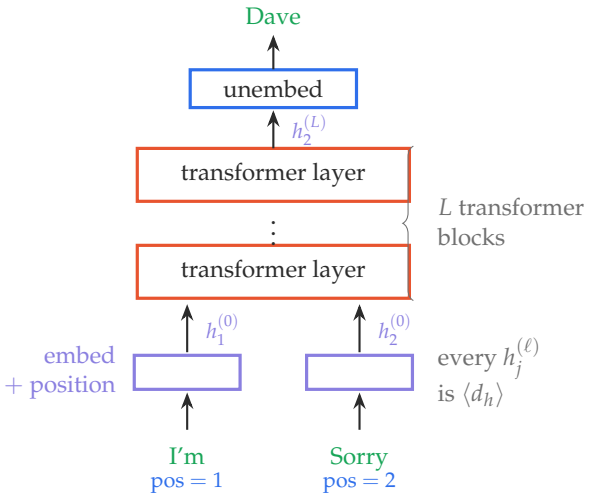
Two choices for the score S:

$$\underbrace{\frac{k_j \cdot q}{\sqrt{d}}}_{\text{dot-product attention}} \qquad\qquad \underbrace{v_j^\top \tanh\big(W_k k_j + W_q q\big)}_{\text{MLP attention}}$$

The $\sqrt{d}$ is there because a dot product of d-dimensional vectors grows with d; without it the softmax saturates and the gradient dies.

4. Transformers

Tokens in, tokens out. Assume autoregressive generation: a prefix (the context) maps to the next token.



Here we assume fixed positional embeddings added to the token embeddings; current models would use RoPE or similar.

Unembed

The easy part — one linear map from the last hidden state to a score per vocabulary item:

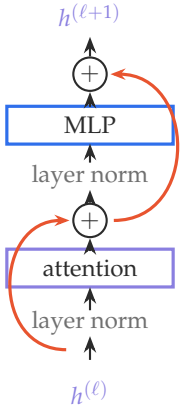
$$\hat{y} = \underbrace{W_U}_{\langle V \times d_h \rangle} h_N^{(L)} + \underbrace{b_U}_{\langle V \rangle} \quad \text{(logits)}, \qquad y = \text{softmax}(\hat{y}) \quad \text{(a distribution over the vocabulary)}.$$

Recall $\text{softmax}(y)_i = \exp\{y_i\} / \sum_j \exp\{y_j\}$. We can add a **temperature**:

$$\frac{\exp\{y_i/\tau\}}{\sum_j \exp\{y_j/\tau\}}, \qquad \tau > 1 \Rightarrow \text{flatter, more random;} \qquad \tau \to 0 \Rightarrow \text{argmax.}$$

5. Inside a transformer layer

Two components: **self-attention** and an **MLP**, each wrapped in a skip connection.



Layer norm keeps the magnitude of the hidden states under control, which eases optimization:

$$\mathrm{LN}(h_i)_k \;=\; \gamma_k \frac{h_{ik} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} + \beta_k, \qquad \gamma, \beta \text{ learned}$$

where μ_i and σ_i^2 are the mean and variance of the feature values *for token* i, taken over h_i. Note the contrast with batch norm from Lecture 10: there we normalized each feature across the batch; here we normalize each token across its features, so nothing depends on which other examples are in the batch.

Self-attention, in the layer

Every token produces a query, a key and a value, and every token's new representation is a weighted average of all the values:

$$a_{ij} \;=\; S(q_i, k_j), \qquad \alpha_i \;=\; \mathrm{softmax}(a_{i\cdot}) \quad \text{(the attention distribution for token } i\text{)}, \qquad \bar{h}_i \;=\; \sum_j \alpha_{ij} v_j.$$

For the token "attend" in *self attend to words*, that means four scores and one softmax over them:

$$\begin{array}{ll} \text{self} & S(q_{\text{attend}}, k_{\text{self}}) \\ \text{attend} & S(q_{\text{attend}}, k_{\text{attend}}) \\ \text{to} & S(q_{\text{attend}}, k_{\text{to}}) \\ \text{words} & S(q_{\text{attend}}, k_{\text{words}}) \end{array} \quad \longrightarrow \text{ softmax } \longrightarrow \quad \alpha^{\text{attend}}$$

$$\tilde{x}_{\text{attend}} \;=\; \sum_j \alpha_j^{\text{attend}} v_j$$

> **Causal masking.** For autoregressive generation, token i must not see tokens after it — otherwise predicting the next token is trivial. So before the softmax we set $a_{ij} = -\infty$ for all $j > i$, which sends those weights to zero.

One attention layer, start to finish

Three tokens — *the cat sat* — and $d = 2$, so everything fits on a line. Take this as an *encoder* layer for the moment: no mask, every token sees every other. Suppose the projections W^q, W^k, W^v have produced

	q		k		v	
the	1	0	0	1	1	1
cat	0	1	2	0	0.2	0
sat	2	0	0	1	0	2

Scores. $S_{ij} = q_i \cdot k_j / \sqrt{d}$ with $\sqrt{d} = \sqrt{2}$:

$$S \;=\; \begin{bmatrix} 0 & 1.414 & 0 \\ 0.707 & 0 & 0.707 \\ 0 & 2.828 & 0 \end{bmatrix} \qquad \text{(row } i\text{: what token } i \text{ is asking for)}$$

S is *not symmetric*: $S_{12} = 1.414$ while $S_{21} = 0.707$. That is the point of having two projections instead of one. If we scored with a single vector per token, $S_{ij} = h_i \cdot h_j$, the matrix would be symmetric by construction — and worse, its largest entry in row i would usually be the diagonal $\|h_i\|^2$, so every token would mostly attend to itself. Splitting q from k lets a token look for something *unlike* itself, which is normally what we want: a verb looking for its subject is not looking for another verb.

Softmax each row:

$$\alpha = \begin{bmatrix} 0.164 & 0.673 & 0.164 \\ 0.401 & 0.198 & 0.401 \\ 0.053 & 0.894 & 0.053 \end{bmatrix}, \qquad \bar{h}_i = \sum_j \alpha_{ij} v_j = \begin{bmatrix} 0.298 & 0.491 \\ 0.441 & 1.203 \\ 0.232 & 0.159 \end{bmatrix}$$

Now look at $\bar{h}_3$, where the third projection earns its place. Its attention distribution says this position is overwhelmingly "about" cat, but split the sum by source:

$$\underbrace{0.894\, v_{\text{cat}} = \begin{bmatrix} 0.179 \\ 0 \end{bmatrix}}_{\text{89\% of the weight}} \qquad \underbrace{0.053\, v_{\text{the}} + 0.053\, v_{\text{sat}} = \begin{bmatrix} 0.053 \\ 0.159 \end{bmatrix}}_{\text{11\% of the weight}}$$

The second coordinate of the output comes *entirely* from the two tokens that barely got any attention, because v_{cat} is small in that direction. Attention weight decides how much of each value gets mixed in; the value decides what there is to mix. Different questions, hence different matrices.

> Three projections, three jobs: k advertises what a token offers, q states what it wants, and v carries what gets passed along. Collapse any two and something is lost — $q = k$ makes the score matrix symmetric and diagonal-heavy, and folding v into k ties "how relevant am I" to "what do I contribute."

The same example with the causal mask

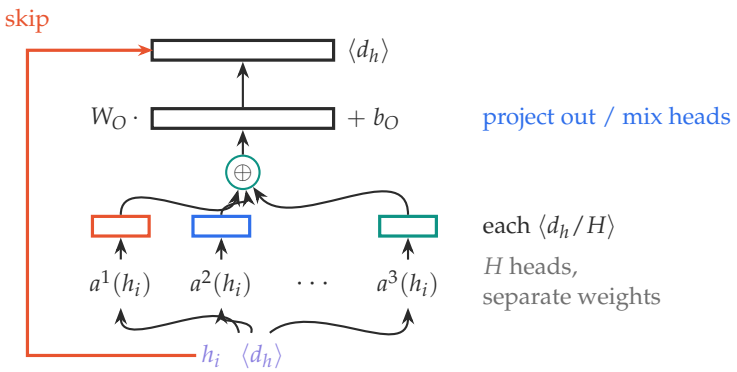
If this were a decoder layer we would zero the upper triangle of S first and renormalize what is left:

no mask (encoder)	**causal mask (decoder)**
$\begin{bmatrix} 0.164 & 0.673 & 0.164 \\ 0.401 & 0.198 & 0.401 \\ 0.053 & 0.894 & 0.053 \end{bmatrix}$	$\begin{bmatrix} \mathbf{1} & 0 & 0 \\ 0.670 & 0.330 & 0 \\ 0.053 & 0.894 & 0.053 \end{bmatrix}$
$\bar{h}_1 = [\,0.298 \;\; 0.491\,]$	$\bar{h}_1 = [\,1 \;\; 1\,]$
$\bar{h}_2 = [\,0.441 \;\; 1.203\,]$	$\bar{h}_2 = [\,0.736 \;\; 0.670\,]$
$\bar{h}_3 = [\,0.232 \;\; 0.159\,]$	$\bar{h}_3 = [\,0.232 \;\; 0.159\,]$

Two things to notice. Token 1's row collapses to 1 — with nothing else to look at, its output is just its own value, whatever the scores said. And the last row is *identical* in both: token 3 could already see everything, so masking changes nothing for it. Masking only ever restricts the earlier positions.

Multi-head attention

Define H heads with independent W_q^h, W_k^h, W_v^h, each of shape $\langle d_h \times d_k \rangle$ with $d_k = d_h / H$. The heads run in parallel on the same input, their outputs are concatenated, and a final W_O mixes them back to d_h:



The MLP

Just a position-wise feed-forward network — the same two-layer MLP applied to each token's vector independently, widening to more than d_h in the middle and back:

$$d_h \longrightarrow \sigma(\,\cdot\,) \text{ at } > d_h \longrightarrow d_h.$$

Nothing here mixes positions; that is attention's job.

6. Implications for interpretability

- **Attention heads route** — or mix — information between tokens. They move things sideways across the sequence.
- **MLPs transform** individual tokens, nonlinearly, with no communication between positions.

Every operation in the block is one or the other, which is a useful thing to hold onto when asking what a particular layer is doing.