LLM from scratch - BuildLog

Prerequisites and Requirements

Technical Prerequisites

Developing an LLM requires a strong foundation in several domains. Here are the key areas that must be understood before starting:

- Mathematical Knowledge: A solid grasp of linear algebra, probability, and statistics is essential for understanding neural network operations and optimization algorithms.
- Deep Learning Fundamentals: Familiarity with transformers, self-attention mechanisms, and backpropagation techniques is necessary.
- Programming Skills: Experience with Python is crucial, particularly using deep learning frameworks such as PyTorch and TensorFlow.
- Data Handling Expertise: Knowledge of SQL, big data processing, and dataset preparation methods is required.
- Cloud & Infrastructure Management: Familiarity with setting up and managing local GPU clusters and budget-friendly cloud solutions is beneficial.
- Ethical Considerations: Understanding bias mitigation, regulatory compliance, and responsible AI deployment practices is vital to creating an ethically sound model.

Hardware and Software Requirements

Hardware

The choice of hardware significantly impacts the cost of LLM development. Instead of opting for high-end GPU clusters, cost-effective alternatives include:

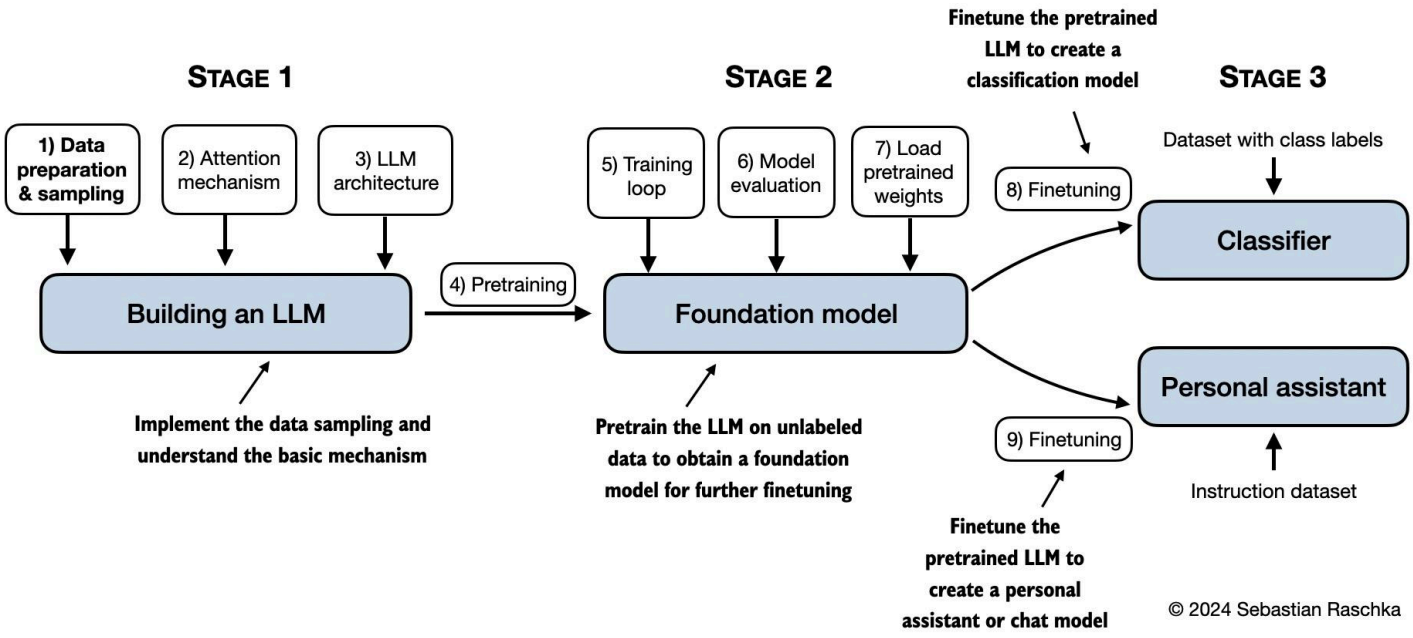
- Compute: Utilizing 2–4 NVIDIA RTX 4090 GPUs or second-hand NVIDIA A100s can provide sufficient computational power at a fraction of the cost of high-end data center GPUs.
- Storage: Using 2–5TB SATA SSDs instead of high-end NVMe storage reduces storage costs while maintaining adequate performance.
- Networking: A Gigabit Ethernet setup for local clusters is sufficient for training models in a distributed fashion.
- RAM: A system with 128–256GB of RAM balances performance and cost when handling large datasets.

Software

To streamline development and optimize costs, the following software tools and frameworks are recommended:

- Machine Learning Frameworks: TensorFlow or PyTorch for model training and experimentation.
- Distributed Training Libraries: DeepSpeed and Fully Sharded Data Parallel (FSDP) to enhance computational efficiency.
- Data Handling Tools: SQLite, Dask, and Apache Arrow for efficient data processing.
- Containerization Tools: Docker and lightweight Kubernetes alternatives for deployment.
- Monitoring & Debugging: Free tools like TensorBoard and the community version of Weights & Biases for model tracking and performance analysis.

Step-by-Step Development Process



Data Collection and Preprocessing

Data Sources

High-quality datasets are crucial for training an LLM. Free datasets such as CC-News, Wikipedia, and OpenWebText provide a rich foundation for language model training. Leveraging publicly available data reduces costs significantly.

Data Cleaning

Before training, data must be preprocessed to ensure consistency and quality. This includes:

- Removing duplicate entries to prevent bias.
- Filtering out low-quality text with heuristic and statistical methods.
- Standardizing formats to maintain uniformity.

Tokenization

Tokenization converts raw text into a format suitable for model training. Using SentencePiece, a lightweight tokenization tool, optimizes memory consumption while preserving token quality.

Dataset Splitting

The dataset should be divided into:

- 80% for training to enable model learning.
- 10% for validation to monitor model performance.
- 10% for testing to evaluate the model after training.

Model Architecture Design

Selecting Model Type

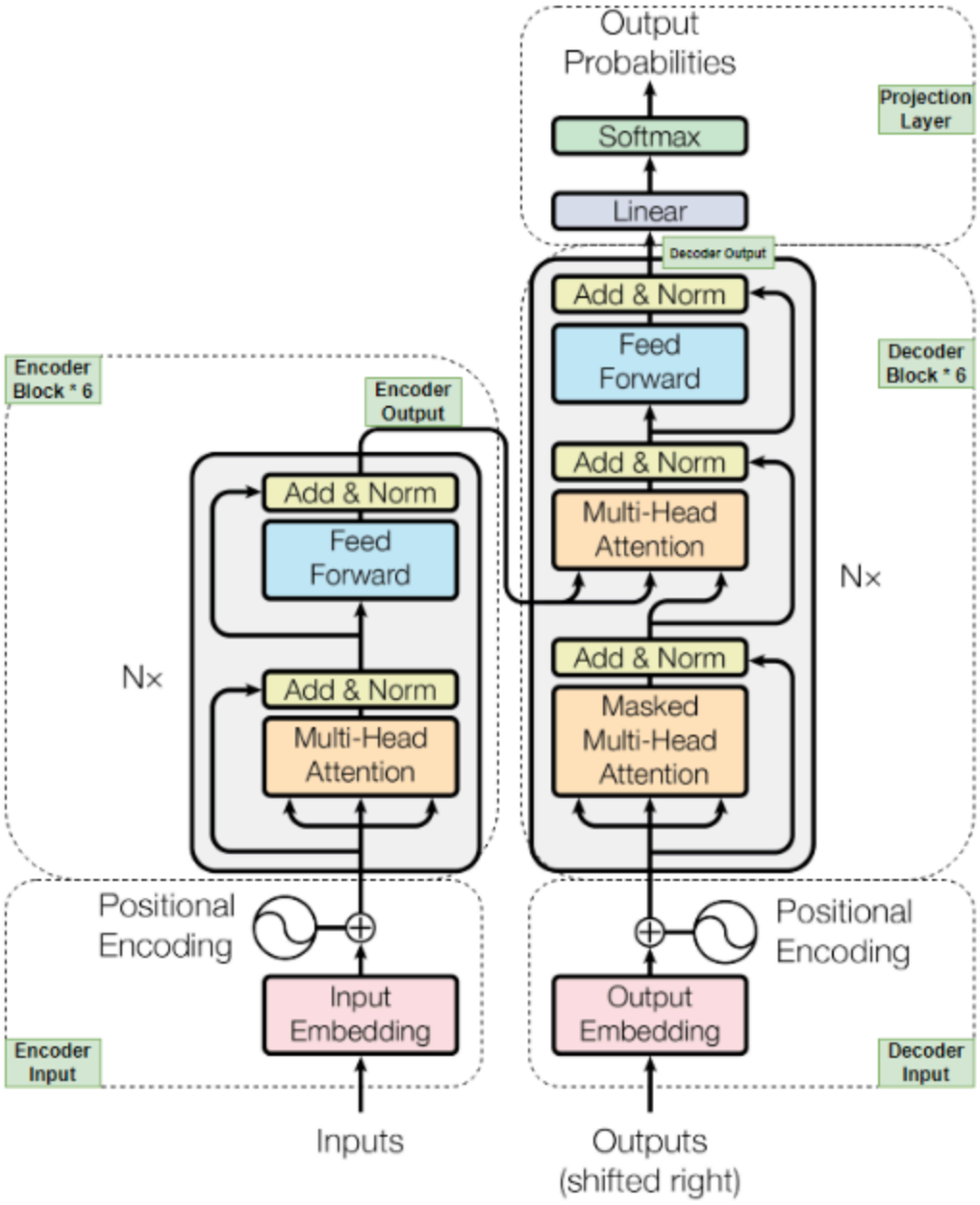
Efficient Transformer variants such as ALBERT and DistilBERT provide cost-effective alternatives to full-scale transformers. These models achieve high performance with fewer parameters, reducing computational requirements.

Layer Design

- Fewer layers (6–12) with wider attention heads ensure efficient attention computation.
- Layer sharing reduces the number of parameters, making the model memory-efficient.
- Quantized computations help reduce memory footprint without significantly affecting performance.

Hyperparameter Selection

- Hidden Size: 512–1024 dimensions provide a balance between expressivity and computational efficiency.
- Attention Heads: 8–16 heads improve contextual understanding without excessive computational burden.
- Context Window: 512 tokens maintain manageable sequence lengths for practical processing.



Training Setup

Parallelism

To efficiently train the model on limited hardware, techniques such as gradient checkpointing and mixed precision training are used.

Optimization Techniques

- The Adam optimizer is used with low-memory adjustments to reduce training cost.

- Dynamic batch sizing allows adaptive resource allocation, optimizing GPU utilization.

Loss Functions

A cross-entropy loss function with label smoothing prevents overfitting while ensuring generalization.

Fine-tuning and Reinforcement Learning from Human Feedback (RLHF)

Supervised Fine-Tuning

Fine-tuning on domain-specific datasets enhances model performance for targeted applications.

Reinforcement Learning

A simplified reward model enables reinforcement learning with minimal compute resources.

Human Feedback Collection

Affordable crowdsourced platforms facilitate efficient data collection for reinforcement learning.

Evaluation and Benchmarking

Performance Metrics

- Perplexity: Measures language model fluency.
- BLEU and ROUGE scores: Assess text generation quality.

Bias and Fairness Testing

Community-driven open-source tools help ensure ethical AI development by detecting bias in model predictions.

Deployment and Scaling

Inference Optimization

- 4-bit quantization, pruning, and distillation reduce inference latency.

Scaling Approaches

- On-premise inference reduces cloud dependency.
- Serverless inference utilizes budget-friendly cloud providers for cost efficiency.

Monitoring & Logging

Free-tier observability tools enable real-time performance monitoring.