LLM과 Programing의 만남
DeepSeek-Coder
아키텍쳐 리뷰

코드 특화 LLM의 학습 구조와 성능

학부생 전범기

26.06.26

목 차

DeepSeek-Coder의 모델 구분

Base Model
기본 코드 모델

DeepSeek-Coder Base 모델은 코드 생성 능력의 기반이 되는 모델이다.

주어진 이전 문맥을 바탕으로 다음 토큰을 예측하는 방식으로 학습되며, 코드 자동완성, 함수 작성, 코드 이해와 같은 기본적인 코드 처리 능력을 갖춘다.

Instruct Model
사용자의 지시를 추가 학습한 모델

DeepSeek-Coder Instruct 모델은 Base 모델을 바탕으로, 사용자의 질문이나 명령에 맞게 응답하도록 학습된 모델이다.

단순히 코드를 이어서 생성하는 것이 아니라,
“이 코드를 수정해줘”, “오류를 설명해줘”, “파이썬으로 구현해줘”와 같은 지시를 이해하고 수행하는 데 초점을 둔다.

데이터 셋 구성

DATASET 1

87%

Github 소스코드

DATASET 2

10%

코드 관련 영어 문서
(Markdown, Stack Exchange1)

DATASET 3

3%

코드와 무관한
고품질 중국어 기사

DeepSeek-Coder의 학습 데이터는 GitHub 소스코드를 중심으로 구성되며,
코드 이해를 돕기 위한 영어 문서와 일반 언어 능력 보완을 위한 중국어 고품질 문서가 함께 포함된다.
데이터 셋은 Repositoy 단위로 필터링하여 삭제할지를 결정하며,
Python의 import, C의 include와 같이 의존관계를 필요로하는 파일들은 의존 파일이 먼저 오도록 정렬 과정을 거칩니다.

아키텍쳐 미리보기

Training Strategy
→ Next Token Prediction
→ Fill-in-the-middle(FIM)

DeepSeek-Coder가
코드를 어떤 방식으로 학습하는지 살펴본다.

단순한 다음 토큰 예측을 넘어,
코드의 중간을 채우는 학습 방식을 확인한다.

Tokenizer
→ Byte-level BPE

코드를 모델이 이해할 수 있는 단위를
어떻게 나누는지 살펴본다.

특히 코드에서 자주 등장하는 기호, 줄바꿈, 변수명 처리 방식이 왜 중요한지 확인한다.

그 외 설정
→ Model Architecture
→ Optimization

학습 전략, 토크나이저 이 외 적용한 다른 설정들을 살표본다.

Training Strategy

기존 전략의 한계

한계점
:Next Token Prediction만 존재

기존 Causal- Decoder 모델은 왼쪽에서 오른쪽으로만 텍스트를 생성하며, 뒤에 오는 문맥(Suffix)을 고려하지 못함

변경점
:가운데 채우기(infilling) task의 도입

코딩 어시스턴트를 만들 때 Infilling은

- 주석 생성
- 라이브러리 import문 생성
- 부분적으로 작성된 함수를 완성

등에 사용될 수 있습니다

Training Strategy

Next Token Prediction
:기존 Decoder-only 학습 전략

모델은 앞선 토큰들을 바탕으로 다음 토큰의 확률을 예측합니다.
학습 과정에서는 예측한 토큰과 실제 정답 토큰의 차이를 줄이도록 가중치를 업데이트합니다.

이를 통해 코드의 문법, 구조, 변수 사용 패턴을 학습합니다.

Fill-in-the-middle (FIM)
:추가된 학습 전략

코드의 앞부분과 뒷부분을 함께 보고,
비어 있는 중간 부분을 예측하는 학습 방법입니다.

이를 통해 모델은 코드 자동완성뿐 아니라, 중간 코드 수정이나 함수 내부 구현처럼 실제 코딩 상황에 가까운 능력을 익힙니다.

각각은 5:5 비율로 학습되며,
PPT에서는 FIM 위주로 다룹니다.

Fill-in-the-middle (FIM)

OPEN AI 논문

핵심 메커니즘

1. 문서를 Prefix, Middle, Suffix 세부분으로 나눈다.
2. Middle 부분을 지우고, Middle 을 예측하는 task로 학습을 진행한다.

FIM의 종류

1. Prefix -Suffix → Middle (PSM 방식)
2. Suffix- Prefix → Middle (SPM 방식)

특수 토큰 예시(PSM)

각각을 토큰으로 구분한다.

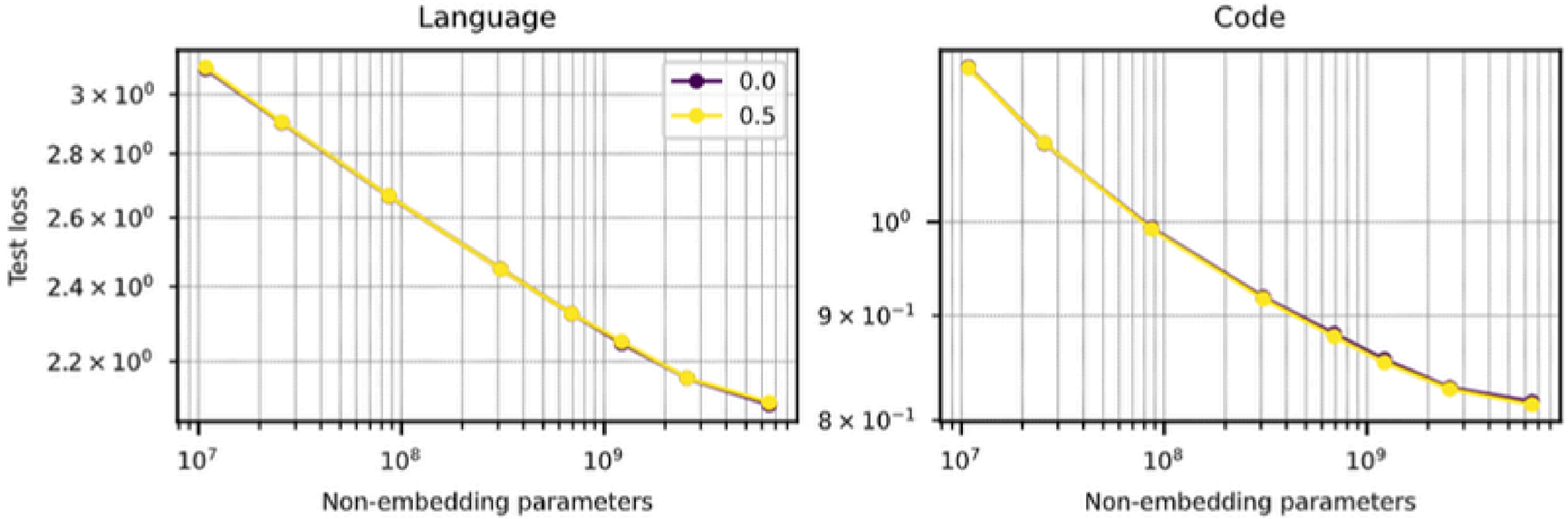
$$\texttt{<PRE>} \circ \text{Enc}(\text{prefix}) \circ \texttt{<SUF>} \circ \text{Enc}(\text{suffix}) \circ \texttt{<MID>} \circ \text{Enc}(\text{middle}),$$

Fill-in-the-middle (FIM)

OPEN AI 논문

FIM의 주요발견

- FIM - For - Free 특성 : FIM 학습을 위해 데이터를 50% 섞었음에도 기존 능력이 유지된다.(NTP 100% → NTP 50%, FIM 50%)



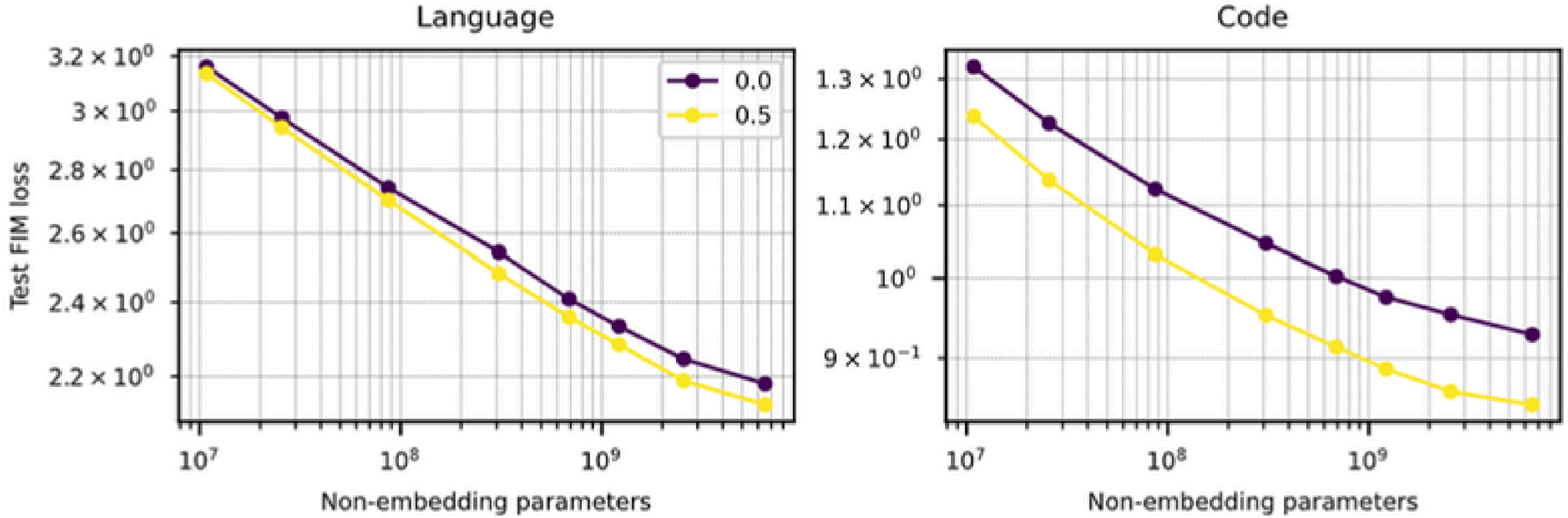
그림은 FIM의 비율을 0% 50%로 각각 설정하여 cross entropy값 측정한 그림이다.
task는 다음 토큰을 예측하는 것이다.

Fill-in-the-middle (FIM)

OPEN AI 논문

FIM의 주요발견

- Infilling 특성 : 또한, infilling(채워넣기)라는 추가적인 기능이 생긴다.



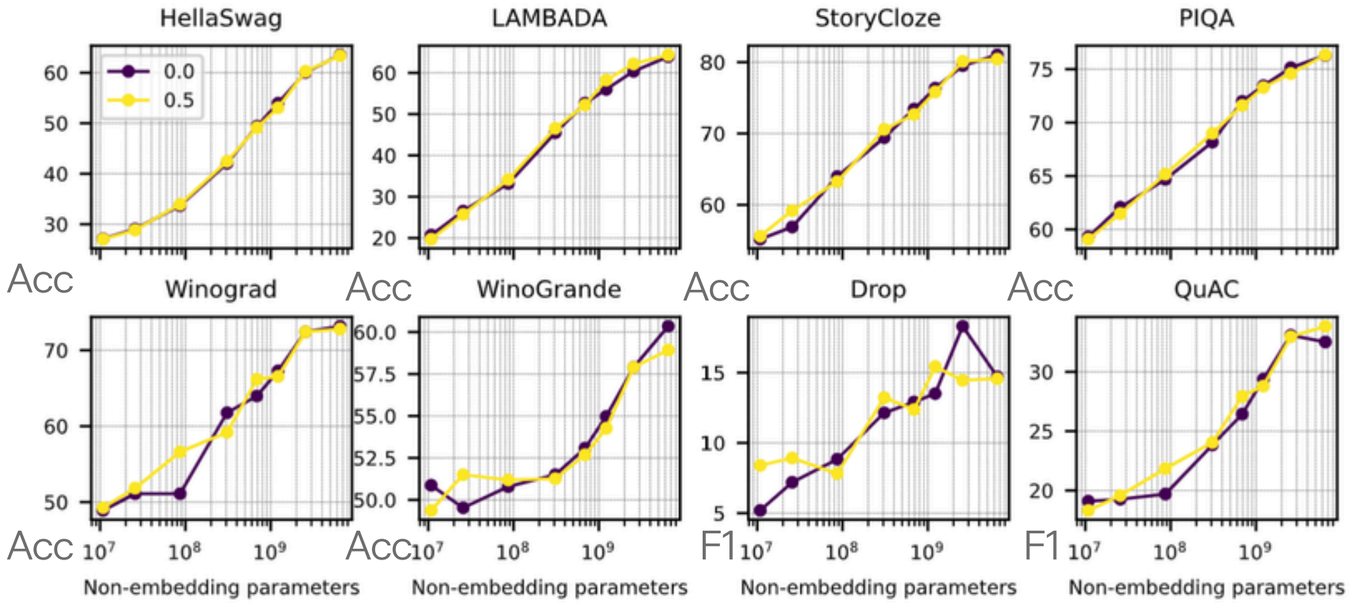
그림은 FIM의 비율을 0% 50%로 각각 설정하여 cross entropy값 측정한 그림이다.
task는 가운데 채우기이다.

Fill-in-the-middle (FIM)

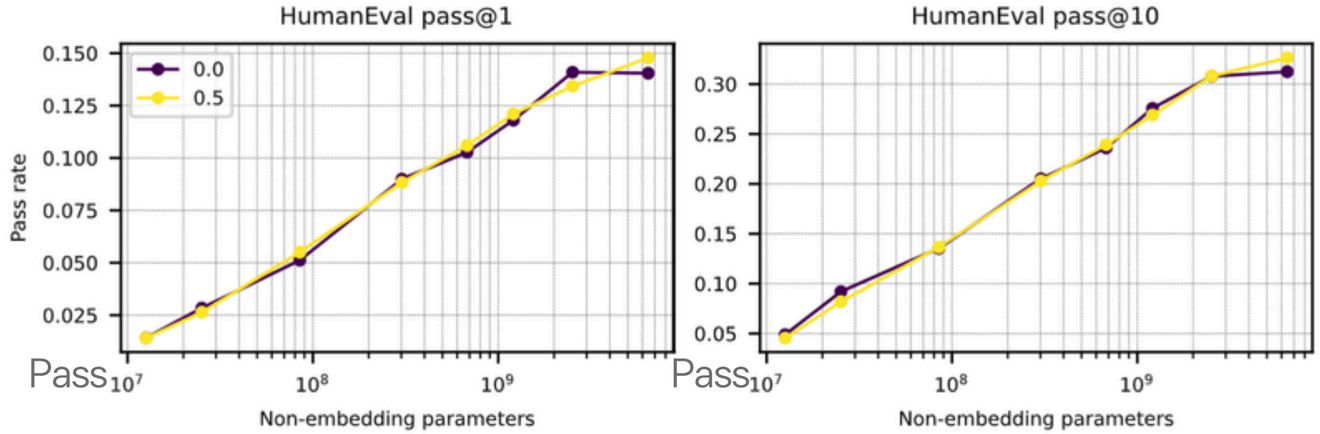
OPEN AI 논문

FIM의 주요발견

- 여러 벤치마크에서의 성능
  - **HellaSwag**: 상식적 문장 완성
  - **LAMBADA**: 문맥 기반 단어 예측
  - **StoryCloze**: 이야기 결말 추론
  - **PIQA**: 물리적 상식 추론
  - **Winograd / WinoGrande**: 대명사 지칭 대상 파악
  - **DROP**: 수치 계산 및 독해
  - **QuAC**: 대화형 지문 답변
  - **HumanEval**: 코딩 문제 해결 및 실행 검증



(a) Comparison of natural language results. We report F1 for Drop and QuAC and accuracy for the rest.



Fill-in-the-middle (FIM)

OPEN AI 논문

- 추가적인 실험

학습 전략인
NTP와 FIM의 비율

FIM의 종류인
PSM과 SPM의 비율

FIM의 학습 타이밍인
Fine-tuning
vs Pre-training

NTP와 FIM의 비율

Fill-in-the-middle (FIM)

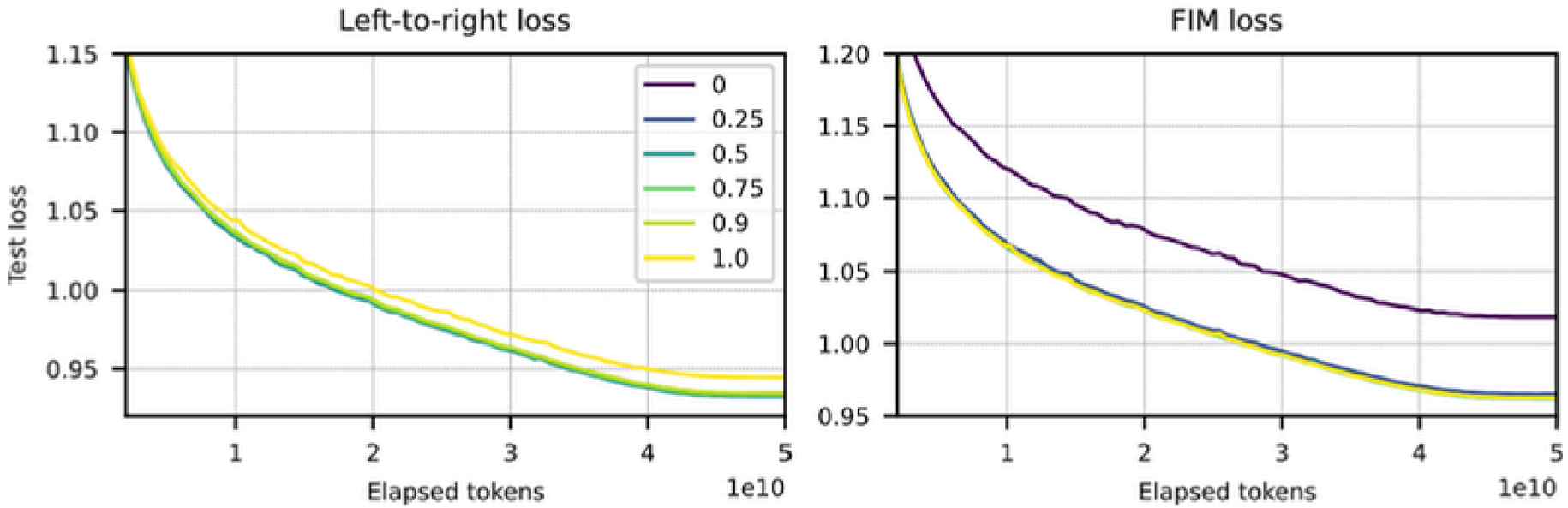
OPEN AI 논문

NTP와 FIM의 비율

- AR성능은 90%까지는 기존의 다음 토큰 예측 능력이 전혀 떨어지지 않지만, 100% FIM 데이터로만 학습하면 기존 생성 능력이 저하
- Infilling성능은 FIM비율에 따라 높아지는것을 확인

논문 추천 FIM 비율

50% ~ 90%



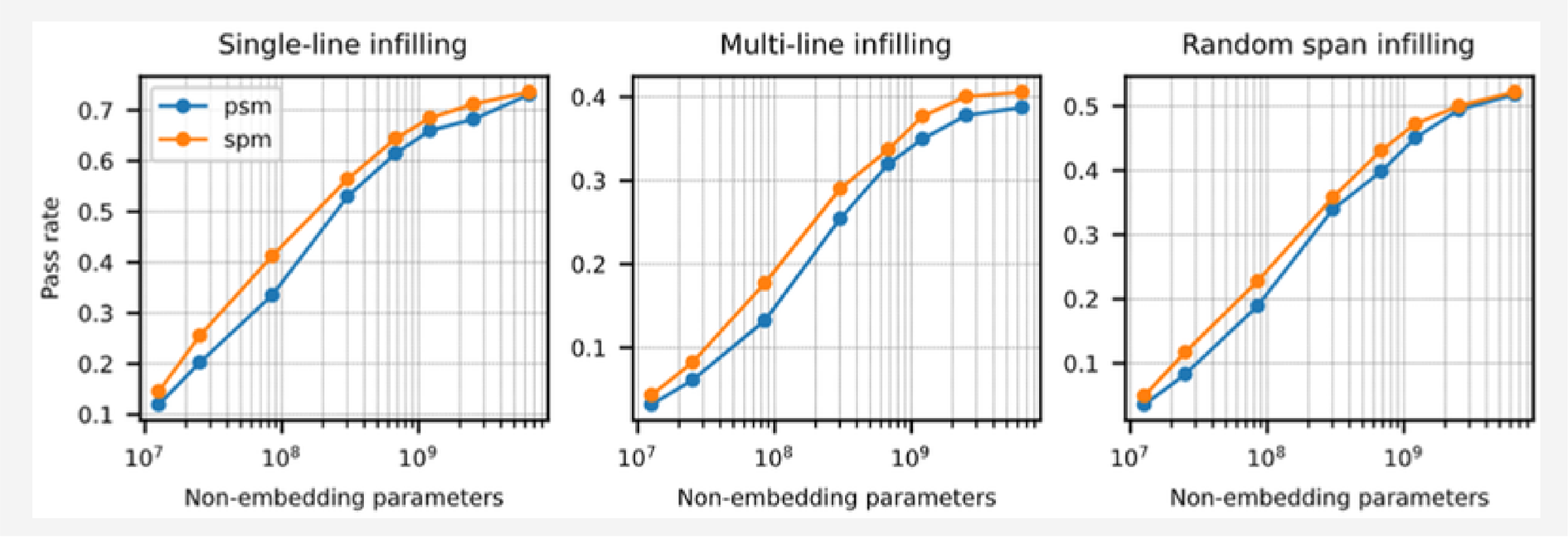
PSM과 SPM의 비율

Fill-in-the-middle (FIM)

OPEN AI 논문

NTP(0.5),FIM(0.5) 모델 성능

한 줄 채우기 여러 줄 채우기 무작위 구간 채우기



Fill-in-the-middle (FIM)

OPEN AI 논문

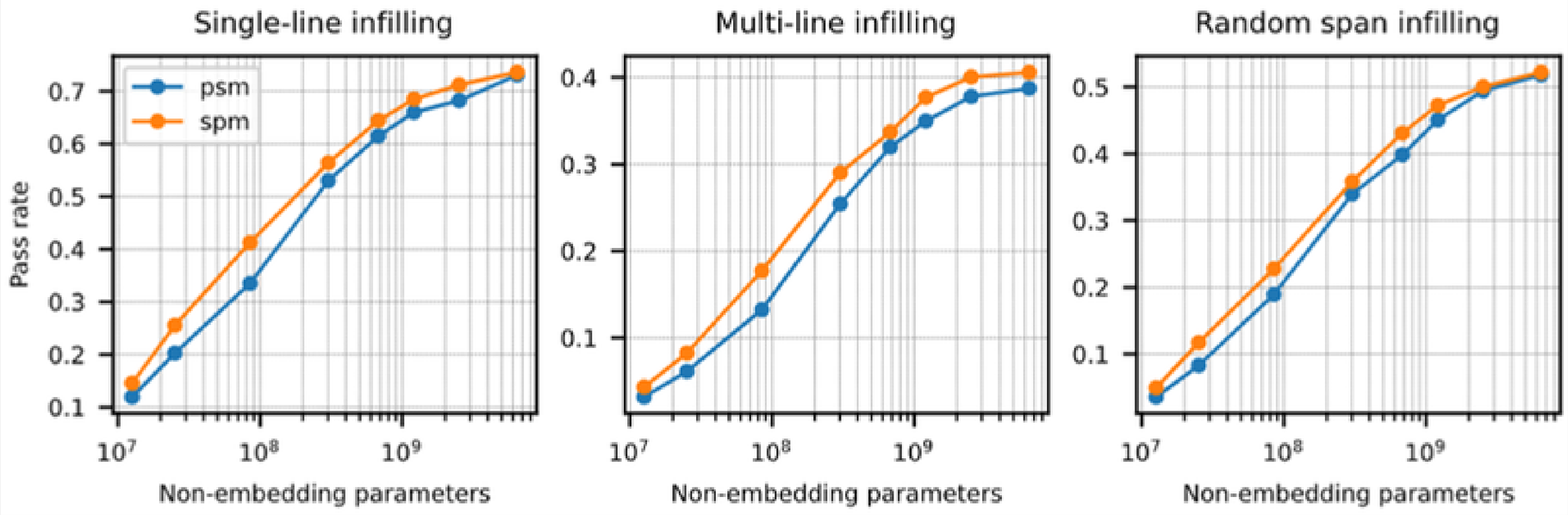
NTP(0.5),FIM(0.5) 모델 성능

SPM이 더 높은 성능을 가진다.

=> OPEN AI 논문에서는 SPM의 구조가 PSM보다 유리하다고 예측

==> 왜냐하면 PM 의 순서가 기존 PMS 순서와 유사하기 때문일 것이다.

한 줄 채우기 여러 줄 채우기 무작위 구간 채우기



Fill-in-the-middle (FIM)

OPEN AI 논문

그럼에도 SPM 하나만을 쓰지 않는 이유

- 긍정적 전이 효과: PSM과 SPM을 섞어 학습하면 성능이 더 좋아진다.

논문 추천 FIM 비율

50% : 50%

1st

2nd

Train distribution	FIM rate	Single-line		Multi-line		Random span	
		PSM	SPM	PSM	SPM	PSM	SPM
Joint	0.5	0.550	0.595	0.265	0.293	0.367	0.379
Joint	0.9	0.616	0.622	0.290	0.305	0.397	0.420
PSM	0.9	0.583	0.625	0.273	0.305	0.362	0.274
SPM	0.9	0.023	0.586	0.008	0.301	0.007	0.386

한 줄 채우기 여러 줄 채우기 무작위 구간 채우기

FIM의 적용방법

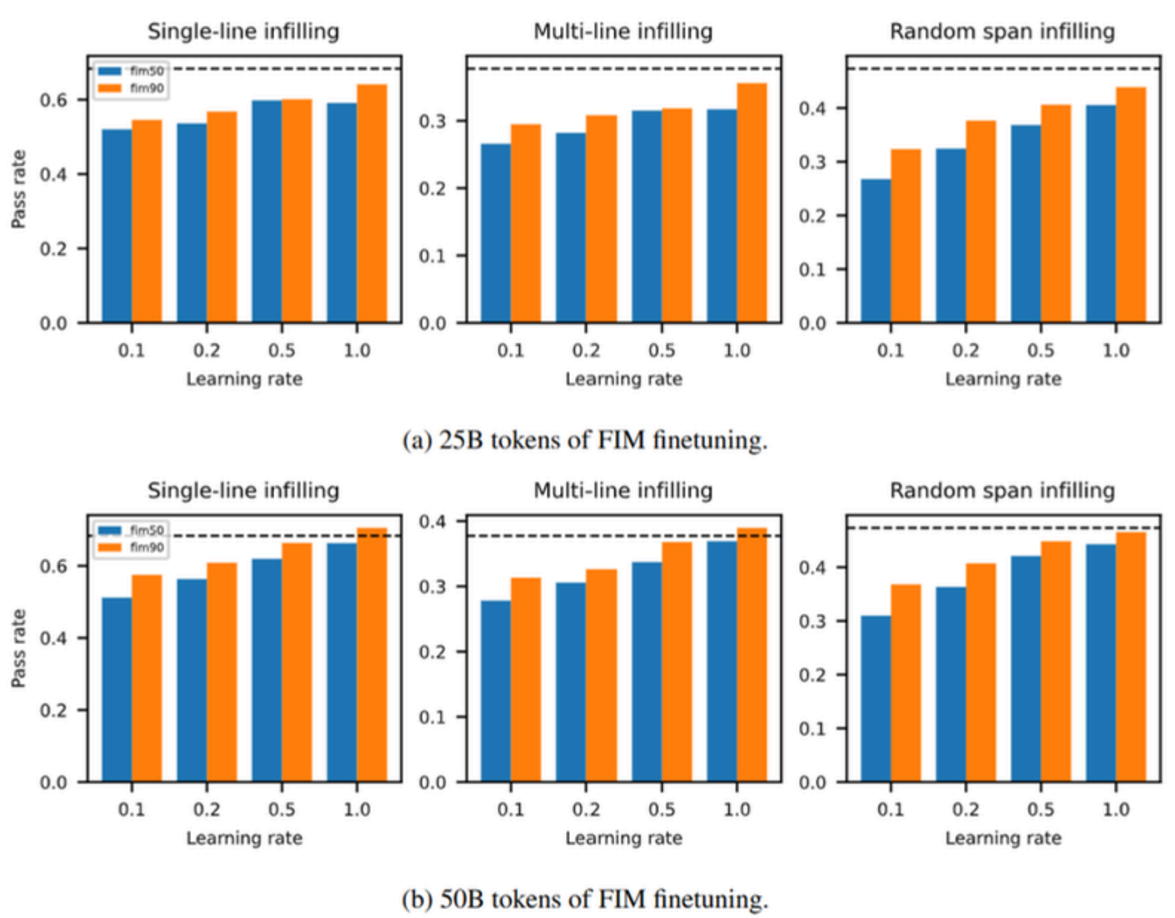
Fine tuning vs Pre-training

Fill-in-the-middle (FIM)

OPEN AI 논문

PRE-training의 승리

PRE-training의 성능이 압도적이라
Fine tuning이 따라잡기에 비용이 너무크다.



(a) 25B tokens of FIM finetuning.

(b) 50B tokens of FIM finetuning.

점선:baseline(FIM 50%비율)
파랑막대: FIM 50%
주황막대:FIM 90%
Fine tuning때 비율을 설정한
이유는 AR능력을 잃지 않기 위함.

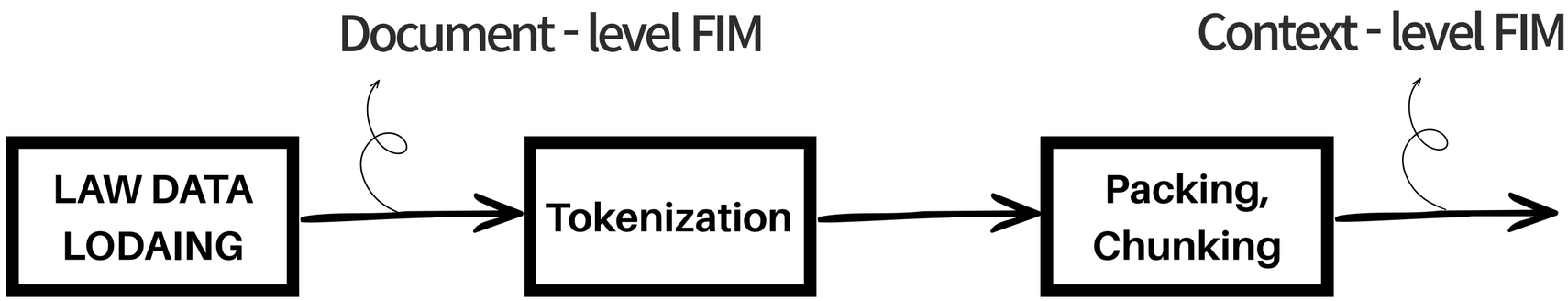
Fill-in-the-middle (FIM)

OPEN AI 논문

추가 설정(FIM 데이터 정제 과정)

FIM 데이터 정제의 두 가지 적용 레벨

- FIM의 데이터를 나누는 단계는 Document-level, Context-level두가지로 나뉜다.
- Law Data를 문자열 상태에서 Prefix, Middle, Suffix로 삼등분하는 Document -level FIM
- 데이터를 토큰화 및 청킹을 끝난후에 삼등분하는 Context-level FIM



Fill-in-the-middle (FIM)

OPEN AI 논문

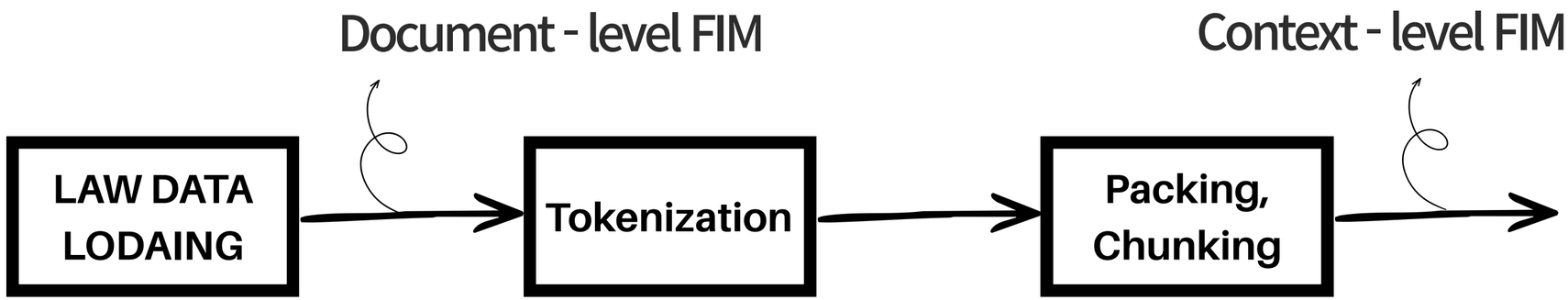
추가 설정(FIM 데이터 정제 과정)

Document - level FIM의 단점

- 문서를 FIM으로 변환 후 Context Size(2048 길이)로 자르면, 운이 나쁠경우 구간이 Context Size 안에 모두 들어가지 못하거나 잘리게 된다.

Context - level FIM의 장점

- 이미 Context Size로 잘린 덩어리 안에서 FIM을 적용하므로, Prefix, Suffix, Middle 세 조각이 반드시 Context 안에 존재하게 된다.



Tokenizer

Tokenizer(BBPE)

Coder에서는 BBPE를 썼습니다.

논문에서는 BPE라고 나와있지만, 공식 Github Q&A 항목에서 토크나이저를 BBPE라고 언급했습니다. 논문에 나와있는 BPE라고 적힌것은 BPE의 종류 중 하나인 BBPE라고 간략히 설명한 것으로 보입니다.

또한, 학습 방법은 다음 논문을 따릅니다.

“Neural Machine Translation of Rare Words with Subword Unit”
- University of Edinburgh

Tokenizer(BBPE)

BPE의 적용 과정 Unviversity of Edinburgh 논문

1. 모든 단어를 문자 단위로 분리하고, 특수 기호를 추가한다.
2. 모든 기호 쌍(symbol pairs)의 빈도를 계산하여 빈번하게 등장하는 쌍을 하나의 새 기호로 병합한다.
   →빈도 높은 단어는 하나의 긴 유닛으로 병합되고, 드문 단어는 작은 하위 단어로 쪼개진 채 남게 된다.
3. 테스트 시에 처음 보는 단어가 등장하더라도, 훈련 시 학습한 병합 규칙을 순차적으로 적용하여 이미 알고 있는 기호들의 시퀀스로 분할할 수 있으므로 미등록어 문제를 해결한다.

BBPE란?

BPE를 문자 단위가 아닌 Byte level에서 수행하는 토크나이저 방식.

BPE는 문자 기반이기에 더 작은 문자 단위로 쪼갤수가 없다.
즉, 문자 자체가 Out of Vocabulary(단어장 이외) 문자라면,
UNK으로 표시된다.

이는, 세미콜론, 큰 따옴표 등 특수 문자를 자주 쓰는 코딩에서
큰 단점이기에 Byte level에서 처리하는 BBPE를 적용했다.

그 외 설정

그 외 설정

DeepSeek v1모델을 기반을 따릅니다.

Model Architecture

- RoPE:회전 행렬을 통해 위치 정보를 주입하여 상대적 거리를 보존하는 인코딩
- GQA:성능과 속도의 완벽한 균형점을 찾은 Attention 구조
- Flash Attention v2:GPU 메모리 계층을 이해하고 병목 현상을 제거한 혁신적 알고리즘

Optimization

- AdamW:Adam optimizer에 가중치 감소(weight decay)를 더 정확히 분리 적용해, 모델이 과적합되지 않도록 하면서 안정적으로 학습시키는 최적화 알고리즘

그 외 설정

DeepSeek v1모델과 다른 점

Long Context

- 배경: 일반적인 모델은 한 번에 읽을 수 있는 토큰의 양이 제한적입니다. 하지만 DeepSeek-Coder는 전체 프로젝트의 의존성을 파악해야 하는 'Repository-level' 시나리오를 처리하기 위해 긴 컨텍스트 지원이 필수적이었습니다.

$$\begin{cases} \boldsymbol{p}_{i,2t} & = \sin(k/10000^{2t/d}) \\ \boldsymbol{p}_{i,2t+1} & = \cos(k/10000^{2t/d}) \end{cases}$$

변경 전

- 적용 내용: RoPE의 식에서 scaling factor를 1 → 4로 늘리고, base frequency를 10000 → 100000으로 변경했습니다.

  또한, 16K 시퀀스 길이와 512 배치 사이즈로 1,000 step의 추가 학습을 진행했습니다.

$$\begin{cases} \boldsymbol{p}_{i,2t} & = \sin((k/4)/100000^{2t/d}) \\ \boldsymbol{p}_{i,2t+1} & = \cos((k/4)/100000^{2t/d}) \end{cases}$$

변경 후

결과

결과

1st

2nd

Model	Size	Python	C++	Java	PHP	TS	C#	Bash	JS	Avg	MBPP
Multilingual Base Models											
code-cushman-001	12B	33.5%	31.9%	30.6%	28.9%	31.3%	22.1%	11.7%	-	-	-
CodeGeeX2	6B	36.0%	29.2%	25.9%	23.6%	20.8%	29.7%	6.3%	24.8%	24.5%	36.2%
StarCoderBase	16B	31.7%	31.1%	28.5%	25.4%	34.0%	34.8%	8.9%	29.8%	28.0%	42.8%
CodeLlama	7B	31.7%	29.8%	34.2%	23.6%	36.5%	36.7%	12.0%	29.2%	29.2%	38.6%
CodeLlama	13B	36.0%	37.9%	38.0%	34.2%	45.2%	43.0%	16.5%	32.3%	35.4%	48.4%
CodeLlama	34B	48.2%	44.7%	44.9%	41.0%	42.1%	48.7%	15.8%	42.2%	41.0%	55.2%
DeepSeek-Coder-Base	1.3B	34.8%	31.1%	32.3%	24.2%	28.9%	36.7%	10.1%	28.6%	28.3%	46.2%
DeepSeek-Coder-Base	6.7B	49.4%	50.3%	43.0%	38.5%	49.7%	50.0%	28.5%	48.4%	44.7%	60.6%
DeepSeek-Coder-Base	33B	56.1%	58.4%	51.9%	44.1%	52.8%	51.3%	32.3%	55.3%	50.3%	66.0%
Instruction-Tuned Models											
GPT-3.5-Turbo	-	76.2%	63.4%	69.2%	60.9%	69.1%	70.8%	42.4%	67.1%	64.9%	70.8%
GPT-4	-	84.1%	76.4%	81.6%	77.2%	77.4%	79.1%	58.2%	78.0%	76.5%	80.0%
DeepSeek-Coder-Instruct	1.3B	65.2%	45.3%	51.9%	45.3%	59.7%	55.1%	12.7%	52.2%	48.4%	49.4%
DeepSeek-Coder-Instruct	6.7B	78.6%	63.4%	68.4%	68.9%	67.2%	72.8%	36.7%	72.7%	66.1%	65.4%
DeepSeek-Coder-Instruct	33B	79.3%	68.9%	73.4%	72.7%	67.9%	74.1%	43.0%	73.9%	69.2%	70.0%

시간의 흐름~ 타임라인 레이아웃

Step 1

모델 구분

Base는 코드 생성 기반, Instruct는 사용자 지시 수행 모델이다.

Step 2

데이터 구성

GitHub 코드 중심으로 문서 데이터를 함께 학습해 코드 이해 능력을 높였다.

Step 3

학습 전략

NTP에 FIM을 추가해 코드 중간을 채우는 능력을 학습했다.

Step 4

Tokenizer

Byte-level BPE로 특수 문자, 변수명, 줄바꿈 등을 안정적으로 처리한다.

Step 5

아키텍처 보완

RoPE, GQA, Flash Attention v2, Long Context로 긴 코드 문맥 처리 능력을 높였다.

감사합니다.