CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning

Séminaire TALia du 23/09/2022

Source

Computer Science > Machine Learning

[Submitted on 5 Jul 2022 (v1), last revised 23 Jul 2022 (this version, v2)]

CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, Steven C.H. Hoi

La tâche de génération de code

- La génération de code est la tâche consistant à générer un programme informatique satisfaisant à une description exprimée en langage naturel.
- Cette tâche peut être considéré comme un problème de traduction où la séquence d'entrée correspond à la spécification du problème en Anglais et l'on génère séquentiellement le code correspondant.
- On utilise des architectures classiques comme le Transformer pour résoudre cette tâche, notamment avec des modèles pré-entrainés comme **Github Copilot (=Codex)**.
- On teste ensuite les codes générés à l'aide de tests unitaires pour vérifier leur validité.

Problem Specification

A string is a palindrome if it reads the same from the left to the right and from the right to the left….If there is such a substring in s that is not a palindrome, print the maximum length of such a substring….
Example:
Input: 'hannah' Output: 5

Solution Program

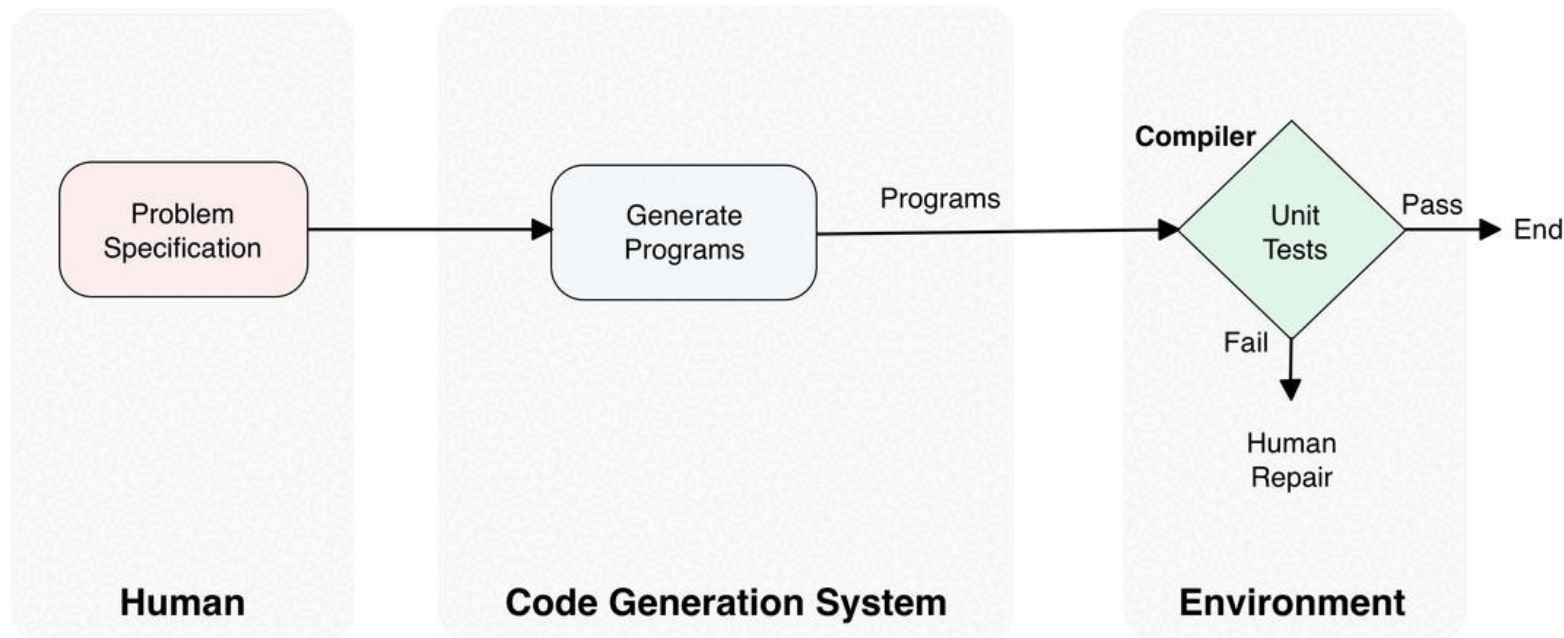
```
1 s = input()
2 ans = 0
3 for i in range(len(s)):
4     for j in range(i + 1, len(s) + 1):
5         if s[i:j]!= s[i:j][::-1]:
6             ans = max(ans, j - i)
7 print(ans)
```

Unit Tests

Input: wuffuw Output: 5
Input: iiiiiii Output: 0
… …

Méthode Transformer classique

$$\mathcal{L}_{ce}(\theta) = -\sum_t \log p_\theta(W|D) = -\sum_t \log[p_\theta(w_t|w_{1:t-1}, D)]$$

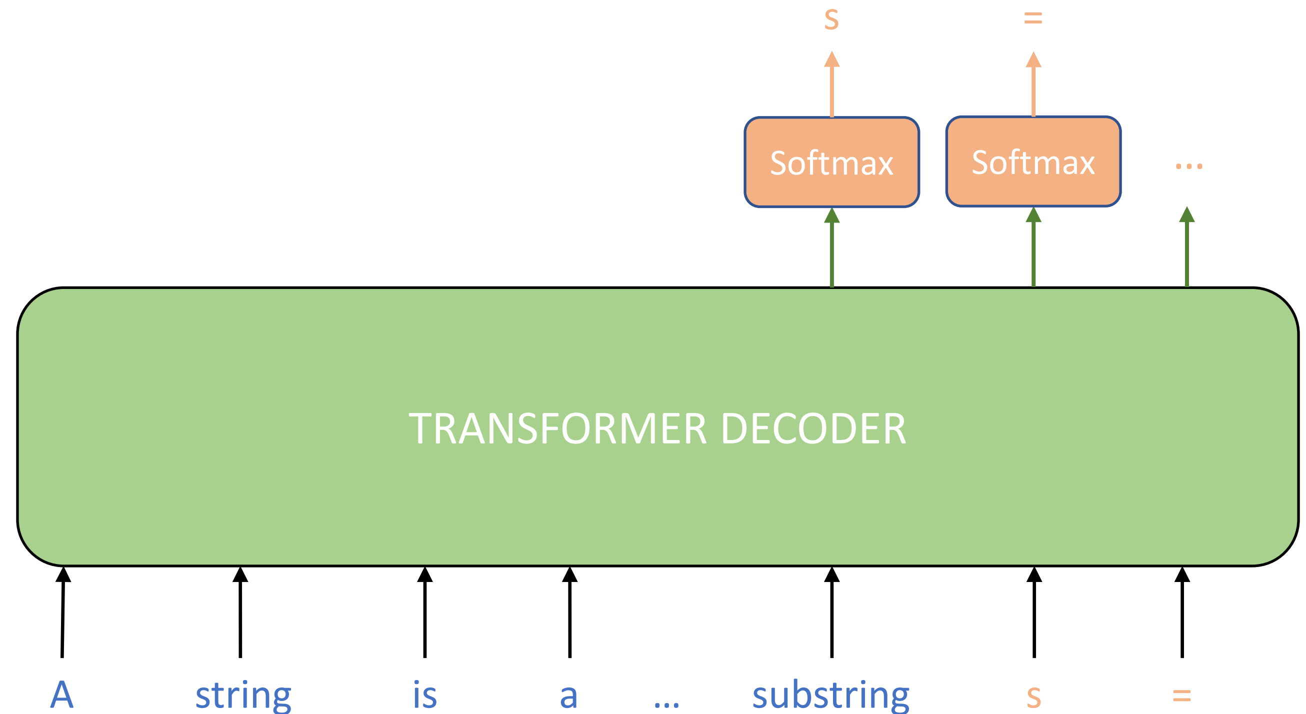


Les limites de l'approche *next-token prediction*

- A l'entraînement, on maximise la probabilité de générer le token suivant en faisant du *teacher forcing*. Au test, les erreurs s'accumulent à partir de l'instant où notre décodeur génère un mauvais token.

- Dans le cadre de la génération de code, c'est un problème encore plus important étant donné qu'un seul token mal placé peut entraîner un code non compilable.

- On n'intègre pas de signal propre au code dans cet entraînement, comme par exemple les tests unitaires qui doivent être utiles pour guider l'entraînement.



TALia

CodeRL – *fine-tuning* guidé par les tests unitaires

On formule le problème de synthèse de programme comme un problème de d'apprentissage par renforcement :

- θ *policy* décidant de la prochaine *action* comme la prédiction du prochain *token*.
- r *reward* obtenue par le modèle à la fin de la génération du code, mesurant la validité du code.

L'objectif du *fine-tuning* avec le *RL* est de minimiser :

$$\mathcal{L}_{rl}(\theta) = -\mathbb{E}_{W^s \sim p_\theta}[r(W^s)]$$

En suivant le *policy gradient theorem* et l'algorithme **REINFORCE**, on peut approximer le gradient $\nabla_\theta \mathcal{L}_{rl}(\theta)$ comme :

$$\begin{aligned}\nabla_\theta \mathcal{L}_{rl}(\theta) &\approx -\mathbb{E}_{W^s \sim p_\theta}[r(W^s)\nabla_\theta \log p_\theta(W^s|D)] \\ &\approx -\mathbb{E}_{W^s \sim p_\theta}[r(W^s)\sum_t \nabla_\theta \log p_\theta(w_t^s|w_{1:t-1}^s, D)]\end{aligned}$$

CodeRL à l'entraînement – les étapes de construction du modèle

1ère tentative : définir une heuristique vérifiant la validité du code

$$r(W^s) = \begin{cases} \text{-1.0} & \text{, if } W^s \text{ cannot be compiled (i.e. compile error)} \\ \text{-0.6} & \text{, if } W^s \text{ cannot be executed with unit tests (i.e. runtime error)} \\ \text{-0.3} & \text{, if } W^s \text{ failed any unit test} \\ \text{+1.0} & \text{, if } W^s \text{ passed all unit tests} \end{cases}$$

Problème :

- Enorme variance du gradient à l'entraînement, divergence du modèle.

CodeRL à l'entraînement

2ème tentative : calcul de la *reward* relativement à une *baseline*

$$\nabla_\theta \mathcal{L}_{rl}(\theta) \approx -\mathbb{E}_{W^s \sim p_\theta}[(r(W^s) - r(W^b)) \sum_t \nabla_\theta \log p_\theta(w_t^s | w_{1:t-1}^s, D)]$$

où W^b est le code généré en mode *greedy*.

Avantages :

- On diminue la variance lors de l'entraînement.
- On considère des programmes imparfaits mais meilleurs que la *baseline.*

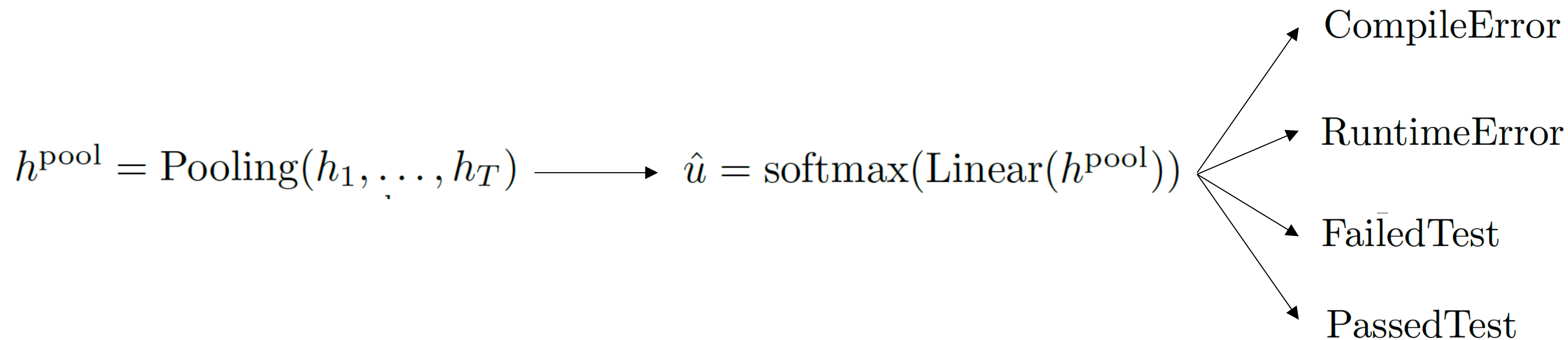
Problème :

- L'estimation du gradient ne se fait qu'à la fin du décodage.

CodeRL à l'entraînement

3ème tentative : *reward* intermédiaire calculée grâce à un modèle *critic*

$$\mathcal{L}_{critic}(\phi) = -\log p_\phi(u|W^s, D)$$



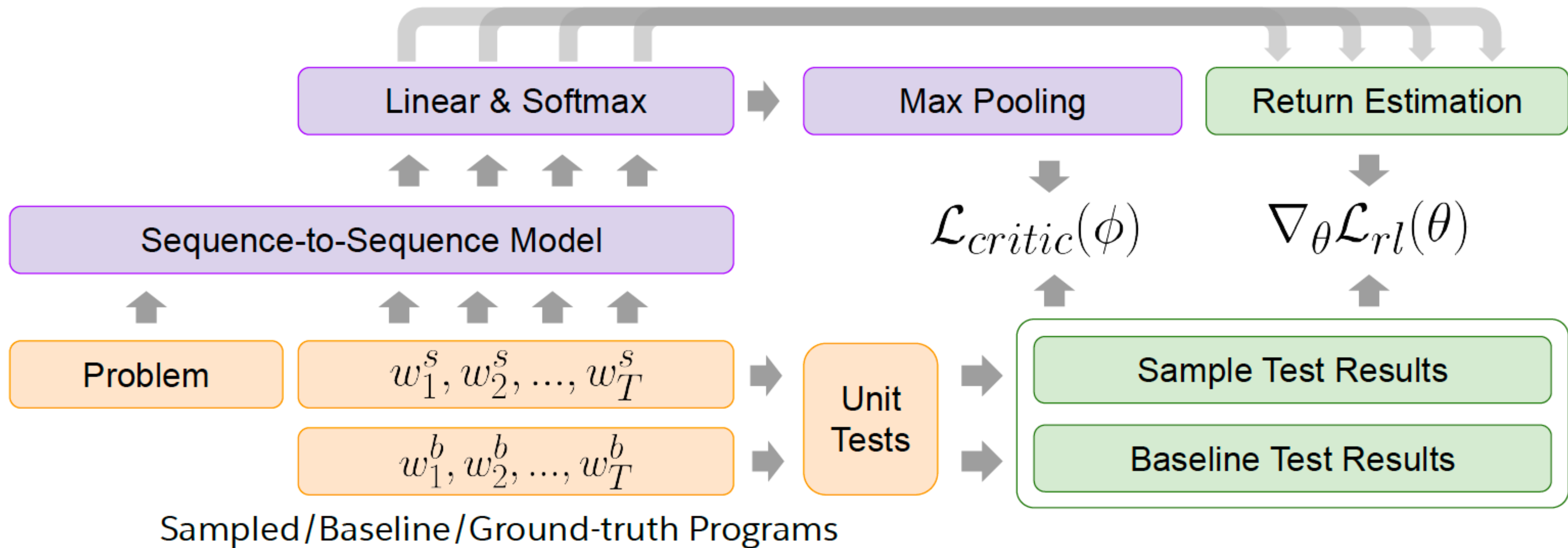
$$\nabla_\theta \mathcal{L}_{rl}(\theta) \approx -\mathbb{E}_{W^s \sim p_\theta}[(r(W^s) - r(W^b)) \sum_t \hat{q}_\phi(w_t^s) \nabla_\theta \log p_\theta(w_t^s | w_{1:t-1}^s, D)]$$

où

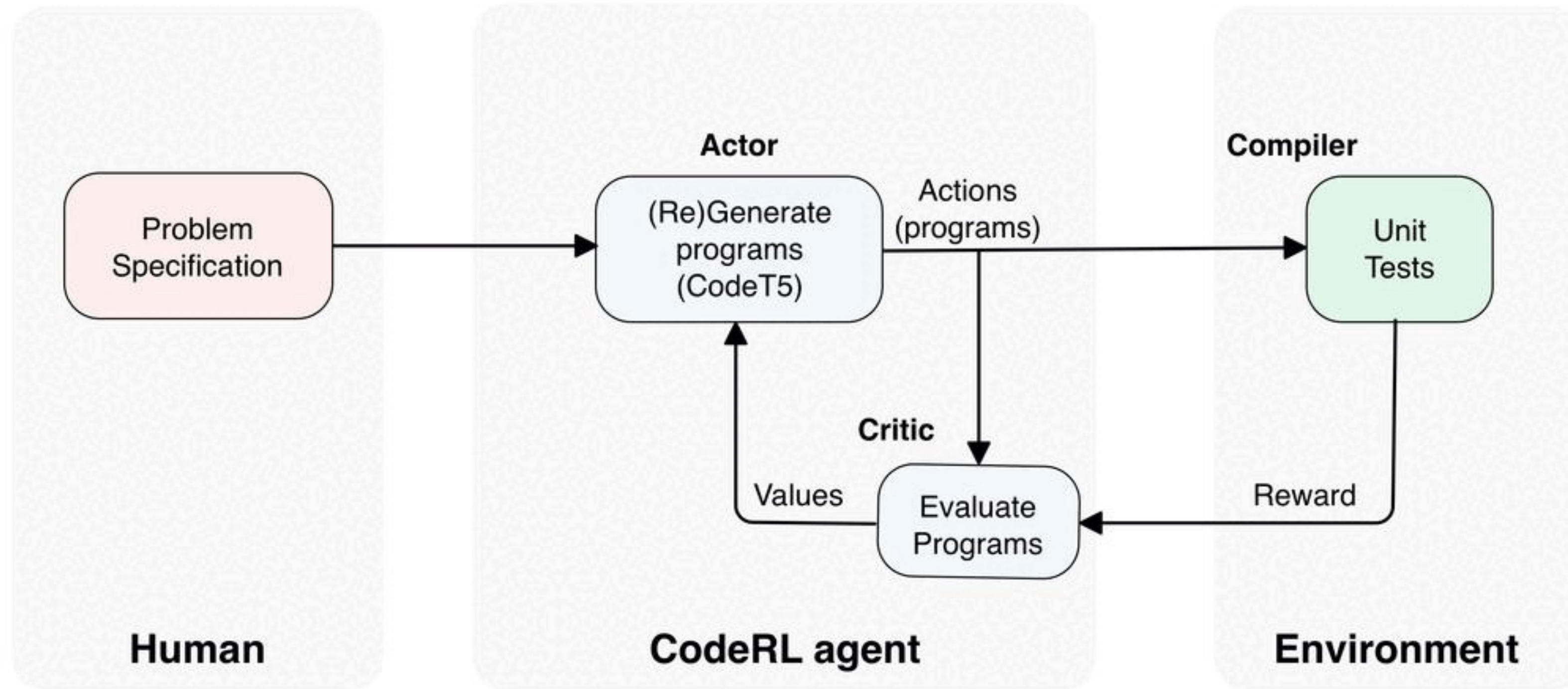
$$\hat{q}_\phi(w_t^s) = \hat{v}_t[u]$$

$$\hat{v}_t = \text{softmax}(\text{Linear}(h_t))$$

CodeRL à l’entraînement



CodeRL à l'entraînement



CodeRL à l'inférence – *Critic Sampling*

- Pour chaque problème, à l'inférence, on génère N programmes potentiels. On note $\mathcal{P}$ les programmes qui passent les tests unitaires contenus dans la spécification et $\mathcal{F}$ les programmes qui ne les passent pas.

- De la même manière qu'à l'entraînement, on cherche à améliorer la génération de nos programmes dans le cas où ils réussissent les tests unitaires d'exemples et dans le cas où ils ne le font pas.

- Le *critic sampling* se décompose en deux méthodes :
  - *Program refining* pour les programmes $\mathcal{F}$.
  - *Program repairing* pour les programmes $\mathcal{P}$.

CodeRL à l'inférence – *Program Refining*

- De la même manière qu'à l'entraînement, on entraîne un *critic model* ϕ_{test} dans le cadre de classification binaire $\{\text{FailedTest}, \text{PassedTest}\}$.

- Pour chaque étape *t*, on utilise notre *critic model* pour calculer la probabilité que la sous-séquence de *token* de *[1:t]* passe le test final :

$$\hat{q}_{\phi_{\text{test}}}(w_t) = p_{\phi_{\text{test}}}(\hat{u} = \text{PassedTest}|w_{1:t}, D)$$

- Si la séquence finale contient un *token* tel que $p_{\phi_{\text{test}}}(\text{FailedTest}) > p_{\phi_{\text{test}}}(\text{PassedTest})$, on sélectionne tous les *tokens* à sa droite dont on se sert comme *prompt* pour un autre essai.

CodeRL à l'inférence – *Program Repairing*

- Dans le cas où $|\mathcal{F}| = N$, on utilise un processus préliminaire de *program repairing* avant de pouvoir utiliser le *program refining.*

- On utilise le même *critic model* ϕ_{test} pour déterminer les candidats de $\mathcal{F}$ avec le plus de chance d'être « réparé » :

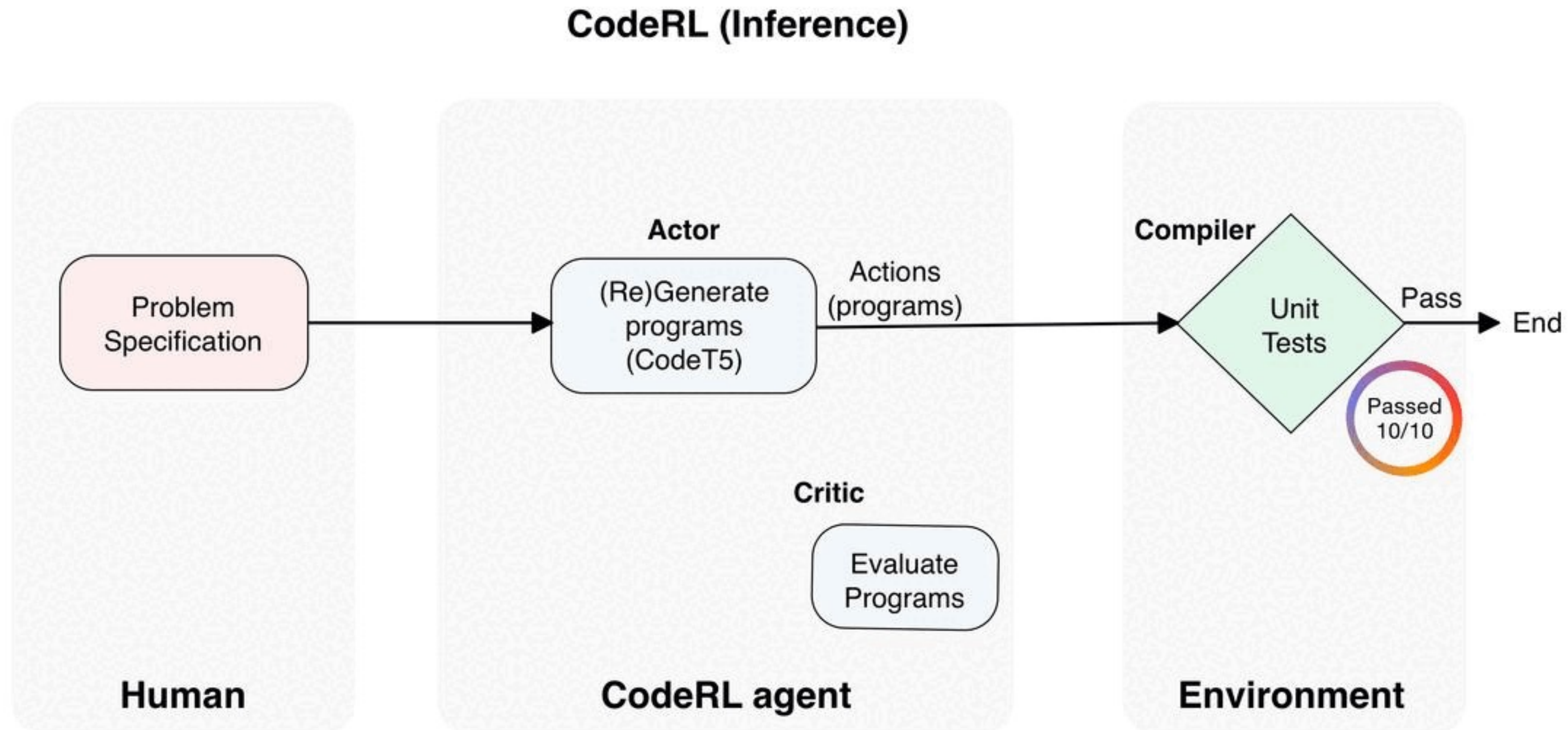
$$\hat{q}_{\phi_{\text{test}}}(W^{\text{fail}}) = p_{\phi_{\text{test}}}(\hat{u} = \text{PassedTest}|W^{\text{fail}}, D)$$

- On utilise un *seq2seq* pour corriger les programmes qu'on entraîne séparément avec les codes erronés prédit à l'entraînement :

$$\mathcal{L}_{ce}^{\text{repair}}(\omega) = -\sum_t \log p_\omega(W|D, W^{\text{fail}}, u, c) = -\sum_t \log[p_\omega(w_t|w_{1:t-1}, D, W^{\text{fail}}, u, c)]$$

avec $u \in \{\text{CompileError}, \text{RuntimeError}, \text{FailedTest}, \text{PassedTest}\}$ et c le sous-type de l'erreur.

CodeRL à l'inférence



Les étapes en 1 exemple

Input Problem

QUESTION:
Holidays have finished. Thanks to the help of the hacker Leha, Noora managed to enter the university of her dreams which is located in a town Pavlopolis. It's well known that universities provide students with dormitory for the period of university studies. Consequently Noora had to leave Vičkopolis and move to Pavlopolis. Thus Leha was left completely alone in a quiet town Vičkopolis. He almost even fell into a depression from boredom!

Leha came up with a task for himself to relax a little. He chooses two integers A and B and then calculates the greatest common divisor of integers "A factorial" and "B factorial". Formally the hacker wants to find out GCD(A!, B!). It's well known that the factorial of an integer x is a product of all positive integers less than or equal to x. Thus x! = 1·2·3·...·(x - 1)·x. For example 4! = 1·2·3·4 = 24. Recall that GCD(x, y) is the largest positive integer q that divides (without a remainder) both x and y.

Leha has learned how to solve this task very effective. You are able to cope with it not worse, aren't you?

-----Input-----

The first and single line contains two integers A and B (1 ≤ A, B ≤ 10^9, min(A, B) ≤ 12).

-----Output-----

Print a single integer denoting the greatest common divisor of integers A! and B!.

ANSWER:

Example Tests

```
Input:   4 3
Output:  6
```

Hidden Tests

```
Input:   10 399603090
Output:  3628800

Input:   5 5
Output:  120
```

Reference Program

```
1 a, b = map(int, input().split())
2 a = min(a, b)
3 ans = 1
4 for i in range(1, a + 1):
5     ans *= i
6 print(ans)
```

CodeT5 Output Program

```
1 from math import gcd
2 a, b = map(int, input().split())
3 c = math.gcd(a,b)
4 print(c)
```

Example tests: ✘
Hidden tests: ✘

Interpret the problem wrongly

CodeRL+CodeT5 Output Program (before CS)

```
1  from math import gcd
2  a, b = list(map(int, input().split()))
3  gcd = lambda x, y: x if y == 0 else gcd(y, x % y)
4
5  a_prod = 1
6  b_prod = 1
7  for i in range(1, a + 1):
8      a_prod *= i
9  for i in range(1, b + 1):
10     b_prod *= i
11
12 print(gcd(a_prod, b_prod))
```

Example tests: ✔
Hidden tests: ✘ (TimeOut)

Good Functional Correctness but timeout in test cases with extreme values of *b*

CodeRL+CodeT5 Output Program (after CS)

```
1  from math import gcd
2  a, b = list(map(int, input().split()))
3  gcd = lambda x, y: x if y == 0 else gcd(y, x % y)
4
5  a_prod = 1
6  b_prod = 1
7  for i in range(1, min(a, b) + 1):
8      a_prod *= i
9      b_prod *= i
10
11 print(gcd(a_prod, b_prod))
```

Example tests: ✔
Hidden tests: ✔

More efficient program by using *min(a,b)* instead

Conclusion

(a) Performance by *pass@k* with $k = \{1, 5, 1000\}$

Model	Size	*pass@1*				*pass@5*				*pass@1000*			
		Intro	Inter	Comp	All	Intro	Inter	Comp	All	Intro	Inter	Comp	All
Codex	12B	4.14	0.14	0.02	0.92	9.65	0.51	0.09	2.25	25.02	3.70	3.23	7.87
AlphaCode	1B	-	-	-	-	-	-	-	-	17.67	5.24	7.06	8.09
GPT3	175B	0.20	0.03	0.00	0.06	-	-	-	-	-	-	-	-
GPT2	0.1B	1.00	0.33	0.00	0.40	2.70	0.73	0.00	1.02	-	-	-	-
GPT2	1.5B	1.30	0.70	0.00	0.68	3.60	1.03	0.00	1.34	25.00	9.27	8.80	12.32
GPT-Neo	2.7B	3.90	0.57	0.00	1.12	5.50	0.80	0.00	1.58	27.90	9.83	11.40	13.76
GPT-J	6B	5.60	1.00	0.50	1.82	9.20	1.73	1.00	3.08	35.20	13.15	13.51	17.63
CodeRL+CodeT5	770M	**7.08**	**1.86**	**0.75**	**2.69**	**16.37**	**4.95**	**2.84**	**6.81**	**40.00**	**15.67**	**17.90**	**20.98**

(b) Performance by $n@k$ with k up to 50000 and $n = \{1, 5\}$

Model	Size	k	*1@k*				*5@k*			
			Intro	Inter	Comp	All	Intro	Inter	Comp	All
Codex	12B	1000	**22.78**	2.64	3.04	6.75	24.52	3.23	3.08	7.46
AlphaCode	1B	1000	-	-	-	-	14.36	5.63	4.58	7.17
AlphaCode	1B	10000	-	-	-	-	18.18	8.21	6.65	9.89
AlphaCode	1B	50000	-	-	-	-	20.36	**9.66**	7.75	11.42
CodeRL+CodeT5	770M	1000	17.17	**6.78**	**4.88**	**8.48**	**25.61**	9.53	**8.91**	**12.62**