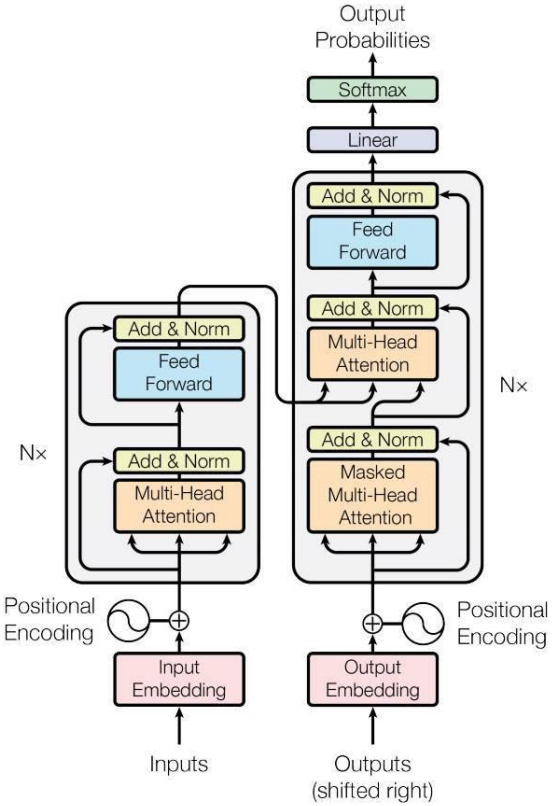
UniXcoder: Unified Cross-Modal Pre-training for Code Representation, ACL 2022

Background

- History: applications of the **transformer** in understanding and generation of **texts, images, audios, videos**
- This paper: a pre-trained transformer model for the understanding and generation of **codes**

Some previous work on code understanding & generation:

- CuBERT: pretraining a BERT contextual embedding on source code
- GPT-C: left-to-right (unidirectional) transformer for code generation
- CodeBert: bimodal pretraining on codes and comments



Related work

1. Encoder-only model: bidirectional, good for learning from context, but requires an additional decoder for generation (bad).
    a. E.g. CuBERT, CodeBERT, GraphCodeBERT, SYN-CoBERT.
2. Decoder-only model: unidirectional, good for augo-regressive tasks, but bad for understanding tasks (since understanding requires bidirectional contexts).
    a. E.g. GPT-C, CodeGPT.
3. Encoder-Decoder model: supports both understanding and generation tasks.
    a. PLBART, BART, CodeT5, TreeBERT

For UniXcoder, it uses kind of an encoder, utilizing mask attention matrices with prefix adapters to control the behavior of the model. (so it supports all of the above 3 modes)

Example related work: CodeBERT

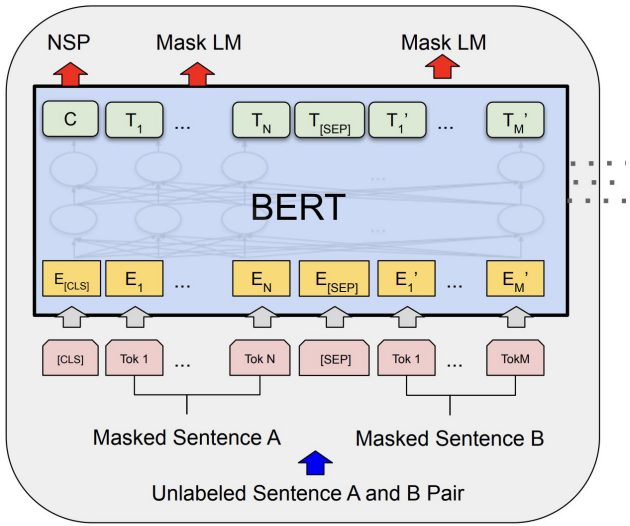
Model architecture: same as BERT

Input: **[CLS] W [SEP] C [EOS]** *(W: NL comments. C: codes)*

Intuition: learns the association between NL and PL

```
def _parse_memory(s):
    """
    Parse a memory string in the format supported by Java (e.g. 1g, 200m) and
    return the value in MiB

    >>> _parse_memory("256m")
    256
    >>> _parse_memory("2g")
    2048
    """
    units = {'g': 1024, 'm': 1, 't': 1 << 20, 'k': 1.0 / 1024}
    if s[-1].lower() not in units:
        raise ValueError("invalid format: " + s)
    return int(float(s[:-1]) * units[s[-1].lower()])
```



Pre-training

Beyond the BERT model: UniXcoder

Major contributions:

1. Proposes a cross-modal pretrained model for code understanding and generation, **with one "encoder" accomplishing 3 modes**
2. Proposes a **one-to-one mapping function**: AST -> sequence
3. Constructs a **new dataset**: zero-shot code-code search

1. Input representation

Combining the following two to provide a rich representation of the codes:

1. **Comment**: a high level description of the codes' functionality
2. **AST**: provides rich syntax information (e.g. parameters -> (data) tells us data is a parameter), superior to raw codes.

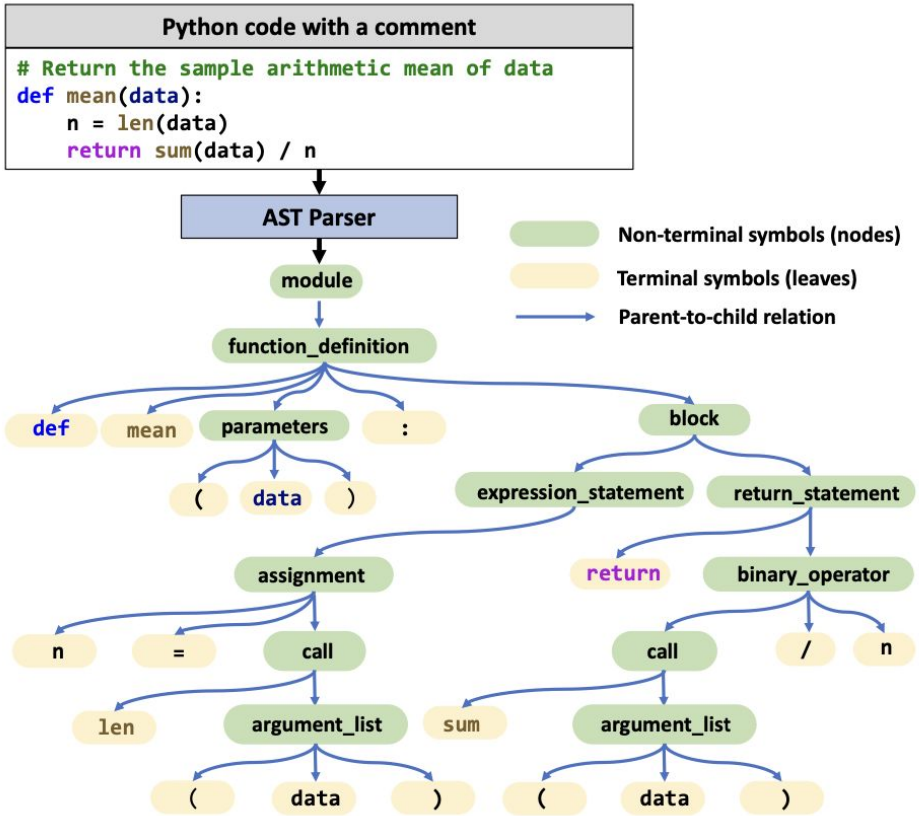


Figure 1: A Python code with its comment and AST.

2. Model architecture

- Input: prefix || comment || flattened AST
  - Prefix: {[Enc], [Dec], [E2D]}
- N transformer layers, multi-headed self-attention (but not bidirectional! Because of the mask matrix).
- Mask matrix M: controls the context a token can attend to, kind of an “encoder-decoder” switch
- At the l-th layer:

$$Q = H^{l-1}W^{Q}, K = H^{l-1}W^{K}, V = H^{l-1}W^{V} \tag{1}$$

$$head = \mathrm{softmax}(\frac{\mathrm{QK^T}}{\sqrt{\mathrm{d_k}}} + \mathrm{M})\mathrm{V} \tag{2}$$

QK^TV:

$$\begin{bmatrix} — & q_1 & — \\ — & q_2 & — \\ & \vdots & \\ — & q_n & — \end{bmatrix} \begin{bmatrix} | & | & & | \\ k_1 & k_2 & \cdots & k_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} — & v_1 & — \\ — & v_2 & — \\ & \vdots & \\ — & v_n & — \end{bmatrix}$$

Q $n \times d_k$ — K^T $d_k \times n$ — V $n \times d_v$

$$= \begin{bmatrix} q_1^T k_1 & q_1^T k_2 & q_1^T k_3 & \cdots \\ q_2^T k_1 & q_2^T k_2 & q_2^T k_3 & \cdots \\ \vdots & \vdots & \vdots & \end{bmatrix} \begin{bmatrix} — & v_1 & — \\ — & v_2 & — \\ & \vdots & \\ — & v_n & — \end{bmatrix}$$

QK^T $n \times n$ — V $n \times d_v$

$$= \begin{bmatrix} \sum (q_1^T k_i) v_i \\ \sum (q_2^T k_i) v_i \\ \vdots \\ \sum (q_n^T k_i) v_i \end{bmatrix}$$

$(QK^T)V$ $n \times d_v$

☆ Setting $M_{ij} = -\infty$

⇔ setting ith token's query on the jth token to be $-\infty$ (i.e. masked out)

☆ M allows us to precisely mask out any token pair.

2. Model architecture

- At the l-th layer:
- M_{ij} :
  - Encoder mode: All elements set to 0
  - Decoder mode: upper triangular part of M is -inf, others 0
  - Encoder-Decoder mode: source input is bidirectional, target input is unidirectional

$$Q = H^{l-1}W^Q, K = H^{l-1}W^K, V = H^{l-1}W^V \quad (1)$$

$$head = \text{softmax}(\frac{\text{QK}^\text{T}}{\sqrt{\text{d}_\text{k}}} + \text{M})\text{V} \quad (2)$$

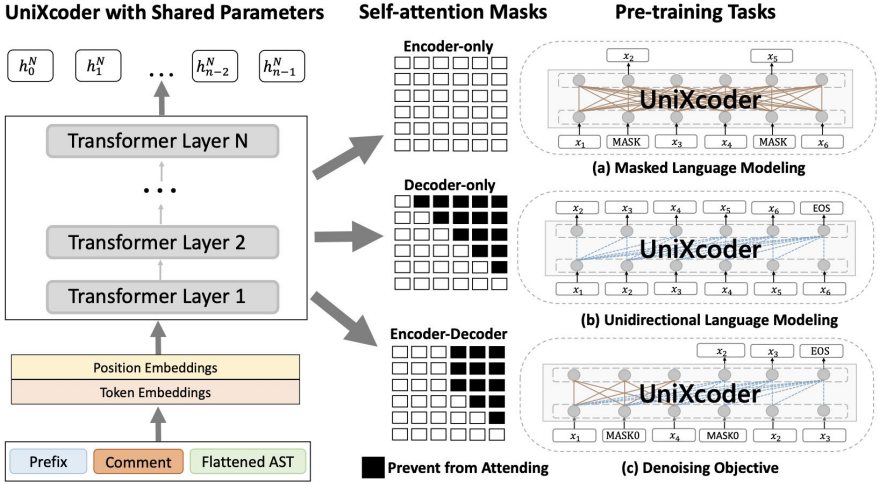


Figure 2: Model architecture of UniXcoder. The model takes comment and flattened AST as the input (more specific input examples can be found in Figure 3). Model parameters are shared in different modes. We use different self-attention masks to control the behavior of the model and use various tasks to pre-train the model including masked language modeling, unidirectional language modeling, and denoising objective.

3. Pre-Training objectives

1. Masked Language Modeling (MLM): for the encoder-only mode
   a. Sample 15% of the tokens to "mask", of which replacing 80% with [MASK], 10% with random words, 10% unchanged. (forces the model to keep a distributional contextual representation for **every input token**)
   b. Leverages semantic information from AST to infer masked code tokens, **encouraging the model to learn code representations**
   c. Training objective:

$$loss_{MLM} = -\sum_{x_i \in S_m} logp(x_i | X^{mask}) \qquad (3)$$

      i. X^{mask}: the masked input sequence.
      ii. I.e. to max the prob. of masked tokens, given the input sequence (masked)
   d. Intuition: so the model learns to understand codes

3. Pre-Training objectives

2. Unidirectional Language Modeling (ULM): for the decoder-only mode

a. To support auto-regressive tasks: predict the next token one by one, conditioned on previous tokens and the current token itself
b. Uses an upper triangular matrix for M
c. Training objective:
    i. I.e. to max the prob. of next token, given the generated ones
d. Intuition: so the model learns to generate codes

$$loss_{ULM} = -\sum_{i=0}^{n-1} logp(x_i|x_{t<i}) \qquad (4)$$

3. Pre-Training objectives

3. Denoising Objective (DNS): for the encoder-decoder mode

a. Randomly masks spans with arbitrary lengths and then generates these masked spans
b. Training objective:
    i. I.e. to max the prob. of generated span of code, given the masked inputs
c. Intuition: so the model learns to generate codes based on its understanding of the context

$$loss_{DNS} = -\sum_{i=0}^{n-1} logp(y_i|X^{mask}, y_{t<i}) \quad (5)$$

3. Pre-Training objectives

$$loss_{MCL} = -\sum_{i=0}^{b-1} log \frac{e^{cos(\tilde{h}_i, \tilde{h}_i^+)/\tau}}{\sum_{j=0}^{b-1} e^{cos(\tilde{h}_i, \tilde{h}_j^+)/\tau}} \quad (6)$$

$$loss_{CMG} = -\sum_{i=0}^{m-1} logp(w_i | X, w_{t<i}) \quad (7)$$

4. Code Fragment Representation Learning

a. To learn semantic embedding of a code segment
b. 2 pre-training tasks:
    i. Multi-modal contrastive learning (MCL): encourages augmentations of the same input to have more similar representations.
    ii. Cross-modal Generation (CMG): ask the model to generate its own comments
c. Intuition: so the model learns similarity of code segments and is able to understand code segments (by generating comments)

3. Pre-Training objectives: summary

1. Masked Language Modeling (MLM): for the encoder-only mode: **learns to understand codes**
2. Unidirectional Language Modeling (ULM): for the decoder-only mode: **learns to generate codes**
3. Denoising Objective (DNS): for the encoder-decoder mode: **learns to generate codes based on its understanding of the context**
4. Code Fragment Representation Learning: **learns to understand codes at a finer-grain level**

4.1. Experiment: Understanding tasks

- **Clone detection** (are these two pieces of code similar?) : to measure the similarity between two code segments. (2 datasets)
  a. POJ-104: given two codes, to predict whether they have the same semantics
  b. BigCloneBench: given a query code, to retrieve semantically similar codes
- **Code search** (find the desired codes from a collection of codes): to find the most relevant code from a collection of candidates given a natural language query. (3 datasets)
  a. CSN: from CodeSearchNet, 6 programming languages.
  b. AdvTest: normalizes python function and variable names
  c. CosQA: based on CodeSearchNet, queries from search logs of Bing search engine

The intuition:

- **Clone detection**: is the model able to **tell similarity of codes?** (during training, the Multi-modal Contrastive Learning objective encourages the model to tell similarity of codes)
- **Code search**: if the model able to **translate natural language words into an understanding of codes?** (during training, the model was encouraged to relate NL with PL)

4.1. Experiment: Understanding tasks: results

Ablation: removal of a component of an AI system

Summary: UniXcoder achieves promising performance in understanding codes

Note: improvement mainly comes from **contrastive learning** and the **use of multi-modality**.

Model	Clone Detection				Code Search		
	POJ-104	BigCloneBench			CosQA	AdvTest	CSN
	MAP@R	Recall	Precision	F1-score	MRR		
RoBERTa	76.67	**95.1**	87.8	91.3	60.3	18.3	61.7
CodeBERT	82.67	94.7	93.4	94.1	65.7	27.2	69.3
GraphCodeBERT	85.16	94.8	95.2	95.0	68.4	35.2	71.3
SYNCOBERT	88.24	-	-	-	-	38.3	74.0
PLBART	86.27	94.8	92.5	93.6	65.0	34.7	68.5
CodeT5-base	88.65	94.8	94.7	95.0	67.8	39.3	71.5
UniXcoder	**90.52**	92.9	**97.6**	**95.2**	**70.1**	**41.3**	**74.4**
-w/o contras	87.83	94.9	94.9	94.9	69.2	40.8	73.6
-w/o cross-gen	90.51	94.8	95.6	95.2	69.4	40.1	74.0
-w/o comment	87.05	93.6	96.2	94.9	67.9	40.7	72.6
-w/o AST	88.74	92.9	97.2	95.0	68.7	40.3	74.2
-using BFS	89.44	93.4	96.7	95.0	69.3	40.1	74.1
-using DFS	89.74	94.7	94.6	94.7	69.0	40.2	74.2

Table 1: Results on understanding tasks. **contras** is contrastive learning, **cross-gen** indicates cross-modal generation, and **BFS** (**DFS**) means that our mapping function is replaced by breath-first (deep-first) search algorithm.

4.2. Experiment: Generation tasks

1. **Code summarization** (to summarize the codes) : to generate an NL summary of a code snippet
   a. CodeXGLUE: “General Language Understanding Evaluation benchmark for CODE”
2. **Code Generation** (to generate codes from words): to generate a code snippet based on an NL description. (3 datasets)
   a. CONCODE: input consists of an NL description and code environments

The intuition:

3. **Code summarization**: is the model able to **generate human language from codes?** (during training, the comment generation objective encourages the model to summarize codes in NL)
4. **Code Generation**: is the model able to **generate codes from human language?** (during training, the model learns to associate NL with PL, and learned to generate codes)

4.2. Experiment: Generation tasks: results

Comparable to/slightly worse than CodeT5-base, reasons:

1. CodeT5's training data is not public (not fair for comparison)
2. CodeT5 has much larger model size

Summary: the performance of UniXcoder in terms of generating codes is still comparable to the state-of-art

Note: incorporating **comments** helped with generation the most, but **AST** brings down the performance (unlike code understanding)

Model	Summarization	Generation	
	BLEU-4	EM	BLEU-4
RoBERTa	16.57	-	-
CodeBERT	17.83	-	-
GPT-2	-	17.35	25.37
CodeGPT	-	20.10	32.79
PLBART	18.32	18.75	36.69
CodeT5-small	19.14	21.55	38.13
CodeT5-base	**19.55**	22.30	**40.73**
UniXcoder	19.30	**22.60**	38.23
-w/o contras	19.20	22.10	37.69
-w/o cross-gen	19.27	22.20	35.93
-w/o comment	18.97	21.45	37.15
-w/o AST	19.33	22.60	38.52
-using BFS	19.24	21.75	38.21
-using DFS	19.25	22.10	38.06

Table 2: Results on two generation tasks, including code summarization and code generation.

4.3. Experiment: Code completion

1. **Code completion** (fill in missing codes) : to complete the unfinished line given previous context

The intuition:

2. **Code completion**: is the model able to **generate codes at a finer-grain level, from context?**

The result: significant improvement to the previous models

Model	PY150		JavaCorpus	
	EM	Edit Sim	EM	Edit Sim
Transformer	38.51	69.01	17.00	50.23
GPT-2	41.73	70.60	27.50	60.36
CodeGPT	42.37	71.59	30.60	63.45
PLBART	38.01	68.46	26.97	61.59
CodeT5-base	36.97	67.12	24.80	58.31
UniXcoder	**43.12**	**72.00**	**32.90**	**65.78**
-w/o contras	43.02	71.94	32.77	65.71
-w/o cross-gen	42.66	71.83	32.43	65.63
-w/o comment	42.18	71.70	32.20	65.44
-w/o AST	42.56	71.87	32.63	65.66
-using BFS	42.83	71.85	32.40	65.55
-using DFS	42.61	71.97	32.87	65.75

Table 3: Results of code completion task.

4.4. Experiment: Zero-shot code-to-code search

Goal: to retrieve codes with the same semantics from a collection of candidates, in a zero-shot setting

Intuition: to further evaluate the performance of code fragment embeddings (if the code embedding works well, then codes of similar functionality should have similar embedding vectors)

Note: significant improvements to the state-of-art, much better performance in terms of understanding code segment semantics (e.g. given two pieces of code, maybe we can figure out if they are working on the same functionality, for example, fixing and introducing bugs)

Model	Ruby			Python			Java			Overall
	Ruby	Python	Java	Ruby	Python	Java	Ruby	Python	Java	
CodeBERT	13.55	3.18	0.71	3.12	14.39	0.96	0.55	0.42	7.62	4.94
GraphCodeBERT	17.01	9.29	6.38	5.01	19.34	6.92	1.77	3.50	13.31	9.17
PLBART	18.60	10.76	1.90	8.27	19.55	1.98	1.47	1.27	10.41	8.25
CodeT5-base	18.22	10.02	1.81	8.74	17.83	1.58	1.13	0.81	10.18	7.81
UniXcoder	**29.05**	**26.36**	**15.16**	**23.96**	**30.15**	**15.07**	**13.61**	**14.53**	**16.12**	**20.45**
-w/o contras	24.03	17.35	7.12	15.80	22.52	7.31	7.55	7.98	13.92	13.73
-w/o cross-gen	28.73	24.16	12.92	21.52	26.66	12.60	11.14	10.82	13.75	18.03
-w/o comment	22.24	15.90	7.50	15.09	19.88	6.54	7.84	7.12	13.20	12.81
-w/o AST	27.54	23.37	10.17	21.75	27.75	9.94	9.79	9.21	14.06	17.06
-using BFS	26.67	23.69	13.56	21.31	27.28	13.63	11.90	12.55	14.92	18.39
-using DFS	27.13	22.65	11.62	20.21	25.92	11.85	9.59	10.19	13.30	16.94

Table 4: MAP score (%) of zero-shot setting on code-to-code search task.

5. Analysis

1. Ablation study shows the two pre-training tasks are effective
2. The multi-modal data are very effective for both understanding and generation, especially the comments (intuition: comments are most helpful for associating NL with PL)
3. The algorithm on AST mapping outperforms both BFS and DFS in sequelizing the AST (since the latter 2 are not 1-to-1, and can confuse one tree with another)

pre-trained models: 3 are provided:

unixcoder-base-unimodal: Pre-trained on C4 and CodeSearchNet dataset (without NL)

unixcoder-base: Continue pre-training `unixcoder-base-unimodal` on NL-PL pairs of CodeSearchNet dataset. The model can support six languages: **java, ruby, python, php, javascript, and go**. This model is reported in the paper.

unixcoder-base-nine: Continue pre-training `unixcoder-base-unimodal` on NL-PL pairs of CodeSearchNet dataset and additional 1.5M NL-PL pairs of C, C++ and C# programming language. The model can support nine languages: **java, ruby, python, php, javascript, go, c, c++ and c#**.

Thanks!