# **Understanding and Improving Large Language Models (LLMs) for Coding Tasks**

### **1. Introduction to LLMs**

**Large Language Models (LLMs) are powerful artificial intelligence systems trained to understand and generate human language, including natural language (like English) and even programming languages (like Python or JavaScript). These models are based on deep learning, especially the Transformer architecture, which enables them to understand the relationship between words, sentences, or lines of code.**

**They’ve become extremely popular in recent years and are now widely used for many tasks: writing articles, answering questions, summarizing texts, tutoring students, and generating code. For developers, tools like GitHub Copilot, OpenAI Codex, Google Gemini, and Amazon CodeWhisperer use LLMs to assist with coding, debugging, and documentation. These tools are shaping the future of software development.**

### **2. How LLMs Work**

#### **2.1 Architecture: Transformer**

* **Proposed in the paper “*Attention is All You Need*” by Vaswani et al. in 2017**
* **Replaced older models like RNNs and LSTMs which processed data sequentially**
* **Uses self-attention, allowing the model to read entire sequences in parallel**
* **Built using encoder and decoder blocks:**
  + **BERT uses only the encoder (good for understanding)**
  + **GPT uses only the decoder (good for generation)**
* **Scales well with massive datasets and model sizes (billions of parameters)**

#### **2.2 Key Concepts**

* **Tokenization: Splits text/code into smaller pieces like words, subwords, or characters. Models process these tokens.**
* **Embeddings: Converts tokens into dense vectors in a high-dimensional space. These vectors help the model understand similarities between tokens.**
* **Attention Mechanism: Helps the model focus on important parts of the input sequence when making predictions.**
* **Feedforward Layers: Additional layers that apply mathematical transformations to the token vectors.**
* **Positional Encoding: Since Transformers don't read sequentially, positional encodings give them a sense of order.**

#### **2.3 Training Process**

* **Objective: Predict the next token given a sequence (language modeling)**
* **Loss Function: Cross-entropy loss is commonly used to measure prediction accuracy**
* **Optimizer: Algorithms like Adam or AdamW update the model weights to reduce loss**
* **Training Data: Includes massive datasets from the web — books, articles, forums, code repositories (like GitHub), and more**
* **Epochs: The dataset is read multiple times so the model can learn patterns deeply**

#### **2.4 Simplified Code (Using PyTorch)**

**import torch**

**from transformers import GPT2LMHeadModel, GPT2Tokenizer**

**model = GPT2LMHeadModel.from\_pretrained('gpt2')**

**tokenizer = GPT2Tokenizer.from\_pretrained('gpt2')**

**input\_text = "def quicksort(arr):"**

**inputs = tokenizer(input\_text, return\_tensors='pt')**

**outputs = model(\*\*inputs, labels=inputs['input\_ids'])**

**loss = outputs.loss**

**print("Loss:", loss.item())**

**This short script shows how code can be turned into tokens, passed through a model, and evaluated using a loss function.**

### **3. Mathematics Behind LLMs**

#### **3.1 Attention Mechanism**

**The attention mechanism is a key innovation in Transformers. It allows the model to weigh different parts of the input based on relevance.**

**Formula:  
 Attention(Q,K,V)=softmax(QKTdk)⋅V\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d\_k}}\right) \cdot V  
 Where:**

* **Q = Queries (What are we looking for?)**
* **K = Keys (What do we have?)**
* **V = Values (What information do we pass on?)**
* **dkd\_k = dimension of the key vectors**

#### **3.2 Loss Function**

**Cross-Entropy Loss is used to compare predicted probabilities with the actual correct answers.  
 Loss=−∑iyilog⁡(y^i)\text{Loss} = -\sum\_i y\_i \log(\hat{y}\_i)  
 This helps in adjusting model weights during training to improve accuracy.**

#### **3.3 Optimization**

* **Backpropagation: Calculates gradients to show how much each weight affected the loss**
* **Gradient Descent: Updates the weights to reduce loss**
* **Adam/AdamW: Popular optimizers that adaptively adjust learning rates for each parameter**

### **4. Limitations in Large Coding Tasks**

#### **4.1 Limited Context Window**

* **Most LLMs can only process a limited number of tokens at once (e.g., 2,048–8,192 tokens)**
* **If a codebase exceeds this size, important context may be lost, leading to incorrect outputs**

#### **4.2 Shallow Code Understanding**

* **LLMs treat code like text, not like a logical program**
* **They can miss the connection between functions, files, or variables spread out over a project**

#### **4.3 Poor Error Recovery and Debugging**

* **LLMs may generate code with bugs but won’t always detect or fix them**
* **They lack an internal model of program state or execution**

#### **4.4 Hallucination and Overconfident Outputs**

* **LLMs sometimes produce code that looks good but is logically flawed**
* **They may suggest non-existent functions or APIs with great confidence**

### **5. Improving LLMs for Coding**

#### **5.1 Data-Centric Improvements**

* **Use more real-world, diverse, and clean code samples**
* **Include labeled datasets with documentation, bug-fix commits, and unit tests**
* **Add edge cases and bad code examples for contrastive learning**

#### **5.2 Model-Centric Improvements**

* **Larger Context Windows: Like Gemini 1.5 (1M tokens) help understand entire codebases**
* **Instruction Fine-Tuning: Training with prompts and expected responses improves task-following**
* **RLHF: Uses human feedback to fine-tune models for better quality responses**

#### **5.3 Tool-Augmented Models**

* **Equip LLMs with access to external tools:**
  + **Code linters for syntax/style checking**
  + **Static analyzers for bug detection**
  + **Git search engines for finding related examples**
* **Examples: GitHub Copilot X, Cursor IDE, OpenAI Code Interpreter**

### **6. Making LLMs Write Better Code**

#### **6.1 Prompt Engineering**

* **Specify what the input is, what output you want, and any constraints**
* **Include test cases or examples in your prompt to guide the model**

#### **6.2 Iterative Refinement**

* **Generate code → test → debug → regenerate**
* **Repeat until the output is correct and optimized**

#### **6.3 Few-shot / Chain-of-Thought Prompting**

* **Include examples or breakdowns of logic in the prompt**
* **Helps the model mimic step-by-step reasoning, especially for complex logic**

#### **6.4 Use Structured Outputs**

* **Ask the model to output in structured formats like JSON, YAML, XML**
* **Makes parsing and further processing easier in applications**

### **7. Recommended Algorithms to Improve LLMs for Coding**

#### **7.1 Code-Specific Learning Techniques**

* **Contrastive Learning: Helps distinguish good vs bad code outputs**
* **Retrieval-Augmented Generation (RAG): Brings in context from external sources in real-time**
* **Tree/Graph-Based Program Synthesis: Uses ASTs or CFGs to reason over code structure**
* **Neuro-Symbolic Models: Combine neural networks with logic-based reasoning (rules, constraints)**

#### **7.2 Algorithms to Fit Large Codebases into the Context Window**

* **Semantic Chunking: Split code into logical blocks (e.g., function/class level)**
* **Sliding Window Attention: Process overlapping segments to maintain broader context**
* **Summarization and Abstraction: Replace detailed blocks with summarized versions**
* **Vector Databases (e.g., FAISS, Pinecone): Search and fetch similar code embeddings**
* **Hierarchical Planning: Understand high-level code structure first, then drill into details**
* **Auto-Indexing: Create searchable indexes of classes, functions, and modules**
* **Hybrid Models: Combine traditional program analysis with LLM prompts for better efficiency**

### **8. Conclusion**

**LLMs have brought big improvements to coding productivity, making tasks like code completion, bug fixing, and documentation easier. But there’s still room for improvement, especially in handling large codebases and generating error-free logic.**

**By improving datasets, scaling model capabilities, using external tools, and writing better prompts, we can push the boundaries of what AI can do in software development.**

**As college students and aspiring developers, understanding these models helps us not just use them effectively—but also contribute to their future.**

### **9. References**

* **Vaswani et al. (2017). *Attention Is All You Need***
* **OpenAI GPT Research Papers**
* **GitHub Copilot & Cursor IDE Documentation**
* **Jay Alammar’s Illustrated Transformer Blogs**

**Prepared by: Debanik Debnath  
 Date:** 13/06/25