Ժամանակակից կրթական հարթակներում ավտոմատ գնահատման համակարգերը հիմնականում հիմնված են ծրագրի ելքային արդյունքի ստուգման վրա, այսինքն՝ ստուգում են՝ արդյոք ուսանողի ծրագիրը վերադարձնում է ճիշտ արդյունքը: Սակայն այս մոտեցումը չունի իմաստային խորություն. այն չի հասկանում, թե արդյոք ուսանողը իրականում կատարել է առաջադրանքի պահանջը այն եղանակով, որը նկարագրված է առաջադրանքի տեքստում: Օրինակ՝ երկու ուսանող կարող են ստանալ նույն թվային արդյունքը, բայց մեկի լուծումը կարող է լիովին հակասել առաջադրանքի տրամաբանությանը, իսկ մյուսինը՝ լիովին անհակասական լինել:

Այս խնդիրը լուծելու համար անհրաժեշտ է համակարգ, որը կարող է հասկանալ ոչ միայն ծրագրի արդյունքը, այլև առաջադրանքի իմաստը: Այն պետք է կարողանա լեզվական տեքստից դուրս բերել requirement-ներ, համադրել դրանք hint-երի հետ, ստուգել semantic consistency-ն և համադրել ուսանողի code behavior-ի հետ: Այս նպատակով կիրառվում են արհեստական բանականության և բնական լեզվի մշակման համակցված մոդելներ՝ **SpanBERT**, **MPNet**, **DeBERTa**, **Graph Neural Network (GGNN)** և **constraint-based test generation**, որոնք միասին ձևավորում են իմաստային վերլուծության ենթահամակարգ՝ գնահատման գործընթացը դարձնելով բացատրելի, ճշգրիտ և ուսուցողական:

Առաջին փուլում իրականացվում է առաջադրանքի իմաստային պահանջների դուրսբերում՝ կիրառելով **SpanBERT** մոդելը (Joshi et al., *SpanBERT: Improving Pre-training by Representing and Predicting Spans*, arXiv:1907.10529, [1907.10529] SpanBERT: Improving Pre-training by Representing and Predicting Spans):

SpanBERT մոդելը առաջարկվել է որպես BERT-ի խորացված տարբերակ, որը կարողանում է ներկայացնել ոչ միայն առանձին token-ների իմաստը, այլ նաև դրանց ամբողջական span-ային կառուցվածքը: Այս մոդելի հիմնական նպատակը այն է, որ մոդելը կարողանա հասկանալ արտահայտությունները որպես միասնական իմաստային միավոր, այլ ոչ թե միայն token-ների հաջորդականություն: Դա իրականացվում է span-wise masking-ի և span boundary objective-ի միջոցով:

Ենթադրենք, որ տրված է նախադասություն կամ որևէ տեքստային հատված, որը tokenized է.

$$X=(x_1,x_2,\ldots,x_n)$$

որտեղ x_i — i-րդ token-ն է, իսկ n ՝ ամբողջ հաջորդականության երկարությունը: SpanBERT-ը աշխատում է ոչ թե ամբողջ հաջորդականության կամ անհատական token-ների, այլ՝ span-ների հետ:

$$(x_s,\ldots,x_e), \qquad 1 \le s \le e \le n$$

Այս span-ը (այսինքն՝ ենթաշարքը) հանդիսանում է ուսուցման բազային միավորը. այսինքն՝ տվյալ span-ը մոդելը պետք է ամբողջությամբ հասկանա:

.

3.1 Span-ի երկարության ընտրություն

Ի տարբերություն BERT-ի, որը պատահական token է փոխարինում, SpanBERT-ը փոխարինում է ամբողջ span-ը: Span-ի երկարությունը՝

$$L = e - s + 1$$

ընտրվում է geometric բաշխումից.

$$P(L = k) = (1 - p)^{k-1} p, \quad k = 1, 2, \ldots$$

Այս բաշխումը ապահովում է, որ ավելի հաճախ ընտրվեն կարճ span-ներ (օր.՝ 2–5 token), սակայն երբեմն կարող են լինել մինչև Lmax=10 token-անոց span-ներ:

Միջին արժեքը.

$$\mathbb{E}[L] = \frac{1}{p} = 5$$

Այսպիսով, մոդելը սովորում է աշխատել տարբեր չափերի span-ների հետ, ինչը օգնում է հասկանալ և՛ բառային խմբերը, և՛ ավելի երկար արտահայտությունները:

3.2 Span-wise Masking

Եթե ընտրվում է span (s,e), ապա մուտքային հաջորդականությունը X դառնում է մասնավորված հաջորդականություն՝

$$X' = \text{mask}(X, s, e)$$

$$X' = M(X) = (x'_1, x'_2, \ldots, x'_n)$$

որտեղ

$$x'_i = \begin{cases} [MASK], & \text{եթե } i \in [s, e] \text{ որոշ span-ի համար} \\ x_i, & \text{այլ դեպքերում} \end{cases}$$

այսինքն՝ span-ի բոլոր *token*-ները միաժամանակ փոխարինվում են [MASK] կամ պատահական token-ներով:

Սա նշանակում է, որ մոդելը չի կարող տեսնել span-ի ներքին բառերը և ստիպված է վերականգնել դրանք ամբողջական համատեքստից:

4. Encoder-ի իմաստային ներկայացումները

SpanBERT-ի հիմքում BERT-ի bidirectional Transformer encoder-ն է: Այս encoder-ը յուրաքանչյուր token-ի համար հաշվում է context-dependent embedding՝

$$h_i \in \mathbb{R}^d$$

և ամբողջ հաջորդականության համար՝

$$H = (h_1, h_2, \ldots, h_n)$$

Այս embeddings-ը համարվում են span-ի իմաստային վերլուծության համար հենակետային տեղեկատվությունը:

5. MLM կորուստը SpanBERT-ում

SpanBERT-ը պահպանում է տիպիկ Masked Language Modeling մեխանիզմը.

$$\mathcal{L}_{MLM} = -\sum_{i \in M} \log P(x_i | H)$$

որտեղ MM-ը span-wise մասֆավորված token-ների բազմությունն է:

Սա ապահովում է, որ մոդելը սովորի token-ի համատեքստային ներկայացումը, սակայն հիմնին span-ի ընդհանուր իմաստը հասկանալու համար այն բավարար չէ, այդ պատճառով ավելացվում է SBO-ն:

6. Span Boundary Objective (SBO)

MLM-ը վերականգնում է token-ները, բայց span-ի կառուցվածքային իմաստը մոդելը լավ չի սովորում միայն MLM-ով: SBO-ի հիմնական գաղափարն այն է, որ span-ի ներքին token-ները պետք է վերականգնվեն միայն boundary embedding-ների՝

$$h_{s-1}, \quad h_{e+1}$$

օգնությամբ:

Սա մոդելին սովորեցնում է span-ը ընկալել որպես ամբողջական իմաստային միավոր:

Հարաբերական դիրքերը **span**-ի ներսում

Span-ի յուրաքանչյուր token x_i, որտեղ $s \le i \le e$, ունի հարաբերական դիրք՝

$$p_i = i - s$$

և դրան համապատասխան embedding՝

$$r_i = E(p_i)$$

Սա ասում է մոդելին՝ token-ը span-ի որ մասում է գտնվում:

Ներքին **token**-ի ներկայացում **boundary**-ից

Յուրաքանչյուր մասնավորված token (x_i) վերականգնվում է՝

$$\tilde{h}_i = f_{FFN}\left([h_{s-1}; h_{e+1}; r_i]\right)$$

որտեղ f_{FFN}ը երկշերտ feed-forward ցանց է GeLU ակտիվացմամբ:

Այս ներկայացումը արդեն այստեղ է պահում span-ի ամբողջական իմաստը. ձախ+աջ սահման + token-ի դիրք:

6.5 SBO կորուստի ֆունկցիա

Կորուստը հաշվարկվում է softmax կանխատեսման հիման վրա.

$$\mathcal{L}_{SBO} = -\sum_{i \in M} \log P(x_i \mid \tilde{h}_i)$$

որտեղ $P(x_i \mid \tilde{h}_i)$ հաշվարկվում է՝

$$P(x_i \mid \tilde{h}_i) = \frac{\exp(w_{x_i}^\top \tilde{h}_i)}{\sum_{v \in V} \exp(w_v^\top \tilde{h}_i)}$$

V — բառարանի չափն է, իսկ w_v — vocab embedding matrix-ի v-րդ տողը:

7. Ընդհանուր **Loss** ֆունկցիա

SpanBERT-ի ուսուցումը հիմնված է երկու կորուստների համադրության վրա՝

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{MLM} + \mathcal{L}_{SBO}$$

Սա ապահովում է երկակի օպտիմալացում.

- MLM-ը սովորեցնում է token-ի համատեքստային իմաստը,
- SBO-ն սովորեցնում է span-ի կառուցվածքային ամբողջությունը:

8. Single-Sequence Training մեխանիզմը

SpanBERT-ը սովորում է մեկ շարունակական տեքստի վրա՝ առանց նախադասությունների բաժանումների:

Սա կարևոր է, որովհետև իրական span-ները միշտ չէ, որ համընկնում են նախադասության սահմանի հետ:

Բերված բանաձևային համակարգը ներկայացնում է **SpanBERT**-ի **span**-ների խմաստային ներկայացման մեխանիզմը, որը մաթեմատիկորեն ձևակերպվում է որպես ֆունկցիոնալ կոմպոզիցիա՝

$$F : \mathcal{X}^n \to \mathbb{R}^d, \quad F(X) = H = (h_1, \ldots, h_n)$$

Այստեղ՝

- $X=(x_1,x_2,\ldots,x_n)$— մուտքային հաջորդականություն է (tokenized input sequence),
- F(X) — Transformer encoder-ի միջոցով ստացված վեկտորային ներկայացում է,
- $h_i \in R^d$ — յուրաքանչյուր token-ի բաշխված ներկայացում (hidden state) է embedding տարածությունում:

Այնուհետև span-ի ներկայացումը սահմանվում է՝

$$R_{s,e} = g(h_{s-1}, h_{e+1})$$

որտեղ՝

- s և e — span-ի սկիզբն ու ավարտն են,
- h_{s-1} և h_{e+1} — համապատասխանաբար span-ի ձախ և աջ սահմանային վեկտորներն են,
- $g(\cdot)$— non-linear կոմպոզիցիոն ֆունկցիա է (օր.՝ feed-forward կամ bilinear pooling), որը միավորում է սահմանային case-երը մեկ համակցված ներկայացման մեջ:

Այս մեխանիզմը հայտնի է որպես **boundary pooling**, քանի որ span-ի խմաստային վեկտորը ստացվում է ոչ թե ներքին token-ների միջինով կամ գումարով, այլ դրա եզրային case-երի համադրությամբ:

Ենթադրենք նախադասությունը.

“The quick brown fox jumps over the dog”

SpanBERT-ը այս նախադասության վրա կատարում է span-wise masking և span reconstruction:

$$X = (x_1 = \text{The}, x_2 = \text{quick}, x_3 = \text{brown}, x_4 = \text{fox}, x_5 = \text{jumps}, x_6 = \text{over}, x_7 = \text{the}, x_8 = \text{dog})$$

Tokenization և մուտքային հաջորդականության կառուցում

Նախադասությունն օրենքի համաձայն բաժանվում է token-ների.

$$X = (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$$

Որտեղ՝

$x_1 =$ “The”
$x_2 =$ “quick”
$x_3 =$ “brown”
$x_4 =$ “fox”
$x_5 =$ “jumps”
$x_6 =$ “over”
$x_7 =$ “the”
$x_8 =$ “dog”

Այս հաջորդականությունը հանդիսանում է SpanBERT-ի encoder-ի մուտքը:

Span-ի ընտրություն և նրա կառուցվածքը

Ընտրված span-ը.

$$(s, e) = (2, 4)$$

այսինքն՝ span-ը պարունակում է token-ները՝

“quick brown fox”

Սա 3 token երկարությամբ span է.

$$L = e - s + 1 = 4 - 2 + 1 = 3$$

Այս span-ն ընտրված է geometric բաշխման հիման վրա, որտեղ p=0.2, այսինքն՝ կարճ span-ները ավելի հավանական են:

Span-wise Masking — ամբողջ **span-**ի միաժամանակյա թաքցում

SpanBERT-ը մասֆավորում է ոչ թե մեկ token, այլ ամբողջ span-ը:
Այսպիսով, մուտքային հաջորդականությունը դառնում է.

```
The  [MASK]  [MASK]  [MASK]  jumps  over  the  dog
```

Սա SpanBERT-ի շատ կարևոր առանձնահատկություններ է.

մոդելը span-ը ամբողջությամբ չի տեսնում և պետք է այն վերականգնի *span*-ի սահմաններից:

Encoder-ի context embeddings

Encoder-ը ամբողջ նախադասության համար հաշվում է contextual embeddings.

$$H = (h_1, h_2, \ldots, h_8)$$

Յուրաքանչյուր $h_i \in \mathbb{R}^d$ է, որտեղ d-ը սովորաբար 768 կամ 1024 է՝ կախված մոդելից:

- h_1-ը՝ “The”-ի embedding-ն է՝ հաշվի առնելով ամբողջ նախադասության համատեքստը:
- h_5-ը՝ “jumps”-ի embedding-ն է, որը նույնպես context-aware ներկայացում է:

Մեզ համար կարևոր են հենց այս երկուսը, որովհետև դրանք են boundary embedding-ները span-ի համար:

SpanBERT-ը վերցնում է span-ի արտաքին երկու token-ների embeddings.

$$h_{s-1} = h_1$$

$$h_{e+1} = h_5$$

Այսինքն:

- Ձախ boundary embedding՝
  h_1=emb("The")
- Աջ boundary embedding՝
  h_5=emb("jumps")

Այս երկու token-ները span-ի իրական բառերի հետ հաճախ ունեն syntactic և semantic կապեր.

Օրինակ.
“The quick brown fox” արտահայտության մեջ “The”-ը span-ի դետերմինանտն է:
“jumps” բայը կապում է span-ը նախադասության գործողության հետ:

Այսինքն՝ span-ի իմաստը բնականորեն ընդգրկված է հենց այդ երկու **boundary token-**ների մեջ:

Span-ի ներսի token-ները ստանում են հարաբերական դիրքեր՝

$$p_i = i - s$$

Այս span-ի համար.

$$x_2 = \text{“quick”} \rightarrow p_2 = 0$$
$$x_3 = \text{“brown”} \rightarrow p_3 = 1$$
$$x_4 = \text{“fox”} \rightarrow p_4 = 2$$

Դիրքային embedding-ներ՝

$$r_2 = E(0), \quad r_3 = E(1), \quad r_4 = E(2)$$

Դիրքային embedding-ը մոդելին ասում է span-ի ներսում token-ի տեղը:

Օրինակ՝ “quick” span-ի սկիզբն է, “fox”՝ վերջը:

Յուրաքանչյուր span-ի ներսի token (մասնավորված) ներկայացվում է boundary embedding-ների և դիրքային embedding-ի համակցությամբ.

$$\tilde{h}_i = f_{FFN}\left([h_{s-1}; h_{e+1}; r_i]\right)$$

Ի՞նչ է սա իրականում նշանակում:

Span-ի յուրաքանչյուր token վերականգնվում է՝

Ձախ սահմանի իմաստը + Աջ սահմանի իմաստը + **span-ի** ներսի դիրքը

Օրինակ.

Token 1 — “quick”

$$\tilde{h}_2 = f([h_1; h_5; r_2])$$

Այսինքն՝ մոդելը փորձում է հասկանալ.

— “The” բառից առաջ ինչ ածական կարող է գալ → “quick”

— “jumps”-ից առաջ ինչ կենդանատիպ կամ օբյեկտային սուբյեկտ կարող է գտնվել

Token 2 — “brown”

$$\tilde{h}_3 = f([h_1; h_5; r_3])$$

Այս token-ը span-ի մեջ երկրորդն է, ու դիրքային embedding-ը մոդելին ասում է, որ այստեղ կարող է լինել ևս մեկ ածական:

Token 3 — “fox”

$$\tilde{h}_4 = f([h_1; h_5; r_4])$$

Այս token-ը span-ի վերջում է՝
r_2 և r_3 embedding-ներից դա մոդելը տարբերում է:

Մոդելը սեմանտիկորեն հետևում է, որ.
“quick brown” → ուժեղ ածական+ածական կառուցվածք → “fox” ամենահավանական գոյական

SBO-ն պահանջում է, որ մոդելը միայն (h_1,h_5) և դիրքային embedding-ների միջոցով վերականգնի span-ը:

$$\tilde{h}_i \rightarrow P(x_i)$$

Մեր span-ի դեպքում՝

$$\tilde{h}_2 \rightarrow \text{“quick”}$$
$$\tilde{h}_3 \rightarrow \text{“brown”}$$
$$\tilde{h}_4 \rightarrow \text{“fox”}$$

Span-ը վերականգնվում է ճշգրտորեն, որովհետև “The ___ ___ ___ jumps” կառուցվածքում ամենահավանականը հենց “quick brown fox”-ն է:

SpanBERT-ը սովորում է նման կառուցվածքներ միլիոնավոր օրինակների վրա:

BERT-ը առանձին token-ներ էր կռահում:
SpanBERT-ը span-ը կռահում է որպես ամբողջություն:

Ի տարբերություն BERT-ի՝ SpanBERT-ը սովորում է.

- ածական+ածական+գոյական շղթաներ
- անվանական խմբեր
- անհատական entity-ների (օրինակ՝ անձի անուններ) ամբողջական կառուցվածքը
- գործողություններ, որոնք ներկայացվում են մի քանի token-ներով
- dependency հարաբերություններ span-ի ներսում
- syntactic consistency ամբողջ արտահայտության համար

Ուստի “quick brown fox”-ը վերականգնվելը SpanBERT-ի համար ավելի բնական է, քան BERT-ի:

Հաջորդ փուլում կատարվում է առաջադրանքի պահանջների և hint-երի համադրությունը՝ կիրառելով **MPNet** մոդելը (Song et al., *MPNet: Masked and Permuted Pre-training for Language Understanding*, NeurIPS 2020, https://arxiv.org/abs/2004.09297):

MPNet-ի գաղափարը

MPNet-ը միավորում է երկու նախորդ հիմնական նախապատրաստական մեթոդները՝ **Masked Language Modeling (MLM)**՝ որը օգտագործում է BERT, և **Permuted Language Modeling (PLM)**՝ որը օգտագործում է XLNet:

Masked Language Modeling (MLM) մոտեցումը հնարավորություն է տալիս մոդելին հասկանալ նախադասության երկկողմանի (bidirectional) համատեքստը, սակայն այն ենթադրում է, որ փակված (**masked**) token-ները միմյանցից անկախ են: Այսինքն՝ եթե նախադասության մեջ մի քանի բառ փոխարինված են [MASK] նշանով, մոդելը յուրաքանչյուր բառի կանխատեսումն իրականացնում է առանձին՝ առանց հաշվի առնելու դրանց միջև եղած շարահյուսական կամ իմաստային կապը:

Իսկ **Permuted Language Modeling (PLM)** մեթոդը (օր.՝ XLNet-ում) հաղթահարում է այս սահմանափակումը՝ ներմուծելով **token dependency** գաղափարը: Այն սովորում է կանխատեսել հաջորդ token-ը՝ հիմնվելով նախորդների վրա՝ կիրառելով **autoregressive modeling**: Այս եղանակը ապահովում է token-ների փոխկապակցված կանխատեսում, բայց քանի որ մոդելը տեսնում է միայն նախորդ token-ները, այն կորցնում է նախադասության ամբողջական դիրքային կառուցվածքը (**position information**):

MPNet-ը միավորում է MLM-ի և PLM-ի առավելությունները՝ միաժամանակ

1. պահելով **dependency modeling**՝ կանխատեսվող token-ների միջև,

2. ապահովելով **full position awareness**, որպեսզի մոդելը տեսնի նախադասության ամբողջ դիրքային կառուցվածքը pre-training-ի ժամանակ:

Հաջորդաբար բաժանենք token embedding-ների շարքը երկու ենթաբաժնի՝

- $Z \leq c=(z_1,\ldots,z_c)$ — չփակված (non-predicted) կամ դիտելի token-ներ,
- $Z > c=(z_c+1,\ldots,z_n)$ — փակված (predicted) կամ կանխատեսվող token-ներ:

Այս բաժանումը սահմանում է, թե որ հատվածն է մոդելին հասանելի և որի հիման վրա պետք է կանխատեսվեն բացակայող token-ները:

MLM մեթոդի նպատակը կայանում է փակված **token-**ների պայմանական հավանականության մաքսիմալացման մեջ, այսինքն՝ մոդելը սովորում է հավանական բաշխումը

$$P(x_{>c} \mid x_{\leq c}, M_{z_{>c}})$$

որտեղ՝

- $x_{>c}$ — փակված (masked) բառերի հավաքն է,
- $x_{\leq c}$ — չփակված բառերի համատեքստն է,
- $M_{z_{>c}}$ — դիմակավորման (masking) մատրիցան է, որը սահմանում է, թե որ token-ներն են փակված:

Ուսուցման նպատակային ֆունկցիան հետևաբար ունի այս տեսքը՝

$$\log P(x_{>c} \mid x_{\leq c}, M_{z_{>c}})$$

Two-Stream Self-Attention

MPNet-ը օգտագործում է **Two-Stream Self-Attention** մեխանիզմ՝ ինչպես XLNet-ում: Այն ունի երկու հոսք՝

- **Content Stream**՝ որն աշխատում է ամբողջ իրական կոնտեքստի վրա,
- **Query Stream**՝ որն կանխատեսում է տվյալ token-ը՝ առանց տեսնելու դրա բովանդակությունը:

Այս կառուցվածքը հնարավորություն է տալիս մոդելին սովորել կանխատեսվող token-ների միջև եղած կախվածությունները՝ առանց ուսուցման ընթացքում անմիջապես բացահայտելու կամ օգտագործելու իրական թիրախ բառերը:

Position Compensation

MPNet-ը կիրառում է **Position Compensation** մեխանիզմը, որի շնորհիվ յուրաքանչյուր **query stream**ստանում է ամբողջ նախադասության դիրքային **embedding**-ների հասանելիություն: Այսինքն՝ մոդելը չի տեսնում դեռ չմշակված բառերի բովանդակությունը, բայց գիտի դրանց տեղակայությունը նախադասության մեջ: Սա թույլ է տալիս պահպանել նախադասության ամբողջական կառուցվածքային պատկերացումը՝ առանց խախտելու հերթականության կանոնները:

Երբ requirement-ներն ու hint-երը վերլուծված են, համակարգը վերլուծում է դասախոսի կոդը՝ կիրառելով static և dynamic մեթոդներ: Static վերլուծությունը հիմնված է **Abstract Syntax Tree (AST)** կառուցվածքի վրա, որտեղ յուրաքանչյուր գործողություն ներկայացվում է որպես գրաֆի հանգույց, իսկ կապերը՝ որպես կողեր: Այս մեթոդը հիմնված է Allamanis et al. (*Learning to Represent Programs with Graphs*, ICLR 2018,

https://arxiv.org/abs/1711.00740) աշխատության վրա և կիրառում է **Graph Gated Neural Network (GGNN)** մեխանիզմը՝

Ծրագրի կոդը կարելի է դիտարկել ոչ թե որպես սովորական token-ների հաջորդականություն, այլ որպես գրաֆ **(Graph)**, որտեղ կան ոչ միայն շարահյուսական կապեր, այլև սեմանտիկ կապեր՝ օրինակ՝ տվյալների հոսք (data flow), փոփոխականների օգտագործման շղթա, ֆունկցիայի վերադարձի ուղղություն և այլն:

Այսպիսով, ամբողջ ծրագիրը մոդելավորվում է որպես գրաֆ

$$G = (V, E, X),$$

որտեղ՝

- V — բոլոր հոդերի բազմությունն է (օրինակ՝ AST nodes, tokens, variables),
- E — կապերի (edges) բազմությունն է, ներառյալ տարբեր տեսակների կողեր՝ AST, data flow, control flow և այլն,
- X — հոդերի սկզբնական հատկանիշների մատրիցն է, որտեղ յուրաքանչյուր հոդ ունի embedding-ային ներկայացում (token embedding + type embedding և այլն):

Յուրաքանչյուր հոդ $v \in V$ ունի սկզբնական ներկայացում

$$h_v^{(0)} \in \mathbb{R}^d$$

որն ստացվում է embedding-ների միավորումից, օրինակ՝

$$h_v^{(0)} = [t_v; e_v],$$

Որտեղ

- t_v — token embedding-ն է (օրինակ՝ “sum”, “for”, “+”),
- e_v — one-hot vector է, որը ներկայացնում է հոդի տիպը (identifier / operator / literal / keyword / AST node և այլն)։

Այս գրաֆը որպես ամբողջություն փոխանցվում է **Gated Graph Neural Network (GGNN)** մոդելին, որը պետք է սովորի յուրաքանչյուր հոդի իմաստային ներկայացումը՝ հաշվի առնելով նրա հարևանների տեղեկատվությունը։

GGNN-ի հիմնական գաղափարն այն է, որ յուրաքանչյուր հոդի embedding-ը փուլ առ փուլ (steps) թարմացնենք՝ հաշվի առնելով նրա հարևանների ներկա վիճակները։

Պատկերացնենք, որ ունենք կապերի տեսակների բազմություն՝

$$R = \{\text{AST}, \text{DataFlow}, \text{ControlFlow}, \ldots\}$$

Յուրաքանչյուր կապի համար կա իր բաշային մատրիցա W_r։

t-րդ քայլում հոդ v-ի համար հարևաններից եկող «հաղորդագրությունը» հաշվարկում ենք՝

$$m_v^{(t,r)} = \sum_{(u,v)\in E_r} W_r h_u^{(t)}$$

որտեղ՝

- E_r — r տեսակի կողերով կապ ունեցող զույգերն են,
- $h_u^{(t)}$ — u հոդի embedding-ը t-րդ քայլում,
- W_r — տվյալ կապի տեսակին բնորոշ բաշային մատրից։

Այնուհետև, բոլոր կապերի տեսակներից ստացված հաղորդագրությունները միացնում ենք՝

$$m_v^{(t)} = \sum_{r \in R} m_v^{(t,r)}$$

որտեղ $m_v^{(t)}$ արդեն վեկտոր է, որը ներկայացնում է ամբողջ հարևանի ինֆորմացիան տվյալ նոդի մասին t-րդ քայլում:

Gated Update (GRU)

Հաջորդ քայլում յուրաքանչյուր նոդի embedding-ը թարմացվում է GRU (Gated Recurrent Unit) մեխանիզմով՝

$$h_v^{(t+1)} = \mathrm{GRU}(h_v^{(t)}, m_v^{(t)})$$

Այսինքն՝ նոդի նոր վիճակը կախված է.

- նրա նախորդ embedding-ից $h_v^{(t)}$,
- հարևաններից եկող ընդհանուր հաղորդագրությունից $m_v^{(t)}$

Այս քայլը կրկնվում է T անգամ (T message-passing steps), որի արդյունքում ստանում ենք վերջնական ներկայացում

$$h_v^{(T)}$$

որն արդեն պարունակում է ծրագիր-գրաֆի գլոբալ կոնտեքստից բխող իմաստային ինֆորմացիա:

VARMISUSE task–ի նպատակը հետևյալն է.
տրված է ծրագիր, որտեղ որոշ կետում սխալ փոփոխական է օգտագործվել (բայց տիպով ճիշտ է), և մոդելը պետք է որոշի՝ որ փոփոխականը պետք է լիներ այնտեղ իրականում:

Ցորմալ ձևակերպում.

Թող V_t լինի տիպով թույլատրելի բոլոր փոփոխականների բազմությունը տվյալ դիրքում (slot-ում): Մեր խնդիրն է ընտրել ճիշտ փոփոխականը $v* \in Vt$:

Քննարկենք պարզ C-անման կոդի օրինակ՝ միջինի սխալ հաշվարկով.

```
int sum = 0;
int count = 0;
for (int i = 0; i < n; i++) {
   sum = sum + a[i];
   count = count + 1;
}
double avg = sum / count;  // այստեղ ամեն ինչ ճիշտ է
```

Հիմա ձևավորենք **VARMISUSE** օրինակ.
ենթադրենք, որ ծրագրավորողը սխալմամբ գրել է.

```
double avg = sum / n;  // կամ հակառակը՝ sum / count → sum / n
```

կամ հակառակ ուղղությամբ սխալը.

Այստեղ slot-ը denominator-ի տեղն է՝ sum / ?:

Վերցնենք կոնկրետ սխալը.

double avg = sum / n;

բայց ground-truth ճիշտ տարբերակը պետք է լիներ.

double avg = sum / count;

Այս դեպքում տիպով ճիշտ փոփոխականներն են `{sum, count, n}`, և slot-ի համար candidate set-ը կլինի.

$$V_t = \{\text{sum}, \text{count}, \text{n}\}$$

Մեր նպատակը՝ ընտրել `count`–ը՝ որպես ճշգրիտ փոփոխական:

Գրաֆային ներկայացում

Ծրագրի գրաֆը G=(V,E,X) ներառում է՝

- հոդեր՝
  - յուրաքանչյուր փոփոխականի սահմանում (definition) և օգտագործում (use),
  - օպերատորներ (`+`, `/`, `<`),
  - control flow կառուցվածքներ (`for`),
  - literal-ներ (`0`, `1`, և այլն),
- կողեր (edges)՝
  - Data Flow edges — օրինակ՝ `sum`–ի արժեքը line 4-ից հոսում է line 6,
  - Control Flow edges — from `for` condition to loop body,
  - AST edges — ծնող–զավակ կապեր AST ծառի մեջ:

VARMISUSE slot-ը ճշտանակում ենք հատուկ **node**-ով՝ v^{SLOT}, որը ներկայացնում է «դատարկ տեղը» denominator-ի ցուցադրված դիրքում.

Ավելին, յուրաքանչյուր կանդիդատ փոփոխականի համար՝ $v \in V_t$, ստեղծվում է **speculative node** $v_{t,v}$, որը ներկայացնում է «եթե տվյալ slot-ում օգտագործենք v փոփոխականը» սցենարը: Այս նոդերը նույնպես ներառվում են գրաֆում և անցնում GGNN propagation:

GGNN propagation VARMISUSE-ի դեպքում

GGNN message passing–ից հետո ստանում ենք երկու կարևոր տեսակի ներկայացումներ.

1. Slot-ի կոնտեքստի ներկայացում.

$$c(t) = h_{v^{SLOT}}^{(T)}$$

2. Յուրաքանչյուր կանդիդատ փոփոխականի «օգտագործման» ներկայացում slot-ում.

$$u(t, v) = h_{v_{t,v}}^{(T)}$$

Այստեղ՝

- c(t) — encode է անում, թե ինչպիսի փոփոխական է տրամաբանական այս կետում (օրինակ, «loop-ի long-term counter», «sum-ի վրա հիմնված something», և այլն),
- u(t,v) — encode է անում, թե ինչպիսի semantic ազդեցություն կունենա, եթե slot-ում օգտագործենք հենց v փոփոխականը:

Այնուհետև հաշվարկվում է յուրաքանչյուր կանդիդատի համար սկոր.

$$s(t, v) = f(c(t), u(t, v))$$

որտեղ f-ը կարող է լինել պարզ dot-product կամ փոքր MLP (multi-layer perceptron):

Մոդելը ընտրում է այն փոփոխականը, որի սկորը առավելագույն է.

$$\hat{v} = \arg\max_{v \in V_t} s(t, v)$$

Թիրախ՝ $\hat{v} = v^* = \text{count}$:

Կորուստի ֆունկցիա **(Loss)**

Օգտագործվում է max-margin objective, որը ապահովում է, որ ճիշտ փոփոխականի score-ը մեծ լինի բոլոր սխալներից առնվազն margin-ով.

$$\mathcal{L}(t) = \sum_{v \in V_t \setminus \{v^*\}} \max\left(0,\ 1 - s(t, v^*) + s(t, v)\right)$$

Այստեղ՝

- v^* — ground-truth փոփոխականն է (մեր օրինակում `count`),
- եթե սխալ կանդիդատի score-ը շատ է աճում, կորուստը «պատժում է» մոդելին՝ ստիպելով բարձրացնել ճիշտ փոփոխականի score-ը:

```
int sum = 0;
int count = 0;

for (int i = 0; i < n; i++) {
   sum = sum + a[i];
   count = count + 1;
}

double avg = sum / count;
return avg;
```

Քայլ 1 — AST (Abstract Syntax Tree) – շարահյուսական ծառեր

Առաջին շերտը՝ *syntax*–ն է: Compiler–ը / parser–ը կոդից կառուցում է AST:Պարզեցված AST node-ներ՝ (ID + նկարագրություն)

Node	Նոդի բովանդակություն
N1	VarDecl: `int sum = 0`
N2	Literal: `0`
N3	VarDecl: `int count = 0`

N4 Literal: `0`

N5 ForStatement

N6 VarDecl: `int i = 0`

N7 Literal: `0`

N8 Cond: `i < n`

N9 Id: `i` (condition)

N10 Id: `n`

N11 Update: `i++`

N12 Assign: `sum = sum + a[i]`

N13 Id: `sum` (LHS)

N14 Expr: `sum + a[i]`

N15 Id: `sum` (RHS)

N16 ArrayAccess: `a[i]`

N17 Id: `a`

N18 Id: `i` (array index)

N19 Assign: `count = count + 1`

N20 Id: `count` (LHS)

N21 Expr: `count + 1`

N22 Id: `count` (RHS)

N23 Literal: `1`

N24 VarDecl: `double avg = sum / count`

N25 Expr: `sum / count`

N26 Id: `sum` (for avg)

N27 Id: `count` (for avg)

N28	ReturnStatement
N29	Id: `avg` (returned)

Սա նույն կոդն է, բայց ծառի նոդերի տեսքով:

AST edges (ծնող → զավակ) օրինակներ.

- N1 → N2 (sum-ի հայտարարությունը պարունակում է literal 0)
- N3 → N4 (count-ի հայտարարությունը պարունակում է literal 0)
- N5 → N6, N5 → N8, N5 → N11, N5 → N12, N5 → N19
- N8 → N9, N8 → N10
- N12 → N13, N12 → N14
- N14 → N15, N14 → N16
- N16 → N17, N16 → N18
- N19 → N20, N19 → N21
- N21 → N22, N21 → N23
- N24 → N25
- N25 → N26, N25 → N27
- N28 → N29

Սրանք դեռ միայն շարահյուսական կապեր են:

Քայլ 2 — Control Flow Graph (CFG) – կատարողական ուղի

Հաջորդը կառուցում ենք **Control Flow Graph**, որը ցույց է տալիս՝
ծրագիրը որ կարգով է «վազում»:

Կարող ենք մտածել statement-level node-ներով.

- C1: `int sum = 0;` (կապում ենք N1 AST node-ին)

- C2: `int count = 0;` (N3)
- C3: `for (...) { ... }` (լոգիկական loop node՝ հիմնված N5 վրա)
- C4: `sum = sum + a[i];` (N12)
- C5: `count = count + 1;` (N19)
- C6: `double avg = sum / count;` (N24)
- C7: `return avg;` (N28)

Control Flow edges.

- C1 → C2
- C2 → C3
- C3 → C4 (loop body)
- C4 → C5
- C5 → back to C3 condition (loop),
- երբ condition-ը false է՝ C3 → C6
- C6 → C7

Սա արդեն կատարման հերթականություն է:

Քայլ 3 — Data Flow Graph (DFG) – արժեքների հոսք

Հիմա նույն կոդից հանում ենք՝ ով ում վրա է ազդում:

Օրինակներ.

- Literal `0` → `sum`
  N2 → N1 (sum = 0)
- Literal `0` → `count`
  N4 → N3
- `sum`–ի նախորդ արժեքը → RHS sum + a[i].
  N1 → N15 (sum-ի 0 արժեքը առաջին անգամ RHS-ում)
  հետո loop-ի մեջ N13/N10 → N15 → N14 → N12 և այլն (կուտակում)
- `i` փոփոխականը.
  N6 (i=0) → N9 (i<n) → N18 (array index) → N11 (i++) → նորից N9 ...

- `count` փոփոխականի data flow.
  N3 (count=0) → N22 (RHS) → N21 → N20 (LHS) → հետո նորից RHS ...
  → վերջում N27 (avg-ի հաշվարկում)
- `sum` → `avg` → `return`
  N26 (sum in avg expr) ← N13/N15 data-flow շղթաներ,
  N24 (avg decl) → N29 (return avg)

DFG edges-ից մի քանի օրինակ.

From	To	Իմաստ
N2	N1	0 → sum
N4	N3	0 → count
N1	N15	initial sum → RHS
N3	N22	initial count → RHS
N15	N14	sum → sum + a[i]
N16	N14	a[i] → sum + a[i]
N14	N12	expr → new sum
N12	N13	new sum assign to LHS
N22	N21	count → count + 1
N21	N19	expr → new count
N19	N20	new count assign
N13	N26	final sum → avg expr
N20	N27	final count → avg expr
N24	N29	avg → return value

DFG-ը իմաստային **skeleton**–ն է, որի վրա GGNN–ը սովորում է, թե ով ինչ դեր ունի ծրագրում (accumulator, counter, bound և այլն)։

Քայլ 4 — Program Graph = AST + CFG + DFG

Հիմա արդեն ունենք ամբողջական program graph.

$$G = (V, E, X)$$

որտեղ՝

- **V** – node-ների բազմություն
  - AST node-ներ (N1…N29)
  - (կամ statement-level node-ներ CFG-ի համար)
- **E** – կողերի բազմություն
  - AST մինչ parent→child edges
  - CFG (control-flow) edges
  - DFG (data-flow) edges
- **X** – node feature-ներ
  - token embedding (օր. word embedding “sum”, “for”, “return”)
  - type embedding (identifier / literal / operator / keyword / AST kind)
  - հնարավոր է՝ լրացուցիչ հատկանիշներ (line number, scope level, is_definition/use, և այլն)

Ցուրաքանչյուր node v–ի համար ունենք սկզբնական ներկայացում.

$$h_v^{(0)} = [t_v; e_v]$$

որտեղ t_v — token embedding, e_v — type one-hot embedding։

Այս գրաֆը հետո տալիս ենք GGNN-ին, որը T քայլ message passing-ի միջոցով ստանում է $h_v^{(T)}$ — արդեն իմաստային հագեցած node embeddings, որոնք օգտագործում ենք VARMISUSE / VARNAMING / defect detection–ի համար։

Dynamic վերլուծության համար կիրառվում է **constraint-based test generation** մեթոդը (Zeller et al., *Automated Test Generation: Beyond Coverage*, ICSE 2022, https://dl.acm.org/doi/10.1145/3510003.3510139)։ Համակարգը requirement-ից գեներացնում է թեստային դեպքեր՝ ներառելով ծայրահեղ մուտքեր՝ դատարկ զանգվածներ, բացասական արժեքներ, կրկնվող տարրեր, division by zero և այլն։ Այս եղանակով ստուգվում է ոչ միայն ծրագրի ճիշտ աշխատելը, այլ նաև նրա վարքագիծը՝ edge-case իրավիճակներում։

Constraint-based test generation (CBTG) մեթոդը հիմնված է այն գաղափարի վրա, որ ծրագրի վարքագիծը կարելի է արտահայտել մաթեմատիկական սահմանափակումների համակարգի (constraints system) միջոցով։ Այս մոտեցման նպատակն է ավտոմատ կերպով գեներացնել այնպիսի մուտքային արժեքներ, որոնք ոչ միայն ստուգում են ծրագրի ճիշտ կատարողականությունը, այլև բացահայտում են եզրային և խոցելի իրավիճակները (edge cases)՝ օրինակ՝ դատարկ զանգվածներ, բացասական թվեր, բաժանում զրոյի, կրկնվող տարրեր, կամ սահմանային արժեքների գերազանցում։

CBTG-ն համատեղում է ծրագրի սիմվոլիկ ներկայացումը (symbolic execution) և սահմանափակումների լուծման մեթոդաբանությունը (constraint solving), ինչի արդյունքում թեստերի ստեղծումը դառնում է ֆորմալ և ապացուցելի գործընթաց։

Թող ծրագիրը ներկայացվի որպես պատկերում

$$P : D \to O$$

որտեղ D — մուտքային արժեքների բազմությունն է, իսկ O — ելքային արդյունքների բազմությունը։

Ծրագրի կատարման ընթացքում յուրաքանչյուր հյուղային որոշում (if, while, switch և այլն) կարելի է ներկայացնել որպես տրամաբանական արտահայտություն՝ $g_i(x)$։

Ծրագրի յուրաքանչյուր ուղի (execution path) կարելի է նկարագրել հետևյալ կոնյունկցիայով՝

$$C_\pi(x) = \bigwedge_{i=1}^{n} g_i(x),$$

որտեղ Cπ(x) — տվյալ ուղու սահմանափակման բանաձև է:

Այն ընդգրկում է բոլոր այն պայմանները, որոնք պետք է բավարարվեն, որպեսզի ծրագիրը կատարի հենց այդ ընթացքը:

Այսպիսով՝ ծրագրի վարքագիծը կարելի է ներկայացնել constraints-ի միջոցով, իսկ մուտքային արժեքների որոնումը՝ որպես դրանց լուծում:

Թեստի գեներացման խնդիրը ձևակերպվում է որպես սահմանափակումների բավարարման խնդիր (Constraint Satisfaction Problem, CSP).

$$\text{Գտնել } x^* \text{ այնպես, որ } \Phi(x) = C_\pi(x) \wedge C_{\text{input}}(x) \wedge C_{\text{req}}(x)$$

Այստեղ.

- Cπ(x)— ծրագրի ուղու սահմանափակման բանաձև է (path constraint),
- Cinput(x) — մուտքային տիպերի և սահմանների սահմանափակումներն են (օրինակ՝ x>0, |A|>1),
- Creq(x)— պահանջներից (requirements) բխող սահմանափակումներ են, որոնք ներկայացնում են ծրագրի ակնկալվող վարքագիծը:

Եթե Φ(x) բավարարված է, ապա ստացվող լուծումը x* հանդիսանում է թեստային դեպք (test case):

Եթե բանաձևը անբավարարված է (UNSAT), ապա դա ապացույց է, որ տվյալ ուղին կամ պահանջը անիրագործելի է:

4. Պահանջների ներկայացում որպես **constraints**

Պահանջները (requirements) CBTG-ում ունեն տրամաբանական բնույթ և կարող են լինել երեք տեսակի՝

1. Նորմալ պայմաններ **(postconditions)** — արտահայտում են ակնկալվող արդյունքը, օրինակ՝

$$R(x, y): \ y > 0 \quad \text{կամ} \quad y = \sum_i x_i.$$

2. Անվտանգության պայմաններ **(safety constraints)** — ապահովում են ծրագրի չվթարային աշխատանքը, օրինակ՝

$$C_{\text{safety}}: d \neq 0 \quad \text{(չբաժանվի զրոյի վրա)}.$$

3.Հարաբերական պայմաններ **(metamorphic relations)** — կապում են տարբեր մուտքերի արդյունքները, օրինակ՝

$$P(T(x)) = T'(P(x)).$$

Այս պահանջները միավորվում են որպես ընդհանուր constraint Creq(x), որը solver-ը հաշվի է առնում test case-ի որոնման ժամանակ:

5. Եզրային դեպքերի մոդելավորում

CBTG-ի առավելություններից մեկն այն է, որ այն ուղղակիորեն ներառում է **edge-case constraints**՝ հատուկ սահմանափակումներ, որոնք ապահովում են ծրագրի ստուգումը ոչ սովորական իրավիճակներում:

Օրինակային ձևակերպումները.

Դատարկ զանգված:	$\text{len}(A) = 0,$
Միակ տարրով զանգված:	$\text{len}(A) = 1,$
Բացասական արժեքներ:	$\exists i : A_i < 0,$
Կրկնվող տարրեր:	$\exists i \neq j : A_i = A_j,$
Բաժանում զրոյի:	$d = 0,$
Սահմանային արժեքներ:	$A_i = \text{INT}_{\max}$ կամ $\text{INT}_{\min}.$

Այս սահմանափակումները solver-ին ուղղորդում են ոչ թե սովորական, այլ բացառիկ վարքագծեր առաջացնող մուտքերի որոնման ուղղությամբ:

Արդյունքում ստացվում են թեստեր, որոնք արտահայտում են ծրագրի գործառական սահմանները:

6. SMT լուծիչի դերը

Constraint-based test generation մեթոդը հիմնված է SMT (Satisfiability Modulo Theories) լուծիչների վրա: Լուծիչը ստանում է բանաձևերի համակարգը Φ(x) և որոշում՝ այն բավարա՞ր է, թե ոչ:

Բավարարվածության տեսությունները, որոնց հիման վրա աշխատում են solver-ները, ներառում են.

- **LIA**՝ Linear Integer Arithmetic՝ ամբողջ թվերի համար,
- **BV**՝ Bit-vectors՝ overflow-երի և թվային սահմանների համար,
- **Array Theory**՝ զանգվածների ինդեքսավորման համար,
- **String Theory**՝ տողերի և նիշերի մոդելավորման համար:

Լուծիչը վերադարձնում է կոնկրետ արժեքների բազմություն x^*, որը բավարարում է բոլոր սահմանափակումները: Այդ x^*-ն դառնում է ավտոմատ գեներացված թեստ:

7. Coverage-ի գաղափարը

CBTG-ի որակը չափվում է տարբեր տեսակի ծածկույթներով (coverage metrics)։

- **Path coverage** — ֆանի տարբեր ուղի է կատարվել ծրագրում։
- **Branch coverage** — ֆանի ճյուղային պայման է ստուգվել երկու ուղղություններով։
- **Requirement coverage** — ֆանի պահանջ է ստուգվել կամ խախտվել։
- **Boundary coverage** — ֆանի եզրային արժեք է ներառվել մուտքային տարածքում։

8. Թեստերի գեներացման ալգորիթմի տրամաբանական կառուցվածք

Թեստերի գեներացումը CBTG մեթոդով հետևում է տրամաբանական հաջորդականությանը՝

1. Ծրագիրը գործարկվում է որոշակի նախնական մուտքով։
2. Կատարման ընթացքում հավաքվում են բոլոր ճյուղային պայմանները՝ ձևավորելով Cπ
3. Solver-ը ընտրում է պայմաններից մեկը և հակադարձում է այն՝ ստեղծելով նոր ուղի։
4. Ստեղծվում է ընդհանուր constraint՝

$$\Phi(x) = C'_{\pi}(x) \wedge C_{\text{input}}(x) \wedge C_{\text{req}}(x).$$

5. SMT լուծիչը փնտրում է մուտք x*, որը բավարարում է Φ(x)-ը։

6. Ստացված մուտքը դառնում է նոր test case, որը բացում է չտեսած ուղի։

Այս պրոցեսը կրկնվում է մինչև ծածկույթը բավարար է կամ պահանջների ամբողջ հավաքածուն ստուգված է։

Semantic consistency-ի ստուգման վուլում օգտագործվում է **DeBERTa-v3-large** մոդելը (He et al., *DeBERTa: Decoding-enhanced BERT with Disentangled Attention*, ACL 2021, https://arxiv.org/abs/2006.03654)։ Այն առանձնացնում է semantic embedding-ները positional embedding-ներից՝ ապահովելով ավելի ճշգրիտ ուշադրություն բարդ նախադասությունների վրա։ DeBERTa-ն կիրառվում է Natural Language Inference (NLI) առաջադրանքների համար՝ որոշելու entailment, neutral և contradiction հարաբերությունները requirement–code զույգերի միջև։ Եթե requirement-ում գրված է «Մի օգտագործիր sort()», իսկ կոդում առկա է sort(), ապա հարաբերությունը contradiction է, իսկ եթե ուսանողը ինքն է իրականացնում դասավորությունը՝ entailment։

BERT մոդելը (Bidirectional Encoder Representations from Transformers) դարձավ բնական լեզվի մշակման (NLP) ամենազարգացած հիմքերից մեկը, բայց այն ունի երկու տեսական սահմանափակում.

BERT-ը յուրաքանչյուր token-ի embedding-ին ավելացնում է նաև նրա դիրքային embedding-ը (position embedding) և դրանք գումարում է.

$$h_i = e_i + p_i$$

1. Այս գումարումը ճշանակում է, որ մոդելը չի կարող տարբերակել՝ տվյալ ազդակը վերաբերում է բառի իմաստին, թե դրա դիրքին նախադասության մեջ:
2. Բացարձակ դիրքային կախվածություն **(Absolute positional dependency)** — BERT-ը չի հասկանում բառերի հարաբերական հեռավորությունը (relative distance):
   Օրինակ՝ «The cat sat on the mat» և «On the mat sat the cat» նախադասությունները ունեն նույն իմաստը, բայց BERT-ում դրանք տարբեր embedding-ներ ունեն, որովհետև դիրքերը փոխվել են:

Այս սահմանափակումները խանգարում են մոդելին հասկանալի ձևով սովորել շարահյուսական հարաբերությունները (syntactic relations) և իմաստային կապերը (semantic dependencies):

DeBERTa-ն առաջարկում է լուծում՝ բաժանելով այս երկու ազդակները և վերածևելով attention մեխանիզմը այնպես, որ մոդելը կարողանա հատուկ ձևով հաշվի առնել բառերի հարաբերական դիրքերը:

DeBERTa-ն առաջարկում է բաժանել յուրաքանչյուր token-ի ներկայացումը երկու անկախ բաղադրիչի՝

- Բովանդակային ներկայացում **(Content representation)** — բառի իմաստային embedding-ը, որը կոդավորում է ինչ բառ է դա:
- Դիրքային ներկայացում **(Position representation)** — embedding, որը կոդավորում է տվյալ բառի դիրքը մյուսների համեմատ:

Այս մոտեցումը կոչվում է **disentanglement**՝ բառացիորեն՝ “անջատում”, քանի որ այն բաժանում է տարբեր տիպի ինֆորմացիան երկու ենթատարածքների:

Այնուամենայնիվ, այս երկու ազդակները պետք է ինչ-որ պահի համադրվեն՝ հասկանալու համար բառերի միջև կապերը:
Դա տեղի է ունենում attention-ի հաշվարկի փուլում:

DeBERTa-ում յուրաքանչյուր token-ի համար հաշվում են երկու տեսակի query (հարցում) և key (բանալիներ) վեկտորներ.

- Content query/key — Qc,Kc
- Position query/key — Qr,Kr

Թող ii-րդ բառը գնահատի jj-րդ բառի հետ կապը:
Այդ կապը (attention score) հաշվարկվում է երեք բաղադրիչով.

$$A_{i,j} = (Q_{c,i}K_{c,j}^{\top}) + (Q_{c,i}K_{r,\delta(i,j)}^{\top}) + (K_{c,j}Q_{r,\delta(j,i)}^{\top})$$

որտեղ՝

- $Q_{c,i}K_{c,j}^{\top}$ → **content-to-content (C2C)** կապ,
- $Q_{c,i}K_{r,\delta(i,j)}^{\top}$ → **content-to-position (C2P)** կապ,
- $K_{c,j}Q_{r,\delta(j,i)}^{\top}$ → **position-to-content (P2C)** կապ:

Այսպիսով, DeBERTa-ն միաժամանակ հաշվի է առնում՝

1. բառերի իմաստային նմանությունը,
2. նրանց հարաբերական դիրքերը,
3. և ինչպես է դիրքը ազդում իմաստային կախվածության վրա:

Վերջնական attention-ը ստացվում է՝

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{A}{\sqrt{3d}}\right)V_c,$$

որտեղ dd՝ embedding-ի չափն է, իսկ $\sqrt{3d}$- ը հավասարակշռում է երեք բաղադրիչների ազդեցությունը:

Բոլոր բառերի զույգերի միջև հաշվարկվում է հարաբերական հեռավորությունը՝

$$\delta(i, j) = \text{clip}(j - i, -k + 1, k - 1),$$

որտեղ k՝ առավելագույն դիտարկվող պատուհանի երկարությունն է (օրինակ՝ k=64)։

Այս մոտեցումը թույլ է տալիս մոդելին սովորել՝

- որ բառերը մոտ են (հավանաբար փոխկապակցված են),
- իսկ որոնք հեռու են (ավելի թույլ կապ ունեն)։

Relative embeddings-ների կիրառումը նաև նվազեցնում է հիշողության ծախսը, քանի որ անհրաժեշտ է պահել միայն 2k2k embedding, ոչ թե N2N2 բոլոր զույգերի համար։

3.2. **DeBERTa-**ի լուծումը

DeBERTa-ն absolute դիրքի ինֆորմացիան ավելացնում է միայն pre-training-ի ժամանակ՝ Masked Language Modeling-ի decoder-ում։

Այս մեխանիզմը կոչվում է **Enhanced Mask Decoder (EMD)** և այն ապահովում է ուշ միավորում (*late fusion*).

- encoder-ում մոդելը սովորում է միայն հարաբերական կապերը,
- decoder-ում, երբ կանխատեսում է դիմակավորված բառը, այն օգտագործում է absolute դիրքը՝ ճշգրիտ token ընտրելու համար։

Այս բաժանումը օգնում է մոդելին պահել հարաբերական կապերի մաքրությունը encoder-ում և միևնույն ժամանակ հասկանալ նախադասության կառուցվածքային կարգը decoder-ում։

Վերջնական գնահատականը ստացվում է **Triad Consistency Model**-ով, որը միավորում է requirement–hint, requirement–code և hint–code փոխազդեցությունները՝ հաշվի

առնելով նրանց հարաբերական կարևորությունը: Ընդհանուր consistency score-ը հաշվարկվում է՝

$$Score = w_{RC} \cdot s_{RC} + w_{RH} \cdot s_{RH} + w_{HC} \cdot s_{HC}$$

որտեղ ($s_{\{\}}$)-երը համապատասխան *similarity* կամ *entailment* արժեքներն են, իսկ ($w_{\{\}}$)-երը՝ կշիռները: Փորձնական տվյալներով ($w_{\{RC\}} > w_{\{RH\}} > w_{\{HC\}}$), քանի որ requirement–code համապատասխանությունը ամենաշատն է ազդում ընդհանուր գնահատման ճշգրտության վրա:

Այս մոդուլները fine-tune են արվում adapter-based մեթոդներով՝ **LoRA (Low-Rank Adaptation)** և **IA3**, որոնք թույլ են տալիս վերապատրաստել մոդելները առանց ամբողջական retraining-ի: Փոփոխությունը կատարվում է փոքր մատրիցային շերտերում՝

$$h' = h + A_1(\sigma(A_2 h))$$

որտեղ (A_1) և (A_2) սովորելու մատրիցաներ են, իսկ σ ակտիվացման ֆունկցիա: Սա նվազեցնում է հաշվարկային ծախսերը և դարձնում համակարգը հարմար ուսումնական սերվերների վրա տեղակայման համար: