Fundamentals of Large Language Models

Adnane Ait Nasser | AI Research Consultant at ACENET

A regional partner of the
Digital Research Alliance of Canada

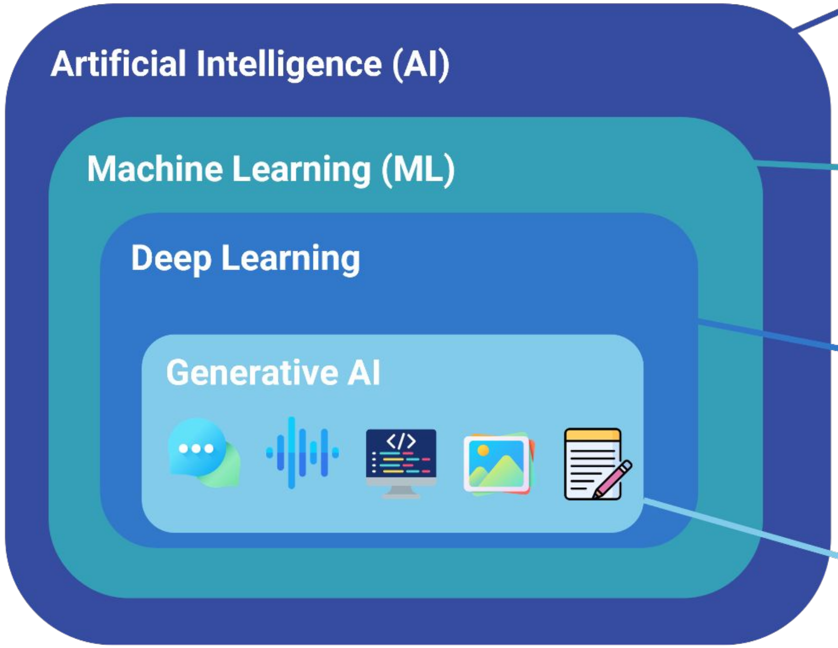
Overview

- What Are LLMs and Why They Matter ?
  - Definition and evolution (from RNNs → Transformers → LLMs)
  - **Examples:** GPT, BERT, T5, LLaMA, Falcon
- Core Architecture – Transformers.
  - Attention mechanism (self-attention, multi-head attention)
  - Encoder vs Decoder (BERT vs GPT-style)
  - Positional encoding
- Tokenization & Input Processing
  - What is a token?
  - Byte-Pair Encoding (BPE), SentencePiece, WordPiece
  - Special tokens (padding, start/end, mask)
- Pre-training vs Fine-tuning vs Prompting
- Capabilities and Use Cases
- Risks & Limitations

Learning Objectives

- Understand the architecture and capabilities of LLMs
- Learn how tokenization, attention, and transformers work
- Distinguish between pre-training, fine-tuning, and prompting
- Explore key use cases and limitations
- Fine-tune a small language model using Python and Hugging Face

General Introduction



Artificial Intelligence:

A multidisciplinary field of computer science that aims to create systems capable of emulating and surpassing human-level intelligence.

Machine Learning:

Learn from existing data and make predictions/prediction without being explicitly programmed.

Deep Learning:

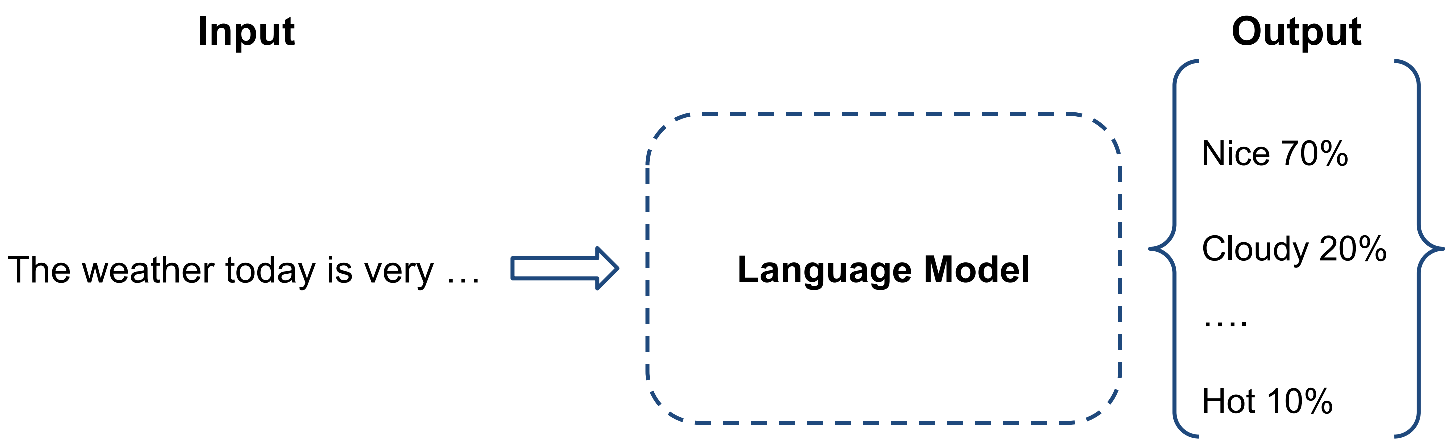
Uses "Artificial Neural Networks" to learn from data.

Generative Artificial Intelligence:

Sub-field of AI that focuses on generating new content such as: Text, Images, Audio/Music, Video Code, 3D Objects, and Synthetic Data

What is a Language Model?

- A language model (LM) predicts the next word or token in a sequence
- Learns statistical patterns in human language
- Input: partial text → Output: most likely next word(s)
- Can be trained on text from books, websites, conversations, etc.
- Foundation of modern Natural Language Processing (NLP)
- Power increases with scale, data, and model architecture

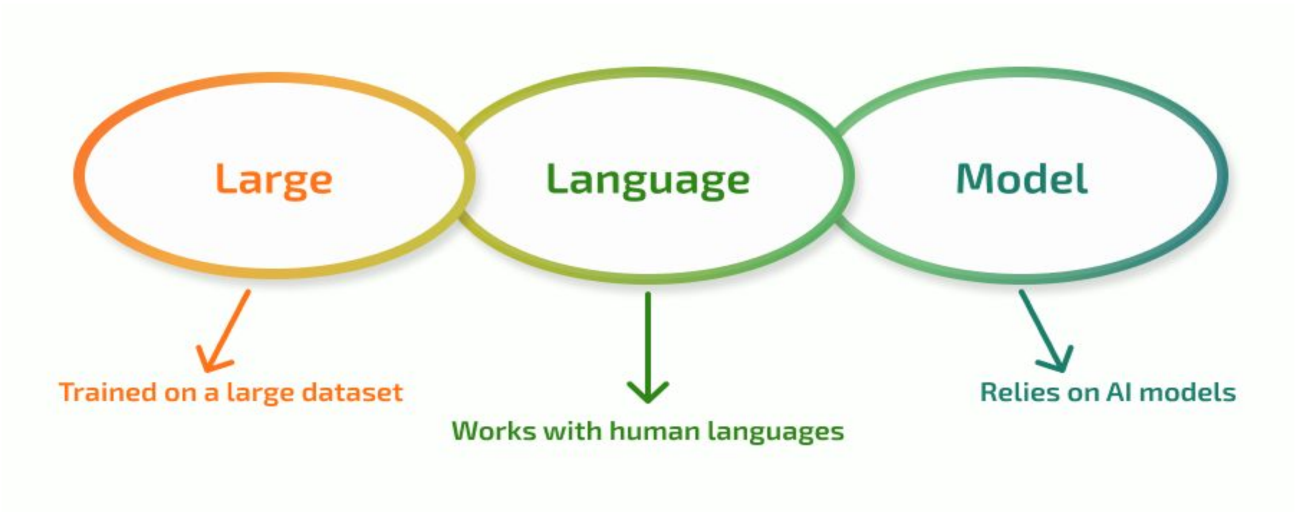


ACENET
accelerate discovery

A regional partner of the
Digital Research Alliance of Canada

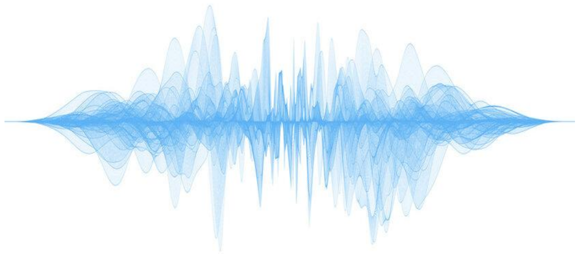
What is a Large Language Model (LLM) ?

- LLMs are deep learning models trained on massive text corpora to understand and generate human language.
  - Built using transformer architecture
  - Pre-trained on large datasets (books, web pages, code)
  - Scale: billions of parameters
  - Versatile: zero-shot, few-shot, and fine-tuning capable



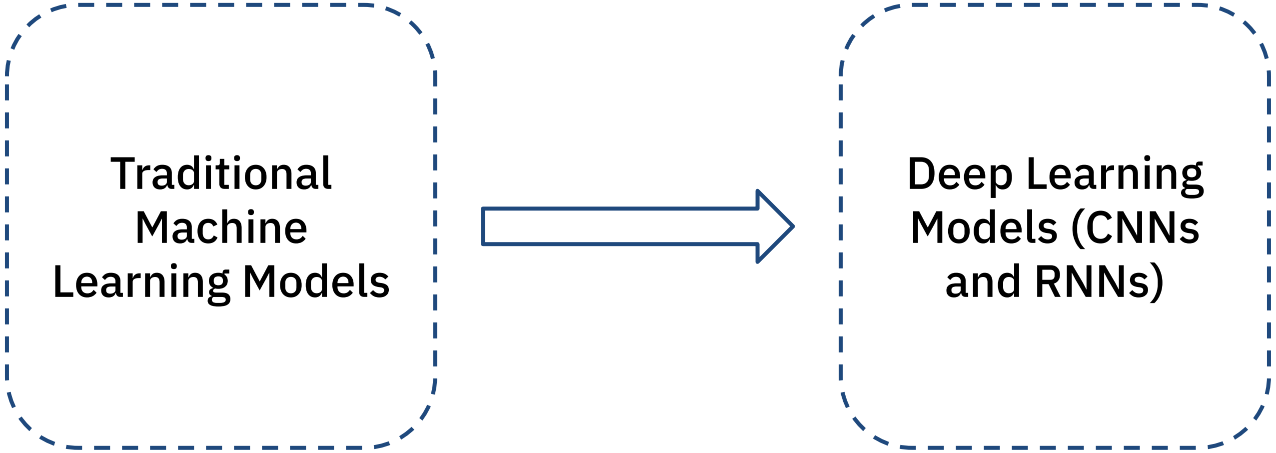
Why LLMs Matter?

- Foundation for modern NLP applications across industries
- Deliver state-of-the-art results on language understanding and generation tasks
- Power chatbots, virtual assistants, and conversational agents
- Automate tasks like summarization, translation, and content creation
- Accelerate coding, research, and knowledge retrieval
- Adapt easily to new domains with minimal labeled data



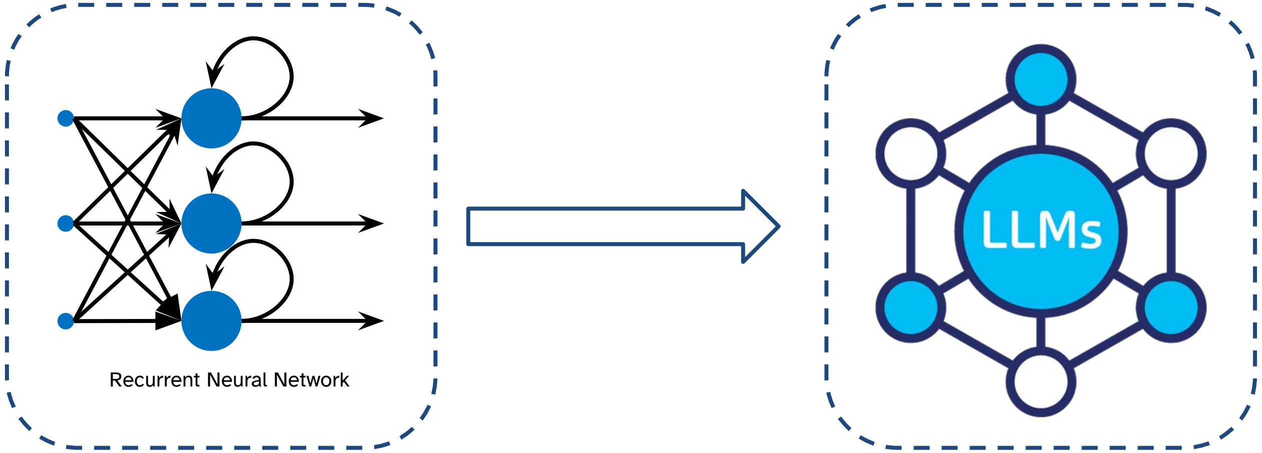
From Machine Learning to Deep Learning in NLP

- **Traditional ML** relied on manual feature engineering (e.g., n-grams, POS tags, TF-IDF).
- Classical models like **Logistic Regression, SVMs, and Naive Bayes** dominated early NLP.
- Performance was limited by domain expertise and shallow pattern extraction.
- DL introduced neural networks that learn features automatically from raw text.
- **Word embeddings** (e.g., **Word2Vec, GloVe**) enabled richer text representation.
- DL models like CNNs and RNNs replaced hand-crafted pipelines with end-to-end learning.



From Recurrent Neural Networks to Large Language Models

- **RNNs** process sequences step-by-step → slow and limited context.
- **LSTMs/GRUs** improved memory but still struggled with long texts.
- **Attention mechanism** introduced to weigh all tokens dynamically.
- **Transformers** removed recurrence → faster, parallelizable training.
- **LLMs = scaled transformers + massive pre-training data.**
- **Result:** flexible, general-purpose language understanding and generation.



Transformers Architecture (Overview)

- Introduced in 2017: **"Attention is All You Need"**.
- Replaces recurrence with self-attention.
- Introduced to fix the drawbacks of traditional ML/DL models.
- Parallelization problem is solved with the self-attention mechanism.
- Better memory of long range dependencies, no more vanishing gradient problem.

Core Components:

➔ Input embeddings + Positional encodings

➔ Multi-head self-attention

➔ Feedforward layers

➔ Residual connections + Layer Norm

ACENET
accelerate discovery

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

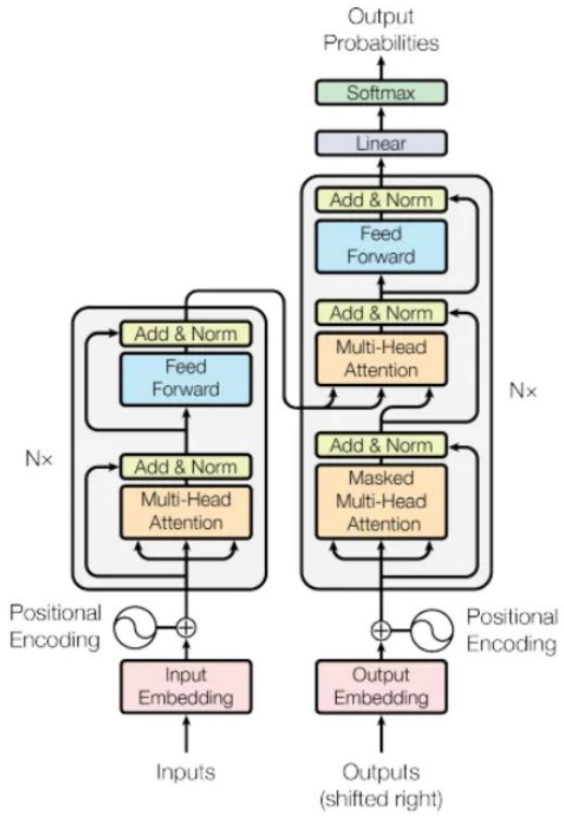
Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

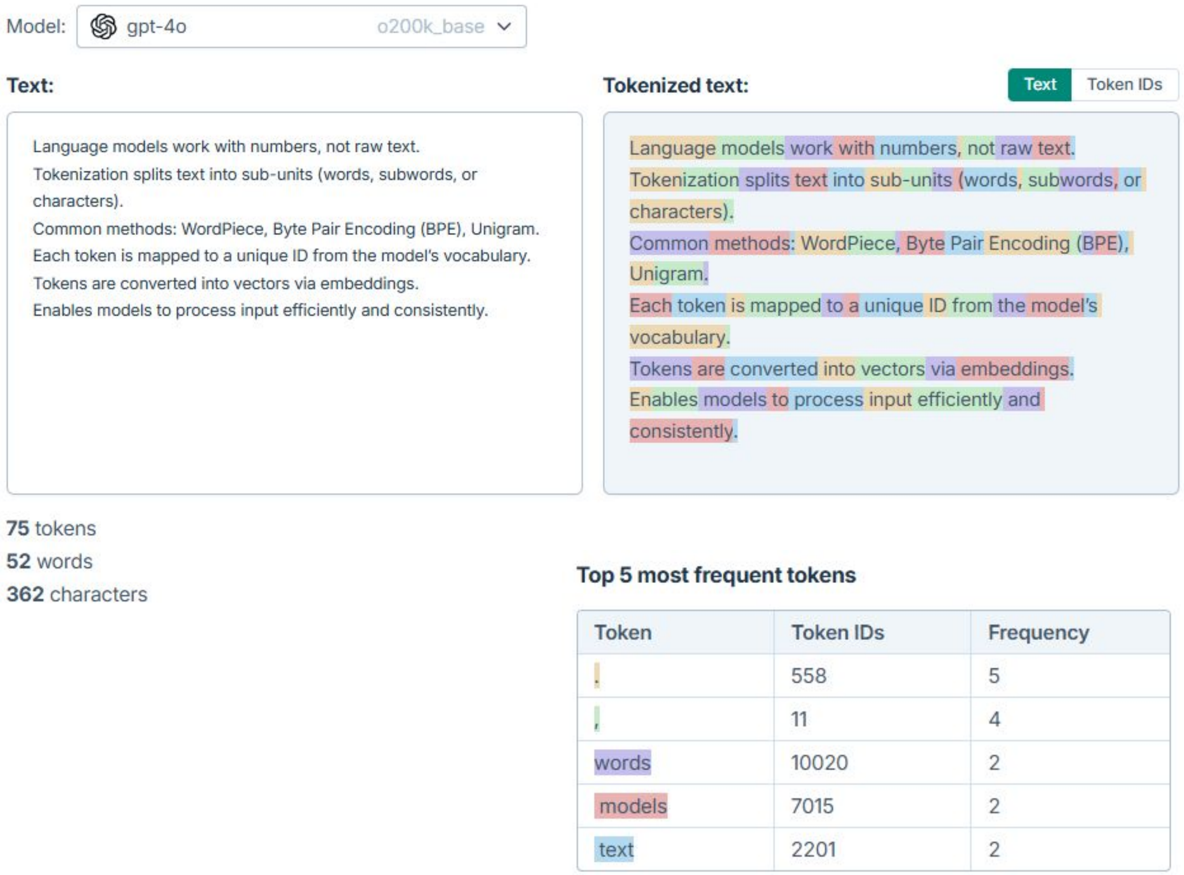
Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com



Tokenization & Text Representation

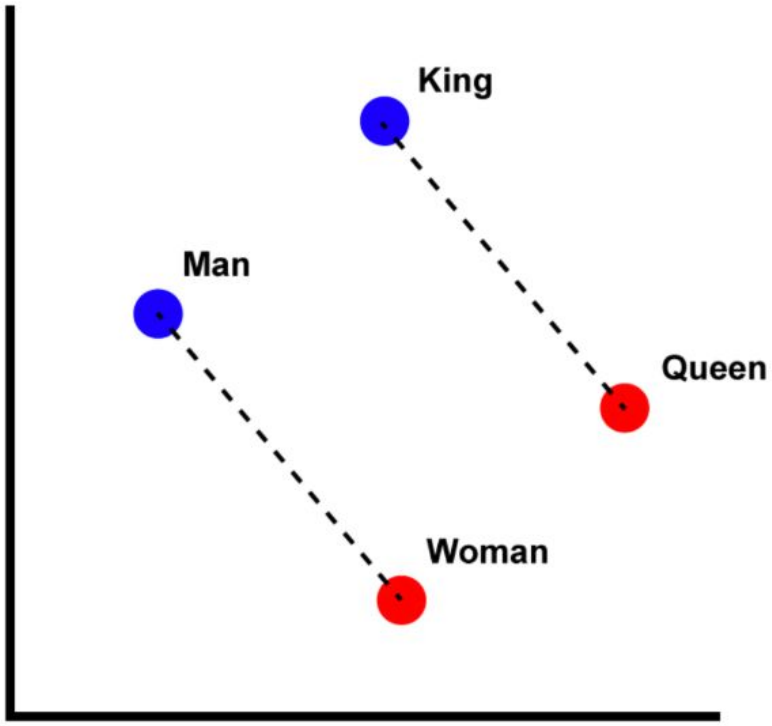
- Language models work with numbers, not raw text.
- Tokenization splits text into sub-units (words, subwords, or characters).
- Common methods: WordPiece, Byte Pair Encoding (BPE), Unigram.
- Each token is mapped to a unique ID from the model's vocabulary.
- Tokens are converted into vectors via embeddings.
- Enables models to process input efficiently and consistently.



[GPT Tokenizer]

Input Embeddings + Positional Encoding

- **Token IDs are mapped to dense vectors** using an embedding matrix learned during training
- Embeddings capture **semantic similarity** (e.g., king and queen are closer than king and car)
- All tokens are embedded into the same vector space, enabling uniform processing
- Positional encodings inject order information, since Transformers lack recurrence
- **Common methods:** sinusoidal encoding (fixed) or learned position vectors
- **Final input** = token embedding + positional encoding, added element-wise.



Self-Attention Mechanism

- Self-attention allows the model to weigh the importance of other tokens in a sequence
- Each token is transformed into **query, key, and value** vectors
- Attention scores are computed as **dot products between queries and keys**
- Softmax is applied to generate a weight distribution over all tokens
- The output is a **weighted sum of value vectors**, highlighting relevant context
- Enables dynamic context understanding and long-range dependencies

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

➔ **Query:** What are the things I am looking for?
➔ **Key:** What are the things that I have?
➔ **Value:** What are the things that I will communicate?

Self-Attention Mechanism (Steps)

Let's say our input has n tokens, and each token is a vector of size d_model (like 512).

Step 1: Create Query, Key, and Value vectors

➔ Each input token vector **x** is multiplied by 3 matrices (learned during training):

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

Query: What are the things I am looking for?

Key: What are the things that I have?

Value: What are the things that I will communicate?

Each of these is of shape:

$$Q, K, V \in \mathbb{R}^{n \times d_k}$$

Self-Attention Mechanism (Steps)

Step 2: Create Query, Key, and Value vectors

➔ We compare each query with every key using the dot product to measure relevance:

$$\text{score}_{ij} = Q_i \cdot K_j^T$$

This gives a matrix of shape $n \times n$, showing how much each word attends to every other word.

Step 3: Scale the Scores

➔ To avoid extremely large values (which can destabilize gradients), we scale the scores:

$$\text{scaled\_scores} = \frac{QK^T}{\sqrt{d_k}}$$

Self-Attention Mechanism (Steps)

Step 4: Apply Softmax

➔ Now we apply softmax to each row — turning raw scores into attention weights (all values between 0 and 1, summing to 1):

$$\text{Attention Weights} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

Step 5: Weighted Sum of Values

➔ Each output vector is the weighted sum of all value vectors (from the input), weighted by attention weights:

$$\text{Output} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

This final output is passed into the next layer of the model.

Self-Attention Mechanism (Simplified Example)

"The cat sat on the mat because it was tired"

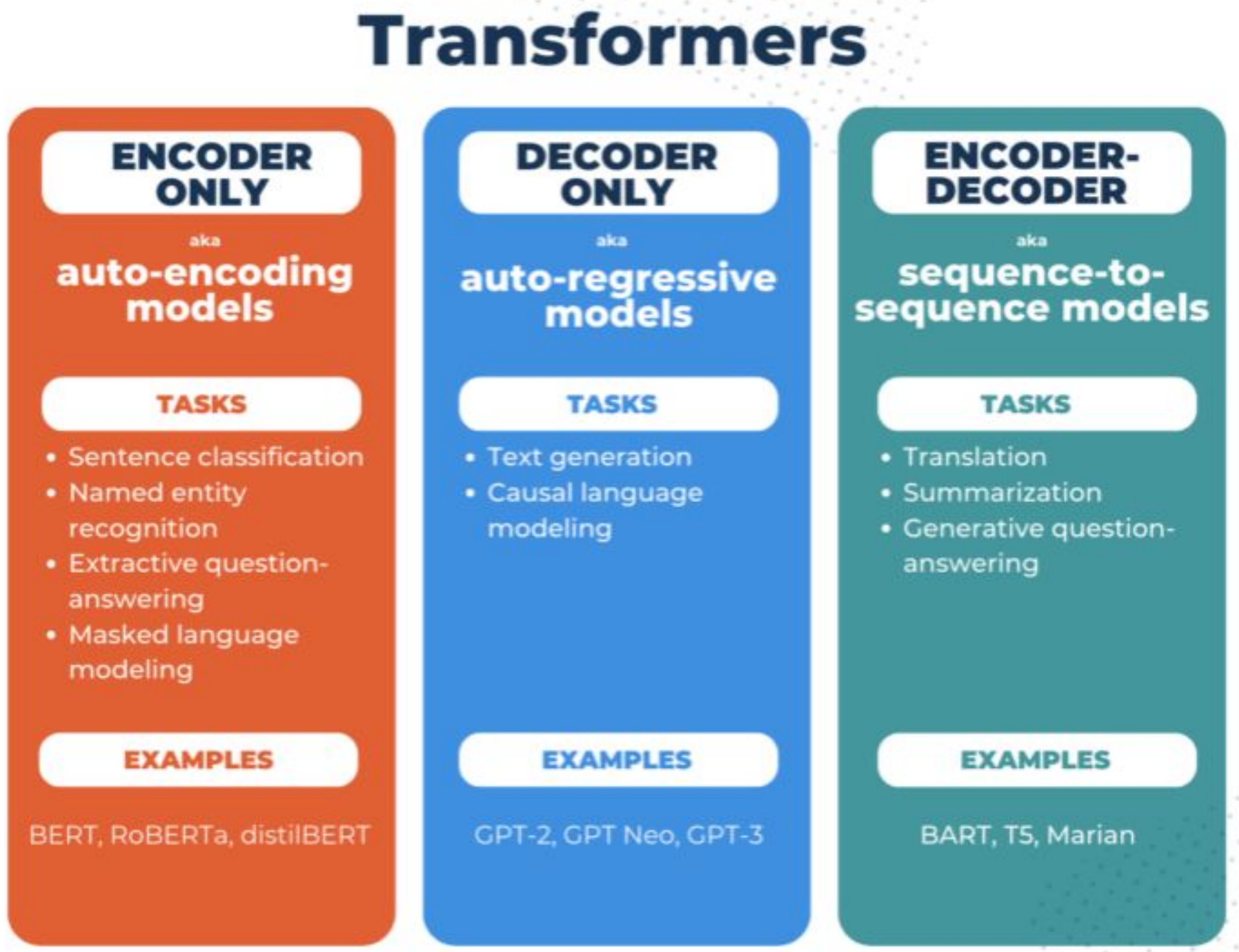
The attention mechanism lets the model focus on the **most relevant words** in a sentence when understanding a specific word.

- For **“it”**, the model attends to or pays attention to **“cat”** more than **“mat”**.
- Every word in a sentence gets a chance to look at all the other words and decide which ones matter most to it.
- Without attention, the model might treat words in isolation.
- With attention, it understands relationships — like who “it” is, or what “because” connects to.

Real-World Applications of LLMs

- **Customer Support:** Chatbots and virtual assistants that answer questions, resolve issues, and automate conversations
- **Content Generation:** Drafting emails, articles, reports, social media posts, marketing copy
- **Legal & Medical Support:** Document summarization, contract review, clinical report generation (with human oversight)
- **Programming Help:** Code completion, generation, explanation, and debugging (e.g., GitHub Copilot)
- **Language Translation:** High-quality translation with context awareness

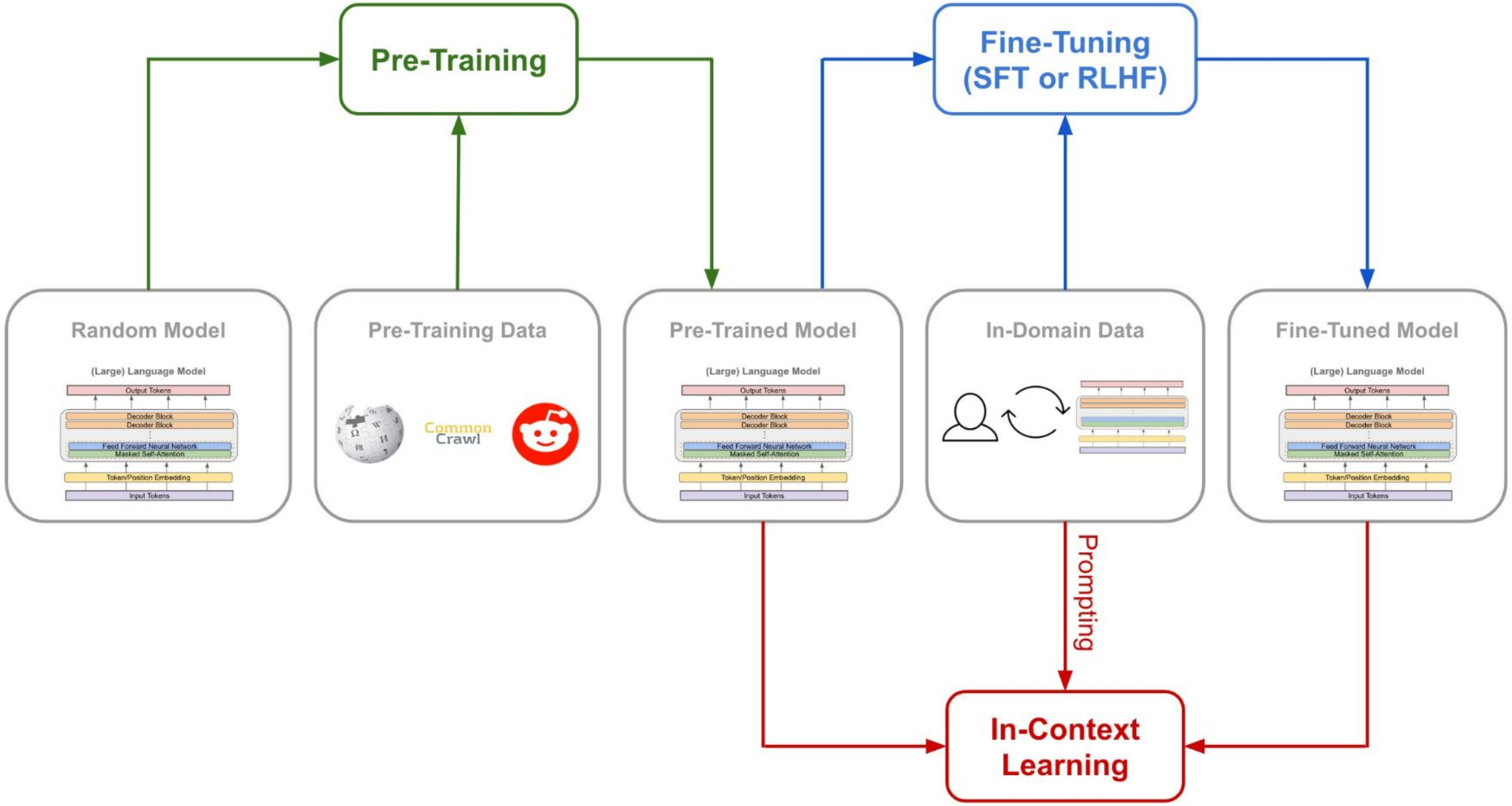
Types of LLMs Architectures



Popular LLMs – Key Models and Their Strengths

Model	Type	Known For	Specific Features
GPT-3 / 4	Decoder-only	Text generation, chat, few-shot learning	Trained on massive web-scale corpora; autoregressive; powers ChatGPT
BERT	Encoder-only	Text understanding, classification, NER	Bidirectional masked language model; strong embeddings
T5	Encoder-decoder	Text-to-text tasks (QA, summarization, etc.)	Everything is framed as a text-to-text problem
LLaMA 2/3	Decoder-only	Open-source GPT-style alternatives	Lightweight and efficient; good performance with fewer parameters
Falcon	Decoder-only	Efficient text generation	High throughput, optimized for inference
Claude	Decoder-only	Chat and reasoning, safer outputs	Reinforcement learning with human feedback focus

How LLMs are Trained: Pre-training and Fine-tuning



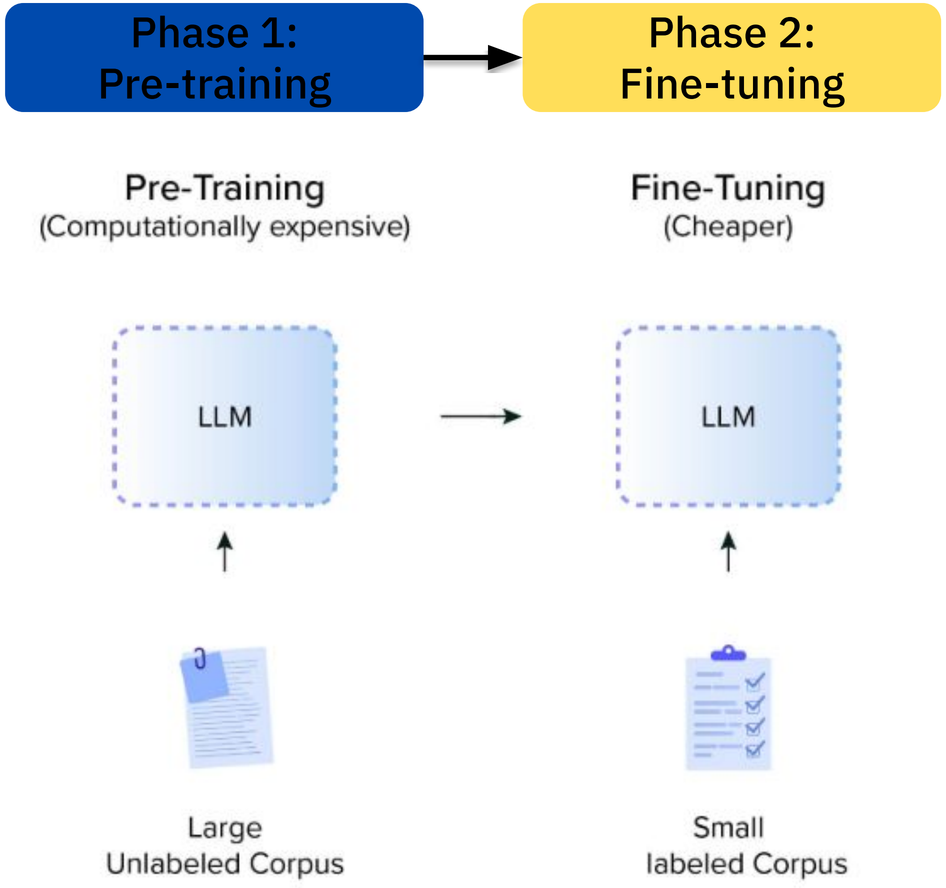
How LLMs are Trained: Pre-training and Fine-tuning

Phase 1

- **Pretraining:** Unsupervised training on massive unlabeled text data (e.g., books, websites, code)
- **Objective:** predict the next token (for decoder models) or fill in missing tokens (for encoder models).
- Creates a general-purpose model with broad language understanding

Phase 2

- **Fine-tuning:** Supervised training on a specific task or domain (e.g., legal QA)
- Improves performance in narrow, targeted applications
- Some models also use instruction tuning and reinforcement learning from human feedback (RLHF) to align with human preferences

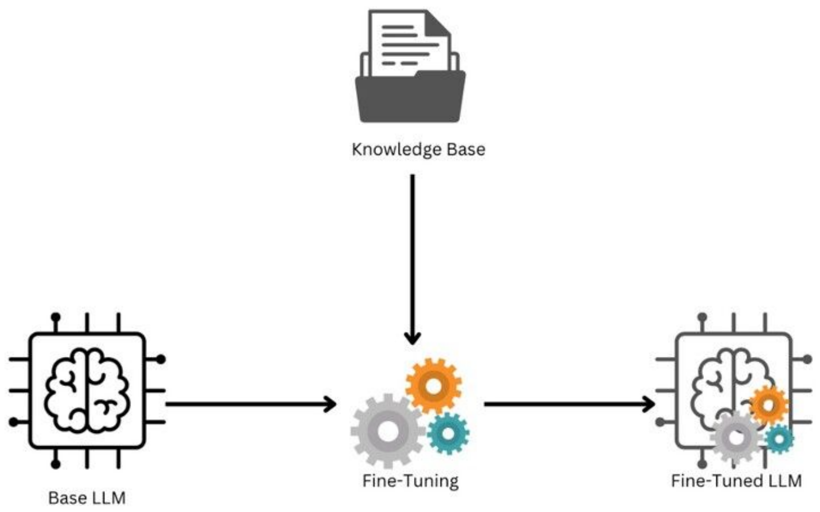


Pre-training vs Fine-tuning – Key Differences

Aspect	Pre-training	Fine-tuning
Goal	Learn general language patterns	Adapt to a specific task or domain
Data	Unlabeled large-scale corpora	Smaller, labeled datasets
Supervision	Self-supervised (e.g., next token)	Supervised (e.g., QA pairs, labels)
Cost	Very high (compute-intensive)	Lower (faster and cheaper)
Flexibility	General-purpose language capabilities	Specialized behavior (e.g., medical QA)
Examples	GPT, BERT (pretrained base models)	GPT-fine-tuned for legal summarization

Why Fine-Tune an LLM?

- Specialize a general model for a specific domain (e.g., legal, medical, finance)
- Improve accuracy on task-specific outputs (e.g., summarization, classification, dialogue)
- Adapt the model to custom tone, structure, or language use
- Add knowledge not covered during pre-training
- Enable on-premise or privacy-preserving model use (e.g., company-owned data)



What You Need to Fine-Tune an LLM ?

- Access to a pre-trained base model (e.g., BERT, GPT-2, LLaMA, Falcon, etc.)
- Labeled dataset specific to your task (e.g., QA pairs, summaries, classifications)
- Matching tokenizer and vocabulary
- Compute resources (GPU or TPU, depending on model size)
- Training loop + framework (e.g., Hugging Face Trainer, LoRA libraries, PEFT libraries)
- Evaluation metrics and a validation set

Important Hyperparameters in Fine-Tuning ?

- **Context window:** Max number of tokens model can process (e.g., 2048, 4096, 8192)
- **Batch size:** How many samples processed at once
- **Learning rate:** How fast the model updates its weights
- **Number of epochs:** How many times the model sees the full dataset
- **Evaluation strategy:** When and how often to validate performance
- **Early stopping & checkpointing:** Avoid overfitting, save progress

How to Fine-Tune an LLM – Step by Step ?

Stage 1
Data Preparation

- Data collection and curation.
- Data cleaning and augmentation.
- Data splitting.

Stage 2
Model Initialisation

- Choose pre-trained model.
- Load pre-trained model weights.

Stage 3
Training Setup

- Configure training environment.
- Define hyper-parameters.
- Initialize optimizers and loss function.

Stage 4
Fine-Tuning

- Standard and advanced fine-tuning.
- Parameter efficient fine-tuning
- Task/domain specific fine-tuning

Stage 5
Validation & Evaluation

- Evaluation metrics.
- Understand loss curve and noisy gradients.
- Hyper-parameters tuning.
- Prevent overfitting/underfitting.

Stage 6
Deployment

- Deployment strategies.
- Optimization for inference.
- Export fine-tuned model.

Stage 7
Monitoring

- Continuous monitoring.
- Periodic retraining and update.

Coding Example – Fine-Tuning GPT model using a custom Dataset

Notebook

What Can You Do with a Fine-Tuned LLM?

- **Classification:** sentiment, intent, spam detection
- **Named Entity Recognition (NER):** extract people, places, orgs
- **Question Answering:** extractive answers from documents
- **Information Retrieval:** embedding-based search and ranking
- **Text Similarity:** semantic comparison for clustering or deduplication
- **Embedding generation:** for downstream ML or vector DB use
- Often deployed as:
  - **APIs**
  - **On-premise inference**
  - **Embedded in pipelines (search, analytics, etc.)**

Evaluating a Fine-Tuned LLM

- Use a validation set to test generalization
- Choose metrics based on task:
  - **Classification** → Accuracy, F1-score, ROC AUC
  - **QA** → Exact Match, F1
  - **Summarization** → ROUGE, BLEU
- Monitor:
  - Loss curves (training vs. validation)
  - Overfitting (great training performance, poor validation)
- Use early stopping to avoid wasted compute
- Perform error analysis to identify failure cases
- Log everything: use tools like W&B, TensorBoard, or MLflow

Risks and Limitations of LLMs

- **Hallucinations:** LLMs can generate fluent but factually incorrect or misleading answers
- **Biases and stereotypes:** Models may reflect and amplify biases in the training data
- **Lack of real understanding:** LLMs don't reason or truly understand language — they pattern-match
- **Context window limits:** Can only consider a limited amount of input (e.g., 2k–32k tokens)
- **Data sensitivity:** May unintentionally memorize or leak personal or confidential data
- **Compute & energy cost:** Training and even inference with large models can be resource-intensive
- **Adversarial prompts:** LLMs can be manipulated to produce harmful, misleading, or toxic content

Mitigating the Limitations of LLMs

- **Fact-checking & retrieval augmentation**

  → Combine LLMs with external knowledge bases (RAG) to reduce hallucinations

- **Bias reduction strategies**

  → Apply data curation, counterfactual examples, and de-biasing prompts

- **Human-in-the-loop (HITL)**

  → Keep a human reviewer in the loop for sensitive or high-stakes use cases

Mitigating the Limitations of LLMs

- **Prompt engineering & safety filters**

  → Design prompts carefully and apply content moderation tools

- **Fine-tuning on domain-specific, curated data**

  → Improves reliability and reduces irrelevant outputs

- **Monitoring & continuous evaluation**

  → Use logging, evaluation benchmarks, and error tracking to monitor real-world behavior

Questions?

ACENET
accelerate discovery

A regional partner of the
Digital Research Alliance of Canada