발표자료 Draft

보조 데이터를 활용하여 pretrained 모델을 구축하고 학습 데이터로 fine-tuning을 진행하여 pretrained 모델의 성능을 평가하십시오.

학습 데이터

Project CodeNet[링크]

보조 데이터

월간 데이콘 코드 유사성 판단 AI 경진대회[링크]

CodeNet_EDA

CodeNet 전처리

Ideas

💡 실행되지 않는 코드는 어떻게 구분할 것인가?
코드 일관성을 어떻게 유지할 것인가?
주석와 docstring은 어떻게 처리할 것인가?
중복된 코드를 어떻게 감지하고 처리할 것인가?
스페셜 토큰은 얼마나 만들 것인가?
메타데이터를 어떻게 이용할 것인가?

Brainstorming

💡 AST 파서를 통과하는 코드만 사용
python black 코드 포매터 사용
주석/docstring 제거 또는 input으로 사용
AST트리를 통해 동일 문제에서 동일 구조인 코드들은 중복된 것으로 판별
스페셜토큰 <NEWLINE> <INDENT> <DEDENT>등 추가
코드 내 숫자 변수와 문자열 변수(Literal)는 <NUM> <STR> 등으로 치환
동일 코드 내 동일 변수는 같은 이름으로 대체 (var_1, var_2 ...)
메타데이터는 사용하지 않고, Status: Accepted만 사용
메타데이터를 사용해서 Positive/Negative Pairing을 통해 대조 학습

전처리 test

코드포맷팅 - 주석제거 -AST파싱 후 스코프 일관 치환 → 식별자 정규화 → 리터럴 값 정규화

▼ 전처리 sample

```
--- sample (no_comments) ---
 def makelist(n, m):
    return [[0 for _ in range(m)] for _ in range(n)]


N, M = map(int, input().split())
A = [0] + list(map(int, input().split()))

imos = [0]*(N+1)
for i in range(1, N+1):
    imos[i] = (A[i] + imos[i-1]) % M

d = {}
for i in range(N+1):
    if imos[i] not in d:
        d[imos[i]] = 1
    else:
        d[imos[i]] += 1

ans = 0
for v in d.values():
    ans += (v * (v-1)) // 2

print(ans)


--- sample (anon) ---
 def func_1(var_1, var_2):
    return [[0 for var_3 in range(var_2)] for var_3 in range(var_1)]
var_7, var_8 = map(int, input().split())
var_9 = [0] + list(map(int, input().split()))
var_10 = [0] * (var_7 + 1)
for var_12 in range(1, var_7 + 1):
    var_10[var_12] = (var_9[var_12] + var_10[var_12 - 1]) % var_8
var_21 = {}
for var_12 in range(var_7 + 1):
    if var_10[var_12] not in var_21:
        var_21[var_10[var_12]] = 1
    else:
        var_21[var_10[var_12]] += 1
var_33 = 0
```

```
for var_34 in var_21.values():
    var_33 += var_34 * (var_34 - 1) // 2
print(var_33)

--- sample (norm) ---
 def func_1(var_1, var_2):
    return [[<NUM_D1> for var_3 in range(var_2)] for var_3 in range(var_1)]
var_7, var_8 = map(int, input().split())
var_9 = [<NUM_D1>] + list(map(int, input().split()))
var_10 = [<NUM_D1>] * (var_7 + <NUM_D1>)
for var_12 in range(<NUM_D1>, var_7 + <NUM_D1>):
    var_10[var_12] = (var_9[var_12] + var_10[var_12 - <NUM_D1>]) % var_8
var_21 = {}
for var_12 in range(var_7 + <NUM_D1>):
    if var_10[var_12] not in var_21:
        var_21[var_10[var_12]] = <NUM_D1>
    else:
        var_21[var_10[var_12]] += <NUM_D1>
var_33 = <NUM_D1>
for var_34 in var_21.values():
    var_33 += var_34 * (var_34 - <NUM_D1>) // <NUM_D1>
print(var_33)

--- sample (with_layout) ---
 def func_1 ( var_1 , var_2 ) : <NL> <INDENT> return [ [ < NUM_D1 > for var_3 in range ( var_2 ) ] for var_3 in range ( var_1 ) ] <NL> <DEDENT> var_7 , var_8 = map ( int , input ( ) . split ( ) ) <NL> var_9 = [ < NUM_D1 > ] + list ( map ( int , input ( ) . split ( ) ) ) <NL> var_10 = [ < NUM_D1 > ] * ( var_7 + < NUM_D1 > ) <NL> for var_12 in range ( < NUM_D1 > , var_7 + < NUM_D1 > ) : <NL> <INDENT> var_10 [ var_12 ] = ( var_9 [ var_12 ] + var_10 [ var_12 - < NUM_D1 > ] ) % var_8 <NL> <DEDENT> var_21 = { } <NL> for var_12 in range ( var_7 + < NUM_D1 > ) : <NL> <INDENT> if var_10 [ var_12 ] not in var_21 : <NL> <INDENT> var_21 [ var_10 [ var_12 ] ] = < NUM_D1 > <NL> <DEDENT> else : <NL> <INDENT> var_21 [ var_10 [ var_12 ] ] += < NUM_D1 > <NL> <DEDENT> <DEDENT> var_33 = < NUM_D1 > <NL> for var_34 in var_21 . values ( ) : <NL> <INDENT> var_33 += var_34 * ( var_34 - < NUM_D1 > ) // < NUM_D1 > <NL> <DEDENT> print ( var_33 ) <NL>
```

문제

1. Black formatter는 시간이 너무 오래 걸림 → 정규식을 이용해 들여쓰기, 공백만 포매팅
2. AST트리를 이용한 중복제거는 포맷팅이 크게 필요하지 않음 → 유지
3. 코드 내 개행과 들여쓰기 의미 보존을 위해 <NEWLINE><INDENT><DEDENT>등의 스페셜토큰을 사용하고자 했으나, 리서치를 통해 code-specific BPE 등을 통해 의미 보존이 가능한 것을 확인함 →사용 중지

최종 전처리 과정

1. 주석 제거
2. 공백, 들여쓰기 정규화
3. AST 파서 검증
4. AST 기반 중복 코드 제거

Code-specific LM Research

모델 개요 비교

구분	RoBERTa-small	ELECTRA-small	CodeT5-small
아키텍처	Encoder-only	Encoder-only (Gen + Disc)	Encoder-Decoder
크기	6L × H512 (50-60M)	Gen: 2-3L × H256 Disc: 6L × H512	Encoder + Decoder (소형)
학습 방식	MLM (Masked LM)	RTD (Replaced Token Detection)	Span Denoising
최적 용도	임베딩 비교, 유사도/검색, 클론 탐지	데이터 효율 극대화, 소규모 데이터 학습	이해+생성 겸용, 요약/주석/복원

학습 특성 비교

구분	RoBERTa-small	ELECTRA-small	CodeT5-small
학습 신호	마스킹된 15% 위치만	모든 토큰 위치 (진짜/가짜 판별)	연속 span 복원
샘플 효율	보통	최고	보통-높음
수렴 속도	빠름	가장 빠름	상대적으로 느림
구현 복잡도	단순	중간 (2개 네트워크)	복잡 (Enc-Dec)
메모리/속도	가장 효율적	효율적	상대적으로 높음

토크나이저

구분	공통 사항
타입	SentencePiece Unigram (32k)
스페셜 토큰	`<indent>` `<dedent>` `<newline>` `<str>` `<num>` `<pad>` `<s>` `</s>` `<mask>` `<nl>` `<py>` `<unk>`
선택 이유	코드의 희소성에 강함, 소규모 데이터 수렴 안정적, OOV 억제

학습 하이퍼파라미터 (T4 기준)

구분	RoBERTa-small	ELECTRA-small	CodeT5-small
max_len	512	512	512
batch_size	32-48	32-48	작게 조정 필요
epochs	2-3	2-3	2-3
learning_rate	3e-4 ~ 8e-4	유사	2e-4 ~ 6e-4
warmup_ratio	0.05-0.1	0.05-0.1	0.05-0.1
마스킹 비율	15%	15% (Gen)	15-30% (span)

400MB 데이터 적합성

모델	400MB 성능	이유
RoBERTa-small	⭐⭐⭐⭐⭐	가장 안정적, 빠른 수렴, 검증된 성능
ELECTRA-small	⭐⭐⭐⭐⭐	최고 샘플 효율, 소규모 데이터에 최적
CodeT5-small	⭐⭐⭐⭐	가능하나 수렴 시간/안정성 주의 필요

장단점 요약

RoBERTa-small

장점

- 샘플 효율 높음
- 임베딩 추출 속도 빠름
- 검증된 안정성

단점

- 생성 작업 불가
- 디코더형 태스크 약함

ELECTRA-small

장점

- 샘플 효율 최고
- 소규모 데이터에 최적
- 빠른 성능 도달
- 임베딩 품질 우수

단점

- 구현 복잡도 증가
- 두 네트워크 관리 필요
- 튜닝 파라미터 많음

CodeT5-small

장점

- 이해+생성 멀티태스크
- 확장성 최고 (요약/주석/변환)
- 구조적 문맥 학습 우수
- 결손 블록 복원에 강함

단점

- 메모리/시간 증가
- 수렴 상대적으로 느림
- 복잡한 아키텍처

선택 가이드

```
┌─ 속도·안정성 최우선 → RoBERTa-small
│
├─ 데이터 효율 극대화 → ELECTRA-small
│
└─ 이해+생성 겸용 → CodeT5-small
```

코드 유사도 판별 - 5명 종합 비교 분석

📋 Overview

구분	이서율	조병률	황호성	오정탁	이홍기
Pretraining 방식	대조학습 (Contrastive Learning)	Custom BERT (MLM)	RoBERTa-small (MLM)	Custom CodeBERT (MLM + Role)	CodeBERT (MLM + RTD)
핵심 아이디어	MoCo 기반 듀얼 인코더	BERT from scratch	RoBERTa 최적화 구현	Role Embedding 추가	RTD 보조 태스크
구조적 특징	Text + AST 융합	단일 Transformer	단일 Transformer	단일 Transformer + Role	단일 Transformer
학습 목표	의미적 유사도 학습	토큰 복원 학습	토큰 복원 학습	MLM + Role 예측	MLM + RTD
Tokenizer	Unigram (32k)	SentencePiece (32k)	Unigram (32k)	BPE (50k)	BPE (50k)

성능 비교

📊 최종 점수

이름	접근법	Public Score	Private Score	특징
황호성	RoBERTa	**0.9167**	**0.9164**	전처리 최적화
이홍기	MLM + RTD	**0.9284**	**0.9286**	HF 사전학습 모델 활용
조병률	Custom BERT	**0.8633**	**0.8632**	기본 MLM
오정탁	Role Embedding	**0.9317**	**0.9318**	구문 정보 활용
이서율	대조학습	**0.6218**	**0.6203**	AST 구조 활용

1️⃣ 이서율 - 대조학습 (Contrastive Learning)

🎯 핵심 컨셉

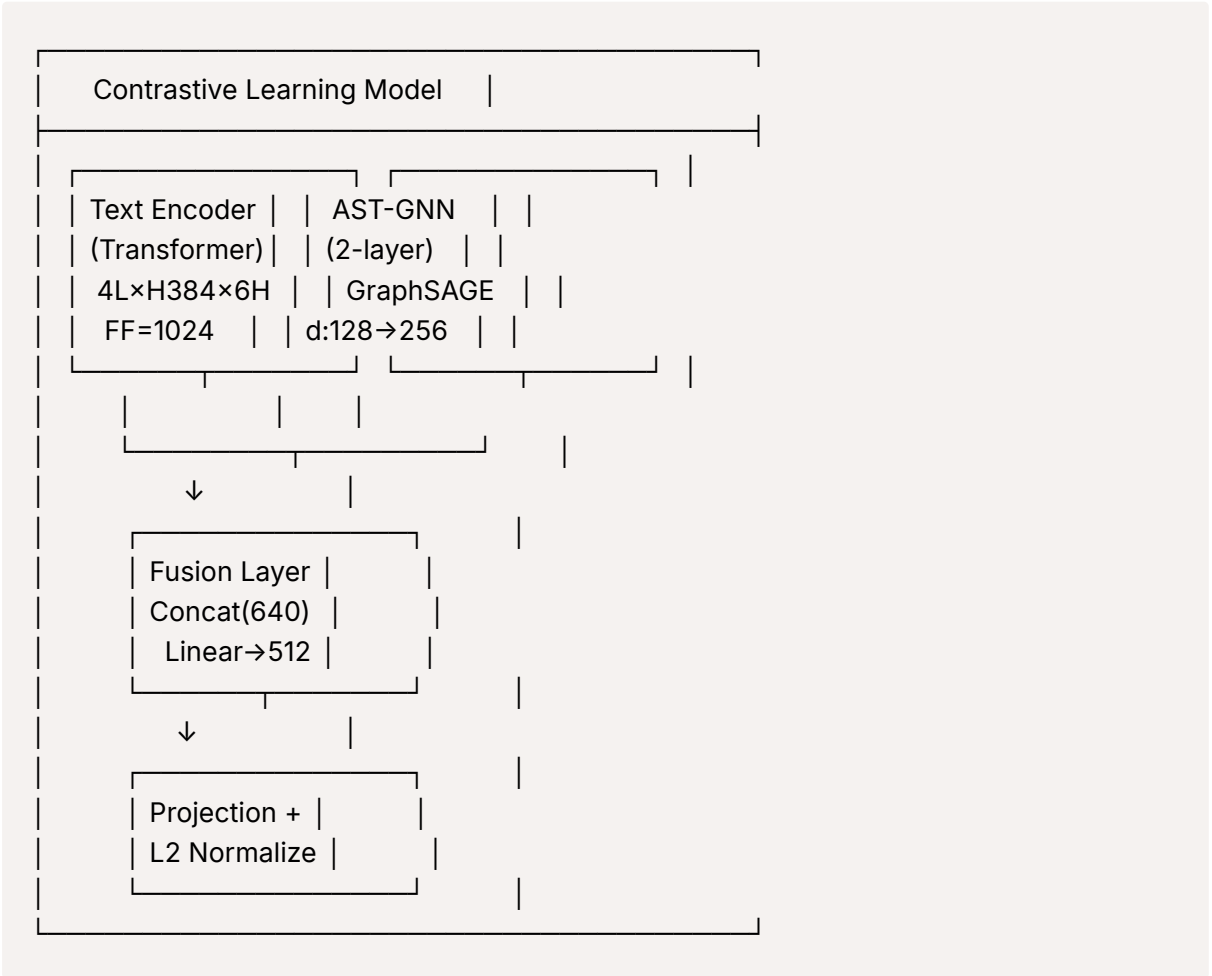
MoCo(Momentum Contrast)를 코드 도메인에 적용한 **듀얼 인코더 구조**

📦 학습 데이터 구성

Pair 생성 전략

쌍 종류	설명	학습 목표	가중치
Positive	동일 문제 AC 쌍	유사(1) 학습	1.0
Hard Negative	동일 문제 AC vs WA	미묘한 차이 구분	1.5
Semi-hard Negative	다른 문제, 유사 구조 (n-gram 사용)	일반화 학습	1.3
Negative	완전히 다른 코드	기본 음성 샘플	1.0

🏗️ 모델 구조



📐 하이퍼파라미터

- **Text Encoder**: 4 layers, 6 heads, hidden 384, FFN 1024
- **AST-GNN**: 2 layers GraphSAGE, 128→256
- **Loss**: CE(0.3) + BCE(1.0), τ=0.2
- **Optimizer**: AdamW, lr=2e-4, warmup=1000

💭 회고

장점
- AST 구조 정보 명시적 반영
- Hard negative로 미묘한 차이 학습

개선 필요
- CV 기법의 코드 도메인 적응 부족 (MLM과 같은 의미 학습 보강 필요성)
- 구현 복잡도 높음
- 태스크 특화 수정 필요

2️⃣ 조병률 - Custom BERT

🎯 핵심 컨셉

처음부터 빌드한 **BERT 아키텍처**로 CodeNet에서 MLM 학습

🏗️ 모델 구조

항목	값
어휘 크기	32,005
히든 차원	512
레이어 수	12
어텐션 헤드	8
FFN 차원	2,048
드롭아웃	0.1

```
Input IDs
   ↓
Token Embeddings (32005 × 512)
   ↓
Position Embeddings (max_pos × 512)
   ↓
LayerNorm + Dropout
   ↓
12× Transformer Encoder Layer
   ├─ Multi-Head Attention (8 heads)
   ├─ FFN (512 → 2048 → 512)
   └─ Residual + LayerNorm
   ↓
MLM Head
   └─ Linear (512 → 32005)
```

📚 Pretraining: MLM

마스킹 전략

- **마스킹 비율**: 15%
- **마스킹 규칙**: 80% [MASK], 10% 원본, 10% 랜덤

학습 설정

- learning_rate: 5e-4
- batch_size: 32
- epochs: 10
- optimizer: adamw_torch_fused
- Dataset: CodeNet

🎯 Fine-tuning

구조 변경
- MLM Head 제거
- Sequence Classification Head 추가 (Linear 512→2)

📊 결과

- **Public**: 0.8632535708
- **Private**: 0.8631608236

💭 의의

- 기본에 충실한 구현
- Baseline 역할
- 확장 가능성

3️⃣ 황호성 - RoBERTa-small

🎯 핵심 컨셉

RoBERTa 최적화 기법을 적용한 효율적 사전학습

🔤 Tokenizer: Unigram (32k)

Special Tokens 설계

```
# 구조 마커<indent>, <dedent>, <newline># 리터럴 추상화<str>, <num># 기본 토큰<pad>, <s>, </s>, <mask>, <nl>, <py>, <unk>
```

선택 이유

- ✅ 희귀 식별자 분절 안정성 ↑
- ✅ 400MB 소규모 데이터 수렴 빠름
- ✅ OOV 억제 효과
- ✅ 개념 일반화 (`<str>`, `<num>`)
- ✅ 구조 이해 (`<indent>`, `<dedent>`)

🏗️ 모델 구조: RoBERTa-small

```
Input (Token IDs)
    ↓
Embeddings
    ├─ Token Embedding (32000 × 512)
    ├─ Position Embedding (514 × 512)
    └─ LayerNorm + Dropout
    ↓
6× Transformer Encoder Layer
    ├─ Multi-Head Self-Attention (8 heads)
    └─ Position-wise FFN (512→2048→512)
    ↓
MLM Head
    └─ Decoder (512 → 32000)
```

세부 구성

- Embeddings: word(32000×512) + position(514×512)
- Encoder Layers: 6 layers
- Attention: 8 heads, d_model=512
- FFN: 512 → 2048 → 512 + GELU
- Total Params: ~50-60M

📚 Pretraining: MLM

학습 철학

> “사람이 문맥을 이해해야 빈칸을 채우듯, 모델도 문맥적 표현을 학습하여 임베딩 품질 향상”

하이퍼파라미터

- epochs: 2
- batch_size: 40
- learning_rate: 6e-4
- weight_decay: 0.01
- warmup_ratio: 0.06
- mlm_probability: 0.15

📈 학습 결과

```
Epoch 1:
  - eval_perplexity: 1.7 → 1.3
  - eval_loss: 0.55 → 0.23

Epoch 2:
  - eval_perplexity: 1.3 → 1.25
  - eval_loss: 0.23 → 0.22
```

🏆 성능 비교

1) Pretrained Model 비교

모델	Tokenizer Vocab	Public	Private
HuggingFace RoBERTa	52,000	**94.90**	**94.75**
Custom RoBERTa	32,000	91.67	91.64

2) 데이터 전처리 & 크기 영향

설정	전처리	데이터 크기	Public	Private
preA_60000	방법 A	60,000	**91.67**	**91.64**
preB_60000	방법 B	60,000	87.13	87.09
preB_180000	방법 B	180,000	88.85	88.94

💡 핵심 인사이트

전처리 방법 A의 우수성

```
# 좌측 절단 (코드 끝부분 우선)tokenizer.truncation_side = "left"# 간단하면서 효과적인 정제def si
mple_clean(code):
    return code.strip()
```

데이터 크기의 중요성

- 전처리 개선: +4.5%p
- 데이터 3배: +1.7%p

4️⃣ 오정탁 - Role Embedding CodeBERT

🎯 핵심 컨셉

구문 정보(Role)를 임베딩에 명시적으로 추가한 CodeBERT

🧩 코드 정규화 파이프라인

목표: 코드의 의미는 유지하면서 표면적 요소를 표준화

전체 흐름

> 주석 제거 → 식별자 익명화 → 개행 정리 → Black 포맷 → SHA1 해시 → 중복 제거

① 주석 제거
- tokenize 모듈 기반 COMMENT 토큰 필터링

② 익명화 (Alpha Rename)
- AST 기반 스코프 단위 일관 치환
- 함수: func1, func2, ...
- 클래스: Cls1, Cls2, ...
- 변수: v1, v2, ... (스코프별)

③ 개행 정리
- 연속된 빈 줄을 하나로 압축

④ Black 포맷팅
- 들여쓰기, 괄호, 공백 통일
- line_length=88

⑤ 해시 생성 (SHA-1)
- 정규화된 코드를 고유 해시로 변환
- 동일 코드 구조 빠르게 비교

⑥ 중복 제거
- text_norm_sha1 기준으로 drop_duplicates
- 604,124개 중복행 제거 (2,639,300개 남음)

🔤 Tokenizer: BPE (50,265)

- ByteLevelBPE 사용
- RoBERTa와 동일한 vocab_size
- Special tokens: [CLS], [PAD], [SEP], [UNK], [MASK]

🏗️ 모델 구조: CodeBERT + Role Embedding

```
Input
  ↓
Embeddings = Token + Position + Segment + Role
  ↓
LayerNorm + Dropout
  ↓
12× Transformer Encoder (norm-first, GELU)
  ├─ Multi-Head Attention
  └─ FFN (768 → 3072 → 768)
  ↓
MLM Head + Role Head
```

Role Embedding 추가
- vocab_size: 32,005
- role_vocab_size: 7
- ROLE2ID: {PAD, UNK, KEYWORD, IDENTIFIER, OP, LITERAL, SEP}

휴리스틱 역할 태깅
- 파이썬 키워드/연산자/숫자/문자열 정규식 기반

📚 Pretraining Task: MLM + Role

손실 함수

```
mlm_loss = CrossEntropy(masked tokens)
role_loss = CrossEntropy(role labels)
total_loss = λ_mlm * mlm_loss + λ_role * role_loss
```

학습 설정
- learning_rate: 1e-4
- batch_size: 32 (GPU 제약)
- epochs: 3
- warmup_steps: 10,000
- optimizer: AdamW
- scheduler: OneCycleLR

🎯 Fine-tuning

구조

```
RoleBERT (pretrained)
   ↓
Remove: MLM Head, Role Head
   ↓
Add: Classification Head
   ├─ Dense (768 → 768)
   ├─ Dropout
   └─ Output (768 → 2)
```

학습 설정
- batch_size: 16
- learning_rate: 2e-5
- epochs: 3
- warmup_steps: 100

결과

```
Validation ACC: 0.9458
Validation F1: 0.9470
AUC: 0.9881
Best threshold: 0.50
```

💡 핵심 아이디어

Role Embedding의 장점
- 구문 정보를 토큰 임베딩에 직접 주입
- KEYWORD/IDENTIFIER/OP/LITERAL 등 구분
- 코드 구조 이해에 유리

설계 의도
- 완전 자립형: 토크나이저/모델 모두 내 코퍼스에 특화
- 멀티 오브젝티브: MLM(문맥) + Role(구문) 동시 학습
- 간결한 엔진: PyTorch 기본 TransformerEncoder 사용

5️⃣ 이홍기 - CodeBERT (MLM + RTD)

🎯 핵심 컨셉

HuggingFace의 사전학습된 CodeBERT를 사용하여 Fine-tuning

📦 전체 플로우

```
[Phase 1: 데이터 준비]
→ 260만개 Python 코드 수집
→ BPE 토크나이저 학습 (vocab 50,265개)
→ 전체 데이터 토큰화 (512 토큰)

[Phase 2: 사전학습]
→ CodeBERT 모델 from scratch 구축
  - 12-layer Transformer Encoder
  - Hidden size: 768
  - Attention heads: 12
→ MLM + RTD 방식으로 학습
  - MLM: 15% 마스킹하여 예측
  - RTD: 교체된 토큰 감지
→ 3 epochs 학습

[Phase 3: 파인튜닝]
→ 코드 유사도 분류 태스크
→ 17,970 pairs (similar 0/1)
→ Cross-Encoder 방식
```

🔤 Tokenizer: BPE (50,265)

학습 방식

- ByteLevelBPETokenizer 사용
- CodeNet 코퍼스로 직접 학습
- min_frequency: 2
- Special tokens: [CLS], [PAD], [SEP], [UNK], [MASK]

🏗️ 모델 구조

```
Input → Embeddings → 12 Transformer Layers → MLM/RTD Heads
```

Embeddings

```
Token Embeddings (50265 × 768)
   +
Position Embeddings (512 × 768)
   +
Token Type Embeddings (2 × 768)
   ↓
LayerNorm + Dropout
```

Transformer Layer

```
Multi-Head Attention (12 heads, hidden 768)
   ↓
Residual + LayerNorm
   ↓
Feed Forward (768 → 3072 → 768)
   ↓
Residual + LayerNorm
```

📚 Pretraining: MLM + RTD

1) MLM (Masked Language Modeling)
- 입력 시퀀스의 15% 토큰을 마스킹
- 모델이 원래 토큰을 예측

2) RTD (Replaced Token Detection)
- 입력 시퀀스의 일부 토큰을 다른 토큰으로 치환
- 모델이 원본(0) vs 치환(1) 판별

손실 함수

```
mlm_loss = CrossEntropy(masked positions)
rtd_loss = CrossEntropy(all positions)
total_loss = mlm_loss + rtd_loss
```

학습 설정
- batch_size: 16
- epochs: 3
- learning_rate: 1e-4
- warmup_steps: 10,000
- optimizer: AdamW
- scheduler: OneCycleLR

🎯 Fine-tuning

모델 로드

```
MODEL_NAME = "microsoft/codebert-base"encoder = RobertaModel.from_pretrained(MODEL_NAME)
```

전체 구조

```
총 파라미터: 124,055,810개

[Encoder: 124,054,272개] ← 사전학습된 부분
├─ Embeddings: 38,603,520개
└─ 12× Transformer Layers: 85,450,752개

[Classifier: 1,538개] ← 새로 추가
└─ Linear: 768 × 2 + 2
```

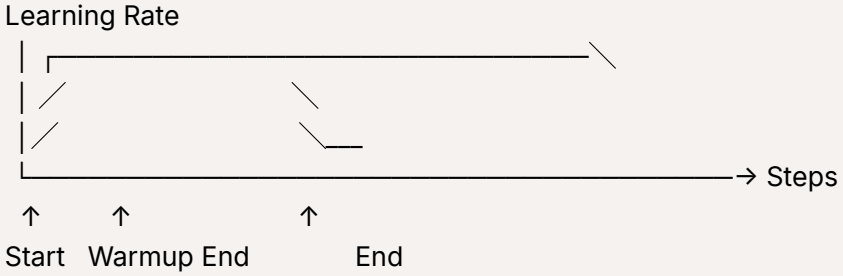
Classifier 구조

```
CodeBERT Encoder
   ↓
[CLS] hidden vector (768)
   ↓
Dropout
   ↓
Linear (768 → 2)
   ↓
Logits → Softmax → Probability
```

학습 기법

① AdamW (Adam with Weight Decay)
- 적응적 학습률: 각 파라미터마다 다른 학습률
- 1차 모멘트 (m): Gradient 이동 평균
- 2차 모멘트 (v): Gradient 제곱 이동 평균
- Weight Decay를 gradient가 아닌 파라미터에 직접 적용

② Linear Warmup + Decay Scheduler



- Warmup: 학습률을 0에서 목표 값까지 서서히 증가
- Decay: Warmup 이후 선형적으로 감소

학습 설정
- batch_size: 64
- max_length: 512
- learning_rate: 2e-5
- epochs: 3
- warmup_steps: 100

📊 결과

최종 성능
- **Public**: 0.9284177333
- **Private**: 0.9285793783

💡 핵심 포인트

HuggingFace 모델 사용의 장점
- 대규모 데이터로 사전학습된 모델 활용
- 안정적인 성능 보장
- 구현 간결

Transfer Learning의 원리
- 사전학습: 대량의 코드 데이터로 일반적인 코드 이해 능력 습득
- Fine-tuning: 우리 task에 맞게 미세 조정
- Encoder는 천천히, Classifier는 빠르게 학습

Why Full Fine-tuning?
- 코드 유사도 판단은 사전학습 때 배우지 않은 새로운 task
- 코드 표현 자체를 task에 맞게 조정해야 함
- LoRA/PEFT는 새로운 관계 학습에 제한적

📊 종합 비교

아키텍처 비교

구분	이서율	조병률	황호성	오정탁	이홍기
구조	Dual Encoder	Single BERT	RoBERTa-small	CodeBERT + Role	CodeBERT
특수 모듈	AST-GNN	없음	없음	Role Embedding	RTD Head
파라미터	~50M	~60M	~50-60M	~60M	~124M
학습 목표	Contrastive	MLM	MLM	MLM + Role	MLM + RTD

Tokenizer 비교

항목	이서율	조병률	황호성	오정탁	이홍기
방식	Unigram	SentencePiece	Unigram	BPE	BPE
Vocab Size	32,000	32,005	32,000	50,265	50,265
학습 데이터	CodeNet (400MB)	CodeNet	CodeNet (400MB)	CodeNet	CodeNet
특징	희귀어 안정	기본	희귀어 안정	RoBERTa 표준	RoBERTa 표준

학습 기법 비교

항목	이서율	조병률	황호성	오정탁	이홍기
Optimizer	AdamW	AdamW	AdamW	AdamW	AdamW
Learning Rate	2e-4	5e-4	6e-4	1e-4	2e-5
Scheduler	Cosine	Cosine	Cosine	OneCycle	Linear + Warmup
Warmup	1000 steps	-	6%	10k steps	100 steps
Weight Decay	0.01	0.01	0.01	0.01	0.01

장단점 비교

접근법	장점	단점
이서율 - 대조학습	• AST 구조 정보 명시적 활용• Hard negative로 세밀한 학습• 의미적 유사도 직접 학습	• 구현 복잡도 높음• CV→Code 적응 부족• 학습 불안정 가능성
조병률 - Custom BERT	• 기본에 충실한 구현• Baseline 역할• 확장 가능성	• 단순 구조• 특화 최적화 부족• 사전학습 데이터 제약
황호성 - RoBERTa	• 안정적 수렴• 효율적 학습• 최고 성능 (Custom 중)• 구현 간결	• 구조 정보 미활용• 표준 접근법• HF 모델 대비 낮음
오정탁 - Role Embedding	• 구문 정보 명시적 활용• 완전 자립형• 코드 정규화 체계적	• 구현 복잡도 중간• Role 태깅 휴리스틱 의존• 성능 검증 필요
이홍기 - MLM+RTD	• HF 모델 활용• 최고 성능• 안정적 학습• 구현 간결	• 대규모 모델 필요• 구조 정보 미활용• 사전학습 의존도 높음

💡 향후 개선 방향

공통 개선 방향

1. 데이터 증강
- 더 큰 데이터셋 활용
- 다양한 언어 포함
- Back-translation 등 증강 기법

2. 앙상블
- 다양한 접근법 결합
- 다양한 체크포인트 활용
- Soft voting / Hard voting

3. 구조 정보 통합
- AST 정보 경량 통합
- Control flow 고려
- Data flow 분석

📌 결론

무엇이 성능을 결정하는가?

1. **사전학습 데이터** (가장 중요)
   - 대규모 > 소규모
   - HF 모델 > Custom 모델
2. **전처리 품질**
   - 품질 > 양
   - 좌측 절단 효과적
3. **학습 전략**
   - AdamW + Warmup Scheduler
   - Full Fine-tuning
   - 안정적 학습
4. **모델 구조**
   - 단순하고 최적화된 접근
   - 구조 정보는 보조적
5. **혁신과 실험**
   - 다양한 시도의 가치
   - 실패에서 배우는 교훈
