Generative AI | GenAI

Zusammenfassung

1. LATENT SPACE

Latent space: learned hidden vector space where inputs are encoded; sampling/moving yields meaningful variations.
- Nearby points ≈ semantically similar.
- Vector arithmetic can encode relations *king - man + woman ≈ queen*.

2. DEEP NEURAL NETWORKS (DNN) - RECAP

N-Grams: Fixed-size context windows → sparse reps, limited generalization; cannot capture long-range deps beyond window size n.

Hyperparameters: Learning rate, number of epochs, batch size, architecture choices *(layers, neurons per layer, activations)*, regularization *(L1, L2, dropout)*.

2.1. TRAINING A NEURAL NETWORK

Stochastic Gradient Descent (SGD): Update weights based on mini-batches to reduce loss

$$w_{t+1} \leftarrow w_t - \alpha \frac{\partial}{\partial w} \log(p(x_i))$$

where α is learning rate

Batch Gradient Descent: Update weights based on entire dataset

$$w_{t+1} \leftarrow w_t - \alpha \left(\frac{1}{N}\right) \sum_{i=1}^{N} \frac{\partial}{\partial w} \log(p(x_i))$$

More stable but computationally expensive

Mini-batch Gradient Descent: Compromise between SGD and Batch GD

$$w_{t+1} \leftarrow w_t - \alpha \left(\frac{1}{M}\right) \sum_{i=1}^{M} \frac{\partial}{\partial w} \log(p(x_i))$$

Balances stability and computational efficiency

2.2. ACTIVATION FUNCTIONS

Function	*Formula*	*Pros*	*Cons*
ReLU	$f(x) = \max(0, x)$	Computationally efficient, mitigates vanishing gradient	"Dying ReLU" problem where neurons can become inactive
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	Outputs in $(0, 1)$, useful for probabilities	Vanishing gradient for large inputs, not zero-centered
Tanh	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	Outputs in $(-1, 1)$, zero-centered	Vanishing gradient for large inputs
Leaky ReLU	$f(x) = \max(\alpha x, x)$ *($\alpha \approx 0.01$)*	Mitigates dying ReLU, allows small gradient when inactive	Introduces hyperparameter α
Softmax	$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$	Outputs probability distribution, used in final layer	Only suitable for output layer

Derivative of ReLU: $f'(x) = \begin{cases} 1 \text{ if } x>0 \\ 0 \text{ if } x \le 0 \end{cases}$
Derivative of Sigmoid: $f'(x) = f(x)(1 - f(x))$
Derivative of Tanh: $f'(x) = 1 - f(x)^2$

2.3. LOSS FUNCTIONS

Loss Function	*Formula*	*Use Case*
Mean Squared Error (MSE)	$L = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$	Regression
Mean Absolute Error (MAE)	$L = \frac{1}{n}\sum_{i=1}^{n}\lvert y_i - \hat{y}_i \rvert$	Regression *(robust to outliers)*
Binary Cross-Entropy	$L = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$	Binary classification
Categorical Cross-Entropy	$L = -\sum_{i=1}^{n}\sum_{c=1}^{C} y_{i,c}\log(\hat{y}_{i,c})$	Multi-class classification

Cross-Entropy measures the difference between two probability distributions. Lower values indicate better match between predicted and true distributions.

2.4. CONVOLUTIONAL NEURAL NETWORKS (CNN)

Why CNNs for images? Fully-connected networks ignore spatial structure and have too many parameters for high-resolution images.

Key concepts:
- **Convolution:** slide kernel/filter over input to detect local patterns
- **Padding:** add borders to maintain spatial dimensions *(SAME padding: output size = input size; VALID: no padding)*
- **Stride:** step size of kernel movement, controls overlap of receptive fields

$$\text{output size} = \left(\frac{n + 2p - f}{s}\right) + 1$$

  where n = input size, p = padding, f = filter size, s = stride
- **Pooling:** downsample feature maps to reduce dimensions and computation *(Max pooling: take maximum value; Average pooling: take mean)*

Typical CNN architecture: Input → Conv + ReLU → Pool → Conv + ReLU → Pool → Flatten → FC → Softmax

2.5. EVALUATION METRICS

Metric	*Formula*	*When to Use*
Accuracy	$\frac{\text{TP} + \text{TN}}{n}$	Balanced datasets
Precision	$\frac{\text{TP}}{\text{TP} + \text{FP}}$	When false positives are costly
Recall (Sensitivity)	$\frac{\text{TP}}{\text{TP} + \text{FN}}$	When false negatives are costly
F1 Score	$\frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$	Imbalanced datasets, balance precision/recall
Specificity	$\frac{\text{TN}}{\text{TN} + \text{FP}}$	True negative rate

Confusion Matrix:

	Predicted Positive	*Predicted Negative*
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

Precision-Recall Trade-off: Increasing classification threshold typically increases precision but decreases recall, and vice versa.

2.6. REGULARIZATION TECHNIQUES

L1 Regularization (Lasso):

$$L = L_{\text{original}} + \lambda \sum_i |w_i|$$

Promotes sparsity *(many weights become exactly zero)*

L2 Regularization (Ridge):

$$L = L_{\text{original}} + \lambda \sum_i w_i^2$$

Encourages small weights, prevents overfitting

Dropout: Randomly deactivate neurons during training with probability p *(typically $p = 0.5$)*. Forces network to learn robust features that work with different subnetworks.

Early Stopping: Monitor validation loss and stop training when it stops improving *(prevents overfitting to training data)*.

Data Augmentation: Artificially expand training set with transformations *(rotation, scaling, flipping for images)*.

3. TRANSFORMERS

A ***transformer*** is a model that uses attention to boost the speed with which the models can be trained.

3.1. FLAVORS OF TRANSFORMERS

- ***Encoder-Only*** (e.g., BERT): Good for understanding tasks like classification, QA (Embedding Models)
- ***Decoder-Only*** (e.g., GPT): Good for generation tasks like text generation (Causal ML / autoregressive)
- ***Encoder-Decoder*** (e.g., T5): Good for seq2seq tasks like translation (Seq2Seq, MT models)

3.2. INPUTS: TOKENS, EMBEDDINGS, POSITIONAL ENCODING

Tokenization: Text split into tokens *(subwords)*, mapped to integer ids
Embedding: Matrix E maps token ids to vectors in d_{model} dimensions
Positional Encoding: Explicit position information added to embeddings

Let ***pos*** be the position $(0..n-1)$, and ***i*** be the embedding dimension index.

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

Final input vectors $X = \text{Embedding}(\text{tokens}) + \text{PE}$ with $X \in \mathbb{R}^{\text{token} \times d_{\text{model}}}$

(There are also learned positional embeddings and newer variants, but sinusoidal is a classic baseline.)

3.3. SELF-ATTENTION

Q **(queries)**: XW_Q "What am I looking for?"
K **(keys)**: XW_K "What do I offer / how should others match me?"
V **(values)**: XW_V the content to be transferred if a match is strong

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Here $QK^T \in \mathbb{R}^{n \times n}$ are similarity scores; dividing by $\sqrt{d_k}$ stabilizes training; each row of the softmax matrix sums to 1.

3.4. ATTENTION HEADS

An ***attention head*** is one independent attention computation with its own parameters W_Q^h, W_K^h, W_V^h.

Multi-head attention (MHA) runs H heads in parallel:

$$Y_h = \text{Attention}(Q_h, K_h, V_h), \quad \text{MHA}(X) = \text{Concat}(Y_1, ..., Y_H)W_O.$$

3.5. FEED-FORWARD LAYER (FNN)

Position-wise ***MLP*** (multi layer perceptron) applied independently to each position:

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

Shapes (per token): $x \in \mathbb{R}^{d_{\text{model}}}$, hidden dim d_{ff} (often $\approx 4d_{\text{model}}$), output in $\mathbb{R}^{d_{\text{model}}}$.

Parameters per layer:
- $W_1 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$, $b_1 \in \mathbb{R}^{d_{\text{ff}}}$
- $W_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$, $b_2 \in \mathbb{R}^{d_{\text{model}}}$

$$\begin{aligned} \text{FNN} &= (d_{\text{model}} * d_{\text{ff}} + d_{\text{ff}}) + (d_{\text{ff}} * d_{\text{model}} + d_{\text{model}}) \\ &= 2 * d_{\text{model}} * d_{\text{ff}} + d_{\text{ff}} + d_{\text{model}} \end{aligned}$$

3.6. LAYER NORMALIZATION

Normalization across the feature dimension for each token independently

$$\mu = \frac{1}{d}\sum_{j=1}^{d} x_j, \quad \sigma^2 = \frac{1}{d}\sum_{j=1}^{d}(x_j - \mu)^2$$

$$\text{LN}(x) = \gamma \odot \frac{x - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta$$

Where γ and β are learnable parameters, μ and σ^2 are mean and variance of features. $\odot$ means element-wise multiplication.

3.6.1. BatchNorm

BatchNorm normalizes **per feature/channel** using statistics computed over the **mini-batch** (and, for images, often also over spatial positions). For a given feature k:

$$\mu_k = \left(\frac{1}{m}\right)\sum_{b=1}^{m} x_{b,k}, \quad \sigma_k^2 = \left(\frac{1}{m}\right)\sum_{b=1}^{m}(x_{b,k} - \mu_k)^2$$

$$\text{BN}(x_{b,k}) = \gamma_k \odot \frac{x_{b,k} - \mu_k}{\sqrt{\sigma_k^2 + \varepsilon}} + \beta_k$$

Here, m is the batch size and (b, k) indexes example b and feature k. BatchNorm behaves differently in training vs. inference: during inference it typically uses running averages of μ_k and σ_k^2 computed during training.

3.7. CROSS-ATTENTION

In encoder–decoder models: Encoder output $H \in \mathbb{R}^{n_{\text{src}} \times d}$, decoder states $D \in \mathbb{R}^{n_{\text{tgt}} \times d}$.

$$\text{CrossAttention}(D, H) = \text{Attention}(DW_Q, HW_K, HW_V).$$

The decoder can attend to the most relevant source tokens while generating each target token.

3.8. CAUSAL MASKING

Prevents seeing the future tokens. In self attention we apply a ***triangular mask*** so position i can only attend to position $\le i$. Scores for $j > i$ are set to $-\infty$ before ***softmax***, making their attention weight 0

3.9. EXAMPLE

Vocalbulary size: $|V| = 100$, model dimension: $d_{\text{model}} = 32$, number of heads: $h = 4 \Rightarrow d_{\text{head}} = \frac{d_{\text{model}}}{h} = 8$, feed-forward hidden dimension: $d_{\text{ff}} = 64$, Layer $L = 2$ (only Encoder), Seq-length $n = 10$.

Token-Embeddings: $V \times d_{\text{model}} = 100 \times 32 = 3200$
Positional-Embeddings: $n \times d_{\text{model}} = 10 \times 32 = 320$
$W_Q, W_K, W_V \Rightarrow 3 \times (d_{\text{model}} \times d_{\text{model}}) = 3(32 \times 32) = 3072$, Output $W_O : (h \times d_{\text{head}}) \times d_{\text{model}} = (4 \times 8) \times 32 = 1024$
Total attention params per layer: $3072 + 1024 = 4096$
FNN Parameter per layer: $2 \times d_{\text{model}} \times d_{\text{ff}} + d_{\text{ff}} + d_{\text{model}} = 2(32 \times 64) + 64 + 32 = 4192$
LayerNorm params per layer: $2 \times d_{\text{model}} = 2 \times 32 = 64$
Total per layer: $4096 + 4192 + 128 = 8416$
Overall parameters: $L * \text{Total per layer} + \text{Embedding} + \text{Positional Embedding} = 2 \times 8416 + 3200 + 320 = 20352$

4. LLMS

Auto-regressive LLMs: Predict next token given previous tokens. Trained with causal masking.

4.1. KV CACHE

In autoregressive decoding, recomputing K,V for all past tokens is wasteful. Cache per layer: $K_{1:t} = [K_1; ...; K_t], \quad V_{1:t} = [V_1; ...; V_t]$. At step $t + 1$, compute only K_{t+1}, V_{t+1} and reuse the cache:

$$y_t = \text{softmax}\left(\frac{Q_t K_{1:t}^T}{\sqrt{d_k}}\right)V_{1:t}$$

Benefit: faster inference; cost: extra memory.

4.2. TRAINING VS INFERENCE (DECODER-ONLY LM)

Training: predict all next tokens in parallel with a causal mask. $L = -\sum_{t=1}^{T}\log p(x_t \mid x_{<t})$.
Inference: generate step-by-step. $x_{t+1} \sim p(. \mid x_{\le t})$. Often use KV cache to reuse past K,V.

4.3. ENCODER-ONLY (BERT, DISTILBERT)

Encoder-only Transformers use bidirectional self-attention to produce contextual embeddings for each token (good for understanding tasks, not autoregressive generation).

BERT (Bidirectional Encoder Representations from Transformers): pretrained with Masked Language Modeling (MLM) and (in the original paper) Next Sentence Prediction (NSP). MLM: mask tokens and predict them using left+right context.

DistilBERT: a smaller BERT-like encoder trained via knowledge distillation (student mimics teacher). DistilBERT reduces depth (about half the layers) and uses distillation losses (incl. cosine loss aligning hidden states) during training.

4.4. SPECIAL TOKENS

Special tokens are reserved tokens used for structure/control (not normal text).

- <pad> / [PAD]: padding for batching (ignored via attention_mask)
- <unk> / [UNK]: unknown token
- <bos> / <s>: begin of sequence
- <eos> / </s>: end of sequence (often stops generation)
- [CLS]: sequence-level representation for classification (BERT-style)
- [SEP]: separates segments/sentences (BERT-style)
- [MASK]: masked LM pre-training target (BERT-style)

4.5. POST-TRAINING (MAKE AN ASSISTANT)

Language modeling != assisting users: we want the model to follow instructions and align with safety/helpfulness goals. Problem: high-quality "desired behavior" data is scarce/expensive compared to web-scale pre-training data.
Supervised Fine-Tuning (SFT): train on instructions → response pairs
- ***Full Fine-Tuning (FFT)***: update all model weights → expensive, **SFT** data is smaller then pre-trained
- ***Less is more*** idea (***LIMA***): little instruction data can teach format/behavior; most knowledge is in pre-trained weights.

4.6. PREFERENCE TUNING / RLHF (REINFORCEMENT LEARNING WITH HUMAN FEEDBACK)

- Train on **reward model** from human preferences (preferred vs rejected answers), then optimize the policy model
- Multiple reward models possible (helpfulness, safety, etc.)
- Methods: ***Proximal Policy Optimization (PPO)*** (classic RLHF), and ***Direct Preference Optimization*** (DPO) alternative without RL loop
- InstructGPT pipeline: collect demos (SFT) → collect comparisons (reward model) → optimize with RL (PPO).

4.7. PEFT (PARAMETER-EFFICIENT FINE-TUNING)

- Motivation: ***FFT*** is costly in time, memory, storage.
- Methods: ***Adapters***, ***LoRA***, ***QLoRA***, ***Prefix / Prompt tuning***
- ***Adapters***: small modules inside transformer; different adapters can specialize per task.
- ***LoRA***: freeze W, learn low-rank update: $W' = W + \alpha * B * A(\text{rank } r)$. Only A,B trained.
- ***QLoRA***: quantize original weights (e.g., 4-bit) to reduce memory, then apply LoRA.

5. PROMPT / CONTEXT ENGINEERING

6. RAG (RETRIEVAL-AUGMENTED GENERATION)

LLMs have limited context windows and may not know up-to-date facts. ***RAG*** augments generation by retrieving relevant documents from an external knowledge base.

6.1. CORE PIPELINE

Query → Embed → Search Vector DB → get relevant context → append to prompt → LLM answers ("grounded generation").

6.2. RETRIEVAL BASICS

- Dense retrieval uses ***embeddings*** and similarity search (e.g., cosine similarity) to find relevant documents.
- Vector databases (e.g., FAISS, Pinecone) store embeddings for efficient similarity search

6.3. CHUNKING

Need chunking because LLM context window is limited; can not feed whole long docs. Techniques:
- 1 vector per doc (too compressed / loses detail)
- truncate (loses info)
- split into chunks (lines/paragraphs), possibly overlapping windows.

6.4. RETRIEVING EVALUATION

$\text{Precision@K} = \frac{\#\text{ relevant in top-K}}{K}$ and $\text{Recall@K} = \frac{\#\text{ relevant retrieved in top-K}}{\#\text{ relevant in dataset}}$.

6.5. RETRIEVAL SHORTCOMINGS + FIX

- **Top-K** cutoff / threshold matters
- Exact phrase match → dense retrieval may fail → use **hybrid search** (semantic + keyword)
- Domain shift: retrieval trained on web/Wikipedia may perform worse on legal/medical unless trained with domain data.
- Long-context issue ("lost in the middle"): LLM may miss relevant info if it is buried in the prompt.

6.6. RERANKING (SOLUTION)

- Goal tradeoff: **retrieve many** (high retrieval recall) but **send few** to LLM (LLM uses short context better)
- Two-stage: first retrieve (dense/keyword/hybrid) → rerank → take top-n.
- How: cross-encoder scores each (query, doc) pair jointly → reorder by relevance score.

7. EVALUATION

7.1. QUALITY OF GENERATED TEXT: BLEU (MT / NLG)

- BLEU: compares **candidate** vs **reference** using overlapping **n-grams** (often for translation/summarization).
- BLEU_n intuition:
  - small n → more about meaning/word choice
  - large n → more about fluency/well-formedness
- Common practice: geometric mean of BLEU_n for n=1..4 (often called mean BLEU).
- **Brevity Penalty (BP)**: penalizes candidates shorter than reference.
- Final: **BLEU = BP x MEAN_BLEU**

7.1.1. BLEU limitations (know these)

- Doesn't truly capture meaning/semantics.
- Doesn't directly capture sentence structure.
- Weak for morphologically rich languages.
- Correlates imperfectly with human judgment.

7.2. SUMMARIZATION: ROUGE (RECALL-FOCUSED)

- ROUGE: compares machine summary to references via overlap; emphasizes **recall** (getting important content).
- ROUGE variants:
  - **ROUGE-N**: n-gram overlap (like BLEU but recall-oriented).
  - **ROUGE-L**: longest common subsequence (captures sequence/structure).
  - **ROUGE-S**: skip-bigram overlap (words in same order, not necessarily adjacent).

7.3. LLM EVALUATION: PERPLEXITY (PPL)

- Perplexity = exp(average negative log-likelihood) over a token sequence.
- Range: ~ 1 (best) to $\sim |V|$ (vocab size, worst-case).
  - perfect prediction → PPL = 1
  - uniform uncertainty (each next token prob 1/|V|) → PPL = |V|
- Intuition: "how many tokens the model is hesitating between" (lower = more confident).
- Depends on tokenizer + dataset/benchmark (not directly comparable across setups).

7.3.1. Fine-tuning & PPL

- Fine-tuning should reduce hesitation on the target data → **lower perplexity** expected after fine-tuning.

8. AGENTS

8.1. WHAT IS AN AI AGENT?

- Classical: perceives environment + acts (sensors/effectors); RL: chooses actions to maximize reward.
- LLM-agent: LLM **directs its own process** + **uses tools dynamically** to complete tasks.
- Core loop idea: plan → act (tools) → observe/evaluate → iterate.
- Agent needs: tools (actions), memory/state, reasoning/planning.

8.2. TOOLS + FUNCTION CALLING (WHY IT MATTERS)

- Pure LLM = "string in → string out" (no real-world actions).
- Tools = callable functions/APIs the agent can select and invoke.
- Naive tool use issues: output parsing, brittle multi-step chains, adding tools requires prompt+parsing updates.

8.3. $M \times N$ INTEGRATION PROBLEM → MCP

- With M models and N tools, direct integration is $M \times N$.
- MCP reduces this to M + N:
  - apps implement an MCP client once,
  - tools implement an MCP server once,
  - standardized interface prevents repeated custom integrations.
- MCP: open protocol enabling 2-way communication between LLM apps and tool servers.

8.4. MCP COMPONENTS (MUST-KNOW TERMS)

- **Host**: the LLM app (e.g., chat app/IDE) that uses tools.
- **Client**: inside host; manages connections, tool discovery, request routing (1:1 to servers).
- **Server**: exposes tools/resources/prompts; connects to local/remote services/data.
- **Resources**: exposed data/services.

8.4.1. MCP server capabilities (3)

- **Tools** = actions (callable functions)
- **Resources** = data (files, DB, docs, etc.)
- **Prompts** = templates/instructions

8.5. MCP FLOW (HIGH LEVEL)

- User prompt → client discovers tools from server
- Prompt + tool list → LLM
- LLM selects tool → client calls server → server calls external service
- Results → LLM → final grounded answer

8.6. TRANSPORT (TRADEOFFS)

- **HTTP/HTTPS**: universal + easy; higher latency; streaming not default; good for external/web APIs.
- **gRPC**: fast + streaming + typed; more setup (Protobuf); good for internal high-performance systems.
- **stdio**: minimal + local; typically single-client; good for teaching/local experiments.
- Slides note: streamable HTTP often used for remote MCP servers (SSE-style streaming).

8.7. SMALL MODELS + TOOLRAG IDEA

- Question: can small language models emulate function-calling well?
- ToolRAG ("RAG for tools") can miss auxiliary tools if embeddings don't match query.
  - Example: scheduling may also require get_email_address.
- Fix idea: tool selection as **classification** (which tools are needed), not just similarity search.

8.8. MINIMAL SERVER: FASTMCP

```python
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("Calculator")  # creates MCP server instance
# Auto-registers @mcp.tool / @mcp.resource / @mcp.prompt
```

8.9. TOOL EXAMPLE: @MCP.TOOL (ASYNC + CTX LOGGING)

```python
@mcp.tool()
async def add(a: float, b: float, ctx) -> str:
    result = a + b
    await ctx.info(f"Adding {a} and {b} = {result}")
    return f"{a} + {b} = {result}"

# Notes:
# - async: non-blocking I/O/logging
# - inputs validated/converted from JSON into Python types
# - ctx has ctx.info / ctx.warning / ctx.error and is session-scoped
```

8.10. PROMPT EXAMPLE: @MCP.PROMPT (DYNAMIC TEMPLATE)

```python
@mcp.prompt()
async def calculate_operation(operation: str) -> str:
    return f"""
Use any tools available to you to calculate the operation: {operation}.
Use the voice of an extremely advanced embodied AI that has convinced
itself that it is a pocket calculator.
"""
```

8.11. RESOURCE EXAMPLE: @MCP.RESOURCE (DATA VIA URI)

```python
import math

@mcp.resource("resource://math-constants")
async def math_constants() -> str:
    constants = {
        "π (Pi)": math.pi,
        "e (Euler's number)": math.e,
        "τ (Tau)": math.tau,
        "φ (Golden ratio)": (1 + math.sqrt(5)) / 2,
        "√2 (Square root of 2)": math.sqrt(2),
    }
    out = "Mathematical Constants:\n" + "=" * 25 + "\n\n"
    for name, value in constants.items():
        out += f"{name:<25} = {value:.10f}\n"
    out += "\nThese constants can be used in calculations with the calculator tools."
    return out
```

8.12. CLIENT PATTERN: CONNECT OVER STDIO (SUBPROCESS) + INITIALIZE SESSION

```python
from contextlib import AsyncExitStack
from mcp.client.stdio import stdio_client, StdioServerParameters
from mcp.client.session import ClientSession

class MCPClient:
    def __init__(self, command: str, server_args: list[str], env_vars: dict | None = None):
        self.command = command
        self.server_args = server_args
        self.env_vars = env_vars
        self._exit_stack = AsyncExitStack()
        self._session: ClientSession | None = None
        self.read = None
        self.write = None
        self._connected = False

    async def connect(self) -> None:
        if self._connected:
            raise RuntimeError("Client is already connected")

        params = StdioServerParameters(
            command=self.command,
            args=self.server_args,
            env=self.env_vars if self.env_vars else None,
        )
```

```python
        # start MCP server subprocess + get (read_stream, write_stream)
        self.read, self.write = await self._exit_stack.enter_async_context(
            stdio_client(params)
        )

        # create client session (JSON-RPC over stdio)
        self._session = await self._exit_stack.enter_async_context(
            ClientSession(read_stream=self.read, write_stream=self.write)
        )

        # capability negotiation + ready
        await self._session.initialize()
        self._connected = True

    async def close(self) -> None:
        await self._exit_stack.aclose()
        self._connected = False
```

8.13. LISTING TOOLS AND CALLING A TOOL (TYPICAL USAGE)

```python
async def list_tools(self):
    assert self._session is not None
    tools = await self._session.list_tools()
    # tools.tools is usually the list of tool descriptors (name, schema, etc.)
    return tools

async def call_add(self, a: float, b: float):
    assert self._session is not None
    # Tool name must match the @mcp.tool registration name (often the function name)
    result = await self._session.call_tool("add", {"a": a, "b": b})
    return result
```

8.14. CLIENT CALLBACKS (SERVER $\rightarrow$ CLIENT REQUESTS)

- Sampling: server asks host app to run an LLM completion (server stays model-independent)
- Elicitation: server asks user for extra info/confirmation
- Logging: server emits logs to client (debug/monitoring)

```python
async def _handle_logs(self, level: str, message: str, **kwargs):
    print(f"[{level}] {message}")

async def _handle_sampling(self, messages, **kwargs):
    # host decides which model to use + returns completion
    # (pseudo-code: call your LLM provider here)
    return {"content": "model completion here"}

async def _handle_elicitation(self, prompt: str, **kwargs):
    # ask user for extra info (CLI example)
    return input(prompt + "\n> ")

async def connect_with_callbacks(self) -> None:
    params = StdioServerParameters(command=self.command, args=self.server_args, env=self.env_vars or None)
    self.read, self.write = await self._exit_stack.enter_async_context(stdio_client(params))

    self._session = await self._exit_stack.enter_async_context(
        ClientSession(
            read_stream=self.read,
            write_stream=self.write,
            logging_callback=self._handle_logs,
            sampling_callback=self._handle_sampling,
            elicitation_callback=self._handle_elicitation,
        )
    )
    await self._session.initialize()
    self._connected = True
```

8.15. TRANSPORT: STREAMABLE HTTP (SHAPE ONLY)

- Same JSON-RPC messages, different transport.
- HTTP: client→server via POST; streaming responses via SSE possible.
- Auth: bearer/API key headers; OAuth commonly used to obtain tokens.

```python
# Pseudocode: the core idea is JSON-RPC requests over HTTP.
# (Exact client helpers differ; conceptually:)

request = {
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": {"name": "add", "arguments": {"a": 2, "b": 3}},
}

# send POST /mcp with Authorization: Bearer <token>
# optionally open SSE stream for incremental server messages
```

8.16. WHY ASYNCEXITSTACK?

- Manages multiple async context managers (stdio transport, session, etc.) cleanly.
- Ensures subprocess + streams close even if errors occur.

8.17. MULTIPLE SERVERS / CONCURRENCY

- Host may connect to many servers at once; run multiple sessions concurrently.
- Pattern: maintain a list of ClientSession objects (or a SessionGroup helper) and route tool calls per server.

9. AE / VAE

9.1. AUTOENCODER (AE)

- AE = encoder + decoder trained to **reconstruct** input (output ≈ input).
- "Magic": latent space gives compressed embeddings; sampling/interpolating in latent can generate variants (via decoder).

9.1.1. Convolution notes (for image AEs)

Encoder/decoder often use conv + (transposed conv) for down/up-sampling.

9.1.2. Activation: ReLU vs LeakyReLU

- ReLU: $f(x) = \max(0, x)$ (dead neurons possible for negative region).
- LeakyReLU: small slope for $x < 0$ (keeps gradients alive).

9.1.3. Problems with vanilla AE latent space (why VAE exists)

- Latent clusters uneven; distribution unknown → hard to sample "good" points.
- Gaps / discontinuities: many latent points decode poorly.
- Not forced to be smooth/continuous; nearby latent points may not decode similarly.
- Higher latent dimensions → "empty space" problem worsens.

9.1.4. Reconstruction losses (examples)

- RMSE (L2-style reconstruction).
- Binary cross entropy (often for normalized pixel outputs; asymmetric).

9.2. VARIATIONAL AUTOENCODER (VAE)

- Instead of mapping x → single latent point, map x → **distribution** in latent space.
- Each input produces parameters of a multivariate normal distribution.

9.2.1. Encoder outputs (per input)

- Latent $\dim = d$
- Encoder predicts:
  - mean vector: $z_{\text{mean}} \in \mathbb{R}^d$
  - variance (often via log-variance): $z_{\text{log\_var}} \in \mathbb{R}^d$
- Use **log variance** because variance must be positive, but log-var can be any real number.

9.2.2. Reparameterization trick (crucial)

- Sample using: $z = \mu + \sigma * \varepsilon$ where $\varepsilon \sim N(0, I)$
- With log variance:

$$\sigma = \exp(0.5 * z_{\text{log\_var}})$$

$$z = z_{\text{mean}} + \exp(0.5 * z_{\text{log\_var}}) * \varepsilon$$

9.3. VAE LOSS (2 PARTS)

- Total loss = **reconstruction loss** + **KL divergence** term.
- KL term pushes learned latent distributions toward standard normal N(0, I) → smoother, more "fillable" latent space.

9.3.1. KL divergence (common form shown)

- kl_loss: $-0.5 * \sum(1 + z_{\text{log\_var}} - z_{\text{mean}}^2 - \exp(z_{\text{log\_var}}))$
- Minimized (→0) when $z_{\text{mean}} = 0$ and $z_{\text{log\_var}} = 0$ for all dims.

9.4. NICE PROPERTIES (INTUITION)

- Sampling: pick $z \sim N(0, I)$ → decode → plausible outputs (less "gaps" than AE).
- Smoothness: nearby latent samples decode to similar outputs (ideally).

9.5. LATENT SPACE ARITHMETIC / EDITING

- Attribute direction vector (e.g., "smile"):
  - take average latent of smiling faces minus average latent of non-smiling faces → z_{diff}
- Edit: $z_{\text{new}} = z_{\text{original}} + \alpha * z_{\text{diff}}$

9.6. MORPHING / INTERPOLATION

- Linear interpolation between two latent points: $z_{\text{new}} = z_A * (1 - \alpha) + z_B * \alpha$
- Decode along the path → gradual transition from A to B.

10. VISION TRANSFORMERS + CLIP

10.1. WHY VISION TRANSFORMERS (VIT)?

- Transformers successful in NLP → applied to images.
- Naive self-attention on pixels is **quadratic** in # pixels → too expensive.
- Fix: split image into **patches** and treat patches as **tokens** (like words).

10.1.1. ViT core pipeline (must know)

1. Split image into patches (e.g., 16×16).
2. Flatten each patch and linearly project to d_{model}.
3. Add **[CLS]** token + **positional embeddings**.
4. Feed sequence into **Transformer Encoder** (encoder-only).
5. Use CLS output + MLP head for classification logits → probabilities.

Patch math (from original paper slides)

- Input image: $X \in \mathbb{R}^{H \times W \times C}$
- Patch size: $P \times P$
- # patches (sequence length): $N = \frac{H \times W}{P^2}$

10.1.2. ViT "flavors" (scale table idea)

- ViT-Base / Large / Huge vary by # layers, hidden size D, # heads, params.

10.1.3. ViT advantages

- Inherits Transformer **scaling** behavior.
- Can model **long-range/global dependencies** via self-attention across patches.

10.2. VIT VS CNN (KEY CONCEPTUAL DIFFERENCES)

10.2.1. Locality / receptive field

- CNNs assume **nearby pixels are related** (locality inductive bias).
- ViT makes no locality assumption → attention can be **global** from early layers.

Mean attention distance (definition)

For a query pixel/patch q: 1) compute distance d_i to each key k_i 2) weight by attention a_i 3) weighted distance = $\sum a_i * d_i$ 4) average over queries + images → layer mean attention distance.

10.2.2. Translational invariance

- CNNs: translation invariance (object recognized even if shifted).
- ViT: no built-in translation invariance → often needs **more data** to learn it.

10.2.3. When to use ViT vs CNN (rules of thumb)

- Limited data / small compute / real-time (edge/mobile) → CNNs.
- Need global spatial relationships + can use big pretraining/transfer → ViT.

10.3. IMPLEMENTATION NOTES (HUGGING FACE)

10.3.1. Pretrained ViT (ImageNet-1k)

- `ViTForImageClassification.from_pretrained("google/vit-base-patch16-224")`
- Pretrained on ImageNet-1k → 1000 classes.

10.3.2. Feature extractor / preprocessing

- ViT expects RGB, resized to 224×224, normalized (ImageNet stats).
- Newer API: `AutoImageProcessor` (unified).

```python
from transformers import AutoImageProcessor, ViTForImageClassification
from PIL import Image
import requests

image = Image.open(requests.get("https://example.com/image.jpg", stream=True).raw)

processor = AutoImageProcessor.from_pretrained("google/vit-base-patch16-224")
model = ViTForImageClassification.from_pretrained("google/vit-base-patch16-224")

inputs = processor(images=image, return_tensors="pt")  # pixel_values: [1, 3, 224, 224]
outputs = model(**inputs)
logits = outputs.logits
```

10.3.3. Embedding shape + CLS token

- Example shown: embeddings shape $\approx [1, 197, 768]$
- 196 patches (14×14 for 224 with 16×16 patches) + 1 CLS token.

10.3.4. Hybrid CNN + ViT (patch embedding via convolution)

- Idea: use a conv layer to embed patches (instead of explicit patch flattening).
- For 224×224 with 16×16 stride:
  - $14 \times 14 = 196$ patch tokens
  - often 768 conv filters → embedding dim 768.

10.4. MULTI-MODAL TRANSFORMERS: CLIP (CONTRASTIVE LANGUAGE-IMAGE PRETRAINING) (+ SIGLIP)

- Two encoders:
  - text encoder → text embedding v_i
  - image encoder (often ViT) → image embedding w_j
- Train on many image-text pairs; map both modalities into a shared embedding space.

10.5. CONTRASTIVE LEARNING OBJECTIVE (BATCH)

- Compute cosine similarities between all (text, image) combos in batch.
- Maximize similarity for matching pairs (i=j), minimize for mismatches.
- Implementation view: for each v_i, softmax over similarities to all w_j, cross-entropy with correct w_i as target.

10.6. SIGLIP (SIGMOID LOSS VARIANT)

- Instead of softmax over all negatives, uses binary (sigmoid) loss per pair:
- classify each pair as positive (i=j) or negative (i≠j)
- No global normalization across batch required.

10.7. ZERO-SHOT CLASSIFICATION WITH CLIP

- Turn labels into text prompts (e.g., "a photo of a ").
- Encode image + all label texts; choose label with max cosine similarity.

10.8. CLIP IN GENERATION (DALL·E NOTE)

- CLIP text encoder can be used to embed prompts (DALL·E 2 mentioned).

11. DIFFUSION

- Train a network to **denoise** images with different noise levels.
- Inference: start from **pure Gaussian noise** and iteratively denoise → sample from training distribution.
- Key difference vs VAE/GAN: VAE/GAN generate in **one** forward pass; diffusion does **many refinement steps** (can "correct itself").

11.1. TWO PROCESSES

1) **Forward diffusion** q (fixed): gradually add Gaussian noise until image ≈ standard normal noise. 2) **Reverse diffusion** p_θ (learned): neural net gradually removes noise to recover an image.

11.2. PREPROCESSING

- Normalize training images to **zero mean, unit variance** (per-pixel over dataset).

11.3. FORWARD DIFFUSION q

11.3.1. One-step noising

- Add small noise at each timestep $t = 1..T$ with variance β_t:
  - Sample $\varepsilon \sim N(0, I)$
  - $x_t = \sqrt{1 - \beta_t} * x_{\{t-1\}} + \sqrt{\beta_t} * \varepsilon$
- Scaling ensures: if $x_{\{t-1\}}$ has mean 0 and var 1, then x_t also stays mean 0, var 1.

11.3.2. Why "final image becomes Gaussian noise"?

- With enough steps T and a schedule β_t, x_T becomes indistinguishable from $N(0, I)$.

11.3.3. Jump-to-any-timestep (reparameterization)

- Define $\alpha_t = 1 - \beta_t$, and $|(\alpha)_t = \prod_{\{i=1..t\}} \alpha_i$
- Then we can sample directly from x_0: $x_t = \sqrt{|(\alpha)_t} * x_0 + \sqrt{1 - |(\alpha)_t} * \varepsilon$
- Benefit: training can pick a random t without simulating all intermediate steps.

11.3.4. Noise schedules

- β_t (or equivalently $|(\alpha)_t$) changes with time.
- **Linear** schedule (example): β_t increases linearly (e.g., 0.0001 → 0.02):
  - early: tiny noising steps
  - late: larger steps (image already very noisy)
- **Cosine** schedule: noise increases more gradually → often improves training efficiency + generation quality.

11.4. REVERSE DIFFUSION p_θ

11.4.1. Goal

- We want $p\left(x_{\{t-1\}} \mid x_t\right)$ (denoise), but true distribution is **intractable**.
- Learn an approximation with a neural network (parameters θ).

11.4.2. What the NN predicts (DDPM training simplification)

- Provide network with: noisy image x_t + timestep (or schedule value).
- Network predicts the noise $\varepsilon_{\theta(x_t, t)}$.
- Train with squared error: minimize $\|\varepsilon - \varepsilon_{\theta(x_t, t)}\|^2$.

11.4.3. Reverse process model form (Gaussian assumption)

- Assume reverse step is Gaussian:
  - $p_{\theta\left(x_{\{t-1\}} \mid x_t\right)} = N\left(x_{\{t-1\}}; \mu_{\theta(x_t, t)}, \Sigma_{\theta(x_t, t)}\right)$
- DDPM choice: keep variance fixed, learn only the mean (later "improved diffusion" also learns variance).

11.5. TRAINING (RECIPE)

- Sample real image $x_0 \sim q(x_0)$
- Sample timestep $t \sim \text{Uniform}(1..T)$
- Sample noise $\varepsilon \sim N(0, I)$
- Form x_t using known schedule
- Train NN to predict ε from (x_t, t) (SGD on batches)

11.6. ARCHITECTURE: U-NET

- Use U-Net rather than AE/VAE for pixel-precise noise prediction.
- U-Net structure:
  - downsampling blocks (conv + pooling/downsample)
  - upsampling blocks (conv + upsample / transposed conv)
  - **skip connections** copy features from down path to up path (preserve details)
- Residual blocks + skips help gradient flow (avoid vanishing gradients in deep nets).

11.6.1. Timestep encoding

- Use **sinusoidal embedding** to map scalar timestep/noise-level to a higher-dim vector (like Transformers).

11.7. GENERATION (SAMPLING)

- Start with $x_T \sim N(0, I)$
- For t = T..1:
  - predict noise (or mean)
  - compute $x_{\{t-1\}}$ (denoise one step)
- Model predicts total noise component; iterative updates move from noisy → clean.

11.8. LATENT DIFFUSION / STABLE DIFFUSION / IMAGEN

11.8.1. Latent Diffusion Models (LDM)

- Key idea: run diffusion in **latent space** instead of pixel space:
  - autoencoder encodes image → latent
  - diffusion operates on latent (cheaper)
  - decoder reconstructs final image

11.8.2. Stable Diffusion (key points)

- Released Aug 2022 (public weights via Hugging Face).
- Denoising U-Net can be lighter because it operates in latent space.
- Autoencoder handles encoding/decoding "heavy lifting".
- Can be guided by text prompt via text encoder (v1 used CLIP; later versions use OpenCLIP).

11.8.3. Imagen (pipeline idea)

- Frozen text encoder: T5-XXL.
- Text-to-image diffusion model (U-Net conditioned on text embeddings).
- Super-resolution diffusion upsamplers: 64×64 → 256×256 → 1024×1024 (still conditioned on text).

12. MULTI-MODALITY (DALL·E, FLAMINGO)

- Learn to convert between **different modalities** (e.g., text ↔ image, text ↔ video).
- Key requirement: learn a **shared representation** to "bridge" modalities.
- Text-to-image: generate high-quality images from a text prompt.

12.1. DALL·E 2

12.1.1. Architecture overview (3 parts)

- **Text encoder** → text embedding
- **Prior** → converts text embedding → image embedding
- **Decoder** → generates image conditioned on (text + predicted image embedding)

12.1.2. Text encoder

- Need discrete text → continuous vector (latent embedding).
- DALL·E 2 uses **CLIP** as text encoder.

12.1.3. The Prior (text emb → image emb)

- Goal: map **CLIP text embedding** to **CLIP image embedding**.
- Two options:
  - **Autoregressive prior**: encoder-decoder Transformer; generates image-embedding elements sequentially (teacher forcing).
  - **Diffusion prior**: diffusion process in embedding space; found to outperform AR prior and be computationally efficient.
- Diffusion prior intuition:
  - noise image embedding → random noise
  - learn to denoise step-by-step, conditioned on text embedding
  - loss: average MSE across denoising steps

12.1.4. Decoder (image generation)

- Decoder is a **diffusion model**:
  - U-Net = denoiser
  - Transformer text encoder provides conditioning
- Generates a base image at 64×64 conditioned on:
  - the text prompt
  - the predicted CLIP image embedding (from the prior)
- Then apply **Upsamplers** (two diffusion models):
  - 64×64 to 256×256,
  - 256×256 to 1024×1024

12.1.5. Image variations (how)

- Compute **CLIP image embedding** for an input image using CLIP image encoder.
- Feed that embedding into decoder → generate variations.

12.1.6. Limitations (know these)

- **Attribute binding**:
  - Must distinguish relationships in prompts (e.g., "red cube on blue cube" vs reversed).
  - DALL·E 2 can struggle with correct binding.
- **Text rendering**:
  - Often fails to reproduce text accurately in images.
  - Explanation on slides: CLIP embeddings capture high-level semantics, not exact spelling.

12.2. FLAMINGO (VISION-LANGUAGE MODEL)

- A VLM that can handle **interleaved text + visual inputs** (images + video frames).

12.3. CORE COMPONENTS (3)

- **Vision encoder** (frozen)
- **Perceiver Resampler**
- **Language model**

12.4. VIDEO HANDLING (SLIDES' RECIPE)

- Sample video at 1 frame/sec.
- Run each frame through the vision encoder → feature grids.
- Add learned temporal encodings, flatten, concatenate.

12.5. PERCEIVER RESAMPLER (WHY + HOW)

- Problem: images produce **many** spatial tokens (e.g., 14×14×1024) → too expensive for LM.
- Solution: compress to a **fixed-size** smaller set of visual tokens:
  - Learn latent queries (few) that attend over all image tokens.
  - Cross-attention form:

$$Z = \text{softmax}(QK^T)V$$

  where:
  - Q = latent queries (small count),
  - K,V = image tokens.
- Output: compact visual embeddings usable by the language model.