Attention Mechanisms and Tiny Transformers: From Scaled Dot-Product Attention to a Mini LLM

TAE Program — Core Track

November 15, 2025

Abstract

This handout develops the core ideas behind modern transformer-based language models at a scale suitable for from-scratch implementation. Starting from sequence modeling and content-based attention, we derive scaled dot-product attention, specialize it to self-attention, and assemble multi-head attention, position-wise feedforward networks, residual connections, and positional encodings into a standard transformer block. We then build a tiny decoder-only transformer language model, specify its training objective and sampling procedures, and map each concept to a set of NumPy/PyTorch implementations in an accompanying Colab notebook. The note is nearly self-contained: small numeric examples, shape cheat sheets, and implementation sketches are provided so that you can reproduce and extend the model without relying on high-level transformer libraries.

THE AI
ENGINEER

Contents

1 Motivation and Goal

Neural networks from Weeks 1–2 gave us the tools to optimize parameters with gradient-based methods and to understand how gradients are computed through computation graphs. Transformers apply exactly the same principles to *sequences* of tokens, with attention providing a flexible way for each position to look at—and learn from—other positions.

In this note we work toward a concrete capstone: a tiny decoder-only transformer language model ("mini LLM") that predicts the next token in a short text sequence. The emphasis is on transparent, from-scratch implementations that you can reproduce and extend in a Colab notebook.

By the end you should be able to:

1. state the sequence modeling problem we solve (autoregressive language modeling) and describe an appropriate toy dataset,
2. derive and interpret the scaled dot-product attention formula, including shapes and masking,
3. implement single-head and multi-head self-attention layers in NumPy/PyTorch,
4. assemble these building blocks into a small decoder-only transformer with positional encodings and a vocabulary projection head,
5. train the model with standard optimizers from Week 1 and generate short text samples.

The Colab accompanying this handout mirrors this structure: each conceptual block here corresponds to a small, testable code cell there (forward pass, numeric examples, training loop, and sampling utilities).

Learning Principle

Attention is a differentiable *content-based addressing* mechanism: each query asks "which other tokens are relevant?" and aggregates them with a learned, data-dependent weighting. A transformer stacks attention with simple position-wise MLPs, residual connections, and normalization into a deep computation graph; backpropagation from Week 2 flows through this graph in exactly the same way as for earlier feedforward networks.

2 How to Use This Capstone

This handout is dense on purpose: it aims to expose the core mechanics behind modern GPT-style models, not just their APIs. At the same time, your time during the program may be limited. This section offers a pragmatic reading and implementation roadmap for different time budgets.

2.1 Why This Matters Now

Large language models (LLMs) are built from the same ingredients developed here: attention, transformer blocks, positional encodings, and gradient-based training on sequence data. Understanding these ideas at the "tiny" scale pays off in several ways:

- it demystifies how GPT-type models process prompts and generate completions,
- it makes AI engineering decisions (prompting vs. fine-tuning, context-window trade-offs, model sizing) more grounded,

- it prepares you to read and reason about model logs, shape mismatches, and optimization issues when working with real LLM stacks.

Even if you cannot study every derivation in full detail, having a concrete mental model of how a small transformer works will make you more effective with today's powerful models.

2.2 If You Have 90–150 Minutes

With roughly 1.5–2.5 hours, the goal is to internalize the essentials and get a minimal end-to-end implementation running.

Theory priorities.

- Read *Motivation and Goal* (current section) and skim *From Sequence Data to Attention* for overall context.
- Study *Scaled Dot-Product Attention (Single Head)* carefully, including the shapes and the tiny numeric example.
- Skim *Self-Attention for Sequences* to see how Q, K, V are derived from embeddings and why masking is needed.

Implementation priorities.

- In the Colab, implement and test `scaled_dot_product_attention` using the numeric example.
- Implement a single-head self-attention layer and plug it into a minimal transformer block with:
  - one attention layer,
  - one small feedforward network,
  - residuals + layer norms.
- Use existing PyTorch utilities (e.g., `nn.Embedding`, `nn.LayerNorm`) for everything else; keep the model small (e.g., one block, low d_{model}) and train just enough to see non-trivial samples.

2.3 If You Have 150–270 Minutes

With roughly 2.5–4.5 hours, you can build a more complete understanding and a closer analogue of practical GPT-like models.

Theory additions.

- Read *Self-Attention for Sequences* in full, including the complexity discussion.
- Work through *Multi-Head Attention and Transformer Blocks*, focusing on:
  - how multi-head shapes are organized,
  - why residual connections and layer normalization matter.
- Read *Positional Encodings* (at least the sinusoidal scheme) to understand how order enters the model.

Implementation additions.

- Extend your code to use:
  - a proper `MultiHeadAttention` module,
  - a configurable stack of `TransformerBlocks` (e.g., $L = 2$ or $L = 4$),
  - the `PositionalEncoding` module.
- Implement the full `TinyTransformerLM` and a basic training loop on a small corpus.
- Add simple sampling routines (greedy + temperature) and inspect generated sequences as training progresses.

2.4 If You Can Work Systematically

If your schedule allows a more relaxed pace, the best path is to treat the note as both a conceptual and implementation guide and to work through it linearly:

- read each section in order and, where possible, check the algebra (especially for attention shapes and masks),
- implement the corresponding notebook pieces as you go rather than saving all coding for the end,
- use the appendices (shape cheat sheet and numeric examples) to debug mismatches or unexpected behavior.

In all cases, aim to finish with a working tiny transformer LM in Colab and at least a coarse understanding of how attention, transformer blocks, and positional encodings interact. The exact level of mathematical detail you reach can vary; the key is to connect the formulas to concrete tensors and code.

3 From Sequence Data to Attention

This section places transformers in the broader landscape of sequence modeling. We clarify what kind of data we work with, how we represent tokens as vectors, and which baseline approaches motivate the move toward attention-based architectures.

3.1 Sequence Modeling Landscape

Many tasks in machine learning involve *ordered* data:

- language modeling: given a prefix of tokens $x_{1:t-1}$, predict the next token x_t,
- sequence-to-sequence mapping: map an input sequence (e.g., a sentence in one language) to an output sequence (e.g., its translation),
- code completion, time series forecasting, or event streams, all of which rely on the order of past observations.

In this capstone we focus on autoregressive language modeling. Given a sequence of tokens $(x_1, \ldots, x_T)$ drawn from a vocabulary of size V, the model assigns probabilities

$$p(x_1, \ldots, x_T) = \prod_{t=1}^{T} p\big(x_t \mid x_{<t}\big),$$

and training reduces to maximizing the log-likelihood (or, equivalently, minimizing cross-entropy) on a dataset of sequences.

To make this concrete in Colab, we will use a *small* corpus: for example, a few thousand characters of text (quotes, short stories, or code snippets). The details of the corpus are not critical, as long as:

- the vocabulary is manageable (e.g., characters or a few hundred tokens),
- we can split the corpus into training and validation chunks,
- we fix a context length T (often called *block size*) that fits easily in memory.

Tokens are first mapped to vectors in $\mathbb{R}^{d_{\text{model}}}$ via an *embedding matrix*. Let d_{model} be the model dimension and let

$$E \in \mathbb{R}^{V \times d_{\text{model}}}$$

be a learnable embedding matrix whose i-th row $E_{i:}$ is the embedding for vocabulary index i. For a sequence of token indices $(x_1, \dots, x_T)$ we obtain an embedding sequence

$$X = \begin{bmatrix} e_1 \\ \vdots \\ e_T \end{bmatrix} \in \mathbb{R}^{T \times d_{\text{model}}}, \qquad e_t = E_{x_t:}.$$

In code, this corresponds to an `nn.Embedding` layer or a simple lookup into a weight matrix.

Classical baselines for language modeling include n-gram models (using counts over short windows of tokens) and bag-of-words plus MLP classifiers. These methods either:

- ignore long-range dependencies (because they only look at fixed-size n-grams), or
- discard order entirely (bag-of-words), making them unsuitable for sequence prediction.

Transformers overcome these limitations by using attention to let each position directly inspect the entire context window, while still being trainable with the same gradient-based machinery from earlier weeks.

3.2 Why Not Just RNNs? (Context Only)

Before transformers, recurrent neural networks (RNNs) and their gated variants (LSTMs, GRUs) were the default tools for sequence modeling. An RNN maintains a hidden state h_t that is updated step by step,

$$h_t = f(h_{t-1}, x_t; \theta), \qquad \hat{y}_t = g(h_t),$$

so that all information about the past must be compressed into h_t.

RNN-style models raise several practical issues:

- **Sequential processing.** The recurrence couples time steps: to compute h_t you need h_{t-1}, which depends on h_{t-2}, and so on. This limits parallelism on modern accelerators.
- **Vanishing and exploding gradients.** Backpropagating through many time steps multiplies Jacobians repeatedly, often leading to gradients that either decay to zero or blow up, making learning long-range dependencies difficult.
- **Fixed-size memory.** The hidden state has fixed dimension, so it must summarize all past tokens into a single vector. Subtle dependencies may be lost or hard to recover.

In this capstone we will *not* implement RNNs. Instead, we keep them as a conceptual contrast: attention-based models avoid explicit recurrence, allow highly parallel computation, and use a richer mechanism for accessing past information.

3.3 Attention as Content-Based Addressing

Attention replaces a single compressed hidden state with a more flexible pattern: a set of *memory slots* and a mechanism to read from them based on content similarity.

Imagine you have a notebook filled with short sentences (your memory). When you ask a question like "Who is the main character?", you instinctively scan all sentences and mentally weight them by how relevant they seem. You then form an answer by focusing on the most relevant lines and effectively ignoring the rest. This is content-based addressing: retrieve information by *what it is about*, not by a fixed index.

In neural attention, we formalize this as:

- a *query* vector $q \in \mathbb{R}^d$ representing "what we are looking for",
- a set of *key* vectors $\{k_j\}_{j=1}^T$ representing "what each memory slot is about",
- a set of *value* vectors $\{v_j\}_{j=1}^T$ representing the information stored at each slot.

We compute a similarity score between the query and each key (e.g., via a dot product), turn these scores into weights with a softmax, and form a weighted average of the values:

$$\alpha_j = \frac{\exp(s(q, k_j))}{\sum_{\ell=1}^T \exp(s(q, k_\ell))}, \qquad \mathrm{Attn}(q; K, V) = \sum_{j=1}^T \alpha_j v_j,$$

where $s(q, k_j)$ is a similarity function. The output is a vector of the same dimension as the values, emphasizing information from positions that are most relevant to the query.

In self-attention (Section 5), the queries, keys, and values all come from the same sequence of token embeddings X. Before that, we make the dot-product variant precise, including shapes and masking, in the next section.

4 Scaled Dot-Product Attention (Single Head)

This section formalizes the notion of attention introduced above. We define the query–key–value matrices, spell out the scaled dot-product attention formula with masking, and work through a tiny numerical example that the Colab notebook will reproduce exactly.

4.1 Definitions and Shapes

We now move from a single query to a matrix formulation that is convenient for vectorized implementation.

Consider a single sequence of length T with token embeddings collected in a matrix

$$X \in \mathbb{R}^{T \times d_{\text{model}}}.$$

We choose dimensions d_k (for queries and keys) and d_v (for values), and learn three projection matrices

$$W_Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, \qquad W_K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \qquad W_V \in \mathbb{R}^{d_{\text{model}} \times d_v}.$$

From X we build:

$$Q = XW_Q \in \mathbb{R}^{T \times d_k}, \quad K = XW_K \in \mathbb{R}^{T \times d_k}, \quad V = XW_V \in \mathbb{R}^{T \times d_v}.$$

Row t of Q is the query for position t, row j of K is the key for position j, and row j of V is the value at position j.

In batched form, with B sequences at once, we treat X as a three-dimensional tensor

$$X \in \mathbb{R}^{B \times T \times d_{\text{model}}},$$

and maintain the same last-two-dimension shapes:

$$Q, K \in \mathbb{R}^{B\times T\times d_k}, \qquad V \in \mathbb{R}^{B\times T\times d_v}.$$

The Colab implementation will use this batched convention. A small "shape cheat sheet" in the appendix will summarize all tensor dimensions used throughout the model.

4.2 Attention Formula and Masking

For a single sequence (no batch dimension), collect all query–key dot products in a score matrix

$$S = \frac{1}{\sqrt{d_k}} QK^\top \in \mathbb{R}^{T\times T},$$

where

$$S_{tj} = \frac{1}{\sqrt{d_k}} \langle q_t, k_j\rangle$$

is the scaled similarity between query q_t at position t and key k_j at position j. The factor $1/\sqrt{d_k}$ controls the variance of the dot products when d_k is large, preventing them from becoming too large in magnitude and driving the softmax into regions with tiny gradients.

We then apply a softmax row-wise to obtain attention weights:

$$\alpha_{tj} = \frac{\exp(S_{tj})}{\sum_{\ell=1}^{T} \exp(S_{t\ell})}, \qquad A = \mathrm{softmax}_{\mathrm{row}}(S) \in \mathbb{R}^{T\times T},$$

so that each row of A sums to 1. Finally, each output vector is a weighted average of the values:

$$Y = AV \in \mathbb{R}^{T\times d_v}, \qquad y_t = \sum_{j=1}^{T} \alpha_{tj} v_j.$$

This is *scaled dot-product attention*. In batched form, we use the same formula but treat Q, K, V as $(B, T, \cdot)$ tensors and apply the operations over the last two dimensions for each batch element.

For autoregressive language modeling, we must ensure that position t never uses information from positions $> t$ when predicting x_t. We enforce this with a *causal mask*. Let $M \in \mathbb{R}^{T\times T}$ be a matrix with

$$M_{tj} = \begin{cases} 0, & j \le t, \\ -\infty, & j > t, \end{cases}$$

and define masked scores

$$S' = S + M.$$

When we compute $\mathrm{softmax}_{\mathrm{row}}(S')$, the entries with $j > t$ effectively receive probability zero. In code, we approximate $-\infty$ with a large negative constant such as -10^9.

When working with padded sequences (to form a rectangular batch from variable-length examples), we introduce a *padding mask* that sets scores involving padding tokens to $-\infty$ in the same way. The Colab will construct masks as Boolean tensors and then convert them into additive score masks before the softmax.

4.3 Tiny Numeric Example

To make the formulas concrete, consider $T = 3$ tokens and $d_k = d_v = 2$. Let

$$Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}, \quad K = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad V = \begin{bmatrix} 1 & 0 \\ 0 & 2 \\ 3 & 1 \end{bmatrix}.$$

First compute the (unscaled) score matrix $QK^\top$:

$$QK^\top = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 2 & 1 \end{bmatrix}.$$

With $d_k = 2$, we scale by $1/\sqrt{2}$ to obtain $S = QK^\top/\sqrt{2}$. Applying a row-wise softmax to S gives an attention matrix $A \in \mathbb{R}^{3\times3}$ with rows that sum to 1:

$$A = \mathrm{softmax}_{\mathrm{row}}(S).$$

Finally, the outputs are

$$Y = AV \in \mathbb{R}^{3\times2}.$$

In the Colab, we will:

1. implement a `numpy` (or `torch`) function that computes Y from Q, K, V using the attention formula,
2. print $QK^\top$, S, A, and Y to verify that the code matches the hand calculations above,
3. repeat the computation with a causal mask and observe how the first row remains unchanged while later rows lose access to future positions.

The resulting attention weights can be visualized as a small heatmap, as illustrated in Figure 1, which makes the effect of the causal mask immediately visible.

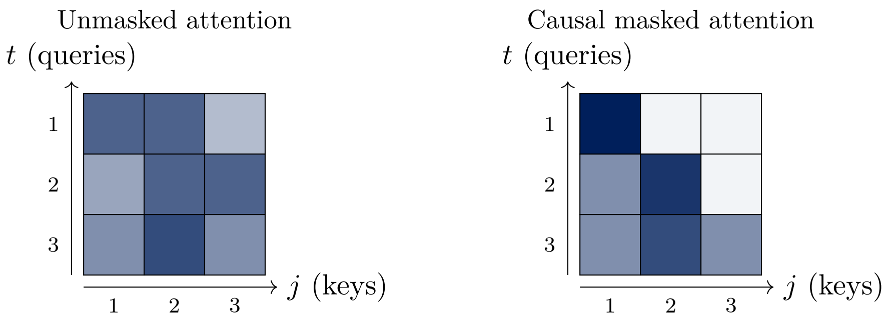

Figure 1: Attention weight matrices for the tiny numeric example: unmasked (left) and with a causal mask (right). Darker colors indicate higher attention weight; the mask zeroes out probability mass on future positions.

4.4 Colab Implementation Specs: `scaled_dot_product_attention`

The first core function in the notebook will implement scaled dot-product attention directly from the equations above. A minimal PyTorch-style skeleton is:

Skeleton `scaled_dot_product_attention`

```
def scaled_dot_product_attention(Q, K, V, mask=None):
    """
    Q, K, V: (..., T, d_k / d_v) tensors
    mask:    optional boolean or float mask broadcastable to (..., T, T)
    returns: (..., T, d_v)
    """
    # 1. Compute scaled scores
    # 2. Add mask (if given)
    # 3. Apply softmax along the last-but-one dimension
    # 4. Multiply by V to get the output
    ...
```

Design requirements:

- Accept both two-dimensional (T, d) inputs and batched (B, T, d) inputs by treating leading dimensions generically.
- Compute scores as $S = QK^\top/\sqrt{d_k}$ using a suitable batch matrix multiply (e.g., `torch.matmul`).
- If a mask is provided, convert it into additive scores (e.g., set entries corresponding to disallowed positions to a large negative number) before taking the softmax.
- Apply a numerically stable softmax by subtracting the row-wise maximum from S before exponentiating.
- Return an output tensor Y with the same leading dimensions as Q and last dimension d_v.

In the notebook, add at least one unit test that constructs the small example above, runs it through `scaled_dot_product_attention`, and checks that the intermediate and final tensors match the hand-computed values up to small numerical tolerance.

5 Self-Attention for Sequences

Having defined attention for an arbitrary collection of queries, keys, and values, we now specialize to *self*-attention, where all three are derived from the same sequence of token embeddings. This turns attention into a building block that can be stacked repeatedly to form transformer layers.

5.1 From Pairwise Attention to Self-Attention

In self-attention we apply the attention mechanism to a *single* sequence: every token position attends to every other position (subject to masks). Concretely, for embeddings

$$X \in \mathbb{R}^{T\times d_{\text{model}}}$$

we reuse the projection matrices from Section 3:

$$W_Q, W_K \in \mathbb{R}^{d_{\text{model}}\times d_k}, \qquad W_V \in \mathbb{R}^{d_{\text{model}}\times d_v},$$

and form

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V.$$

We then apply scaled dot-product attention with an appropriate mask:

$$Y = \text{Attention}(Q, K, V) \in \mathbb{R}^{T\times d_v},$$

so that the output y_t at each position t is a learned mixture of the value vectors from all positions $j \le t$ (for a causal mask) or all j (for bidirectional attention).

Intuitively, self-attention lets each token ask: "Which other tokens in this sequence are relevant for my representation?" For a language model, a token may attend to recent punctuation, matching brackets, or long-range subject–verb dependencies, all via learned similarity in the query–key space.

One important structural property is that self-attention is *permutation-equivariant*: if we shuffle the rows of X in the same way for Q, K, and V, the rows of Y are shuffled in the same way. This means that self-attention, by itself, does not know about absolute or relative positions—a fact that will motivate positional encodings in Section 7.

5.2 Complexity and Parallelism

The main computational cost of self-attention comes from forming the score matrix $S \in \mathbb{R}^{T\times T}$ and the attention matrix A. For model dimension d (suppressing the distinction between d_{model}, d_k, and d_v), the complexity of a single attention layer is roughly

$$\text{time} = O(T^2 d), \qquad \text{memory} = O(T^2),$$

since we compute all T^2 pairwise query–key dot products.

In contrast, a simple RNN layer with hidden size d has per-step cost $O(d^2)$, yielding $O(Td^2)$ overall, but the steps *cannot* be parallelized across time: each h_t depends on h_{t-1}. Self-attention is the opposite trade-off:

- more expensive in T (quadratic vs. linear),
- but highly parallelizable across positions and heads on modern accelerators.

In practice, the context length T is therefore limited by memory. In the capstone we choose a small `block_size` (e.g., $T = 64$ or $T = 128$) so that:

- all intermediate tensors comfortably fit on a Colab GPU,
- you can still easily print and inspect full attention matrices for debugging and visualization.

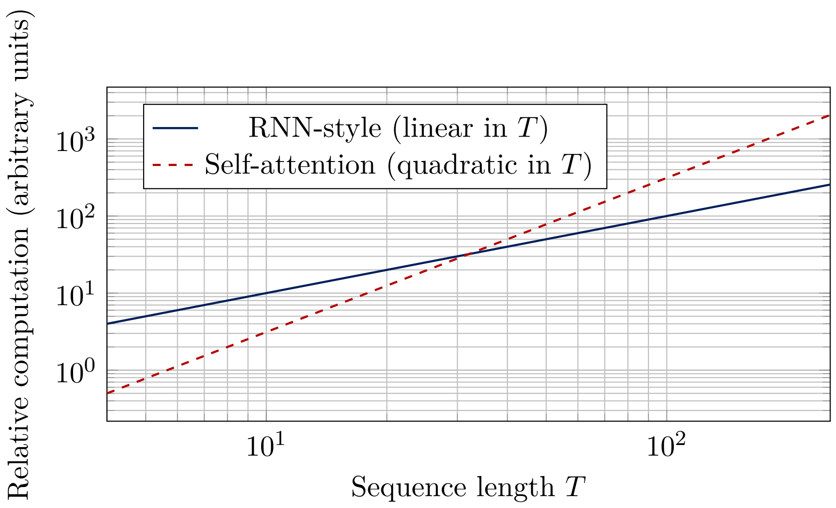

Figure 2: Illustrative comparison of how computation scales with sequence length for RNN-style models (linear in T for fixed hidden size) versus self-attention (quadratic in T). The vertical axis is in arbitrary units; only the slope matters.

5.3 Implementation Specs: `SelfAttention` Layer

In the notebook, we will wrap single-head self-attention into a small PyTorch module that reuses `scaled_dot_product_attention`. A minimal structure is:

`SelfAttention` (single head)

```
class SelfAttention(nn.Module):
    def __init__(self, d_model, d_k=None, d_v=None, causal=True):
        super().__init__()
        d_k = d_k or d_model
        d_v = d_v or d_model
        self.W_Q = nn.Linear(d_model, d_k, bias=False)
        self.W_K = nn.Linear(d_model, d_k, bias=False)
        self.W_V = nn.Linear(d_model, d_v, bias=False)
        self.causal = causal

    def forward(self, x):
        # x: (B, T, d_model)
        Q = self.W_Q(x)    # (B, T, d_k)
        K = self.W_K(x)    # (B, T, d_k)
        V = self.W_V(x)    # (B, T, d_v)
        mask = make_causal_mask(x) if self.causal else None
        y = scaled_dot_product_attention(Q, K, V, mask=mask)
        return y           # (B, T, d_v)
```

Key design points:

- The input shape is (B, T, d_{model}); the output has shape (B, T, d_v) (often with $d_v = d_{\text{model}}$ so that we can add residual connections later).
- The projection layers can be implemented with `nn.Linear` as above or with explicit `nn.Parameter` matrices and manual matrix multiplies.
- The causal mask construction `make_causal_mask` should produce a broadcastable tensor of shape $(1, T, T)$ or $(B, 1, T, T)$ that is compatible with `scaled_dot_product_attention`.
- You can add optional dropout on the attention weights or outputs as a later enhancement; keep the base implementation as simple and transparent as possible.

Once implemented, you can verify correctness by:

- checking that for $W_Q = W_K = W_V = I$ and an identity mask, the layer reduces to a simple average over positions,
- comparing against a naive double-loop implementation on a tiny batch and sequence length.

6 Multi-Head Attention and Transformer Blocks

With single-head self-attention in place, we can enrich the model by using multiple heads and by combining attention with simple MLPs, residual connections, and normalization. This section sketches how these ingredients fit together into a standard transformer block.

6.1 Motivation for Multiple Heads

Single-head self-attention is already expressive, but in practice it can be useful to let the model learn *several* different attention patterns in parallel. Multi-head attention achieves this by:

- splitting the model dimension d_{model} into H smaller *head dimensions* d_{head},
- learning separate projection matrices $(W_Q^{(h)}, W_K^{(h)}, W_V^{(h)})$ for each head h,
- applying attention independently in each head, then concatenating and mixing the results.

Intuitively, different heads can specialize:

- one head might focus on local patterns (e.g., bigrams or trigrams),

- another on long-range dependencies (e.g., subject–verb agreement across clauses),
- another on structural cues (e.g., matching brackets or indentation in code).

The original Transformer paper reports that multi-head attention (with, e.g., $H = 8$) significantly improves performance over single-head attention for the same parameter budget.

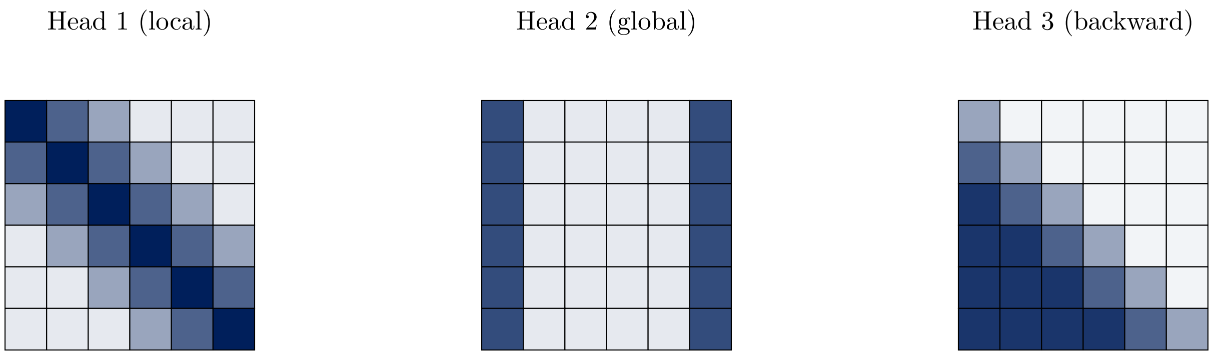

Figure 3: Schematic attention patterns for three different heads over a sequence of length 6. In practice, heads often specialize to different roles (e.g., local dependencies, global structure), and visualization of learned attention maps can reveal these patterns.

6.2 Shapes and Concatenation

Let d_{model} be divisible by the number of heads H, and define the head dimension

$$d_{\text{head}} = d_{\text{model}}/H.$$

Given an input $X \in \mathbb{R}^{B\times T\times d_{\text{model}}}$, we typically implement multi-head attention with a *single* linear layer that produces all queries, keys, and values at once:

$$X \mapsto [Q\ K\ V] \in \mathbb{R}^{B\times T\times 3d_{\text{model}}}.$$

We then reshape and permute to expose the head dimension:

$$Q, K, V \in \mathbb{R}^{B\times T\times H\times d_{\text{head}}} \rightsquigarrow Q, K, V \in \mathbb{R}^{B\times H\times T\times d_{\text{head}}},$$

and apply scaled dot-product attention *per head*, yielding

$$Y^{(h)} \in \mathbb{R}^{B\times T\times d_{\text{head}}}, \qquad h = 1, \dots, H.$$

Concatenating the head outputs along the last dimension gives

$$Y_{\text{concat}} \in \mathbb{R}^{B\times T\times (Hd_{\text{head}})} = \mathbb{R}^{B\times T\times d_{\text{model}}},$$

which we finally pass through an *output projection* matrix

$$W_O \in \mathbb{R}^{d_{\text{model}}\times d_{\text{model}}}$$

to obtain the multi-head attention output with the same shape as the input. In the notebook, printing tensor shapes after each reshape/transpose step is an effective way to debug this part.

6.3 Implementation Specs: `MultiHeadAttention`

The multi-head attention module in the Colab will closely follow the shape discussion above. A typical skeleton is:

`MultiHeadAttention` skeleton

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads, dropout=0.0, causal=True):
        super().__init__()
        assert d_model % num_heads == 0
        self.d_model = d_model
        self.num_heads = num_heads
        self.d_head = d_model // num_heads
        self.causal = causal

        self.proj_qkv = nn.Linear(d_model, 3 * d_model, bias=False)
        self.proj_out = nn.Linear(d_model, d_model, bias=False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        B, T, D = x.shape  # D = d_model
        qkv = self.proj_qkv(x)              # (B, T, 3*D)
        qkv = qkv.view(B, T, 3, self.num_heads, self.d_head)
        Q, K, V = qkv.unbind(dim=2)         # each: (B, T, H, d_head)
        Q = Q.transpose(1, 2)               # (B, H, T, d_head)
        K = K.transpose(1, 2)
        V = V.transpose(1, 2)

        mask = make_causal_mask_from_T(T) if self.causal else None
        # scaled_dot_product_attention should support (..., T, d_head) shapes
        Y = scaled_dot_product_attention(Q, K, V, mask=mask)  # (B, H, T,
        ↪  d_head)

        Y = Y.transpose(1, 2).contiguous().view(B, T, D)      # concat heads
        Y = self.proj_out(Y)                                  # (B, T, D)
        Y = self.dropout(Y)
        return Y
```

Design and testing notes:

- The causal mask should be broadcastable to shape (B, H, T, T); one simple strategy is to construct a single $(1, 1, T, T)$ mask and rely on broadcasting.
- Using a single linear layer `proj_qkv` is more efficient than separate W_Q, W_K, W_V, but both approaches are instructive and easy to implement.
- A simple test is to set all projection matrices so that each head receives identical parameters and to confirm that the multi-head output equals that of a single-head attention layer (up to numerical precision).

6.4 Position-wise Feedforward Network

Each transformer block includes, in addition to attention, a small MLP applied *independently at each position*. For input $x \in \mathbb{R}^{d_{\text{model}}}$ at a single position, the feedforward network is typically

$$\text{FFN}(x) = W_2\,\sigma(W_1 x + b_1) + b_2,$$

where σ is a nonlinearity such as ReLU or GELU. In matrix form, applied to all positions at once,

$$X \in \mathbb{R}^{B\times T\times d_{\text{model}}} \;\mapsto\; \text{FFN}(X) \in \mathbb{R}^{B\times T\times d_{\text{model}}},$$

with the same parameters shared across all positions.

The hidden dimension d_{ff} (the width of the intermediate layer) is usually larger than d_{model}, e.g., $d_{\text{ff}} = 4d_{\text{model}}$ in many transformer variants. This increases the representational capacity of the block beyond what attention alone can provide.

In the Colab, the FFN can be implemented as a simple `nn.Sequential`:

Position-wise feedforward network

```
ffn = nn.Sequential(
    nn.Linear(d_model, d_ff),
    nn.GELU(),
    nn.Linear(d_ff, d_model),
)
```

Optionally, you can add dropout between the layers. Since the FFN is applied identically at each position, it behaves like a 1×1 convolution over the sequence dimension.

6.5 Residual Connections and Layer Normalization

Deep stacks of attention and FFN layers benefit from two stabilization techniques:

- **Residual (skip) connections.** Instead of replacing x with the output of a sublayer, we add them: $x \mapsto x + \text{SubLayer}(x)$. This makes optimization easier, since each sublayer only needs to learn a residual correction to the identity.
- **Layer normalization.** A `LayerNorm` normalizes activations across the feature dimension, helping to keep them in a reasonable range and improving training stability in deep networks.

There are two common patterns: *Post-LN* (as in the original Transformer) and *Pre-LN* (as in many modern GPT-style models). For this capstone we adopt a Pre-LN structure, which tends to be more stable:

$$\begin{aligned} \text{Attention sublayer:} \quad & x' = x + \text{MHA}\big(\text{LN}_1(x)\big), \\ \text{Feedforward sublayer:} \quad & y = x' + \text{FFN}\big(\text{LN}_2(x')\big). \end{aligned}$$

Here LN_1 and LN_2 are independent layer norms, and MHA is the multi-head attention module.

In the notebook, these pieces are wrapped into a single `TransformerBlock` module:

- multi-head self-attention + residual connection + layer norm,
- position-wise feedforward network + residual connection + layer norm.

You can verify the block by checking shapes and by ensuring it behaves sensibly on simple inputs (e.g., identity behavior when weights are near zero).

7 Positional Encodings

Self-attention by itself is agnostic to the order of tokens. Here we explain why explicit position information is necessary for language modeling and outline the positional encoding schemes used in the tiny transformer.

7.1 Why Positions Matter

Self-attention is permutation-equivariant: if we permute the input tokens and apply the same permutation to all positions, the outputs are permuted in the same way. This is a strength in some settings (e.g., sets), but for language we must distinguish:

$$\text{“dog bites man”} \quad \text{vs.} \quad \text{“man bites dog”}.$$

Without any notion of position, the model only knows which tokens co-occur, not in which order.

To break this symmetry, we add a position-dependent vector to each token embedding. For each position index $t \in \{1, \dots, T\}$ we define a positional encoding $\mathrm{PE}_t \in \mathbb{R}^{d_{\text{model}}}$ and form

$$Z_t = e_t + \mathrm{PE}_t,$$

where e_t is the token embedding. The sequence Z is then fed into the transformer blocks instead of X. This lets the attention mechanism learn not only *which* tokens to attend to, but also *where* they appear.

7.2 Sinusoidal Positional Encodings

One popular choice, introduced in the original Transformer paper, is to use fixed sinusoidal positional encodings. For position $pos \geq 0$ and dimension index i (starting at 0), define

$$\mathrm{PE}_{(pos,2i)} = \sin\!\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right),$$
$$\mathrm{PE}_{(pos,2i+1)} = \cos\!\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right).$$

Each pair of dimensions $(2i, 2i+1)$ corresponds to a sinusoid with a specific wavelength. Lower i gives slower variation across positions (capturing coarse positional trends), while higher i gives faster variation (capturing fine-grained differences between nearby positions).

These encodings have a few convenient properties:

- they can be computed for arbitrarily long sequences without additional training,
- relative offsets between positions can be expressed as linear functions of the encodings,
- no extra parameters are introduced, keeping the tiny model simple.

In the Colab, we will precompute a matrix $\mathrm{PE} \in \mathbb{R}^{T_{\max} \times d_{\text{model}}}$ of sinusoidal encodings for all positions up to a maximum context length $T_{\max}$ and reuse it across batches.

As an optional exercise, you can implement *learned* positional embeddings (another `nn.Embedding` over position indices) and compare performance to the sinusoidal scheme.

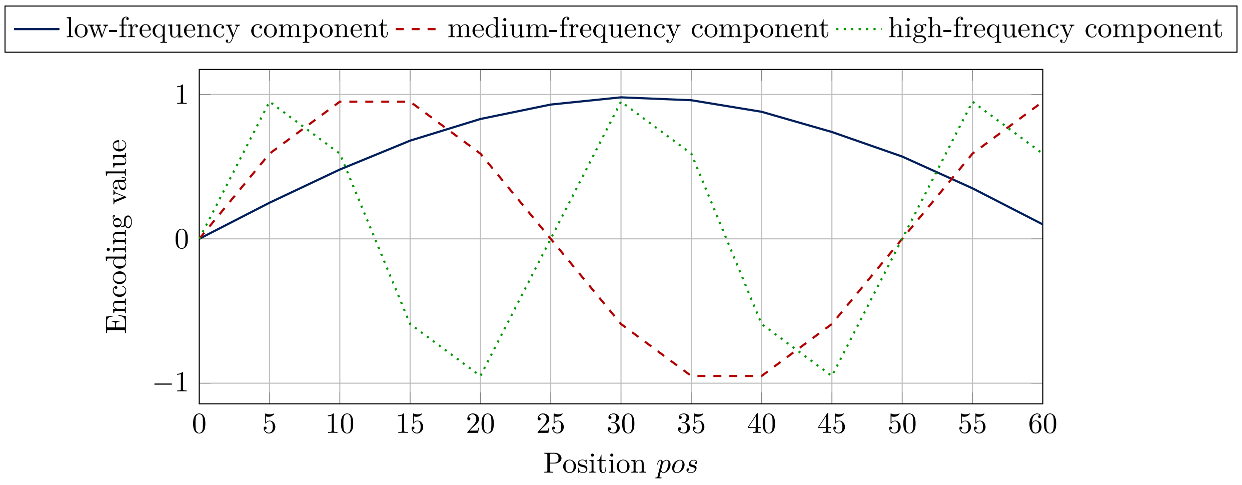

Figure 4: Schematic illustration of different sinusoidal components used in positional encodings. Lower frequencies change slowly with position (capturing coarse trends), while higher frequencies oscillate more rapidly (capturing fine-grained positional differences).

7.3 Implementation Specs: `PositionalEncoding`

The positional encoding module in the notebook will encapsulate the construction and use of PE. A minimal PyTorch implementation looks like:

`PositionalEncoding` skeleton

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len):
        super().__init__()
        pe = torch.zeros(max_len, d_model)        # (T_max, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(
            torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model)
        )
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)                      # (1, T_max, d_model)
        self.register_buffer('pe', pe)           # not a parameter

    def forward(self, x):
        # x: (B, T, d_model)
        T = x.size(1)
        return x + self.pe[:, :T, :]
```

Important details:

- `register_buffer` ensures that PE moves with the module across devices but is not updated by gradients.
- The `forward` method slices the precomputed encodings to the current sequence length and adds them to the token embeddings.
- The maximum length `max_len` should be at least as large as the chosen `block_size`; for the capstone, a modest value (e.g., 256 or 512) is sufficient.

8 A Tiny Decoder-Only Transformer for Language Modeling

Bringing all ingredients together, this section describes the end-to-end tiny transformer language model used in the capstone: the task, dataset, architecture, training objective, and the structure of the Colab implementation.

8.1 Task Definition and Dataset

Our capstone model is a small *decoder-only* transformer that performs autoregressive next-token prediction. Given a context $(x_1, \dots, x_{t-1})$, it outputs a distribution over the next token x_t and is trained to maximize the likelihood of the observed sequences in a corpus.

To keep the focus on modeling rather than data engineering, we will use a simple dataset:

- either character-level modeling on a short text corpus (e.g., concatenated quotes, a small story, or a code file),
- or token-level modeling on a few thousand tokens obtained via a lightweight tokenizer.

The preprocessing pipeline in Colab will:

- construct a vocabulary (set of unique characters or tokens) and assign each a unique integer ID,

- map the raw text to a long sequence of integer IDs,
- split this sequence into a training part and a validation part (e.g., 90%/10%),
- sample fixed-length contexts of size `block_size` by slicing contiguous windows from the training sequence.

For a batch of size B, the input tensor will have shape (B, T) with $T =$ `block_size` and integer entries, and the corresponding target tensor will contain the same sequences shifted by one position to the right. Minimal Python utilities (no external data loaders) suffice for this setup.

8.2 Model Architecture Overview

The tiny transformer LM follows the standard GPT-style blueprint but at much smaller scale. A typical forward pass looks like:

1. **Token embeddings.** Integer token IDs $X_{\text{idx}} \in \{0, \dots, V-1\}^{B\times T}$ are mapped to embeddings
$$E(X_{\text{idx}}) \in \mathbb{R}^{B\times T\times d_{\text{model}}},$$
using a learnable embedding matrix $E \in \mathbb{R}^{V\times d_{\text{model}}}$.

2. **Positional encodings.** Sinusoidal (or learned) positional encodings $\text{PE} \in \mathbb{R}^{1\times T\times d_{\text{model}}}$ are added, giving
$$Z = E(X_{\text{idx}}) + \text{PE}_{[:,:T,:]}.$$

3. **Transformer blocks.** The sequence Z passes through a stack of L identical `TransformerBlocks`. Each block consists of:
   - multi-head self-attention with a causal mask,
   - a position-wise feedforward network,
   - residual connections and layer normalization as described earlier.

   Denote the output after the last block by $H \in \mathbb{R}^{B\times T\times d_{\text{model}}}$.

4. **Output projection.** A final linear layer maps H to unnormalized logits over the vocabulary:
$$\text{logits} = HW_{\text{out}}^{\top} + b_{\text{out}} \in \mathbb{R}^{B\times T\times V},$$
where $W_{\text{out}} \in \mathbb{R}^{V\times d_{\text{model}}}$.

In the Colab, these components are wrapped into a single `TinyTransformerLM` class with a `forward` method that returns logits for all positions. A small table of hyperparameters (e.g., $d_{\text{model}}, d_{\text{ff}},$ `num_heads`, `num_layers`, `block_size`, `batch_size`) will guide typical choices for the tiny model.

8.3 Loss Function and Training Objective

At each position t, the model outputs a vector of logits $\ell_t \in \mathbb{R}^V$ representing unnormalized log-probabilities over the vocabulary. After applying softmax, this defines a conditional distribution
$$p_\theta(x_t \mid x_{<t}) = \text{softmax}(\ell_t)_{x_t}.$$

For a single sequence, the negative log-likelihood loss is
$$L(\theta) = -\sum_{t=1}^{T} \log p_\theta(x_t \mid x_{<t}),$$
and for a batch we average this quantity across sequences and positions.

In PyTorch, we implement this with `nn.CrossEntropyLoss` applied to reshaped tensors:

- reshape logits from (B, T, V) to (BT, V),
- reshape targets from (B, T) to (BT),
- apply cross-entropy to get a scalar loss.

For character-level models, reporting the loss in *bits per character* (bpc) via bpc $= L/(\log 2)$ is common; for token-level models, *perplexity* $\exp(L)$ is another useful metric. The validation loss/bpc/perplexity over held-out data serves as the main gauge of generalization.

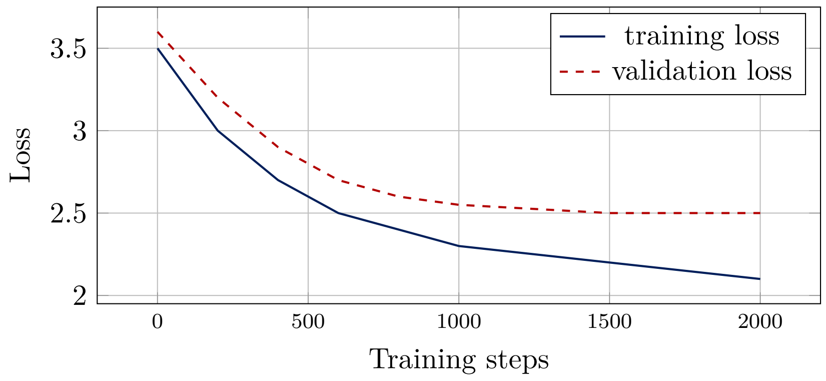

Figure 5: Illustrative training and validation loss curves for the tiny transformer language model. In practice, you should monitor both to detect overfitting and to decide when further training yields diminishing returns.

8.4 Colab Implementation Roadmap (Model and Training Loop)

The notebook will mirror the conceptual structure of this handout. A suggested sequence of code cells:

1. **Configuration.** Define a simple configuration object or dictionary with hyperparameters (vocabulary size, d_{model}, `num_heads`, `num_layers`, `block_size`, `batch_size`, learning rate, etc.).
2. **Data utilities.** Implement functions to:
   - load raw text,
   - build vocabulary and encode/decode between text and integer sequences,
   - sample mini-batches of input and target sequences of shape (B, T) from the training and validation splits.
3. **Core modules.** Implement:
   - `scaled_dot_product_attention`,
   - `SelfAttention` and `MultiHeadAttention`,
   - `PositionalEncoding`,
   - `TransformerBlock`.
4. **Model class.** Assemble these components into a `TinyTransformerLM` `nn.Module` with:
   - token embedding and optional output embedding weight tying,
   - a stack of L transformer blocks,
   - a final linear projection to logits.

5. **Training loop.** Write a loop that:
   - samples batches, computes logits, reshapes them, evaluates cross-entropy loss,
   - runs backpropagation (`loss.backward()`),
   - performs an optimizer step and zeroes gradients.
6. **Logging and checkpoints.** Add periodic logging of training and validation loss, and (optionally) save model checkpoints using `state_dict()`.

For optimization, we recommend Adam with a modest learning rate (e.g., $3 \cdot 10^{-4}$), no weight decay for the tiny model, and optional gradient clipping if gradients become unstable. A few thousand training steps are typically enough to see recognizable patterns in the generated samples on a small corpus.

8.5 Sampling and Qualitative Evaluation

After training, the most satisfying way to assess the model is to sample text. The sampling procedure repeatedly:

1. takes a current context (a sequence of token IDs of length $\leq$ `block_size`),
2. feeds it through the model to obtain logits for the last position,
3. converts logits to probabilities (optionally adjusting with a temperature parameter),
4. draws the next token according to a chosen sampling rule,
5. appends the sampled token to the context and repeats.

In the Colab, you can implement:

- **Greedy sampling:** at each step pick the token with the highest probability,
- **Temperature sampling:** divide logits by a temperature $\tau > 0$ before softmax; $\tau < 1$ makes outputs more peaked, $\tau > 1$ makes them more random,
- **Top-k or top-p sampling (optional):** restrict sampling to the top-k tokens or the smallest set of tokens whose probabilities sum to at least p.

Use short prompts as seeds (e.g., a word or two, or a short code prefix) and qualitatively inspect the generated continuations. Signs that the model is learning include:

- locally coherent character or token sequences (correct spelling or syntax patterns),
- reasonable repetition and diversity, rather than a single loop or degenerate output,
- improved quality when lowering the temperature slightly from 1.0.

Comparing samples at different training checkpoints is a useful way to build intuition for how transformer LMs improve over time.

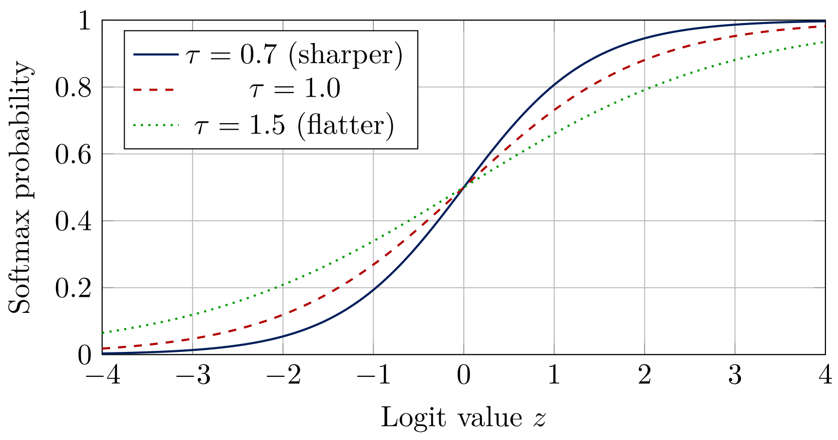

Figure 6: Effect of temperature on a simple two-class softmax. Lower temperature ($\tau < 1$) makes the distribution more peaked around the largest logit, while higher temperature ($\tau > 1$) spreads probability mass more evenly, increasing randomness in sampling.

9 Summary and Connections

We conclude by collecting the main ideas introduced in this note and relating them back to the optimization and backpropagation concepts from earlier weeks, as well as to the practical Colab implementation.

- We started from the sequence modeling problem and showed how attention provides a flexible, differentiable way to retrieve information from a context based on content similarity.
- We defined scaled dot-product attention, extended it to self-attention over token embeddings, and examined its computational characteristics and masking schemes.
- We then introduced multi-head attention, position-wise feedforward networks, residual connections, and layer normalization, assembling them into a standard transformer block.
- Positional encodings supplied the missing notion of order, enabling transformers to distinguish between sequences that share the same tokens but in different positions.
- Finally, we combined these pieces into a tiny decoder-only transformer language model, specified its training objective, and mapped the concepts to a concrete Colab implementation and sampling routines.

Conceptually and technically, this capstone builds directly on earlier weeks:

- from Week 1, we reuse gradient-based optimization for training the model parameters;
- from Week 2, we rely on the chain rule and computation graphs (now much larger) to justify how gradients flow through attention, residuals, and normalization layers.

By the time you complete the accompanying Colab, you should be able to:

- derive and interpret the equations for scaled dot-product attention and self-attention,
- implement attention, multi-head attention, positional encodings, and transformer blocks from scratch in PyTorch,
- build, train, and sample from a small transformer language model on a toy corpus,

- debug implementation issues by comparing against the numeric and shape-based guidance in this handout.

The same ideas extend, with larger models and datasets, to state-of-the-art language models, but the core mathematics and code patterns remain essentially the same as in this tiny transformer.

A Shape Cheat Sheet and Notation

This appendix compiles the most important symbols and tensor shapes used throughout the handout for quick reference while working in the notebook.

Indices and sizes.

- B — batch size (number of sequences processed in parallel), typically small (e.g., $B = 32$ or $B = 64$).
- T — sequence length / context length (also called `block_size`); for the tiny model T is modest (e.g., $T = 64$ or $T = 128$).
- V — vocabulary size (number of distinct characters or tokens).
- d_{model} — model dimension (width of embeddings and hidden states inside transformer blocks).
- d_k — query/key dimension per head.
- d_v — value dimension per head.
- H — number of attention heads.
- d_{head} — per-head model dimension, with $d_{\text{model}} = H \cdot d_{\text{head}}$ in the standard setup.
- d_{ff} — hidden dimension of the feedforward network (often $4d_{\text{model}}$).

Core tensors.

- Token indices: $X_{\text{idx}} \in \{0, \ldots, V-1\}^{B\times T}$ — integer IDs before embedding (not a float tensor).
- Token embeddings: $E(X_{\text{idx}}) \in \mathbb{R}^{B\times T\times d_{\text{model}}}$.
- Positional encodings: $\text{PE} \in \mathbb{R}^{1\times T_{\max}\times d_{\text{model}}}$ (buffer in `PositionalEncoding`); slice to length T before adding.
- Transformer block inputs/outputs: $X, H \in \mathbb{R}^{B\times T\times d_{\text{model}}}$.
- Logits: $\text{logits} \in \mathbb{R}^{B\times T\times V}$ before reshaping for the loss.

Attention-specific tensors.

- Single-head projections (per block):
$$Q, K \in \mathbb{R}^{B\times T\times d_k}, \qquad V \in \mathbb{R}^{B\times T\times d_v}.$$
- Score matrix (single head, single sequence): $S \in \mathbb{R}^{T\times T}$, with entries $S_{tj} = \langle q_t, k_j\rangle/\sqrt{d_k}$.
- Attention weights: $A = \text{softmax}_{\text{row}}(S) \in \mathbb{R}^{T\times T}$.
- Attention output (single head, single sequence): $Y = AV \in \mathbb{R}^{T\times d_v}$.
- Causal mask: $M \in \mathbb{R}^{T\times T}$ with $M_{tj} = 0$ for $j \le t$ and $M_{tj} = -\infty$ for $j > t$, broadcast to $(1, 1, T, T)$ or similar in code.

Multi-head shapes.

- Combined QKV projection: $qkv \in \mathbb{R}^{B \times T \times 3d_{\text{model}}}$ from `proj_qkv`.
- After reshaping: $Q, K, V \in \mathbb{R}^{B \times T \times H \times d_{\text{head}}}$.
- Head-major view: $Q, K, V \in \mathbb{R}^{B \times H \times T \times d_{\text{head}}}$.
- Per-head outputs: $Y^{(h)} \in \mathbb{R}^{B \times T \times d_{\text{head}}}$, concatenated into $Y_{\text{concat}} \in \mathbb{R}^{B \times T \times d_{\text{model}}}$.

Where they appear in code.

- Configuration fields: `d_model`, `num_heads`, `d_ff`, `block_size`, `batch_size`, `vocab_size`.
- Model attributes: embedding layer weights E, projection layers W_Q, W_K, W_V, output projection W_{out}, feedforward layers, and layer norms within `TransformerBlock`.
- Utility functions: attention masks constructed from `block_size` and sequence lengths, used inside `SelfAttention` and `MultiHeadAttention`.

B Detailed Numeric Examples

This appendix outlines small numeric test cases that can be mirrored exactly in the Colab notebook for debugging and unit tests.

Scaled Dot-Product Attention (3 Tokens, 2D Vectors)

- Choose $T = 3$, $d_k = d_v = 2$ and explicit matrices Q, K, V with small integers (e.g., the example in Section 3).
- Compute $QK^\top$ by hand, then scale by $1/\sqrt{d_k}$ to get S.
- Apply a row-wise softmax to obtain A and multiply by V to get Y.
- In Colab, implement `scaled_dot_product_attention`, print $QK^\top$, S, A, and Y, and compare to the hand calculation.

Self-Attention with Causal Mask

- Reuse the same Q, K, V but apply a causal mask so that position t cannot attend to positions $> t$.
- Manually zero out (or set to $-\infty$ before softmax) the entries above the main diagonal in S.
- Recompute A and Y with masking and note how the second and third rows change relative to the unmasked case.
- In Colab, construct the same mask tensor and verify that the implementation matches the masked hand computation.

Single Transformer Block on a Tiny Sequence

- Pick a very small configuration: $B = 1$, $T = 3$, $d_{\text{model}} = 4$, $H = 2$, $d_{\text{ff}} = 8$.
- Initialize all projection and feedforward weights with simple values (e.g., small integers or scaled identity matrices) so that manual computation is feasible.
- Run one forward pass through:
  - positional encoding addition,
  - a single `TransformerBlock` (multi-head attention + FFN).
- Print intermediate tensors (Q, K, V, attention weights, block outputs) in Colab and check that their shapes and rough magnitudes match expectations.

Loss and Sampling Sanity Checks

- Verify that for a model that always outputs uniform logits, the loss is close to $\log V$ and samples look like random noise over the vocabulary.
- Construct a toy dataset with a trivial pattern (e.g., repeating "AB" or simple arithmetic sequences) and check that the tiny model can quickly overfit and produce nearly perfect samples.

C Suggested Extensions and Experiments

The final appendix outlines optional experiments for delegates who want to go beyond the core capstone, using the same tiny transformer as a starting point.

Architectural Variations

- Vary depth (L) and width ($d_{\text{model}}, d_{\text{ff}}$) to study how model capacity affects training loss and sample quality on the same dataset.
- Try different numbers of heads H while keeping d_{model} fixed to investigate the impact of multi-head structure.
- Replace sinusoidal positional encodings with learned positional embeddings and compare validation performance and sample characteristics.

Training and Optimization Experiments

- Explore different learning rates, batch sizes, and gradient clipping thresholds; monitor how they affect convergence speed and stability.
- Add weight decay or dropout in attention and feedforward layers, and observe changes in overfitting on small corpora.
- Experiment with simple learning rate schedules (e.g., warmup + cosine decay) and compare to a constant learning rate.

Sampling and Analysis

- Compare greedy, temperature, top-k, and top-p sampling on the same trained model; qualitatively assess diversity and coherence.
- Log and visualize attention maps for a handful of sequences to see which tokens each head focuses on; relate this to intuitive linguistic or structural patterns.
- Track and plot training/validation loss curves and, if applicable, bits-per-character or perplexity over time.

Beyond Text

- Apply the same architecture to non-text sequences (e.g., simple 1D signals or event streams) to see how attention behaves in other domains.
- For code-savvy delegates, try training on a small code corpus and examine whether the model learns indentation and bracket-matching patterns.