Mathematical Note on Informer-Style Transformer with ProbSparse Self-Attention and Distillation

Quan Nguyen

Contents

1 Problem Setup and Notation

Let $(x_t)_{t\in\mathbb{Z}}$ denote a multivariate time series with

$$x_t \in \mathbb{R}^{d_{\text{in}}}, \qquad t = 1, 2, \ldots, T.$$

Our goal is to learn a forecasting model that predicts the next L_p steps of a chosen target coordinate $a \in \{1, \ldots, d_{\text{in}}\}$ using the past L_e observations and any known future covariates.

Formally, we aim to approximate

$$\hat{y}_{t+1:t+L_p} = f_\theta\Big(X_{t-L_e+1:t},\, \tilde{X}_{t-L_g+1:t+L_p}\Big) \approx (x^{(a)}_{t+1}, \ldots, x^{(a)}_{t+L_p}),$$

where:

- $X_{t-L_e+1:t} \in \mathbb{R}^{L_e\times d_{\text{in}}}$ is the encoder input sequence,
- $\tilde{X}_{t-L_g+1:t+L_p} \in \mathbb{R}^{(L_g+L_p)\times d_{\text{in}}}$ is the decoder input, obtained by concatenating:

  (i) the overlap context $X_{t-L_g+1:t}$,
  (ii) the masked future segment $\text{MaskFuture}(X_{t+1:t+L_p}; a)$,

  where $\text{MaskFuture}(\cdot; a)$ zeroes out the target coordinate $x^{(a)}$ while keeping all other known future features unchanged.

Encoder–decoder structure. The model comprises:

- **Encoder window (L_e):** the number of past observations used to encode historical information up to time t.
- **Guiding window (L_g):** an overlap region shared between encoder and decoder, corresponding to the most recent L_g steps of the encoder output. This overlap provides temporal continuity at the prediction boundary and stabilises decoder dynamics by allowing attention to the latest context.
- **Prediction horizon (L_p):** the number of future steps to forecast for the target variable.

Feature space and per-layer lengths. All hidden representations lie in a shared latent space $\mathbb{R}^{d_{\text{model}}}$. Multi-head attention uses n_{heads} parallel heads, each of per-head dimension

$$d_k = \frac{d_{\text{model}}}{n_{\text{heads}}}.$$

We denote by L_l the sequence length *entering* encoder layer l; with distillation enabled, $L_l \approx L_{l-1}/2$ (otherwise $L_l = L_0 = L_e$ for all l). We write L'_e for the encoder output length after the entire stack.

2 Embedding and Temporal Encoding

2.1 Content Embedding

Each input vector $x_t \in \mathbb{R}^{d_{\text{in}}}$ is projected to the model space by

$$E(x_t) = x_t W_E + b_E, \quad W_E \in \mathbb{R}^{d_{\text{in}}\times d_{\text{model}}},\ b_E \in \mathbb{R}^{d_{\text{model}}}.$$

For a batch of size B and sequence length L, this yields

$$E(X) \in \mathbb{R}^{B\times L\times d_{\text{model}}}.$$

2.2 Positional Encoding

Self-attention is inherently permutation-invariant: it treats its input sequence as a set rather than an ordered list. To enable the model to exploit the sequential structure of the data, we inject information about token positions by adding a positional encoding to the token embeddings before attention layers.

We adopt the deterministic sinusoidal positional encoding, defined as

$$\mathrm{PE}(p, 2i) = \sin\Big(\frac{p}{10000^{2i/d_{\text{model}}}}\Big), \qquad \mathrm{PE}(p, 2i+1) = \cos\Big(\frac{p}{10000^{2i/d_{\text{model}}}}\Big),$$

for positions $p = 0, \dots, L_{\max} - 1$ and dimensions $i = 0, \dots, d_{\text{model}}/2 - 1$. The encoded input to the transformer is then

$$X_{\text{pos}} = E(X) + \mathrm{PE}[:, :L, :], \qquad X_{\text{pos}} \in \mathbb{R}^{B \times L \times d_{\text{model}}}.$$

The sinusoidal design has several appealing properties:

- **Deterministic and parameter-free:** no extra parameters, aiding stability and extrapolation to longer sequences.
- **Multi-scale representation of position:** exponentially spaced frequencies allow both long- and short-wavelength components; relative offsets are linearly recoverable.
- **Smooth and continuous:** nearby positions produce similar vectors, providing a continuous notion of locality.

Comparison with learned positional embeddings. Learned positional embeddings can capture task-specific patterns but may extrapolate poorly to unseen lengths. The sinusoidal scheme offers a fixed, continuous function of position that generalises naturally and is particularly effective in sequence modelling and time-series settings.

2.3 Learnable Time Embedding (Optional)

Many time series contain strong calendar-based patterns such as hourly, daily, or monthly seasonality. To incorporate such temporal information, we introduce learnable embeddings for discrete time components.

Discrete time features. For each timestamp t:

$$h_t \in \{0, 1, \dots, 23\}, \qquad w_t \in \{0, 1, \dots, 6\}, \qquad m_t \in \{1, 2, \dots, 12\},$$

representing *hour of day*, *day of week*, and *month of year*.

Each component is passed through a separate embedding table:

$$e_h = \mathrm{Embed}_{24}(h_t), \qquad e_w = \mathrm{Embed}_7(w_t), \qquad e_m = \mathrm{Embed}_{12}(m_t),$$

where $e_h, e_w, e_m \in \mathbb{R}^{d_t}$ and d_t (e.g. 8 or 16) is the time-embedding dimension.

Concatenation and projection. Concatenate and project into model space:

$$e_{\text{time}} = \mathrm{concat}(e_h, e_w, e_m) W_1 + b_1, \qquad W_1 \in \mathbb{R}^{3d_t \times d_{\text{model}}},\ b_1 \in \mathbb{R}^{d_{\text{model}}}.$$

Thus, $e_{\text{time}} \in \mathbb{R}^{d_{\text{model}}}$.

Nonlinear refinement. A lightweight feed-forward transform:

$$e'_{\text{time}} = \text{ReLU}(e_{\text{time}})W_2 + b_2, \qquad W_2 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}},\ b_2 \in \mathbb{R}^{d_{\text{model}}}.$$

Integration with positional encoding. Combine with positionally encoded input:

$$Z = \text{LayerNorm}\big(X_{\text{pos}} + e'_{\text{time}}\big),$$

with $X_{\text{pos}} \in \mathbb{R}^{B \times L \times d_{\text{model}}}$ broadcast with e'_{time} across time.

3 Multi-Head Attention

Given $Q, K, V \in \mathbb{R}^{B \times L \times d_{\text{model}}}$, attention uses n_{heads} projections:

$$Q_h = QW_h^Q, \quad K_h = KW_h^K, \quad V_h = VW_h^V, \tag{1}$$

$$W_h^{Q,K,V} \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad d_k = d_{\text{model}}/n_{\text{heads}}. \tag{2}$$

After reshaping, $Q_h, K_h, V_h \in \mathbb{R}^{B \times n_{\text{heads}} \times L \times d_k}$.

Let L_Q and L_K denote query and key sequence lengths (typically $L_Q = L_K = L$). Let $M \in \{0,1\}^{L_Q \times L_K}$ be an optional attention mask, where $M_{ij} = 0$ indicates a disallowed (masked) key for query position i. The scaled dot-product attention per head is

$$A_h(Q, K, V; M) = \text{Softmax}\left(\frac{Q_h K_h^\top}{\sqrt{d_k}} + \Xi\right) V_h,$$

where $\Xi_{ij} = 0$ if $M_{ij} = 1$ and $\Xi_{ij} = -\infty$ if $M_{ij} = 0$ (i.e. masked positions contribute zero probability after softmax). Outputs from all heads are concatenated and projected:

$$\text{MultiHead}(Q, K, V; M) = \big[A_1; \ldots; A_{n_{\text{heads}}}\big] W^O, \quad W^O \in \mathbb{R}^{(n_{\text{heads}} d_k) \times d_{\text{model}}}.$$

Implementation note. In code, we implement this operation using PyTorch's fused routine `torch.nn.functional.scaled_dot_product_attention`, which internally supports dropout and causal masking and automatically dispatches to FlashAttention kernels when available.

4 ProbSparse Self-Attention

Standard scaled dot-product attention requires $O(L^2)$ computation and memory, where L is the sequence length. The *ProbSparse* mechanism reduces this to $O(L \log L)$ by computing attention only for a subset of *dominant* queries—those with sharply peaked attention distributions.

4.1 Query Sampling and Sparsity Measure

Let

$$Q, K, V \in \mathbb{R}^{B \times H \times L \times d_k},$$

where B is the batch size, H the number of heads, L the sequence length (for both queries and keys in self-attention), and d_k the per-head dimension.

To estimate which queries are most informative, sample a subset of key indices $\Pi \subset \{1, \ldots, L\}$ of size

$$k = \lfloor \alpha \log L \rfloor,$$

where $\alpha > 0$ is a sparsity factor. In code, this corresponds to `perm = torch.randperm(L_K)[:sample_k]` and

$$K_\Pi = K[:, :, \Pi, :] \in \mathbb{R}^{B\times H\times k\times d_k}.$$

Each query Q_i interacts only with the sampled keys:

$$S_{i\Pi} = \frac{Q_i K_\Pi^\top}{\sqrt{d_k}} \in \mathbb{R}^{B\times H\times L\times k}.$$

The resulting tensor $S^{(\text{samp})}$ has shape $[B, H, L, k]$ and stores approximate attention scores between all queries and k sampled keys.

The sparsity of each query is measured by

$$s_i = \log\sum_{j\in\Pi} \exp\Big(S_{ij}^{(\text{samp})}\Big) - \frac{1}{k}\sum_{j\in\Pi} S_{ij}^{(\text{samp})},$$

producing

$$s \in \mathbb{R}^{B\times H\times L}.$$

Large s_i indicates a high-variance, peaked attention pattern.

The top-u dominant queries are selected as

$$u = \lfloor \alpha \log L \rfloor, \qquad \mathcal{I} = \mathrm{Top}_u(s_1, \ldots, s_L).$$

In implementation, `M_top = torch.topk(M, n_top, dim=-1)[1]` with

$$M_{\text{top}} \in \mathbb{R}^{B\times H\times u}.$$

4.2 Sparse Computation

The output context tensor $C \in \mathbb{R}^{B\times H\times L\times d_k}$ is constructed in three stages:

(a) **Initialization.** All queries start with a coarse global mean of values:

$$C^{(0)} = \frac{1}{L}\sum_{j=1}^{L} V_j \in \mathbb{R}^{B\times H\times 1\times d_k}, \quad C^{(0)}_{\text{expanded}} = \mathrm{repeat}(C^{(0)}, L) \in \mathbb{R}^{B\times H\times L\times d_k}.$$

(b) **Selective attention.** For the dominant queries indexed by $\mathcal{I}$, gather their query vectors:

$$Q_\mathcal{I} = Q[:, :, \mathcal{I}, :] \in \mathbb{R}^{B\times H\times u\times d_k}.$$

Compute full attention scores against all keys:

$$A = \frac{Q_\mathcal{I} K^\top}{\sqrt{d_k}} \in \mathbb{R}^{B\times H\times u\times L}, \qquad W = \mathrm{Softmax}(A, \mathrm{dim} = -1),$$

and weighted values

$$C_\mathcal{I} = WV \in \mathbb{R}^{B\times H\times u\times d_k}.$$

(c) **Aggregation.** The refined outputs $C_\mathcal{I}$ replace the corresponding rows in $C^{(0)}$:

$$C[:, :, \mathcal{I}, :] \leftarrow C_\mathcal{I}.$$

In code this corresponds to an in-place scatter: `context.scatter_(2, M_top_expanded, context_top)`. Finally, the multi-head tensor is reshaped to

$$\mathrm{reshape}\big(C,\ [B, L, Hd_k]\big)$$

and linearly projected as in standard multi-head attention.

4.3 Complexity and Memory Analysis

Per head, the cost is

$$O(Lkd_k + uLd_k) = O(L \log L \, d_k),$$

compared to $O(L^2 d_k)$ for dense attention. Memory reduces from $O(L^2)$ to $O(L(k+u)) = O(L \log L)$.

Why It Works. In many time series, attention maps are sparse: only a few queries focus sharply on key historical positions. For nearly uniform queries, replacing their context by the mean introduces negligible error, while dominant queries are recovered with probability $1 - O(1/L)$ under random key sampling.

Limitations. If the true attention is dense, the mean-context approximation is poor. Larger α (increasing k and u) improves accuracy at the cost of higher computation.

5 Encoder Layer with Distillation

Each encoder layer l receives a hidden representation

$$H^{(l-1)} \in \mathbb{R}^{B \times L_{l-1} \times d_{\text{model}}},$$

and outputs $H^{(l)} \in \mathbb{R}^{B \times L_l \times d_{\text{model}}}$. With distillation disabled, $L_l = L_0 = L_e$ for all l. With distillation enabled, $L_l \approx L_{l-1}/2$ (pooling defined below).

5.1 Computation (Pre-Norm Form)

The encoder layer follows the standard pre-normalised Transformer structure:

$$\begin{aligned}
Z_1 &= \text{LN}(H^{(l-1)}), & (3)\\
H' &= H^{(l-1)} + \text{Drop}\big(\text{PSA}(Z_1, Z_1, Z_1)\big), & (4)\\
Z_2 &= \text{LN}(H'), & (5)\\
H^{(l)} &= H' + \text{Drop}\big(\text{FFN}(Z_2)\big), & (6)
\end{aligned}$$

where PSA$(\cdot)$ is **ProbSparse self-attention** (Section 4; no mask in the encoder), and FFN is position-wise:

$$\text{FFN}(x) = \text{GELU}(xW_1 + b_1)W_2 + b_2, \qquad W_1 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}, \quad W_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}.$$

Residual connections and layer normalisation stabilise gradient flow and preserve feature scale.

5.2 Convolutional Distillation Block

If the encoder stack is configured with `distill=True`, a convolution–pooling module is inserted after each encoder layer (except the last). This block performs temporal compression while retaining channel dimensionality:

$$H^{(l)} \longmapsto H^{(l+1)}_{\text{pool}}.$$

It consists of two operations: a 1D convolution for local temporal feature extraction, and a MaxPool1D operator for nonlinear downsampling.

5.2.1 Conv1D Layer: Local Temporal Feature Extraction

Given $H^{(l)} \in \mathbb{R}^{B\times L\times d_{\text{model}}}$, transpose to channel-first:

$$Z = \text{Transpose}(H^{(l)}, (0,2,1)) \in \mathbb{R}^{B\times C\times L}, \quad C = d_{\text{model}}.$$

Apply a 1D convolution with kernel $k_c = 3$, stride $s_c = 1$, padding $p_c = 1$:

$$U = \text{Conv1D}(Z; k_c, s_c, p_c) = Z * W_{\text{conv}} + b_{\text{conv}},$$

with

$$W_{\text{conv}} \in \mathbb{R}^{C\times C\times k_c}, \qquad b_{\text{conv}} \in \mathbb{R}^{C}.$$

Output length. For input length L,

$$L_{\text{conv}} = \left\lfloor \frac{L + 2p_c - (k_c - 1) - 1}{s_c} \right\rfloor + 1 = L,$$

so the convolution preserves length.

Index-level formula. Let $Z_{\text{pad}} \in \mathbb{R}^{B\times C\times(L+2p_c)}$ be the zero-padded input. For batch b, channel c, temporal index t:

$$U[b,c,t] = \sum_{j=0}^{k_c-1} \sum_{c'=1}^{C} Z_{\text{pad}}[b,c',t+j]\, W_{\text{conv}}[c,c',j] + b_{\text{conv}}[c].$$

Activation. Apply a pointwise nonlinearity (e.g. ReLU or ELU):

$$\tilde{U} = \phi(U), \quad \phi(x) = \max(0,x) \text{ or } \phi(x) = \begin{cases} x, & x > 0, \\ e^x - 1, & x \le 0. \end{cases}$$

5.2.2 MaxPool1D Layer: Nonlinear Temporal Downsampling

Input and parameters.

$$\tilde{U} \in \mathbb{R}^{B\times C\times L_{\text{conv}}}, \quad V = \text{MaxPool1D}(\tilde{U}; k_p = 3, s_p = 2, p_p = 1) \in \mathbb{R}^{B\times C\times L_{\text{out}}}.$$

Output length.

$$L_{\text{out}} = \left\lfloor \frac{L_{\text{conv}} + 2p_p - (k_p - 1) - 1}{s_p} \right\rfloor + 1 = \left\lceil \frac{L}{2} \right\rceil.$$

Index-level definition. Let $\tilde{U}_{\text{pad}} \in \mathbb{R}^{B\times C\times(L_{\text{conv}}+2p_p)}$ be the padded input. For temporal index u:

$$W_u = \{s_p u, s_p u + 1, \ldots, s_p u + k_p - 1\}, \quad V[b,c,u] = \max_{j\in W_u} \tilde{U}_{\text{pad}}[b,c,j].$$

Gradient flow. During backpropagation, only the maximal element in each window receives gradient:

$$\frac{\partial V[b,c,u]}{\partial \tilde{U}[b,c,j]} = \begin{cases} 1, & j = \arg\max_{k\in W_u} \tilde{U}[b,c,k], \\ 0, & \text{otherwise.} \end{cases}$$

Interpretation.

- **Dominance-based selection:** keeps the most salient activation in each local window.
- **Translation robustness:** small timing shifts do not alter the pooled value.
- **Noise suppression:** weak activations vanish; dominant responses remain.
- **Information bottleneck:** nonlinear projection $P : \mathbb{R}^{B\times C\times L} \to \mathbb{R}^{B\times C\times L/2}$, compressing redundancy while preserving core activations.

5.2.3 Output and Hierarchical Representation

Transpose back to (B, L, C):

$$H^{(l+1)} = \mathrm{Transpose}(V, (0, 2, 1)) \in \mathbb{R}^{B\times L_{\mathrm{out}}\times d_{\mathrm{model}}}.$$

Thus, $L_l \approx L_{l-1}/2$ while d_{model} remains constant. The convolution–pooling pair forms a temporal pyramid:

$$H^{(0)} \xrightarrow[\text{Conv+Pool}]{} H^{(1)} \xrightarrow[\text{Conv+Pool}]{} H^{(2)} \xrightarrow[\text{Conv+Pool}]{} \cdots,$$

progressively shortening the sequence while enriching temporal abstraction.

Intuitive summary. Conv1D linearly *aggregates* local temporal context; MaxPool1D nonlinearly *selects* dominant activations; together, they perform hierarchical *temporal distillation*. This reduces compute in deeper layers and provides multi-scale receptive fields.

Learnable parameters. All convolutional parameters $\{W_{\mathrm{conv}}, b_{\mathrm{conv}}\}$ are trained jointly with attention and feed-forward weights. Pooling is parameter-free but participates in backpropagation.

6 Decoder Layer and Causal Masking

Each decoder layer receives

$$D^{(l-1)} \in \mathbb{R}^{B\times L_d\times d_{\mathrm{model}}},$$

and attends to the encoder output

$$E \in \mathbb{R}^{B\times L'_e\times d_{\mathrm{model}}}.$$

6.1 Masked Self-Attention

To ensure autoregressive generation, apply a lower-triangular mask:

$$M_{ij} = \mathbb{1}_{\{i\geq j\}}, \qquad M \in \{0,1\}^{L_d\times L_d}.$$

Here $M_{ij} = 1$ (visible) means position i can attend to position j, while $M_{ij} = 0$ (masked) prevents access to future positions $j > i$. The mask is broadcast across all heads.

For $L_d = 5$:

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}.$$

Implementation note. In practice, the causal mask is stored as a boolean tensor of shape (L_d, L_d) where `True` indicates a visible (unmasked) position and `False` indicates a masked one. Before the softmax operation, logits corresponding to `False` entries are assigned $-\infty$, so their attention weights vanish after normalization.

The decoder layer computation (pre-norm) is:

$$D_1 = D^{(l-1)} + \mathrm{Drop}\Big(\mathrm{PSA}\big(\mathrm{LN}(D^{(l-1)}), \mathrm{LN}(D^{(l-1)}), \mathrm{LN}(D^{(l-1)});\, M\big)\Big)\,, \tag{7}$$

$$D_2 = D_1 + \mathrm{Drop}\big(\mathrm{MHA}\big(\mathrm{LN}(D_1), E, E\big)\big)\,, \tag{8}$$

$$D^{(l)} = D_2 + \mathrm{Drop}\big(\mathrm{FFN}\big(\mathrm{LN}(D_2)\big)\big)\,. \tag{9}$$

Here, PSA denotes *masked* ProbSparse self-attention (Section 4) using the causal mask M, and MHA denotes standard multi-head *cross*-attention attending to the encoder output E (no causal mask between decoder and encoder).

Causal semantics. The mask enforces

$$p(y_{t+1:t+L_p} \mid x_{1:t}) = \prod_{h=1}^{L_p} p(y_{t+h} \mid y_{t+1:t+h-1},\, x_{1:t}).$$

Implementation-wise, masked logits are set to $-\infty$ so their softmax probabilities vanish.

7 End-to-End Flow

For batch size B:

1. Encoder input $X_{\text{enc}} \in \mathbb{R}^{B\times L_e\times d_{\text{in}}} \to$ embedding + positional/time encoding $\to H^{(0)} \in \mathbb{R}^{B\times L_e\times d_{\text{model}}}$.
2. Encoder stack (with optional distillation) $\to H^{(N_e)} \in \mathbb{R}^{B\times L'_e\times d_{\text{model}}}$.
3. Decoder input (context L_g + masked future L_p) $X_{\text{dec}} \in \mathbb{R}^{B\times(L_g+L_p)\times d_{\text{in}}} \to$ embedding + encoding $\to D^{(0)}$.
4. Decoder stack with causal self-attention and cross-attention $\to D^{(N_d)} \in \mathbb{R}^{B\times(L_g+L_p)\times d_{\text{model}}}$.
5. Output projection

   $$\hat{Y} = D^{(N_d)} W_{\text{proj}} + b_{\text{proj}}, \qquad W_{\text{proj}} \in \mathbb{R}^{d_{\text{model}}\times d_{\text{out}}}.$$

   For univariate forecasting of coordinate a, $d_{\text{out}} = 1$ and we take the last L_p steps as $\hat{y} \in \mathbb{R}^{B\times L_p}$.

8 Gradient Flow and Stability

For a pre-norm residual block

$$H_{\text{out}} = H_{\text{in}} + \mathcal{F}(\mathrm{LN}(H_{\text{in}})),$$

the backward gradient satisfies

$$\frac{\partial \mathcal{L}}{\partial H_{\text{in}}} = \frac{\partial \mathcal{L}}{\partial H_{\text{out}}}\Big(I + J_{\mathcal{F}} J_{\text{LN}}\Big),$$

where $J_{\mathcal{F}}$ and J_{LN} are the respective Jacobians. The identity path preserves gradient magnitude even when $J_{\mathcal{F}}$ is small. Since $\|J_{\text{LN}}\|_2$ is bounded, gradients remain stable across depth.

9 Loss, Optimisation, and Learning Schedule

9.1 Training Objective

The model is trained by minimising the mean-squared error between predicted and true target values:

$$\mathcal{L}(\theta) = \frac{1}{BL_p} \sum_{b=1}^{B} \sum_{h=1}^{L_p} \left(\hat{y}_{b,h} - y_{b,h}\right)^2,$$

where B is the batch size and L_p the prediction horizon. This objective penalises large forecast errors and corresponds to the negative log-likelihood under an isotropic Gaussian assumption.

9.2 Optimiser and Regularisation

We employ the AdamW optimiser with decoupled weight decay. Let θ_t denote model parameters at step t. The AdamW update rule is:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta \mathcal{L}_t, \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2)\big(\nabla_\theta \mathcal{L}_t\big)^2,$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t},$$
$$\theta_{t+1} = \theta_t - \eta_t \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} - \eta_t\, \lambda\, \theta_t,$$

where λ is the weight-decay coefficient and η_t is the learning rate. The second term, $-\eta_t \lambda \theta_t$, applies *decoupled* L_2 regularisation, preventing bias terms and LayerNorm parameters from being penalised. In implementation, parameters are divided into two groups: those with $\dim(\theta) \geq 2$ (matrix weights) receive weight decay λ, while 1D parameters such as biases and normalization gains use $\lambda = 0$. This selective regularisation matches the configuration:

$$\texttt{decay\_params:}\ \ \texttt{dim}(\theta) \geq 2, \qquad \texttt{nodecay\_params:}\ \ \texttt{dim}(\theta) < 2.$$

The AdamW coefficients follow transformer conventions: $\beta_1 = 0.9$, $\beta_2 = 0.95$, and $\epsilon = 10^{-8}$. When CUDA is available, the optimiser uses fused operations for efficiency.

Weight Decay Intuition. Decoupled weight decay discourages overfitting by shrinking large weights toward zero without coupling the penalty to the gradient magnitude. Unlike classical L_2 regularisation within the loss function, AdamW's decoupled formulation maintains consistent learning dynamics even under adaptive gradient scaling.

9.3 Gradient Clipping and Mixed Precision

To prevent exploding gradients, the global gradient norm is constrained by a fixed upper bound γ:

$$\nabla\theta \;\leftarrow\; \frac{\nabla\theta}{\max\!\Big(1,\ \frac{\|\nabla\theta\|_2}{\gamma}\Big)}.$$

If $\|\nabla\theta\|_2 > \gamma$, the entire gradient tensor is rescaled to have norm γ. In code, this is performed with `torch.nn.utils.clip_grad_norm_(model.parameters(), grad_clip)`. This stabilises updates for deep transformer stacks and ensures that outlier batches do not destabilise training.

Training is performed in mixed precision (AMP) when running on GPU, with gradient scaling via `torch.amp.GradScaler` to avoid underflow. Gradients are unscaled prior to clipping to ensure correct norm computation.

9.4 Learning-Rate Schedule

The learning rate $\eta(s)$ follows a two-phase schedule combining linear warm-up and cosine decay:

$$\eta(s) = \begin{cases} \eta_0 \, \dfrac{s}{S_w}, & s < S_w, \\ \eta_{\min} + (\eta_0 - \eta_{\min}) \dfrac{1 + \cos\left(\pi \, \frac{s - S_w}{S - S_w}\right)}{2}, & s \geq S_w, \end{cases}$$

where:

- η_0 is the initial (peak) learning rate,
- $\eta_{\min}$ is the minimum rate at the end of training,
- S_w is the number of warm-up steps, and
- S is the total training step budget.

During warm-up ($s < S_w$), the learning rate increases linearly from 0 to η_0 to prevent early divergence. After warm-up, cosine decay gradually anneals η toward $\eta_{\min}$ for smooth convergence.

9.5 Parameter Initialization

To ensure stable training and consistent gradient scales, all linear projection layers are initialized following a zero-mean normal distribution:

$$W_{ij} \sim \mathcal{N}(0,\, 0.02^2), \qquad b_i = 0.$$

Layer normalization parameters retain their defaults ($\gamma = 1,\ \beta = 0$). This *partial GPT-style initialization* controls the variance of forward activations and supports stable gradient propagation across attention and feed-forward blocks.

Implementation note. In the codebase, this scheme is encapsulated in the utility `init_weights(module, std=0.02)`, which applies normal initialization to all linear layers and zeroes their biases, leaving normalization layers unchanged.