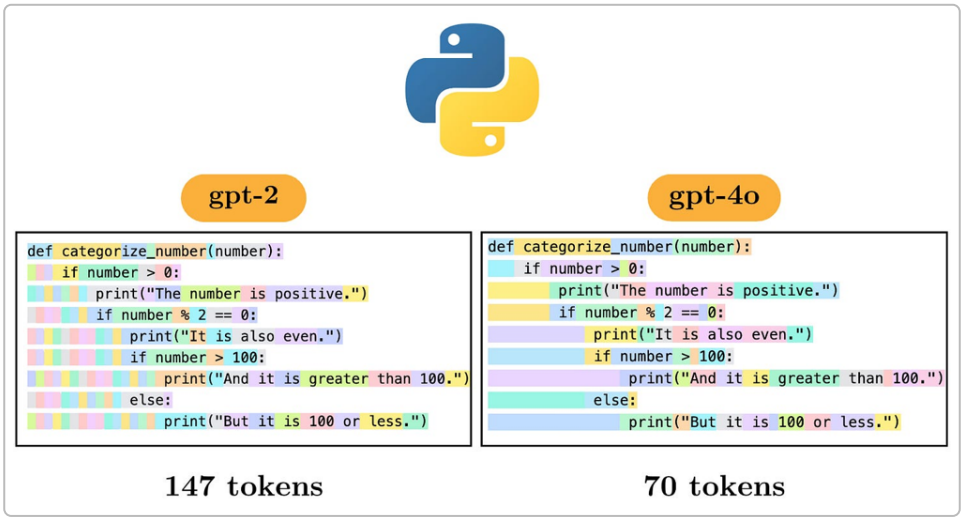
ChatGPT

코드 특화 LLM vs 일반 LLM: 토크나이저와 문맥 모델링의 차이

코드 특화 토크나이저의 특징과 차별점

코드 전용 토크나이저는 일반 자연어 토크나이저와 달리 프로그래밍 언어의 **문법적 경계**를 더 잘 반영하도록 설계됩니다. 일반 LLM은 주로 BPE(Byte Pair Encoding) 같은 **서브워드 기반 토크나이저**를 사용하여 통계적으로 자주 등장하는 문자열을 한 토큰으로 합치는데, 이 방식은 코드의 어휘 경계(키워드, 연산자, 식별자 등)와 **맞지 않는 분할**을 초래할 수 있습니다 [1]. 예를 들어, **공백 하나**의 유무만으로도 토큰 시퀀스가 크게 달라질 수 있습니다. 실제 연구에서는 `“.factorial”`과 `“. factorial”`처럼 **구문상 동일한 코드**에서도, 마침표 뒤의 공백 유무에 따라 토큰화 결과가 `[".factor", "ial"]` vs `[".", "␣factorial"]`로 완전히 달라졌고, 모델의 출력도 달라졌다고 보고했습니다 [2]. 이러한 현상은 코드 LLM이 **문법 토큰 단위**로 코드를 인식하지 못하고 서브워드 통계에 따라 분절되기 때문에 발생하며, 작은 포맷 차이가 모델 예측을 요동시키는 **불안정성**의 원인이 됩니다 [1] [2]. 따라서 코드 특화 모델들은 일반적으로 **문법 인지적 토크나이저** 개발의 필요성이 제기되고 있습니다. 실제로 Qwen-2.5와 같은 모델의 코드 버전은 **동일한 토크나이저와 어휘집**을 공유하여 이러한 불일치를 줄이는 시도를 합니다 [3].

또 한 가지 중요한 차이는 **공백과 들여쓰기 처리**입니다. 일반적인 GPT-2계열 토크나이저는 파이썬과 같은 언어의 들여쓰기 공백 한 칸 한 칸을 **각각 개별 토큰**으로 취급했는데, 이는 코드에서 엄청난 **토큰 낭비**를 낳았습니다 [4]. GPT-2의 경우 파이썬 코드 한 줄의 4칸 들여쓰기가 4개의 토큰으로 분할되었고, 결과적으로 작은 코드 조각도 수백 개의 토큰이 되곤 했습니다. 반면 **최신 GPT-4** 계열에서는 이러한 **반복 공백 시퀀스를 하나의 토큰으로 묶는** 등 개선된 토크나이저를 도입했습니다 [5]. 아래 그림은 동일한 파이썬 함수를 GPT-2와 GPT-4의 토크나이저로 인코딩한 예시입니다. 좌측 GPT-2 토크나이저는 각 들여쓰기 공백을 별도 토큰으로 처리해 총 147개의 토큰을 생성한 반면, 우측 GPT-4 토크나이저(cl100k)는 공백들을 뭉쳐 처리함으로써 70여 개 토큰만으로 동일한 코드를 표현하고 있습니다 [6] [5]. 이처럼 **들여쓰기 등의 구조적 공백을 효율적으로 압축**함으로써, 모델이 실제 **코드 의미**를 담은 토큰들에 더 많은 **맥락 용량**을 할애할 수 있게 되었습니다 [5] [7].



GPT-2 vs GPT-4 토크나이저의 파이썬 코드 토큰화 비교. 좌측 GPT-2는 들여쓰기 공백 each를 개별 토큰으로 처리(많은 색 블록), 우측 GPT-4는 공백들을 묶어 토큰화(더 적은 블록)하여 전체 토큰 수를 절반 이하로 줄였다 [5]. 공백 토큰을 줄이면 모델이 코드의 실제 논리 토큰에 더 집중할 수 있어 성능 향상에 기여한다 [7].

또한 **코드 특화 모델**들은 **특수 토큰**을 활용하기도 합니다. 예를 들어 Meta의 Code Llama는 **코드 인필(infill)** 기능을 위해 `<PRE>`, `<MID>`, `<SUF>` 같은 특별 토큰을 도입했습니다 [8]. 이는 프리픽스/서픽스/중간 부분을 표시하여 코드의 중간에 새 코드를 삽입하는 **Fill-in-the-Middle** 과제를 지원하는 용도입니다. 일반 LLM에서는 이런 토큰이 없지만, **코드 LLM**은 코드 완성기의 요구에 맞춰 중간에 코드를 끼워넣도록 **토크나이저 수준에서 학습**되어 있습니다 [8]. 이처럼 코드 특화 토크나이저는 **키워드와 식별자, 연산자, 공백** 등을 세심하게 다루고, 필요에 따라 **추가 토큰**으로 코딩 작업에 최적화된 처리를 하는 특징이 있습니다.

코드 문맥 모델링: n-그램 한계 극복과 긴 컨텍스트 처리

프로그래밍 언어 코드는 일반 문장보다 **긴 범위의 문맥과 복잡한 구조**를 갖습니다. 과거에는 코드 확률 모델링에 **n-그램 모델**을 활용하여, 이전 n개의 토큰을 기반으로 다음 토큰을 예측하곤 했습니다. 실제로 Hindle 등(2012)은 소프트웨어가 인간이 만든 **자연스러운 언어**와 유사하여 **중복성과 예측 가능성**이 높음을 보이고, 작은 n-그램 모델로도 코드 자동완성에 효과가 있음을 보였습니다 [9]. 그러나 n-그램은 고정된 윈도우 크기 n 내의 정보만 활용하므로, **멀리 떨어진 문맥**(예: 수십 줄 위에서 선언한 변수의 사용, 여는 괄호와 닫는 괄호의 매칭 등)을 반영하지 못하는 한계가 있습니다. 반면 **Transformer 기반 LLM**은 **자기주의(attention)** 메커니즘을 통해 **시퀀스 전체의 토큰들에 동시**에 주의를 기울일 수 있으므로, 코드의 **장거리 의존성**도 포착할 수 있습니다. 예를 들어, 한 파일의 처음에 정의된 함수 시그니처를 모델이 벡터 형태로 기억해두고, 수백 라인 아래에서 그 함수를 호출할 때 관련 정보를 불러와 일관된 호출 방법을 생성할 수 있습니다. 이는 LLM의 **벡터화된 문맥 표현** 덕분으로, 단순 문자열 일치가 아닌 **의미 공간상의 유사성**으로도 패턴을 잡아내기 때문입니다. 결과적으로 코드 LLM은 n-그램 빈도에만 의존하지 않고, **자연어보다 엄격한 코드 문법과 패턴을 내부화**하여 더 정확한 다음 토큰 예측을 수행합니다.

Transformer의 자기주의 기제는 특히 코드에서 유용한데, 서로 **짝을 이루는 토큰**들을 직접 연결하기 쉽기 때문입니다. 예를 들어 여는 `{` 토큰과 이에 대응하는 닫는 `}` 토큰은 수십 줄 떨어져 있어도, 적절한 포지셔널 인코딩과 자기주의를 통해 모델 내부에서 **상호 연관**될 수 있습니다. 마찬가지로 `if (...)` 문의 조건과 그에 따른 `else` 블록, 들여쓰기 수준 등 **구조적 관계**도 모델이 **주의 헤드**를 통해 학습합니다. 이는 기존 n-그램 모델이 이웃한 토큰 패턴만 보던 것에 비해 **구조적 패턴**을 포착할 수 있게 해줍니다. 예컨대, 한 함수 내의 코드 블록이 끝나면 들여쓰기가 감소한다는 규칙이나, 파이썬에서 `:` 콜론 다음에는 새 블록이 시작되어 들여쓰기가 증가한다는 패턴 등을 LLM은 수많은 코드를 보면서 **암묵적으로 내재화**하게 됩니다. 이러한 구조 학습 덕분에, 최신 코드 LLM들은 **구문 오류**를 거의 내지 않고 여는 태그에 대응하는 닫는 태그를 정확히 출력하는 등 **문법적으로 완결된 코드**를 생성하는 능력이 크게 향상되었습니다.

한편, LLM이 문맥을 **벡터로 압축**하여 이해한다는 것은, 코드의 특정 패턴이 정확히 일치하지 않아도 **유사한 맥락**이면 올바른 출력을 낼 수 있음을 의미합니다. 예를 들어, 이전에 봤던 코드 사례에서 `for` 루프를 리스트 내포(list comprehension)로 변환하는 패턴을 학습했다면, 변수 이름 등이 달라도 모델은 벡터화된 지식을 활용해 **유사한 변환을 적용**할 수 있습니다. 이는 마치 n-그램 모형이 과거에 본 토큰 시퀀스를 참조하는 것과 비슷하지만, 꼭 같은 문자열이 아니어도 **추상적인 패턴 유사성**으로 일반화된다는 점에서 다릅니다. 따라서 코드 LLM은 훈련 단계에서 축적한 **다양한 코드 예제의 변환 지식**을 활용하여, 현재 입력 코드에 맞는 리팩토링이나 코드 변환 출력을 생성합니다. 이러한 **패턴 매칭과 응용** 능력은 Transformer의 큰 장점으로, 코드 리팩토링/변환 시 **이전 사례와 현재 문제를 연결짓는** 역할을 합니다.

긴 문맥 및 위치 인코딩의 개선

일반 LLM과 비교할 때, **코드 특화 LLM은 긴 맥락을 다루는 능력**이 특히 중요합니다. 코드 파일은 수천 줄에 달할 수 있고, 여러 파일에 걸친 **프로젝트 전체 문맥**을 한 번에 처리해야 할 수도 있기 때문입니다. 이를 위해 **맥락 윈도우 크기**를 확장하고 **위치 인코딩** 방식을 조정하는 노력이 이루어집니다. 예를 들어, Meta의 **Code Llama**는 기본 Llama2 모델(최대 4K 토큰)을 코드 데이터로 추가 학습하면서 **컨텍스트 길이를 16K까지 늘렸고**, 일부 모델은 **최대 100K 토큰까지 안정적인 생성**이 가능함을 보였습니다 [10]. 이는 RoPE(Rotary Position Embedding) 등의 **포지셔널 인코딩 기법**을 활용하여 학습 시보다 훨씬 긴 위치도 일반화하도록 미세조정한 결과입니다 [11]. 즉, 학습 시 16k 길이까지 보았지만, **위치 좌표를 스케일 조정(Interpolation)**하는 방법으로 모델이 100k 위치에서도 큰 성능 저하 없이 작동하게 한 것입니다 [11]. 이러한 확장된 문맥 창 덕분에, Code Llama는 **긴 코드**나 **여러 파일에 걸친 코드**도 한 번에 맥락으로 처리하여 일관성 있는 결과를 낼 수 있습니다.

Anthropic의 **Claude 2** 모델도 현재 **100K 토큰**이라는 매우 긴 컨텍스트를 지원하여 주목받습니다. 이는 **75,000단어 이상**에 해당하는 분량으로, 수백 페이지에 달하는 코드와 문서를 한 번에 입력해도 이해할 수 있는 수준입니다 [12]. 실제로 Anthropic은 한 번에 **전체 코드베이스**를 프롬프트에 넣고 그 위에서 수정이나 신규 코드를 작성하는 시나리오를 시연하였는데, 벡터 DB로 나눠 검색하는 방식보다 이런 **직접 대용량 컨텍스트** 입력이 복잡한 질문에 더 효과적일 수 있다고 밝혔습니다 [13]. 이처럼 **맥락창 증가**는 코드 LLM의 활용도를 크게 넓혀주었고, 사람으로 치면 **프로젝트 전체를 머리에 넣고 코딩하는** 수준의 맥락 활용이 가능해졌습니다. 한편, 구글의 차세대 모델인 **Gemini 3**는 **무려 100만 토큰**에 달하는 맥락창을 제공한다고 알려져 있습니다 [14]. 이는 사실상 **제한 없는 컨텍스트**를 지향하는 것으로, 복잡한 멀티모달 입력과 방대한 코드베이스도 모두 맥락으로 처리해주겠다는 목표로 보입니다 (아마도 내부적으로는 필요한 정보만 추출하는 **Retrieval** 기술과 결합되었을 가능성이 높습니다). 어쨌든 이런 발전을 통해 코드 LLM은 **n-그램 기반 모델이 상상도 못할 긴 코드 문맥**을 단번에 고려하여, 코드 전체의 구조를 파악하고 일관성 있는 출력을 생성할 수 있습니다.

코드 구조 이해와 식별자 처리 방식

일반 LLM과 달리, **코드 특화 LLM은 구조적인 정보**를 보다 중시합니다. **들여쓰기, 중괄호/괄호, 세미콜론, 들여쓰기 수준** 등은 모두 코드의 블록 구조를 나타내는 신호입니다. LLM은 명시적으로 AST(추상 구문 트리)를 사용하지는 않지만, 대량의 코드 학습을 통해 **이러한 구조적 패턴을 암묵적으로 학습**합니다. 예를 들어, 파이썬의 들여쓰기는 AST에서 블록 노드 계층으로 나타나지만, LLM은 텍스트 상의 뉴라인과 공백 패턴으로부터 **블록의 시작과 끝**을 추론하게 됩니다. Transformer의 **포지션 인코딩**과 자기주의 메커니즘은 **상대적 거리**나 **패턴 반복**도 인식할 수 있어서, 동일한 들여쓰기 수준의 라인들끼리는 유사한 맥락으로 처리하고, 들여쓰기 변화가 생기면 새로운 범위로 판단하는 등의 **규칙성을 내재화**합니다. 즉, 들여쓰기와 블록 구조를 **일종의 위치 정보**로 활용하고 있는 셈입니다. 일부 연구에서는 여기에 더해 AST를 트리 순회하여 시퀀스로 펼쳐 넣거나, AST 노드 타입 정보를 추가 피처로 주는 시도도 있지만, 주류 LLM들은 **별도 구조 입력 없이도** 충분한 파라미터 규모와 학습데이터로 이러한 **코드 문법 규칙**을 습득하는 것으로 나타납니다.

식별자(변수명, 함수명 등) 처리도 코드 LLM의 중요한 과제입니다. 자연어에서는 단어를 유의어로 바꾸어도 문장 의미가 유지되는 경우가 많지만, 코드에서 식별자 이름을 임의로 바꾸면 참조가 달라지거나 의미가 통째로 달라질 수 있습니다. 따라서 코드 LLM은 **동일한 철자의 식별자**는 문맥에서 **동일한 개체**로 엄격히 간주하고 일관되게 사용하려는 경향을 보입니다. 예를 들어 함수 정의의 이름과 호출문의 이름이 같아야 한다는 것을 학습을 통해 알고 있기 때문에, 프롬프트에 나온 변수명을 **최대한 그대로 재사용**하려 합니다. 이는 Transformer의 자기주의가 **동일 토큰 시퀀스끼리 강한 연관**을 짓는 속성으로 뒷받침됩니다. 다만, 이러한 접근은 철자가 한 글자만 달라도 전혀 다른 토큰으로 인식하기 때문에, 사람이 볼 때는 같은 개념의 변수여도 이름이 다르면 모델이 연결짓지 못할 수 있습니다. 실제 한 연구에서는 **코드 LLM이 "구문 토큰"과 "식별자 토큰" 간의 관계를 잘 인코딩하지 못한다**는 흥미로운 결과를 보고했습니다 [15]. 즉, 모델은 키워드나 중괄호 등의 **구문 요소들 간의 상호관계**(예: `if`와 `else`의 쌍 등)나, 변수들 **식별자끼리의 관계**(예: 같은 이름 변수가 할당되고 참조되는 부분) 등은 잘 학습하지만, 정작 **구문과 식별자 사이**의 관계(예: 어떤 제어구조 내에서 어떤 변수가 쓰인다든지, 변수의 스코프 범위 등)는 덜 명시적으로 인코딩한다는 것입니다 [15]. 이는 모델이 코드의 **전체적인 의미**보다는 **패턴**에 많이 의존함을 시사하며, 변수 스코프를 정확히 인지하지 못해 **가끔 전역에 없는 함수를 호출하거나 잘못된 변수명을 제시**하는 실수로 이어지기도 합니다 [16] [17]. 이러한 한계를 보완하기 위해, 일부 최신 접근은 아예 **컴파일러 피드백**을 활용해 틀린 출력이 나오면 수정하도록 RLHF를 하는 등 시도도 있습니다. 또한 변수를 추상화하여 **<VAR1>** 같은 토큰으로 표현하고, 변환 시 맥락에 맞게 다시 치환하는 기법이나, **코드 조회/재사용(RAG)**으로 훈련 데이터에서 유사한 코드 조각을 참조하는 방법 등이 연구되고 있습니다. 그러나 현재 대표적인 코드 LLM들은 주로 **순수한 언어 모델 방식**으로 작동하며, 식별자도 **그저 시퀀스의 일부**로 다루되 대규모 데이터 학습을 통해 **일관성 유지 패턴**을 익힌 형태라고 볼 수 있습니다.

코드 특화 사전학습 및 미세조정 방식

사전학습 단계에서 코드 특화 LLM은 방대한 **코드 코퍼스**를 사용하여 일반 LLM과 차별화됩니다. 예를 들어, OpenAI의 Codex(및 GPT-4 코드 능력)는 GitHub의 공개 저장소 등에서 수백억 토큰 이상의 코드를 학습했고, Meta Code Llama는 **500B 토큰** 규모의 코드로 Llama2를 추가 훈련하여 얻어졌습니다 [18]. 이러한 대량의 코드 학습을 통해 모델은 **다양한 언어의 문법**, 표준 라이브러리 사용법, 디자인 패턴 등을 폭넓게 습득하게 됩니다. 또한 사전학습 중에 **주석과 코드**(NL-PL 페어) 데이터를 함께 학습하여, 코드에 대한 요약이나 설명도 어느 정도 이해하도록 합니다. 이는 코드 컨텍스트에서 자연어 지시나 주석을 만났을 때 더 잘 처리할 수 있게 하는 밑바탕이 됩니다.

미세조정 단계에서는 보다 **특정한 작업**에 맞게 모델을 최적화합니다. 코드 LLM의 대표적 미세조정으로 **명령-팔로우 (Instruct) 튜닝**이 있는데, 사용자의 명령에 따라 코드를 생성하거나 수정하는 능력을 기르는 것입니다. 예컨대 "주어진 코드 버그를 찾아 고쳐줘" 같은 프롬프트에 적절히 응답하도록, HumanEval 문제 풀이나 GitHub 이슈-패치 쌍 등의 데이터로 **지도학습**을 하거나, **RLHF**를 통해 사용자 선호에 맞는 출력을 내도록 조정합니다. 이 과정에서 모델은 단순 다음 토큰 예측을 넘어서 **맥락에서 요구되는 작업 의도**를 파악하고 수행하는 능력이 향상됩니다. 특히 RLHF를 할 때, 실행 결과가 제대로 돌아가는 코드에 높은 보상을 주고 오류가 난 코드에는 페널티를 주는 식으로 하면 모델이 **구문 및 의미적으로 정확한 코드**를 선호하게끔 유도할 수 있습니다. OpenAI의 GPT-4가 코드 관련 질문에 뛰어난 성능을 보이는 것은 거대한 파라미터 수와 코드 사전학습 덕분일 뿐 아니라, 이런 **Human Feedback**을 통한 정확성 향상 과정이 있었기 때문으로 추정됩니다.

또 다른 미세조정으로 **Fill-in-the-Middle**(코드 중간 삽입) 학습을 들 수 있습니다. InCoder, Code Llama 등은 완성하고자 하는 부분을 `<FILL_ME>` 등으로 비워두고 양 옆의 문맥으로 가운데를 채우는 훈련을 별도로 시행했습니다 [19]. 이를 통해 IDE에서 코드 중간을 자동 완성하거나, 함수 본체만 채워주는 등의 기능에 특화된 모델을 만들 수 있습니다. 이처럼 **작업 맞춤 학습**을 통해 모델은 **문맥을 해석하고 필요한 부분만 생성**하는 능력을 습득하게 됩니다.

요약하면, 코드 특화 LLM은 **데이터 면에서 코드 비중을 높이고**, **토큰화부터 학습목표까지 코드 친화적으로** 설계됩니다. 그리고 후처리로 **Temperature 조절**이나 **Stop 시퀀스** 설정 등을 통해 코드 블록만 출력하도록 한다든지, 프롬프트에 자동으로 ``` 같은 **코드 표시**를 넣어준다든지 하는 세부 튜닝도 이뤄집니다. 이러한 전방위적인 최적화 덕분에, 코드 LLM들은 일반 LLM 대비 **코드 작성 및 이해**에서 월등한 성능을 보여줍니다.

주요 모델의 코드 문맥 처리 비교 (GPT-4, Claude, Code Llama, Gemini)

마지막으로, 대표적인 **대규모 모델들의 코드 처리 특성**을 비교해보겠습니다:

- **GPT-4 (OpenAI)**: GPT-4는 범용 모델이지만, 방대한 코드 데이터로 학습되고 인간 피드백으로 튜닝되어 **탁월한 코드 이해/생성 능력**을 보입니다. 토크나이저는 100k 어휘의 `cl100k_base`를 사용하여, 다중 공백을 하나로 묶는 등 **코드 친화적 토큰화**를 지원합니다 [5]. 최대 **32K 토큰**까지 컨텍스트를 지원하는 버전이 있어 긴 코드도 처리 가능하며, 복잡한 알고리즘 문제도 **논리적으로 추론**해 풀어내는 능력이 뛰어납니다. HumanEval 등 코딩 벤치마크에서 GPT-4는 **80% 이상**의 높은 문제 정답률을 기록하여 거의 **사람 수준의 코딩 능력**을 입증했습니다. 다만 비공개 모델이라 내부 구현은 추측에 의존하지만, **체인-of-Thought**를 활용한 추론과 엄격한 RLHF로 **코드 정확성**을 담보하는 것으로 알려져 있습니다.

- **Claude 2 (Anthropic)**: Claude는 **대화형**에 초점을 둔 모델로서, 코드 이해/생성 능력도 GPT-4에 버금갑니다. 특히 **100K 토큰**에 달하는 거대한 문맥 창을 제공하는 것이 특징으로, 한 번에 **프로젝트 전체를 입력**하고 분석하거나 수정 지시를 내릴 수 있습니다 [12] [13]. 이런 긴 문맥 처리 덕분에, 여러 파일에 걸친 버그 찾기나 대용량 코드 리팩토링 작업에 강점을 보입니다. 토크나이저 측면에서는 OpenAI와 유사하게 다중 공백/들여쓰기를 효율화한 것으로 보이며, 프롬프트 길이에 비례해 **응답 시간을 조절**하는 등 최적화가 되어 있습니다. Claude 2는 HumanEval 기준 GPT-4와 비슷한 수준의 성능을 보이며, 대화형으로 이유 설명과 함께 코드를 제안하는 등 **유연한 인터페이스**를 갖추고 있습니다. 한편 Anthropic은 별도로 **Claude Code**라는 코드 특화 버전도 제공하고 있는데, 이는 추가적인 코드 튜닝을 거쳐 **코딩 작업에 더욱 최적화**된 모델로 활용되고 있습니다.

- **Code Llama (Meta)**: Code Llama는 공개된 **오픈소스 코드 LLM** 중 가장 대표적인 모델입니다. 기본 Llama2를 약 **5000억 토큰의 코드 데이터**로 추가 학습하여 파생되었으며, **7B에서 34B, 70B** 등 여러 크기로 제공됩니다 [20]. 토크나이저는 SentencePiece 기반으로, Llama2와 동일한 32K 규모 어휘를 사용하면서도 **<PRE>, <MID>, <SUF>** 등의 **특수 토큰**을 포함하여 **중간 코드 채우기(infill)** 기능을 지원합니다 [8]. 컨텍스트 길이는 기본 16K로 늘어났고, 위치 인코딩 조정으로 **최대 100K 토큰까지** 성능 저하 없이 처리됨이 확인되었습니다 [10]. Code Llama는 파이썬에 특화된 모델과, 사용자 지시에 답하도록 미세조정된 Instruct 모델 변종도 있어 용도에 맞게 선택 가능합니다. 성능적으로는 HumanEval 기준 70B 모델이 **약 53%** 수준의 pass@1을 달성하여, OpenAI의 Codex-Cushman 등과 비슷하거나 그 이상의 결과를 보여줍니다. **오픈모델**인 만큼 커뮤니티에서 다양한 파인튜닝(예: 문제해결 능력을 높인 WizardCoder 등)이 나오고 있으며, 자유롭게 **자체 호스**

팅하여 사용할 수 있는 장점이 있습니다. 다만 GPT-4나 Claude에 비하면 아직 추론 능력이나 안정성 면에서 약간 뒤처진다는 평가가 있으며, 이는 파라미터 수와 RLHF 정도의 차이로 추정됩니다.

- **Google Gemini**: Gemini는 구글 딥마인드가 선보인 차세대 멀티모달 LLM으로, 코드 분야에서도 큰 기대를 받고 있습니다. 최신 **Gemini 3** 모델은 텍스트, 이미지, 오디오, **코드까지 복합적으로 이해**하며, 특히 **1백만 토큰**이라는 엄청난 맥락길이를 지원해 방대한 양의 정보도 한꺼번에 처리할 수 있다고 합니다 [14]. 구글은 이미 PaLM 2 기반으로 **Codey**라는 코드 생성 모델을 만들어 자사 서비스(Bard 등)에 적용한 바 있으며, Gemini에서는 이를 한층 발전시켜 **최고 수준의 코드 생성 성능**을 구현한 것으로 보입니다. 공식 발표에 따르면, Gemini 3은 여러 벤치마크에서 GPT-4를 능가하는 **최첨단 추론 능력**을 보이고 있고, **코드로부터 시각화 생성**이나 **에이전트식 문제해결** 등 새로운 능력을 갖추고 있습니다 [21] [14]. 구체적인 구현은 공개되지 않았지만, 아마도 **구글의 내부 코드 저장소와 Competitive Programming 문제** 등 광범위한 데이터를 활용하고, **툴 사용** (예: 코드 실행 검증)과 결합하여 신뢰도를 높인 것으로 추측됩니다. Gemini의 등장은 **코드 LLM 분야의 또 다른 도약**을 의미하며, 추후 공개되는 세부 정보에 따라 업계 표준이 어떻게 바뀔지 주목됩니다.

요약하면, 코드에 특화된 LLM들은 **토크나이저 단계에서 코드 문법 요소를 고려**하고, **Transformer 모델 내부적으로 긴 문맥과 구조 정보를 처리**하도록 각종 기법을 동원합니다. 일반 LLM에 비해 **긴 코드 컨텍스트, 구조적 패턴, 식별자 일관성** 면에서 강화되어 있으며, 사전학습/미세조정 과정에서도 **코드 도메인에 최적화된 목표**를 추구합니다. 그 결과 최신 코드 LLM들은 자연어 처리 모델로는 어려운 **정확한 구문**의 코드 생성, **복잡한 코드베이스 이해**, **포괄적인 리팩토링**을 수행해내고 있습니다. 이는 Transformer의 강력한 **문맥 압축** 능력과 대용량 학습의 산물로, 앞으로도 토크나이저 개선이나 구조 정보 통합(AI 코드 분석 등)이 더해지면, 인간 프로그래머와 협업하는 더욱 똑똑한 코드 생성기가 탄생할 것으로 기대됩니다.

참고 문헌: 본 답변에서는 TokDrift 논문【5】, Vizura의 토크나이저 분석【19】, HuggingFace의 Code Llama 문서【9】, Anthropic 블로그【33】, 구글 Gemini 발표【31】, ACL 2024 연구【26】 등의 출처를 참조하여 코드 특화 LLM의 구현상의 차이를 종합적으로 설명하였습니다.

---

[1] [2] [3] TokDrift: When LLM Speaks in Subwords but Code Speaks in Grammar
https://arxiv.org/html/2510.14972v1

[4] [5] [6] [7] The necessary (and neglected) evil of Large Language Models: Tokenization
https://vizuara.substack.com/p/the-necessary-and-neglected-evil

[8] [10] [19] [20] CodeLlama
https://huggingface.co/docs/transformers/en/model_doc/code_llama

[9] dpfried.github.io
https://dpfried.github.io/talks/starcoder_slides.pdf

[11] Interpolation in Positional Encodings and Using YaRN for Larger Context Window - MachineLearningMastery.com
https://machinelearningmastery.com/interpolation-in-positional-encodings-and-using-yarn-for-larger-context-window/

[12] [13] Introducing 100K Context Windows \ Anthropic
https://www.anthropic.com/news/100k-context-windows

[14] [21] Gemini 3: Introducing the latest Gemini AI model from Google
https://blog.google/products/gemini/gemini-3/

[15] [16] [17] aclanthology.org
https://aclanthology.org/2024.findings-acl.939.pdf

[18] Understanding Llama 2 and the New Code Llama LLMs
https://news.ycombinator.com/item?id=37321032