L'IA dans la synthèse de programmes

Formulée par les pionniers de l'intelligence artificielle dès les années 1950, la synthèse de programmes est l'un des plus anciens rêves du domaine. Le but ? Automatiser la résolution de tâches à l'aide d'algorithmes. Dit autrement, il s'agit de transformer de manière automatique la description d'un problème en sa solution sous la forme d'un algorithme. L'utilisation de modèles de langues massifs, apparus ces dernières années, est devenue incontournable en la matière.

CREDIT CREDIT

Nathanaël Fijalkow
INFORMATICIEN, BORDEAUX
Chercheur CNRS spécialiste de théorie des jeux et de synthèse de programmes, il mène ses recherches au laboratoire bordelais de recherche en informatique (CNRS – université de Bordeaux – ENSEIRB).

Un problème ancien. Le défi algorithmique que pose la synthèse de programmes informatiques peut se formuler ainsi : étant donné un ensemble de programmes, comment trouver dans cet ensemble un programme satisfaisant des spécifications ? L'approche par énumération, qui a d'abord prévalu, consiste à énumérer un par un les programmes et à vérifier pour chacun s'il satisfait les spécifications. Simple à mettre en œuvre, elle a ensuite été supplantée par la résolution de contraintes, qui cherche à écrire un programme comme un ensemble de contraintes logiques. L'utilisation pratique de ces approches était limitée. Mais personne ne s'attendait à la révolution qu'allait apporter l'apprentissage automatique par réseaux de neurones.

Lorsque sort la publication présentant une nouvelle architecture de réseaux de neurones fondée sur un mécanisme d'attention – les transformeurs –, l'objectif initial était de résoudre des tâches textuelles, ce qu'on appelle *Seq2seq* dans le jargon, c'est-à-dire des problèmes qui prennent en entrée un texte et produisent en sortie un autre texte. Ce n'est pourtant pas à cela qu'elle a été employée au départ : l'architecture transformeur a d'abord été utilisée pour le traitement automatique des langues et, en particulier, la traduction automatique, où elle a fait preuve d'une grande efficacité (les mécanismes d'attention permettent en effet de localiser dans une phrase les liens sémantiques entre différents éléments, le cœur de la difficulté de la traduction).

Partant de là, une idée a germé : si cette architecture est aussi performante pour traduire du français vers l'anglais par exemple, pourquoi ne pourrait-elle pas être mise à profit pour générer automatiquement du code informatique ou, pour le dire autrement, faire de la synthèse de programmes ? Après tout, dans les deux cas, il s'agit d'une « langue » : d'un côté, des langues naturelles (français, anglais, etc.) et, de l'autre, des langages de programmation (Python, C, etc.). La synthèse de programmes est longtemps restée un fantasme confiné aux laboratoires de recherche : malgré des décennies de recherche, les applications pratiques étaient limitées. Une exception notable, cependant, est l'outil FlashFill, inclus dans le tableur Microsoft Excel depuis 2013. Il permet d'automatiser l'écriture de ce qu'on appelle les « macros », un petit programme qui réalise des tâches répétitives. Par exemple, pour créer une nouvelle colonne dans une feuille Excel à partir de colonnes existantes, on écrit à la main quelques exemples, et on laisse à FlashFill le soin de généraliser toute la colonne. De manière déjà impressionnante – à l'époque –, FlashFill remplissait la colonne correctement dans la majorité des cas, évitant aux utilisateurs d'écrire laborieusement à la main les macros correspondantes. D'un point de vue technique, FlashFill diffère des outils récents de synthèse de programmes, dans le sens où il n'utilise pas d'apprentissage automatique.

C'est bien la nouvelle architecture transformeur, appliquée à l'apprentissage automatique, qui a déclenché la deuxième vague de la synthèse de programmes. L'outil Copilot, disponible depuis juin 2021, illustre parfaitement le phénomène. La société qui l'a publié, OpenAI – la même qui a sorti ChatGPT –, l'a conçu comme une aide à la conception de programmes informatiques, qui peut être utilisée de plusieurs manières. L'une d'elles consiste, pour le développeur, à décrire avec des mots (en langage naturel) ce qu'il veut que fasse le programme ; Copilot se sert alors de ce texte pour produire une trame de programme informatique. Le développeur peut aussi écrire le squelette d'un programme en code informatique ; Copilot se charge alors de combler les trous. L'idée générale, dans un cas comme dans l'autre, est d'éviter à l'humain les tâches les plus répétitives, et de lui faire gagner du temps.

L'outil Copilot, s'appuyant sur l'apprentissage automatique, épargne à l'humain les tâches les plus répétitives

Quel avenir pour les développeurs ?

Les progrès en synthèse de programmes pourraient donner l'impression que l'IA va remplacer, et donc rendre inutile, le métier de développeur. Rien n'est moins sûr. Même avec les progrès spectaculaires des LLM, on est très loin d'avoir un outil à même de remplacer l'humain. Comme OpenAI l'avait suggéré, il faut voir le LLM comme un compagnon, ou un copilote : le développeur possède maintenant un outil lui permettant de gagner du temps sur des tâches routinières. Mais il y a deux écueils importants : d'abord, ce « copilote » ne peut pas se substituer au développeur pour une partie essentielle, à savoir la formalisation du problème, la structuration du code, le choix des structures de données. En bref, la réflexion repose entièrement sur les épaules du développeur. Par ailleurs, les LLM ne peuvent – du moins pour le moment – produire que des programmes courts et suffisamment précis : au-delà de quelques dizaines de lignes apparaissent des phénomènes d'hallucination, la qualité du code produit se dégrade. Et même si (ou plutôt, quand) les futures générations de LLM s'attaquent également à ces aspects, il restera une question essentielle : le code généré résout-il correctement le problème posé ? L'expertise et la validation d'un humain resteront la clé de voûte du développement, même quand la performance de l'outil rendra ces tâches plus faciles. N.F.

Copilot est un modèle de langue – un LLM – et, comme les autres, est entraîné sur une quantité colossale de données. Pour l'application à la synthèse de programmes, ces LLM sont entraînés à combler des trous dans des programmes trouvés sur Internet.

LIMITE MÉTHODOLOGIQUE, LIMITE STRUCTURELLE ET…

Si avec l'arrivée de ces nouveaux outils, on assiste à un nouveau paradigme de programmation qui est sans nul doute en train de changer en profondeur le travail des développeurs (lire l'encadré ci-dessus), il ne faut pas perdre de vue leurs limites. La première est méthodologique : le LLM qui se trouve au cœur des outils comme Copilot traite les langages de programmation comme des langues naturelles, ou, dit autrement, le code comme du texte libre, sans aucune limitation sur la syntaxe. Plus concrètement : dans la plupart des langages de programmation, il y a des balises ouvrantes et des balises fermantes, qui permettent de délimiter un ensemble d'instructions, par exemple <h1> et </h1> définissent un titre dans le langage HTML [qui permet de structurer une page Web]. Or, il arrive à Copilot de produire des programmes ne respectant pas ces conventions de syntaxe extrêmement simples. Plus généralement, attacher au code sa structure est la condition *sine qua non* pour raisonner sur ce code, prouver ses propriétés, l'optimiser, et même le comprendre. Par exemple, les environnements de développement – logiciels aidant le développeur dans la production de code – s'appuient sur la structure du code pour l'afficher et l'éditer de manière optimisée. En oubliant la structure du code, les LLM se privent des enseignements de dizaines d'années de recherche en langages de programmation.

… DES CODES AUX EFFETS PARFOIS INDÉSIRABLES

Ce premier écueil entre en résonance avec le second : les LLM qui engendrent du code n'offrent aucune garantie, ni de correction ni de sécurité. En effet, il est impossible d'affirmer que le code produit répond effectivement aux attentes du développeur. Ceci s'interprète à plusieurs niveaux : soit le programme a mal interprété l'attente du développeur et répond à un problème différent, soit le programme ne peut pas être exécuté parce qu'il contient des erreurs, ou pire, l'exécuter pourrait avoir des conséquences néfastes. Ce dernier point est le plus inquiétant, car il existe sur Internet de multiples exemples de codes ayant des effets indésirables – accès mémoire illégaux, modifications de variables externes, etc. Les LLM reproduisent ce qu'ils ont vu au cours de l'entraînement : rien ne garantit qu'ils ne puissent pas reproduire ces effets. Se pose alors cette question : dans quelle mesure le développeur vérifie-t-il le code engendré ? Une autre problématique, légale cette fois, concerne la licence du code généré : encore une fois, rien ne garantit que les LLM ne produisent pas de code sous licence (commerciale ou non).

D'autres techniques, dites neurosymboliques, permettent d'éviter de créer des programmes contenant des erreurs et des bugs

Il existe une autre limitation encore à la synthèse de programmes par LLM : par définition, ces modèles nécessitent de très grands volumes de données, qui ne sont pas toujours disponibles. Ainsi, les LLM sont très performants quand il s'agit de générer des fonctions auxiliaires classiques en Python, parce qu'Internet fourmille d'exemples et de solutions. Mais qu'en est-il des langages de programmation moins populaires ? Par exemple, de nombreuses industries utilisent ceux qu'elles ont créés elles-mêmes (on parle de langage propriétaire), et donc pour lesquels il n'existe pas le même volume de données. Une solution à ce dernier problème consiste à faire des traductions d'un langage moins usité à un plus répandu. C'est le sens de l'outil Granite, proposé par IBM fin 2023. S'il fonctionne de la même manière que Copilot ou d'autres générateurs de code, sa particularité est de proposer la traduction du COBOL – un langage de programmation inventé dans les années 1950, spécialisé dans la gestion, et encore très utilisé dans les banques – vers le Python. Cela pourrait résoudre le problème de la recherche de programmeurs connaissant COBOL, devenus très rares…

Quel futur pour la synthèse de programmes ? L'immense bond en avant des LLM n'efface pas les autres progrès faits en recherche dans le domaine. En effet, depuis une petite dizaine d'années, les techniques dites « neurosymboliques » ont également permis de grandes avancées pratiques (1). Le terme désigne des techniques combinant à la fois de l'apprentissage par réseaux de neurones avec des algorithmes de raisonnement ou de recherche symbolique. L'objectif est d'avoir des garanties beaucoup plus fortes sur le code engendré, en s'appuyant sur du raisonnement logique : ceci permet d'éviter de créer des programmes contenant des erreurs ou des bugs. Dans un sens, les LLM sont un extrême : ce sont des techniques purement neuronales, où le LLM engendre directement le programme. L'autre extrême, purement logique, n'a pas permis de développer des outils utilisables en pratique (à l'exception de FlashFill). Les techniques neurosymboliques découpent le problème en deux : d'abord, le réseau de neurones lit la spécification et fait des prédictions sur le programme à créer, qui sont ensuite mises à profit par un algorithme de raisonnement ou de recherche. En introduisant une composante logique, ces méthodes donnent des garanties théoriques sur le programme engendré, tout en profitant de la puissance de prédiction des réseaux de neurones.

DREAMCODER ET DEEPSYNTH, MUNIS DE GARANTIES

Les approches neurosymboliques sont très populaires à la fois dans le monde académique et industriel. L'outil DreamCoder, proposé en 2021 par des informaticiens du MIT (2), ainsi que DeepSynth, que nous avons mis au point avec des chercheurs de mon laboratoire et disponible depuis juin 2022 (3), s'appuient sur les techniques développées en théorie des langages de programmation, en méthodes formelles, en apprentissage automatique et en algorithmique. On peut donc espérer que la brèche ouverte par les LLM en synthèse de programmes amènera à la construction d'outils fiables en lesquels les utilisateurs pourront avoir pleinement confiance, dotés en particulier de garanties théoriques. ■

POUR EN SAVOIR PLUS

■ Le cours en ligne sur la synthèse de programmes d'Armando Solar-Lezama, professeur au MIT, est une mine d'informations (en anglais) : https://people.csail.mit.edu/asolar/SynthesisCourse/TOC.htm

(1) S. Chaudhuri *et al.*, *Foundations and Trends in Programming Languages*, *7*, 158, 2021.

(2) K. Ellis *et al.*, PLDI 2021, doi : 10.1145/3453483.3454080, 2021.

(3) N. Fijalkow *et al.*, *Proceedings of the AAAI Conference on Artificial Intelligence, 36*, 6623, 2022.