Attention: Queries, Keys, and Values

Lecture Notes Consolidated with Claude AI — CS6140 Machine Learning

---

*Prerequisites: soft k-NN / kernel density (**LOCAL_CLASSIFIERS** note); softmax and matrix products (**NN_BASICS**); the **RNN** note's translator (padding, teacher forcing, greedy decoding, BLEU).*

Attention lets each vector look up the others: a query is compared with keys, the similarities become weights that sum to one, and the output is the weighted average of the values.

1 The Big Picture: One Vector Is Not Enough

HW6's translator read the Spanish sentence with a GRU and kept only the encoder's **final state**: one vector of 512 numbers. Every English word was generated from that single vector (and the words already produced). For a four-word sentence that works. For a twelve-word one, the vector has to hold the whole sentence at once, and BLEU drops:

Spanish length (tokens)	1–4	5–6	7–8	9–12
HW6 GRU translator, BLEU (reference run)	32.4	37.0	30.1	18.6

(Very short sentences score a little lower than 5–6 tokens for a separate reason: one wrong word already breaks most 3- and 4-grams.) A human translator does not memorize the sentence and then write; they *look back* at the word they are translating. Attention gives the decoder that ability.

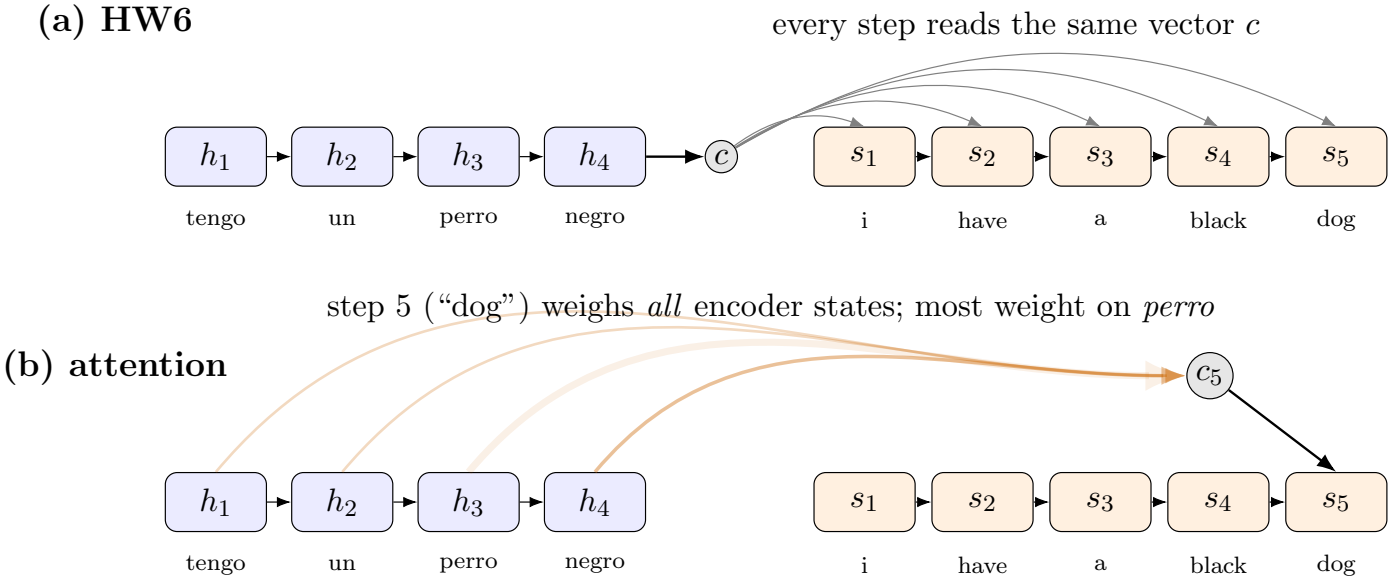


Figure 1: (a) HW6's encoder–decoder: the decoder sees the source only through one fixed vector. (b) With attention, each decoder step t builds its own context $c_t = \sum_s \alpha_{ts} h_s$, a weighted average of all encoder states.

The whole note is about one formula. For a query q and a set of key–value pairs $(k_1, v_1), \ldots, (k_m, v_m)$:

$$\text{Attention}(q; \{(k_j, v_j)\}) = \sum_{j=1}^{m} \alpha_j \, v_j, \qquad \alpha_j = \frac{\exp\big(a(q, k_j)\big)}{\sum_{j'} \exp\big(a(q, k_{j'})\big)}, \tag{1}$$

where $a(q, k)$ is a **score**: a number that is large when q and k are similar. In words: score every key against the query, turn the scores into weights with a softmax (non-negative, summing to 1), and return the weighted average of the values.

Part	**Where you have seen it**	**What changes**
Weighted average with similarity weights	Soft k-NN / KDE (`LOCAL_CLASSIFIERS`); kernel regression	The similarity is learned
Softmax turns scores into weights	Multiclass logistic regression; NN_BASICS	Over keys, not classes
Dot product and cosine as similarity	`LOCAL_CLASSIFIERS`; word vectors (TOKENS)	Between learned projections
Gaussian and other kernels	Weeks 2 and 5–6 (KDE, kernels)	Only used as motivation
Padding mask	HW6 Problem 4's masked loss	Masks keys inside the softmax
Encoder states h_s, decoder state s_t	HW6 Problem 4	Kept: we add one step on top
Queries, keys, values; $\sqrt{d}$ scaling	**NEW** (§4)	
Several heads at once	**NEW** (§6)	
Positional encoding; causal mask	**NEW** (§8, §9)	

Table 1: The parts of attention.

2 Flashback: Attention Is a Kernel Smoother You Already Know

Soft k-NN, again. In `LOCAL_CLASSIFIERS` (the KDE section) we classified a query $\mathbf{x}$ by letting *every* training point vote, with a vote of size $k(\mathbf{x}, \mathbf{x}_i)$ instead of 1:

$$\hat{P}(C \mid \mathbf{x}) = \frac{\sum_{i:\, y_i = C} k(\mathbf{x}, \mathbf{x}_i)}{\sum_i k(\mathbf{x}, \mathbf{x}_i)} = \sum_i \underbrace{\frac{k(\mathbf{x}, \mathbf{x}_i)}{\sum_{i'} k(\mathbf{x}, \mathbf{x}_{i'})}}_{\alpha_i} [y_i = C].$$

The second form is Eq. (1): the weights α_i are similarities normalized to sum to one, and the "value" of point i is its one-hot label. In that note's spam example, points A, B (non-spam) and C (spam) had kernel values $0.951, 0.908, 0.994$, so $\alpha = (0.333, 0.318, 0.348)$ and $\hat{P}(\text{non-spam}) = 0.333 + 0.318 = 0.65$. That is an attention computation.

Kernel regression (Nadaraya–Watson) does the same with real-valued y_i:

$$\text{pred}(z) = \sum_i \alpha(z, x_i) \, y_i, \qquad \alpha(z, x_i) = \frac{k(z, x_i)}{\sum_{i'} k(z, x_{i'})}. \tag{2}$$

The prediction at z is an average of the training targets, with the most weight on the training points closest to z (Figure 2). Your instructor's board puts it as: *"α high $\Rightarrow$ pred(z) should pay **attention** to x_i"* (after the board, Lecture 8/6 p5).

Worked example: one query, three keys Keys $x = (1, 2, 4)$, values $y = (10, 20, 40)$, query $z = 2.5$, Gaussian kernel $k = e^{-(z-x)^2/2}$. The kernel values are $(0.3247, 0.8825, 0.3247)$ (the query is 1.5 away from both 1 and 4), the weights $\alpha = (0.212, 0.576, 0.212)$, and pred$(2.5) = 0.212{\cdot}10 + 0.576{\cdot}20 + 0.212{\cdot}40 = 22.12$. Most of the answer comes from the nearest key.

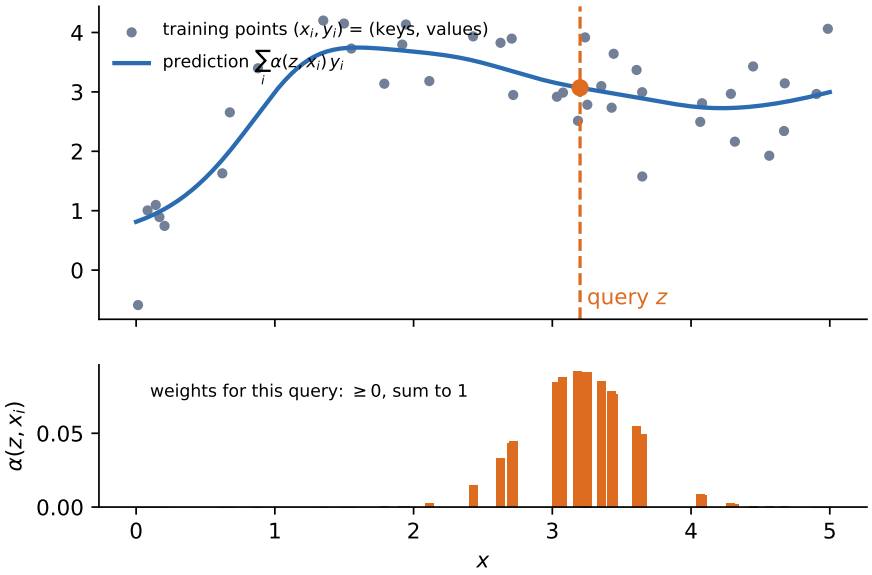


Figure 2: Kernel regression as attention. Top: 40 training points and the smoothed prediction. Bottom: the weights $\alpha(z, x_i)$ for the query $z = 3.2$: only points near z matter. Every query gets its own set of weights.

The database picture. A Python dictionary does a *hard* lookup: the query must equal one key exactly, and you get that key's value. Attention is a *soft* lookup: the query is compared with every key, and you get a blend of all values, weighted by how well each key matches.

Key point: Attention = kernel smoothing with three upgrades. (1) The **query**, **keys** and **values** are vectors computed by the network, not fixed data. (2) The similarity is **learned** (through linear projections, §6). (3) The whole computation is differentiable, so backprop trains it together with everything else. Kernel and score are related by $k(q, k_i) = \exp\big(a(q, k_i)\big)$: the network computes the *score* a (a log-kernel), the softmax exponentiates it and normalizes so that $\sum_i \alpha_i = 1$.

3 Scores Are Similarities: from the Gaussian Kernel to the Dot Product

Any similarity can serve as the score, as long as larger means "more alike". The board lists the usual kernels (Figure 3): Gaussian, narrow Gaussian, boxcar, triangle. All of them weigh near keys heavily and far keys little or not at all; the width decides how many keys share the weight, exactly as the bandwidth did in KDE.

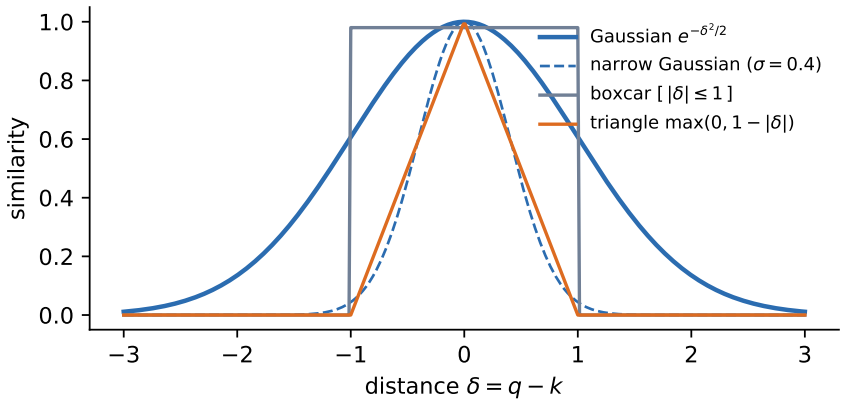


Figure 3: Kernel shapes (after the board, Lecture 8/6 p8). Attention mostly uses the Gaussian in disguise: the dot product.

From the Gaussian to the dot product (after the board, Lecture 8/6 p7). Take the Gaussian kernel $k(q,k) = \exp(-\frac{1}{2}\|q-k\|^2)$; its *log*, $a(q,k) = -\frac{1}{2}\|q-k\|^2$, is the score that goes into the softmax (the softmax does the exponentiating). Expand the square:

$$-\tfrac{1}{2}\|q-k\|^2 = q^\top k - \tfrac{1}{2}\|k\|^2 - \tfrac{1}{2}\|q\|^2.$$

- $\frac{1}{2}\|q\|^2$ is the *same for every key*, so it cancels *exactly* in the softmax (adding a constant to all scores does not change the weights).
- $\frac{1}{2}\|k\|^2$ differs from key to key, so it cancels only when all keys have the *same length*. Then the weights depend on $q^\top k$ alone.

With $q = (1,2)$ and keys $(1,1),(2,0),(0,3)$ the Gaussian weights are $(0.574, 0.078, 0.348)$, while the plain dot product gives $(0.047, 0.017, 0.936)$: the long key $(0,3)$ wins. Rescale the three keys to a common length and the two agree exactly. So the dot product is "the Gaussian kernel when key lengths don't vary"; in a trained network the projections are free to use length as well as direction, and nothing is lost.

The two scores in use:

Score	Formula	When
Dot product (scaled)	$a(q,k) = q^\top k/\sqrt{d}$	q, k of the same size; one matrix product for all pairs, fast on a GPU. Transformers.
Additive (Bahdanau)	$a(q,k) = w_v^\top \tanh(W_q q + W_k k)$	q, k of different sizes; a one-hidden-layer network on the pair. The HW7 translator (§10).

4 Scaled Dot-Product Attention

Stack the queries as the rows of Q, the keys as the rows of K, the values as the rows of V (rows are items, as for every data matrix in this course). All query–key scores are one matrix product, and attention for all queries at once is

$$\boxed{\text{Attention}(Q,K,V) = \text{softmax}\Big(\frac{QK^\top}{\sqrt{d}}\Big)V} \tag{3}$$

with the softmax taken along each row (over the keys). Row i of the result is query i's weighted average of the values.

Tensor	Shape (one sentence)	Shape (batch, as in the code)
Q	$T_q \times d$	`(B, T_q, d)`
K	$T_k \times d$	`(B, T_k, d)`
V	$T_k \times d_v$	`(B, T_k, d_v)`
scores $QK^\top/\sqrt{d}$, weights A	$T_q \times T_k$	`(B, T_q, T_k)`
output AV	$T_q \times d_v$	`(B, T_q, d_v)`

Read the shapes as a sentence: *every query* (T_q rows) *gets a weight for every key* (T_k columns), *and a* d_v*-dimensional answer*. In a batch, use `K.transpose(-2,-1)`, not `K.T`, which would also flip the batch axis.

Worked example: two tokens, $d = 4$ $Q = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$, $K = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$, $V = I_2$. Then $QK^\top = \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}$, divided by $\sqrt{4} = 2$: $\begin{bmatrix} 1 & 0.5 \\ 0 & 0.5 \end{bmatrix}$. Row-wise softmax: $e^1/(e^1 + e^{0.5}) = 0.6225$, so

$$A = \begin{bmatrix} 0.6225 & 0.3775 \\ 0.3775 & 0.6225 \end{bmatrix}, \qquad AV = A \ \text{ (since } V = I\text{)}.$$

Query 1 matches key 1 best and puts 62% of its weight there; query 2 prefers key 2. This is the first checkpoint of HW7 Problem 2(A).

Algorithm: scaled dot-product attention (with a mask)

1. scores $\leftarrow QK^\top/\sqrt{d}$ `(B, T_q, T_k)`
2. where the mask says "blocked", set the score to $-\infty$ (in practice -10^9) (§5)
3. $A \leftarrow$ softmax of each row of the scores (over the keys)
4. return AV `(B, T_q, d_v)`, and A for plotting

4.1 Why divide by $\sqrt{d}$?

Suppose the entries of q and k are independent with mean 0 and variance 1. Then $q^\top k = \sum_{j=1}^{d} q_j k_j$ is a sum of d products, each with variance 1, so $\mathrm{Var}(q^\top k) = d$ and its standard deviation is $\sqrt{d}$. As d grows, the scores of different keys spread further apart, and the softmax of widely spread scores is nearly one-hot: it *saturates*. Dividing by $\sqrt{d}$ puts the standard deviation back at 1 for every d.

Why saturation hurts: the softmax's derivative with respect to its scores is $\partial w/\partial s = \mathrm{diag}(w) - ww^\top$, which is the zero matrix when w is one-hot. Every gradient that reaches the query and key projections passes through this factor, so a saturated softmax passes back only a weak learning signal. HW7 Problem 2(B) measures this with 10 random keys (2,000 trials per d); the reference run gives:

d	unscaled $q^\top k$ score std	unscaled $q^\top k$ largest weight	unscaled $q^\top k$ sensitivity	scaled $q^\top k/\sqrt{d}$ score std	scaled $q^\top k/\sqrt{d}$ largest weight	scaled $q^\top k/\sqrt{d}$ sensitivity
4	2.00	0.49	0.337	1.00	0.30	0.340
16	4.02	0.72	0.268	1.01	0.32	0.346
64	7.93	0.87	0.158	0.99	0.32	0.347
256	16.02	0.93	0.093	1.00	0.32	0.348

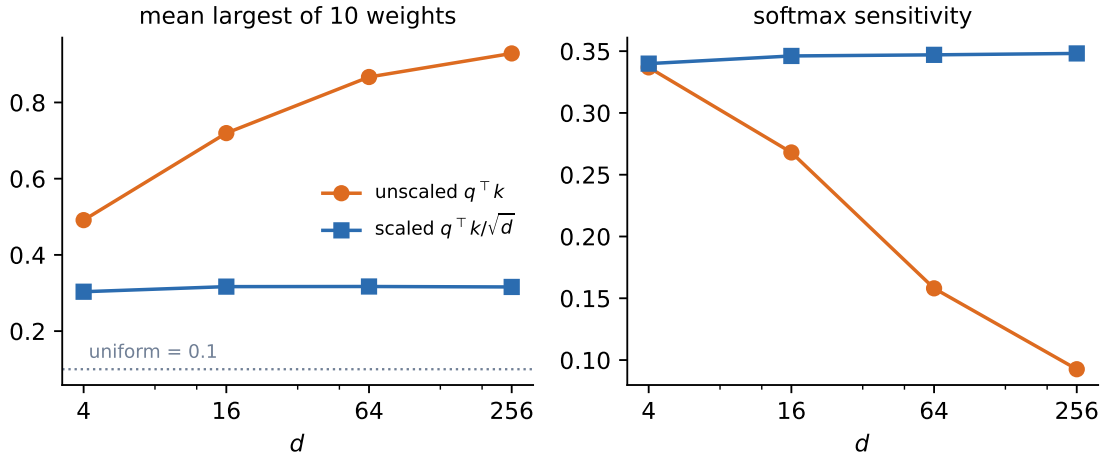


Figure 4: Without scaling, the softmax drifts towards a hard choice of one key as d grows, and its sensitivity (the size of $\partial w/\partial s$) shrinks about $4\times$. With scaling, both stay put.

"Sensitivity" is how much the weights move when the scores move; it is a factor inside every gradient to W_q and W_k, not the full gradient. Read the table as: *unscaled attention in a wide model starts out almost as a hard lookup, and learns slowly.* The scaling keeps it soft whatever the width of the heads.

5 Masking: Keys That Must Not Be Seen

Sentences in a batch have different lengths, so they are padded (HW6). A query must not put weight on a padding key. The rule: **set the blocked scores to $-\infty$ *before* the softmax**. Then $e^{-\infty} = 0$, the blocked keys get exactly zero weight, and the remaining weights still sum to 1.

Worked example: only the first key is real Same Q, K as above, but the second key is padding (valid length 1). The masked scores are $[[1, -\infty], [0, -\infty]]$ and the weights $[[1, 0], [1, 0]]$: all weight on the only real key.

Why not zero the weights after the softmax? Because the padding key has already taken its share of the normalization: zeroing it leaves rows $(0.6225, 0)$ and $(0.3775, 0)$, which no longer sum to 1, and the answer still depends on what was in the padding. One more caveat: a row in which *every* key is blocked has nothing to average; every query must keep at least one valid key (in the HW it always does).

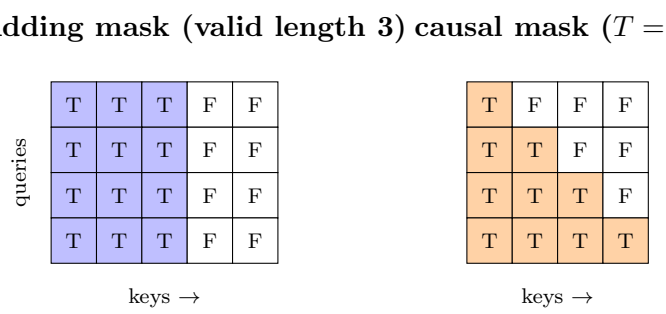


Figure 5: The two masks of HW7, in the HW's convention: **True = may attend**. Left: the last two keys are padding, for every query. Right: query i may see keys $0 \dots i$ only (§9).

One idea, four conventions. Every library implements "blocked scores become $-\infty$", but they disagree on how the mask is written. HW7 uses one convention throughout and gives helpers that convert:

API	Mask given as	Meaning
HW7's own code; `F.scaled_dot_product_attention`	boolean	**True = may attend**
`nn.MultiheadAttention`, `nn.Transformer*`	boolean `attn_mask`, `key_padding_mask`	**True = blocked** (the opposite!)
`nn.Transformer.generate_square_subsequent_mask`	float, added to the scores	0 = may attend, $-\infty$ = blocked
d2l `masked_softmax`	`valid_lens` (lengths)	positions $\geq$ length are blocked

A flipped mask is the most common silent bug in Transformer code: the model still trains, just badly, because it attends to the padding and ignores the words.

6 Multi-Head Attention: Several Lookups at Once

So far Q, K, V were given. Inside a network they are *computed* from the input by learned linear maps (after the board, Lecture 8/6 p12):

$$Q = X_q W_q, \qquad K = X_k W_k, \qquad V = X_k W_v.$$

W_q decides what a position is *looking for* ("the current decoder demand", in the board's words), W_k what each position *offers to be matched on*, W_v what it *hands over* when it is chosen. Learning these three matrices is learning the similarity.

One attention gives one weighted average per query. **Multi-head attention** runs h attentions side by side, each on its own d/h-dimensional slice of the projections, then concatenates the h answers and mixes them with one more matrix W_o:

$$\text{head}_i = \text{Attention}\big(Q^{(i)}, K^{(i)}, V^{(i)}\big), \qquad \text{MHA} = [\text{head}_1, \dots, \text{head}_h]\, W_o, \tag{4}$$

where each head scales its scores by $\sqrt{d_{\text{head}}} = \sqrt{d/h}$, the width of the vectors it actually compares (not by $\sqrt{d}$ of the whole model). where $Q^{(i)}$ is columns $(i-1)\frac{d}{h} + 1 \dots i\frac{d}{h}$ of Q (likewise K, V). Why several heads help is §11; here is how it is computed.

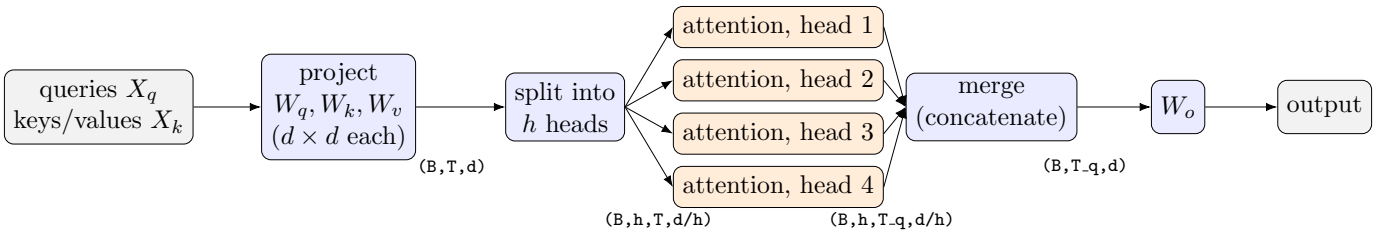


Figure 6: Multi-head attention. All heads run in one batched call: the head axis is treated as one more batch axis.

The shape walk. This is where most code breaks, so here it is with the numbers of the HW's checkpoint ($B = 3$ sentences, $T = 6$ tokens, $d = 16$, $h = 4$ heads, so $d/h = 4$):

Step	Shape	What happened
input X	`(3, 6, 16)`	6 tokens, 16 features each
project: XW_q	`(3, 6, 16)`	one 16×16 matrix for all heads together
reshape: cut the last axis into 4×4	`(3, 6, 4, 4)`	features 0–3 = head 0, 4–7 = head 1, …
transpose axes 1 and 2	`(3, 4, 6, 4)`	head axis next to the batch axis
attention (§4) on the last two axes	`(3, 4, 6, 4)`	each head: 6×6 weights; scores divided by $\sqrt{4}$, the *head* width
transpose back, reshape	`(3, 6, 16)`	concatenate the 4 heads per token
W_o	`(3, 6, 16)`	mix the heads

The transpose is not optional. Reshaping `(3,6,16)` directly to `(3,4,6,4)` produces the right shape with the wrong content: in the HW's checkpoint tensor, "head 1, position 0" would then hold features 8–11 of *position 1*, so a head would mix different tokens. The correct reshape-then-transpose gives features 4–7 of position 0. This is why the HW checks the split before anything else.

Parameters. W_q, W_k, W_v, W_o are $d \times d$ with biases: $4(d^2 + d)$, which is 1,088 at $d = 16$ and 263,168 at $d = 256$. This does *not* depend on h: four heads of size 64 cost exactly as much as one head of size 256. PyTorch's `nn.MultiheadAttention` stores W_q, W_k, W_v stacked in one `in_proj_weight`; the HW's helper copies yours in so the outputs can be compared.

7 Self-Attention, and Why It Ignores Word Order

In the translator the queries come from the decoder and the keys from the encoder. In **self-attention** all three come from the same sequence: $Q = XW_q$, $K = XW_k$, $V = XW_v$. Every token looks at every token of its own sentence, including itself, and its output is a new vector that mixes in what it found. "Bank" next to "river" and "bank" next to "money" start with the same word vector (TOKENS note) and leave self-attention with different ones: the vectors become *contextual*.

Your instructor's board (Lecture 8/8 p2): *"each token attends each step; no sequential [computation] ⇒ new mechanism instead of recurrence; run in parallel ⇒ AI cap race"*. An RNN must process position 1 before position 2; self-attention computes all positions in one matrix product, which is what GPUs are built for:

Layer, sequence of T tokens, width d	Work per layer	Sequential steps	Longest path between two tokens
Convolution, kernel k	$O(k\,T\,d^2)$	$O(1)$	$O(T/k)$
RNN	$O(T\,d^2)$	$O(T)$	$O(T)$
Self-attention	$O(T^2 d)$	$O(1)$	$O(1)$

(d2l 11.6 has the same comparison.) The price is the T^2: every pair of tokens is scored.

The order problem. Look again at Eq. (3): the weight between tokens i and j depends only on their *vectors*, never on their positions. Shuffle the rows of X and the rows of the output are shuffled in exactly the same way, and are otherwise unchanged. This is **permutation equivariance**: "dog bites man" and "man bites dog" produce the same set of vectors. HW7 Problem 2(D) checks it on a random input: shuffled input, shuffled output, difference 10^{-16}. For language that is unacceptable, so we must add the positions on purpose.

8 Positional Encoding: Giving Every Position a Timestamp

The fix is to add a fixed vector $P_i \in \mathbb{R}^d$ to the input of position i: the network receives $X + P$. Then the same word at two positions is two different vectors. Which vectors? The board's idea (Lecture 8/8 p8): **count in binary, but continuously**. In a binary counter (Figure 7, left) the last bit flips every step, the next every 2 steps, the next every 4: each column is a wave of a different frequency, and together the columns give every position a unique pattern. The sinusoidal encoding (Vaswani et al., 2017) does the same with smooth waves: for $j = 0, \ldots, d/2 - 1$,

$$P_{i,2j} = \sin(\omega_j i), \qquad P_{i,2j+1} = \cos(\omega_j i), \qquad \omega_j = \frac{1}{10000^{2j/d}}. \tag{5}$$

The first columns oscillate fast (like the last bit), the last ones very slowly (like the first bit).

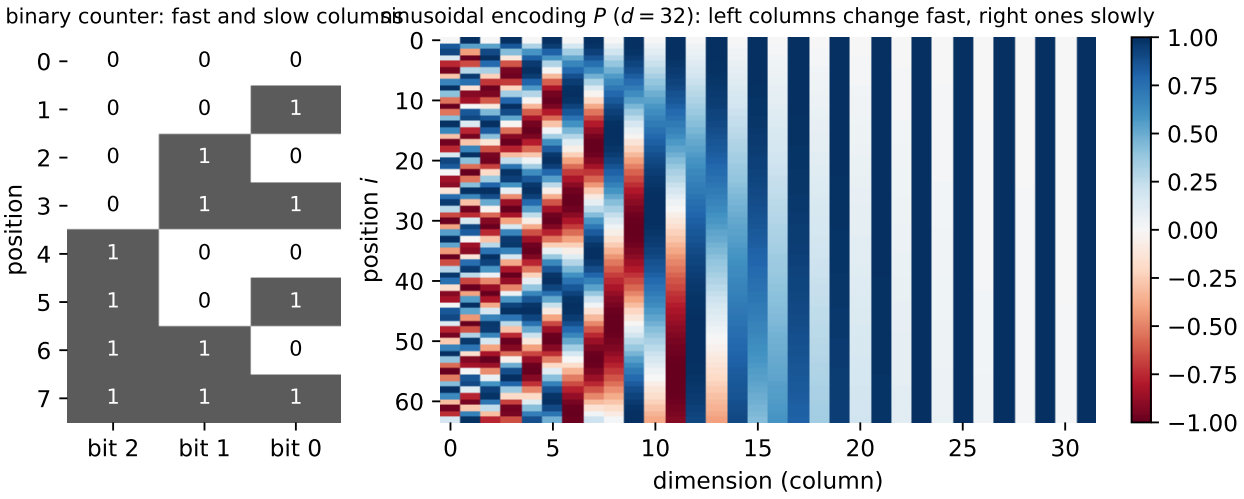


Figure 7: Left: a 3-bit counter for positions 0–7. Right: the sinusoidal encoding for 64 positions and $d = 32$, a continuous version of the same pattern (after the board, Lecture 8/8 p8).

Worked example: $d = 4$ Here $\omega_0 = 1$ and $\omega_1 = 1/100$, so row i is $(\sin i, \cos i, \sin(i/100), \cos(i/100))$ for positions 0–3:

$$P_0 = (0,\ 1,\ 0,\ 1), \qquad P_1 = (0.8415,\ 0.5403,\ 0.0100,\ 1.0000),$$
$$P_2 = (0.9093,\ -0.4161,\ 0.0200,\ 0.9998), \qquad P_3 = (0.1411,\ -0.9900,\ 0.0300,\ 0.9996).$$

The first pair of columns changes a lot from row to row; the second pair barely moves. This is HW7 Problem 2(D)'s checkpoint.

Relative positions are rotations. Each pair of columns $(\sin \omega_j i, \cos \omega_j i)$ is a point on a circle at angle $\omega_j i$. Moving δ positions forward turns it by the angle $\omega_j \delta$, whatever i is:

$$\begin{bmatrix} P_{i+\delta,2j} \\ P_{i+\delta,2j+1} \end{bmatrix} = \begin{bmatrix} \cos \omega_j \delta & \sin \omega_j \delta \\ -\sin \omega_j \delta & \cos \omega_j \delta \end{bmatrix} \begin{bmatrix} P_{i,2j} \\ P_{i,2j+1} \end{bmatrix}$$

(the sine and cosine addition formulas; for $j = 0$, $i = 2$, $\delta = 1$ it maps P_2's first pair to $(0.1411, -0.9900)$, which is P_3's). So "three words later" looks the same everywhere in the sentence, and a linear layer can learn to use it. This rotation idea returns in modern LLMs as RoPE (**TRANSFORMER_ADVANCED**).

Key point: Think of P_i as a **timestamp** added to each token. Attention itself still compares only vectors; now the vectors carry their positions, and because every Transformer block adds its result to its input (a residual connection, as in HW6's ResNet), the timestamp survives through the stack.

9 The Causal Mask: No Peeking at the Future

A language model predicts the next word. If position i could attend to position $i+1$, it would simply read the answer. The **causal mask** (Figure 5, right) is lower-triangular: query i may attend to keys $0, \ldots, i$ and nothing later. It is applied exactly like the padding mask, as $-\infty$ before the softmax, and the two can be combined.

Worked example: the running mean Four positions with values $v = (1, 2, 3, 4)$ and all scores equal to 0. Row i of the weights is uniform over the allowed keys: (1), $(\frac{1}{2}, \frac{1}{2})$, $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$, $(\frac{1}{4}, \ldots)$. The outputs are $1, 1.5, 2, 2.5$: the *running mean* of the values. And if you change v_4, outputs 1–3 do not move. Both are checkpoints in HW7 Problem 2(D).

The causal mask is what makes training a language model efficient: one forward pass trains *every* position at once, each predicting its own next word without seeing it. This is HW6's teacher forcing, done in parallel. GPT (**TRANSFORMER** note) is self-attention + positions + this mask.

10 Attention Inside the HW6 Translator

Now back to Figure 1. HW7 Problem 3 keeps HW6's encoder and decoder *unchanged* and adds one attention step on top of the decoder (the wiring of Luong et al., 2015):

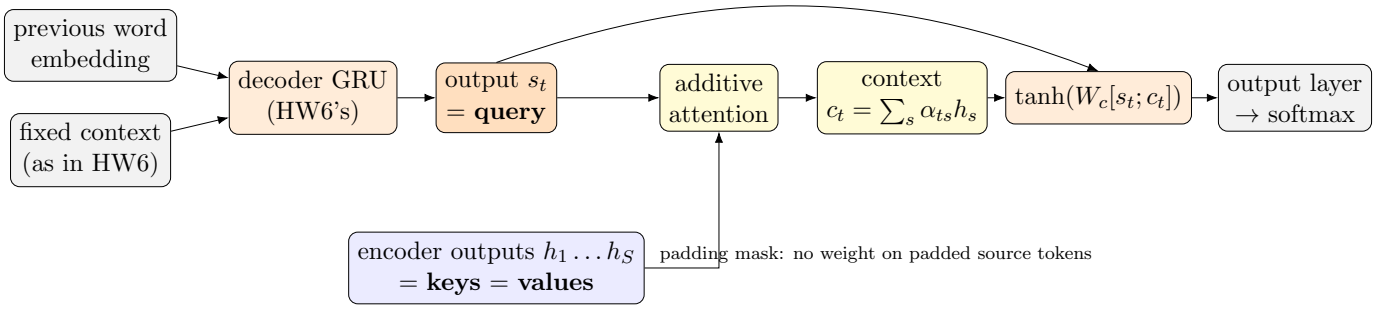


Figure 8: HW7's attention decoder: HW6's decoder (left) plus one attention step. Only the yellow boxes and W_c are new.

1. The GRU runs exactly as in HW6, giving an output s_t at every target position.

2. **Score** s_t against every encoder output with the additive score, $a(s_t, h_s) = w_v^\top \tanh(W_q s_t + W_k h_s)$; padding keys are masked.

3. **Weights** $\alpha_{ts} = \text{softmax over } s$; **context** $c_t = \sum_s \alpha_{ts} h_s$.

4. **Combine** $\tilde{s}_t = \tanh(W_c[s_t; c_t])$ and feed $\tilde{s}_t$ to HW6's output layer.

In the board's words (Lecture 8/6 p10): *"at decoder step t, which encoder steps are most relevant ⇒ high $\alpha(t, s)$"*; the letters t (target step) and s (source position) are the board's.

All pairs at once, by broadcasting. The additive score for every (query, key) pair is one tensor expression. Project the queries to `(B, T_q, 1, n)` and the keys to `(B, 1, T_k, n)`; adding them broadcasts to `(B, T_q, T_k, n)`, one n-vector per pair; tanh and w_v turn each into a single score, `(B, T_q, T_k)`. Write the three steps as three named tensors and check each shape. HW6's encoder outputs are time-major `(T, B, H)`; they are swapped to batch-major once, in the given code.

Why this wiring. The query is the GRU's *output*, so during training all target positions are still computed in one GRU call, as fast as HW6. Only the attention step is new, so the comparison with HW6 is about attention and nothing else. And "the decoder's current state queries all encoder outputs" is exactly the Transformer's **cross-attention** (TRANSFORMER note).

Remark. **Bahdanau's original wiring** (Bahdanau, Cho & Bengio, 2015; the board, Lecture 8/6 pp 9–11; d2l 11.4) uses the same attention in a different place: the query is the decoder's *previous* state, and the context is fed into the GRU's *input* together with the previous word. Then the GRU must run one step at a time even in training. Same weighted average of encoder states; only where it enters differs.

What the reference run shows (HW6's sizes, data and 2,500 steps):

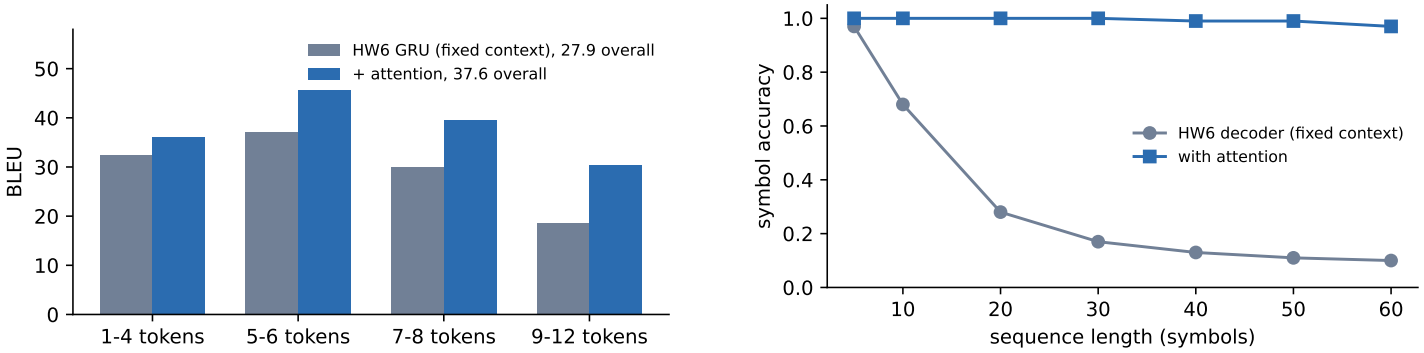


Figure 9: Left: BLEU by Spanish sentence length, HW6's GRU vs the same model with attention (reference run). Right: the reverse-a-sequence task, symbol accuracy by length.

- **Translation:** BLEU about 27.9 → 37.6, better in every length bucket, and most on the longest sentences (about 18.6 → 30.3 on 9–12 tokens).

- **Reversing a random sequence** of 5–60 symbols, a task where every output symbol depends on one specific input position. Without attention, accuracy collapses with length: about 97% at 5 symbols, 68% at 10, 28% at 20, 10% at 60. With attention it stays near 100% up to 60. (The encoder here is bidirectional, so that every encoder output also knows how far it is from the end.)

Careful with the conclusion: these runs show that, *at this budget*, squeezing a long sequence into one vector is hard to learn, while looking things up directly is easy. They do not prove that a fixed vector *cannot* hold the information; a much larger model trained much longer might.

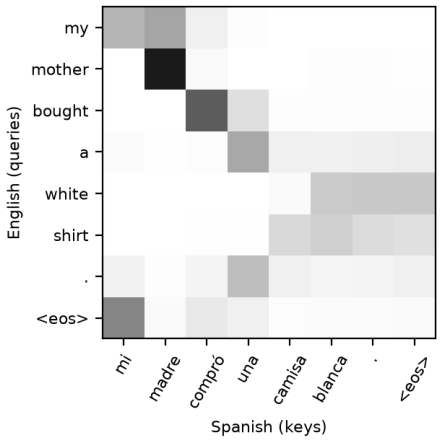


Figure 10: Attention weights of the reference model for *mi madre compró una camisa blanca .* → *my mother bought a white shirt .* (rows: English words being written; columns: Spanish tokens; darker = more weight).

Alignments are soft. The model was never told which Spanish word matches which English word, yet some rows are clearly aligned: *mother*→*madre* (weight 0.91), *bought*→*compró*, *a*→*una*. Others are not. *white* and *shirt*, where Spanish has the opposite order (*camisa blanca*), spread their weight over the end of the sentence instead of picking one word, and *my* already looks partly at *madre*. The largest weight in a row averages about 0.5 here. That is expected: a left-to-right encoder's later outputs summarize the sentence so far, so "looking near the end" is a reasonable strategy, and the model only needs weights that help the loss, not weights that look like a dictionary.

Reading your own maps: for each English word, find the Spanish token with the largest weight and how large that weight is; then look for the rows that do *not* match a dictionary (word-order swaps, words that come from a verb, rows spread over the end) and ask what the encoder output at that position already contains.

11 Why Several Heads?

One head computes, for each query, one softmax and therefore one weighted average. A query that needs two different things, say the Spanish word being translated *and* its neighbour that decides the word order, has a problem. Keep the total width fixed at $d = 2$ and take two words whose value vectors are $v_1 = (1, 0)$ and $v_2 = (0, 1)$.

- **One head of width 2.** Its weights $(\alpha, 1 - \alpha)$ give $(\alpha,\ 1 - \alpha)$: it can take word 1's feature fully *or* word 2's fully, or a blend in which each is diluted, e.g. $(0.5, 0.5)$. It can never return $(1, 1)$.
- **Two heads of width 1**, same total width. Head 1's value projection keeps the first coordinate (values 1 and 0), head 2's the second (0 and 1); each head has its own query and key projections, so head 1 can put all its weight on word 1 and head 2 on word 2. The concatenation is $(1, 1)$: both words, each at full strength.

Same parameters ($4(d^2 + d)$ either way, §6), more patterns: several separate weight distributions instead of one. Each head also has its own $W_q^{(i)}, W_k^{(i)}$, i.e. its own notion of similarity: one can match by meaning, another by position. The price is that each head compares queries and keys in a smaller space (d/h dimensions). Nobody assigns the heads their roles; they specialize because that lowers the loss, so look at *your* maps to see what they did.

12 HW7 Map: Problems 2, 3 and 6

- **2(A) masked softmax and scaled dot-product attention:** §4–§5; the 2-token example and the valid-length-1 case are the checkpoints. Your masks use True = may attend.
- **2(B) why $\sqrt{d}$:** §4.1. Explain the table in terms of the score's spread, saturation, and the sensitivity factor.
- **2(C) multi-head attention:** §6; follow the shape walk; the split is checked first.
- **2(D) order, positions, causality:** §7–§9.
- **3(A) the attention step:** §10 (broadcasting, the combine step); checked against the d2l book's `AdditiveAttention`.
- **3(B) alignments** and **3(C) where attention pays off:** §10 (what the maps can and cannot show; what the reverse task does and does not prove).
- **6 (theory) several heads:** §11, with your own Problem 4 cross-attention maps.

Common bugs.

- Dividing the scores by d instead of $\sqrt{d}$; taking the softmax over the query axis (rows must sum to 1 over the *keys*).
- Masking after the softmax; a mask with the opposite convention (True = blocked) passed to your own code, or yours passed to `nn.MultiheadAttention`.
- `K.T` on a batched tensor; splitting heads with a reshape and no transpose; merging them back in the wrong order.
- The padding mask with the wrong shape: `(B, 1, T_k)` for 3-D scores, `(B, 1, 1, T_k)` once the head axis is there.
- Positional encoding with sines and cosines swapped between even and odd columns, or the frequency inverted.
- In the translator: using HW6's time-major encoder outputs as batch-major; scoring against padded source tokens.

Take-home points

- Attention is a soft lookup: score the query against every key, softmax, average the values. It is the kernel smoother of Week 2 with a learned similarity.
- $\text{softmax}(QK^\top/\sqrt{d})V$; the $\sqrt{d}$ keeps the softmax from saturating as the width grows.
- Masks set blocked scores to $-\infty$ before the softmax: padding masks and the causal mask.
- Multi-head attention runs several smaller lookups at the same cost, so one query can gather several different things.
- Self-attention ignores order; positional encodings add a timestamp to every token.
- In the translator, attention replaces "remember the whole sentence" by "look back at it": about $28 \to 38$ BLEU, and a task where the fixed context collapses becomes easy.

References

- d2l book, Chapter 11 (§11.1–11.6): attention mechanisms, Bahdanau attention, multi-head attention, self-attention and positional encoding.
- D. Bahdanau, K. Cho, Y. Bengio (2015), *Neural Machine Translation by Jointly Learning to Align and Translate*, ICLR.
- M.-T. Luong, H. Pham, C. D. Manning (2015), *Effective Approaches to Attention-based Neural Machine Translation*, EMNLP.
- A. Vaswani et al. (2017), *Attention Is All You Need*, NeurIPS.
- C. Olah, S. Carter (2016), *Attention and Augmented Recurrent Neural Networks*, Distill (optional reading, with good visualizations).