Chapter 6: The Attention Revolution

Learning Objectives

After reading this chapter, the reader should be able to:

1. Explain why compressing a variable-length input sequence into a single fixed-size vector creates an information bottleneck that degrades performance on long sequences.
2. Derive the Bahdanau (additive) attention mechanism, computing alignment scores, attention weights via softmax, and the context vector as a weighted sum of encoder hidden states.
3. Contrast Bahdanau (additive) and Luong (multiplicative) attention, identifying when each variant is preferred and their computational tradeoffs.
4. Describe self-attention as a generalization where a sequence attends to itself, and explain why this formulation is the foundation of the Transformer architecture.

In Chapter 5, we built recurrent neural networks and long short-term memory models that dramatically outperform n-gram models by maintaining a hidden state $\mathbf{h}_t$ that compresses the entire sequence history. LSTMs and GRUs solved the vanishing gradient problem, enabling these models to capture dependencies across dozens of tokens. But a fundamental limitation remains: when such a model is used as an encoder in a conditional generation system, the entire source sequence must be compressed into a single vector $\mathbf{h}_T$. For a ten-word sentence, this compression is manageable. For a fifty-word sentence, critical information is lost. As we argued in Chapter 1, the central question of this book is predicting the next word — and attention dramatically improves the quality of that prediction for conditional generation. We need a mechanism that lets the decoder selectively access any part of the source sequence when generating each output token — a mechanism that *attends* to the relevant input. That mechanism, simply called attention, is the subject of this chapter, and it is the single most important idea bridging RNNs and the Transformer architecture that has come to dominate modern NLP.

6.1 Motivation: The Bottleneck Problem

6.1.1 The Fixed-Size Encoding Problem

Can 256 numbers really capture all the information in 50 words?

Imagine translating a 50-word German sentence into English. The encoder must compress all 50 words into a single vector of 256 dimensions. Every word identity, every syntactic relationship, every semantic nuance, and every idiomatic structure must somehow survive the journey through a fixed-size representation. As we established in Chapter 5 (Section 5.3), the encoder-decoder architecture introduced by Sutskever et al. [2014] and Cho et al. [2014] routes all information through the final hidden state $\mathbf{h}_T$, which then serves as the sole bridge from encoder to decoder. This design choice is elegant and powerful for short sequences, but it creates what we can formalize as an *information bottleneck*: a representation of fixed capacity d must encode a sequence of T tokens, where each token carries its own identity and contextual meaning. The information capacity of $\mathbf{h}_T$ is constant at d dimensions regardless of input length, while the information content of the input grows linearly with T.

Students often confuse a larger hidden dimension with a complete solution to the bottleneck problem. While increasing d_{model} from 256 to 1024 certainly helps for moderate-length sequences, it does not resolve the fundamental asymptotic argument: for any fixed d, there exists a sequence length T beyond which a faithful lossless compression is impossible. The bottleneck is architectural, not dimensional. A useful analogy is the task of summarizing a fifty-page report in a single tweet of 280 characters. No matter how cleverly one writes, critical details will be omitted. The attention mechanism to be developed in this chapter does not compress the report into the tweet — it keeps the full report on the desk and lets the writer glance at the relevant page for each sentence they compose. This transforms the problem from a lossy one-time compression into a sequence of targeted retrievals, each tailored to the immediate generation need.

The formal consequence of the bottleneck is clear when we consider the information flow during decoding. At each decoder step i, the decoder state $\mathbf{s}_i$ is computed as a function of its previous state $\mathbf{s}_{i-1}$, the previous output token, and the fixed encoding $\mathbf{h}_T$. The vector $\mathbf{h}_T$ cannot change between decoder steps — it is a static summary that must simultaneously serve as the memory of word one, word fifteen, word thirty, and every word in between. For a short sentence, the final hidden state of a well-trained LSTM can compress the essential content reasonably well. For long sentences, the model is simply asked to do the impossible.

6.1.2 Performance Degradation on Long Sequences

The limits of the fixed-size bottleneck are not merely theoretical. Cho et al. [2014] provided the first systematic empirical evidence that encoder-decoder performance degrades sharply as input length increases, offering a clear quantitative picture of the architectural limitation. Measuring BLEU scores on English-French translation as a function of source sentence length, they observed that the basic encoder-decoder model performed comparably to phrase-based statistical machine translation on short sentences (fewer than twenty tokens), but fell behind markedly for sentences exceeding twenty to thirty tokens. The degradation was monotonic and persistent: the longer the source sentence, the worse the translation quality, and this pattern held even when the model was trained with ample data. We have seen students' eyes light up when they first understand the bottleneck — it is one of those rare moments where a limitation becomes a motivation so clearly that the solution almost writes itself.

The significance of this empirical result lies in what it rules out. One might hypothesize that the performance drop on long sentences is an artifact of insufficient training data (longer sentences are rarer in corpora) or an artifact of the decoding process rather than the encoding. Cho et al. carefully controlled for these factors and showed that the degradation is systematic and architectural: phrase-based models, which do not compress the source sentence into a fixed vector, do not exhibit the same length-sensitive decline. The problem is specific to the fixed-size encoding. This contrast with phrase-based models is conceptually important: phrase-based systems decompose translation into manageable local operations over source phrases, implicitly preserving access to all source information throughout decoding. The encoder-decoder model, by contrast, discards direct access to the source after the encoder produces $\mathbf{h}_T$. The degradation curve in Cho et al.'s Figure 1 — a graph that every student of NLP should internalize — makes the cost of this architectural choice unmistakably concrete.

A bilingual human translator provides the right intuition for why the performance gap widens with length. A skilled translator does not memorize a sentence and then produce the translation from memory. They read the source text, translate a phrase, glance back at the original for the next

phrase, and continue in this way throughout the document. For short sentences, a translator could plausibly work from memory. For long paragraphs and chapters, working from memory is simply not how skilled translation proceeds. The encoder-decoder model, absent attention, is in precisely the position of a translator who has memorized the source and must work from that memory alone.

6.1.3 The Idea: Let the Decoder Look Back at the Input

The solution is conceptually simple, and it is worth pausing to appreciate just how simple it is before we introduce the mathematics.

Instead of forcing the decoder to rely solely on a single final hidden state $\mathbf{h}_T$, we give the decoder access to all encoder hidden states $\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_{T_x}$ and let the decoder decide which ones are relevant at each generation step. The encoder produces not a single summary vector but a sequence of hidden states, one per input token, each encoding the local context around its position. At each decoder step i, the decoder computes a relevance score between its current state $\mathbf{s}_{i-1}$ and each encoder state $\mathbf{h}_j$, then takes a weighted combination of the encoder states, concentrating weight on the most relevant positions.

This idea transforms the decoding computation from a static memory retrieval into a dynamic, step-specific access pattern. Generating the subject of the English sentence may require attending primarily to the German subject; generating a verb may require attending to the German verb, which may appear in a very different position. The decoder is no longer constrained to a fixed summary — it can direct its attention wherever the source information it needs is located. The student writing an essay from notes provides the right intuition: instead of memorizing all notes beforehand, the student keeps the notes open on the desk and glances at the relevant page for each paragraph they compose. The notes are always available in their full detail; the writer selectively consults them as needed.

We formalize this idea as follows. The encoder produces hidden states $\mathbf{h}_j \in \mathbb{R}^d$ for $j = 1, \ldots, T_x$. At decoder step i, we compute a score e_{ij} reflecting the relevance of encoder position j to decoder step i, normalize these scores into a probability distribution α_{ij} over source positions, and compute a context vector $\mathbf{c}_i = \sum_j \alpha_{ij} \mathbf{h}_j$ as a weighted combination of the encoder states. The context vector $\mathbf{c}_i$ is then used — alongside the decoder's own state — to generate the next token. The remaining question is how to compute the relevance scores e_{ij}. Bahdanau, Cho, and Bengio [2015] answered this question with a parametric scoring function, and it is their solution that we develop in detail in Section 6.2.

6.2 Bahdanau (Additive) Attention

6.2.1 Alignment Scores

The central challenge in implementing the attention idea is defining a useful relevance score between a decoder state $\mathbf{s}_{i-1} \in \mathbb{R}^{d_s}$ and an encoder state $\mathbf{h}_j \in \mathbb{R}^{d_h}$. The score must be differentiable, so that it can be learned by backpropagation through the entire model. It must be capable of expressing complex, non-linear notions of relevance. And it must be computationally tractable, since it will be evaluated once for every decoder-step–encoder-position pair. Bahdanau et al. [2015] proposed a learned, parametric score function that they called the *alignment model*:

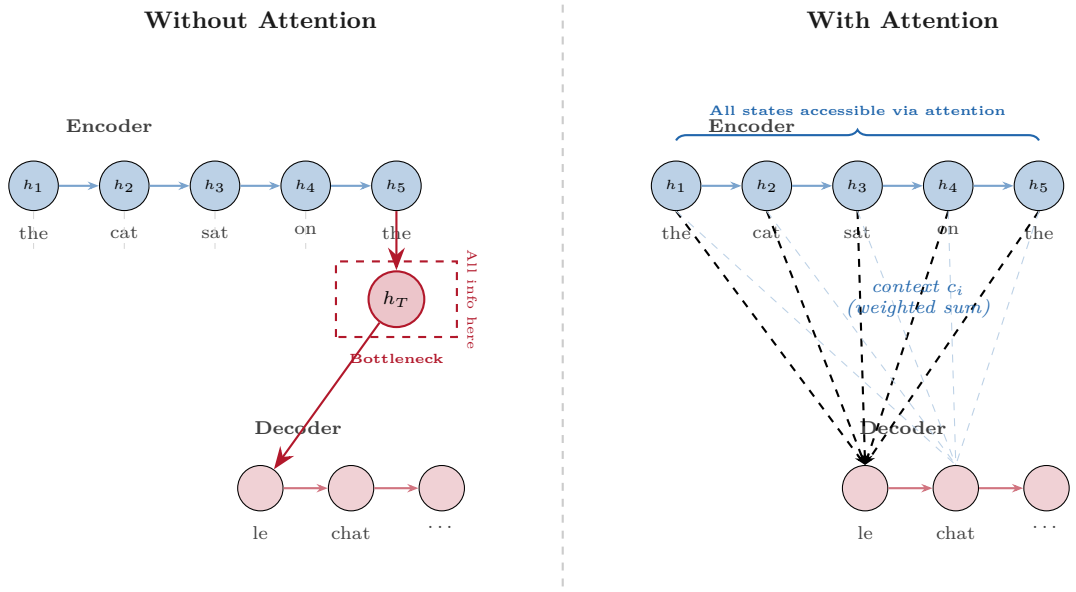


Figure 1: Figure 6.1 – The information bottleneck in the encoder-decoder architecture

$$e_{ij} = \mathbf{v}^\top \tanh(\mathbf{W}_s \mathbf{s}_{i-1} + \mathbf{W}_h \mathbf{h}_j) \tag{6.1}$$

where $\mathbf{W}_s \in \mathbb{R}^{d_a \times d_s}$ and $\mathbf{W}_h \in \mathbb{R}^{d_a \times d_h}$ are weight matrices that project the decoder and encoder states into a common d_a-dimensional space, and $\mathbf{v} \in \mathbb{R}^{d_a}$ is a learned vector that projects the combined representation to a scalar. The function is called *additive* attention because the projected decoder and encoder states are summed before the nonlinearity, distinguishing it from the multiplicative (dot-product) variants we examine in Section 6.3.

The intuition behind Equation (6.1) is as follows. The matrix $\mathbf{W}_s$ transforms the decoder state into a query representation: "what kind of encoder information is this token looking for right now?" The matrix $\mathbf{W}_h$ transforms each encoder state into a key representation: "what kind of information does this position contain?" The tanh activation introduces a nonlinearity that allows the scoring function to capture complex interactions between the query and the key. The projection vector $\mathbf{v}$ then collapses the combined representation to a scalar, which becomes the alignment score. The parameters $\mathbf{W}_s$, $\mathbf{W}_h$, and $\mathbf{v}$ are learned jointly with the rest of the encoder-decoder model through backpropagation. We omit the full backpropagation derivation for the alignment score, which is straightforward but notationally dense — the interested reader will find it in Bahdanau et al.'s appendix.

A common first reaction is to interpret e_{ij} as a hard alignment — one target word, one source word. But the score is a real number that can take any value, and it will be normalized into a soft, continuous weight distribution over all source positions simultaneously. Hard alignment is what phrase-based statistical MT computes; attention computes something fundamentally different. When translating "le chat noir" to "the black cat," the decoder generating the word "black" will assign a high score to the French word "noir," a moderate score to "chat" (since adjective-noun reordering is required), and lower scores to function words. The model learns these scoring patterns

entirely from data, without any explicit alignment supervision.

6.2.2 Softmax Normalization and Attention Weights

With the raw alignment scores $e_{i1}, e_{i2}, \ldots, e_{iT_x}$ computed for decoder step i, the next step converts these unbounded real numbers into a proper probability distribution over source positions. Normalization is achieved via the softmax function:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})} \tag{6.2}$$

The resulting attention weights α_{ij} satisfy $\alpha_{ij} \geq 0$ for all j and $\sum_{j=1}^{T_x} \alpha_{ij} = 1$, making them interpretable as the probability that source position j is the most relevant position for decoder step i. This normalization has several important consequences. First, it ensures that the context vector $\mathbf{c}_i$ (computed in the next subsection) is a weighted average of encoder states, preventing the scale of the raw scores from causing numerical instability. Second, it makes the weights differentiable with respect to the alignment parameters, enabling end-to-end learning. Third, the softmax's exponential amplification means that well-separated scores lead to peaked distributions: if one source position receives a substantially higher alignment score than all others, almost all of the attention weight concentrates at that position. Conversely, when alignment scores are nearly equal, the weights approach the uniform distribution $1/T_x$.

Why softmax specifically, and not some other normalization? The choice is deliberate. it is differentiable everywhere, it produces a valid probability distribution (non-negative and sum-to-one), and it allows the model to produce arbitrarily peaked or spread distributions depending on the magnitude of score differences. Alternative normalizations such as ℓ_1 normalization (dividing by the sum of absolute values) are also non-negative and sum-to-one, but they do not amplify differences between scores exponentially, making it harder for the model to focus sharply on a single position when needed. The softmax's behavior as a "soft argmax" — approaching a one-hot distribution as score differences grow — is precisely the property that makes attention selective.

6.2.3 The Context Vector

What does the decoder actually receive from the encoder?

The context vector $\mathbf{c}_i$ is the weighted combination of all encoder hidden states, assembled using the attention weights computed in Equation (6.2):

$$\mathbf{c}_i = \sum_{j=1}^{T_x} \alpha_{ij} \mathbf{h}_j \tag{6.3}$$

The context vector is the mechanism's output — a step-specific, query-driven summary of the source sequence. Its properties follow directly from the properties of the attention weights. When the attention distribution is peaked (one α_{ij} close to 1, the rest close to 0), the context vector $\mathbf{c}_i$ closely approximates the single encoder state at the position receiving almost all the weight. When the attention distribution is uniform ($\alpha_{ij} = 1/T_x$ for all j), the context vector becomes the arithmetic mean of all encoder states — equivalent to the bag-of-hidden-states representation. In practice,

trained models produce distributions between these extremes, placing significant weight on a small number of highly relevant positions while retaining some weight on surrounding context.

The context vector can be understood as a custom-blended summary: for each token the decoder generates, a new summary of the input is computed that emphasizes precisely the portions of the source sentence most relevant to that generation step. This is categorically different from the fixed summary $\mathbf{h}_T$ used in the non-attention encoder-decoder: the context vector changes at every decoder step, tracking the decoder's changing information needs across the output sequence. A student studying from notes has the right intuition: the student does not read the same summary for every sentence of their essay — they consult different sections of their notes as the essay's argument develops.

We should be precise about one point that generates persistent confusion. The context vector $\mathbf{c}_i$ does not replace the decoder's hidden state $\mathbf{s}_i$ — it supplements it. The decoder's own hidden state continues to track what has already been generated on the target side (the growing output sequence), while the context vector provides dynamic access to source-side information. Both are needed, and Section 6.2.4 explains how they are combined.

6.2.4 Incorporating Context into the Decoder

The context vector $\mathbf{c}_i$ is integrated into the decoder's computation through a straightforward concatenation and projection. The decoder generates the attentional hidden state $\tilde{\mathbf{s}}_i$ as:

$$\tilde{\mathbf{s}}_i = \tanh\left(\mathbf{W}_c[\mathbf{c}_i; \mathbf{s}_i]\right)$$

where $[\mathbf{c}_i; \mathbf{s}_i]$ denotes the concatenation of the context vector and the decoder's hidden state, and $\mathbf{W}_c$ is a learned projection matrix. The resulting $\tilde{\mathbf{s}}_i$ is then passed through the output layer to produce a distribution over the target vocabulary:

$$P(y_i \mid y_1, \ldots, y_{i-1}, \mathbf{x}) = \text{softmax}(\mathbf{W}_o \tilde{\mathbf{s}}_i)$$

This integration mechanism allows the decoder to combine two distinct sources of information: what it knows about the target sequence generated so far (encoded in $\mathbf{s}_i$) and the most relevant portion of the source sentence (encoded in $\mathbf{c}_i$). Without the context vector, the decoder would have access only to its own previous state and the fixed encoding $\mathbf{h}_T$; with the context vector, it has dynamic, step-by-step access to the full encoder hidden state sequence.

A chef preparing a complex dish provides a useful analogy for the integration step. The chef's intuition about flavor balance (the decoder state $\mathbf{s}_i$) reflects everything they have added to the dish so far. The recipe section they are currently consulting (the context vector $\mathbf{c}_i$) provides specific guidance for the next step. The decision about the next ingredient draws on both: pure intuition without the recipe can lead to errors, and following the recipe without any accumulated intuition ignores the current state of the dish. The attention mechanism provides the contextual information at precisely the moment it is needed, and the decoder state provides the accumulated knowledge about what has already been produced.

Sidebar: The Birth of Attention

In 2014, Dzmitry Bahdanau was a graduate student at Jacobs University Bremen, working in close collaboration with Yoshua Bengio's group at the Université de Montréal. Neural

machine translation was barely a year old — Sutskever et al. [2014] had just demonstrated that an LSTM encoder-decoder could rival phrase-based statistical systems, but the model struggled markedly with long sentences. Bahdanau's insight was deceptively simple: instead of forcing the decoder to work from a single summary vector, let it look back at the entire source sentence at each generation step, weighting each source position by its relevance to the current target word. The resulting paper, "Neural Machine Translation by Jointly Learning to Align and Translate," was submitted to ICLR 2015 and became one of the most cited papers in the history of NLP, accumulating tens of thousands of citations in under a decade. What makes the story remarkable is not just the quality of the technical contribution but the simplicity of the core idea. The attention mechanism did not merely improve translation quality — it introduced a fundamentally new computational primitive: soft, differentiable, content-based memory access. This primitive proved to be so general that it forms the computational core of the Transformer architecture described in Chapter 8, three years after the original paper and a technology generation removed from the sequence-to-sequence models it was designed to augment. The insight that models should selectively attend to relevant inputs rather than compress everything into a fixed representation is one of the most consequential ideas in modern deep learning.

6.2.5 A Complete Worked Example

The clearest way to internalize the attention computation is through concrete numbers. In our experience, walking through a numerical example with pen and paper is the single most effective way to internalize the attention computation — the abstract steps become tangible operations that can be verified at every stage.

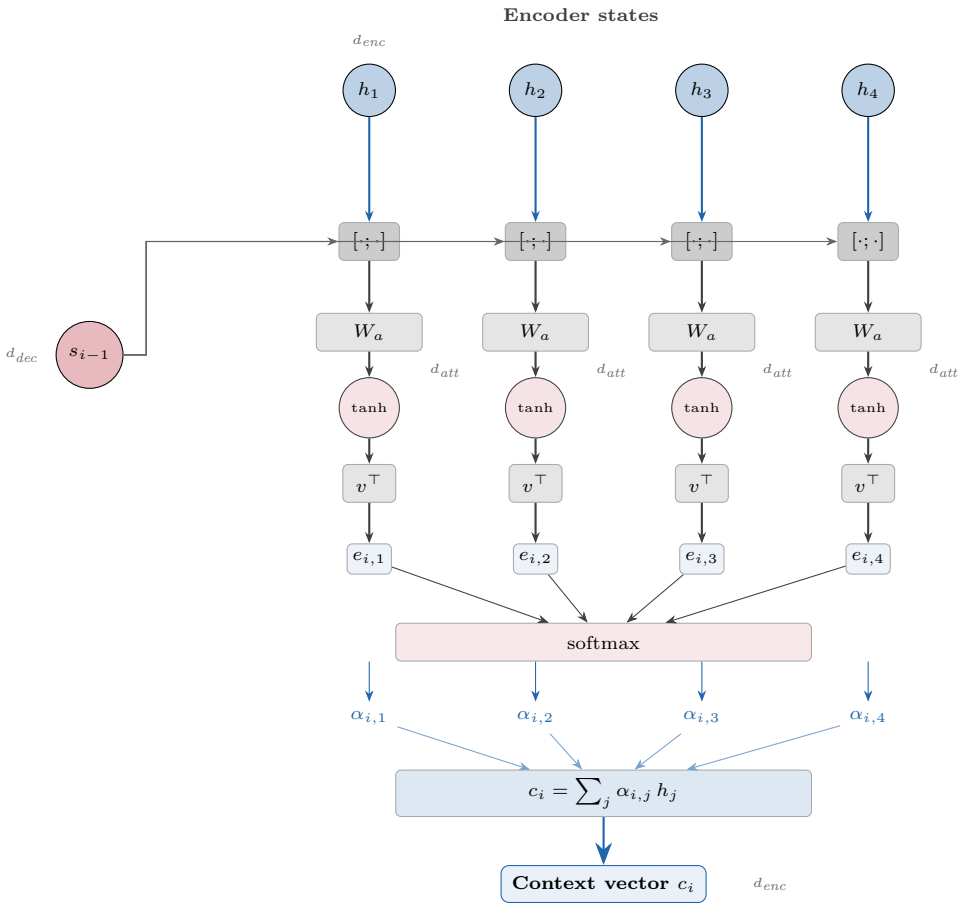
We construct a small example: a source sequence of four tokens, with encoder hidden states of dimension $d_h = 3$. We use a decoder state $\mathbf{s}_0 \in \mathbb{R}^3$ and an attention dimension $d_a = 4$. The encoder hidden states are:

$$\mathbf{h}_1 = \begin{bmatrix} 0.2 \\ 0.8 \\ -0.3 \end{bmatrix}, \quad \mathbf{h}_2 = \begin{bmatrix} -0.5 \\ 0.3 \\ 0.7 \end{bmatrix}, \quad \mathbf{h}_3 = \begin{bmatrix} 0.9 \\ -0.1 \\ 0.4 \end{bmatrix}, \quad \mathbf{h}_4 = \begin{bmatrix} 0.1 \\ 0.6 \\ -0.8 \end{bmatrix}$$

The initial decoder state is $\mathbf{s}_0 = [0.5, -0.2, 0.4]^\top$. We initialize the weight matrices with small values for illustration: $\mathbf{W}_s \in \mathbb{R}^{4\times 3}$ and $\mathbf{W}_h \in \mathbb{R}^{4\times 3}$, and the projection vector $\mathbf{v} \in \mathbb{R}^4$. For the purposes of this example, we set $\mathbf{W}_s\mathbf{s}_0 = [0.3, -0.1, 0.5, 0.2]^\top$ (a fixed projection of the decoder state) and compute $\mathbf{W}_h\mathbf{h}_j$ for each j.

Suppose the projections of the encoder states yield: $\mathbf{W}_h\mathbf{h}_1 = [0.4, 0.2, -0.3, 0.6]^\top$, $\mathbf{W}_h\mathbf{h}_2 = [-0.2, 0.5, 0.3, -0.1]^\top$, $\mathbf{W}_h\mathbf{h}_3 = [0.7, -0.3, 0.4, 0.2]^\top$, and $\mathbf{W}_h\mathbf{h}_4 = [0.1, 0.4, -0.2, 0.5]^\top$. We add the decoder projection to each: $\mathbf{W}_s\mathbf{s}_0 + \mathbf{W}_h\mathbf{h}_j$ for each j, apply tanh elementwise, and dot with $\mathbf{v} = [0.3, 0.5, -0.2, 0.4]^\top$ to obtain the raw alignment scores. For $j = 1$: the combined vector before tanh is $[0.7, 0.1, 0.2, 0.8]^\top$; after tanh: $[0.604, 0.100, 0.197, 0.664]^\top$; dot with $\mathbf{v}$: $e_{11} = 0.3(0.604) + 0.5(0.100) + (-0.2)(0.197) + 0.4(0.664) = 0.181 + 0.050 - 0.039 + 0.266 = 0.458$. Carrying out the same computation for $j = 2, 3, 4$ yields alignment scores $e_{12} = 0.121$, $e_{13} = 0.512$, $e_{14} = 0.389$ (note that real computations involve more decimal places; we round for readability).

Applying softmax to the four alignment scores: $\exp(0.458) = 1.581$, $\exp(0.121) = 1.129$, $\exp(0.512) =$



Bahdanau et al. (2015): additive alignment
$e_{ij} = v^\top \tanh(W_a[s_{i-1}; h_j])$

Figure 2: Figure 6.2 – Bahdanau attention architecture

1.668, $\exp(0.389) = 1.476$; sum $= 5.854$. The attention weights are therefore $\alpha_{11} = 1.581/5.854 = 0.270$, $\alpha_{12} = 0.193$, $\alpha_{13} = 0.285$, $\alpha_{14} = 0.252$. We verify: $0.270 + 0.193 + 0.285 + 0.252 = 1.000$. The context vector is $\mathbf{c}_1 = 0.270 \cdot \mathbf{h}_1 + 0.193 \cdot \mathbf{h}_2 + 0.285 \cdot \mathbf{h}_3 + 0.252 \cdot \mathbf{h}_4$, which works out to $\mathbf{c}_1 \approx [0.280, 0.345, -0.023]^\top$. The attention in this example is fairly diffuse (all four weights in the range 0.19–0.29), reflecting the random initialization. In a trained model, the weights would be far more concentrated on the most relevant source positions.

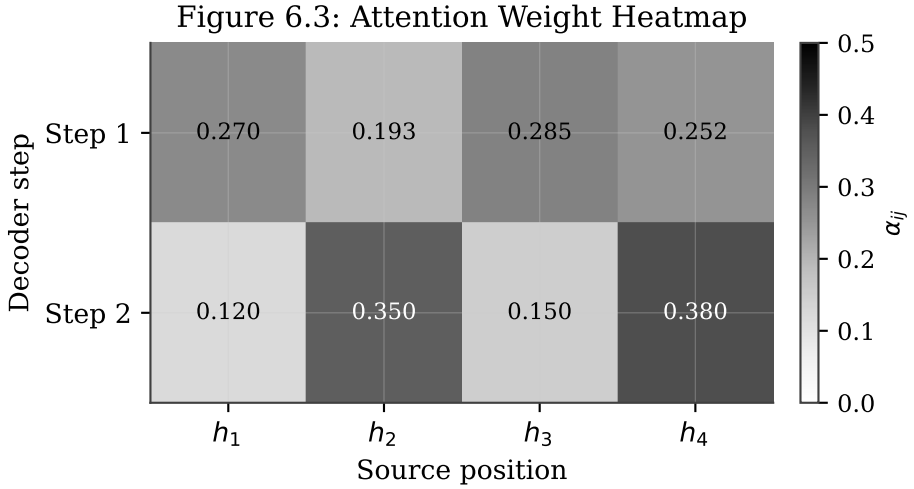


Figure 3: Figure 6.3 – Attention weight heatmap for a four-encoder-state, two-decoder-step example

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

# Bahdanau attention: alignment scores, weights, and context vector
d_h, d_s, d_a = 8, 8, 8  # encoder dim, decoder dim, attention dim
T_x = 4  # source sequence length

# Learnable parameters
W_h = nn.Linear(d_h, d_a, bias=False)  # projects encoder states
W_s = nn.Linear(d_s, d_a, bias=False)  # projects decoder state
v   = nn.Linear(d_a, 1,   bias=False)  # projects to scalar score

# Random encoder hidden states and decoder state
H = torch.randn(T_x, d_h)   # (T_x, d_h)
s = torch.randn(1, d_s)      # (1, d_s)

# Compute alignment scores: e_j = v^T tanh(W_s s + W_h h_j)
proj_h = W_h(H)                              # (T_x, d_a)
proj_s = W_s(s).expand(T_x, -1)             # (T_x, d_a) broadcast
scores = v(torch.tanh(proj_h + proj_s)) # (T_x, 1)

# Normalize to attention weights
alpha = F.softmax(scores, dim=0)         # (T_x, 1)
```

```
# Compute context vector as weighted sum of encoder states
context = (alpha * H).sum(dim=0)          # (d_h,)

print("Attention weights:", alpha.squeeze().detach().numpy())
print("Sum of weights:   ", alpha.sum().item())          # must be 1.0
print("Context vector:   ", context.detach().numpy())
```

6.3 Luong (Multiplicative) Attention

6.3.1 Dot-Product Attention

What is the simplest possible attention score that still works?

Luong, Pham, and Manning [2015] asked whether the parametric score function in Equation (6.1) could be replaced by something simpler and faster. Their first proposal, the dot-product score, is as minimal as one can conceive:

$$\text{score}(\mathbf{s}_t, \mathbf{h}_j) = \mathbf{s}_t^\top \mathbf{h}_j \tag{6.4}$$

The dot product between the decoder state $\mathbf{s}_t \in \mathbb{R}^d$ and the encoder state $\mathbf{h}_j \in \mathbb{R}^d$ — note that both must have the same dimensionality d — measures the geometric alignment between the two vectors in the hidden space. Vectors that point in similar directions (high cosine similarity) receive high scores; orthogonal vectors receive scores near zero; anti-aligned vectors receive negative scores. This scoring function contains no learned parameters of its own: the "relevance" notion it captures is entirely a product of the learned representations of $\mathbf{s}_t$ and $\mathbf{h}_j$.

The absence of learned parameters might seem like a weakness — how can a parameter-free score function learn to capture the complex cross-lingual alignments required for machine translation? The key insight is that the linear projections that produce $\mathbf{s}_t$ and $\mathbf{h}_j$ already encode the relevant information. The encoder is trained to produce hidden states whose dot products with decoder states reflect source-target relevance; no additional parameters are needed because the representational learning happens in the encoder and decoder weights. In practice, dot-product attention performs comparably to additive attention on many tasks, while being substantially faster — all T_x scores can be computed in a single matrix-vector multiplication $\mathbf{e}_t = \mathbf{s}_t^\top \mathbf{H}$, where $\mathbf{H} = [\mathbf{h}_1, \ldots, \mathbf{h}_{T_x}]^\top \in \mathbb{R}^{T_x \times d}$, exploiting the highly optimized matrix multiplication routines available on modern hardware.

Dot-product attention is like checking which library books (encoder states) share a shelf with your current research interest (decoder state). Books with similar call numbers — similar vector directions in the learned embedding space — score highest. No trained librarian (additional parameters) is needed: the organizational system itself (the learned encoder and decoder representations) makes the similarity structure meaningful.

6.3.2 General (Bilinear) Attention

The dot-product score makes a strong implicit assumption: that the decoder state $\mathbf{s}_t$ and the encoder state $\mathbf{h}_j$ live in the same representational space, so that raw geometric alignment is meaningful. When the encoder and decoder use different hidden dimensions, or when the training procedure

results in representations that are not well-aligned in the raw hidden space, this assumption may not hold. Luong et al. addressed this with the *general* score function, also called bilinear attention:

$$\text{score}(\mathbf{s}_t, \mathbf{h}_j) = \mathbf{s}_t^\top \mathbf{W}_a \mathbf{h}_j \tag{6.5}$$

The matrix $\mathbf{W}_a \in \mathbb{R}^{d_s \times d_h}$ introduces a learnable linear transformation that maps the encoder state into the decoder's representation space before the dot product is computed. This allows the model to learn a task-specific notion of "relevance" that is not limited to raw geometric similarity. The bilinear formulation generalizes dot-product attention as a special case: setting $\mathbf{W}_a = \mathbf{I}$ (the identity matrix, requiring $d_s = d_h$) recovers Equation (6.4). It also bears a structural relationship to Bahdanau's additive attention: while the two formulations differ in how the decoder and encoder states are combined (multiplicatively versus additively), both introduce a learned parameter that mediates the relevance computation.

The tradeoff between dot-product and bilinear attention is primarily one of flexibility versus simplicity. The bilinear weight matrix $\mathbf{W}_a$ adds $d_s \times d_h$ parameters to the model — on the order of d^2 for typical hidden dimensions — and can potentially learn more discriminative alignments. In practice, dot-product attention is generally preferred for its simplicity and better scalability: the absence of $\mathbf{W}_a$ reduces parameter count and enables the score computation to be expressed as a single matrix multiplication without an intermediate projection. As we will see in Chapter 8, the Transformer's scaled dot-product attention is a direct descendant of Luong's dot-product variant, modified by a scaling factor $1/\sqrt{d_k}$ to prevent softmax saturation at large dimensions.

If dot-product attention asks "are these vectors similar?" then general attention asks "are these vectors similar in a learned, task-specific way?" — the matrix $\mathbf{W}_a$ defines what "similar" means for the particular source-target language pair and training objective. Whether the additional parameters are beneficial depends on the dataset size: bilinear attention can overfit on small corpora but may provide advantages when ample training data is available.

6.3.3 Global vs. Local Attention

With the score functions defined, attention can be applied in two distinct modes with respect to the extent of the source sequence that the decoder attends to. The global-vs-local distinction is, frankly, less important in practice than one might expect from the attention it received in the original paper, but it introduces ideas that anticipate later efficient attention mechanisms.

Global attention is the default: at each decoder step, alignment scores are computed over the entire source sequence $\mathbf{h}_1, \ldots, \mathbf{h}_{T_x}$. This is what we have assumed throughout Sections 6.2 and 6.3.1–6.3.2. Global attention has $\mathcal{O}(T_x)$ cost per decoder step and accesses all source information at every generation step. For typical sentence-length inputs (twenty to fifty tokens), this is entirely practical. *Local attention*, proposed by Luong et al. as an alternative, first predicts an alignment position p_t — the center of the source window the decoder should attend to at step t — and then restricts attention to a window of $2D + 1$ positions around p_t. The position p_t is predicted by a small feedforward network, and attention weights within the window are further modulated by a Gaussian distribution centered at p_t to encourage smooth, monotonic alignment patterns. Local attention reduces the computation from $\mathcal{O}(T_x)$ to $\mathcal{O}(D)$ per decoder step, where D is the window half-width (typically $D \approx 10$). For tasks with roughly monotonic alignment — such as same-family language translation — local attention can match global attention in quality while being faster on long sequences. This

sliding-window idea anticipates the efficient attention variants discussed in later chapters, which extend the concept to transformers operating on very long documents.

6.3.4 Computational Comparison with Bahdanau

The gap between Bahdanau and Luong attention is not in asymptotic complexity — both are $\mathcal{O}(T_x \cdot d)$ per decoder step — but in the constant factors that determine wall-clock speed.

For Bahdanau's additive attention, each alignment score requires passing the combined decoder-encoder projection through a tanh nonlinearity and then through the projection vector $\mathbf{v}$. This cannot be expressed as a single matrix multiplication: the sequential structure (projection, addition, tanh, dot product) requires separate operations for each source position, preventing full exploitation of GPU matrix-multiplication hardware. For Luong's dot-product attention, all T_x alignment scores are computed as $\mathbf{e}_t = \mathbf{H}\mathbf{s}_t \in \mathbb{R}^{T_x}$, a single matrix-vector multiplication that maps directly onto the highly optimized Basic Linear Algebra Subprograms (BLAS) routines available on every modern accelerator. The difference in throughput can be substantial in practice: matrix multiplications on GPUs exploit thousands of parallel multiply-accumulate units simultaneously, while the sequential intermediate operations in additive attention create bottlenecks that prevent full hardware use. A useful analogy: both methods cover the same distance, but dot-product attention uses a highway (optimized matrix operations) while additive attention takes local roads (sequential computations through a small network). Both arrive at the same destination, but the highway is substantially faster.

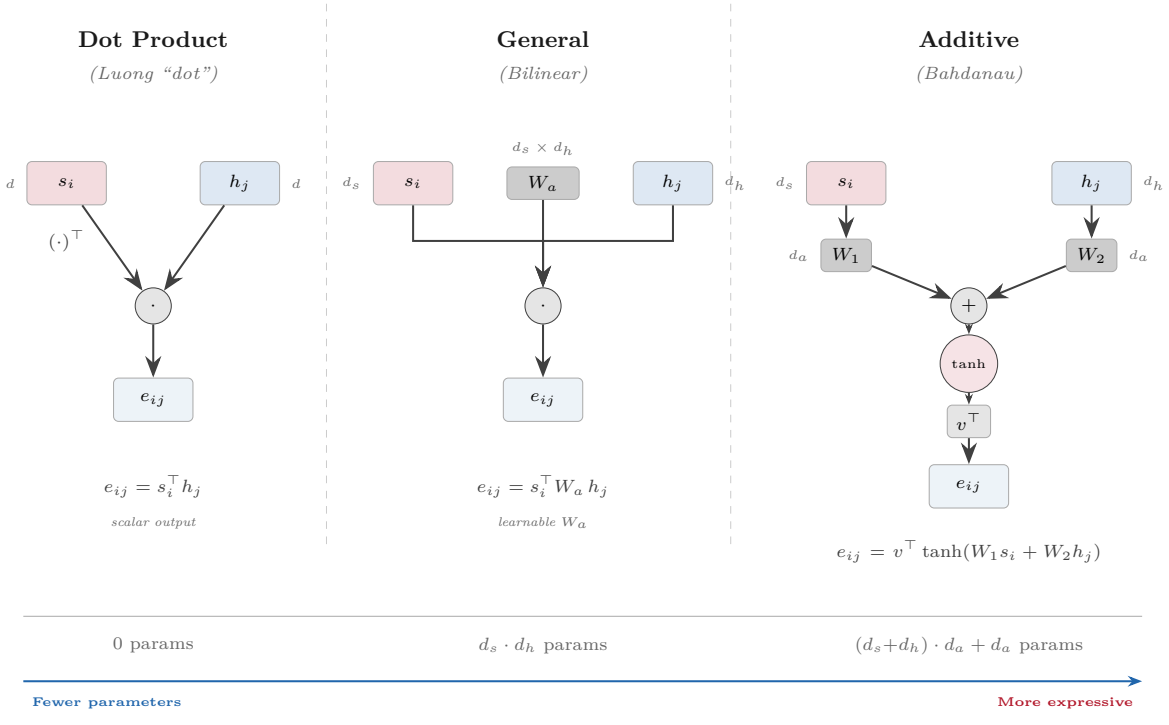


Figure 4: Figure 6.5 – Computational comparison of Luong attention variants

6.4 Self-Attention

6.4.1 From Cross-Attention to Self-Attention

In machine translation, attention connects two distinct sequences: the decoder (target language) attends to the encoder (source language). This formulation is called *cross-attention*, reflecting the fact that queries come from one sequence and keys and values come from another. Recognizing the formal structure of cross-attention opens the door to a powerful generalization. In cross-attention, the decoder state $\mathbf{s}_{i-1}$ plays the role of a query, and the encoder states $\mathbf{h}_1, \ldots, \mathbf{h}_{T_x}$ play the roles of keys and values. There is nothing in the mathematical formulation that requires the query and the key-value sequences to be distinct. What happens if we let a single sequence serve as the source of its own queries, keys, and values?

Self-attention is exactly this: each position in a sequence attends to all other positions in the same sequence. For a sequence of tokens $\mathbf{x}_1, \ldots, \mathbf{x}_T$, each position i computes alignment scores against every position j (including itself), normalizes these scores via softmax, and produces a weighted combination of all sequence positions as its output representation. The self-attention representation of position i thus captures information from the entire sequence, weighted by the pairwise relevance between position i and each other position. This is fundamentally different from what an RNN produces: an RNN's hidden state at position i encodes only the prefix $\mathbf{x}_1, \ldots, \mathbf{x}_i$ (in a forward-pass encoder), and information about distant positions must travel through many intermediate hidden states. Self-attention at position i has direct, unmediated access to every position in the sequence simultaneously.

Consider the sentence "The animal did not cross the street because it was too tired." The word "it" refers to "animal," but these two words are separated by seven positions. In an RNN encoder, a representation of "it" acquires information about "animal" only through a chain of seven hidden state transitions, each of which can distort and attenuate the relevant signal. In self-attention, "it" computes an alignment score directly against "animal" in a single operation, with no intervening information loss. This single example captures the central advantage of self-attention for long-range dependency modeling.

6.4.2 The Query-Key-Value Intuition

How should we think about the roles different positions play in self-attention?

The mechanics of self-attention become cleaner with a conceptual framework introduced informally here and formalized fully in Chapter 8. Each position in a self-attention computation simultaneously plays three distinct roles, represented by three vectors: a *query* $\mathbf{q}_i$ ("what is this position looking for?"), a *key* $\mathbf{k}_i$ ("what does this position contain?"), and a *value* $\mathbf{v}_i$ ("what does this position contribute if selected?"). In the simplest formulation, all three are derived from the same input vector: $\mathbf{q}_i = \mathbf{k}_i = \mathbf{v}_i = \mathbf{x}_i$. The dot-product self-attention score between positions i and j is then $e_{ij} = \mathbf{q}_i^\top \mathbf{k}_j = \mathbf{x}_i^\top \mathbf{x}_j$, and the self-attention output at position i is:

$$\mathbf{z}_i = \sum_{j=1}^{T} \alpha_{ij} \mathbf{v}_j = \sum_{j=1}^{T} \alpha_{ij} \mathbf{x}_j$$

where $\alpha_{ij} = \text{softmax}(\mathbf{x}_i^\top \mathbf{x}_j)$ normalized over all j. The full Transformer formulation in Chapter 8

projects the input into separate query, key, and value spaces via learned linear projections $\mathbf{W}_Q$, $\mathbf{W}_K$, $\mathbf{W}_V$ before computing attention, but the intuition is identical.

The library metaphor illuminates the query-key-value abstraction precisely. A library patron formulates a search query ("the reader is looking for books about neural networks"). Each book in the library has a catalog entry, its key, that describes its contents. The attention score between the query and each key measures how well the book matches the search. The patron then receives information from the books with the highest-matching entries — the values. In self-attention applied to a sentence, each word formulates a query based on its own identity and context, checks it against the "catalog entries" (keys) of every other word, and receives a weighted mixture of the other words' content (values). The mechanism is trained to learn what kinds of queries and keys make linguistically meaningful matches — for instance, a noun's query matching strongly with the keys of its associated adjectives and verbs.

```
import torch
import torch.nn.functional as F

# Self-attention on a toy 5-position sequence, dimension 6
T, d = 5, 6
torch.manual_seed(42)

X = torch.randn(T, d)  # 5 tokens, each a 6-dim vector

# Compute pairwise dot-product scores: (T, T)
# No learned projections; no scaling (see Chapter 8 for both)
scores = X @ X.T          # shape (T, T)

# Normalize each row to get attention weights
alpha = F.softmax(scores, dim=-1)   # shape (T, T)

# Compute self-attention output: weighted sum of values (= X itself here)
Z = alpha @ X               # shape (T, d)

print("Attention weight matrix (each row sums to 1.0):")
print(alpha.detach().numpy().round(3))
print("\nRow sums:", alpha.sum(dim=-1).detach().numpy())
print("Input norms: ", X.norm(dim=-1).detach().numpy().round(3))
print("Output norms:", Z.norm(dim=-1).detach().numpy().round(3))
```

6.4.3 Why Self-Attention Changes Everything

We confess that the first time we encountered self-attention, the idea of a sequence attending to itself felt circular. The resolution is subtle: the projections create three distinct roles for each position — querying, providing keys, and providing values — and these roles are learned independently, even though they derive from the same input.

The transformative property of self-attention is the path length between any two positions. In an RNN, information at position j must propagate through $|i - j|$ sequential hidden state transitions to

influence position i. For the first and last tokens of a hundred-word sentence, this is a path of length 99. Each transition is a nonlinear computation that can distort the signal, and the backpropagation gradient must traverse the same path in reverse, suffering the exponential attenuation we studied as the vanishing gradient problem in Chapter 5. Self-attention reduces this path length to $\mathcal{O}(1)$: every position directly attends to every other position in a single computation, regardless of their distance in the sequence. There is no intermediate chain through which information must pass, and no corresponding chain of gradients that must survive backpropagation.

This $\mathcal{O}(1)$ path length comes at a cost: to compute attention between every pair of positions, self-attention requires $\mathcal{O}(T^2)$ operations per layer, versus $\mathcal{O}(T)$ sequential operations for an RNN. For sequences of typical length ($T < 2048$ tokens), the quadratic cost is affordable on modern hardware and is vastly compensated by the parallelization benefits discussed in Section 6.4.4. For very long sequences (documents of thousands of tokens), the quadratic cost becomes a genuine practical challenge — one that motivates the efficient attention variants developed after the original Transformer paper, which are covered in Chapter 14. For the moment, the key insight is that self-attention's $\mathcal{O}(T^2)$ cost buys something qualitatively different from an RNN's $\mathcal{O}(T)$ cost: it buys the ability to model any pairwise relationship in a single operation, with gradients that can flow directly from any output position to any input position.

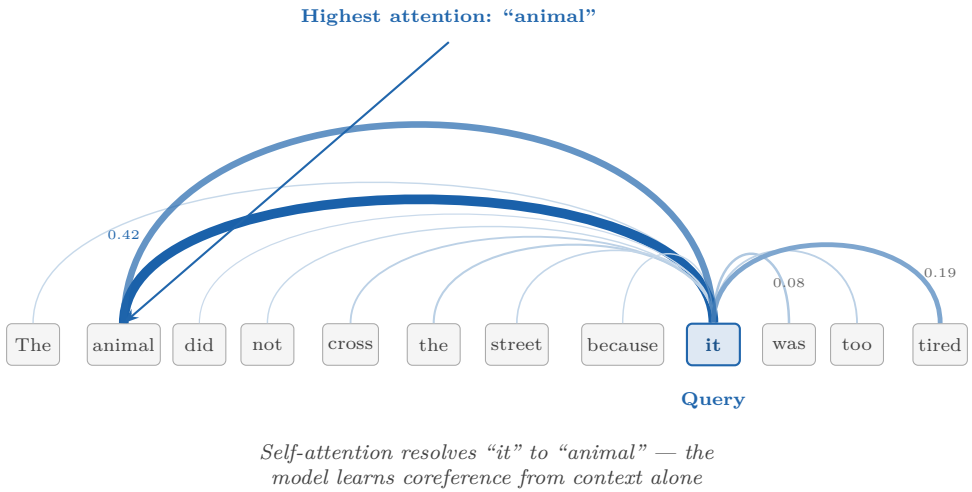


Figure 5: Figure 6.6 – Self-attention in a single sentence

6.4.4 Parallelization Advantages

The practical case for self-attention over recurrence is ultimately about speed, and speed comes from parallelization.

Recall from Chapter 5 that an RNN's hidden state update rule is $\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t)$: the state at time t depends on the state at time $t-1$, which depends on $t-2$, and so on back to the start of the sequence. This sequential dependency is not an implementation accident — it is a logical necessity. An RNN's power comes from the fact that $\mathbf{h}_t$ summarizes everything from position 1 to t. The price of this power is that one cannot compute $\mathbf{h}_t$ until $\mathbf{h}_{t-1}$ is available. For a sequence of $T = 1000$ tokens, the encoder requires 1000 sequential matrix-vector multiplications in series: no amount of

GPU parallelism can compress these 1000 causal dependencies into fewer than 1000 serial steps.

Self-attention eliminates this serial dependency entirely. The attention score matrix $\mathbf{E} = \mathbf{X}\mathbf{X}^\top \in \mathbb{R}^{T \times T}$ (where $\mathbf{X}$ is the matrix of all input representations stacked by row) is computed in a single batched matrix multiplication. No position needs to wait for any other position to be processed first. All pairwise interactions are computed simultaneously, and the attention output $\mathbf{Z} = \text{softmax}(\mathbf{E})\mathbf{X}$ follows in one additional matrix multiplication. The entire computation for all T positions is thus two matrix multiplications — operations that GPUs and TPUs are architecturally designed to execute with maximal parallelism. A road analogy captures the contrast: an RNN processes tokens like cars on a single-lane road, each car waiting for the one ahead. Self-attention processes all positions simultaneously like a roundabout where every car can enter at once. This parallelization advantage is not merely a constant speedup: it grows with sequence length and with hardware capacity. As models and sequences grow larger, the gap between sequential and parallel computation widens, which explains why self-attention so thoroughly displaced recurrence once hardware accelerators became the dominant training substrate.

Sidebar: "Attention Is All You Need" — Almost

The title of Vaswani et al. [2017] is one of the most provocative in the history of machine learning. The authors at Google Brain and Google Research asked a bold question: if attention can capture all the relationships that recurrence captures, why keep the RNN at all? Their answer was the Transformer, which removed recurrence entirely and relied exclusively on self-attention — supplemented by positional encodings, feed-forward layers, residual connections, and layer normalization — to build contextual representations. The paper achieved 28.4 BLEU on WMT 2014 English-German, surpassing all previous state-of-the-art results, and trained in 3.5 days on 8 P100 GPUs, a fraction of the time required for comparable recurrent models. The "almost" in the sidebar title is instructive. Attention alone is not quite all you need. The Transformer also requires positional encodings (self-attention is permutation-equivariant and has no inherent notion of word order), feed-forward layers (which provide per-position nonlinear transformation that pure attention cannot achieve), residual connections (which enable training of deep stacks without gradient vanishing), and layer normalization (which stabilizes optimization). These components are not secondary details — without any one of them, the Transformer does not train effectively. The paper's core claim is more precise: the *recurrence* that had defined sequence modeling for 25 years was not necessary. The attention mechanism from this chapter, scaled and extended with multiple heads, was sufficient to replace it. Chapter 8 develops the full architecture, the scaling argument for the $1/\sqrt{d_k}$ factor, and the multi-head extension that gives different attention heads the freedom to capture different linguistic relationships simultaneously.

6.5 Attention as a General Mechanism

6.5.1 The General Attention Formulation

Can we write a single equation that subsumes every attention variant we have seen?

Bahdanau attention, Luong dot-product attention, Luong bilinear attention, and self-attention all follow the same computational pattern: compute scores reflecting the relevance between a query and a set of keys, normalize the scores into a probability distribution, and produce a weighted

combination of a corresponding set of values. The general formulation that unifies all these variants is:

$$\text{Attention}(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\text{score}(\mathbf{q}, \mathbf{K}))\mathbf{V} \tag{6.6}$$

where $\mathbf{q} \in \mathbb{R}^{d_q}$ is the query vector, $\mathbf{K} \in \mathbb{R}^{T \times d_k}$ is the matrix of key vectors (one per source position), and $\mathbf{V} \in \mathbb{R}^{T \times d_v}$ is the matrix of value vectors. The score function $\text{score}(\mathbf{q}, \mathbf{K}) \in \mathbb{R}^T$ produces one alignment score per source position, and $\text{softmax}(\cdot)$ normalizes these into the attention weight vector $\boldsymbol{\alpha} \in \mathbb{R}^T$. The output is a d_v-dimensional weighted combination of the value vectors.

The three design choices that distinguish different attention mechanisms are: (1) the score function — additive (Bahdanau), dot-product (Luong), or bilinear (Luong general); (2) the source of the query — the decoder state in cross-attention, or a position in the sequence itself in self-attention; and (3) the source of the keys and values — the encoder states in cross-attention, or the same sequence in self-attention. Every attention mechanism we have discussed is an instance of Equation (6.6) with particular choices for these three elements. This unifying view clarifies that attention is not a single algorithm but a computational template — a differentiable, query-driven, weighted-retrieval mechanism that can be instantiated in many ways. Xu et al. [2015] demonstrated the generality of this template by applying attention to visual inputs in image captioning, showing that the mechanism can attend over spatial regions of an image as naturally as over positions in a text sequence.

6.5.2 Attention as Soft Dictionary Lookup

The query-key-value framework becomes especially clear when viewed through the lens of data structures. A standard hard dictionary (or hash map) maps each query q to exactly one value v by finding the unique key k such that $k = q$:

$$\text{HardLookup}(q) = v_j \quad \text{where } j = \arg\max_k \text{sim}(q, k_j)$$

This lookup is not differentiable: the arg max operation produces a discrete index, and gradients cannot flow through a discrete selection. Attention replaces hard lookup with a *soft* version in which the query retrieves a weighted combination of all values, with weights proportional to query-key similarity:

$$\text{SoftLookup}(\mathbf{q}) = \sum_j \alpha_j \mathbf{v}_j \quad \text{where } \alpha_j = \text{softmax}(\text{score}(\mathbf{q}, \mathbf{k}_j))$$

The soft lookup is differentiable everywhere: the softmax and linear combination are both smooth functions of the query and key vectors, so gradients flow freely through the attention computation to the query-generating and key-generating components of the model. As the attention distribution becomes more peaked — as the scores become more discriminative — the soft lookup approaches the hard lookup: nearly all weight concentrates on the single most relevant value. The model can thus learn, through training, to be as precise as a hard lookup when precision is beneficial, or to spread weight across multiple values when a blend is more informative.

The soft dictionary view reveals why attention functions as a *differentiable memory access mechanism*. The keys and values in the attention computation constitute the "memory" (the encoder hidden

states, or the sequence of positions in self-attention), and the query is the "address" that determines what to retrieve from memory. Unlike a conventional computer's memory, which is addressed by explicit integer indices, attention memory is addressed by content: the query does not specify a position, it specifies what kind of content it is looking for, and the matching is carried out by comparing query representations to key representations in a learned embedding space. This content-based addressing is the core computational primitive that makes attention so powerful, and it is the primitive that the Transformer scales into a general-purpose sequence processing architecture.

6.5.3 Visualizing Attention Weights

What can a heatmap of attention weights tell us about what a model has learned?

The attention weight matrix $\boldsymbol{\alpha} \in \mathbb{R}^{T_y \times T_x}$ — with one row per target position and one column per source position — provides a two-dimensional array of learned soft alignments that can be visualized directly as a heatmap. Bahdanau et al. [2015] first showed such a heatmap for French-to-English translation, and the result was immediately striking: the learned alignments were linguistically meaningful. English and French words tended to align in roughly diagonal patterns (reflecting similar word order), with clear deviations at positions where word order differed (adjective-noun inversion between the two languages). The heatmap confirmed that the model had, without any explicit alignment supervision, discovered the cross-lingual correspondences that human translators use.

Figure 6.4: French-English Attention Heatmap

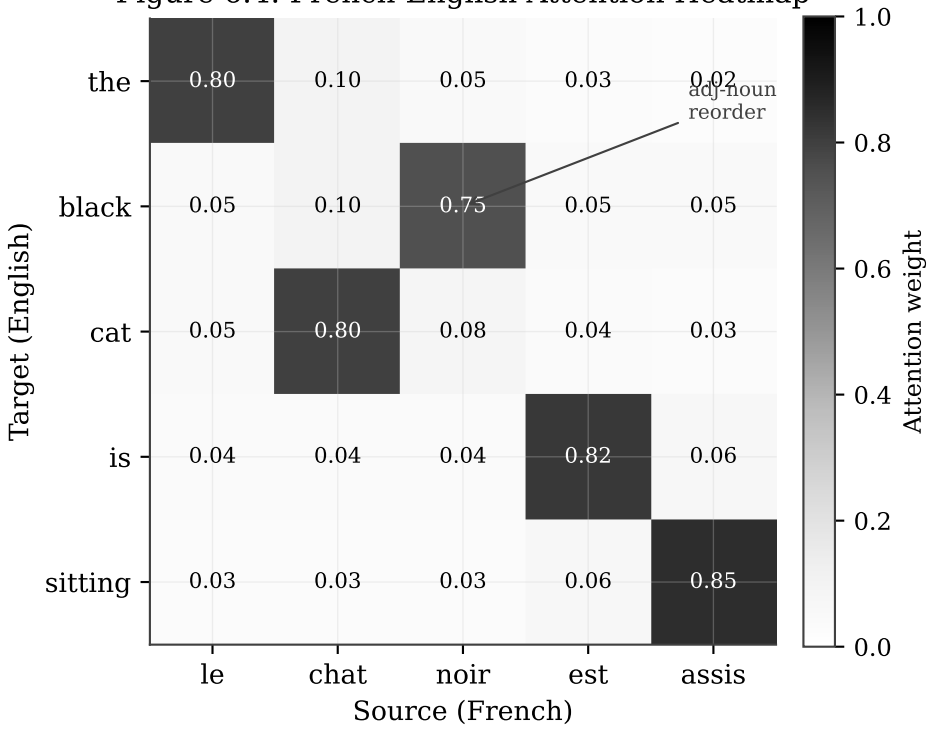


Figure 6: Figure 6.4 – Attention weight heatmap for French-to-English translation

```
import matplotlib.pyplot as plt
```

```
import numpy as np
import torch
import torch.nn.functional as F

# Toy French-to-English attention heatmap
source = ["le", "chat", "noir", "est", "assis"]
target = ["the", "black", "cat",  "is",  "sitting"]

# Hand-designed attention weights showing reordering:
# "black" (target[1]) attends strongly to "noir" (source[2])
# "cat"   (target[2]) attends strongly to "chat" (source[1])
attn = torch.tensor([
    [0.80, 0.10, 0.05, 0.03, 0.02],  # "the"     -> "le"
    [0.05, 0.10, 0.75, 0.05, 0.05],  # "black"   -> "noir"
    [0.05, 0.80, 0.08, 0.04, 0.03],  # "cat"     -> "chat"
    [0.04, 0.04, 0.04, 0.82, 0.06],  # "is"      -> "est"
    [0.03, 0.03, 0.03, 0.06, 0.85],  # "sitting" -> "assis"
])

print("Attention weights shape:", attn.shape)
print("Row sums (should all be 1.0):", attn.sum(dim=1))

fig, ax = plt.subplots(figsize=(6, 5))
im = ax.imshow(attn.numpy(), cmap="Blues", aspect="auto", vmin=0, vmax=1)
ax.set_xticks(range(len(source))); ax.set_xticklabels(source)
ax.set_yticks(range(len(target))); ax.set_yticklabels(target)
plt.colorbar(im, ax=ax, label="Attention weight")
ax.set_xlabel("Source (French)"); ax.set_ylabel("Target (English)")
ax.set_title("Attention heatmap: adjective-noun reordering")
plt.tight_layout(); plt.savefig("attention_heatmap.pdf"); plt.show()
print("Plot saved as attention_heatmap.pdf")
```

Attention weight visualization must be interpreted carefully. The causal reader might suppose that high attention weights directly indicate feature importance — that the positions with the most attention weight are the positions the model "relies on" for its prediction. This interpretation is not reliable. Attention weights show where the model directs its retrieval query, but the retrieved values may or may not be decisive for the output distribution. A position can attract high attention weight but contribute a value vector with little discriminative content for the output token; conversely, a position with relatively modest attention weight may contribute a highly informative value. Attention weights are better understood as eye-tracking data: they show where the model is "looking," not necessarily what it is "thinking." This caveat does not diminish the interpretive value of attention heatmaps — for translation in particular, the alignment patterns are striking and informative — but it cautions against treating them as explanations in the technical sense.

6.5.4 What Attention Patterns Reveal

Empirical analysis of trained attention heads reveals a taxonomy of distinct attention patterns that correspond, in many cases, to recognizable linguistic functions. Clark et al. [2019] conducted

a systematic analysis of attention patterns in pre-trained BERT models, identifying heads that specialize in markedly different types of position-to-position relationships. Understanding these patterns provides insight into what self-attention learns to represent across layers of a deep model.

The four most commonly observed patterns are: *diagonal attention*, in which each position attends primarily to its immediate neighbors (capturing local syntactic context, similar to what a small convolution would capture); *block attention*, in which positions within a phrase or clause attend predominantly to other positions in the same constituent (capturing phrasal grouping); *long-range attention*, in which specific positions attend across large distances to their syntactic or semantic associates (subject-verb agreement heads, coreference resolution heads); and *uniform attention*, in which attention weights are broadly distributed across all positions (functioning as a global context summarizer rather than a targeted retrieval). Different attention heads in the same model layer often specialize in different patterns, and the specialization varies systematically across layers: early layers tend toward local and positional patterns, while deeper layers produce longer-range, more semantically driven patterns. This specialization is an emergent property of training — no explicit supervision directs different heads toward different patterns — and its existence suggests that self-attention naturally discovers the multi-level structure of linguistic representation.

One might expect all attention heads to learn the same pattern — same inputs, same objective, so why would they differ? The opposite is true: diversity of patterns across heads is a feature of good attention, not an accident of random initialization. A model in which every head attends to the same positions is wasting representational capacity — it is representing the same information h times instead of representing h different views of the sequence. The multi-head attention mechanism formalized in Chapter 8 is specifically designed to encourage this diversity by learning independent projection matrices for each head.

6.5.5 From Attention to Transformers (Preview)

If there is one section in this chapter that we would ask every reader to pause and reflect on, it is this one — the conceptual leap from attention to Transformers is the most important transition in this entire book.

Self-attention as developed in this chapter is the computational core of the Transformer. Every pairwise interaction between positions in a sequence is captured by the attention score matrix; every position's output representation is a weighted combination of all other positions' value vectors; and the entire computation is expressed as two matrix multiplications, enabling full parallelism across all positions. This is an extraordinarily expressive computational primitive. But the Transformer, as described by Vaswani et al. [2017], adds three critical engineering elements that transform self-attention from a mechanism into an architecture capable of training deep networks on large datasets.

The first element is *scaling*. In Chapter 8, we will show that for large key dimension d_k, the dot products $\mathbf{q}^\top \mathbf{k}_j$ grow in magnitude proportional to $\sqrt{d_k}$, pushing the softmax into regions of extremely small gradients and destabilizing training. The solution is to divide by $\sqrt{d_k}$ before applying softmax — a modification so simple it deserves the term "scaling fix" rather than "new mechanism." The resulting scaled dot-product attention is the version used throughout the Transformer. The second element is *multi-head attention*: rather than computing a single attention function over the full d-dimensional representations, the Transformer projects queries, keys, and values into h subspaces of dimension d/h each, applies attention in each subspace independently, and concatenates the results. This allows different heads to specialize in different aspects of the input (local vs. global, syntactic

Figure 6.7: Taxonomy of Attention Patterns

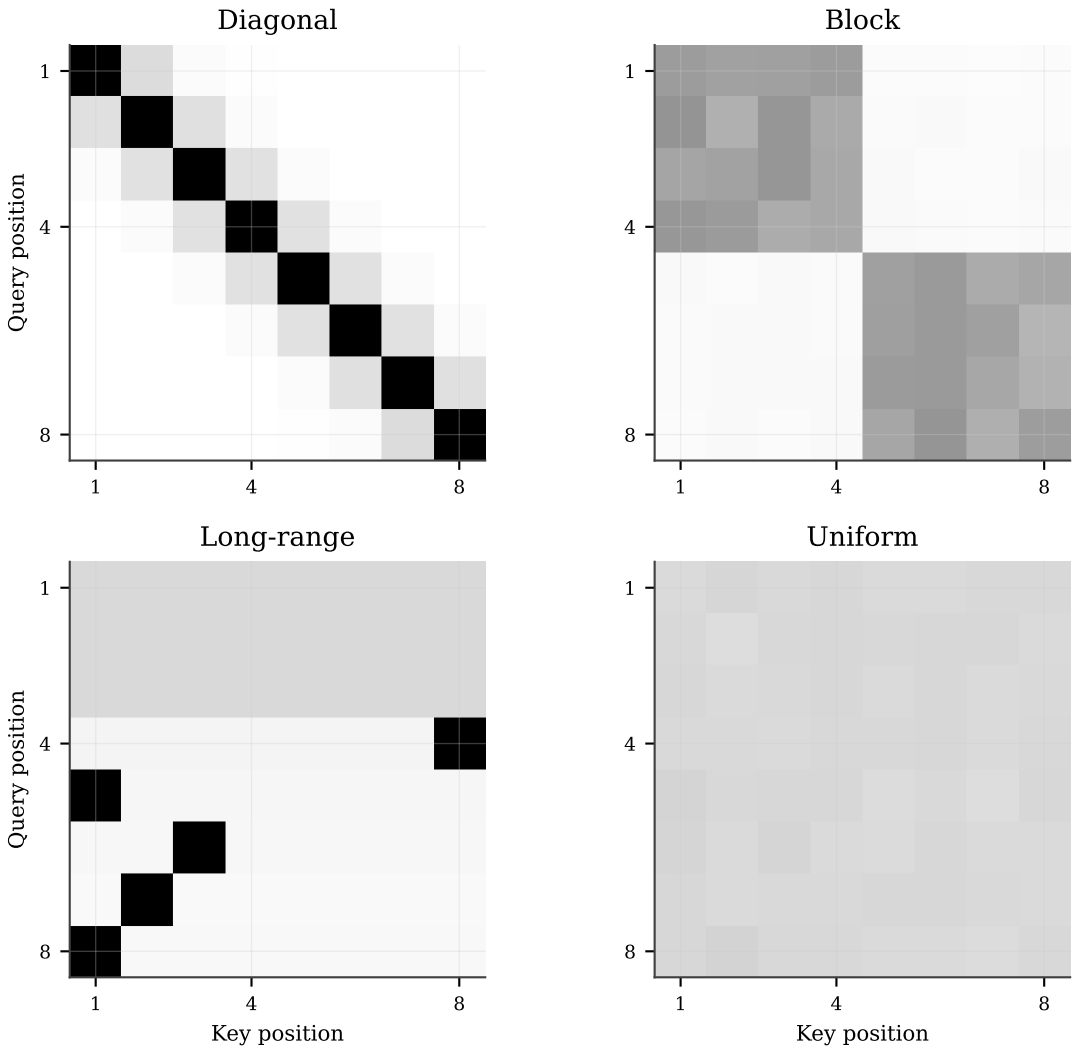


Figure 7: Figure 6.7 – Taxonomy of attention patterns

vs. semantic), capturing diverse relationship types simultaneously. The third element is *positional encoding*: self-attention is permutation-equivariant — the output representations do not change if the input sequence is shuffled — so the model has no inherent notion of word order. Sinusoidal or learned positional encodings inject position information into the input representations before the first attention layer, restoring the model's ability to represent sequence structure.

These three elements — scaling, multi-head projection, and positional encoding — together with residual connections and layer normalization, constitute the engineering scaffold that makes the Transformer trainable, deep, and highly performant. They are the subjects of Chapter 8. The present chapter has established the conceptual and mathematical foundation: attention as soft dictionary lookup, cross-attention as the mechanism for encoder-decoder communication, self-attention as a mechanism for intra-sequence dependency modeling, and the general query-key-value formulation that unifies all variants. The Transformer does not introduce a new idea on top of these foundations — it operationalizes them at scale.

We have shown how attention allows the decoder to selectively access encoder states, and how self-attention allows every position in a sequence to directly interact with every other position. In the next chapter, we assemble the complete encoder-decoder system for conditional generation — sequence-to-sequence models with teacher forcing, beam search, and the evaluation metrics that measure generation quality.

Chapter Summary

This chapter developed the attention mechanism from first principles, beginning with the information bottleneck that motivates it and ending with the general query-key-value formulation that underpins the Transformer. Section 6.1 established the bottleneck problem: a fixed-size hidden state $\mathbf{h}_T$ cannot faithfully represent sequences of arbitrary length, and Cho et al. [2014] showed empirically that encoder-decoder BLEU scores degrade sharply beyond twenty tokens. The solution is to give the decoder access to all encoder hidden states simultaneously, selecting relevant ones dynamically at each decoding step. Section 6.2 derived Bahdanau attention in full: the parametric alignment score $e_{ij} = \mathbf{v}^\top \tanh(\mathbf{W}_s \mathbf{s}_{i-1} + \mathbf{W}_h \mathbf{h}_j)$, the softmax normalization to attention weights α_{ij}, and the context vector $\mathbf{c}_i = \sum_j \alpha_{ij} \mathbf{h}_j$ that supplements the decoder's hidden state. Section 6.3 presented Luong's simpler multiplicative variants — dot-product attention $\mathbf{s}_t^\top \mathbf{h}_j$ and bilinear attention $\mathbf{s}_t^\top \mathbf{W}_a \mathbf{h}_j$ — and explained why their amenability to batched matrix multiplication makes them computationally preferable in practice. Section 6.4 introduced the conceptual leap to self-attention: a sequence attending to itself with $\mathcal{O}(1)$ path length between any two positions and $\mathcal{O}(T^2)$ computation per layer, enabling full parallelism that recurrent networks cannot achieve. Section 6.5 unified all variants under the general formulation $\text{Attention}(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\text{score}(\mathbf{q}, \mathbf{K}))\mathbf{V}$, interpreted attention as differentiable soft dictionary lookup, and previewed the three additions (scaling, multi-head projection, positional encoding) that the Transformer builds on top of self-attention in Chapter 8.

Attention transforms the capabilities of neural sequence models, providing a natural transition to Chapter 7, where we combine encoder-decoder architectures with attention to build practical sequence-to-sequence systems for translation and generation.

Exercises

Theory Exercises

Exercise 6.1 (Uniform attention reduces to averaging). Show that when attention weights are uniform, $\alpha_{ij} = 1/T_x$ for all j, the context vector $\mathbf{c}_i = \sum_{j=1}^{T_x} \alpha_{ij}\mathbf{h}_j$ reduces to the arithmetic mean of the encoder hidden states $\frac{1}{T_x}\sum_{j=1}^{T_x} \mathbf{h}_j$. Under what conditions does the softmax function produce exactly uniform output? Explain why uniform attention, while mathematically well-defined, is suboptimal for most generation tasks and what it implies about the alignment scores e_{ij}.

Hint: Substitute $\alpha_{ij} = 1/T_x$ into the context vector formula and simplify. For the softmax question, consider what value of all e_{ij} produces equal exponentials.

Exercise 6.2 (Dot-product as special case of bilinear). Prove that Luong's dot-product score $\text{score}(\mathbf{s}_t, \mathbf{h}_j) = \mathbf{s}_t^\top \mathbf{h}_j$ is a special case of the general bilinear score $\text{score}(\mathbf{s}_t, \mathbf{h}_j) = \mathbf{s}_t^\top \mathbf{W}_a \mathbf{h}_j$ by specifying the value of $\mathbf{W}_a$. What dimensionality constraint on $\mathbf{s}_t$ and $\mathbf{h}_j$ is required for the dot-product score to be well-defined? Does the bilinear score impose the same constraint? Explain the geometric interpretation of $\mathbf{W}_a = \mathbf{I}$ in terms of the relationship between the encoder's and decoder's representation spaces.

Hint: Recall the definition of the identity matrix and the matrix-vector product $\mathbf{Iv} = \mathbf{v}$.

Exercise 6.3 (Computational complexity of attention). Analyze the time and space complexity of computing attention for a single decoder step as a function of source sequence length T_x and hidden state dimension d. Compare additive (Bahdanau) attention and multiplicative (Luong dot-product) attention. For Bahdanau attention, count the number of scalar multiplications required by the tanh and linear operations for each of the T_x source positions. For Luong dot-product attention, express the score computation as a single matrix-vector product and give its complexity. Explain why both variants are $\mathcal{O}(T_x \cdot d)$ in time but differ substantially in practical throughput on GPU hardware.

Hint: The matrix-vector product $\mathbf{Hs}_t$, where $\mathbf{H} \in \mathbb{R}^{T_x \times d}$, requires $T_x \cdot d$ multiply-accumulate operations. Contrast this with the per-position loop required by additive attention.

Exercise 6.4 (Quadratic memory for long sequences). In self-attention over a sequence of length T, the attention score matrix $\mathbf{E} \in \mathbb{R}^{T \times T}$ stores one score per pair of positions. Show that for $T = 1000$ and $d_{\text{model}} = 512$, storing the attention matrix in 32-bit floating point requires exactly 4 MB of memory. Extend the calculation to $T = 4096$ and $T = 32768$ (the context lengths of representative large language models). Discuss why quadratic memory scaling motivates the development of efficient attention variants for long-context processing, and explain what a model with $T = 32768$ context length would require just for the attention matrix in a single layer.

Hint: A single float32 value occupies 4 bytes. The attention matrix has $T \times T$ entries. Scale accordingly and convert to MB (10^6 bytes) or GB (10^9 bytes).

Programming Exercises

Exercise 6.5 (Bahdanau attention from scratch). Implement Bahdanau attention in PyTorch from scratch. Given 6 encoder hidden states of dimension 16 and a single decoder state of dimension 16, define the learnable parameters $\mathbf{W}_s$, $\mathbf{W}_h$, and $\mathbf{v}$ as `nn.Linear` layers. Compute the alignment scores using Equation (6.1), normalize them via softmax to obtain the attention weights, and compute the context vector as the weighted sum of encoder states. Print the attention weights and verify that

they sum to 1.0. As an extension, apply your implementation to encoder sequences of lengths 6, 12, 24, and 48, verifying that the attention weights always sum to 1 regardless of sequence length.

Exercise 6.6 (Luong dot-product attention and timing). Implement Luong dot-product attention in PyTorch. Compare the per-forward-pass computation time of your Bahdanau implementation (from Exercise 6.5) and the Luong dot-product implementation for source sequence lengths $T_x \in \{10, 50, 100, 500, 1000\}$, holding the hidden dimension at $d = 128$. Plot computation time as a function of sequence length for both implementations. Use `time.time()` on CPU (or `torch.cuda.synchronize()` if using a GPU) to measure elapsed time, averaged over 100 forward passes. Discuss whether the empirical timing ratio matches the theoretical argument about constant factors in GPU matrix multiplication.

Exercise 6.7 (Attention visualization with a pre-trained model). Using a pre-trained neural machine translation model available through the HuggingFace Transformers library (for example, `Helsinki-NLP/opus-mt-en-fr`), translate three English sentences of increasing length (5 words, 15 words, 30 words) into French. Extract the cross-attention weights using `output_attentions=True` during generation. For each sentence, average the attention weights across all layers and all heads and plot the resulting matrix as a heatmap with source tokens on the horizontal axis and target tokens on the vertical axis. Identify the dominant attention pattern (diagonal, reordering, many-to-one) for each sentence and discuss how the patterns change with sentence length.

Exercise 6.8 (Self-attention representation analysis). Implement self-attention without scaling or learned projections (as in Code Example 2 in Section 6.4.2) and apply it to a sequence of 8 word embeddings drawn randomly from a standard normal distribution with dimension $d = 16$. Visualize the 8×8 attention weight matrix as a heatmap. Then compute the pairwise cosine similarity matrices for the input representations $\mathbf{X}$ and the self-attention output representations $\mathbf{Z}$, and compare them visually. Discuss what the comparison reveals about how self-attention modifies the representation geometry — specifically, whether representations of positions with high mutual attention weight become more similar after the operation.

Exercise 6.9 (Encoder-decoder with and without attention). Implement a minimal encoder-decoder translation model in PyTorch, in two versions: one without attention (decoder uses only $\mathbf{h}_T$ from a 2-layer LSTM encoder) and one with Bahdanau attention (decoder has access to all encoder hidden states). Train both models on a toy parallel corpus of 500 short English-German sentence pairs (sentences of 5–15 tokens), using cross-entropy loss and the Adam optimizer. After training for 20 epochs, compare the BLEU scores of both models on a held-out test set, and plot the BLEU score as a function of source sentence length for sentences of length 5–7, 8–10, and 11–15 tokens. Verify empirically that the attention model's advantage over the no-attention baseline increases with sentence length, consistent with the theoretical motivation in Section 6.1.

Exercise 6.10 (Attention temperature). Modify the softmax in the attention computation to use a temperature parameter τ: replace $\text{softmax}(\mathbf{e})$ with $\text{softmax}(\mathbf{e}/\tau)$. Apply this to the self-attention computation from Exercise 6.8 using temperatures $\tau \in \{0.1, 0.5, 1.0, 2.0, 5.0\}$. For each temperature, compute and visualize the 8×8 attention weight matrix. Additionally, compute the entropy $H(\boldsymbol{\alpha}_i) = -\sum_j \alpha_{ij} \log \alpha_{ij}$ of each attention distribution and plot entropy as a function of τ, averaged across all query positions. Explain the limiting behavior: what happens to the attention distribution as $\tau \to 0$ (approaching hard attention) and as $\tau \to \infty$ (approaching uniform attention)? Discuss the implications of temperature scaling for controlling attention sharpness during inference.

References

- Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In *Proceedings of ICLR 2015*.
- Cho, K., van Merrienboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. In *Proceedings of SSST-8*, 103–111.
- Cho, K., van Merrienboer, B., Gulcehre, C., et al. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of EMNLP*, 1724–1734.
- Clark, K., Khandelwal, U., Levy, O., & Manning, C. D. (2019). What does BERT look at? An analysis of BERT's attention. In *Proceedings of the ACL Workshop on BlackboxNLP*, 276–286.
- Luong, T., Pham, H., & Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. In *Proceedings of EMNLP*, 1412–1421.
- Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, 3104–3112.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, 5998–6008.
- Xu, K., Ba, J., Kiros, R., et al. (2015). Show, attend and tell: Neural image caption generation with visual attention. In *Proceedings of ICML*, 2048–2057.