JEKARDAH AI LABS

Jekardah.com

AI Engineering Series

2026 EDITION

AI Engineering Handbook

and Interview Preparation

Transformers & Vision Transformers (ViT)

Self-Attention Multi-Head Attention Encoder-Decoder Positional Encoding ViT Swin Transformer LoRA Flash Attention Scaling Laws RAG LLM Deployment DINO

Romi Nur Ismanto

rominur@gmail.com

150+ Pages | 20 Deep Dives

50 Interview Questions | Python Code

Complete Theory to Production Guide

TRANSFORMERS | SELF-ATTENTION | ViT | LLM | SCALING LAWS | LoRA | DEPLOYMENT

AI Engineering Handbook and Interview Preparation: Transformers & Vision Transformers (ViT)

Information	Details
Title	AI Engineering Handbook and Interview Preparation
Publisher	Jekardah AI Labs
Author	Romi Nur Ismanto
Email	rominur@gmail.com
Website	Jekardah.com
Edition	2026 Edition
Language	English
Category	Deep Learning, AI, Computer Vision, NLP
Coverage	12 Chapters + Appendix: 50 Interview Questions
License	Internal Use - Jekardah AI Labs

***About This Book:** This material is designed as a comprehensive guide to understanding the Transformer and Vision Transformer (ViT) architectures, from mathematical foundations to practical implementation. It includes 50 interview questions with Python code in the appendix.*

Table of Contents

CHAPTER 1

Introduction: The Transformer Era

1.1 Background & Motivation

The Transformer architecture, introduced in the landmark 2017 paper **"Attention Is All You Need"** by Vaswani et al., has fundamentally reshaped the deep learning landscape. Before Transformers, sequence-to-sequence models were dominated by Recurrent Neural Networks (RNNs) and their variants such as Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU).

Although RNNs and LSTMs achieved success across many NLP tasks, they suffer from fundamental limitations that constrain their scalability and performance:

- **Sequential Processing:** RNNs process tokens one at a time. The hidden state at step t depends on step t-1, creating a bottleneck that prevents parallelisation during training.
- **Vanishing Gradient Problem:** Information from distant tokens tends to be lost as gradients shrink exponentially with sequence length. LSTM mitigates this with gating mechanisms, but does not fully solve it.
- **Poor GPU Utilisation:** Due to the sequential nature, GPUs/TPUs cannot be leveraged optimally, resulting in slow training on long sequences.
- **Limited Scalability:** Training time scales linearly with sequence length, making training on long documents impractical.

Transformers solve all of these problems by replacing recurrence with **self-attention**, which allows every token to access information from all other tokens directly and in parallel.

Key Insight: *Self-attention provides an O(1) path length between any two tokens in a sequence, compared to O(n) in RNNs. This means information from the first token can directly influence the last token without signal degradation.*

1.2 From RNNs to Transformers: An Evolution

The evolution of sequence modelling architectures follows a fascinating trajectory. Each generation addresses the limitations of its predecessor:

Era	Architecture	Strengths	Limitations
2014	Seq2Seq + Attention (Bahdanau)	First attention mechanism; focuses on relevant parts	Still sequential; requires RNN encoder
2015	LSTM/GRU with Attention	Gate mechanisms; better long-range	Still O(n) path; slow training
2017	Transformer (Vaswani et al.)	Fully parallel; O(1) path length	O(n^2) memory; data hungry
2018+	BERT, GPT and derivatives	Pre-train + fine-tune; SOTA across NLP	Very large models; high compute cost
2020+	ViT, Swin, CLIP	Transformers for vision; multimodal capability	Data hungry; quadratic scaling

1.3 Modern Transformer Landscape

Today, the Transformer architecture underpins virtually all state-of-the-art models across domains. Here is the primary classification by architecture type:

Encoder-Only Models

Encoder-only models use bidirectional attention where every token can see all other tokens. Best suited for understanding tasks such as text classification, Named Entity Recognition (NER), and question answering. Key examples: **BERT**, **RoBERTa**, **ALBERT**, **DeBERTa**.

Decoder-Only Models

Decoder-only models use causal (masked) attention where each token can only see preceding tokens. Optimised for text generation. This is the architecture used by most modern Large Language Models. Examples: **GPT-2/3/4**, **LLaMA**, **Mistral**, **Claude**.

Encoder-Decoder Models

Encoder-decoder models combine bidirectional attention (encoder) with causal attention plus cross-attention (decoder). Ideal for sequence-to-sequence tasks such as translation and summarisation. Examples: **T5**, **BART**, **mBART**.

Vision Transformers

The application of the Transformer architecture to computer vision by converting images into sequences of patch embeddings. Examples: **ViT**, **Swin Transformer**, **DeiT**, **BEiT**.

CHAPTER 2

Mathematical Foundations

2.1 Linear Algebra for Attention

Understanding the attention mechanism requires mastery of several fundamental linear algebra concepts. Below are the key ideas you need to grasp:

Dot Product

The dot product between two vectors measures the similarity of their directions and magnitudes. In the attention context, the dot product between a query q and key k yields a score indicating how relevant that key is to the query:

```
q . k = sum(q_i * k_i) for i = 1 ... d_k
```

The higher the dot product, the more similar the two vectors are, and the larger the attention weight assigned.

Matrix Multiplication in Attention

All attention operations can be expressed as a series of matrix multiplications, enabling highly efficient parallel computation on GPUs. The matrices Q (queries), K (keys), and V (values) each have shape (n x d_k), where n is the sequence length and d_k the per-head dimension.

The operation QK^T produces an (n x n) score matrix where each element (i, j) represents how much token i should attend to token j.

Linear Projections

Before attention is computed, input embeddings are transformed through learned linear projections using weight matrices W_Q, W_K, W_V. These projections map the input into different subspaces, allowing the model to focus on different aspects of the representation:

```
Q = X * W_Q, K = X * W_K, V = X * W_V

where W_Q, W_K, W_V in R^(d_model x d_k)
```

2.2 Softmax & Probability Distributions

The softmax function is a critical component in the attention mechanism. It converts a vector of raw scores into a valid probability distribution where all values are non-negative and sum to 1:

```
softmax(z_i) = exp(z_i) / sum(exp(z_j)) for all j
```

In the attention context, softmax is applied to each row of the (n x n) score matrix, producing attention weights. Important properties of softmax:

- **Non-negativity:** All outputs >= 0, ensuring valid attention weights.
- **Normalisation:** Each row sums to 1, forming a probability distribution.
- **Differentiable:** Softmax is differentiable everywhere, supporting backpropagation.
- **Temperature sensitivity:** Input scale affects the sharpness of the output distribution.

Important Note: *Without the scaling factor 1/sqrt(d_k), dot products in high dimensions produce very large values, pushing softmax into saturation regions where gradients approach zero. This is why scaled dot-product attention is essential.*

2.3 Gradients & Optimisation

Training Transformers uses optimisation techniques tailored to the unique challenges of this architecture:

Adam Optimizer

Adam (Adaptive Moment Estimation) is the standard optimizer for Transformers. It combines momentum (moving average of gradients) with RMSprop (moving average of squared gradients) to produce adaptive per-parameter learning rates.

Learning Rate Warmup

Transformers are highly sensitive to the learning rate at the start of training. Warmup gradually increases the learning rate from zero to the target value over several thousand initial steps, then decays it. The original paper uses:

```
lr = d_model^(-0.5) * min(step^(-0.5), step *
warmup_steps^(-1.5))
```

Warmup prevents excessively large updates early in training when parameters are still random, which could cause irrecoverable divergence.

Gradient Clipping

To prevent exploding gradients, gradient clipping caps the total gradient norm at a threshold (typically 1.0). If the gradient norm exceeds the threshold, all gradients are scaled down proportionally.

CHAPTER 3

Transformer Architecture In-Depth

3.1 Core Components of a Transformer Block

Each Transformer encoder block consists of four core components that work together to process the input sequence. A deep understanding of each component is the key to grasping the entire architecture.

Input Embedding

The first step converts discrete tokens (words, subwords, or image patches) into continuous d_model-dimensional vectors. For NLP, this typically uses an embedding lookup table of size |V| x d_model, where |V| is the vocabulary size.

```
import torch.nn as nn

vocab_size, d_model = 32000, 512
embedding = nn.Embedding(vocab_size, d_model)

token_ids = torch.tensor([[1, 42, 789, 10, 5]])
embeddings = embedding(token_ids)  # (1, 5, 512)
embeddings = embeddings * (d_model ** 0.5)  # Scale per original paper
```

Multi-Head Self-Attention

The core component that allows every token to gather information from the entire sequence. Input embeddings are projected to Q, K, V via learned weight matrices, then attention is computed in parallel across heads. Full details in Chapters 4 and 5.

Feed-Forward Network (FFN)

After attention, each position is independently processed through two linear layers with a non-linear activation in between. The FFN acts as the model's "memory", storing factual knowledge learned during pre-training:

```
FFN(x) = max(0, x * W1 + b1) * W2 + b2
```

The inner dimension (d_ff) is typically 4x d_model. For d_model = 512, d_ff = 2048. Modern models use GELU or SwiGLU activations instead of ReLU. SwiGLU, used in LLaMA, shows improved performance:

```
SwiGLU(x) = (x * W1) . sigmoid(x * W_gate) * W2
```

Residual Connections & Layer Normalisation

Every sub-layer is wrapped with a residual connection and layer normalisation. Residual connections allow gradients to flow directly through the identity path, which is crucial for training deep models (GPT-3 uses 96 layers).

Aspect	Post-LN (Original Paper)	Pre-LN (Modern)
Formula	LN(x + Sublayer(x))	x + Sublayer(LN(x))
Stability	Requires careful warmup	Stable without warmup
Performance	Slightly higher peak	Consistent, easier to train
Used by	Original Transformer	GPT-2+, LLaMA, most LLMs

3.2 Positional Encoding

Self-attention is permutation-invariant: shuffling input tokens produces identical attention scores. Without explicit position signals, the model cannot distinguish "dog bites man" from "man bites dog".

The original paper uses deterministic sinusoidal encoding with different frequencies for each dimension:

```
PE(pos, 2i) = sin(pos / 10000^(2i/d))
```

```
PE(pos, 2i+1) = cos(pos / 10000^(2i/d))
```

These are added (not concatenated) to the token embeddings. Alternative approaches used in modern models:

Type	Model	Advantage
Sinusoidal (fixed)	Original Transformer	Generalises to unseen lengths
Learned absolute	BERT, GPT-2, ViT	Flexible, optimised during training
Rotary (RoPE)	LLaMA, Mistral	Relative position encoding, scalable
ALiBi	BLOOM	Linear bias, no extra parameters

```
import torch, math
```

```
def sinusoidal_positional_encoding(seq_len, d_model):
    pe = torch.zeros(seq_len, d_model)
    position = torch.arange(0, seq_len).unsqueeze(1).float()
    div_term = torch.exp(
        torch.arange(0, d_model, 2).float() *
        (-math.log(10000.0) / d_model)
    )
    pe[:, 0::2] = torch.sin(position * div_term)
    pe[:, 1::2] = torch.cos(position * div_term)
    return pe  # (seq_len, d_model)
```

3.3 Layer Normalisation & Residual Connections

Layer Normalisation normalises across the feature dimension for each sample independently, unlike Batch Normalisation which normalises across the batch dimension:

LN(x_i) = gamma * (x_i - mu_i) / (sigma_i + epsilon) + beta

Residual connections provide several critical benefits in deep Transformers:

- **Gradient flow:** Gradients can flow directly through the identity path, alleviating the vanishing gradient problem even in 96+ layer models.
- **Learning residuals:** Each sub-layer only needs to learn small refinements (residuals) rather than full transformations, making optimisation easier.
- **Ensemble effect:** Residual connections create an implicit ensemble effect where the network effectively acts as a collection of sub-networks of varying depth.

3.4 Feed-Forward Network

The position-wise FFN is applied identically and independently to each token position after the attention sub-layer. The expand-then-contract structure lets the network project into a higher-dimensional space for non-linear feature mixing before returning to the residual stream.

Recent research shows that FFNs act as **key-value memories**: weight matrix W1 stores "keys" (input patterns) and W2 stores "values" (associated factual knowledge). This explains why larger models with wider FFNs store more factual knowledge.

```
class PositionwiseFFN(nn.Module):
    def __init__(self, d_model=512, d_ff=2048, dropout=0.1):
        super().__init__()
        self.w1 = nn.Linear(d_model, d_ff)
        self.w2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.w2(self.dropout(F.gelu(self.w1(x))))
```

CHAPTER 4

Self-Attention: The Heart of the Transformer

4.1 Query, Key, Value (Q, K, V)

The attention mechanism is inspired by information retrieval systems. A useful analogy: imagine a library where you search for books (query), match against book titles/tags (keys), and retrieve the book contents (values).

Component	Role	Analogy
Query (Q)	What information am I looking for?	Search query
Key (K)	What information do I have to offer?	Book index / tags
Value (V)	What content do I actually contribute?	Book contents

Crucial point: in **self-attention**, all three projections (Q, K, V) come from the same input sequence. This differs from cross-attention where Q comes from the decoder and K, V come from the encoder.

```
import torch, torch.nn as nn

d_model, d_k = 512, 64
W_Q = nn.Linear(d_model, d_k, bias=False)
W_K = nn.Linear(d_model, d_k, bias=False)
W_V = nn.Linear(d_model, d_k, bias=False)

x = torch.rand(2, 10, d_model)  # (batch, seq_len, d_model)
Q = W_Q(x)  # (2, 10, 64) - queries
K = W_K(x)  # (2, 10, 64) - keys
V = W_V(x)  # (2, 10, 64) - values
```

4.2 Scaled Dot-Product Attention

The complete scaled dot-product attention formula is the heart of the entire Transformer architecture. Each step serves a specific purpose:

```
Attention(Q, K, V) = softmax(Q * K^T / sqrt(d_k)) * V
```

- **Q * K^T:** Matrix multiplication producing an (n x n) similarity score matrix. Element (i, j) indicates how relevant token j is to token i.
- **/ sqrt(d_k):** Scaling factor. Without it, large d_k produces large dot products that push softmax into saturation where gradients approach zero.
- **softmax(...):** Converts scores to a probability distribution. Each row sums to 1, forming attention weights.
- **(...) * V:** Weighted sum of values. Each query retrieves a blend of all values proportional to its attention weights.

Why scale by sqrt(d_k)? *For queries and keys with zero mean and unit variance, the dot product q.k = sum(q_i * k_i) has variance d_k. With d_k = 64, the standard deviation of the dot product is 8, large enough to make softmax "sharp" with near-zero gradients. Dividing by sqrt(64) = 8 restores variance to 1.*

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

def scaled_dot_product_attention(Q, K, V, mask=None):
    d_k = Q.size(-1)
    scores = torch.matmul(Q, K.transpose(-2, -1))
    scores = scores / math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, float('-inf'))
    weights = F.softmax(scores, dim=-1)
    output = torch.matmul(weights, V)
    return output, weights
```

4.3 Attention Masks

Padding Mask

In a batch, sequences have different lengths and are padded to the same length. The padding mask marks PAD positions as -inf before softmax so they receive zero weight. Without this, the model would allocate attention to meaningless tokens.

Causal (Look-ahead) Mask

Used in the decoder to ensure position i can only attend to positions <= i. This prevents the decoder from "seeing the future" during training. The mask is an upper-triangular matrix with -inf above the diagonal.

```
def create_causal_mask(seq_len):
    mask = torch.triu(
        torch.ones(seq_len, seq_len, dtype=torch.bool),
        diagonal=1
    )
    return mask
# Result for seq_len=5:
# [[0, 1, 1, 1, 1],
#  [0, 0, 1, 1, 1],
#  [0, 0, 0, 1, 1],
#  [0, 0, 0, 0, 1],
#  [0, 0, 0, 0, 0]]
```

4.4 Complexity & Efficiency

Self-attention has O(n^2) complexity in both time and memory because every token must compute an attention score with every other token. For long sequences (n > 4096), this becomes a significant bottleneck.

Method	Complexity	Principle
Standard Attention	O(n^2 * d)	Full attention matrix
Sparse Attention (Longformer)	O(n * w * d)	Local window + global tokens
Linformer	O(n * k * d)	Project K,V to lower dimension
Flash Attention	O(n^2 * d)	IO-aware, no n^2 in HBM
Performer	O(n * d^2)	Random feature approximation
Mamba (SSM)	O(n * d)	State space model, not attention

CHAPTER 5

Multi-Head Attention

5.1 Concept & Motivation

Multi-Head Attention (MHA) runs h independent attention operations ("heads") in parallel, each with its own learned projection matrices. Each head can learn to attend to different aspects of the input:

```
MultiHead(Q, K, V) = Concat(head_1, ..., head_h) * W_O
```

```
head_i = Attention(Q * W_Q_i, K * W_K_i, V * W_V_i)
```

Research (Clark et al., 2019) shows that individual heads tend to specialise:

Head Type	Behaviour	Example
Syntactic	Follows dependency arcs	subject-verb agreement
Positional	Focuses on adjacent tokens	bigram/trigram patterns
Coreference	Links pronouns to antecedents	'it' -> 'the animal'
Rare token	Focuses on informative tokens	domain-specific keywords
Separator	Tracks delimiter tokens	[SEP], periods, commas

Insight: *Total parameter count is the same for 1 large head vs h small heads (since d_k = d_model / h), but multiple heads provide richer subspace representations. A single head must average over all aspects, whereas h heads can specialise.*

5.2 Step-by-Step Computation

- **Step 1 - Linear Projections:** Project input X to Q_i, K_i, V_i for each head i, where d_k = d_model / h.
- **Step 2 - Scaled Dot-Product:** Each head computes attention independently: head_i = Attention(Q_i, K_i, V_i)

- **Step 3 - Concatenation:** Stack all head outputs: [h1; h2; ...; hh], producing a vector of dimension h * d_k = d_model.
- **Step 4 - Output Projection:** Multiply by W_O to mix information across heads.

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model=512, num_heads=8):
        super().__init__()
        self.h = num_heads
        self.d_k = d_model // num_heads
        self.W_q = nn.Linear(d_model, d_model, bias=False)
        self.W_k = nn.Linear(d_model, d_model, bias=False)
        self.W_v = nn.Linear(d_model, d_model, bias=False)
        self.W_o = nn.Linear(d_model, d_model, bias=False)

    def split_heads(self, x, B, T):
        return x.view(B, T, self.h, self.d_k).transpose(1, 2)

    def forward(self, x):
        B, T, _ = x.shape
        Q = self.split_heads(self.W_q(x), B, T)
        K = self.split_heads(self.W_k(x), B, T)
        V = self.split_heads(self.W_v(x), B, T)
        scores = Q @ K.transpose(-2, -1) / math.sqrt(self.d_k)
        weights = F.softmax(scores, dim=-1)
        heads = (weights @ V).transpose(1, 2).reshape(B, T, -1)
        return self.W_o(heads)
```

5.3 Grouped Query Attention (GQA)

Grouped Query Attention (GQA), used in LLaMA-2/3, is a memory-efficient variant that shares K and V projections across groups of query heads. This is critical for reducing KV-cache size during inference.

Variant	Q Heads	KV Heads	KV-Cache Size
MHA (standard)	h	h	Baseline (100%)
MQA (Multi-Query)	h	1	1/h (12.5% for h=8)
GQA (Grouped)	h	g	g/h (25% for g=2, h=8)

GQA provides an excellent trade-off: performance approaching full MHA while significantly reducing memory footprint during inference. LLaMA-2 70B uses GQA with 8 KV heads and 64 query heads.

CHAPTER 6

Encoder-Decoder Architecture

6.1 Encoder Stack

The encoder processes the entire source sequence in parallel. Each of its N identical blocks applies multi-head self-attention followed by a position-wise FFN, with residual connections and layer normalisation.

The output is a sequence of contextualised embeddings Z of shape (n x d) - one vector per source token, enriched with global context from all other tokens. This representation is passed to every decoder layer via cross-attention.

6.2 Decoder Stack & Masked Attention

Each decoder block has three sub-layers, compared to two in the encoder:

- **Masked Multi-Head Self-Attention:** The decoder attends to its own previous outputs, but a causal mask prevents position t from seeing positions > t, preserving the autoregressive property.
- **Encoder-Decoder Cross-Attention:** Q comes from the decoder, K and V from the encoder memory Z. This lets the decoder "read" the source representation.
- **Position-wise FFN:** Same as in the encoder, applied independently to each position.

Why is masked self-attention needed? During training, the entire target sequence is fed to the decoder simultaneously (teacher forcing). Without masking, position t could see ground-truth tokens at positions t+1, t+2, ... - "cheating" by reading the answer.

6.3 Cross-Attention

Cross-attention is the mechanism that connects the encoder and decoder. Unlike self-attention, Q comes from the current decoder representation, while K and V come from the encoder's final output.

```
class CrossAttention(nn.Module):
    def __init__(self, d_model=512, num_heads=8):
        super().__init__()
        self.attn = nn.MultiheadAttention(
            d_model, num_heads, batch_first=True)
        self.norm = nn.LayerNorm(d_model)

    def forward(self, decoder_x, encoder_output):
        out, _ = self.attn(
            query=decoder_x,
            key=encoder_output,
            value=encoder_output)
        return self.norm(decoder_x + out)
```

6.4 KV-Cache & Decoding Strategies

KV-Cache

During autoregressive decoding, K and V projections for all previous tokens never change. Recomputing them at every step is wasteful. The KV-cache stores K and V tensors from all past decoder steps; at step t only the new token's K/V pair is computed and appended. This reduces per-step computation from O(t * d) to O(d).

Decoding Strategies

Strategy	Description	Pros / Cons
Greedy	Take highest-probability token at each step	Fast but suboptimal; deterministic
Beam Search	Keep top-k partial sequences at each step	Better than greedy; slower
Top-k Sampling	Sample from the k highest-probability tokens	Diverse; can be incoherent if k too large
Top-p (Nucleus)	Sample from tokens whose cumulative prob >= p	Adaptive; standard for modern LLMs
Temperature	Scale logits before softmax	Controls diversity; often combined with others

CHAPTER 7

Vision Transformer (ViT)

7.1 From NLP to Computer Vision

Vision Transformer (ViT), introduced by Dosovitskiy et al. in 2020, demonstrated that the standard Transformer encoder can be applied directly to images with competitive performance against CNNs, especially when training data is sufficiently large.

The key insight of ViT is to treat an image as a sequence of "visual words" (patches), just as an NLP Transformer treats text as a sequence of tokens. Each patch is projected into embedding space and processed by a standard Transformer encoder.

- **1. Patch Splitting:** The image H x W x C is divided into N = HW/P^2 non-overlapping patches of size P x P. For a 224x224 image with P=16, N = 196 patches.
- **2. Linear Projection:** Each patch is flattened to a vector of size P^2 * C and projected to d_model via a learned linear layer.
- **3. CLS Token:** A learnable [CLS] embedding is prepended. Its final-layer output is used for classification.
- **4. Positional Embeddings:** Learned 1-D positional embeddings are added to each patch embedding (including the CLS token).
- **5. Transformer Encoder:** The N+1 token sequence is processed by a standard Transformer encoder.

7.2 Patch Embedding & CLS Token

```
class PatchEmbedding(nn.Module):
    def __init__(self, img_size=224, patch_size=16,
                 in_channels=3, d_model=768):
        super().__init__()
        self.num_patches = (img_size // patch_size) ** 2
        self.proj = nn.Conv2d(
            in_channels, d_model,
            kernel_size=patch_size, stride=patch_size)
        self.cls_token = nn.Parameter(torch.zeros(1, 1, d_model))
        self.pos_embed = nn.Parameter(
            torch.zeros(1, self.num_patches + 1, d_model))
    def forward(self, x):  # x: (B, 3, 224, 224)
        B = x.size(0)
        x = self.proj(x).flatten(2).transpose(1, 2)  # (B, N, d)
        cls = self.cls_token.expand(B, -1, -1)
        x = torch.cat([cls, x], dim=1)  # (B, N+1, d)
        return x + self.pos_embed
```

The CLS token aggregates information from all patches through self-attention across all layers. Its final hidden state is fed into an MLP classification head. Interestingly, ViT uses learned 1-D positional embeddings rather than 2-D spatial encodings - the model internally learns to encode 2D structure from 1D positional embeddings.

7.3 ViT Architecture Deep Dive

Model	d_model	Heads	Layers	Params	Patch
ViT-Small	384	6	12	22M	16
ViT-Base/16	768	12	12	86M	16
ViT-Large/16	1024	16	24	307M	16
ViT-Huge/14	1280	16	32	632M	14

Key Finding: *ViT requires pre-training on very large datasets (ImageNet-21k or JFT-300M) to achieve optimal performance. On small datasets, CNNs with built-in inductive biases (locality, translation equivariance) typically outperform ViT.*

CHAPTER 8

CNN vs ViT & Hybrid Models

8.1 Inductive Bias

Inductive bias is a set of assumptions built into the architecture that constrains the hypothesis space, aiding generalisation when data is limited.

- **Translation Equivariance (CNN):** The same filter is applied at all spatial locations. A feature detected in the top-left is detected everywhere.
- **Locality (CNN):** Each neuron sees only a local receptive field, reflecting the assumption that nearby pixels are most correlated.

ViT has neither by default: it treats patches as unordered tokens and must learn spatial structure from data alone, requiring more examples to compensate.

8.2 When to Choose CNN vs ViT

Scenario	Prefer CNN	Prefer ViT
Dataset size	Small/medium (<100k)	Large (>1M)
Compute budget	Constrained	High
Task	Dense prediction (segmentation, detection)	Classification, retrieval
Domain shift	Low	High (with pre-training)
Edge deployment	EfficientNet, MobileNet	ViT-Tiny, Mobile-ViT

8.3 Hybrid Models: Swin, ConvNeXt, TransUNet

Swin Transformer

Swin Transformer (Liu et al., 2021) addresses ViT's scalability issues with two key innovations: shifted window attention (reducing complexity from O(N^2) to O(N)) and hierarchical feature maps (producing multi-scale features like CNN feature pyramids). Swin has become the backbone for many detection and segmentation frameworks.

ConvNeXt

ConvNeXt (Liu et al., 2022) takes the opposite approach: modernising a pure CNN with Transformer design choices (large 7x7 kernels, LayerNorm, GELU, inverted bottleneck). The result is a CNN competitive with Swin Transformer without using attention at all.

Hybrid CNN-Transformer

Hybrid models combine CNN local feature extraction with Transformer global context modelling. Examples include ViT with a ResNet stem, TransUNet for medical image segmentation, and CoAtNet which balances convolution in early layers with attention in later layers.

```
class HybridViT(nn.Module):
    def __init__(self, num_classes=1000):
        super().__init__()
        resnet = torchvision.models.resnet50(pretrained=True)
        self.cnn_stem = nn.Sequential(*list(resnet.children())[:-2])
        enc_layer = nn.TransformerEncoderLayer(
            2048, 8, dim_feedforward=4096, batch_first=True)
        self.transformer = nn.TransformerEncoder(enc_layer, 2)
        self.pool = nn.AdaptiveAvgPool1d(1)
        self.fc = nn.Linear(2048, num_classes)

    def forward(self, x):
        features = self.cnn_stem(x)             # (B, 2048, 7, 7)
        tokens = features.flatten(2).transpose(1, 2)  # (B, 49, 2048)
        z = self.transformer(tokens)            # (B, 49, 2048)
        z = self.pool(z.transpose(1, 2))[:, :, 0]
        return self.fc(z)
```

CHAPTER 9

Training Techniques & Optimisation

9.1 Learning Rate Scheduling

Learning rate scheduling is critical for successful Transformer training. The original paper introduced a warmup schedule that has remained standard practice.

Scheduler	Description	Used In
Linear Warmup + Cosine Decay	Linear warmup, then cosine annealing to 0	ViT, BERT, most modern models
Warmup + Inverse Sqrt Decay	Original paper; lr ~ min(step^-0.5, ...)	Original Transformer
OneCycleLR	Cosine cycle with warmup & cooldown	Fast training; super-convergence
Constant + Decay	Constant during training, then decay at end	LLM pre-training (Chinchilla)

9.2 Flash Attention

Flash Attention (Dao et al., 2022) is an IO-aware exact attention algorithm that avoids materialising the full n x n attention matrix in high-bandwidth memory (HBM):

- **Tiling:** Q, K, V are divided into blocks that fit in SRAM (on-chip memory).
- **Kernel Fusion:** Softmax + matmul operations are fused into a single GPU kernel.
- **Memory O(n):** No n^2 matrix stored in HBM.
- **2-4x Speedup:** Significant wall-clock speedup, enabling context windows of 32k, 128k, and even 1M+ tokens.

9.3 LoRA & Parameter-Efficient Fine-Tuning

LoRA (Low-Rank Adaptation, Hu et al., 2021) enables fine-tuning large Transformer models by training only a small fraction of parameters. Instead of updating the entire weight matrix W, LoRA adds a low-rank decomposition:

```
W' = W + delta_W = W + A * B

A in R^(d x r), B in R^(r x k), r << min(d, k)
```

W is frozen; only A and B are trained. With r = 8, a 768 x 768 matrix goes from 590k to 12k trainable parameters per layer (98% reduction), while matching or approaching full fine-tuning quality.

Method	Description	Advantage
LoRA	Low-rank A*B decomposition	Simple, effective, 98% param reduction
QLoRA	LoRA + 4-bit quantization of base model	Fine-tune 65B model on a single GPU
DoRA	Weight-Decomposed LoRA; separate direction & magnitude	Closer to full fine-tuning
AdaLoRA	Adaptive rank allocation per layer	Optimal rank distribution

```python
class LoRALinear(nn.Module):
    def __init__(self, in_f, out_f, rank=8, alpha=16):
        super().__init__()
        self.linear = nn.Linear(in_f, out_f, bias=False)
        self.linear.weight.requires_grad = False  # Frozen
        self.A = nn.Parameter(torch.randn(rank, in_f) * 0.01)
        self.B = nn.Parameter(torch.zeros(out_f, rank))
        self.scale = alpha / rank

    def forward(self, x):
        base = self.linear(x)
        lora = (x @ self.A.T) @ self.B.T
        return base + self.scale * lora
# Trainable: 2 * 8 * 768 = 12,288 params (vs 589,824 full)
```

CHAPTER 10

Modern Transformer Variants

10.1 BERT, GPT, T5: Three Paradigms

BERT (Encoder-Only)

BERT (Bidirectional Encoder Representations from Transformers) uses an encoder-only architecture with bidirectional attention. Pre-training uses Masked Language Modeling (MLM) where 15% of tokens are masked and the model must predict them, plus Next Sentence Prediction (NSP). BERT excels at understanding tasks: classification, NER, extractive QA.

GPT (Decoder-Only)

GPT uses a decoder-only architecture with causal attention. Pre-training uses autoregressive language modeling: predicting the next token. This architecture dominates the modern LLM era (GPT-3/4, LLaMA, Mistral, Claude). Scaling laws show performance improves predictably with model size and data.

T5 (Encoder-Decoder)

T5 (Text-to-Text Transfer Transformer) uses a full encoder-decoder architecture. All NLP tasks are framed as "text-to-text": text input goes to the encoder, text output is generated by the decoder. This approach is very flexible and achieved SOTA on many NLP benchmarks.

Aspect	BERT	GPT	T5
Architecture	Encoder-only	Decoder-only	Encoder-Decoder
Attention	Bidirectional	Causal (left-to-right)	Bidirectional + Cross
Pre-training	MLM + NSP	Autoregressive LM	Span corruption
Strength	Understanding (NLU)	Generation (NLG)	Seq2Seq tasks
Examples	BERT, RoBERTa, DeBERTa	GPT-2/3/4, LLaMA, Mistral, Claude	T5, mT5, BART, mBART

10.2 Large Language Models (LLMs)

Model	Year	Parameters	Training Data	Key Architecture
GPT-2	2019	1.5B	40GB (WebText)	Decoder-only, Post-LN
GPT-3	2020	175B	570GB	Decoder-only, Pre-LN
PaLM	2022	540B	780B tokens	Decoder-only, SwiGLU
LLaMA-2	2023	70B	2T tokens	GQA, RoPE, SwiGLU
Mistral 7B	2023	7B	Undisclosed	GQA, Sliding Window
LLaMA-3	2024	405B	15T+ tokens	GQA, RoPE, SwiGLU

Recent trends show that smaller models trained on more data (following Chinchilla scaling laws) often outperform larger models. LLaMA-2 7B is competitive with much larger models thanks to extensive training data and an optimised architecture.

10.3 DINO & Self-Supervised Learning

DINO (Self-DIstillation with NO labels) is a self-supervised learning framework for ViT that shows high-quality visual representations can be learned without labels. DINO trains a student ViT to match the output distribution of a teacher ViT (updated via exponential moving average) using different augmented views of the same image. The CLS token learns semantically meaningful representations, and attention maps produce surprisingly clear object segmentation ("emergent segmentation").

CHAPTER 11

Practical Implementation Guide

11.1 Building from Scratch with PyTorch

Step 1: Self-Attention Module

```
class SelfAttention(nn.Module):
    def __init__(self, d_model, d_k):
        super().__init__()
        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_k, bias=False)
        self.scale = math.sqrt(d_k)

    def forward(self, x, mask=None):
        Q, K, V = self.W_q(x), self.W_k(x), self.W_v(x)
        scores = torch.bmm(Q, K.transpose(1, 2)) / self.scale
        if mask is not None:
            scores = scores.masked_fill(mask, float('-inf'))
        weights = F.softmax(scores, dim=-1)
        return torch.bmm(weights, V), weights
```

Step 2: Multi-Head Attention

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model=512, num_heads=8):
        super().__init__()
        self.h = num_heads
        self.d_k = d_model // num_heads
        self.W_qkv = nn.Linear(d_model, 3 * d_model, bias=False)
        self.W_o = nn.Linear(d_model, d_model, bias=False)

    def forward(self, x, mask=None):
        B, T, D = x.shape
        qkv = self.W_qkv(x).reshape(B, T, 3, self.h, self.d_k)
        qkv = qkv.permute(2, 0, 3, 1, 4)
        Q, K, V = qkv[0], qkv[1], qkv[2]
        scores = (Q @ K.transpose(-2, -1)) / math.sqrt(self.d_k)
        if mask is not None:
            scores = scores.masked_fill(mask, float('-inf'))
        attn = F.softmax(scores, dim=-1)
        out = (attn @ V).transpose(1, 2).reshape(B, T, D)
        return self.W_o(out)
```

Step 3: Transformer Encoder Block

```
class TransformerBlock(nn.Module):
    def __init__(self, d_model=512, num_heads=8, d_ff=2048, dropout=0.1):
        super().__init__()
        self.attn = MultiHeadAttention(d_model, num_heads)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, d_ff), nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(d_ff, d_model), nn.Dropout(dropout))
        self.ln1 = nn.LayerNorm(d_model)
        self.ln2 = nn.LayerNorm(d_model)

    def forward(self, x, mask=None):
        x = x + self.attn(self.ln1(x), mask)   # Pre-LN
        x = x + self.ffn(self.ln2(x))
        return x
```

Step 4: Full Transformer Encoder

```
class TransformerEncoder(nn.Module):
    def __init__(self, vocab_size, d_model=512, num_heads=8,
                 num_layers=6, d_ff=2048, max_seq_len=512):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.pos_enc = nn.Parameter(torch.zeros(1, max_seq_len, d_model))
        self.blocks = nn.ModuleList([
            TransformerBlock(d_model, num_heads, d_ff)
            for _ in range(num_layers)])
        self.norm = nn.LayerNorm(d_model)
        self.scale = math.sqrt(d_model)

    def forward(self, x, mask=None):
        seq_len = x.size(1)
        x = self.embedding(x) * self.scale + self.pos_enc[:, :seq_len]
        for block in self.blocks:
            x = block(x, mask)
        return self.norm(x)
```

11.2 Using Hugging Face Transformers

```
from transformers import (
    AutoTokenizer, AutoModelForSequenceClassification,
    Trainer, TrainingArguments
)

model_name = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSequenceClassification.from_pretrained(
    model_name, num_labels=2)

inputs = tokenizer("Hello, Transformers!", return_tensors="pt")
outputs = model(**inputs)
```

```
training_args = TrainingArguments(
    output_dir="./results", num_train_epochs=3,
    per_device_train_batch_size=16, learning_rate=2e-5,
    warmup_ratio=0.1, weight_decay=0.01)

trainer = Trainer(model=model, args=training_args,
    train_dataset=train_dataset, eval_dataset=eval_dataset)
trainer.train()
```

CHAPTER 12

Industry Applications & Case Studies

12.1 NLP: Translation, Summarisation, QA

Machine Translation

Encoder-decoder models like T5 and mBART achieve near-human performance on many language pairs. Google Translate has used the Transformer architecture since 2017. For low-resource languages, multilingual pre-training (mT5, BLOOM) enables knowledge transfer from high-resource languages.

Text Summarisation

Abstractive summarisation, where the model generates summaries in its own words, became practical thanks to Transformers. BART and PEGASUS are optimised specifically for summarisation. Modern LLMs can perform summarisation without task-specific fine-tuning.

Question Answering

BERT revolutionised QA with extractive QA capability. Modern models support open-domain QA through Retrieval-Augmented Generation (RAG), where relevant documents are retrieved and then an LLM generates the answer. RAG combines a retriever (BERT-based bi-encoder) with a generator (GPT/T5-based decoder).

12.2 Computer Vision: Detection, Segmentation

Task	Model	Approach
Image Classification	ViT, DeiT, Swin	Patch embedding + Transformer encoder
Object Detection	DETR, DINO-DETR	End-to-end detection; no NMS/anchors

Semantic Segmentation	SegFormer, Mask2Former	Hierarchical features + pixel decoder
Medical Imaging	TransUNet, Swin-UNETR	Hybrid CNN-Transformer for 3D volumes
Video Understanding	TimeSformer, ViViT	Spatial-temporal attention on video frames

12.3 Multimodal: CLIP, GPT-4V, Flamingo

CLIP (Contrastive Language-Image Pre-training)

CLIP (OpenAI, 2021) trains an image encoder (ViT) and text encoder (Transformer) jointly to map images and text into a shared embedding space. Trained on 400M image-text pairs, CLIP enables zero-shot image classification using text descriptions alone.

Large Multimodal Models

GPT-4V, Claude, Gemini, and LLaVA extend LLMs with image processing capabilities. The typical architecture: a vision encoder (ViT or CLIP) produces visual tokens that are interleaved with text tokens and processed by a decoder-only LLM. This enables visual QA, chart analysis, document reading, and complex visual reasoning.

Industry	Application	Model/Approach
Healthcare	Radiology report generation, pathology analysis	BiomedCLIP, Med-PaLM
E-commerce	Visual search, product recommendation	CLIP-based retrieval
Autonomous Driving	Scene understanding, planning	BEVFormer, UniAD
Document AI	Invoice processing, form understanding	LayoutLM, Donut
Creative	Image generation, text-to-video	DALL-E, Stable Diffusion, Sora

ADVANCED TOPICS

Attention Visualisation & Interpretability

A.1 Why Interpretability Matters

As Transformer models grow larger and are deployed in critical applications like healthcare, fintech, and legal, understanding **why** a model makes a particular decision becomes essential. Attention weights provide a unique window into the model's internal processes, though with important caveats.

Attention weights show how much each token "attends to" other tokens. However, research by Jain & Wallace (2019) found that attention weights do not always correlate strongly with other importance measures. High attention weight does not always mean that token is important for the model's final decision.

Warning: *Attention maps are informative but not definitive causal explanations. Use them alongside other interpretability methods such as integrated gradients, SHAP, or probing classifiers for a more complete picture.*

A.2 Visualisation Techniques

Attention Weight Heatmaps

The most basic way to visualise attention is to plot the (n x n) attention weight matrix as a heatmap. Each row shows the attention distribution from one query token, and each column shows how much attention a key token receives.

```
import matplotlib.pyplot as plt
import seaborn as sns
import torch

tokens = ["The", "animal", "didn't", "cross", "because", "it", "was", "tired"]
n = len(tokens)
weights = torch.softmax(torch.randn(n, n), dim=-1).numpy()

fig, ax = plt.subplots(figsize=(8, 6))
sns.heatmap(weights, xticklabels=tokens, yticklabels=tokens,
            cmap='Blues', ax=ax, annot=True, fmt='.2f')
ax.set_title("Self-Attention Weights (Head 3, Layer 5)")
plt.tight_layout()
plt.savefig("attention_heatmap.png", dpi=150)
```

Attention Rollout & Flow

Attention Rollout (Abnar & Zuidema, 2020) combines attention weights across all layers by multiplying attention matrices successively, accounting for residual connections. This provides a more accurate picture of how information flows from input to output compared to examining a single layer.

```
Rollout(l) = 0.5 * I + 0.5 * A(l)
Total = Rollout(1) * Rollout(2) * ... * Rollout(L)
```

A.3 Probing Classifiers

Probing classifiers test what information is encoded in Transformer hidden representations. A simple classifier (usually linear) is trained on frozen representations to predict linguistic properties. Research shows different layers encode different information: early layers encode surface-level information (POS tags), middle layers encode syntactic information (dependency trees), and final layers encode task-specific information.

Efficient Transformers: Overcoming $O(n^2)$

B.1 Taxonomy of Efficient Attention

Category	Method	Strategy	Complexity
Sparse	Longformer BigBird	Local window + global tokens	O(n * w)
Low-Rank	Linformer Nystrom	Project K,V to lower dimension	O(n * k)
Kernel-based	Performer Random Feature	Approx softmax with kernels	O(n * d^2)
IO-Aware	Flash Attention Flash-2	Tile in SRAM; same math	O(n^2) but 2-4x faster
State Space	Mamba S4	Replace attention with SSM	O(n * d)
Hybrid	MoA H-Transformer	Combine local + global	O(n * sqrt(n))

B.2 Longformer: Sliding Window + Global Attention

Longformer (Beltagy et al., 2020) combines two attention patterns: local sliding window attention where each token only attends to neighbours within a window of size w, and global attention where special tokens (like [CLS]) attend to all tokens and are attended by all tokens. This enables processing documents with 16k+ tokens.

B.3 Flash Attention: IO-Aware Computing

Flash Attention takes a different approach: rather than reducing computation, it optimises **how** computation is performed. The main bottleneck is memory bandwidth, not arithmetic.

- **Tiling:** Divides Q, K, V into blocks that fit in SRAM.
- **Online Softmax:** Computes softmax incrementally without storing the full matrix.
- **Fused Kernel:** Combines matmul, softmax, and masking into a single GPU kernel.
- **Recomputation:** During backward pass, the attention matrix is recomputed from Q, K, V rather than stored, reducing memory from O(n^2) to O(n).

B.4 State Space Models: Beyond Attention

Mamba (Gu & Dao, 2023) represents a new paradigm that replaces attention with Selective State Space Models (SSM). Instead of computing pairwise interactions between tokens, Mamba processes sequences through input-conditioned recurrence, achieving linear O(n) complexity while maintaining long-range dependency modelling. Key advantages: linear inference complexity, constant memory per step (no expanding KV-cache), and much higher throughput on long sequences.

Training at Scale

C.1 Distributed Training Strategies

Strategy	What is Split	When to Use	Example Framework
Data Parallelism (DP)	Mini-batch divided across GPUs	Model fits on 1 GPU	PyTorch DDP, Horovod
Tensor Parallelism (TP)	Weight matrices split across GPUs	Layer too large for 1 GPU	Megatron-LM
Pipeline Parallelism (PP)	Layers assigned to different GPUs	Very deep models (many layers)	GPipe, PipeDream
Expert Parallelism (EP)	MoE experts on different GPUs	Mixture of Experts models	DeepSpeed-MoE, Switch
ZeRO	Optimizer states, gradients, params	Memory optimisation for DP	DeepSpeed ZeRO Stage 1-3
FSDP	Parameters sharded and gathered	Large models with PyTorch	PyTorch FSDP

C.2 Mixed Precision Training

Mixed precision training uses smaller floating point formats (FP16 or BF16) for most computations while keeping master weights in FP32. Benefits: 2x memory savings and 2-4x throughput on GPUs with Tensor Cores.

- **Loss Scaling:** Gradients are scaled up to prevent underflow in FP16, then scaled down before the optimizer step.
- **Master Weights in FP32:** The optimizer stores FP32 weight copies for numerical accuracy while forward/backward passes use FP16/BF16.
- **BF16 vs FP16:** BF16 has the same range as FP32 (8-bit exponent) but lower precision. Usually more stable and does not require loss scaling.

```
from torch.cuda.amp import autocast, GradScaler
```

```
model = TransformerModel().cuda()
optimizer = torch.optim.AdamW(model.parameters(), lr=3e-4)
scaler = GradScaler()

for batch in dataloader:
    optimizer.zero_grad()
    with autocast():  # Forward in FP16
        outputs = model(batch['input_ids'].cuda())
        loss = criterion(outputs, batch['labels'].cuda())
    scaler.scale(loss).backward()
    scaler.unscale_(optimizer)
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
    scaler.step(optimizer)
    scaler.update()
```

C.3 Tokenisation

Algorithm	Used In	Principle
BPE (Byte-Pair Encoding)	GPT-2/3/4, LLaMA, Mistral	Merge most frequent byte pairs iteratively
WordPiece	BERT, DistilBERT	Maximise likelihood on training corpus
SentencePiece (Unigram)	T5, mBART, LLaMA, ALBERT	Probabilistic model; language-independent

Practical Tips & Troubleshooting

D.1 Common Training Issues

Issue	Symptoms	Solution
Loss not decreasing	Stagnant or oscillating loss	Reduce LR; check data; verify architecture
Loss NaN/Inf	Loss becomes NaN after some steps	Use grad clipping; reduce LR; check BF16
Training divergence	Loss spikes then becomes NaN	Reduce LR; add warmup; check batch size
Overfitting	Low train loss, rising val loss	Add dropout; weight decay; data augmentation
Slow training	Low GPU utilisation	Optimise dataloader; mixed precision; packing
Memory OOM	CUDA out of memory error	Gradient accumulation; activation checkpointing

D.2 Hyperparameter Guidelines

Hyperparameter	Small (< 100M)	Medium (100M-1B)	Large (> 1B)
Learning Rate	3e-4 to 1e-3	1e-4 to 3e-4	1e-5 to 1e-4
Warmup Steps	1k - 5k	2k - 10k	2k - 5k
Batch Size (tokens)	32k - 128k	256k - 1M	1M - 4M
Weight Decay	0.01 - 0.1	0.01 - 0.1	0.1
Dropout	0.1 - 0.3	0.0 - 0.1	0.0
Gradient Clipping	1.0	1.0	1.0
Optimizer	AdamW	AdamW	AdamW / Adafactor

LR Schedule	Cosine / Linear	Cosine	Cosine / WSD

D.3 Evaluation & Benchmarking

Domain	Benchmark	Metric	Description
NLU	GLUE / SuperGLUE	Accuracy, F1	8-10 NLU tasks incl. sentiment, NLI, QA
LLM General	MMLU	Accuracy	57 subject areas; knowledge & reasoning
LLM Reasoning	GSM8K, MATH	Accuracy	Math word problems; formal math
LLM Coding	HumanEval, MBPP	pass@k	Code generation benchmarks
Vision	ImageNet-1K	Top-1 Acc	1000-class image classification
Vision	COCO	mAP	Object detection & segmentation
Multimodal	VQAv2, TextVQA	Accuracy	Visual question answering

Emerging Trends & the Future

E.1 Mixture of Experts (MoE)

Mixture of Experts (MoE) enables models with very large parameter counts while keeping compute costs manageable. Instead of one large FFN, MoE uses many "expert" FFNs and a router that selects a subset of experts for each token. Example: Mixtral 8x7B has 47B total parameters but only activates ~13B per forward pass.

Model	Total Params	Active Params	Experts	Top-k
Switch Transformer	1.6T	~100B	128	1
GShard	600B	~10B	64	2
Mixtral 8x7B	47B	~13B	8	2
Mixtral 8x22B	141B	~39B	8	2
DeepSeek-V2	236B	~21B	160	6

E.2 Long Context Models

- **RoPE Scaling:** Positional Interpolation (PI) and NTK-aware scaling allow models trained on 4k context to extend to 32k-128k without full re-training.
- **Ring Attention:** Distributes attention computation across many devices in a ring, enabling context windows > 1M tokens.
- **Landmark Attention:** Inserts special landmark tokens to mark important segments, enabling efficient retrieval from long contexts.
- **Sliding Window + Sink Tokens:** StreamingLLM shows that keeping a few initial tokens ("attention sinks") + a sliding window suffices for streaming inference.

E.3 Multimodal & Foundation Models

Model	Modalities	Architecture	Key Capability
GPT-4V/o	Text + Image	Decoder-only + Vision Encoder	Visual QA, reasoning, document analysis
Gemini	Text + Image + Audio + Video	Multimodal from scratch	Native multimodal; long context
Claude 3+	Text + Image	Undisclosed	Strong reasoning; document analysis
LLaVA	Text + Image	CLIP + LLaMA + projection	Open-source; visual instruction
Sora	Text -> Video	DiT (Diffusion Transformer)	Video generation; world modelling

E.4 Quantisation & Efficient Deployment

Technique	Precision	Compression	Quality Impact
FP16/BF16	16-bit	2x vs FP32	Negligible
INT8 (GPTQ)	8-bit	4x vs FP32	< 1% degradation
INT4 (GPTQ)	4-bit	8x vs FP32	1-3% degradation
GGUF (llama.cpp)	2-8 bit mixed	4-16x	Depends on quant level
AWQ	4-bit	8x vs FP32	< 1% (activation-aware)

```
# 4-bit Quantization with bitsandbytes
from transformers import AutoModelForCausalLM, BitsAndBytesConfig

bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16,
    bnb_4bit_use_double_quant=True)

model = AutoModelForCausalLM.from_pretrained(
    "meta-llama/Llama-2-7b-hf",
    quantization_config=bnb_config,
    device_map="auto")
# 7B model: ~14GB FP16 -> ~3.5GB INT4
```

E.5 Agentic AI & Tool Use

Transformer-based LLMs are increasingly used as autonomous agents that can use tools, navigate the web, write and execute code, and interact with APIs. This development encompasses function calling, the ReAct (Reasoning + Acting) framework, and multi-agent systems where multiple LLMs collaborate to solve complex tasks.

Agentic architectures typically combine an LLM as the controller/planner with a tool repository (code interpreter, web search, database query, API calls) and a memory module (short-term working memory + long-term retrieval). Models like Claude, GPT-4, and Gemini support native tool use, where the model generates structured function calls executed by a runtime environment.

The key architectural patterns in agentic AI include: plan-and-execute workflows where the model decomposes complex tasks into sub-steps, reflection loops where the model evaluates and iterates on its own outputs, and tool-augmented generation where external tools extend the model's capabilities beyond text generation.

Deep Dive: Attention Variants

F.1 Relative Position Encodings

While the original Transformer uses absolute positional encodings, modern models increasingly adopt relative position approaches that encode the distance between tokens rather than their absolute positions. This provides better generalisation to unseen sequence lengths and more natural modelling of linguistic phenomena.

Rotary Position Embeddings (RoPE)

RoPE (Su et al., 2021), used in LLaMA and Mistral, encodes position information by rotating the query and key vectors. For a token at position m, the rotation is applied as a block-diagonal rotation matrix that depends only on the position index. The attention score between tokens at positions m and n naturally encodes their relative distance (m - n) through the properties of rotation:

```
f(q, m) . f(k, n) = g(q, k, m-n)
```

This elegant formulation means that position information is baked directly into the dot-product computation, requiring no additional parameters. RoPE also enables extrapolation to longer sequences through techniques like NTK-aware scaling and YaRN.

```
def apply_rotary_emb(x, freqs):
    # x: (B, T, H, d_k), freqs: (T, d_k/2)
    x_complex = torch.view_as_complex(x.float().reshape(*x.shape[:-1], -1, 2))
    freqs_complex = torch.polar(torch.ones_like(freqs), freqs)
    x_rotated = x_complex * freqs_complex
    return torch.view_as_real(x_rotated).flatten(-2).type_as(x)

def precompute_freqs(dim, max_seq_len, theta=10000.0):
    freqs = 1.0 / (theta ** (torch.arange(0, dim, 2).float() / dim))
    t = torch.arange(max_seq_len)
    return torch.outer(t, freqs)
```

ALiBi (Attention with Linear Biases)

ALiBi (Press et al., 2022), used in BLOOM, takes a radically simple approach: it adds a static linear bias to attention scores based on the distance between query and key positions. No learned parameters are needed. The bias is: penalty = -m * |i - j|, where m is a fixed slope that varies per head. ALiBi enables training on short sequences (e.g. 1024 tokens) while generalising to much longer sequences at inference time without any fine-tuning.

F.2 Multi-Query & Grouped Query Attention In-Depth

The KV-cache memory bottleneck during autoregressive inference has motivated several attention variants that reduce the number of key-value heads:

In standard MHA with h = 32 heads and d_k = 128, the KV-cache for a sequence of length n requires 2 * n * h * d_k * L * bytes_per_element memory. For a 70B model with 80 layers and 8192 context length, this can exceed 20GB per batch element.

Configuration	KV Heads	KV-Cache per Layer	Quality vs MHA
MHA (h=32)	32	2 * n * 32 * d_k	Baseline (100%)
MQA (Shazeer 2019)	1	2 * n * 1 * d_k (32x reduction)	Slight degradation on some tasks
GQA-4 (h=32, g=4)	4	2 * n * 4 * d_k (8x reduction)	Near-MHA quality
GQA-8 (h=32, g=8)	8	2 * n * 8 * d_k (4x reduction)	~Same as MHA

The empirical finding is that GQA-8 (8 KV groups for 32 query heads) achieves nearly identical quality to full MHA while using only 25% of the KV-cache memory. This is why LLaMA-2 70B uses GQA with 8 KV heads - it enables deployment with reasonable memory requirements while maintaining model quality.

F.3 Sliding Window Attention

Sliding window attention, used in Mistral 7B, restricts each token to only attend to a fixed window of w preceding tokens. Unlike Longformer which adds global tokens, Mistral relies on the stacking of multiple layers to effectively propagate information beyond the window size.

With a window size w = 4096 and L = 32 layers, information can theoretically flow across w * L = 131,072 token positions. This provides an effective receptive field far larger than the window size while keeping per-layer attention cost at O(n * w) instead of O(n^2). Combined with rolling KV-cache of fixed size, this enables inference on arbitrarily long sequences with constant memory.

Complete ViT Implementation

G.1 Minimal ViT from Scratch

The following complete implementation builds a Vision Transformer from scratch in PyTorch, combining all the components discussed in earlier chapters. This is a production-ready architecture that can be directly used for image classification experiments.

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

class PatchEmbed(nn.Module):
    """Convert image to patch embeddings."""
    def __init__(self, img_size=224, patch_size=16, in_ch=3, embed_dim=768):
        super().__init__()
        self.num_patches = (img_size // patch_size) ** 2
        self.proj = nn.Conv2d(in_ch, embed_dim,
                              kernel_size=patch_size, stride=patch_size)

    def forward(self, x):
        # (B, C, H, W) -> (B, N, D)
        return self.proj(x).flatten(2).transpose(1, 2)


class Attention(nn.Module):
    """Multi-head self-attention with optional dropout."""
    def __init__(self, dim, num_heads=12, qkv_bias=True, attn_drop=0., proj_drop=0.):
        super().__init__()
        self.num_heads = num_heads
        self.head_dim = dim // num_heads
        self.scale = self.head_dim ** -0.5
        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_drop)

    def forward(self, x):
        B, N, C = x.shape
        qkv = self.qkv(x).reshape(B, N, 3, self.num_heads, self.head_dim)
        qkv = qkv.permute(2, 0, 3, 1, 4)
        q, k, v = qkv.unbind(0)
        attn = (q @ k.transpose(-2, -1)) * self.scale
        attn = attn.softmax(dim=-1)
```

```python
        attn = self.attn_drop(attn)
        x = (attn @ v).transpose(1, 2).reshape(B, N, C)
        return self.proj_drop(self.proj(x))
```

```python
class MLP(nn.Module):
    """FFN with GELU activation."""
    def __init__(self, in_features, hidden_features=None, drop=0.):
        super().__init__()
        hidden_features = hidden_features or in_features * 4
        self.fc1 = nn.Linear(in_features, hidden_features)
        self.act = nn.GELU()
        self.fc2 = nn.Linear(hidden_features, in_features)
        self.drop = nn.Dropout(drop)

    def forward(self, x):
        return self.drop(self.fc2(self.drop(self.act(self.fc1(x)))))


class Block(nn.Module):
    """Transformer encoder block with Pre-LN."""
    def __init__(self, dim, num_heads, mlp_ratio=4., drop=0., attn_drop=0.):
        super().__init__()
        self.norm1 = nn.LayerNorm(dim)
        self.attn = Attention(dim, num_heads, attn_drop=attn_drop)
        self.norm2 = nn.LayerNorm(dim)
        self.mlp = MLP(dim, int(dim * mlp_ratio), drop)

    def forward(self, x):
        x = x + self.attn(self.norm1(x))
        x = x + self.mlp(self.norm2(x))
        return x
```

```python
class VisionTransformer(nn.Module):
    """Complete Vision Transformer for image classification."""
    def __init__(self, img_size=224, patch_size=16, in_ch=3, num_classes=1000,
                 embed_dim=768, depth=12, num_heads=12, mlp_ratio=4.,
                 drop_rate=0., attn_drop_rate=0.):
        super().__init__()
        self.patch_embed = PatchEmbed(img_size, patch_size, in_ch, embed_dim)
        num_patches = self.patch_embed.num_patches

        self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
        self.pos_embed = nn.Parameter(torch.zeros(1, num_patches + 1, embed_dim))
        self.pos_drop = nn.Dropout(drop_rate)

        self.blocks = nn.Sequential(*[
            Block(embed_dim, num_heads, mlp_ratio, drop_rate, attn_drop_rate)
            for _ in range(depth)])

        self.norm = nn.LayerNorm(embed_dim)
        self.head = nn.Linear(embed_dim, num_classes)

        # Initialise weights
        nn.init.trunc_normal_(self.pos_embed, std=0.02)
        nn.init.trunc_normal_(self.cls_token, std=0.02)

    def forward(self, x):
        B = x.shape[0]
        x = self.patch_embed(x)                             # (B, N, D)
        cls_tokens = self.cls_token.expand(B, -1, -1)       # (B, 1, D)
        x = torch.cat([cls_tokens, x], dim=1)               # (B, N+1, D)
        x = self.pos_drop(x + self.pos_embed)               # Add pos embed
        x = self.blocks(x)                                   # Transformer
        x = self.norm(x)                                     # Final norm
        return self.head(x[:, 0])                            # CLS -> logits

# Usage:
model = VisionTransformer(img_size=224, num_classes=1000, embed_dim=768,
                           depth=12, num_heads=12)
dummy = torch.randn(2, 3, 224, 224)
logits = model(dummy)  # (2, 1000)
print(f"Parameters: {sum(p.numel() for p in model.parameters()):,}")
```

G.2 ViT Training Recipe

Training a ViT from scratch requires specific techniques that differ from CNN training. Below is the recommended training recipe based on DeiT (Touvron et al., 2021) and modern best practices:

Component	Recommendation	Notes
Optimizer	AdamW (beta1=0.9, beta2=0.999)	Weight decay 0.05; exclude bias, norm
Learning Rate	1e-3 (base), scaled by batch_size / 256	Linear warmup 5 epochs; cosine decay
Batch Size	1024 (effective)	Gradient accumulation if GPU limited
Augmentation	RandAugment + Mixup + CutMix + Erasing	Essential for regularisation
Regularisation	Stochastic Depth (drop_path=0.1)	Label smoothing 0.1; dropout 0.0
Epochs	300 (from scratch) 30-100 (fine-tune)	Longer training helps ViT more than CNNs
Resolution	224x224 train, 384x384 fine-tune	Interpolate pos embed for resolution change

Practical Tip: *If you have less than 1M images, strongly consider fine-tuning a pre-trained ViT (from ImageNet-21k or CLIP) rather than training from scratch. Pre-trained ViTs are available via timm library: model = timm.create_model('vit_base_patch16_224', pretrained=True)*

Transformer Debugging Checklist

H.1 Pre-Training Debugging

Before starting a full training run, a systematic validation process can save many hours of wasted compute. The following checklist covers the most common failure modes in Transformer training:

Step 1: Verify Data Pipeline

- Decode a random batch back to text/images and visually inspect for correctness.
- Check tokeniser coverage: what percentage of tokens are UNK or sub-split?
- Verify attention masks are correct: masked positions should receive -inf, not 0.
- Confirm padding is applied consistently and padding tokens are properly masked.

Step 2: Sanity Check Model

- Can the model overfit a single batch? If not, there is likely a bug in the model or loss.
- Does initial loss match the expected value? For a classification task with C classes, initial loss should be approximately $-\log(1/C) = \log(C)$.
- Check parameter counts: are they in the expected range for the model configuration?
- Verify gradient flow: are any parameters receiving zero gradients?

Step 3: Monitor Training

- Track loss, learning rate, gradient norm, and throughput (tokens/sec) in real-time.
- Watch for gradient norm spikes that may indicate training instability.
- Monitor memory usage to catch OOM errors early.
- Check validation metrics periodically, not just training loss.

H.2 Fine-Tuning Debugging

Fine-tuning a pre-trained Transformer introduces its own set of potential issues. Common problems and solutions include:

Problem	Diagnostic	Solution
Catastrophic forgetting	Fine-tuned model loses general capabilities	Lower LR; freeze layers; use LoRA
Underfitting	Val metrics don't improve from pre-trained baseline	Increase LR; unfreeze more layers; more epochs
Distribution mismatch	Poor performance on target task despite low loss	Check label distribution; balance dataset; augment
Tokeniser mismatch	Model performs poorly on domain-specific text	Add domain tokens to vocab; re-train tokeniser
Evaluation mismatch	Good loss but bad metrics	Verify eval protocol; check for data leakage

H.3 Inference Optimisation Checklist

Deploying Transformers efficiently requires attention to several optimisation layers. The following checklist covers the most impactful optimisations:

- **KV-Cache:** Ensure KV-cache is enabled for autoregressive decoding. Without it, inference is O(n^2) per token instead of O(n).
- **Quantisation:** Apply INT8 or INT4 quantisation for 2-4x memory reduction with minimal quality loss. AWQ and GPTQ are the leading methods.
- **Batching:** Use continuous batching (vLLM, TGI) to maximise GPU utilisation across multiple concurrent requests.
- **Attention Backend:** Ensure Flash Attention or xFormers is enabled. Check with torch.backends.cuda.flash_sdp_enabled().
- **Speculative Decoding:** Use a small draft model to generate candidate tokens verified by the large model, achieving 2-3x speedup.
- **Tensor Parallelism:** For models too large for a single GPU, split layers across GPUs using Tensor Parallelism (e.g., via vLLM or TensorRT-LLM).

Parameter Counting & Memory Analysis

I.1 Parameter Count Analysis

Understanding exactly how many parameters a Transformer model has is crucial for estimating training cost, memory requirements, and inference throughput. Here we walk through a complete parameter breakdown for a standard Transformer encoder.

Embedding Layer

The embedding layer maps each token in the vocabulary to a d_model-dimensional vector. Parameters: |V| x d_model. For GPT-2 (V = 50,257, d_model = 768): 38.6M parameters. If positional embeddings are learned: add max_seq_len x d_model (e.g. 1024 x 768 = 0.8M).

Attention Parameters (per layer)

Each attention layer has four weight matrices: W_Q, W_K, W_V (each d_model x d_model) and W_O (d_model x d_model). Plus biases if used. Total per layer: 4 x d_model^2 (+ 4 x d_model with biases). For d_model = 768: 4 x 768^2 = 2.36M per layer.

FFN Parameters (per layer)

Each FFN has two linear layers: W1 (d_model x d_ff) and W2 (d_ff x d_model), plus biases. With d_ff = 4 x d_model: 2 x d_model x 4 x d_model = 8 x d_model^2. For d_model = 768: 8 x 768^2 = 4.72M per layer. This is ~2x the attention parameters!

Layer Norm Parameters

Each Layer Norm has gamma and beta vectors of size d_model. Two per Transformer block (one for attention, one for FFN): 4 x d_model per layer. Negligible in total count.

Total Count Formula

```
Total = V * d + n_layers * (12 * d^2 + 13 * d) + d * C
```

Where V = vocab size, d = d_model, C = number of classes (for classification head). For BERT-base (V=30522, d=768, L=12): ~110M parameters. For GPT-3 (V=50257, d=12288, L=96): ~175B parameters.

Model	d_model	Layers	d_ff	Vocab	Total Params
BERT-base	768	12	3072	30,522	110M
BERT-large	1024	24	4096	30,522	340M
GPT-2 Small	768	12	3072	50,257	117M
GPT-2 XL	1600	48	6400	50,257	1.5B
GPT-3	12,288	96	49,152	50,257	175B
LLaMA-2 7B	4096	32	11,008	32,000	6.7B
LLaMA-2 70B	8192	80	28,672	32,000	70B

I.2 Memory Requirements

Understanding memory consumption during training and inference is essential for hardware planning. The total GPU memory required includes model parameters, optimizer states, gradients, activations, and the KV-cache.

Training Memory Breakdown

For a model with P parameters trained with AdamW in mixed precision (BF16 forward, FP32 master weights):

Component	Memory (per param)	Example (7B)
Model Parameters (BF16)	2 bytes	14 GB
Master Weights (FP32)	4 bytes	28 GB
Gradients (BF16)	2 bytes	14 GB
Adam m (FP32)	4 bytes	28 GB
Adam v (FP32)	4 bytes	28 GB
Total (no activations)	16 bytes	112 GB
Activations (varies)	~2-4 bytes	14-28 GB
Grand Total	~18-20 bytes	~126-140 GB

This is why training a 7B model requires at least 4x A100 80GB GPUs with ZeRO-3 or FSDP, or a single H100 80GB with aggressive activation checkpointing. Inference is much cheaper: only model parameters (2 bytes/param in FP16) plus KV-cache.

KV-Cache Memory

During autoregressive inference, the KV-cache grows linearly with sequence length. For a model with L layers, h KV heads, d_k per head, and batch size B at sequence length n:

```
KV-Cache = 2 * B * L * h_kv * d_k * n * bytes_per_element
```

For LLaMA-2 70B (L=80, h_kv=8, d_k=128) with BF16 at sequence length 4096 and batch size 1: 2 * 1 * 80 * 8 * 128 * 4096 * 2 = 1.34 GB. With GQA (8 KV heads instead of 64), this is 8x smaller than full MHA, demonstrating why GQA is critical for long-context inference.

I.3 FLOPs Estimation

Estimating the total floating point operations (FLOPs) for training helps predict training time and cost. The standard approximation for a Transformer with P parameters trained on D tokens is:

```
Total FLOPs = 6 * P * D (forward + backward)
```

This means training LLaMA-2 70B on 2T tokens requires approximately: 6 * 70B * 2T = 840 * 10^21 FLOPs. On A100 GPUs achieving ~300 TFLOPS (BF16), this takes: 840 * 10^21 / (300 * 10^12) = 2.8 * 10^9 GPU-seconds, or about ~1,720 A100-GPU-days. At typical utilisation rates (~50%), the actual wall-clock time is roughly double.

Compute Cost Rule of Thumb: *At cloud prices of ~$2/GPU-hour for an A100, training a 7B model on 1T tokens costs approximately $200k-300k. Training a 70B model costs roughly 10-20x more. This is why parameter-efficient fine-tuning methods like LoRA are so valuable - they can reduce fine-tuning cost by 100x or more.*

I.4 Scaling Laws

Scaling laws (Kaplan et al., 2020; Hoffmann et al., 2022) provide empirical formulas that predict model performance as a function of model size, dataset size, and compute budget. These have become the foundation for planning large-scale training runs.

The Chinchilla scaling law (Hoffmann et al., 2022) suggests that for a given compute budget, the optimal allocation is to train a model that is roughly 1/20th the number of tokens in its training set. Specifically, the optimal model size N and training tokens D should scale equally with compute budget C:

```
N_opt ~ C^0.5 D_opt ~ C^0.5
```

This means that GPT-3 (175B params, 300B tokens) was significantly undertrained relative to its size. A compute-optimal model would have been ~70B parameters trained on ~1.4T tokens - which is precisely what LLaMA was designed to be.

Compute Budget	Optimal Model Size	Optimal Training Tokens	Example Model
10^21 FLOPs	~400M	~8B tokens	BERT-large scale
10^22 FLOPs	~1B	~20B tokens	GPT-2 XL scale
10^23 FLOPs	~7B	~150B tokens	LLaMA-7B
10^24 FLOPs	~70B	~1.4T tokens	LLaMA-2 70B
10^25 FLOPs	~500B	~10T tokens	Frontier models

These scaling laws have profoundly influenced the direction of the field. Rather than simply building larger models, the focus has shifted to training smaller models for longer on higher-quality data. Models like Mistral 7B and Phi-2 demonstrate that a well-trained small model can match or exceed much larger models that were trained on less data or lower-quality data.

Glossary

Term	Definition
Attention	Mechanism computing a weighted sum of values based on query-key similarity
Autoregressive	Sequential generation where each token is predicted from preceding tokens
BPE	Byte-Pair Encoding; subword tokenisation algorithm merging frequent byte pairs
Causal Mask	Matrix preventing tokens from attending to future positions
CLS Token	Special token aggregating sequence-wide representation for classification
Cross-Attention	Attention where Q comes from one sequence and K,V from another
d_model	Primary hidden state dimension in a Transformer
Embedding	Continuous vector representation of discrete tokens
FFN	Feed-Forward Network; two-layer MLP in each Transformer block
Fine-tuning	Re-training a pre-trained model on a specific downstream task
Flash Attention	IO-aware exact attention avoiding n x n matrix materialisation in HBM
GQA	Grouped Query Attention; sharing KV heads across groups of query heads
KV-Cache	Caching K and V from previous steps to accelerate autoregressive decoding
Layer Norm	Feature-dimension normalisation for training stabilisation
LoRA	Low-Rank Adaptation; efficient fine-tuning via low-rank matrix decomposition
MHA	Multi-Head Attention; running multiple attention heads in parallel
MoE	Mixture of Experts; routing inputs to a subset of expert networks

Patch	A P x P image crop serving as a visual token in ViT
RoPE	Rotary Position Embedding; relative positional encoding via vector rotation
Self-Attention	Attention where Q, K, V all derive from the same sequence
Softmax	Function converting a score vector into a probability distribution
ViT	Vision Transformer; applying the Transformer architecture to images
Warmup	Gradually increasing learning rate at the start of training for stability

Quick Reference Card

Key Formulas

Concept	Formula
Scaled Dot-Product Attention	Attention(Q,K,V) = softmax(QK^T / sqrt(d_k)) * V
Multi-Head Attention	MultiHead = Concat(head_1, ..., head_h) * W_O
Sinusoidal Pos. Encoding	PE(pos,2i) = sin(pos / 10000^(2i/d))
Feed-Forward Network	FFN(x) = max(0, x*W1 + b1) * W2 + b2
Residual Connection	y = LN(x + Sublayer(x)) or y = x + Sublayer(LN(x))
ViT Patches	N = H*W / P^2, typically P = 16
Self-Attention Complexity	O(n^2 * d) time, O(n^2) memory
LoRA Update	W' = W + A*B, where r << min(d, k)
Layer Normalisation	LN(x) = gamma * (x - mu) / (sigma + eps) + beta
LR Warmup (Original)	lr = d^(-0.5) * min(step^(-0.5), step * warmup^(-1.5))

Popular Architectures

Model	Type	Params	Context	Key Feature
BERT-base	Encoder	110M	512	Bidirectional; MLM
GPT-2	Decoder	1.5B	1024	Autoregressive; Pre-LN
GPT-3	Decoder	175B	2048	In-context learning
T5-base	Enc-Dec	220M	512	Text-to-text framework
ViT-B/16	Encoder	86M	196+1	16x16 patches; CLS token
Swin-T	Encoder	28M	Varies	Shifted windows; hierarchical
LLaMA-2 7B	Decoder	7B	4096	RoPE; GQA; SwiGLU

Mistral 7B	Decoder	7B	8192	Sliding window; GQA
Mixtral 8x7B	Decoder	47B/13B	32768	MoE; 8 experts; top-2

References

[1] Vaswani, A., et al. (2017). "Attention Is All You Need." NeurIPS.

[2] Devlin, J., et al. (2019). "BERT: Pre-training of Deep Bidirectional Transformers." NAACL.

[3] Radford, A., et al. (2019). "Language Models are Unsupervised Multitask Learners." OpenAI.

[4] Brown, T., et al. (2020). "Language Models are Few-Shot Learners." NeurIPS.

[5] Dosovitskiy, A., et al. (2020). "An Image is Worth 16x16 Words." ICLR.

[6] Liu, Z., et al. (2021). "Swin Transformer: Hierarchical ViT using Shifted Windows." ICCV.

[7] Radford, A., et al. (2021). "Learning Transferable Visual Models (CLIP)." ICML.

[8] Caron, M., et al. (2021). "Emerging Properties in Self-Supervised ViTs (DINO)." ICCV.

[9] Hu, E., et al. (2021). "LoRA: Low-Rank Adaptation of Large Language Models." ICLR.

[10] Dao, T., et al. (2022). "FlashAttention: Fast and Memory-Efficient Exact Attention." NeurIPS.

[11] Liu, Z., et al. (2022). "A ConvNet for the 2020s (ConvNeXt)." CVPR.

[12] Touvron, H., et al. (2023). "LLaMA 2: Open Foundation and Fine-Tuned Chat Models." arXiv.

[13] Clark, K., et al. (2019). "What Does BERT Look At?" BlackboxNLP.

[14] Kaplan, J., et al. (2020). "Scaling Laws for Neural Language Models." arXiv.

[15] Hoffmann, J., et al. (2022). "Training Compute-Optimal LLMs (Chinchilla)." arXiv.

[16] Carion, N., et al. (2020). "End-to-End Object Detection with Transformers (DETR)." ECCV.

[17] Oquab, M., et al. (2023). "DINOv2: Learning Robust Visual Features." arXiv.

[18] Dettmers, T., et al. (2023). "QLoRA: Efficient Finetuning of Quantized LLMs." NeurIPS.

[19] Gu, A. & Dao, T. (2023). "Mamba: Linear-Time Sequence Modeling with Selective SSMs." arXiv.

[20] Beltagy, I., et al. (2020). "Longformer: The Long-Document Transformer." arXiv.

APPENDIX

*The following pages contain **50 Deep Learning Interview Questions** with Python Code Snippets, covering Transformer Foundations, Self-Attention, Multi-Head Attention, Encoder-Decoder, Vision Transformer (ViT), and CNN vs ViT & Hybrids.*

Appendix material compiled by Jekardah AI Labs

This document was produced by Jekardah AI Labs
Romi Nur Ismanto — rominur@gmail.com — Jekardah.com
2026 Edition — AI Engineering Handbook and Interview Preparation

PART II

Deep Dive: Extended Material

This section provides in-depth elaboration of every major topic covered in Part I. Each chapter is expanded with detailed mathematical derivations, worked numerical examples, comprehensive code implementations, architecture comparisons, and practical engineering insights.

Jekardah AI Labs — Romi Nur Ismanto — Jekardah.com — rominur@gmail.com

RNN/LSTM vs Transformer: A Comprehensive Comparison

1.1 The Recurrent Neural Network in Detail

Before understanding why the Transformer was such a breakthrough, we need to deeply understand what it replaced. A Recurrent Neural Network processes sequences by maintaining a hidden state vector that is updated at each time step. The fundamental recurrence relation is:

```
h_t = tanh(W_hh * h_(t-1) + W_xh * x_t + b_h)
y_t = W_hy * h_t + b_y
```

At each time step t, the hidden state h_t depends on both the current input x_t and the previous hidden state h_(t-1). This creates a strict sequential dependency chain: h_1 -> h_2 -> h_3 -> ... -> h_n. No step can be computed until all preceding steps are complete.

This sequential nature has profound implications for training efficiency. Consider a sequence of length n = 4096 (a moderately long document). On a modern GPU with thousands of parallel cores, the RNN can only use these cores for the matrix multiplications within a single step. The step-to-step dependency forces serial execution across the sequence dimension. The GPU sits largely idle, processing one thin slice of the sequence at a time.

The Vanishing Gradient Problem

The deeper issue with RNNs is the vanishing gradient problem. During backpropagation through time (BPTT), gradients are multiplied by the recurrent weight matrix W_hh at each step. For a sequence of length T, the gradient of the loss with respect to the hidden state at step t involves a product of T-t Jacobian matrices:

```
dL/dh_t = dL/dh_T * Product(dh_(k+1)/dh_k) for k = t to T-1
```

If the spectral radius (largest eigenvalue) of W_hh is less than 1, this product shrinks exponentially — gradients vanish, and the network cannot learn long-range dependencies. If the spectral radius is greater than 1, the product grows exponentially — gradients explode, causing numerical instability.

To illustrate concretely: if the dominant eigenvalue is 0.9, after 100 steps the gradient is attenuated by a factor of 0.9^100 = 2.66 x 10^-5. After 500 steps: 0.9^500 = 5.15 x 10^-23. The signal from distant tokens is effectively erased.

LSTM: A Partial Solution

The Long Short-Term Memory (LSTM) network introduced a cell state c_t that acts as a "highway" for gradient flow, controlled by three gates:

Gate	Formula	Purpose
Forget gate f_t	sigma(W_f . [h_(t-1), x_t] + b_f)	Decides what to erase from cell state
Input gate i_t	sigma(W_i . [h_(t-1), x_t] + b_i)	Decides what new info to store
Output gate o_t	sigma(W_o . [h_(t-1), x_t] + b_o)	Decides what to output from cell state

The cell state update follows: c_t = f_t * c_(t-1) + i_t * tanh(W_c . [h_(t-1), x_t] + b_c). When the forget gate is close to 1 and the input gate is close to 0, gradients flow through the cell state almost unattenuated. This allows LSTMs to remember information over hundreds of steps — but not thousands.

However, even LSTMs still process tokens sequentially. The hidden state at time t still depends on time t-1. No amount of gating can fix this fundamental serial bottleneck. This is the core limitation that the Transformer architecture was designed to overcome.

1.2 The Transformer's Fundamental Advantages

The Transformer replaces the entire recurrent computation with self-attention, which has fundamentally different computational properties:

Property	RNN/LSTM	Transformer
Parallelism	None across time steps; each step waits for previous	Full parallelism; all positions computed simultaneously
Path length (any two tokens)	O(n) — must traverse the entire chain	O(1) — direct connection via attention
Maximum gradient path length	O(n) — subject to vanishing gradients	O(1) — direct backprop through attention weights
Computational complexity	O(n * d^2) — linear in n, but serial	O(n^2 * d) — quadratic in n, but highly parallel
Memory complexity	O(n * d) — hidden states across time	O(n^2 + n * d) — attention matrix + embeddings
Inductive bias	Strong sequential bias; good for ordered data	No built-in ordering; needs positional encoding
Training speed (wall clock)	Slow — serial dependency limits GPU utilisation	Fast — matrix ops fully utilise GPU parallelism
Effective range	~200-500 tokens (LSTM) ~50-100 (vanilla RNN)	Full sequence length (128k+ with Flash Attention)

The trade-off is clear: the Transformer trades the RNN's O(n) sequential computation for O(n^2) parallel computation. On modern GPUs with massive parallelism, this is an overwhelming win. A single A100 GPU can process all n^2 attention entries simultaneously across its 6,912 CUDA cores, whereas an RNN would use those same cores for just one time step at a time.

1.3 Historical Timeline of Sequence Modelling

The evolution from simple RNNs to modern Transformers spans a decade of rapid innovation. Understanding this timeline helps contextualise each architecture's contribution:

Year	Milestone	Key Innovation	Impact
2013	Word2Vec (Mikolov et al.)	Distributed word representations via shallow neural nets	Foundation for all neural NLP
2014	Seq2Seq (Sutskever et al.)	Encoder-decoder with RNNs for translation	First neural machine translation
2014	Attention (Bahdanau et al.)	Additive attention over encoder hidden states	Solved fixed-length bottleneck
2015	GRU (Cho et al.)	Simplified gating (2 gates instead of LSTM's 3)	Efficient alternative to LSTM
2017	Transformer (Vaswani et al.)	Self-attention replaces recurrence entirely	Paradigm shift; foundation for all LLMs
2018	BERT (Devlin et al.)	Bidirectional pre-training with masked LM	Revolutionised NLU; transfer learning
2018	GPT (Radford et al.)	Autoregressive pre-training with decoder-only	Foundation for generative AI
2019	GPT-2 (Radford et al.)	1.5B params; zero-shot task generalisation	Demonstrated scaling benefits
2020	GPT-3 (Brown et al.)	175B params; in-context learning emerges	Few-shot learning without fine-tuning
2020	ViT (Dosovitskiy et al.)	Transformer applied directly to images	Unified architecture across modalities
2022	ChatGPT (OpenAI)	RLHF-aligned GPT-3.5; conversational AI	AI goes mainstream; public awareness
2023	LLaMA / Mistral	Open-weight efficient LLMs with GQA, RoPE, SwiGLU	Democratised access to large models
2024-25	GPT-4o, Claude 3.5+, Gemini	Multimodal native; reasoning advances	Frontier models; agentic capabilities

Key Takeaway: *The Transformer did not emerge in isolation. It built on a decade of work in attention mechanisms, sequence modelling, and distributed representations. What made it revolutionary was the complete elimination of recurrence — a bold architectural choice that unlocked unprecedented parallelism and scalability.*

Deep Dive 2

Attention Mathematics: Worked Examples

2.1 Numerical Walkthrough of Scaled Dot-Product Attention

Let us trace through a complete attention computation with concrete numbers. This is the kind of exercise frequently asked in technical interviews. We will use a tiny example with sequence length n = 3 and dimension d_k = 4.

Step 1: Define Q, K, V Matrices

Suppose after linear projections, we have the following (3 x 4) matrices for a single head:

```
# Q (queries), K (keys), V (values) - each (3, 4)
Q = [[1.0, 0.0, 1.0, 0.0],    # token 0 query
     [0.0, 1.0, 0.0, 1.0],    # token 1 query
     [1.0, 1.0, 0.0, 0.0]]    # token 2 query

K = [[1.0, 0.0, 0.0, 1.0],    # token 0 key
     [0.0, 1.0, 1.0, 0.0],    # token 1 key
     [1.0, 1.0, 1.0, 1.0]]    # token 2 key

V = [[1.0, 2.0, 3.0, 4.0],    # token 0 value
     [5.0, 6.0, 7.0, 8.0],    # token 1 value
     [9.0, 10., 11., 12.]]    # token 2 value
```

Step 2: Compute Q * K^T (Raw Scores)

Each element (i, j) is the dot product of query_i and key_j:

```
# Score matrix S = Q @ K^T  (3 x 3)
# S[0,0] = Q[0].K[0] = 1*1 + 0*0 + 1*0 + 0*1 = 1.0
# S[0,1] = Q[0].K[1] = 1*0 + 0*1 + 1*1 + 0*0 = 1.0
# S[0,2] = Q[0].K[2] = 1*1 + 0*1 + 1*1 + 0*1 = 2.0
# ...continuing for all 9 entries:

S = [[1.0, 1.0, 2.0],
     [1.0, 1.0, 2.0],
     [1.0, 1.0, 3.0]]
```

Step 3: Scale by 1/sqrt(d_k)

With d_k = 4, sqrt(d_k) = 2.0. Divide all scores by 2.0:

```
# Scaled scores: S / sqrt(4) = S / 2.0
S_scaled = [[0.50, 0.50, 1.00],
            [0.50, 0.50, 1.00],
            [0.50, 0.50, 1.50]]
```

Notice how scaling brings the values closer together. Without scaling, the difference between 1.0 and 3.0 would create a very "peaked" softmax. After scaling, the range is 0.5 to 1.5 — much more moderate.

Step 4: Apply Softmax (Row-wise)

Softmax is applied independently to each row. For row 0: exp([0.50, 0.50, 1.00]) = [1.649, 1.649, 2.718]. Sum = 6.016. Normalised: [0.274, 0.274, 0.452].

```
# Attention weights W = softmax(S_scaled, dim=-1)
# Row 0: softmax([0.50, 0.50, 1.00]) = [0.274, 0.274, 0.452]
# Row 1: softmax([0.50, 0.50, 1.00]) = [0.274, 0.274, 0.452]
# Row 2: softmax([0.50, 0.50, 1.50]) = [0.215, 0.215, 0.570]

W = [[0.274, 0.274, 0.452],
     [0.274, 0.274, 0.452],
     [0.215, 0.215, 0.570]]
```

Interpretation: Token 0 allocates 45.2% of its attention to token 2 (which had the highest score) and splits the remaining 54.8% equally between tokens 0 and 1. Token 2 focuses even more on itself (57.0%) because its self-score was the highest (1.50).

Step 5: Weighted Sum of Values

The output for each token is a weighted combination of all value vectors:

```
# Output = W @ V  (3 x 4)
# output[0] = 0.274 * V[0] + 0.274 * V[1] + 0.452 * V[2]
#           = 0.274 * [1,2,3,4] + 0.274 * [5,6,7,8] + 0.452 * [9,10,11,12]
#           = [0.274+1.370+4.068, 0.548+1.644+4.520, ...]
#           = [5.712, 6.712, 7.712, 8.712]

Output = [[5.712, 6.712, 7.712, 8.712],
          [5.712, 6.712, 7.712, 8.712],
          [6.416, 7.416, 8.416, 9.416]]
```

Each output vector is a blend of all input value vectors, weighted by the attention scores. Token 2's output is pulled more towards its own value (V[2]) because it attends most strongly to itself.

Interview Tip: *Being able to walk through this computation with concrete numbers demonstrates deep understanding. Practice computing 3x3 attention by hand until it becomes second nature. Interviewers are especially impressed when you can explain WHY each step exists, not just HOW to compute it.*

2.2 Understanding the Softmax Temperature Effect

The scaling factor 1/sqrt(d_k) can be generalised as a temperature parameter T that controls the "sharpness" of the attention distribution:

```
weights = softmax(scores / T)
```

Different temperature values produce dramatically different attention patterns:

Temperature T	Effect on Attention	When Used
T << 1 (e.g. 0.1)	Very sharp / peaked; almost one-hot distribution	Argmax approximation; high-confidence selection

T = 1 (standard)	Standard softmax; moderate distribution	Default for most applications
T = sqrt(d_k) (attention default)	Re-normalised; healthy gradient flow	Standard scaled dot-product attention
T >> 1 (e.g. 10.0)	Very flat / uniform; all tokens get similar weight	High exploration; smoothing effect

This is the same principle used in temperature-scaled sampling for text generation: lower temperatures make the output more deterministic (always choosing the highest-probability token), while higher temperatures make it more creative (sampling more diverse tokens).

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

scores = torch.tensor([[2.0, 1.0, 0.5, 0.1]])

for T in [0.1, 0.5, 1.0, 2.0, 5.0]:
    probs = F.softmax(scores / T, dim=-1)
    entropy = -(probs * probs.log()).sum().item()
    print(f"T={T:.1f}  probs={probs.numpy().round(3)}  entropy={entropy:.3f}")

# T=0.1  probs=[[1.000 0.000 0.000 0.000]]  entropy=0.000  (peaked)
# T=1.0  probs=[[0.506 0.186 0.113 0.076]]  entropy=1.159
# T=5.0  probs=[[0.301 0.260 0.237 0.225]]  entropy=1.372  (flat)
```

2.3 Why Dot-Product and Not Other Similarity Functions?

The original Transformer chose dot-product attention over alternatives like additive (Bahdanau) attention. Here is a detailed comparison:

Similarity Function	Formula	Complexity	Pros / Cons
Dot-product (Luong, 2015)	q^T k	O(d)	Fast; parallelisable; scales well with dim
Scaled dot-product (Vaswani, 2017)	q^T k / sqrt(d_k)	O(d)	Same + prevents softmax saturation
Additive (Bahdanau, 2014)	v^T tanh(W_1 q + W_2 k)	O(d)	More expressive but slower; extra parameters
Cosine similarity	q^T k / (||q|| ||k||)	O(d)	Normalised; bounded [-1,1]; less common in practice
Bilinear	q^T W k	O(d^2)	Most expressive; but expensive W matrix

In practice, dot-product and scaled dot-product are preferred because they can be computed as a single matrix multiplication (Q @ K^T), which GPUs execute extremely efficiently. Additive attention requires a non-linearity (tanh) inside the loop, which is harder to parallelise.

Deep Dive 3

Multi-Head Attention: Analysis & Variants

3.1 Why Multiple Heads Are Better: Subspace Theory

The mathematical intuition behind multi-head attention is rooted in subspace learning. A single attention head with d_k = d_model projects all of Q, K, V into one d_model-dimensional space. The attention mechanism can only capture one type of relationship in this single space.

With h = 8 heads and d_k = d_model/8 = 64, each head independently projects into a separate 64-dimensional subspace. Each subspace can learn to capture a different type of relationship:

```
# Visualising what different heads learn (conceptual)
# Head 0: Learns syntactic dependencies (subject -> verb)
# Head 1: Learns positional patterns (attend to previous token)
# Head 2: Learns coreference (pronoun -> antecedent)
# Head 3: Learns semantic similarity (synonym clusters)
# Head 4: Learns delimiter tracking ([SEP], periods)
# Head 5: Learns relative position (attend to nearby tokens)
# Head 6: Learns rare/important token detection
# Head 7: Learns broad context aggregation

# After concatenation + W_O projection, the model combines
# all 8 perspectives into a single rich representation.
```

The output projection W_O is crucial here. It mixes information across heads, allowing the model to learn how to weight and combine these different perspectives. Without W_O, the heads would remain isolated channels that cannot interact.

3.2 Head Pruning: Are All Heads Necessary?

Research by Michel et al. (2019) showed that many attention heads can be removed after training with minimal performance degradation. Key findings:

- In BERT-base (12 layers x 12 heads = 144 total heads), removing 20-40% of heads causes less than 1% accuracy drop on most tasks.
- Some heads are consistently important across tasks, while others are task-specific or redundant.
- The most important heads tend to be in the middle layers, not the first or last layers.
- Head importance varies significantly by layer — some layers need all heads, while others function well with just 2-3.

This has practical implications for model compression and inference optimisation. If you can identify and remove redundant heads, you reduce both memory and computation without sacrificing quality.

Pruning Level	Heads Remaining	GLUE Score Impact	Speedup
0% (baseline)	144 / 144	Baseline (100%)	1.0x
20% pruned	115 / 144	-0.2% to -0.5%	1.15x
40% pruned	86 / 144	-0.5% to -1.5%	1.35x
60% pruned	58 / 144	-1.5% to -4.0%	1.60x
80% pruned	29 / 144	-5% to -15%	2.0x

3.3 GQA, MQA, and MLA — The KV-Cache Memory Crisis

As LLMs scale to longer contexts (32k, 128k, 1M tokens), the KV-cache becomes the dominant memory bottleneck during inference. Let us compute the exact numbers for a concrete model:

KV-Cache Memory Calculation

For a model with L layers, h_kv KV heads, d_k dimension per head, sequence length n, batch size B, in BF16 (2 bytes per element):

```
KV-Cache = 2 * B * L * h_kv * d_k * n * 2 bytes
```

Let us compute this for several models at context length n = 32,768:

Model	L	h_kv	d_k	Attention Type	KV-Cache (32k, B=1)
GPT-3 175B	96	96	128	Full MHA	96 * 96 * 128 * 32768 * 4 = 154.1 GB
LLaMA-2 7B	32	32	128	Full MHA	32 * 32 * 128 * 32768 * 4 = 17.2 GB
LLaMA-2 70B	80	8	128	GQA (g=8)	80 * 8 * 128 * 32768 * 4 = 10.7 GB
Mistral 7B	32	8	128	GQA (g=8)	32 * 8 * 128 * 32768 * 4 = 4.3 GB
Multi-Query (hypothetical)	32	1	128	MQA	32 * 1 * 128 * 32768 * 4 = 0.5 GB

The difference is staggering. GPT-3 with full MHA needs 154 GB of KV-cache at 32k context — more than the model weights themselves. LLaMA-2 70B with GQA reduces this to 10.7 GB, enabling practical deployment. Mistral 7B with its sliding window attention further reduces effective KV-cache by limiting the window to 4096 tokens.

Engineering Insight: *KV-cache memory scales linearly with both batch size and sequence length. For a serving system handling 100 concurrent requests at 32k context, multiply the per-request KV-cache by 100. This is why GQA and MQA are not merely academic optimisations — they are essential for production deployment.*

3.4 Multi-Latent Attention (MLA) — DeepSeek's Innovation

DeepSeek-V2 introduced Multi-Latent Attention (MLA), which compresses both K and V into a joint low-rank latent space before caching. Instead of caching full K and V tensors, MLA caches a compressed representation:

```
c_t = W_DKV * x_t (d_c << h_kv * d_k)
K_t, V_t = W_UK * c_t, W_UV * c_t (reconstructed for attention)
```

This reduces KV-cache by up to 93% compared to full MHA while maintaining quality. The compression ratio is determined by d_c / (h_kv * d_k), which can be as low as 0.07.

Deep Dive 4

Positional Encoding: From Sinusoidal to RoPE

4.1 Sinusoidal Encoding: Mathematical Properties

The original sinusoidal positional encoding has several elegant mathematical properties that are frequently asked about in interviews:

Property 1: Unique Encoding per Position

Each position gets a unique d_model-dimensional vector. The different frequencies across dimensions ensure no two positions share the same encoding. Low-frequency components (small i) vary slowly across positions, encoding coarse position information. High-frequency components (large i) vary rapidly, encoding fine-grained position information.

Property 2: Relative Position via Linear Transformation

For any fixed offset k, there exists a linear transformation M_k such that PE(pos + k) = M_k * PE(pos). This means the model can learn to attend to relative positions through learned weight matrices. The transformation is a rotation in 2D subspaces, which is the mathematical foundation for RoPE.

Property 3: Bounded Magnitude

Since sin and cos are bounded in [-1, 1], the positional encoding does not dominate the token embedding regardless of position. This is important because the two are added together — if PE grew unboundedly, it would overwhelm the semantic content of the token embedding for large positions.

Property 4: Generalisation to Unseen Lengths

Since the sinusoidal functions are defined for any real number, the encoding naturally extends to positions beyond those seen during training. This contrasts with learned positional embeddings which are limited to the maximum training length.

```
import torch, math, matplotlib.pyplot as plt

def sinusoidal_pe(max_len, d_model):
    pe = torch.zeros(max_len, d_model)
    pos = torch.arange(max_len).unsqueeze(1).float()
    freqs = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000) / d_model))
    pe[:, 0::2] = torch.sin(pos * freqs)
    pe[:, 1::2] = torch.cos(pos * freqs)
    return pe

pe = sinusoidal_pe(128, 64)
# Visualise: low dims change slowly, high dims change rapidly
# plt.imshow(pe.numpy(), aspect='auto', cmap='RdBu')
# plt.xlabel('Dimension'); plt.ylabel('Position')

# Verify relative position property:
# dot product PE[pos].PE[pos+k] depends only on k, not pos
for k in [1, 5, 10]:
    dots = [pe[p] @ pe[p+k] for p in range(50)]
    print(f"k={k}: dot products are constant = {dots[0]:.3f}")
```

4.2 Learned Positional Embeddings

BERT, GPT-2, and ViT use learned positional embeddings — a simple trainable parameter of shape (max_seq_len, d_model). Each position gets its own independently learned vector.

Aspect	Sinusoidal (Fixed)	Learned
Parameters	Zero additional	max_len * d_model (e.g. 512 * 768 = 393k)
Generalisation	Extrapolates to unseen lengths	Cannot extrapolate; bounded by max_len
Relative position	Encodes via linear transformation	Must be learned implicitly
Performance	Slightly lower on some benchmarks	Slightly better when data is sufficient
Used by	Original Transformer	BERT, GPT-2, ViT

4.3 Rotary Position Embeddings (RoPE)

RoPE (Su et al., 2021), used in LLaMA, Mistral, and most modern LLMs, represents the most important advancement in positional encoding since the original Transformer. RoPE applies a position-dependent rotation to the query and key vectors:

`f(q, m) = R(m) * q, f(k, n) = R(n) * k`

`where R(m) is a block-diagonal rotation matrix with angle m * theta_i`

The key insight is that when computing the dot product between rotated q and k, the rotation angles subtract:

`f(q, m)^T * f(k, n) = q^T * R(n-m) * k`

This means the attention score naturally depends on the relative distance (n - m) between tokens, not their absolute positions. RoPE achieves this without any additional parameters — the rotation is applied as a simple element-wise operation.

RoPE also enables length extrapolation through frequency scaling techniques. The base frequency theta (default 10000) can be adjusted to extend effective context length. NTK-aware scaling multiplies theta by a factor related to the desired extension, while YaRN applies different scaling to different frequency bands.

```
import torch

def precompute_rope_freqs(dim, max_len, theta=10000.0):
    """Precompute RoPE frequency tensor."""
    freqs = 1.0 / (theta ** (torch.arange(0, dim, 2).float() / dim))
    positions = torch.arange(max_len).float()
    angles = torch.outer(positions, freqs)  # (max_len, dim/2)
    # Stack cos and sin for complex rotation
    return torch.cos(angles), torch.sin(angles)

def apply_rope(x, cos_freqs, sin_freqs):
    """Apply rotary embeddings to input tensor."""
    # x shape: (batch, seq_len, num_heads, head_dim)
    x1 = x[..., 0::2]  # even dimensions
    x2 = x[..., 1::2]  # odd dimensions
    # Complex rotation: (a + bi)(cos + i*sin) = (a*cos - b*sin) + i(a*sin + b*cos)
    rotated_x1 = x1 * cos_freqs - x2 * sin_freqs
    rotated_x2 = x1 * sin_freqs + x2 * cos_freqs
    return torch.stack([rotated_x1, rotated_x2], dim=-1).flatten(-2)

cos_f, sin_f = precompute_rope_freqs(64, 4096)
print(f"Frequency table shape: {cos_f.shape}")  # (4096, 32)
```

4.4 ALiBi: Attention with Linear Biases

ALiBi (Press et al., 2022) takes the simplest possible approach to position encoding: it adds a static, non-learned linear penalty to attention scores based on the distance between query and key positions:

attention_score(i, j) = q_i^T * k_j - m * |i - j|

Where m is a fixed slope that varies per head. With 8 heads, the slopes are: 1/1, 1/2, 1/4, 1/8, 1/16, 1/32, 1/64, 1/128. Closer tokens always receive higher scores than distant tokens, but the penalty rate differs across heads — some heads have a "long view" (small m) and others have a "short view" (large m).

ALiBi's main advantage is zero-shot length extrapolation. A model trained with ALiBi on 1024-token sequences can be evaluated on 2048 or 4096 tokens without any fine-tuning or interpolation, because the linear bias naturally extends to any distance. This makes it uniquely practical for deployment scenarios where the exact context length is not known in advance.

Method	Params	Relative Pos	Extrapolation	Used In
Sinusoidal	0	Via linear transform	Good (smooth)	Original Transformer
Learned	max_len * d	Implicit	None	BERT, GPT-2, ViT
RoPE	0	Native (rotation)	With scaling tricks	LLaMA, Mistral, Qwen
ALiBi	0	Linear bias	Excellent (zero-shot)	BLOOM, MPT
xPos	0	RoPE + decay	Good	Some research models

Deep Dive 5

Feed-Forward Networks as Key-Value Memories

5.1 The FFN Memory Hypothesis

Recent research (Geva et al., 2021; Meng et al., 2022) has revealed a fascinating interpretation of the position-wise FFN: it functions as a differentiable key-value memory store. The first weight matrix W1 acts as "keys" that match input patterns, and the second weight matrix W2 acts as "values" that store the associated knowledge.

```
FFN(x) = ReLU(x * W1) * W2
= sum_i [ ReLU(x . w1_i) * w2_i ]
```

Each row w1_i of W1 is a "key" — a pattern detector that fires when the input x is similar to it (positive dot product after ReLU). Each column w2_i of W2 is the corresponding "value" — the information that is retrieved when that key matches. The output is a weighted sum of all values, where the weights are determined by how well each key matches the input.

This interpretation explains several important observations about large language models:

- **Larger FFNs store more knowledge:** GPT-3 uses d_ff = 4 * d_model = 49,152. This means 49,152 key-value pairs per layer x 96 layers = 4.7 million memory slots. This is why larger models know more facts.
- **Knowledge localisation:** Specific factual associations (e.g. "The Eiffel Tower is in Paris") are stored in specific FFN neurons in specific layers. Meng et al. showed that editing these neurons can change the model's factual knowledge.
- **FFN dominates parameters:** In most Transformers, FFN parameters outnumber attention parameters by ~2:1. For BERT-base: FFN = 4.72M/layer vs attention = 2.36M/layer. The FFN is where the bulk of the model's knowledge resides.

5.2 FFN Activation Functions: ReLU, GELU, SwiGLU

The choice of activation function in the FFN significantly impacts model quality. The evolution from ReLU to SwiGLU reflects our growing understanding of what makes Transformers work well:

Activation	Formula	Properties	Used In
ReLU	max(0, x)	Simple; sparse activations; zero gradient for x < 0	Original Transformer

GELU	x * Phi(x) (Gaussian CDF)	Smooth approximation of ReLU; better gradient flow	BERT, GPT-2, ViT
SiLU / Swish	x * sigmoid(x)	Smooth; non-monotonic; self-gated	EfficientNet, some LLMs
SwiGLU	SiLU(x W1) * (x W_gate) then * W2	Gated linear unit with Swish; best quality in practice	LLaMA, PaLM, Mistral

SwiGLU uses an extra weight matrix W_gate, increasing FFN parameters by 50% (three weight matrices instead of two). To maintain the same total parameter count, the hidden dimension is reduced from 4 * d_model to (8/3) * d_model. Despite having fewer "memory slots," SwiGLU models consistently outperform GELU models of the same size, suggesting that the gating mechanism allows more efficient use of each slot.

```
import torch, torch.nn as nn

class SwiGLU_FFN(nn.Module):
    """FFN with SwiGLU activation as used in LLaMA."""
    def __init__(self, d_model, d_ff=None):
        super().__init__()
        # LLaMA uses 8/3 * d_model, rounded to nearest multiple of 256
        d_ff = d_ff or int(2 * (4 * d_model) / 3)
        d_ff = 256 * ((d_ff + 255) // 256)  # round up
        self.w1 = nn.Linear(d_model, d_ff, bias=False)     # gate projection
        self.w2 = nn.Linear(d_ff, d_model, bias=False)     # down projection
        self.w3 = nn.Linear(d_model, d_ff, bias=False)     # up projection

    def forward(self, x):
        return self.w2(nn.functional.silu(self.w1(x)) * self.w3(x))

ffn = SwiGLU_FFN(4096)
n_params = sum(p.numel() for p in ffn.parameters())
print(f"SwiGLU FFN params: {n_params:,}")  # ~3x more matrices but d_ff reduced
```

Deep Dive 6

Encoder-Decoder: Complete Inference Pipeline

6.1 Step-by-Step Translation Example

Let us trace through a complete machine translation example from English to French using an encoder-decoder Transformer. This illustrates how all components work together in practice.

Phase 1: Tokenisation & Embedding

Input sentence: "The cat sat on the mat." After tokenisation with BPE: ["The", "_cat", "_sat", "_on", "_the", "_mat", "."] (7 tokens). Each token is mapped to a d_model-dimensional embedding and positional encodings are added.

Phase 2: Encoder Forward Pass

The 7 embedded tokens are processed through all N encoder layers simultaneously (full parallelism). Each layer applies multi-head self-attention (every token attends to every other) followed by FFN. The output is a tensor of shape (1, 7, d_model) — the encoder memory Z.

Key point: the encoder processes ALL 7 tokens at once. There is no sequential dependency within the encoder. On a GPU, all attention scores for all heads across all layers can be computed in parallel.

Phase 3: Decoder Autoregressive Generation

The decoder generates one token at a time:

```
# Step-by-step decoder generation (conceptual)

# Step 0: Start with [BOS]
decoder_input = ["<BOS>"]
encoder_memory = encoder("The cat sat on the mat.")  # computed once

# Step 1:
#   Masked self-attention: [BOS] attends only to itself
#   Cross-attention: [BOS] attends to all 7 encoder tokens
#   FFN + softmax -> predicts "Le"
decoder_input = ["<BOS>", "Le"]

# Step 2:
#   Masked self-attention: "Le" attends to [BOS] and itself
#   Cross-attention: "Le" attends to all 7 encoder tokens
#   FFN + softmax -> predicts "chat"
decoder_input = ["<BOS>", "Le", "chat"]

# Step 3: -> "est"  (using KV-cache, only new token computed)
# Step 4: -> "assis"
# Step 5: -> "sur"
# Step 6: -> "le"
# Step 7: -> "tapis"
# Step 8: -> "."
# Step 9: -> "<EOS>"  (generation stops)

# Output: "Le chat est assis sur le tapis."
```

Notice: the encoder memory is computed ONCE and reused at every decoder step. The decoder's masked self-attention grows by one token per step (using KV-cache to avoid recomputation). Cross-attention always looks at all 7 encoder tokens regardless of the current decoder step.

6.2 Beam Search: A Detailed Walkthrough

Beam search with beam width k = 3 for the same translation:

```
# Beam Search with k=3

# Step 1: Top-3 first tokens
# P("Le")   = 0.45
# P("Un")   = 0.25
# P("Chat") = 0.15
# Beams: [("Le", -0.80), ("Un", -1.39), ("Chat", -1.90)]

# Step 2: Expand each beam by top-3, keep overall top-3
# "Le chat"     = -0.80 + (-0.22) = -1.02
# "Le petit"    = -0.80 + (-1.10) = -1.90
# "Un chat"     = -1.39 + (-0.35) = -1.74
# "Le matou"    = -0.80 + (-2.05) = -2.85
# "Chat est"    = -1.90 + (-0.90) = -2.80
# ...
# Keep top-3: [("Le chat", -1.02), ("Un chat", -1.74), ("Le petit", -1.90)]

# Continue until all beams produce <EOS>
# Final: "Le chat est assis sur le tapis." (highest cumulative log-prob)
```

Beam search finds better translations than greedy decoding because it considers multiple hypotheses simultaneously. However, it is k times slower (k forward passes per step instead of 1). For real-time applications, greedy or nucleus sampling is often preferred.

6.3 Sampling Strategies Compared

Modern LLMs use various sampling strategies during inference. Each produces different output characteristics:

Strategy	Mechanism	Diversity	Quality	Use Case
Greedy	Always pick argmax token	None (deterministic)	Often suboptimal; repetitive	Simple tasks; testing
Beam Search (k=4-5)	Keep top-k hypotheses	Low (k hypotheses)	Best for translation	Machine translation; structured output

Temperature sampling	Scale logits by T before softmax	Tunable (T controls)	Depends on T; T~0.7 is good	Creative writing; chat
Top-k sampling	Sample from top k tokens only	Moderate (fixed k)	Can be incoherent if k too large	General generation; k=40-50 typical
Top-p (Nucleus)	Sample from smallest set with cumprob >= p	Adaptive (varies per step)	High quality; adaptive vocab	Default for most LLMs; p=0.9-0.95
Min-p	Keep tokens with prob >= p_min * max_prob	Adaptive (relative to peak)	Good quality; robust	Emerging default; p_min=0.05-0.1

Deep Dive 7

ViT: Training Recipes & Deployment Guide

7.1 Data Augmentation for ViT

ViT lacks the inductive biases of CNNs, making it much more dependent on data augmentation for regularisation. The DeiT training recipe (Touvron et al., 2021) showed that with the right augmentation, ViT can achieve competitive results on ImageNet-1K alone, without the massive JFT-300M dataset.

The standard augmentation pipeline for ViT training:

Technique	Description	Strength	When Applied
RandAugment	Random sequence of image transforms (rotation, colour, sharpness, etc.)	n=2, m=9 (2 ops, magnitude 9)	Every batch
Mixup	Linear interpolation of two training images and their labels	alpha=0.8 (Beta distribution)	Every batch
CutMix	Replace a rectangular patch of one image with another; mix labels	alpha=1.0 (patch size varies)	Every batch
Random Erasing	Randomly erase a rectangular region of the image	prob=0.25 (25% of images)	Every batch
Color Jitter	Random changes to brightness, contrast, saturation, hue	0.3, 0.3, 0.3 (moderate)	Every batch
Random Resized Crop	Crop a random portion and resize to target	scale=(0.08, 1.0) ratio=(3/4, 4/3)	Every batch
Horizontal Flip	Random left-right mirror	prob=0.5	Every batch

Without these augmentations, ViT-B/16 achieves only ~76% top-1 accuracy on ImageNet-1K (vs ~80% for a comparable ResNet). With the full DeiT recipe, ViT-B/16 reaches ~81.8%, competitive with the best CNNs trained with similar compute.

7.2 Regularisation Techniques for ViT

In addition to data augmentation, ViT requires careful regularisation to prevent overfitting:

Technique	Description	Typical Value	Effect
Stochastic Depth (DropPath)	Randomly skip entire Transformer blocks during training	drop_rate=0.1 (linear schedule 0 to 0.1 across depth)	Most important regulariser for ViT; prevents co-adaptation
Label Smoothing	Replace hard labels [0, 1] with soft labels [eps/C, 1-eps]	eps=0.1	Prevents overconfident predictions; improves calibration
Weight Decay	L2 penalty on weights; exclude bias and LayerNorm params	0.05	Standard regularisation; acts as prior
Dropout	Random zeroing of activations and attention weights	0.0 (often not used) or 0.1 for small data	Less important for ViT than for CNNs; DropPath preferred
EMA (Exponential Moving Average)	Maintain a running average of model weights for evaluation	decay=0.9999	Stabilises evaluation; small but consistent improvement

7.3 ViT for Object Detection: DETR

DETR (Carion et al., 2020) was the first fully end-to-end object detection framework using Transformers, eliminating traditional hand-crafted components like anchor boxes, non-maximum suppression (NMS), and region proposals. DETR treats detection as a set prediction problem:

- **CNN backbone** extracts feature maps from the image (e.g. ResNet-50 produces 2048 x 7 x 7 features).
- **Transformer encoder** processes the flattened feature map tokens with self-attention to capture global context.
- **Transformer decoder** uses a fixed set of N learned "object queries" (typically N = 100) that attend to the encoder output via cross-attention.
- **Prediction heads** (FFN) predict class labels and bounding boxes for each query.
- **Hungarian matching** finds the optimal one-to-one assignment between predictions and ground-truth objects during training, using a bipartite matching cost that combines classification and box regression losses.

DETR's simplicity is remarkable: the entire detection pipeline is a single forward pass through backbone + Transformer + FFN heads. No anchors, no NMS, no hand-tuned IoU thresholds. However, DETR trains slowly (500 epochs vs ~36 for Faster R-CNN) and struggles with small objects due to the fixed low-resolution feature map. DINO-DETR and RT-DETR address these limitations.

Modern LLM Architecture: A Complete Blueprint

8.1 LLaMA Architecture Deep Dive

LLaMA (Touvron et al., 2023) represents the most influential open-weight LLM architecture. Understanding its design choices is essential for any modern deep learning practitioner. LLaMA-2 7B consists of:

Component	Configuration	Design Choice	Why

Embedding	d_model = 4096
	vocab = 32,000
		Learned embedding
		(no bias)
			BPE tokeniser;
			compact vocabulary

Attention	32 heads, d_k = 128 32 KV heads (7B) 8 KV heads (70B)	GQA for 70B; full MHA for 7B	KV-cache reduction for large models
Position	RoPE theta = 10,000	Rotary embeddings applied to Q and K	Relative position; length extrapolation
FFN	d_ff = 11,008 SwiGLU activation	3 weight matrices (8/3 * d_model)	Better quality than GELU FFN
Normalisation	RMSNorm (Pre-Norm)	Simplified LayerNorm; no centering	Faster compute; stable training
Layers	32 layers (7B) 80 layers (70B)	Deep stack with residual connections	Capacity scaling
Output	Weight-tied to embedding	Shared embedding and output matrix	Parameter saving; better generalisation

8.2 RMSNorm: Simplified Layer Normalisation

LLaMA uses RMSNorm instead of standard LayerNorm. RMSNorm removes the mean-centering step, computing only the root mean square normalisation:

```
RMSNorm(x) = x / RMS(x) * gamma
```

```
RMS(x) = sqrt( (1/d) * sum(x_i^2) )
```

This is faster than standard LayerNorm because it avoids computing the mean and does not need the beta (bias) parameter. Empirically, the centering step provides little benefit for Transformers, making

RMSNorm a pure efficiency win with no quality loss.

```
class RMSNorm(nn.Module):
    def __init__(self, dim, eps=1e-6):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))

    def forward(self, x):
        rms = torch.sqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)
        return x / rms * self.weight

# Compare parameter count
ln = nn.LayerNorm(4096)    # 2 * 4096 = 8192 params (gamma + beta)
rn = RMSNorm(4096)         # 1 * 4096 = 4096 params (gamma only)
print(f"LayerNorm: {sum(p.numel() for p in ln.parameters()):,}")
print(f"RMSNorm:   {sum(p.numel() for p in rn.parameters()):,}")
```

8.3 Training a Modern LLM: The Complete Pipeline

Training a state-of-the-art LLM involves three main phases, each with distinct objectives and techniques:

Phase 1: Pre-training (Most Expensive)

Objective: Learn general language understanding and world knowledge from a massive text corpus. The model is trained with autoregressive language modelling (predict the next token). Typical scale: 1-15 trillion tokens, weeks to months of training on hundreds to thousands of GPUs.

Key decisions during pre-training:

- **Data mixture:** Carefully curated blend of web text (CommonCrawl, C4), books, Wikipedia, code (GitHub, StackOverflow), scientific papers, and curated datasets. The ratio matters enormously — too much web text produces verbose, low-quality outputs; too much code makes the model overly technical.
- **Data quality filtering:** Perplexity-based filtering, deduplication, toxic content removal, PII scrubbing. LLaMA used a quality classifier trained on Wikipedia/books to score web pages.
- **Tokeniser:** BPE with byte-level fallback, vocabulary size 32k-128k. Larger vocabularies compress text better but increase embedding parameters.
- **Context length:** 2048-4096 tokens for initial training, extended to 32k-128k through continued pre-training with RoPE scaling.

Phase 2: Supervised Fine-Tuning (SFT)

Objective: Teach the model to follow instructions and produce helpful responses. The model is fine-tuned on a curated dataset of (instruction, response) pairs. Typical scale: 10k-100k high-quality examples, hours of training on a few GPUs.

SFT transforms the base model from a next-token predictor into an instruction-following assistant. Without SFT, a prompt like "Explain quantum computing" would receive a completion that continues the sentence rather than an explanatory response.

Phase 3: Alignment (RLHF / DPO)

Objective: Align the model's outputs with human preferences for helpfulness, harmlessness, and honesty. Two main approaches:

- **RLHF (Reinforcement Learning from Human Feedback):** Train a reward model on human preference data, then optimise the LLM policy to maximise the reward using PPO. Complex but well-established.
- **DPO (Direct Preference Optimisation):** Skip the reward model; directly optimise the LLM on preference pairs using a contrastive loss. Simpler, more stable, increasingly preferred.

Deep Dive 9

Inference Optimisation: From Model to Production

9.1 Quantisation Deep Dive

Quantisation reduces the numerical precision of model weights from floating point (FP32 or FP16) to lower-bit integers (INT8, INT4), dramatically reducing memory usage and increasing throughput.

Post-Training Quantisation (PTQ)

PTQ quantises a pre-trained model without additional training. The key challenge is choosing appropriate scale factors and zero points for each weight tensor. Two leading methods:

- **GPTQ (Frantar et al., 2022):** Uses approximate second-order information (Hessian) to find optimal quantisation parameters that minimise reconstruction error layer by layer. Achieves excellent 4-bit results but requires a calibration dataset of ~128 samples.
- **AWQ (Lin et al., 2023):** Observes that a small fraction of weights ("salient channels") disproportionately affect output quality. AWQ protects these channels by per-channel scaling before quantisation, achieving better quality than GPTQ at the same bit width.

Method	Bits	Model Size (LLaMA-7B)	Quality (vs FP16)	Speed (tokens/s)	Best For
FP16	16	13.5 GB	Baseline	1.0x	Training; high quality
INT8 (LLM.int8())	8	7.0 GB	-0.1%	1.2x	Easy deployment; minimal loss
INT4 (GPTQ)	4	3.6 GB	-1.0%	1.8x	Memory-limited deployment
INT4 (AWQ)	4	3.6 GB	-0.5%	2.0x	Best quality at 4-bit
INT3 (GPTQ)	3	2.7 GB	-3.0%	2.2x	Extreme compression
GGUF Q4_K_M	4 mixed	4.1 GB	-0.8%	Variable	CPU inference; llama.cpp

9.2 Speculative Decoding

Speculative decoding (Leviathan et al., 2022) uses a small, fast "draft" model to generate k candidate tokens, which are then verified in parallel by the large "target" model. The insight is that the target model can check k tokens in roughly the same time as generating 1 token (because the verification is a single forward pass over k+1 tokens).

Algorithm outline:

- 1. Draft model generates k tokens autoregressively (fast, ~10x cheaper per token).
- 2. Target model runs a single forward pass on all k+1 tokens (prefix + k drafts).
- 3. Compare draft and target distributions at each position using rejection sampling.
- 4. Accept draft tokens where target agrees; reject and resample from first disagreement.

Expected speedup depends on how often the draft model's predictions match the target model. With a well-matched draft model (e.g. LLaMA-7B drafting for LLaMA-70B), acceptance rates of 70-85% are common, yielding 2-3x overall speedup with mathematically identical output distribution.

9.3 Continuous Batching & Serving Infrastructure

Efficient LLM serving requires specialised infrastructure that goes far beyond simple batch processing. Key techniques used by production serving systems like vLLM and TensorRT-LLM:

Technique	Description	Benefit
Continuous Batching	New requests join the batch as soon as a slot opens; no waiting for batch completion	2-3x throughput vs static batching
PagedAttention (vLLM)	KV-cache stored in non-contiguous memory pages; allocation on demand	Eliminates KV-cache fragmentation; 4-24x throughput
Prefix Caching	Cache KV for common prefixes (system prompts) across requests	50%+ latency reduction for shared-prefix workloads
Tensor Parallelism	Split model across multiple GPUs; each GPU holds a shard	Enables serving models larger than 1 GPU memory
Speculative Decoding	Small model drafts; large model verifies	2-3x generation speedup

Production Reality: *A well-optimised vLLM deployment of LLaMA-2 70B (INT4) on 4x A100 80GB can serve ~200 requests per minute at 2048 output tokens each. Cost: ~$0.001-0.003 per 1000 tokens. Comparatively, a naive PyTorch implementation would serve perhaps 5-10 requests per minute on the same hardware.*

Deep Dive 10

Multimodal Transformers & Agentic AI

10.1 How Vision-Language Models Work

Modern multimodal models (GPT-4V, Claude, LLaVA) combine a vision encoder with a language model decoder. The typical architecture has three components:

- **Vision Encoder:** A pre-trained ViT or CLIP vision encoder processes the image into a sequence of visual tokens. For CLIP ViT-L/14 with 224x224 input: 256 patch tokens of dimension 1024.
- **Projection Module:** A learned linear layer or small MLP projects visual tokens from the vision encoder's dimension to the LLM's dimension. This "bridges" the two modalities.
- **Language Model:** A standard decoder-only LLM (e.g. LLaMA, Vicuna) processes the concatenation of visual tokens and text tokens. The LLM treats visual tokens exactly like text tokens — they participate in self-attention and influence generation.

```
# Simplified VLM forward pass (LLaVA-style)
class VisionLanguageModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.vision_encoder = CLIPVisionModel()   # frozen
        self.projection = nn.Linear(1024, 4096)   # trainable
        self.llm = LLaMAModel()                   # fine-tuned

    def forward(self, image, text_tokens):
        # Encode image into visual tokens
        visual_features = self.vision_encoder(image)     # (B, 256, 1024)
        visual_tokens = self.projection(visual_features) # (B, 256, 4096)

        # Embed text tokens
        text_embeds = self.llm.embed(text_tokens)        # (B, T, 4096)

        # Concatenate visual + text tokens
        combined = torch.cat([visual_tokens, text_embeds], dim=1)

        # Process through LLM (visual tokens = "prefix")
        output = self.llm.forward(combined)
        return output  # generate text conditioned on image
```

This architecture is surprisingly simple yet powerful. The key insight is that a well-pre-trained LLM already has strong reasoning capabilities — the visual encoder and projection just need to "translate" images into a language the LLM already understands.

10.2 CLIP: Contrastive Learning at Scale

CLIP (Radford et al., 2021) trains paired image and text encoders to maximise the cosine similarity of matching image-text pairs while minimising similarity for non-matching pairs. Trained on 400M image-text pairs from the internet.

CLIP's contrastive loss for a batch of N image-text pairs:

```
L = -1/N * sum_i [ log( exp(sim(I_i, T_i)/tau) / sum_j exp(sim(I_i,
                          T_j)/tau) ) ]
```

Where sim(I, T) is the cosine similarity between the image embedding and text embedding, and tau is a learned temperature parameter. This creates a shared embedding space where images and text with similar semantics are close together.

CLIP enables remarkable zero-shot capabilities:

- **Zero-shot classification:** Encode class names as text ("a photo of a dog", "a photo of a cat"), encode the image, and pick the class with highest cosine similarity. No task-specific training needed.
- **Image-text retrieval:** Given a text query, find the most similar images in a database (or vice versa) using embedding similarity search.
- **Visual grounding:** CLIP features serve as the foundation for text-guided image segmentation, generation, and editing (used by DALL-E 2, Stable Diffusion).

10.3 Agentic AI: Tool Use & Reasoning

The latest frontier in Transformer applications is agentic AI — LLMs that can plan, reason, and take actions in the world by using tools. The key architectural patterns include:

Function Calling / Tool Use

Modern LLMs are trained to generate structured function calls when they need external capabilities. For example, when asked about today's weather, the model generates a JSON function call to a weather API instead of hallucinating an answer. The runtime executes the function and feeds the result back to the model for synthesis.

ReAct: Reasoning + Acting

The ReAct framework (Yao et al., 2022) interleaves reasoning (thinking steps) with actions (tool calls). The model alternates between generating thoughts that analyse the problem and generating actions that gather information or make changes. This produces more reliable and interpretable agent behaviour.

Multi-Agent Systems

Complex tasks can be decomposed across multiple specialised LLM agents: a planner agent that breaks down the task, a coder agent that writes code, a reviewer agent that checks quality, and an executor agent that runs code and synthesises results. Each agent can use different tools and have different system prompts optimised for its role.

These agentic patterns are still rapidly evolving, but they represent the direction in which the field is moving — from passive text generation toward active problem-solving with real-world impact.

Deep Dive 11

Building a Complete Transformer from Scratch

11.1 Full Encoder-Decoder Transformer in PyTorch

This section provides a complete, runnable Transformer implementation that ties together every concept discussed in this book. We build each component from scratch and then assemble them into a working encoder-decoder model for sequence-to-sequence tasks.

Component 1: Scaled Dot-Product Attention

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

def attention(query, key, value, mask=None, dropout_fn=None):
    """Scaled dot-product attention with optional mask and dropout."""
    d_k = query.size(-1)
    # (B, H, T_q, d_k) @ (B, H, d_k, T_k) -> (B, H, T_q, T_k)
    scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)

    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)

    attn_weights = F.softmax(scores, dim=-1)

    if dropout_fn is not None:
        attn_weights = dropout_fn(attn_weights)

    # (B, H, T_q, T_k) @ (B, H, T_k, d_v) -> (B, H, T_q, d_v)
    context = torch.matmul(attn_weights, value)
    return context, attn_weights
```

Component 2: Multi-Head Attention Module

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads, dropout=0.1):
        super().__init__()
        assert d_model % num_heads == 0
        self.d_k = d_model // num_heads
        self.num_heads = num_heads

        self.linear_q = nn.Linear(d_model, d_model)
        self.linear_k = nn.Linear(d_model, d_model)
        self.linear_v = nn.Linear(d_model, d_model)
        self.linear_out = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, query, key, value, mask=None):
        batch_size = query.size(0)

        # Linear projections and reshape to (B, H, T, d_k)
        q = self.linear_q(query).view(batch_size, -1, self.num_heads, self.d_k).transpose(
1, 2)
        k = self.linear_k(key).view(batch_size, -1, self.num_heads, self.d_k).transpose(1,
 2)
        v = self.linear_v(value).view(batch_size, -1, self.num_heads, self.d_k).transpose(
1, 2)

        # Apply attention
        context, weights = attention(q, k, v, mask=mask, dropout_fn=self.dropout)

        # Concatenate heads and project
        context = context.transpose(1, 2).contiguous().view(batch_size, -1, self.num_heads
 * self.d_k)
        return self.linear_out(context)
```

Component 3: Position-wise Feed-Forward Network

```
class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff, dropout=0.1):
        super().__init__()
        self.layer1 = nn.Linear(d_model, d_ff)
        self.layer2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.layer2(self.dropout(F.gelu(self.layer1(x))))
```

Component 4: Encoder Layer & Encoder Stack

```
class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads, dropout)
        self.feed_forward = FeedForward(d_model, d_ff, dropout)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)

    def forward(self, x, src_mask=None):
        # Pre-norm self-attention
        normed = self.norm1(x)
        x = x + self.dropout1(self.self_attn(normed, normed, normed, src_mask))
        # Pre-norm feed-forward
        x = x + self.dropout2(self.feed_forward(self.norm2(x)))
        return x

class Encoder(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, num_layers, dropout=0.1):
        super().__init__()
        self.layers = nn.ModuleList([
            EncoderLayer(d_model, num_heads, d_ff, dropout)
            for _ in range(num_layers)])
        self.final_norm = nn.LayerNorm(d_model)

    def forward(self, x, mask=None):
        for layer in self.layers:
            x = layer(x, mask)
        return self.final_norm(x)
```

Component 5: Decoder Layer & Decoder Stack

```
class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads, dropout)
        self.cross_attn = MultiHeadAttention(d_model, num_heads, dropout)
        self.feed_forward = FeedForward(d_model, d_ff, dropout)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)
        self.drop1 = nn.Dropout(dropout)
        self.drop2 = nn.Dropout(dropout)
        self.drop3 = nn.Dropout(dropout)

    def forward(self, x, encoder_output, src_mask=None, tgt_mask=None):
        # Masked self-attention (causal)
        n = self.norm1(x)
        x = x + self.drop1(self.self_attn(n, n, n, tgt_mask))
        # Cross-attention (Q from decoder, KV from encoder)
        n = self.norm2(x)
        x = x + self.drop2(self.cross_attn(n, encoder_output, encoder_output, src_mask))
        # Feed-forward
        x = x + self.drop3(self.feed_forward(self.norm3(x)))
        return x

class Decoder(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, num_layers, dropout=0.1):
        super().__init__()
        self.layers = nn.ModuleList([
            DecoderLayer(d_model, num_heads, d_ff, dropout)
            for _ in range(num_layers)])
        self.final_norm = nn.LayerNorm(d_model)

    def forward(self, x, memory, src_mask=None, tgt_mask=None):
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.final_norm(x)
```

Component 6: Full Transformer Model

```
class Transformer(nn.Module):
    def __init__(self, src_vocab, tgt_vocab, d_model=512, num_heads=8,
                 d_ff=2048, num_layers=6, dropout=0.1, max_len=5000):
        super().__init__()
        self.d_model = d_model
        self.src_embed = nn.Embedding(src_vocab, d_model)
        self.tgt_embed = nn.Embedding(tgt_vocab, d_model)
        self.pos_encoding = nn.Parameter(
            self._sinusoidal_pe(max_len, d_model), requires_grad=False)
        self.encoder = Encoder(d_model, num_heads, d_ff, num_layers, dropout)
        self.decoder = Decoder(d_model, num_heads, d_ff, num_layers, dropout)
        self.output_proj = nn.Linear(d_model, tgt_vocab)
        self.dropout = nn.Dropout(dropout)

    def _sinusoidal_pe(self, max_len, d_model):
        pe = torch.zeros(max_len, d_model)
        pos = torch.arange(max_len).unsqueeze(1).float()
        div = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000) / d_model)
)
        pe[:, 0::2] = torch.sin(pos * div)
        pe[:, 1::2] = torch.cos(pos * div)
        return pe.unsqueeze(0)
    def encode(self, src, src_mask=None):
        x = self.dropout(self.src_embed(src) * math.sqrt(self.d_model)
                         + self.pos_encoding[:, :src.size(1)])
        return self.encoder(x, src_mask)

    def decode(self, tgt, memory, src_mask=None, tgt_mask=None):
        x = self.dropout(self.tgt_embed(tgt) * math.sqrt(self.d_model)
                         + self.pos_encoding[:, :tgt.size(1)])
        return self.decoder(x, memory, src_mask, tgt_mask)

    def forward(self, src, tgt, src_mask=None, tgt_mask=None):
        memory = self.encode(src, src_mask)
        dec_out = self.decode(tgt, memory, src_mask, tgt_mask)
        return self.output_proj(dec_out)

# Instantiate and test
model = Transformer(src_vocab=10000, tgt_vocab=10000)
src = torch.randint(0, 10000, (2, 20))
tgt = torch.randint(0, 10000, (2, 15))
logits = model(src, tgt)
print(f"Output shape: {logits.shape}")     # (2, 15, 10000)
print(f"Parameters: {sum(p.numel() for p in model.parameters()):,}")
```

This implementation contains approximately 44M parameters — comparable to a small Transformer-base model. Every component follows the architecture described in "Attention Is All You Need" with the Pre-LN variant used by modern models.

Deep Dive 12

Scaling Laws & Compute Economics

12.1 Neural Scaling Laws

Kaplan et al. (2020) discovered that language model performance follows remarkably predictable power-law relationships with three variables: model size N (parameters), dataset size D (tokens), and compute budget C (FLOPs):

```
L(N) = (N_c / N)^alpha_N where alpha_N ~ 0.076
L(D) = (D_c / D)^alpha_D where alpha_D ~ 0.095
L(C) = (C_c / C)^alpha_C where alpha_C ~ 0.050
```

These laws predict that loss decreases as a smooth power law in each variable, with no signs of diminishing returns. Doubling compute consistently yields a fixed percentage improvement in loss, regardless of the current scale.

The Chinchilla Optimal Frontier

Hoffmann et al. (2022) refined these laws with the "Chinchilla" finding: for a given compute budget, the optimal allocation is to scale model size and training data equally. The key formula:

```
N_opt proportional to C^0.50, D_opt proportional to C^0.50
```

This means model parameters and training tokens should be roughly equal when counted in comparable units. The rule of thumb: train on ~20 tokens per parameter.

Model Size	Optimal Tokens (Chinchilla)	Actual Tokens (LLaMA-2)	Assessment
7B	140B	2,000B	14x overtrained (intentional)
13B	260B	2,000B	8x overtrained
34B	680B	2,000B	3x overtrained
70B	1,400B	2,000B	1.4x overtrained

LLaMA-2 deliberately overtrains relative to Chinchilla optimality. Why? Because training cost is paid once, but inference cost is paid forever. A smaller model that is trained longer has the same quality as a Chinchilla-optimal larger model but is much cheaper to serve. This insight has reshaped how the industry approaches model training — "smaller model, more data" is now the dominant strategy.

12.2 Training Cost Estimation

For practical planning, you need to estimate how much training will cost. The key formula:

```
FLOPs = 6 * N * D (N = parameters, D = training tokens)
```

```
GPU-hours = FLOPs / (GPU_TFLOPS * utilisation * 3600)
```

Concrete examples with current cloud pricing (~$2/GPU-hour for A100):

Model	Params	Tokens	FLOPs	A100 Hours (50% util)	Cost (cloud)
Small LM	125M	30B	2.25e19	42 hours (1 GPU)	~$84
BERT-base	110M	3.3B (40 epochs)	2.2e18	4 hours (1 GPU)	~$8
GPT-2 XL	1.5B	40B	3.6e20	667 hours (1 GPU)	~$1,334
LLaMA-7B	6.7B	1T	4.0e22	74k hours (= 256 GPUs x 12 days)	~$148k
LLaMA-70B	70B	2T	8.4e23	1.56M hours (= 2048 GPUs x 32 days)	~$3.1M
GPT-4 (estimated)	~1.8T (MoE)	~13T	~1.4e26	~260M hours	~$100M+

Cost Reality Check: *These estimates assume near-optimal training — no failed runs, no hyperparameter search, no restarts from checkpoints. In practice, multiply by 2-5x for realistic total cost including experiments. LLaMA-2 70B cost an estimated $5-10M including all infrastructure and engineering overhead.*

12.3 Inference Cost Analysis

For production deployment, inference cost often exceeds training cost within months. Key metrics for inference efficiency:

Metric	Definition	Typical Values	Optimisation
Latency (TTFT)	Time to first token; how fast user sees first response	50-500ms (depends on prompt length)	KV-cache; prompt caching; short system prompts
Throughput (tokens/s)	Output tokens per second per GPU	20-100 tok/s (autoregressive)	Batching; quantisation; speculative decoding
Tokens per dollar	How many tokens you get per $1 of GPU cost	~500k-2M tokens/$ (depends on model)	Quantisation; efficient serving; smaller models

Memory efficiency	What % of GPU memory is actively used for compute	40-70% (rest is KV-cache and fragmentation)	PagedAttention; GQA; quantisation

A critical insight: for autoregressive generation, the model is memory-bandwidth bound, not compute bound. Each generated token requires reading all model weights from GPU memory, but only performs a small amount of arithmetic per weight. This is why quantisation (reducing bytes per weight) often yields nearly proportional speedup — you are reducing the memory transfer bottleneck, not the compute.

Deep Dive 13

Evaluation, Benchmarking & Safety

13.1 LLM Evaluation Taxonomy

Evaluating LLMs is fundamentally harder than evaluating traditional ML models because the output space is vast and quality is subjective. Modern evaluation spans multiple dimensions:

Dimension	Benchmark	What It Measures	Typical Scores (Frontier Models)
Knowledge	MMLU (57 subjects)	Factual knowledge across academia and professions	GPT-4: ~87% LLaMA-3 70B: ~82% Mistral 7B: ~62%
Reasoning	GSM8K MATH	Grade-school and competition-level math	GPT-4: ~92% / ~50% Claude 3.5: ~95% / ~72%
Coding	HumanEval MBPP	Python code generation from docstrings	GPT-4: ~85% Claude 3.5: ~92%
Instruction Following	MT-Bench AlpacaEval	Quality of instruction-following responses	Rated by GPT-4 as judge; ELO scores
Safety	TruthfulQA BBQ	Truthfulness; bias detection and mitigation	Higher = more truthful / less biased
Multilingual	MGSM XNLI	Cross-lingual math and NLI tasks	Varies widely by language pair
Long Context	RULER NeedleInHaystack	Retrieval from very long documents	Tests at 32k, 128k, 1M tokens

13.2 Perplexity: The Fundamental LM Metric

Perplexity is the standard metric for evaluating language models. It measures how "surprised" the model is by the test data:

```
PPL = exp( -1/N * sum_i log P(token_i | context) )
```

A perplexity of 10 means the model is, on average, as uncertain as if it had to choose uniformly among 10 equally likely tokens at each step. Lower perplexity = better language model. Important caveats:

- Perplexity is only comparable between models using the same tokeniser and evaluated on the same data. BPE tokenisers with different vocabulary sizes produce different token counts, making raw perplexity scores incomparable.

- Perplexity measures language modelling ability, not task performance. A model with lower perplexity is not guaranteed to be better at downstream tasks like QA or summarisation.
- Perplexity on training data measures memorisation; perplexity on held-out data measures generalisation. Always report the evaluation dataset clearly.

13.3 Contamination & Evaluation Pitfalls

A major challenge in LLM evaluation is data contamination — benchmark test sets leaking into pre-training data. If the model has seen MMLU questions during training, its score is inflated and not representative of true capability.

Best practices to mitigate contamination:

- **Holdout analysis:** Check n-gram overlap between training data and benchmark test sets. LLaMA-2's paper reports contamination analysis for each benchmark.
- **Dynamic benchmarks:** Use regularly updated benchmarks with fresh questions (e.g. LiveBench, ChatBot Arena) that cannot be in any model's training data.
- **Private test sets:** Maintain internal evaluation sets that are never published.
- **Capability probing:** Test generalisation by modifying benchmark questions (e.g. change numbers in math problems, rephrase classification examples).

13.4 AI Safety & Alignment

As Transformers become more capable, safety and alignment become critical engineering concerns. Key safety dimensions for deployed models:

Safety Dimension	Risk	Mitigation
Hallucination	Model generates plausible but false information	Retrieval-augmented generation; confidence calibration; citation requirements
Harmful Content	Model produces toxic, biased, or dangerous output	RLHF/DPO alignment; content filters; red-teaming
Privacy	Model memorises and reproduces training data (PII, copyrighted text)	Differential privacy; data deduplication; membership inference tests
Jailbreaking	Adversarial prompts bypass safety filters	Multi-layer filtering; adversarial training; prompt injection detection
Misuse	Model used for scam/spam/deepfakes at scale	Use policies; rate limiting; watermarking; monitoring

The alignment problem — ensuring AI systems reliably pursue intended goals — remains an active area of research. Current approaches (RLHF, Constitutional AI, DPO) provide practical solutions but do not fully solve the problem. Understanding these limitations is essential for responsible deployment of Transformer-based systems.

Modern Vision Transformers: Beyond ViT

14.1 DeiT: Data-Efficient Image Transformers

DeiT (Touvron et al., 2021) demonstrated that ViT can be trained competitively on ImageNet-1K alone (1.3M images), without the massive JFT-300M dataset used by the original ViT. The key innovation was a knowledge distillation approach using a CNN teacher:

- **Distillation token:** In addition to the CLS token, DeiT adds a second special token — the distillation token. This token is trained to mimic the output of a pre-trained CNN teacher (RegNetY-16GF) via hard-label distillation.
- **Aggressive augmentation:** DeiT uses the full augmentation pipeline described in Deep Dive 7 (RandAugment, Mixup, CutMix, Erasing, Stochastic Depth) to compensate for ViT's lack of inductive biases.
- **Longer training:** DeiT trains for 300 epochs (vs typical 90 for CNNs), as ViTs benefit more from extended training than CNNs do.

Results: DeiT-B/16 achieves 83.1% top-1 accuracy on ImageNet-1K — competitive with EfficientNet-B7 and significantly better than the original ViT-B/16 (77.9%) trained on the same data.

Model	Training Data	Augmentation	Epochs	ImageNet Top-1
ViT-B/16 (original)	JFT-300M (300M)	Basic	~30	77.9% (IN-1K eval)
ViT-B/16 (IN-1K only)	ImageNet-1K (1.3M)	Basic	300	~76%
DeiT-B/16	ImageNet-1K (1.3M)	Full DeiT recipe	300	81.8%
DeiT-B/16 + distill	ImageNet-1K (1.3M)	Full + RegNet teacher	300	83.1%
ResNet-50	ImageNet-1K (1.3M)	Standard	90	76.1%
EfficientNet-B7	ImageNet-1K (1.3M)	Standard	600	84.3%

14.2 Swin Transformer Architecture Deep Dive

The Swin Transformer is arguably the most important ViT variant for practical computer vision. Its design principles solve the two main limitations of vanilla ViT: quadratic complexity and single-scale features.

Hierarchical Patch Merging

Swin creates a feature pyramid by progressively merging patches across four stages, analogous to the feature maps at different resolutions in a CNN:

Stage	Resolution	Channels	Patches	Attention Window
Stage 1	H/4 x W/4	C = 96	56 x 56 = 3136	7 x 7 = 49 per window
Stage 2	H/8 x W/8	2C = 192	28 x 28 = 784	7 x 7 = 49 per window
Stage 3	H/16 x W/16	4C = 384	14 x 14 = 196	7 x 7 = 49 per window
Stage 4	H/32 x W/32	8C = 768	7 x 7 = 49	7 x 7 = 49 per window

Between stages, a patch merging layer concatenates 2x2 groups of adjacent tokens and projects them to double the channel dimension. This is analogous to a strided convolution or pooling in CNNs, reducing spatial resolution while increasing feature richness.

Shifted Window Attention

Within each stage, attention is computed only within local 7x7 windows (49 tokens each). This reduces complexity from O(n^2) to O(n * w^2) where w is the window size. But this creates a problem: windows are isolated from each other, preventing information flow between them.

Swin's solution: alternate between regular and shifted window partitions in consecutive layers. In even layers, windows are partitioned starting from position (0, 0). In odd layers, windows are shifted by (w/2, w/2) pixels, creating overlapping connections between adjacent regular windows. This way, tokens at window boundaries can communicate with tokens in neighbouring windows through the shifted partition.

The implementation handles boundary effects with cyclic shifts and masking, making it efficient on GPUs. The overall complexity is O(n) in the number of tokens — linear, not quadratic — while still allowing global information propagation across layers.

14.3 ConvNeXt: The CNN Strikes Back

ConvNeXt (Liu et al., 2022) asked a provocative question: if we modernise a standard CNN with all the design choices from Transformers, can it match Swin Transformer? The answer was yes.

ConvNeXt starts from a ResNet-50 and applies the following modifications, each borrowed from Transformer design:

Modification	From	To	Inspired By
Macro design	ResNet stages (3, 4, 6, 3)	Swin-like ratios (3, 3, 9, 3)	Swin-T stage ratios
Stem cell	7x7 conv, stride 2, pool	4x4 conv, stride 4 (patchify)	ViT patch embedding
Depthwise conv	Standard 3x3 conv	7x7 depthwise conv	Large kernels ≈ local attention
Activation	ReLU	GELU	Transformer FFN

Normalisation	BatchNorm	LayerNorm	Transformer norm choice
Fewer activations	ReLU after every conv layer	GELU only in expanded branch	Transformer has 1 activation in FFN
Inverted bottleneck	Narrow -> wide -> narrow	Wide -> narrow -> wide	Transformer FFN: d -> 4d -> d
Fewer norm layers	BN after every conv	LN only before 1x1 conv	Transformer: 1 LN per sub-layer

The final ConvNeXt-T achieves 82.1% on ImageNet-1K — matching Swin-T's 81.3% and surpassing it slightly, all without any attention mechanism. This demonstrates that much of Transformer's advantage comes from modern training practices and design choices, not from attention itself.

Practical Takeaway: *For dense prediction tasks (detection, segmentation) on standard-resolution images with limited compute, ConvNeXt is often the most practical choice. It has the efficiency of convolutions, the design benefits of Transformers, and works with all existing CNN-based detection/segmentation frameworks without modification.*

Deep Dive 15

Practical Deployment Patterns

15.1 RAG: Retrieval-Augmented Generation

RAG is the dominant pattern for deploying LLMs in enterprise settings where the model needs access to private, up-to-date, or domain-specific knowledge that was not in its training data.

The RAG pipeline consists of two phases:

Indexing Phase (Offline)

- **Document ingestion:** Load documents (PDFs, web pages, databases) and split them into chunks (typically 256-1024 tokens per chunk).
- **Embedding:** Each chunk is converted to a dense vector using an embedding model (e.g. text-embedding-3-small, BGE, E5). These models are fine-tuned BERT/ViT variants optimised for semantic similarity.
- **Indexing:** Embeddings are stored in a vector database (FAISS, Pinecone, Weaviate, Chroma) with metadata for efficient nearest-neighbour search.

Query Phase (Online)

- **Query embedding:** The user's question is embedded using the same embedding model.
- **Retrieval:** Top-k most similar chunks are retrieved from the vector database using cosine similarity or approximate nearest-neighbour search.
- **Augmented generation:** Retrieved chunks are prepended to the LLM prompt as context, and the model generates an answer grounded in the retrieved information.

```
# Simplified RAG Pipeline
from sentence_transformers import SentenceTransformer
import faiss
import numpy as np

# 1. Index documents
embedder = SentenceTransformer('all-MiniLM-L6-v2')
chunks = ["The Eiffel Tower is 330m tall...",
          "It was built in 1889 for the World's Fair...",
          "Located on the Champ de Mars in Paris..."]
embeddings = embedder.encode(chunks)

# Build FAISS index
dim = embeddings.shape[1]
index = faiss.IndexFlatIP(dim)
faiss.normalize_L2(embeddings)
index.add(embeddings)

# 2. Query
query = "How tall is the Eiffel Tower?"
```

```
q_emb = embedder.encode([query])
faiss.normalize_L2(q_emb)
scores, indices = index.search(q_emb, k=3)

# 3. Augmented prompt
context = "\n".join([chunks[i] for i in indices[0]])
prompt = f"Context:\n{context}\n\nQuestion: {query}\nAnswer:"
# Feed prompt to LLM for generation
```

15.2 Fine-Tuning Decision Framework

One of the most common practical questions is: when should I fine-tune a model vs use it out of the box with RAG or prompting? Here is a decision framework:

Scenario	Recommended Approach	Why
Need domain knowledge not in training data	RAG	Retrieval provides knowledge without retraining
Need specific output format or style	Few-shot prompting or SFT	Examples in prompt work well; SFT for consistent format
Need to classify into domain-specific categories	Fine-tune (LoRA)	Classification benefits from task-specific adaptation
Need to generate domain-specific text (medical, legal)	Fine-tune (LoRA) + RAG	Both domain adaptation and knowledge grounding
Need maximum quality on a specific task	Full fine-tune or LoRA with high rank	Task-specific optimisation; largest quality gains
Need to reduce latency/cost	Distillation to smaller model	Train small model on large model's outputs
Need multilingual capability	Fine-tune on target language data	Add language capability through continued training

15.3 Production Monitoring & Observability

Deploying an LLM in production requires continuous monitoring to maintain quality and detect issues:

What to Monitor	How	Alert Threshold
Latency (P50, P95, P99)	Histogram of TTFT and total generation time	P95 > 2x baseline; P99 > 5s
Token throughput	Tokens generated per second across all requests	< 50% of baseline capacity
Error rate	5xx responses, timeouts, OOM errors	> 1% error rate

Output quality	LLM-as-judge scoring; human evaluation sample	Quality score drops > 10% from baseline
Hallucination rate	Factual consistency check vs retrieved sources	Hallucination rate > 15% of responses
Cost per request	GPU time * price per request	Cost exceeds budget threshold
User satisfaction	Thumbs up/down; retry rate	Satisfaction < 80%; retry > 20%
Safety violations	Content filter triggers; user reports	Any safety violation triggers review

A robust observability stack includes: request/response logging with sampling, automated quality evaluation (using a separate LLM as judge), latency/throughput dashboards, cost tracking per user/team, safety filter monitoring, and A/B testing infrastructure for model updates.

15.4 Model Selection Guide

Choosing the right model for your use case is one of the most impactful engineering decisions. Here is a practical guide based on common deployment scenarios:

Use Case	Budget	Recommended Model	Why
Simple classification or extraction	Low	BERT-base / DeBERTa-v3	Small, fast, accurate for NLU
General chat assistant	Medium	LLaMA-3 8B or Mistral 7B (LoRA)	Good quality; runs on 1 GPU
Enterprise RAG with tool use	Medium-High	Claude 3.5 Sonnet / GPT-4o-mini (API)	Strong reasoning; tool use support
Code generation & analysis	Medium	DeepSeek Coder / CodeLlama 34B	Specialised for code; high quality
Vision + language tasks	High	Claude 3.5+ / GPT-4o / Gemini	Native multimodal; strong vision
On-premise high security	High	LLaMA-3 70B / Mistral Large	Self-hosted; full data control
Edge / mobile deployment	Low	Phi-3 mini / Gemma 2B	Tiny models; runs on device
Multilingual applications	Medium	Qwen-2.5 / Aya 23	Strong non-English support

Golden Rule: *Start with the smallest model that meets your quality requirements, and scale up only when necessary. A well-prompted 7B model with RAG often outperforms a poorly-configured 70B model for domain-specific tasks. Engineering effort on prompting, retrieval, and evaluation yields better ROI than simply using a larger model.*

Deep Dive 16

Normalisation: LayerNorm, RMSNorm, & Alternatives

16.1 Why Normalisation is Critical in Deep Transformers

Without normalisation, deep Transformers (32-96+ layers) suffer from internal covariate shift — the distribution of each layer's inputs changes as preceding layers update during training. This forces later layers to constantly adapt to shifting input statistics, slowing convergence and sometimes causing training collapse.

Layer Normalisation addresses this by normalising each sample's features to zero mean and unit variance:

```
LN(x) = gamma * (x - mu) / sqrt(sigma^2 + eps) + beta
mu = 1/d * sum(x_i), sigma^2 = 1/d * sum((x_i - mu)^2)
```

The learned parameters gamma (scale) and beta (shift) allow the network to undo the normalisation if needed. In practice, gamma converges close to 1 and beta close to 0 in most layers, suggesting that normalisation itself — not the learned transformation — is the primary benefit.

Pre-Norm vs Post-Norm: Training Dynamics

The placement of LayerNorm relative to the sub-layer has a profound effect on training stability. Let us analyse both variants mathematically:

Post-Norm (original Transformer): y = LN(x + F(x)). The sub-layer output F(x) is added to x before normalisation. The gradient through this path is: dL/dx = dL/dy * dLN/d(x + F(x)) * (I + dF/dx). The LayerNorm derivative creates a dependency between the gradient and the output magnitude, which can cause instability when F(x) is large early in training.

Pre-Norm (GPT-2+): y = x + F(LN(x)). The gradient through the residual path is: dL/dx = dL/dy * (I + dF/dLN * dLN/dx). The identity term I provides a clean gradient highway regardless of what F does. This makes training much more stable, especially in the early steps when F(x) may produce large, poorly-calibrated outputs.

Empirical evidence overwhelmingly favours Pre-Norm for large models: GPT-2, GPT-3, LLaMA, Mistral, and virtually all modern LLMs use Pre-Norm. Post-Norm is retained only in BERT-family models for backward compatibility.

RMSNorm: The Modern Standard

RMSNorm (Zhang & Sennrich, 2019) simplifies LayerNorm by removing the mean-centering step:

```
RMSNorm(x) = x / sqrt(1/d * sum(x_i^2) + eps) * gamma
```

Advantages over LayerNorm: fewer operations (no mean computation, no beta parameter), ~10-15% faster per-layer computation, and empirically equivalent quality. LLaMA, Mistral, and Qwen all use RMSNorm. The 10-15% speedup per layer compounds across 80 layers and millions of forward passes during training, saving significant compute cost.

```python
class RMSNorm(nn.Module):
    def __init__(self, dim, eps=1e-6):
        super().__init__()
        self.eps = eps
        self.gamma = nn.Parameter(torch.ones(dim))

    def forward(self, x):
        rms = torch.sqrt(x.pow(2).mean(dim=-1, keepdim=True) + self.eps)
        normalised = x / rms
        return normalised * self.gamma

# Comparison: LayerNorm vs RMSNorm parameter count
import torch.nn as nn
d = 4096
ln = nn.LayerNorm(d)       # gamma + beta = 8192 params
rn = RMSNorm(d)             # gamma only = 4096 params
print(f"LayerNorm params: {sum(p.numel() for p in ln.parameters()):,}")  # 8,192
print(f"RMSNorm params:   {sum(p.numel() for p in rn.parameters()):,}")  # 4,096
```

16.2 QK-Norm and Other Emerging Normalisation Techniques

Recent models have introduced additional normalisation at specific points in the architecture:

- **QK-Norm (Dehghani et al., 2023):** Apply RMSNorm to query and key vectors before computing attention scores. This prevents the dot-product magnitude from growing with model dimension, improving training stability for very large models. Used in Gemma 2 and PaLM 2.
- **Sub-LN (Wang et al., 2022):** Add an extra normalisation layer inside each sub-layer (after attention, before the residual addition). Combines benefits of Pre-LN (stability) and Post-LN (performance). Used in some Microsoft models.
- **DeepNorm (Wang et al., 2022):** Scale the residual connection by a factor alpha > 1 and apply post-normalisation. Enables training Transformers with 1000+ layers without stability issues.

Tokenisation: BPE, WordPiece, & SentencePiece

17.1 Why Tokenisation Matters

Tokenisation is the first and often most underappreciated step in any Transformer pipeline. The choice of tokeniser directly affects model capacity, multilingual capability, and computational efficiency. A poor tokeniser can waste 30-50% of the model's context window on redundant tokens.

Key design decisions in tokenisation:

Decision	Options	Trade-off
Vocabulary size	8k (small) to 128k (large)	Larger vocab = better compression but more embedding params
Algorithm	BPE, WordPiece, Unigram	BPE most popular; Unigram most principled
Pre-tokenisation	Word-level, byte-level, character-level	Byte-level handles any language without UNK
Special tokens	[CLS], [SEP], [PAD], [BOS], [EOS]	Task-specific tokens add overhead
Normalisation	Lowercase, NFKC, whitespace handling	Affects casing; multilingual support

17.2 BPE: Byte Pair Encoding

BPE (Sennrich et al., 2016) is the most widely used tokenisation algorithm. It starts with individual characters (or bytes) and iteratively merges the most frequent adjacent pair:

```
# BPE Algorithm (simplified)
# Corpus: "low lower lowest"

# Step 0 (character-level vocabulary):
# {'l', 'o', 'w', 'e', 'r', 's', 't', ' '}

# Step 1: Most frequent pair: ('l', 'o') -> merge to 'lo'
# Vocabulary: {'lo', 'w', 'e', 'r', 's', 't', ' '}
# Corpus: "lo w lo w e r lo w e s t"

# Step 2: Most frequent pair: ('lo', 'w') -> merge to 'low'
# Vocabulary: {'low', 'e', 'r', 's', 't', ' '}
# Corpus: "low low e r low e s t"
# Step 3: Most frequent pair: ('low', 'e') -> merge to 'lowe'
# ...continue until desired vocabulary size is reached

# Modern BPE uses byte-level (256 base tokens)
# GPT-2: 50,257 tokens  |  LLaMA: 32,000  |  GPT-4: 100,000
```

Byte-level BPE (used by GPT-2+) starts with 256 byte values instead of characters. This guarantees that any input can be tokenised without UNK tokens, even for languages or scripts not well-represented in the training data. The downside is that rare scripts may be broken into many individual bytes, wasting context window.

17.3 Vocabulary Size Impact Analysis

The choice of vocabulary size has cascading effects throughout the model:

Vocab Size	Embedding Params	Compression Ratio	Multilingual	Example Mode
8,000	8k * d_model = 6.1M (d=768)	Poor; many tokens per word	Very limited; Non-Latin breaks	Some research models
32,000	32k * d_model = 131M (d=4096)	Good for English; fair for others	Adequate for major languages	LLaMA, LLaMA-2
50,257	50k * d_model = 38.6M (d=768)	Good for English; byte-level fallback	Fair; byte fallback helps	GPT-2
100,000	100k * d_model = 410M (d=4096)	Excellent; efficient encoding	Strong multi-lingual support	GPT-4
128,000	128k * d_model = 524M (d=4096)	Best compression; long-context friendly	Excellent; full Unicode	LLaMA-3

LLaMA-3 expanded from 32k to 128k vocabulary specifically to improve multilingual tokenisation efficiency. For Chinese text, LLaMA-2's 32k tokeniser produces 2-3x more tokens than LLaMA-3's 128k tokeniser, directly impacting context window utilisation and inference cost.

Practical Impact: *A document that uses 2000 tokens with a 128k vocabulary might need 4000 tokens with a 32k vocabulary. This means 2x the inference cost, 2x the latency, and half the effective context window. For production multilingual applications, tokeniser efficiency is a critical cost driver.*

Transfer Learning Strategies for Transformers

18.1 The Pre-train → Fine-tune Paradigm

The fundamental insight driving modern deep learning is that representations learned on large-scale pre-training tasks transfer effectively to downstream tasks. For Transformers, this takes three forms:

Strategy	Description	When to Use	Cost
Feature extraction (frozen backbone)	Use pre-trained model as fixed feature extractor; train only classifier head	Very small dataset; limited compute; no GPU	Minimal (CPU OK)
Linear probing	Train a linear classifier on top of frozen representations	Quick baseline; understanding what the model encodes	Very low (minutes)
Full fine-tuning	Update all model parameters on downstream data	Sufficient data (>10k); maximum quality; domain shift	High (hours-days)
LoRA / QLoRA	Freeze base; train low-rank adapters	Large models; limited GPU memory; multiple tasks	Low-Medium (1 GPU OK)
Prompt tuning / Prefix tuning	Freeze base; train continuous prompt embeddings	API-only access; multiple tasks; no weight access	Very low (soft prompts only)
Adapter layers	Insert small trainable modules between frozen layers	Multiple tasks; modular fine-tuning	Low (~2-5% params)

18.2 Layer Freezing Strategies

When fine-tuning with limited data, selectively freezing layers can prevent overfitting while still adapting the model. The general principle: lower layers learn general features (grammar, syntax), while upper layers learn task-specific features (sentiment, classification).

- **Freeze all but last N layers:** Most common approach. Start by training only the last 2-3 layers, then gradually unfreeze more if validation metrics improve.
- **Discriminative learning rates:** Use different learning rates per layer group — lower rates for early layers (preserve general features), higher rates for later layers (adapt to task). Typically a 10x ratio

between first and last layer groups.

- **Gradual unfreezing:** Start with only the classifier head unfrozen, then unfreeze one layer group per epoch from top to bottom. This gives each layer group time to adapt without disrupting lower layers.

```
# Discriminative learning rates with PyTorch
from torch.optim import AdamW

# Group parameters by layer depth
param_groups = [
    {'params': model.embeddings.parameters(), 'lr': 1e-6},
    {'params': model.encoder.layer[:4].parameters(), 'lr': 5e-6},
    {'params': model.encoder.layer[4:8].parameters(), 'lr': 1e-5},
    {'params': model.encoder.layer[8:].parameters(), 'lr': 2e-5},
    {'params': model.classifier.parameters(), 'lr': 5e-5},
]

optimizer = AdamW(param_groups, weight_decay=0.01)
```

18.3 Domain Adaptation: Continued Pre-training

For domains with specialised vocabulary and knowledge (biomedical, legal, financial), continued pre-training on domain text before fine-tuning on task data consistently improves results. The pipeline becomes:

```
General Pre-training -> Domain Pre-training -> Task Fine-tuning
```

Examples of successful domain adaptation:

Base Model	Domain	Domain Pre-training	Result
BERT	Biomedical	PubMed + PMC (BioBERT)	+3-5% F1 on biomedical NER/QA
BERT	Clinical	MIMIC-III clinical notes (ClinicalBERT)	+2-4% on clinical NLI and readmission
RoBERTa	Legal	Legal corpus (LegalBERT)	+4-7% on contract analysis tasks
RoBERTa	Financial	SEC filings + earnings (FinBERT)	+5-10% on financial sentiment analysis
LLaMA	Code	Code corpus (CodeLlama)	Dramatic improvement on HumanEval

The typical domain pre-training recipe: use the same masked language modelling (for BERT-family) or autoregressive LM (for GPT-family) objective, but on domain-specific text. Train for 50k-500k steps with a small learning rate (1e-5 to 5e-5). The amount of domain text needed varies, but even 1-5GB of domain text typically yields significant improvements.

Deep Dive 19

Prompt Engineering: Getting the Most from Transformers

19.1 Prompt Engineering Principles

Prompt engineering is the practice of crafting inputs to an LLM to elicit the desired output without changing model weights. It is often the most cost-effective way to improve LLM performance for a specific task.

Effective prompts follow several key principles:

- **Be specific and detailed:** Vague prompts produce vague outputs. Specify the desired format, length, tone, audience, and constraints explicitly.
- **Provide examples (few-shot):** Including 2-5 input-output examples in the prompt dramatically improves consistency and quality. The examples serve as an implicit "specification" of the task.
- **Use structured formats:** XML tags, JSON templates, or numbered instructions make it easier for the model to parse complex prompts and produce structured output.
- **Encourage step-by-step reasoning:** For complex tasks, asking the model to "think step by step" or "show your reasoning" activates chain-of-thought reasoning that significantly improves accuracy on math, logic, and analytical tasks.
- **Specify what NOT to do:** Negative constraints ("do not include personal opinions", "do not use bullet points") are often as important as positive instructions.

19.2 Advanced Prompting Techniques

Technique	Description	Best For	Example
Zero-shot	No examples; just the instruction	Simple, well-defined tasks	"Translate to French: Hello world"
Few-shot	Include 2-5 input-output examples	Classification; formatting tasks	Show 3 examples of sentiment -> label
Chain-of-Thought (CoT)	Ask model to reason step by step before answering	Math, logic, complex reasoning	"Let's think step by step..."
Self-Consistency	Generate multiple CoT paths; take majority vote	Math; factual QA; high-stakes answers	Sample 5 answers; pick most common

ReAct	Interleave reasoning steps with tool calls	Research; data analysis; agentic	Thought -> Action -> Observation -> ...
System prompt engineering	Craft the system message for persona, constraints, format	All applications; production systems	"You are a medical assistant. Always cite sources."

19.3 Structured Output with LLMs

For production applications, you typically need the LLM to produce structured output (JSON, XML, specific formats) rather than free-form text. Techniques for reliable structured output:

- **JSON mode:** Many APIs (OpenAI, Anthropic) support a structured output mode that constrains the model to produce valid JSON. Specify the schema in the prompt and enable the mode in the API call.
- **XML tags in prompts:** Wrap different parts of the expected output in XML tags. Models are very reliable at following XML structure when examples are provided.
- **Grammar-constrained decoding:** Tools like Outlines and guidance constrain token generation to follow a context-free grammar, guaranteeing valid output structure.
- **Retry with validation:** Parse the output, validate against schema, and retry with error feedback if validation fails. Simple but effective for production systems.

```
# Example: Structured output with XML tags
prompt = """Analyse the following review and extract structured information.

<review>
The restaurant had amazing pasta but the service was slow.
Overall a decent experience for the price.
</review>

Respond in this exact format:
<analysis>
  <sentiment>positive/negative/mixed</sentiment>
  <food_rating>1-5</food_rating>
  <service_rating>1-5</service_rating>
  <summary>one sentence summary</summary>
</analysis>
"""
# Models reliably follow this XML structure with 95%+ accuracy
```

Deep Dive 20

Beyond Attention: State Space Models & Emerging Architectures

20.1 The Case Against Quadratic Attention

Despite the enormous success of Transformers, the O(n^2) attention mechanism remains a fundamental constraint. For a 128k context window with d_k = 128, the attention matrix alone requires 128k x 128k x 2 bytes = 32 GB of memory per layer per head — clearly impractical without tricks like Flash Attention. This has motivated the search for alternatives that achieve linear O(n) complexity.

20.2 State Space Models (SSMs)

State Space Models draw inspiration from control theory, processing sequences through a learned linear dynamical system:

```
h_t = A * h_(t-1) + B * x_t (state update)
y_t = C * h_t + D * x_t (output)
```

Where A is the state transition matrix, B is the input projection, C is the output projection, and D is a direct feedthrough. Unlike RNNs, SSMs use structured matrices (diagonal or DPLR) that enable efficient parallelisation during training.

The key insight of Mamba (Gu & Dao, 2023) is making A, B, C input-dependent — the state transition changes based on the current input, allowing the model to selectively remember or forget information. This "selectivity" mechanism replaces the role of attention in deciding what to focus on:

Aspect	Standard Attention	Mamba (Selective SSM)
Mechanism	Pairwise token interactions (every pair scores)	Input-dependent state transition (sequential but parallelisable)
Training complexity	O(n^2 * d) — quadratic	O(n * d) — linear
Inference complexity (per step)	O(n * d) — grows with context length	O(d) — constant per step
Memory (inference)	O(n * L * d) — KV-cache grows with context	O(L * d) — fixed state per layer
Parallelism (training)	Excellent (matrix multiply)	Good (parallel scan)

Retrieval ability	Excellent — can attend to any specific position	Weaker — information must persist in state
Models	GPT, LLaMA, Claude, BERT, ViT, etc.	Mamba, Mamba-2, Jamba (hybrid)

20.3 Hybrid Architectures: The Best of Both Worlds

The emerging consensus is that pure SSMs and pure Transformers each have strengths the other lacks. Hybrid architectures attempt to combine both:

- **Jamba (AI21, 2024):** Alternates Transformer layers (with attention) and Mamba layers (with SSM). Uses MoE for efficiency. Achieves strong results with a 256k context window and linear memory scaling.
- **Griffin (Google, 2024):** Combines gated linear recurrences with local attention windows. Achieves Transformer-level quality with sub-quadratic complexity.
- **RWKV:** A "linear attention" architecture that reformulates attention as an RNN-like recurrence during inference, achieving O(1) per-token inference while maintaining Transformer-like training parallelism.

The key question for the future: will attention remain the dominant mechanism, or will these alternatives eventually supersede it? Current evidence suggests that for tasks requiring precise recall of specific information from long contexts, attention remains superior. For tasks requiring general language modelling and generation, SSMs are competitive or better. Hybrid architectures may offer the best of both worlds.

20.4 Diffusion Transformers (DiT)

One of the most exciting recent developments is applying Transformer architectures to diffusion models for image and video generation. Diffusion Transformers (DiT, Peebles & Xie, 2023) replace the traditional U-Net backbone in diffusion models with a Transformer:

- **Architecture:** Similar to ViT — the noisy image is divided into patches, projected to tokens, and processed by a standard Transformer encoder with adaptive layer normalisation (AdaLN) conditioned on the timestep and class label.
- **Scaling:** DiT follows clean scaling laws — larger models and more compute consistently produce higher-quality images, unlike U-Nets which show diminishing returns beyond a certain size.
- **Applications:** DiT is the backbone of DALL-E 3 and Sora (OpenAI's video generation model). Its success demonstrates that the Transformer architecture generalises beyond language and discriminative vision tasks to generative vision as well.

The unifying theme across all these developments is clear: the Transformer's core ideas (attention, residual connections, layer normalisation, pre-training at scale) continue to drive progress across every modality and task type. Whether the attention mechanism itself will persist or evolve into something new remains one of the most exciting open questions in deep learning.

End of Part II: Deep Dive Material

Jekardah AI Labs — Romi Nur Ismanto — Jekardah.com — rominur@gmail.com

2026 Edition — AI Engineering Handbook: Transformers & ViT

APPENDIX

50 Interview Questions & Answers

Transformers & Attention Mechanisms

Self-Attention | Multi-Head Attention | Encoder-Decoder | Positional Encoding

Vision Transformer (ViT) & Hybrid Models

Patch Tokenisation | CLS Token | CNN vs ViT | Swin | DINO

Category	Questions	Topics
1. Transformer Foundations	Q1 – Q10	Architecture, positional encoding, FFN, complexity
2. Self-Attention Mechanism	Q11 – Q20	Q/K/V, scaled dot-product, masks, visualisation
3. Multi-Head Attention	Q21 – Q28	Parallel heads, GQA, parameter analysis
4. Encoder–Decoder	Q29 – Q36	Encoder stack, decoder masks, KV-cache, beam search
5. Vision Transformer (ViT)	Q37 – Q43	Patch embedding, CLS, ViT variants
6. CNN vs ViT & Hybrids	Q44 – Q50	Inductive bias, hybrids, DINO, Flash Attention, LoRA

50 Questions with Python Code Snippets — Interview-Ready Reference

Romi Nur Ismanto — Jekardah AI Labs — Jekardah.com — rominur@gmail.com — 2026 Edition

Category 1: Transformer Foundations (Q1–Q10)

Foundational concepts behind the Transformer architecture: its motivation, building blocks, and operational principles.

Q1: What is the Transformer architecture and what problems motivated its creation?

A Transformer is a neural network architecture built entirely on attention mechanisms, removing the need for recurrence or convolution. It was proposed in the 2017 paper "Attention Is All You Need" to address two key weaknesses of RNN/LSTM models:

- **Sequential computation:** RNNs process one token at a time, blocking parallelisation and slowing training on long sequences.
- **Vanishing long-range signals:** Gradient decay causes information from distant tokens to be lost, even with LSTM gates.

The Transformer tackles both issues by letting every token interact with every other token simultaneously via self-attention, providing O(1) signal propagation regardless of distance.

```
import torch, torch.nn as nn

# Basic Transformer Encoder usage
enc_layer = nn.TransformerEncoderLayer(
    d_model=512, nhead=8, dim_feedforward=2048,
    dropout=0.1, batch_first=True)
transformer_enc = nn.TransformerEncoder(enc_layer, num_layers=6)

inp = torch.rand(2, 10, 512)  # (batch, seq_len, d_model)
result = transformer_enc(inp)  # same shape
print("Result shape:", result.shape)
```

Q2: Describe the four essential components inside a Transformer encoder block.

Every encoder block is assembled from four parts:

- **Token Embedding:** Maps discrete tokens to continuous vectors of dimension d_model.
- **Positional Encoding:** Adds sequence-order information (sinusoidal or learned), since attention alone is permutation-invariant.
- **Multi-Head Self-Attention:** Lets each token compute a weighted average over all tokens in the sequence using parallel attention heads.
- **Position-wise Feed-Forward Network:** Two linear transformations with a non-linearity (ReLU/GELU), applied independently at each position.

Each sub-layer uses a residual connection followed by layer normalisation: output = LayerNorm(x + Sublayer(x)).

```
class EncoderBlock(nn.Module):
    def __init__(self, dim=512, heads=8, ff_dim=2048):
        super().__init__()
        self.self_attn = nn.MultiheadAttention(dim, heads, batch_first=True)
        self.feed_fwd = nn.Sequential(
            nn.Linear(dim, ff_dim), nn.ReLU(),
            nn.Linear(ff_dim, dim))
        self.norm_1 = nn.LayerNorm(dim)
        self.norm_2 = nn.LayerNorm(dim)

    def forward(self, x):
        att_out, _ = self.self_attn(x, x, x)
        x = self.norm_1(x + att_out)
        x = self.norm_2(x + self.feed_fwd(x))
        return x
```

Q3: What is Positional Encoding and why is it indispensable?

Self-attention treats its inputs as a set, not a sequence — reordering tokens produces identical attention scores. Without position signals, the model has no way to distinguish "dog bites man" from "man bites dog".

The original Transformer injects position via fixed sinusoidal functions:

```
PE(pos, 2i) = sin(pos / 10000^(2i/d)), PE(pos, 2i+1) = cos(pos /
                          10000^(2i/d))
```

These vectors are **added** to the token embeddings. Sinusoidal encodings can generalise to sequence lengths unseen during training. Learned positional embeddings perform similarly but are bounded by training-set length.

```
import torch, math

def create_sinusoidal_pe(max_len, dim):
    pe = torch.zeros(max_len, dim)
    positions = torch.arange(0, max_len).unsqueeze(1).float()
    freqs = torch.exp(torch.arange(0, dim, 2).float() * (-math.log(10000) / dim))
    pe[:, 0::2] = torch.sin(positions * freqs)
    pe[:, 1::2] = torch.cos(positions * freqs)
    return pe  # (max_len, dim)

encoding = create_sinusoidal_pe(50, 512)
print(encoding.shape)  # torch.Size([50, 512])
```

Q4: How does the Transformer's parallelism improve training speed over RNNs?

An RNN's hidden state at time t depends on time t-1, creating a strict serial dependency. This means GPU utilisation is poor for long sequences and training time scales linearly with sequence length in a serial manner.

Transformers compute attention for all positions at once via matrix multiplications, allowing the full sequence to be processed in a single forward pass. Training complexity is O(n^2 * d) but is highly parallel, whereas the RNN's O(n) is entirely sequential. In practice, Transformers train much faster on modern GPU/TPU hardware.

```
import torch, time

rnn_model = torch.nn.RNN(512, 512, batch_first=True)
tfm_layer = torch.nn.TransformerEncoderLayer(d_model=512, nhead=8, batch_first=True)
data = torch.rand(8, 512, 512)

start = time.perf_counter()
for _ in range(50): rnn_model(data)
```

```
print(f"RNN  50 passes: {time.perf_counter()-start:.3f}s")

start = time.perf_counter()
for _ in range(50): tfm_layer(data)
print(f"TFM  50 passes: {time.perf_counter()-start:.3f}s")
```

Q5: How does Layer Normalisation differ from Batch Normalisation in Transformers?

Batch Norm normalises across the batch dimension per feature — problematic for variable-length sequences and small batches typical in NLP. Layer Norm normalises across the feature dimension for each individual sample:

```
LN(x_i) = gamma * (x_i - mu_i) / (sigma_i + eps) + beta
```

where mu_i and sigma_i are computed over the d_model features of sample i. This makes it batch-size independent and well-suited for sequence modelling.

```
import torch, torch.nn as nn

tensor = torch.rand(4, 10, 512)  # (batch, seq, features)
layer_norm = nn.LayerNorm(512)
batch_norm = nn.BatchNorm1d(512)

ln_out = layer_norm(tensor)                                    # direct
bn_out = batch_norm(tensor.transpose(1, 2)).transpose(1, 2)   # needs reshape
print("LN shape:", ln_out.shape, " BN shape:", bn_out.shape)
```

Q6: Explain residual (skip) connections and their role in Transformers.

Each sub-layer output follows the pattern y = LayerNorm(x + F(x)), where F is either the attention or FFN sub-layer. Residual connections serve three purposes:

- **Gradient highway:** Gradients flow directly through the identity branch, mitigating vanishing gradients in deep stacks (GPT-3 has 96 layers).
- **Easier optimisation:** The sub-layer only needs to learn a small residual correction rather than the full mapping.
- **Implicit ensembling:** The network effectively behaves as a collection of sub-networks of varying depths.

```
class SkipConnection(nn.Module):
    def __init__(self, sub_layer, dim=512):
        super().__init__()
        self.sub_layer = sub_layer
        self.layer_norm = nn.LayerNorm(dim)

    def forward(self, x, **kwargs):
        # Pre-Norm style (GPT-2 and later):
        return x + self.sub_layer(self.layer_norm(x), **kwargs)
```

Q7: What is the Feed-Forward Network inside a Transformer block?

The position-wise FFN is applied identically at each token position after attention:

```
FFN(x) = max(0, x * W1 + b1) * W2 + b2
```

The inner dimension d_ff is usually 4 * d_model (e.g. 2048 when d_model = 512). This expansion-contraction structure projects tokens into a higher-dimensional space for non-linear mixing. Modern models prefer GELU (BERT, GPT) or SwiGLU (LLaMA).

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

class FeedForward(nn.Module):
    def __init__(self, dim=512, hidden=2048, drop_rate=0.1):
        super().__init__()
        self.linear1 = nn.Linear(dim, hidden)
        self.linear2 = nn.Linear(hidden, dim)
        self.dropout = nn.Dropout(drop_rate)

    def forward(self, x):
        return self.linear2(self.dropout(F.gelu(self.linear1(x))))
```

Q8: Why is self-attention O(n^2) and what techniques reduce this cost?

Every token computes a score against every other token, producing an n x n matrix. Both time and memory are O(n^2). Efficient variants include:

- **Sparse Attention** (Longformer, BigBird) — local window plus selected global tokens.
- **Linformer** — projects K and V to O(k) dimensions, yielding O(n) complexity.
- **Flash Attention** — same O(n^2) math but dramatically fewer HBM reads/writes via tiling in SRAM.
- **Performer** — approximates softmax with random feature maps for O(n) attention.

```
def attention_flops(n, d):
    qk = n * n * d         # Q @ K^T
    softmax = n * n         # row-wise softmax
    av = n * n * d          # attn @ V
    return 2 * d * n**2 + n**2

for n in [128, 512, 1024, 4096]:
    print(f"n={n:5d}  FLOPs ~ {attention_flops(n, 64)/1e6:.1f}M")
```

Q9: Which application domains leverage the Transformer architecture?

Transformers now power state-of-the-art systems across virtually every modality:

Domain	Tasks	Example Models
NLP	Translation, QA, summarisation	BERT, GPT-4, T5
Vision	Classification, detection	ViT, DINO, CLIP
Speech	ASR, TTS	Whisper, SpeechT5
Biology	Protein structure prediction	AlphaFold2, ESMFold
Multimodal	Image + text understanding	GPT-4V, Flamingo, LLaVA

Q10: What distinguishes Pre-LN from Post-LN Transformer variants?

Post-LN (original paper): y = LN(x + Sublayer(x)). Normalisation applied after residual addition. Needs careful learning-rate warmup to avoid early instability.

Pre-LN (GPT-2, most modern LLMs): y = x + Sublayer(LN(x)). Normalisation applied before the sub-layer. More stable training; slightly lower peak performance but strongly preferred for large-scale models.

```
class PreNormTransformerBlock(nn.Module):
    def __init__(self, dim=512, heads=8, ff=2048):
        super().__init__()
        self.norm_attn = nn.LayerNorm(dim)
        self.norm_ffn  = nn.LayerNorm(dim)
        self.self_attn = nn.MultiheadAttention(dim, heads, batch_first=True)
        self.feed_fwd  = nn.Sequential(nn.Linear(dim, ff), nn.GELU(), nn.Linear(ff, dim))

    def forward(self, x):
        normed = self.norm_attn(x)
        attn_out, _ = self.self_attn(normed, normed, normed)
        x = x + attn_out
        x = x + self.feed_fwd(self.norm_ffn(x))
        return x
```

Category 2: Self-Attention Mechanism (Q11–Q20)

The mathematical core of the Transformer: Query-Key-Value formulation, scaling, masking, and interpretability.

Q11: Describe the Query, Key, Value (Q, K, V) abstraction in self-attention.

Each input token is linearly projected into three separate vectors via learned weight matrices W_Q, W_K, W_V:

- **Query (Q):** Encodes what information this token is searching for.
- **Key (K):** Encodes what information this token has to offer.
- **Value (V):** Encodes the actual content to be retrieved.

The dot product between a query and all keys determines how much of each value is aggregated. In self-attention, all three come from the same input sequence.

```
d, dk = 512, 64
W_query = nn.Linear(d, dk, bias=False)
W_key   = nn.Linear(d, dk, bias=False)
W_value = nn.Linear(d, dk, bias=False)

tokens = torch.rand(2, 10, d)
Q, K, V = W_query(tokens), W_key(tokens), W_value(tokens)  # each (2, 10, 64)
```

Q12: Write the scaled dot-product attention formula and explain every term.

```
Attention(Q, K, V) = softmax( Q K^T / sqrt(d_k) ) V
```

- **Q K^T:** Dot products yield an (n x n) similarity matrix — how well each query matches each key.
- **/ sqrt(d_k):** Scaling prevents large dot-product magnitudes from pushing softmax into near-zero-gradient saturation.
- **softmax(.):** Converts raw scores into a probability distribution summing to 1 per row.
- **(.) V:** Weighted combination of values — each query retrieves a blended representation.

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

def scaled_attention(query, key, value, mask=None):
    dk = query.size(-1)
    attn_scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(dk)
    if mask is not None:
        attn_scores = attn_scores.masked_fill(mask == 0, float('-inf'))
    attn_weights = F.softmax(attn_scores, dim=-1)
    context = torch.matmul(attn_weights, value)
    return context, attn_weights

q = torch.rand(2, 10, 64); k = torch.rand(2, 10, 64); v = torch.rand(2, 10, 64)
ctx, wts = scaled_attention(q, k, v)
print(ctx.shape, wts.shape)  # (2,10,64) (2,10,10)
```

Q13: Why is the 1/sqrt(d_k) scaling factor necessary?

When queries and keys have zero mean and unit variance, their dot product q.k has variance equal to d_k. With d_k = 64, the standard deviation reaches 8, producing scores large enough to push softmax into saturation where gradients nearly vanish. Dividing by sqrt(d_k) re-centres the variance to 1, keeping gradients healthy.

```
dim = 64
query = torch.randn(1, 1, dim); keys = torch.randn(1, 10, dim)
raw = torch.bmm(query, keys.transpose(1, 2))
normed = raw / math.sqrt(dim)
print(f"Unscaled std: {raw.std():.3f}  Scaled std: {normed.std():.3f}")
```

Q14: What are attention masks and when are they applied?

- **Padding mask:** Marks padded positions as -inf before softmax so they receive zero weight, preventing the model from attending to meaningless pad tokens.
- **Causal mask:** An upper-triangular mask used in the decoder that prevents position i from seeing positions > i, preserving the autoregressive property during training.

```
def make_causal_mask(size):
    return torch.triu(torch.ones(size, size, dtype=torch.bool), diagonal=1)

def make_padding_mask(seq_lengths, max_len):
    return torch.arange(max_len).unsqueeze(0) >= seq_lengths.unsqueeze(1)

print(make_causal_mask(5).int())
```

Q15: How does self-attention capture long-range dependencies?

In self-attention the path length between any two tokens is constant — O(1) — regardless of their distance in the sequence. Every token directly computes a score with every other token in a single layer. By contrast, an RNN must propagate through O(n) hidden-state transitions, causing gradient decay over long distances.

This direct any-to-any connectivity is why Transformers excel at tasks requiring distant context such as coreference resolution, long-document summarisation, and tasks where relationships span across paragraphs or entire documents.

***Example:** In the sentence 'The cat, which had been sleeping on the mat for hours, finally woke up and stretched its legs,' the word 'its' must attend to 'cat' across 13 intervening tokens — trivial for self-attention but challenging for an RNN.*

Q16: What is cross-attention (encoder-decoder attention)?

In cross-attention the decoder's previous sub-layer output provides **Q**, while the encoder's final output provides **K** and **V**. This lets every decoder position attend over all encoder positions, effectively "reading" the encoded source while generating each output token.

Cross-attention is the key mechanism that enables sequence-to-sequence tasks such as machine translation, summarisation, and any task where an output sequence must be conditioned on an input sequence. Without it, the decoder would have no way to access information from the source.

```
class EncoderDecoderAttention(nn.Module):
    def __init__(self, dim=512, heads=8):
        super().__init__()
        self.mha = nn.MultiheadAttention(dim, heads, batch_first=True)
        self.layer_norm = nn.LayerNorm(dim)

    def forward(self, decoder_hidden, encoder_memory):
        # Query from decoder, Key/Value from encoder
        attended, _ = self.mha(
            query=decoder_hidden, key=encoder_memory, value=encoder_memory)
        return self.layer_norm(decoder_hidden + attended)

cross_attn = EncoderDecoderAttention()
dec_h = torch.rand(2, 10, 512)     # decoder states
enc_m = torch.rand(2, 20, 512)     # encoder memory
output = cross_attn(dec_h, enc_m)   # (2, 10, 512)
```

Q17: What are the strengths and weaknesses of self-attention?

Strengths	Weaknesses
O(1) path length for long-range dependencies	O(n^2) time and memory w.r.t. sequence length
Fully parallelisable — no sequential bottleneck	Position information must be injected explicitly
Rich contextual representations via direct token-to-token scoring	Memory-intensive for very long sequences (genomes, books)

Q18: Implement single-head self-attention from scratch in PyTorch.

```
class SingleHeadAttention(nn.Module):
    def __init__(self, embed_dim=512, head_dim=64):
        super().__init__()
        self.query_proj = nn.Linear(embed_dim, head_dim, bias=False)
        self.key_proj   = nn.Linear(embed_dim, head_dim, bias=False)
        self.value_proj = nn.Linear(embed_dim, head_dim, bias=False)
        self.normaliser = math.sqrt(head_dim)

    def forward(self, x, mask=None):
        q, k, v = self.query_proj(x), self.key_proj(x), self.value_proj(x)
        logits = torch.bmm(q, k.transpose(1, 2)) / self.normaliser
        if mask is not None:
            logits = logits.masked_fill(mask, float('-inf'))
        attn = F.softmax(logits, dim=-1)
        return torch.bmm(attn, v), attn

module = SingleHeadAttention()
inp = torch.rand(2, 10, 512)
output, attention = module(inp)
print(output.shape, attention.shape)  # (2,10,64) (2,10,10)
```

Q19: What role does softmax play in attention, and can it be replaced?

Softmax normalises raw scores into a valid probability distribution (non-negative, sums to 1), ensuring the value-weighted sum stays well-scaled. However, it forces dense distributions where every position gets non-zero weight. Alternatives explored include:

- **Sparsemax** — produces genuinely sparse distributions where many weights are exactly zero.
- **alpha-entmax** — generalises softmax/sparsemax via a temperature-like parameter.
- **ReLU attention** — avoids the normalisation bottleneck entirely, used in some efficient Transformers.

Q20: How can attention weights be visualised for model interpretability?

The (n x n) attention weight matrix for each head can be rendered as a heatmap. High values at position (i, j) indicate token i strongly attends to token j. Tools such as BertViz provide interactive head-level views for BERT-family models.

```
import matplotlib.pyplot as plt, seaborn as sns

words = ["The", "cat", "sat", "on", "the", "mat", ".", "EOS"]
n_tokens = len(words)
attn_map = torch.softmax(torch.randn(n_tokens, n_tokens), dim=-1).numpy()

fig, ax = plt.subplots(figsize=(7, 5))
sns.heatmap(attn_map, xticklabels=words, yticklabels=words, cmap='Blues', ax=ax)
ax.set_title("Attention Heatmap — Head 2, Layer 4")
plt.tight_layout(); plt.savefig("attention_vis.png", dpi=150)
```

Category 3: Multi-Head Attention (Q21–Q28)

Why a single attention head is insufficient, the mathematics of parallel heads, and practical design choices.

Q21: What is Multi-Head Attention and why does it outperform a single head?

MHA runs h independent attention operations in parallel, each with its own projection matrices:

```
MultiHead(Q, K, V) = Concat(head_1, ..., head_h) W_O
```

```
head_i = Attention(Q W_Q_i, K W_K_i, V W_V_i)
```

Each head can specialise on different linguistic aspects (syntax, coreference, positional patterns), whereas a single head must average all of these. Total parameters are similar because d_k = d_model / h.

```
mha = nn.MultiheadAttention(embed_dim=512, num_heads=8, batch_first=True)
x = torch.rand(2, 10, 512)
out, attn = mha(x, x, x)
print("Output:", out.shape, " Weights:", attn.shape)
```

Q22: Walk through the step-by-step computation of Multi-Head Attention.

- **Step 1 — Linear projections:** Project X to Q_i, K_i, V_i for each head i where d_k = d / h.
- **Step 2 — Scaled dot-product:** Compute head_i = Attention(Q_i, K_i, V_i) independently.
- **Step 3 — Concatenation:** Stack all head outputs: [h1; h2; ...; hh] yielding dimension d_model.
- **Step 4 — Output projection:** Multiply by W_O to mix information across heads.

```
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, embed=512, heads=8):
        super().__init__()
        self.n_heads, self.head_dim = heads, embed // heads
        self.proj_q = nn.Linear(embed, embed, bias=False)
        self.proj_k = nn.Linear(embed, embed, bias=False)
        self.proj_v = nn.Linear(embed, embed, bias=False)
        self.proj_out = nn.Linear(embed, embed, bias=False)

    def reshape_for_heads(self, x, bsz, seq):
        return x.view(bsz, seq, self.n_heads, self.head_dim).transpose(1, 2)

    def forward(self, x):
        B, T, _ = x.shape
        q = self.reshape_for_heads(self.proj_q(x), B, T)
        k = self.reshape_for_heads(self.proj_k(x), B, T)
        v = self.reshape_for_heads(self.proj_v(x), B, T)
        attn = F.softmax(q @ k.transpose(-2, -1) / math.sqrt(self.head_dim), dim=-1)
        merged = (attn @ v).transpose(1, 2).contiguous().view(B, T, -1)
        return self.proj_out(merged)
```

Q23: How do you choose the number of attention heads h?

The standard constraint is d_k = d_model / h (integer division). Common configurations:

Model	d_model	h	d_k	Layers

Transformer-base	512	8	64	6
BERT-base	768	12	64	12
GPT-3 175B	12288	96	128	96
ViT-B/16	768	12	64	12

Q24: What patterns do individual attention heads typically learn?

Probing studies (Clark et al., 2019) reveal emergent head specialisation:

- **Syntactic heads:** Track dependency arcs (subject-verb, modifier-noun).
- **Positional heads:** Attend to immediately adjacent tokens (local n-gram patterns).
- **Coreference heads:** Link pronouns to antecedents ("it" -> "animal").
- **Rare-token heads:** Disproportionately attend to infrequent, informative tokens.

Q25: Why is the output projection W_O needed in MHA?

After concatenating the h head outputs (each d_k-dimensional) we get a d_model vector. W_O mixes information across heads, allowing the model to learn how different heads' specialised views should be combined. Without it, each head would remain an isolated channel with no cross-head interaction.

Consider an analogy: each head is like a domain expert examining the input from a different angle. W_O is the committee chair who synthesises all expert opinions into a coherent recommendation. The projection has shape (d_model x d_model), adding d_model^2 parameters.

```
# Demonstrating W_O importance
d_model, h, d_k = 512, 8, 64

# After computing all heads and concatenating:
# concat_output shape: (batch, seq, h * d_k) = (batch, seq, 512)
concat_output = torch.rand(2, 10, d_model)

# W_O mixes information across heads
W_O = nn.Linear(d_model, d_model, bias=False)
final = W_O(concat_output)  # (2, 10, 512)
print(f"W_O params: {d_model * d_model:,}")  # 262,144
```

Q26: Explain Grouped Query Attention (GQA) as used in LLaMA-2/3.

GQA shares K/V projections across groups of query heads to reduce KV-cache memory during inference:

- **MHA:** h Q heads, h K heads, h V heads (full).
- **MQA:** h Q heads, 1 shared K/V head (extreme sharing).
- **GQA:** h Q heads, g K/V heads (1 < g < h), each group shares one K/V pair.

LLaMA-2 70B uses GQA with g = 8 and h = 64, cutting KV-cache to 12.5% of full MHA.

```
# Conceptual GQA: 8 Q heads, 2 KV groups
H_Q, H_KV, d = 8, 2, 512
dk = d // H_Q
Wq = nn.Linear(d, H_Q * dk, bias=False)
Wk = nn.Linear(d, H_KV * dk, bias=False)
Wv = nn.Linear(d, H_KV * dk, bias=False)
```

```
x = torch.rand(1, 10, d)
Q = Wq(x).view(1, 10, H_Q, dk).transpose(1, 2)
K = Wk(x).view(1, 10, H_KV, dk).transpose(1, 2)
V = Wv(x).view(1, 10, H_KV, dk).transpose(1, 2)
K = K.repeat_interleave(H_Q // H_KV, dim=1)
V = V.repeat_interleave(H_Q // H_KV, dim=1)
```

Q27: What is attention dropout and why is it applied?

Attention dropout zeroes out random entries in the attention weight matrix after softmax and before the value-weighted sum. This prevents the model from becoming over-reliant on specific query-key pairs, improving generalisation. Typical rates: 0.1 for base models, 0.0 for large fine-tuning.

```
def attn_with_dropout(Q, K, V, drop=0.1, training=True):
    d_k = Q.size(-1)
    scores = Q @ K.transpose(-2, -1) / math.sqrt(d_k)
    w = F.softmax(scores, dim=-1)
    w = F.dropout(w, p=drop, training=training)
    return w @ V
```

Q28: Compare parameter counts: 1 head vs 8 heads with d_model = 512.

Both configurations have identical total parameter counts:

- 1 head, d_k = 512: 4 x 512^2 = 1,048,576 params (W_Q, W_K, W_V, W_O).
- 8 heads, d_k = 64: 8 x 3 x 512 x 64 + 512^2 = 786,432 + 262,144 = 1,048,576 params.

Cost is the same, but 8 heads yield richer subspace representations.

```
def count(m): return sum(p.numel() for p in m.parameters())

single = nn.MultiheadAttention(512, 1, bias=False, batch_first=True)
multi  = nn.MultiheadAttention(512, 8, bias=False, batch_first=True)
print(f"1-head: {count(single):,}  8-heads: {count(multi):,}")
# Both: 1,048,576
```

Category 4: Encoder–Decoder Architecture (Q29–Q36)

The full Transformer pipeline: encoder stack, decoder stack, masked attention, and generation.

Q29: What does the encoder produce and how is its output used?

The encoder processes the full source sequence in parallel. Its N identical blocks apply MHA + FFN with residuals and layer norm. The output is a tensor of contextualised embeddings Z of shape (n x d), one per source token, enriched with global context from all other tokens in the sequence.

This representation is passed to every decoder layer as the key and value inputs for cross-attention. Crucially, the encoder output is computed only once and reused at every decoder step during inference, making it efficient.

```
enc_layer = nn.TransformerEncoderLayer(d_model=512, nhead=8, batch_first=True)
encoder = nn.TransformerEncoder(enc_layer, num_layers=6)

src = torch.rand(2, 15, 512)  # (batch, src_len, d_model)
# Optional: src_key_padding_mask to handle padded tokens
memory = encoder(src)          # (2, 15, 512) - contextualised
print("Encoder memory shape:", memory.shape)
```

Q30: Describe the three attention sub-layers inside a decoder block.

- **Masked self-attention:** The decoder attends to its own prior outputs. A causal mask blocks position t from seeing positions > t (autoregressive constraint).
- **Encoder-decoder cross-attention:** Q from the decoder; K and V from the encoder memory Z. Lets the decoder read the full source.
- **Position-wise FFN:** Identical to the encoder's FFN, applied independently per position.

Q31: Why must the decoder use masked self-attention?

During training, the entire target is supplied at once (teacher forcing). Without masking, position t could peek at ground-truth tokens t+1, t+2, ... — effectively cheating. The causal mask ensures predictions at position t depend only on positions <= t, replicating the left-to-right process used at inference time.

```
d, h, N = 512, 8, 6
dec_layer = nn.TransformerDecoderLayer(d_model=d, nhead=h, batch_first=True)
decoder = nn.TransformerDecoder(dec_layer, num_layers=N)
tgt = torch.rand(2, 10, d)
memory = torch.rand(2, 15, d)
tgt_mask = nn.Transformer.generate_square_subsequent_mask(10)
out = decoder(tgt, memory, tgt_mask=tgt_mask)
print(out.shape)  # (2, 10, 512)
```

Q32: How does the Transformer generate tokens at inference time?

The decoder generates autoregressively, one token at a time:

- 1. Begin with a [BOS] (beginning-of-sequence) token.
- 2. Feed the encoder output (computed only once) plus the current decoder sequence.
- 3. Apply a linear + softmax layer to the last hidden state to get a distribution over the vocabulary.
- 4. Sample or argmax the next token from this distribution.

- 5. Append the new token to the sequence and repeat until [EOS] or max length is reached.

The KV-cache optimisation stores K/V from prior steps to avoid redundant recomputation, reducing per-step cost from O(t * d) to O(d). Without KV-cache, generating a sequence of length n would require O(n^2) total computation; with it, only O(n) total.

```
# Conceptual autoregressive generation loop
def generate(model, encoder_output, max_len=50, vocab_size=32000):
    bos_token = 1  # [BOS] token ID
    tokens = [bos_token]

    for _ in range(max_len):
        tgt = torch.tensor([tokens]).unsqueeze(0)  # (1, t)
        # Decoder forward with causal mask
        logits = model.decode(tgt, encoder_output)  # (1, t, vocab)
        next_token = logits[0, -1, :].argmax().item()
        tokens.append(next_token)
        if next_token == 2:  # [EOS]
            break
    return tokens
```

Q33: How does the output layer produce token probabilities?

The final decoder hidden state h_t is projected through a linear layer to vocabulary size:

```
logits = h_t @ W_out + b, W_out in R^(d x |V|)
```

Softmax converts logits to probabilities. In many models (GPT-2, LLaMA) W_out is weight-tied to the input embedding matrix — the same matrix is used for both embedding lookup and output projection, reducing parameters and improving generalisation.

Weight tying works because semantically similar tokens should have similar embeddings AND similar output probabilities. The embedding matrix naturally captures both relationships. This technique saves |V| x d_model parameters — for GPT-2, that is 50,257 x 768 = 38.6M parameters, a significant reduction.

```
vocab_size, d_model = 32000, 512
embedding = nn.Embedding(vocab_size, d_model)
output_proj = lambda h: h @ embedding.weight.T  # weight tying

h = torch.rand(2, 10, d_model)
logits = output_proj(h)       # (2, 10, 32000)
probs = F.softmax(logits, dim=-1)
next_tok = probs[:, -1, :].argmax(dim=-1)
print("Next tokens:", next_tok)
```

Q34: Compare encoder-only, decoder-only, and encoder-decoder Transformers.

Type	Example	Attention	Best For
Encoder-only	BERT, RoBERTa	Bidirectional	Classification, NER, QA
Decoder-only	GPT-2/3/4, LLaMA	Causal (masked)	Text generation, LM
Encoder-Decoder	T5, BART	Bidir + Cross	Translation, Seq2Seq

Q35: What is the KV-Cache and how does it speed up autoregressive decoding?

During greedy or beam decoding, K and V for all previous tokens never change — they only depend on the input, not on future tokens. Recomputing them at every step is wasteful.

The KV-cache stores K and V tensors from all past decoder steps. At step t, only the new token's K/V pair is computed and appended, reducing per-step cost from O(t * d) to O(d).

Memory cost is O(n * L * h_kv * d_k * 2) where L is the number of layers and h_kv is the number of KV heads. For LLaMA-2 70B with GQA (8 KV heads instead of 64), the KV-cache at 4096 context length in BF16 requires approximately 1.3 GB per batch element. Without GQA it would be 10.7 GB — demonstrating why GQA is essential for long-context deployment.

```
# Simplified KV-cache concept
def decode_step(x_new, past_k, past_v, attn_layer):
    k_new = attn_layer.key_proj(x_new)
    v_new = attn_layer.val_proj(x_new)
    K = torch.cat(past_k + [k_new], dim=1)
    V = torch.cat(past_v + [v_new], dim=1)
    q = attn_layer.query_proj(x_new)
    out = sdp_attention(q, K, V)
    return out, K, V
```

Q36: What is beam search and how is it used in Transformer decoding?

Beam search is a heuristic decoding strategy that maintains the top-k (beam width) most probable partial sequences at each generation step. At each step, every candidate is expanded by one token, and the top-k best sequences are retained based on cumulative log-probability.

Beam search continues until all beams end with [EOS] or reach maximum length. It approximates finding the globally most probable output sequence without exhaustive search. A beam width of 4-5 is typical for translation; larger beams give diminishing returns.

Alternatives to beam search used in LLMs include nucleus sampling (top-p), top-k sampling, and temperature-scaled sampling — these trade optimality for diversity, which is preferred for creative generation tasks.

```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

tok = AutoTokenizer.from_pretrained("t5-small")
model = AutoModelForSeq2SeqLM.from_pretrained("t5-small")
inputs = tok("translate English to French: Hello world", return_tensors="pt")
outputs = model.generate(**inputs, num_beams=4, max_new_tokens=20)
print(tok.decode(outputs[0], skip_special_tokens=True))
```

Category 5: Vision Transformer — ViT (Q37–Q43)

Applying the Transformer encoder to images: patch tokenisation, positional embeddings, classification, and design considerations.

Q37: What is a Vision Transformer (ViT) and how does it handle an image?

ViT (Dosovitskiy et al., 2020) feeds images directly to a standard Transformer encoder by converting them into a sequence of flat patch embeddings:

- **1. Patch splitting:** Divide the H x W x C image into N = HW / P^2 non-overlapping P x P patches (typically P = 16).
- **2. Linear projection:** Flatten each patch and project to d_model.
- **3. CLS token:** Prepend a learnable [CLS] embedding; its final output drives classification.
- **4. Positional embeddings:** Add learned 1-D positional embeddings.
- **5. Transformer encoder:** Process the N + 1 token sequence.

```
class ImageToPatchTokens(nn.Module):
    def __init__(self, img_size=224, p=16, in_ch=3, embed=768):
        super().__init__()
        num_patches = (img_size // p) ** 2
        self.patch_proj = nn.Conv2d(in_ch, embed, kernel_size=p, stride=p)
        self.class_token = nn.Parameter(torch.zeros(1, 1, embed))
        self.position_emb = nn.Parameter(torch.zeros(1, num_patches + 1, embed))

    def forward(self, images):  # images: (B, 3, 224, 224)
        B = images.size(0)
        patches = self.patch_proj(images).flatten(2).transpose(1, 2)  # (B, N, embed)
        cls = self.class_token.expand(B, -1, -1)
        tokens = torch.cat([cls, patches], dim=1)   # (B, N+1, embed)
        return tokens + self.position_emb
```

Q38: Why does ViT need positional embeddings for patches?

Like its NLP counterpart, ViT's self-attention is permutation-invariant — without position information, patch #5 and patch #37 would be indistinguishable. Positional embeddings encode each patch's spatial location. Surprisingly, learned 1-D embeddings work as well as 2-D ones; the model learns to encode the 2D grid internally. For variable resolution at test time, 2-D interpolation of positional embeddings is used.

Q39: What is the CLS token in ViT and how does it produce a classification?

The CLS token is a learnable parameter prepended to the patch sequence. Through self-attention across all layers, it aggregates information from every patch. Its final hidden state is fed into a lightweight MLP classification head:

$$\hat{y} = \mathrm{MLP}(\ \mathrm{LayerNorm}(\ z_L^0\ )\ )$$

```
class ViTHead(nn.Module):
    def __init__(self, dim=768, n_classes=1000):
        super().__init__()
        block = nn.TransformerEncoderLayer(dim, nhead=12, batch_first=True, norm_first=True)
        self.encoder = nn.TransformerEncoder(block, 12)
        self.ln = nn.LayerNorm(dim)
        self.fc = nn.Linear(dim, n_classes)

    def forward(self, patch_tokens):      # (B, N+1, dim)
        features = self.encoder(patch_tokens)
        cls_repr = self.ln(features[:, 0])  # first token = CLS
        return self.fc(cls_repr)            # (B, n_classes)
```

Implementation note: When fine-tuning ViT at a different resolution (e.g. 384x384 instead of the pre-trained 224x224), the number of patches changes from 196 to 576. The positional embeddings must be 2-D interpolated: reshape the 1-D embeddings to a 2-D grid, apply bicubic interpolation, then flatten back. This is a frequently asked interview detail.

```
# Positional embedding interpolation for resolution change
def interpolate_pos_embed(pos_embed, new_patches, old_size=14):
    cls_tok = pos_embed[:, :1, :]
    patch_emb = pos_embed[:, 1:, :]
    d = patch_emb.shape[-1]
    patch_emb = patch_emb.reshape(1, old_size, old_size, d).permute(0, 3, 1, 2)
    new_size = int(new_patches ** 0.5)
    patch_emb = F.interpolate(patch_emb, size=(new_size, new_size),
                              mode='bicubic', align_corners=False)
    patch_emb = patch_emb.permute(0, 2, 3, 1).reshape(1, -1, d)
    return torch.cat([cls_tok, patch_emb], dim=1)
```

Q40: What advantages does ViT have over CNNs?

- **Global receptive field:** Every patch attends to every other from layer 1; CNNs build global context only gradually through stacking convolutions.
- **Scalability:** ViT performance scales predictably with data and model size — the larger the dataset and model, the better it performs, following clear scaling laws.
- **Modality transfer:** The same architecture works for text, images, audio, and video with minimal changes, enabling unified multimodal models.
- **Interpretability:** Attention maps provide intuitive visualisations of where the model looks, which is harder to obtain from CNN feature maps.
- **Flexibility:** ViT can handle variable-resolution inputs (with positional embedding interpolation) and is easy to adapt for different tasks via different heads.

Q41: What are the limitations of ViT compared to CNNs?

- **Data hungry:** ViT lacks CNN inductive biases (translation equivariance, locality). On ImageNet-1K alone, a ViT-B underperforms ResNet-50. It needs ImageNet-21k or JFT-300M pre-training to shine.
- **Quadratic complexity:** O(N^2) attention w.r.t. patch count. A 224x224 image with P=16 gives 196 patches (manageable), but a 1024x1024 image with P=16 gives 4096 patches — 400x more attention entries.
- **High compute cost:** Pre-training from scratch requires substantial GPU resources — ViT-L was trained on Google's JFT-300M with TPU pods.
- **Positional encoding sensitivity:** Variable-resolution fine-tuning requires careful 2-D interpolation of positional embeddings, which can degrade performance if not done properly.
- **No built-in multi-scale:** Unlike CNNs with feature pyramids, standard ViT produces single-scale features. Swin Transformer addresses this with hierarchical design.

Q42: How does the Swin Transformer solve ViT's scalability problems?

Swin Transformer (Liu et al., 2021) introduces two fundamental innovations that make Transformers practical for dense vision tasks:

- **Shifted window attention:** Attention is computed within local non-overlapping windows (not globally), reducing complexity from O(N^2) to O(N). Windows are shifted between consecutive layers so that adjacent regions can communicate.
- **Hierarchical feature maps:** Patches are progressively merged as depth increases (similar to pooling in CNNs), producing multi-scale features at 1/4, 1/8, 1/16, and 1/32 resolution — essential for detection and segmentation.

Swin achieves SOTA on ImageNet classification and has become the default Transformer backbone for detection (Mask R-CNN + Swin) and segmentation (UPerNet + Swin) frameworks. Its linear complexity with respect to image size makes it practical for high-resolution inputs.

Q43: Implement a minimal ViT forward pass in PyTorch.

```
class SimpleViT(nn.Module):
    def __init__(self, image_size=224, patch=16, channels=3,
                 dim=768, n_heads=12, depth=12, num_classes=1000):
        super().__init__()
        n_patches = (image_size // patch) ** 2
        self.to_patches = nn.Conv2d(channels, dim, patch, stride=patch)
        self.cls_token = nn.Parameter(torch.zeros(1, 1, dim))
        self.positions = nn.Parameter(torch.zeros(1, n_patches + 1, dim))
        block = nn.TransformerEncoderLayer(
            dim, n_heads, dim_feedforward=dim*4, batch_first=True, norm_first=True)
        self.transformer = nn.TransformerEncoder(block, depth)
        self.final_norm = nn.LayerNorm(dim)
        self.classifier = nn.Linear(dim, num_classes)

    def forward(self, img):
        B = img.shape[0]
        patches = self.to_patches(img).flatten(2).transpose(1, 2)
        cls = self.cls_token.expand(B, -1, -1)
        tokens = torch.cat([cls, patches], 1) + self.positions
        encoded = self.transformer(tokens)
        return self.classifier(self.final_norm(encoded[:, 0]))

net = SimpleViT()
sample = torch.rand(2, 3, 224, 224)
print(net(sample).shape)  # (2, 1000)
```

Category 6: CNN vs. ViT & Hybrid Models (Q44–Q50)

Head-to-head comparison of CNNs and ViTs; hybrid architectures; practical model selection guidance.

Q44: What is inductive bias and why do CNNs possess more of it than ViTs?

Inductive bias is a set of assumptions hardwired into the architecture that constrains the hypothesis space, aiding generalisation with limited data. CNN biases:

- **Translation equivariance:** The same filter slides across all spatial locations; a feature detected anywhere is detected everywhere.
- **Locality:** Each neuron sees only a local receptive field, reflecting the prior that nearby pixels correlate most.

ViT lacks both by default — it treats patches as unordered tokens and must learn spatial structure from data alone.

Q45: When should you pick a CNN over ViT, and vice versa?

Scenario	Prefer CNN	Prefer ViT
Dataset size	Small/medium (<100k)	Large (>1M)
Compute budget	Constrained	High
Task type	Dense prediction (segmentation, detection)	Classification, retrieval
Domain shift	Low	High (with pre-training)
Edge deployment	MobileNet, EfficientNet	ViT-Tiny, Mobile-ViT

Q46: What are hybrid CNN-Transformer models? Give two examples.

Hybrids combine CNN local feature extraction with Transformer global context:

- **CNN + Transformer (sequential):** A CNN extracts feature maps, which are flattened and fed as tokens to a Transformer. Examples: ViT with ResNet stem, TransUNet for medical imaging.
- **ConvNeXt:** A pure CNN modernised with Transformer design choices (7x7 kernels, LayerNorm, GELU, inverted bottleneck) — competitive without attention.
- **Swin Transformer:** Uses shifted-window attention inside a CNN-like hierarchical pyramid.

```
class CNNTransformerHybrid(nn.Module):
    def __init__(self, n_classes=1000):
        super().__init__()
        import torchvision.models as models
        backbone = models.resnet50(weights='IMAGENET1K_V1')
        self.feature_extractor = nn.Sequential(*list(backbone.children())[:-2])
        tfm_block = nn.TransformerEncoderLayer(2048, 8, dim_feedforward=4096, batch_first=Tr
ue)
        self.attention_layers = nn.TransformerEncoder(tfm_block, 2)
        self.global_pool = nn.AdaptiveAvgPool1d(1)
        self.head = nn.Linear(2048, n_classes)

    def forward(self, images):
        feats = self.feature_extractor(images)            # (B, 2048, 7, 7)
        tokens = feats.flatten(2).transpose(1, 2)         # (B, 49, 2048)
        encoded = self.attention_layers(tokens)            # (B, 49, 2048)
        pooled = self.global_pool(encoded.transpose(1, 2))[:, :, 0]
        return self.head(pooled)
```

Q47: What is DINO and how does it enable self-supervised ViT training?

DINO (Self-DIstillation with NO labels, Caron et al., 2021) is a self-supervised learning framework that trains a Vision Transformer without any labelled data. The approach uses a student-teacher paradigm:

- A **student** ViT is trained to match the output distribution of a **teacher** ViT.
- The teacher is updated via exponential moving average (EMA) of the student weights — no gradient flows through the teacher.
- Both networks receive different augmented crops of the same image (multi-crop strategy with 2 global views + several local views).
- The loss is a cross-entropy between the student and teacher output distributions, with centering and sharpening applied to the teacher.

Remarkable emergent properties of DINO:

- The CLS token learns semantically meaningful representations without any labels — competitive with supervised features on downstream tasks.
- Attention maps of the last layer produce strikingly clear object segmentation masks without any segmentation supervision ("emergent segmentation").
- DINOv2 scales this approach further, producing powerful general-purpose visual features that serve as a foundation for many vision tasks.

Q48: Compare BERT (encoder-only) and GPT (decoder-only) architectures.

Property	BERT	GPT
Architecture	Encoder-only	Decoder-only
Attention	Bidirectional (all-to-all)	Causal (left-context only)
Pre-training	Masked LM + NSP	Autoregressive LM
Strength	Understanding (NLU)	Generation (NLG)
Fine-tuning	Add classification head	Prompt / instruction tuning
Examples	BERT, RoBERTa, ALBERT	GPT-2/3/4, LLaMA, Mistral

Q49: What is Flash Attention and how does it accelerate Transformer training?

Flash Attention (Dao et al., 2022) is an IO-aware exact attention algorithm. The key insight is that standard attention is bottlenecked by memory bandwidth, not arithmetic. On modern GPUs, SRAM (~20MB, very fast) is much faster than HBM (~80GB, slower). Standard attention writes the full n x n matrix to HBM, then reads it back for softmax — this IO dominates.

Flash Attention solves this with four techniques:

- **Tiling:** Q, K, V are divided into blocks that fit entirely in on-chip SRAM, avoiding HBM round-trips.
- **Online softmax:** Softmax is computed incrementally without ever storing the full n x n matrix, using the online softmax trick (Milakov & Gimelshein, 2018).
- **Kernel fusion:** The softmax, masking, dropout, and matmul operations are fused into a single GPU kernel, eliminating intermediate HBM writes.
- **Recomputation:** During the backward pass, the attention matrix is recomputed from Q, K, V rather than stored, reducing memory from O(n^2) to O(n).

The result: mathematically identical output to standard attention, but 2–4x faster wall-clock time and O(n) memory. This enables training with context windows of 32k, 128k, and even 1M+ tokens that would be impossible with standard attention.

```
# PyTorch 2.0+ dispatches to Flash Attention automatically
import torch, torch.nn.functional as F

Q = torch.rand(2, 8, 1024, 64, device='cuda', dtype=torch.float16)
K = torch.rand(2, 8, 1024, 64, device='cuda', dtype=torch.float16)
V = torch.rand(2, 8, 1024, 64, device='cuda', dtype=torch.float16)

with torch.backends.cuda.sdp_kernel(
        enable_flash=True, enable_math=False, enable_mem_efficient=False):
    out = F.scaled_dot_product_attention(Q, K, V, is_causal=True)
print(out.shape)  # (2, 8, 1024, 64)
```

Q50: What is LoRA and how does it make fine-tuning large Transformers efficient?

LoRA (Low-Rank Adaptation, Hu et al., 2021) fine-tunes large models by injecting trainable low-rank matrices into frozen weight matrices instead of updating all parameters:

```
W' = W + Delta_W = W + A B, A in R^(d x r), B in R^(r x k), r << min(d, k)
```

W remains frozen; only A and B are trained. With r = 8, a 768 x 768 weight drops from 590k to 12k trainable parameters — a 98% reduction — while matching or approaching full fine-tuning quality.

LoRA is typically applied to the attention projection matrices (W_Q, W_V) in each layer, though applying it to all four projections (W_Q, W_K, W_V, W_O) and FFN layers can improve results. The rank r controls the expressiveness-efficiency trade-off: r = 4-16 works well for most tasks.

Key advantages of LoRA: no additional inference latency (A and B can be merged into W after training), multiple LoRA adapters can be swapped at inference time for different tasks, and it enables fine-tuning 7B+ models on a single consumer GPU. QLoRA extends this further by combining LoRA with 4-bit quantization of the base model.

```
class LoRALayer(nn.Module):
    def __init__(self, input_dim, output_dim, rank=8, scale=16):
        super().__init__()
        self.base = nn.Linear(input_dim, output_dim, bias=False)
        self.base.weight.requires_grad = False  # freeze base weights
        self.down_proj = nn.Parameter(torch.randn(rank, input_dim) * 0.01)
        self.up_proj = nn.Parameter(torch.zeros(output_dim, rank))
        self.alpha = scale / rank

    def forward(self, x):
        frozen_out = self.base(x)
        lora_out = (x @ self.down_proj.T) @ self.up_proj.T
        return frozen_out + self.alpha * lora_out
```

```
adapter = LoRALayer(768, 768, rank=8)
sample = torch.rand(2, 10, 768)
print(adapter(sample).shape)  # (2, 10, 768)
n_train = sum(p.numel() for p in adapter.parameters() if p.requires_grad)
print(f"Trainable: {n_train:,}")  # 12,288
```

Supplementary: Model Configurations & Implementation Notes

Common Transformer Model Configurations

The following table summarises the configurations of widely-used Transformer models for quick reference during interviews:

Model	Type	d_model	Heads	Layers	FFN dim	Params
Transformer-base	Enc-Dec	512	8	6+6	2048	65M
BERT-base	Encoder	768	12	12	3072	110M
BERT-large	Encoder	1024	16	24	4096	340M
GPT-2 (small)	Decoder	768	12	12	3072	117M
GPT-2 (XL)	Decoder	1600	25	48	6400	1.5B
GPT-3	Decoder	12288	96	96	49152	175B
ViT-B/16	Encoder	768	12	12	3072	86M
ViT-L/16	Encoder	1024	16	24	4096	307M
Swin-T	Encoder	96	3/6/12/24	2/2/6/2	-	28M
LLaMA-2 7B	Decoder	4096	32	32	11008	6.7B
LLaMA-2 70B	Decoder	8192	64 Q/8 KV	80	28672	70B

Parameter Counting Quick Reference

Understanding where parameters come from is a frequent interview topic:

- **Embedding:** |V| x d_model. For GPT-2: 50,257 x 768 = 38.6M.
- **Attention (per layer):** 4 x d_model^2 = 4 x 768^2 = 2.36M (W_Q, W_K, W_V, W_O).
- **FFN (per layer):** 2 x d_model x d_ff = 2 x 768 x 3072 = 4.72M (W1, W2).
- **LayerNorm (per layer):** 4 x d_model = 3,072 (gamma + beta for 2 norms). Negligible.
- **Total per layer:** ~7.1M for BERT-base. Times 12 layers = ~85M + embedding = ~110M.

Complexity Comparison

Operation	Time	Memory	Notes
Self-Attention	O(n^2 d)	O(n^2 + nd)	Quadratic in seq length
FFN	O(n d^2)	O(nd)	Linear in seq length
Embedding	O(nd)	O(|V|d)	Lookup operation
LayerNorm	O(nd)	O(nd)	Per-sample normalisation
Full Layer	O(n^2 d + nd^2)	O(n^2 + nd)	Attention dominates for large n

Training vs Inference Memory

A common interview question concerns why training requires much more memory than inference. During training, the model must store: forward activations for backpropagation (can be reduced via gradient checkpointing), gradients for all parameters, and optimizer states (2x for Adam's first/second moment estimates). The rule of thumb: training memory is approximately 16-20 bytes per parameter with AdamW in mixed precision, while inference requires only 2 bytes per parameter (FP16/BF16) plus the KV-cache.

Memory Budget Breakdown for Common Models

Model	Params	Inference (FP16)	Training (Mixed)	Min GPUs
BERT-base	110M	~0.2 GB	~2 GB	1x any GPU
GPT-2 XL	1.5B	~3 GB	~30 GB	1x A100
LLaMA-2 7B	6.7B	~13 GB	~120 GB	2x A100 80GB
LLaMA-2 13B	13B	~26 GB	~240 GB	4x A100 80GB
LLaMA-2 70B	70B	~140 GB	~1.3 TB	16x A100 80GB
LLaMA-2 70B + QLoRA (4-bit)	70B	~35 GB	~48 GB	1x A100 80GB

The QLoRA row shows why parameter-efficient methods are transformative: fine-tuning a 70B model becomes feasible on a single GPU by freezing weights in 4-bit and only training small LoRA adapters.

Key Differences for Interview Questions

Concept A	Concept B	Key Difference
Self-Attention	Cross-Attention	Self: Q,K,V from same input Cross: Q from decoder, K,V from encoder
Layer Norm	Batch Norm	LN: normalise over features per sample BN: normalise over batch per feature
Pre-LN	Post-LN	Pre-LN: more stable, no warmup needed Post-LN: original paper, needs warmup
Absolute Pos Enc	Relative Pos Enc	Absolute: sin/cos or learned per position Relative: encode distance (RoPE, ALiBi)
MHA	GQA	MHA: h KV heads GQA: g shared KV heads (g < h)
LoRA	Full Fine-tuning	LoRA: frozen W + low-rank AB (~2% params) Full: update all parameters
Greedy Decoding	Beam Search	Greedy: argmax per step (fast, suboptimal) Beam: keep top-k paths (slower, better)
CNN	ViT	CNN: local receptive field, translation eq. ViT: global attention, no spatial bias
Flash Attention	Standard Attention	Flash: IO-aware, O(n) memory, tiled in SRAM Standard: O(n^2) memory in HBM
Encoder-only	Decoder-only	Enc: bidirectional (BERT) for understanding Dec: causal (GPT) for generation

Interview Preparation Tips

How to Approach Transformer Interview Questions

Transformer-related questions in deep learning interviews typically span three levels of depth. Knowing what level is expected helps you calibrate your answer:

Level	What They Test	How to Prepare
Conceptual	Can you explain the intuition behind each component?	Practice explaining Q/K/V, multi-head, masking in plain English
Mathematical	Can you write the formulas and explain each term?	Memorise the attention formula, FFN, positional encoding. Understand the scaling factor.
Implementation	Can you code attention from scratch?	Practice implementing self-attention and MHA in PyTorch without looking at references.

Top 10 Most Frequently Asked Topics

Based on common interview patterns at top tech companies, these are the topics most likely to come up in a Transformer-focused deep learning interview:

- **1. Scaled dot-product attention formula** — be able to write it, explain each term, and explain why we scale by sqrt(d_k).
- **2. Multi-head attention mechanics** — explain how heads are split, why multiple heads help, and that parameter count is the same as single-head.
- **3. Positional encoding** — explain why it is needed (permutation invariance), sinusoidal vs learned vs RoPE.
- **4. Encoder vs decoder vs encoder-decoder** — know the three paradigms and which models use each (BERT, GPT, T5).
- **5. Causal masking** — explain why the decoder needs it and how it works mechanically.
- **6. ViT patch embedding** — explain how images become token sequences.
- **7. CNN vs ViT trade-offs** — inductive bias, data requirements, when to use which.
- **8. KV-cache** — explain what it stores, why it matters, and how GQA reduces its size.
- **9. LoRA** — explain the low-rank decomposition and why it enables efficient fine-tuning.
- **10. Flash Attention** — explain the IO-awareness insight and that it is mathematically exact.

Common Follow-Up Questions

Interviewers often probe deeper with questions like:

- "What happens if you remove the scaling factor from attention?" — Softmax saturates, gradients vanish, training fails.
- "Why not use batch normalisation in Transformers?" — Variable sequence lengths, small batches, batch-size dependence.
- "How would you handle a 100k-token document?" — Flash Attention, sparse attention (Longformer), or sliding window (Mistral).

- "What is the compute cost of attention vs FFN?" — Attention is O(n^2 d), FFN is O(n d^2). Attention dominates for long sequences, FFN dominates for short ones.
- "How does LoRA compare to full fine-tuning?" — 98%+ parameter reduction, comparable quality, no inference overhead (can merge weights).
- "Explain the difference between pre-training and fine-tuning." — Pre-training learns general representations on massive data; fine-tuning adapts to a specific task on smaller data.

Code Cheat Sheet: Essential Implementations

This section provides complete, copy-paste-ready PyTorch implementations of the most important components. These are the code blocks most commonly requested in interviews.

1. Complete Scaled Dot-Product Attention with All Options

```python
def scaled_dot_product_attention(Q, K, V, mask=None, dropout_p=0.0, training=True):
    """
    Args:
        Q: (B, ..., T, d_k)  queries
        K: (B, ..., S, d_k)  keys
        V: (B, ..., S, d_v)  values
        mask: broadcastable to (B, ..., T, S), True = masked out
    Returns:
        output: (B, ..., T, d_v), attention_weights: (B, ..., T, S)
    """
    d_k = Q.size(-1)
    # Step 1: Compute raw attention scores
    scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(d_k)

    # Step 2: Apply mask (causal or padding)
    if mask is not None:
        scores = scores.masked_fill(mask, float('-inf'))

    # Step 3: Normalise to probability distribution
    attn_weights = F.softmax(scores, dim=-1)

    # Step 4: Optional dropout on attention weights
    if dropout_p > 0.0 and training:
        attn_weights = F.dropout(attn_weights, p=dropout_p, training=True)

    # Step 5: Weighted sum of values
    output = torch.matmul(attn_weights, V)
    return output, attn_weights
```

2. Complete Multi-Head Attention with Fused QKV Projection

```python
class EfficientMHA(nn.Module):
    """Multi-Head Attention with fused QKV for efficiency."""
    def __init__(self, d_model=512, num_heads=8, dropout=0.1):
        super().__init__()
        assert d_model % num_heads == 0
        self.h = num_heads
        self.d_k = d_model // num_heads
        self.qkv = nn.Linear(d_model, 3 * d_model, bias=False)  # Fused
        self.out = nn.Linear(d_model, d_model, bias=False)
        self.dropout = dropout

    def forward(self, x, mask=None):
        B, T, D = x.shape
        # Single matmul for all Q, K, V projections
        qkv = self.qkv(x).reshape(B, T, 3, self.h, self.d_k)
        q, k, v = qkv.permute(2, 0, 3, 1, 4).unbind(0)
        # Attention
        scores = (q @ k.transpose(-2, -1)) * (self.d_k ** -0.5)
        if mask is not None:
            scores = scores.masked_fill(mask.unsqueeze(1), float('-inf'))
        attn = F.softmax(scores, dim=-1)
        attn = F.dropout(attn, p=self.dropout, training=self.training)
        out = (attn @ v).transpose(1, 2).reshape(B, T, D)
        return self.out(out)
```

3. Complete Mini ViT (End-to-End)

```
class MiniViT(nn.Module):
    def __init__(self, img=224, patch=16, ch=3,
                 d=768, heads=12, layers=12, classes=1000):
        super().__init__()
        n_patches = (img // patch) ** 2
        self.patch_proj = nn.Conv2d(ch, d, patch, stride=patch)
        self.cls_token = nn.Parameter(torch.zeros(1, 1, d))
        self.pos_embed = nn.Parameter(torch.zeros(1, n_patches + 1, d))
        nn.init.trunc_normal_(self.pos_embed, std=0.02)
        nn.init.trunc_normal_(self.cls_token, std=0.02)

        encoder_layer = nn.TransformerEncoderLayer(
            d, heads, dim_feedforward=d*4,
            batch_first=True, norm_first=True, dropout=0.1)
        self.encoder = nn.TransformerEncoder(encoder_layer, layers)
        self.head = nn.Sequential(
            nn.LayerNorm(d), nn.Linear(d, classes))

    def forward(self, imgs):
        B = imgs.shape[0]
        patches = self.patch_proj(imgs).flatten(2).transpose(1, 2)
        cls = self.cls_token.expand(B, -1, -1)
        tokens = torch.cat([cls, patches], dim=1) + self.pos_embed
        encoded = self.encoder(tokens)
        return self.head(encoded[:, 0])

# Test: vit = MiniViT(); print(vit(torch.rand(1,3,224,224)).shape)
```

4. LoRA Wrapper for Any Linear Layer

```
class LoRA(nn.Module):
    """Wraps any nn.Linear with a low-rank adapter."""
    def __init__(self, original: nn.Linear, rank=8, alpha=16):
        super().__init__()
        self.original = original
        self.original.weight.requires_grad_(False)
        if self.original.bias is not None:
            self.original.bias.requires_grad_(False)

        in_f, out_f = original.in_features, original.out_features
        self.lora_A = nn.Parameter(torch.randn(rank, in_f) * 0.01)
        self.lora_B = nn.Parameter(torch.zeros(out_f, rank))
        self.scale = alpha / rank

    def forward(self, x):
        return self.original(x) + self.scale * (x @ self.lora_A.T @ self.lora_B.T)

    def merge(self):
        """Merge LoRA weights into the original for zero-overhead inference."""
        self.original.weight.data += self.scale * (self.lora_B @ self.lora_A)

# Usage: model.attn.Wq = LoRA(model.attn.Wq, rank=8)
```

5. Causal and Padding Masks

```
def make_causal_mask(seq_len, device='cpu'):
    """Returns (seq_len, seq_len) bool mask. True = blocked."""
    return torch.triu(torch.ones(seq_len, seq_len, device=device,
                                 dtype=torch.bool), diagonal=1)

def make_padding_mask(lengths, max_len, device='cpu'):
    """Returns (batch, max_len) bool mask. True = pad position."""
    idx = torch.arange(max_len, device=device).unsqueeze(0)
    return idx >= lengths.unsqueeze(1)

# Combine for decoder: causal + padding
def make_decoder_mask(tgt_len, tgt_pad_mask):
    causal = make_causal_mask(tgt_len, tgt_pad_mask.device)
    # Expand padding to (B, 1, 1, S) for broadcasting with (T, S) causal
    return causal.unsqueeze(0) | tgt_pad_mask.unsqueeze(1).unsqueeze(2)
```

Coding Exercise Checklist

Many interviews include a live coding component. Practice implementing these from memory:

Exercise	Difficulty	Key Concepts Tested
Scaled dot-product attention from scratch	Medium	Matrix multiplication, softmax, scaling, masking
Multi-head attention with head splitting	Medium-Hard	Tensor reshaping, parallel heads, output projection
Full Transformer encoder block	Hard	Residual connections, layer norm, FFN, Pre-LN vs Post-LN
ViT patch embedding with CLS token	Medium	Conv2d as projection, positional embeddings
LoRA linear layer	Easy-Medium	Frozen weights, low-rank A and B, scaling factor
Causal mask generation	Easy	Upper triangular matrix, masked_fill with -inf
Positional encoding (sinusoidal)	Medium	sin/cos interleaving, frequency computation
KV-cache decode step (conceptual)	Medium	Caching, concatenation, attention with growing K/V

Final Tip: *During a Transformer interview, always start by drawing the architecture diagram. Label the encoder and decoder stacks, show the attention connections, and mark where masking, residual connections, and layer norm are applied. This demonstrates structural understanding before diving into mathematical or coding details.*

Quick Reference: Key Formulas & Concepts

Concept	Formula / Key Fact
Scaled Dot-Product	Attn(Q,K,V) = softmax(QK^T / sqrt(d_k)) V
Multi-Head	MH = Concat(head_1, ...) W_O
Positional Encoding	PE(p,2i) = sin(p / 10000^(2i/d))
FFN	FFN(x) = max(0, xW1+b1) W2 + b2
Residual	y = LN(x + Sublayer(x))
ViT patches	N = HW / P^2, typically P = 16
Self-attn complexity	O(n^2 d) time, O(n^2) memory
LoRA update	W' = W + AB, r << d
BERT	Encoder-only, bidirectional, MLM
GPT	Decoder-only, causal LM, generation
T5	Encoder-decoder, text-to-text
ViT-B/16	d=768, 12 heads, 12 layers, 86M params

Jekardah AI Labs

Romi Nur Ismanto — Jekardah.com — rominur@gmail.com — 2026 Edition

AI Engineering Handbook — 50 Questions, 6 Categories, Python Code Snippets

Bibliography

The following references are cited throughout this handbook. They represent the foundational and state-of-the-art papers that define the Transformer and Vision Transformer landscape.

Foundational Architecture

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L. & Polosukhin, I. (2017). **Attention Is All You Need.** Advances in Neural Information Processing Systems (NeurIPS), 30.

[2] Ba, J.L., Kiros, J.R. & Hinton, G.E. (2016). **Layer Normalization.** arXiv preprint arXiv:1607.06450.

[3] He, K., Zhang, X., Ren, S. & Sun, J. (2016). **Deep Residual Learning for Image Recognition.** IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

[4] Bahdanau, D., Cho, K. & Bengio, Y. (2015). **Neural Machine Translation by Jointly Learning to Align and Translate.** International Conference on Learning Representations (ICLR).

[5] Sennrich, R., Haddow, B. & Birch, A. (2016). **Neural Machine Translation of Rare Words with Subword Units.** Annual Meeting of the Association for Computational Linguistics (ACL).

NLP Transformer Models

[6] Devlin, J., Chang, M.-W., Lee, K. & Toutanova, K. (2019). **BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.** Proceedings of NAACL-HLT.

[7] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D. & Sutskever, I. (2019). **Language Models are Unsupervised Multitask Learners.** OpenAI Technical Report.

[8] Brown, T.B., Mann, B., Ryder, N., Subbiah, M., et al. (2020). **Language Models are Few-Shot Learners.** Advances in Neural Information Processing Systems (NeurIPS), 33.

[9] Raffel, C., Shazeer, N., Roberts, A., Lee, K., et al. (2020). **Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer (T5).** Journal of Machine Learning Research, 21(140).

[10] Liu, Y., Ott, M., Goyal, N., Du, J., et al. (2019). **RoBERTa: A Robustly Optimized BERT Pretraining Approach.** arXiv preprint arXiv:1907.11692.

[11] Touvron, H., Lavril, T., Izacard, G., Martinet, X., et al. (2023). **LLaMA: Open and Efficient Foundation Language Models.** arXiv preprint arXiv:2302.13971.

[12] Touvron, H., Martin, L., Stone, K., Albert, P., et al. (2023). **LLaMA 2: Open Foundation and Fine-Tuned Chat Models.** arXiv preprint arXiv:2307.09288.

[13] Jiang, A.Q., Sablayrolles, A., Mensch, A., Bamford, C., et al. (2023). **Mistral 7B.** arXiv preprint arXiv:2310.06825.

Vision Transformers

[14] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., et al. (2021). **An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale.** International Conference on Learning Representations (ICLR).

[15] Liu, Z., Lin, Y., Cao, Y., Hu, H., et al. (2021). **Swin Transformer: Hierarchical Vision Transformer using Shifted Windows.** IEEE International Conference on Computer Vision (ICCV).

[16] Touvron, H., Cord, M., Douze, M., Massa, F., et al. (2021). **Training Data-Efficient Image Transformers & Distillation through Attention (DeiT).** International Conference on Machine Learning (ICML).

[17] Liu, Z., Mao, H., Wu, C.-Y., Feichtenhofer, C., Darrell, T. & Xie, S. (2022). **A ConvNet for the 2020s (ConvNeXt).** IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

[18] Carion, N., Massa, F., Synnaeve, G., Usunier, N., Kirillov, A. & Zagoruyko, S. (2020). **End-to-End Object Detection with Transformers (DETR).** European Conference on Computer Vision (ECCV).

Self-Supervised Learning & Multimodal Models

[19] Caron, M., Touvron, H., Misra, I., Jegou, H., et al. (2021). **Emerging Properties in Self-Supervised Vision Transformers (DINO).** IEEE International Conference on Computer Vision (ICCV).

[20] Oquab, M., Darcet, T., Moutakanni, T., Vo, H., et al. (2024). **DINOv2: Learning Robust Visual Features without Supervision.** Transactions on Machine Learning Research.

[21] Radford, A., Kim, J.W., Hallacy, C., Ramesh, A., et al. (2021). **Learning Transferable Visual Models From Natural Language Supervision (CLIP).** International Conference on Machine Learning (ICML).

[22] Liu, H., Li, C., Wu, Q. & Lee, Y.J. (2023). **Visual Instruction Tuning (LLaVA).** Advances in Neural Information Processing Systems (NeurIPS), 36.

[23] Peebles, W. & Xie, S. (2023). **Scalable Diffusion Models with Transformers (DiT).** IEEE International Conference on Computer Vision (ICCV).

Efficiency & Optimisation

[24] Dao, T., Fu, D.Y., Ermon, S., Rudra, A. & Re, C. (2022). **FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness.** Advances in Neural Information Processing Systems (NeurIPS), 35.

[25] Dao, T. (2023). **FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning.** arXiv preprint arXiv:2307.08691.

[26] Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., et al. (2022). **LoRA: Low-Rank Adaptation of Large Language Models.** International Conference on Learning Representations (ICLR).

[27] Dettmers, T., Pagnoni, A., Holtzman, A. & Zettlemoyer, L. (2023). **QLoRA: Efficient Finetuning of Quantized Language Models.** Advances in Neural Information Processing Systems (NeurIPS), 36.

[28] Frantar, E., Ashkboos, S., Hoefler, T. & Alistarh, D. (2022). **GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers.** arXiv preprint arXiv:2210.17323.

[29] Lin, J., Tang, J., Tang, H., Yang, S., Dang, X. & Han, S. (2024). **AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration.** Machine Learning and Systems (MLSys).

[30] Beltagy, I., Peters, M.E. & Cohan, A. (2020). **Longformer: The Long-Document Transformer.** arXiv preprint arXiv:2004.05150.

[31] Kwon, W., Li, Z., Zhuang, S., Sheng, Y., et al. (2023). **Efficient Memory Management for Large Language Model Serving with PagedAttention (vLLM).** Symposium on Operating Systems Principles (SOSP).

Attention Analysis & Variants

[32] Clark, K., Khandelwal, U., Levy, O. & Manning, C.D. (2019). **What Does BERT Look At? An Analysis of BERT's Attention.** Proceedings of the 2019 ACL Workshop BlackboxNLP.

[33] Michel, P., Levy, O. & Neubig, G. (2019). **Are Sixteen Heads Really Better than One?** Advances in Neural Information Processing Systems (NeurIPS), 32.

[34] Shazeer, N. (2019). **Fast Transformer Decoding: One Write-Head is All You Need (Multi-Query Attention).** arXiv preprint arXiv:1911.02150.

[35] Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebron, F. & Sanghai, S. (2023). **GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints.** Proceedings of EMNLP.

[36] Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B. & Liu, Y. (2024). **RoFormer: Enhanced Transformer with Rotary Position Embedding (RoPE).** Neurocomputing, 568.

[37] Press, O., Smith, N.A. & Lewis, M. (2022). **Train Short, Test Long: Attention with Linear Biases Enables Input Length Generalization (ALiBi).** International Conference on Learning Representations (ICLR).

Scaling Laws & Training

[38] Kaplan, J., McCandlish, S., Henighan, T., Brown, T.B., et al. (2020). **Scaling Laws for Neural Language Models.** arXiv preprint arXiv:2001.08361.

[39] Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., et al. (2022). **Training Compute-Optimal Large Language Models (Chinchilla).** Advances in Neural Information Processing Systems (NeurIPS), 35.

[40] Loshchilov, I. & Hutter, F. (2019). **Decoupled Weight Decay Regularization (AdamW).** International Conference on Learning Representations (ICLR).

[41] Zhang, B. & Sennrich, R. (2019). **Root Mean Square Layer Normalization (RMSNorm).** Advances in Neural Information Processing Systems (NeurIPS), 32.

[42] Shazeer, N. (2020). **GLU Variants Improve Transformer (SwiGLU).** arXiv preprint arXiv:2002.05202.

State Space Models & Emerging Architectures

[43] Gu, A. & Dao, T. (2024). **Mamba: Linear-Time Sequence Modeling with Selective State Spaces.** International Conference on Machine Learning (ICML).

[44] Gu, A., Goel, K. & Re, C. (2022). **Efficiently Modeling Long Sequences with Structured State Spaces (S4).** International Conference on Learning Representations (ICLR).

[45] Lieber, O., Lenz, B., Bata, H., Cohen, G., et al. (2024). **Jamba: A Hybrid Transformer-Mamba Language Model.** arXiv preprint arXiv:2403.19887.

[46] Peng, B., Alcaide, E., Anthony, Q., Albalak, A., et al. (2023). **RWKV: Reinventing RNNs for the Transformer Era.** Findings of EMNLP.

Applications & Alignment

[47] Lewis, P., Perez, E., Piktus, A., Petroni, F., et al. (2020). **Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks (RAG).** Advances in Neural Information Processing Systems (NeurIPS), 33.

[48] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. & Cao, Y. (2023). **ReAct: Synergizing Reasoning and Acting in Language Models.** International Conference on Learning Representations (ICLR).

[49] Ouyang, L., Wu, J., Jiang, X., Almeida, D., et al. (2022). **Training language models to follow instructions with human feedback (InstructGPT/RLHF).** Advances in Neural Information Processing Systems (NeurIPS), 35.

[50] Rafailov, R., Sharma, A., Mitchell, E., Ermon, S., Manning, C.D. & Finn, C. (2023). **Direct Preference Optimization: Your Language Model is Secretly a Reward Model (DPO).** Advances in Neural Information Processing Systems (NeurIPS), 36.

[51] Geva, M., Schuster, R., Berant, J. & Levy, O. (2021). **Transformer Feed-Forward Layers Are Key-Value Memories.** Proceedings of EMNLP.

[52] Meng, K., Bau, D., Andonian, A. & Belinkov, Y. (2022). **Locating and Editing Factual Associations in GPT.** Advances in Neural Information Processing Systems (NeurIPS), 35.

[53] Leviathan, Y., Kalman, M. & Matias, Y. (2023). **Fast Inference from Transformers via Speculative Decoding.** International Conference on Machine Learning (ICML).

AI Engineering Handbook and Interview Preparation

Transformers & Vision Transformers (ViT)

Romi Nur Ismanto

Jekardah AI Labs — Jekardah.com — rominur@gmail.com

2026 Edition