Transformer — Sections 1–4: Course Notes

Core training, Autoregressive inference, Scaling Efficiency, KV cache

This file covers Sections 1–4 in a single, self-contained LaTeX manual.

Contents

1 Section 1 — Core training (Next-token pretraining)

1.1 Overview

Pretraining a large language model (LLM) usually means training a Transformer to predict the *next token* given preceding tokens. This objective is called **causal language modeling** or **next-token prediction**.

Next-token objective (causal LM): given a token sequence $w_1, \ldots, w_T$, the loss at position t is

$$\mathcal{L}_t = -\log p(w_t \mid w_{<t})$$

and the training loss is the average across positions and batch elements.

1.2 Why next-token prediction?

- It provides a simple, scalable training signal across billions of tokens.
- Parallelizable on GPUs during training because the full sequence of logits can be computed simultaneously.
- Trained models can be turned into autoregressive generators at inference by sampling from $p(w_t \mid w_{<t})$.

1.3 Data packing and context windows

Training is performed on fixed-size context windows (e.g., 4k, 8k tokens). Documents of varying length are packed into windows to maximize GPU utilization. Packing details matter for performance and for ensuring the model sees diverse contexts.

Practical note: when packing, add a separator token (e.g. `<|endofdoc|>`) between documents to avoid accidental context bleed.

1.4 From embeddings to logits (high level)

- Token ids $\rightarrow$ learned embeddings $E \in \mathbb{R}^{|V| \times d}$ produce $X \in \mathbb{R}^{N \times d}$.
- Transformer stacks process X into contextual token vectors $H^{(L)} \in \mathbb{R}^{N \times d}$.
- Final linear projection (often tied to E) maps the final token vector(s) to vocabulary logits.

From final hidden state to logits: W_{vocab}

The final hidden vector for a position t, denoted $h_t \in \mathbb{R}^d$, is converted to logits over the vocabulary by a linear projection:

$$\nu_t = W_{\text{vocab}}^{\top} h_t \in \mathbb{R}^{|V|},$$

where $W_{\text{vocab}} \in \mathbb{R}^{|V| \times d}$ (or its transpose) is the *vocabulary projection matrix*. Each entry $u_{t,i}$ is an unnormalized score for token i; applying softmax produces probabilities $p(w_i \mid \cdot)$. Frequently W_{vocab} is *tied* to the embedding matrix E (i.e. $W_{\text{vocab}} = E$), which reduces parameters and empirically improves performance.

2 Section 2 — Autoregressive inference (generation)

2.1 Key difference: training vs inference

- **Training:** compute representations for all positions in parallel using batched matrix multiplications (fast on GPUs).
- **Inference:** generation is *sequential* — produce one token at a time, conditioned on previously generated tokens.

Expanded note: training vs inference

During **training** the model processes an entire sequence (or packed window) at once: all token embeddings are projected, all transformer layers run in parallel across positions, and gradients are computed with backpropagation to update model weights. In contrast, during **inference** the model's weights are frozen; the model receives (or generates) tokens step-by-step. For autoregressive generation we produce x_t conditioned on $x_{<t}$ and append it to the context. Inference does not perform any gradient updates — it only runs forward passes — but it still requires substantial computation because attention must consider all tokens in the growing context. KV caching (Section 4) and efficient kernels mitigate this cost.

2.2 Causal masking

During inference (and during causal training), attention must not access future tokens. We implement this by adding a mask M to attention logits where $M_{ij} = -\infty$ for $j > i$ so softmax zeroes those probabilities.

Causal (autoregressive) mask:

$$M_{ij} = \begin{cases} 0 & j \le i, \\ -\infty & j > i. \end{cases}$$

Why this mask enforces autoregressivity

Attention first computes raw logits $S_{ij} = Q_i \cdot K_j / \sqrt{d_k}$ for every pair (i, j). Adding the matrix M sets $S_{i,j} = -\infty$ when $j > i$. The softmax over the columns then assigns zero probability to those positions: $\text{softmax}(-\infty) = 0$. Concretely, for sequence length N the mask matrix is upper-triangular with zeros on and below the diagonal and $-\infty$ above — this forces each position i to attend only to positions $1..i$ (including itself). Use a large negative number (e.g. -1×10^9) instead of exact $-\infty$ in float16 implementations to avoid numerical issues.

2.3 Why inference is expensive naively

Naively, computing attention at step t recomputes many linear projections for all t tokens, which becomes inefficient as generation continues. Section 3 and 4 explain how to remove this redundancy.

2.4 Masked self-attention: what is computed and why

Self-attention produces a contextualized output for every position by *mixing* value vectors from other positions. The key computations (single head) are:

$$Q = XW_Q,$$
$$K = XW_K,$$
$$V = XW_V,$$

where $X \in \mathbb{R}^{N \times d}$ are the input vectors at a given layer and $W_Q, W_K, W_V \in \mathbb{R}^{d \times d_k}$ are learned projection matrices. For position i the raw (unnormalized) attention scores against position j are

$$S_{ij} = \frac{Q_i \cdot K_j}{\sqrt{d_k}}.$$

After applying the causal mask and softmax we obtain attention weights

$$\alpha_{i,j} = \text{softmax}_j(S_{ij} + M_{ij}), \qquad \sum_j \alpha_{i,j} = 1.$$

The *attention vector* (the output of attention for position i) is the weighted sum of values

$$\text{attn}_i = \sum_{j=1}^{i} \alpha_{i,j} V_j.$$

Intuition: Q_i asks *what I need* at position i; K_j says *what each position j offers*; the dot product $Q_i \cdot K_j$ quantifies relevance; $\alpha_{i,j}$ rescales these into a probability; finally the attention vector is the mixture of content (V_j) weighted by relevance.

What alpha$_{i,j}$ does: it distributes one unit of "attention mass" across past positions; a large $\alpha_{i,j}$ means token j strongly influences the representation at i. This is how the model routes information: pronoun resolution, subject-verb agreement, copying tokens, and long-range reasoning are all implemented via large weights on relevant positions.

2.5 Q/K/V vs learned projection matrices

A common confusion: W_Q, W_K, W_V are *learned matrices* (model parameters), while Q_i, K_i, V_i are *vectors computed* per token as linear projections of the token's current representation. That is

$$Q_i = X_i W_Q, \qquad K_i = X_i W_K, \qquad V_i = X_i W_V.$$

The matrices $W_\bullet$ are *shared parameters* (same for every token); the vectors Q_i, K_i, V_i vary with the token and the layer. Multi-head attention duplicates these projections for each head so different heads learn different kinds of relevance (syntax, coreference, locality, copying, etc.).

2.6 Numeric example (tiny, hand-computable)

The following tiny example demonstrates the full flow (Q/K/V $\rightarrow$ *scores* $\rightarrow$ *softmax* $\rightarrow$ *attentionvector* $\rightarrow$ *context* $\rightarrow$ *prediction*).*Allnumbersaresmalltoallowhand* $-$ *calculation.*

Setup (toy): Tokens (context): $x_1 =$ "The", $x_2 =$ "cat", $x_3 =$ "sat"; we want to predict x_4. Embeddings (toy 2D):

$$e(\text{The}) = [1, 0], \quad e(\text{cat}) = [0, 1], \quad e(\text{sat}) = [1, 1].$$

Learned projection matrices (toy 2x2):

$$W_Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad W_K = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad W_V = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

Compute Q,K,V (per token): For x_1 ("The"):

$$Q_1 = [1, 0],\ K_1 = [1, 0],\ V_1 = [1, 1].$$

For x_2 ("cat"):

$$Q_2 = [0, 1],\ K_2 = [1, 1],\ V_2 = [0, 1].$$

For x_3 ("sat"):

$$Q_3 = [1, 1],\ K_3 = [2, 1],\ V_3 = [1, 2].$$

Scores (dot products) using Q_3 against all keys:

$$s_1 = Q_3 \cdot K_1 = 1, \quad s_2 = Q_3 \cdot K_2 = 2, \quad s_3 = Q_3 \cdot K_3 = 3.$$

Softmax (unnormalized exps):

$$\exp(1) \approx 2.718, \quad \exp(2) \approx 7.389, \quad \exp(3) \approx 20.085.$$

Sum ≈ 30.192. Hence attention weights:

$$\alpha_1 \approx 0.09, \quad \alpha_2 \approx 0.24, \quad \alpha_3 \approx 0.66.$$

Attention vector (context) for position 3:

$$\text{attn}_3 = 0.09 \cdot V_1 + 0.24 \cdot V_2 + 0.66 \cdot V_3 = [0.75,\ 1.65].$$

This context vector is then fed to the remaining layers/FFN and finally projected to logits u and softmaxed to obtain a distribution over the vocabulary; the most probable token is selected (or sampled).

Diagram (layer-level flow)

3 Section 3 — Scaling Efficiency

3.1 Quadratic cost of attention

Self-attention computes pairwise similarities between all token pairs in a window of length N. The cost and memory for the attention score matrix scale as $O(N^2)$.

Implication: doubling sequence length roughly quadruples attention compute and memory — motivating efficient attention variants and practical limits on context length.

3.2 Sources of inefficiency at inference

1. **Repeated linear projections:** at every generated step, naive code recomputes $K = XW_K$ and $V = XW_V$ for the whole context.

2. **Unnecessary matmuls for past tokens:** many matmuls depend on previously computed results that do not change.

Why attention is $O(N^2)$ (from first principles)

Self-attention requires computing $Q_i \cdot K_j$ for every pair (i, j) where $i, j \in \{1, \ldots, N\}$. This produces an $N \times N$ matrix of scores. Each of the N rows must be normalized with softmax and used to weight N value vectors, hence both compute and memory scale as $\Theta(N^2)$. Doubling N multiplies the number of pairwise comparisons by four, so practical context length is limited by available compute and memory unless attention is approximated or sparsified.

3.3 High-level strategies to scale

- **KV caching**: store previously computed keys/values and reuse them (Section 4).
- **Sparse or local attention**: restrict attention to nearby tokens or blocks.
- **Efficient kernels (FlashAttention)**: algorithmic and memory optimizations to compute attention with lower memory bandwidth.
- **Long-range transformers**: design attention approximations that are sub-quadratic (e.g., Performer, Reformer, BigBird).

4 Section 4 — KV cache (Key–Value cache) — theory & practice

4.1 Overview and intention

This section is a ground-up, exhaustive explanation of the Key–Value (KV) cache used for autoregressive inference. The goal is to answer every practical question you might have while reading: what is cached, why, how to store it, shapes and data layout, numerical and memory trade-offs, batching and streaming, interactions with positional encodings, FP precision gotchas, implementation patterns (preallocation vs concat), and corner-cases (relative/rotary/ALiBi, multi-GPU, variable-length streams).

4.2 What is being cached (precisely)

At each transformer layer ℓ and attention head h, the model projects the layer's input representations $X^{(\ell)} \in \mathbb{R}^{N \times d}$ into queries, keys and values via learned matrices:

$$Q^{(\ell)} = X^{(\ell)} W_Q^{(\ell)}, \qquad K^{(\ell)} = X^{(\ell)} W_K^{(\ell)}, \qquad V^{(\ell)} = X^{(\ell)} W_V^{(\ell)}.$$

During autoregressive inference the queries for the new token(s) must be compared to the *keys* of all past tokens, and attention computes weighted sums of the corresponding *values*. Since keys and values for past tokens do not change, we store them as

$$K_{\text{cache}}^{(\ell,h)} \in \mathbb{R}^{T \times d_{\text{head}}}, \qquad V_{\text{cache}}^{(\ell,h)} \in \mathbb{R}^{T \times d_{\text{head}}},$$

where T is the number of cached tokens (current context length) and d_{head} is the per-head dimension. Important: a separate cache exists *per layer* and *per head* (or equivalently the cache tensor includes a head dimension).

4.3 Why caching gives a big speedup

- **Naive recomputation:** to compute attention at step t the model would recompute K and V for positions $1..t$ and then compute $Q_t K_{1..t}^{top}$. Recomputing K, V from scratch for all past tokens at every new step is wasted work: those linear projections depend only on the stored weights and past hidden states which are unchanged.

- **Cached mode:** compute K_t, V_t once when token t is produced, append to cache. Future steps reuse cached keys and values and only compute new Q, K, V for newly added tokens. This reduces repeated matmuls and changes overall inference complexity from $O(N^2)$ project+attend per token to $O(N^2)$ only in the attention matmul part (and often to nearly linear per step when using optimized kernels + cache).

4.4 Where caches live (memory) and shapes

A practical, compact layout used by many implementations is a single preallocated tensor with shape

$$(\text{layers},\ \text{batch},\ \text{heads},\ \text{seq}_l en \text{seq}_l en_{\text{seq}_l en,\ d_{\text{head}})}$$

Common variants reorder dimensions for computational convenience (e.g. heads before batch). Here:

- `layers` = number of transformer layers L.
- `batch` = number of independent streams in the batch (often 1 for single-generation stream, but production systems batch many queries).
- `heads` = number of attention heads H.
- `seq_len` = maximum cached sequence length (capacity) or current token count T.
- d_{head} = per-head dimension (d/H if total model dim is d).

Preallocation vs dynamic append

- **Preallocation (recommended):** allocate a buffer for the maximum expected context length (e.g., 8k, 16k) and write new keys/values into slices of this buffer as tokens arrive. This avoids repeated allocations and supports fast matmuls because memory stays contiguous.
- **Dynamic concatenate:** keep a small cache and use concatenation (`torch.cat`) to append new K/V. Simpler to implement but can cause memory fragmentation and cost linear copies when concatenating repeatedly.

4.5 Pseudocode: preallocated cache write and read

```
# shapes and dims: B=batch, H=heads, D=d_head, S_new=seq_new (often 1)
# cache_k: (L, B, H, S_cap, D) preallocated; cursor[L,B] points to current length
# x_new: (B, S_new, d_model)  -- newly produced token(s) embeddings to project
q_new, k_new, v_new = project_qkv(x_new)  # -> shapes: (L, B, H, S_new, D)
# For each layer l: write into cache
cache_k[l, :, :, cursor[l]:cursor[l]+S_new, :] = k_new[l]
cache_v[l, :, :, cursor[l]:cursor[l]+S_new, :] = v_new[l]
cursor[l] += S_new
# During attention: use q_new with cache_k/l to compute scores
scores = einsum('B H S_q D, B H S_k D -> B H S_q S_k', q_new[l], cache_k[l, :, :, :cursor[l]
# proceed with softmax, masking, and value-weighted sum with cache_v
```

4.6 How attention uses the cache at a new step

Given cached keys/values for positions $1..T$, and a newly computed query Q_{T+1}, the attention computation is:

$$S_{T+1,1..T} = \frac{Q_{T+1}K_{1..T}^{top}}{\sqrt{d_k}}, \qquad \alpha = \text{softmax}(S + M), \qquad \text{attn} = \sum_{j=1}^{T} \alpha_j V_j.$$

Only $Q_{T+1}, K_{T+1}, V_{T+1}$ need to be computed at the time token $T+1$ is formed; earlier K, V are read from the cache. Practically this becomes an efficient batched matmul between Q (small) and K_{cache} (long). Note that in multi-head attention this happens per head (or using fused matmuls across heads).

4.7 Complexity analysis

- **Without cache (naive):** at each new token you recompute K, V for all past tokens and perform full attention: leads to repeated $O(Td^2)$ projections and $O(T^2d)$ attention work per token.
- **With cache (preallocated):** you compute K_{T+1}, V_{T+1} (cost $O(d^2)$) and perform $Q_{T+1}K_{1..T}^{top}$ (cost $O(Td)$ per head). Overall per-token cost is linear in T for the attention matmul but avoids repeating the projections. Over the whole generation of length N the total cost becomes $O(N^2)$ for attention matmuls plus $O(Nd^2)$ for projections, compared to a larger constant factor without caching.

4.8 Batching multiple streams / variable lengths

Production systems pack many independent generation streams into one batch to utilize GPUs. Each stream has its own cache cursor and mask. Important points:

- Preallocate caches for the largest stream length; per-stream cursor tracks how many tokens have been generated.
- When some streams end early (shorter outputs), maintain per-stream masks so that attention only considers valid keys for each stream.
- For efficiency you may group streams by length and process groups with similar cursor positions to avoid sparse matmuls.

4.9 Masks and padding when using cache

When combining cached K/V of different streams into a single matmul, use attention masks (or set invalid key logits to a large negative value) to ensure each query only attends to its own past. Example: if stream A has length 100 and stream B length 50, the combined cache for the batch may allocate to 100, but you must mask the upper 50 positions for stream B.

4.10 Positional encodings and cache interactions

KV caching interacts with positional encodings; you must ensure the model still sees correct positions for newly appended tokens.

- **Absolute learned positional embeddings:** typically you add a learned positional embedding to token embeddings at input. When caching, the stored K, V must have been computed with the original positional embedding for that token index. If you preallocate a cache buffer, ensure the positional offset you used when computing K, V matches the eventual absolute position.

- **Relative positional encodings (e.g. T5-style):** these are usually implemented inside attention score computation; caching needs to provide the correct relative offsets when computing S_{ij}. Often relative encodings are a function of $i - j$ and so can be computed on the fly for the current query and cached keys.

- **Rotary embeddings (RoPE):** rotary embeddings modify Q and K by rotating their components based on position index. When caching with rotary, many implementations store the rotated K values already (i.e., K_j saved already multiplied by the rotation for position j) so that at query time you only need to apply rotary to Q_{T+1} and multiply by cached K. If your implementation stores unrotated keys, you must reapply rotations at read time which is expensive; therefore storing rotated keys is common.

- **ALiBi and other relative biases:** ALiBi adds a bias matrix to attention logits based on distance; since it is a deterministic function of positions, you can compute or add it at score time without changing K/V storage.

4.11 FP precision and numerical issues

- **Use of -Inf vs large negative numbers:** when masking logits, use a large negative constant (e.g., -1e9) instead of IEEE -Inf to avoid NaN propagation under fp16.

- **Accumulation precision:** summing many terms (e.g., long contexts) may benefit from higher internal accumulation precision (fp32 accumulators) even when storing K/V in fp16. Libraries like cuBLAS and FlashAttention provide mixed-precision kernels that accumulate in fp32 for stability.

- **Softmax stability with long sequences:** for very long keys, the softmax's exponentials can under/overflow; stable implementations subtract the row max before exponentiating (this is standard) and block-wise softmax (FlashAttention) handles this safely.

4.12 Layer-wise cache: why each layer needs its own cache

Each transformer layer transforms hidden states; keys/values in layer ℓ are computed from the layer ℓ inputs. The caches cannot be shared across layers because $K^{(\ell)}$ depends on $X^{(\ell)}$, which is different per layer. Thus you need per-layer K/V caches. Typically the cache tensor includes the layer dimension so reads/writes are straightforward.

4.13 When NOT to use a KV cache

- **Training / backprop through time:** during training you cannot reuse cached K/V between updates because gradients require the forward activations to be available and parameter updates change weights; therefore KV caching across training steps is generally not used. Some training-time optimizations can reuse parts of computation within a batch but that's different.

- **Bidirectional / encoder models:** encoder-only or encoder-decoder setups used for masked LM or sequence-to-sequence may not benefit from the same streaming cache pattern.

4.14 Practical implementation patterns and optimizations

1. Preallocate + cursor

Preallocate maximum capacity and maintain a per-layer, per-batch cursor; write new K/V into slices. This is memory efficient and avoids copies.

2. Pack queries into a batch

When generating multiple tokens at once (e.g., batched generation of 8 new tokens), compute Q for all new positions and perform one batched matmul against the cache K. This reduces kernel launch overhead.

3. Fuse heads when possible

Many frameworks reshape (B,H,S,D) to (B, S, H*D) and use fused matmuls to leverage larger GEMMs for performance. The cache layout should be compatible with such fusions.

4. Use efficient attention kernels (FlashAttention)

FlashAttention computes softmax and attention in a memory-efficient blockwise manner. When used with caching, it can still be applied for the per-step QK^{top} and QV matmuls — but careful handling is needed if you want to compute attention over a cached long sequence in blocks to fit into on-chip memory.

5. Sparse / windowed cache strategies

For extremely long contexts it may be preferable to maintain only a recent sliding window of K/V (local attention) or a compressed representation (summaries) and selectively attend to sparse tokens (e.g., memory tokens). This reduces memory at the cost of losing some long-range exact attention.

4.15 Multi-GPU and distributed inference

- **Sharded model weights and caches:** when model weights are sharded across GPUs (pipeline or tensor parallelism), caches are correspondingly sharded: each device stores K/V for the part of the model it owns.
- **Pipeline parallelism:** in pipeline-parallel setups caches for layers assigned to a GPU are kept there; inter-stage communication passes only the new activations and not the entire cache.
- **Data parallel inference:** if several GPUs each serve independent batches, each keeps its own cache; when serving many concurrent requests, design an efficient cache eviction policy if VRAM is limited.

4.16 Cache eviction and memory management

When context length grows beyond capacity, common strategies:

- **Truncate oldest tokens:** drop earliest keys/values (FIFO) to keep the most recent context.
- **Hierarchical memory:** compress older context into a smaller set of summary vectors (learned or computed) and store them; attend to both summaries and recent tokens.
- **Spill to CPU/SSD:** move older K/V to host memory or disk and stream them back on-demand; this trades latency for capacity.

4.17 Examples: typical cache sizes and VRAM impact

Recomputing the earlier estimate in practical terms: for a model with $d = 2048$, $H = 32$, $d_{\text{head}} = 64$, $L = 24$, and $T = 8192$ tokens, storing float32 K/V requires about 4 GB as shown

earlier. Using float16 halves memory. Many inference stacks store caches in fp16 (or bf16) and use fp32 accumulation during matmuls for stability.

4.18 Debugging tips

- Verify shapes: mismatched head/seq dims are the most frequent bug.
- Check positional offsets: ensure keys were computed with the same positional encoding used at read-time.
- Masking: if the model attends to tokens from other streams, verify per-stream masks are applied correctly.
- Numerical: if outputs diverge between cached and non-cached runs, compare intermediate Q/K values and ensure rotary/relative transforms are consistent.

4.19 Minimal end-to-end example (PyTorch-style, conceptual)

```
# assume preallocated cache_k, cache_v: shape (L,B,H,cap,D), cursor per batch
for step in range(steps):
    x_new = tokenizer_step(...)  # produce new token ids or embeddings
    # compute layer-wise q,k,v for x_new through the transformer
    q_new, k_new, v_new = model.project_qkv_per_layer(x_new)
    for l in range(L):
        # write into cache
        s = cursor[l]
        cache_k[l,:, :, s:s+1, :] = k_new[l]
        cache_v[l,:, :, s:s+1, :] = v_new[l]
        cursor[l] += 1
        # compute attention using q_new[l] and cache_k[l][:,:,:cursor[l],:]
        scores = einsum('B H S_q D, B H S_k D -> B H S_q S_k', q_new[l], cache_k[l,:,:,:curs
        scores = scores / sqrt(d_head)
        scores = scores + mask  # mask out future or other streams
        attn = softmax(scores, dim=-1)
        out = einsum('... S_q S_k, ... S_k D -> ... S_q D', attn, cache_v[l,:,:,:cursor[l],:
        # continue with residuals and FFN to next layer
    # after final layer project to logits and sample next token
```

4.20 Final practical checklist (what to implement and test)

1. Preallocate per-layer, per-head cache in the desired precision (fp16/bf16), with a safe capacity.
2. Maintain per-stream cursors and masks; test single-stream correctness first.
3. Implement write-path that stores rotated/positioned keys if using rotary encodings.
4. Use fused matmuls and pack queries to reduce kernel overhead.
5. Test cached inference vs full-context inference on short sequences to verify numerical equivalence.
6. Monitor VRAM and implement truncation or spilling policy for long sessions.

Summary checklist (KV cache — deep)

- The KV cache stores per-layer, per-head keys and values for past tokens; it is read during attention and appended to when new tokens are produced.
- Use preallocated buffers + cursors for best performance; avoid repeated concatenation.
- Handle positional encodings carefully (rotary often implies storing rotated keys).
- Masks are essential when batching streams with differing lengths.
- Cache reduces redundant computation and yields large inference speedups at the cost of additional memory.
- When VRAM is limited, consider truncation, hierarchical memory, or spilling.

If you want, I can (A) expand this to include Section 5–7 (instruction tuning / preference learning), (B) produce a printable PDF layout optimized for lecture handouts, or (C) attach runnable PyTorch example notebooks implementing KV caching and a tiny transformer. Which do you prefer?