Sujet de thèse CIFRE : Génération de code métier Python à partir d'une description en langage naturel

Nathanaël Beau

Université de Paris et groupe Onepoint
n.beau@groupeonepoint.com

1 Contexte scientifique et motivation.

Converser avec un machine est l'un des fantasmes de l'humanité depuis la création de l'ordinateur. Dès 1950, Turing publiait son célèbre article [1] testant la faculté d'une machine à imiter une conversation humaine. Le développement récent du traitement automatique du langage naturel offre des perspectives prometteuses dans ce domaine.

La génération de code par une machine à partir d'une description textuelle constitue un pas dans cette direction. Par ailleurs, dans un contexte d'aide à la formalisation, la réalisation d'un algorithme permettant de générer du code aurait une utilité directe :

— pour des personnes non expertes, dans la réalisation complète de programmes (pour des langages ou des demandes simples dans un langage de requête comme SQL [2]))

— pour des programmeurs, comme un créateur de patron approximatif à enrichir (pour des langages généralistes comme Python)

L'hypothèse générale que l'on souhaite étudier est de déterminer dans quelle mesure il est possible pour une **machine de créer un programme conforme à une requête exprimée en langage naturel.**

On peut considérer ce problème comme un travail de traduction classique (d'un langage humain vers un langage formel) en notant qu'un programme informatique comporte des contraintes syntaxiques fortes.

L'état de l'art en traitement automatique des langues repose considérablement sur les méthodes d'apprentissage profond. Toutefois celles-ci comportent des limites pour le sujet envisagé. La première est l'absence de garantie sur la correction du code qui serait engendré. La seconde est l'absence de données d'entraînement, alignant langue naturelle et langage formel, en quantité pour permettre d'entraîner un tel système.

Nous présenterons tout d'abord les objectifs visés par la thèse, ensuite nous décrirons l'état de l'art et les limites rencontrées par la communauté, enfin nous expliciterons les approches envisagées pour la thèse.

2 Contribution souhaitée de la thèse.

Python est aujourd'hui le second langage informatique le plus utilisé au monde. Ce langage complexe et généraliste permet d'aborder une multitude de domaines d'applications (datascience, web, jeux vidéos. . .). Dans le cadre de ce doctorat, le but

serait de **générer des fragments de code du langage Python dans un contexte métier précis à partir d'une courte description en langage naturel**. On cible plusieurs objectifs principaux :

1. trouver un moyen d'engendrer un programme
   (a) syntaxiquement correct compte tenu du langage Python
   (b) conforme par rapport à la description en langage naturel
   (c) d'une longueur et d'une complexité maîtrisée
   (d) intelligible par l'utilisateur pour qu'il puisse le modifier
2. créer un mécanisme d'amélioration incrémental des solutions. L'algorithme pourrait par exemple proposer plusieurs "snippets" de code et l'utilisateur sélectionnerait une des solutions
3. Résoudre le problème des jeux d'entraînements disponibles
   (a) en créant des données labélisées
   (b) en limitant la taille des jeux de données à utiliser

3 Etat de l'art.

Les récents progrès en traitement automatique du langage présentent de nouvelles possibilités pour répondre à ces objectifs.

Pour minimiser le besoin de grandes quantités de données en lien avec une tâche applicative, citons par exemple l'invention de BERT [3] qui s'appuie sur des méthodes d'apprentissage profond avec attention [4] pour réaliser un préentraînement qui permet d'obtenir une représentation vectorielle des mots. L'architecture, du Transformer, sur laquelle il est basé, est largement utilisée pour traiter des problèmes de traduction aujourd'hui et le préentraînement permet d'obtenir de meilleurs résultats avec moins de données. Par exemple, les chercheurs de [5] utilisent cette architecture pour générer un arbre de syntaxe abstraite, structure intermédiaire entre le langage naturel et le code, facilement transformable en code Python. De tels arbres sont engendrés par une grammaire formelle, ce qui permet de controler l'espace des possibilités de génération de code en spécifiant explicitement la grammaire du langage formel ainsi visé [6]. Ces approches atteignent pourtant des limites lorsque la structure des données engendrées se complexifie ou encore pour sélectionner des noms de variables (qu'il est difficile d'identifier dans la description en langage naturel). A l'heure actuelle, l'évaluation des performances de génération de code Python à partir de langue naturelle est possible grâce à deux jeux de données : le *CoNaLa dataset* [7] (récupération de questions de programmation informatique exprimées en langage naturel associées à leurs réponses en Python provenant du célèbre site Stack Overflow) et le plus récent *CodeSearchNet dataset* [8], prometteur par la quantité d'exemples d'entraînement qu'il propose (utilisation de docstrings issus de Github associés à leurs fonctions Python). On note la possible utilisation d'un autoencodeur variationnel [9] pour obtenir plus de données grâce à l'apprentissage semi-supervisé.

Pour des langages moins généralistes, les résultats de génération de requêtes SQL sont très bons [10] grâce à des jeux de données plus conséquents comme *WikiSQL* [11].

On peut relever d'autres techniques novatrices pour la génération de code d'un langage dédié, plus simple que Python ou SQL. Par exemple, l'architecture de réseau de neurones RobustFill [12] qui crée un programme généralisant une série d'exemples donnés en input. Dans ce cas, il s'agit d'effectuer des transformations Excel élémentaires et il faut un nombre minimum d'exemples pour générer un programme fonctionnel (pour éviter un problème de surapprentissage empêchant la généralisation). On peut aussi citer l'idée qui consiste à préentraîner un réseau de neurones grâce à des exemples simples en stockant les fonctions apprises lors de cette étape dans un cache permettant de les réutiliser [13]. Ces exemples doivent être annotés à la main et, pour un langage comme Python, leur construction constituerait un problème en soi qu'on essaie autant que possible d'éviter.

4 Méthodes envisagées

Dans les travaux cités en références on peut identifier deux grandes approches pour traiter le problème de génération de code.

En premier lieu, le recours à une grammaire formelle explicite apparaît être un élément méthodologique incontournable. Les règles insérées dans cette structure intermédiaire pourraient être améliorées, par exemple, au moyen de règles logiques destinées à structurer le le réseau de neurones [14] La dépendance entre un langage de programmation et sa grammaire doit être formalisée en amont du problème d'apprentissage. Cela permettra d'obtenir un code syntaxiquement correct et de se concentrer sur sa conformité à la description en langage naturel.

Une approche complémentaire serait l'entraînement d'un réseau de neurones sur des tâches génériques (comme effectué par BERT avec de la prédiction de la prochaine phrase ou du prochain mot sur Wikipédia) pour notre tâche de génération de code [15]. On dispose d'un nombre important de données open-source sur Github pouvant être utilisées pour préentraîner un réseau de neurones à l'apprentissage de la syntaxe du code. Il nous faudra étudier à la fois la structure précise (domaine métier, nombre de données) des données à utiliser pour effectuer ce préentraînement et ce que l'on souhaite faire prédire à l'algorithme : du code directement ou des arbres.

D'autres techniques, plus pratiques, peuvent améliorer les résultats lorsqu'on les associe aux approches précédentes. Donner des exemples correspondant au résultat final produit par l'algorithme pour renforcer les prédictions faites par le langage naturel constitue une technique intéressante. Dans un premier temps comme validation des différents codes générés après la description en langage naturel. Dans un second temps comme un mécanisme à part entière de génération de fragments de code, combiné aux prédictions de code obtenues grâce au langage naturel.

Finalement, on pourra aussi étudier la mise en oeuvre de méthodes de préentrainement d'un réseau de neurones à l'aide d'exemples algorithmiques simplifiés (parcours d'une liste, suppression ou modification d'un élément...). C'est une technique performante et en principe facile à mettre en oeuvre. Elle permettrait de stocker certaines fonctions de base pour les réutiliser, réduisant le nombre de données nécessaires et facilitant la construction de programmes plus complexes.

5 Chronologie des livrables des travaux de thèse

D'après les méthodes envisagées, nous proposons le calendrier prévisionnel suivant :

— Une appropriation des méthodes de l'état de l'art durant la première année :

1. mise en place de l'architecture TranX [16] qui offre les meilleurs résultats sur le *CoNaLa dataset*.
2. tentative d'amélioration des résultats à l'aide d'un entraînement semi-supervisé de l'architecture de TranX
3. synthèse des résultats dans un article
4. veille bibliographique continue

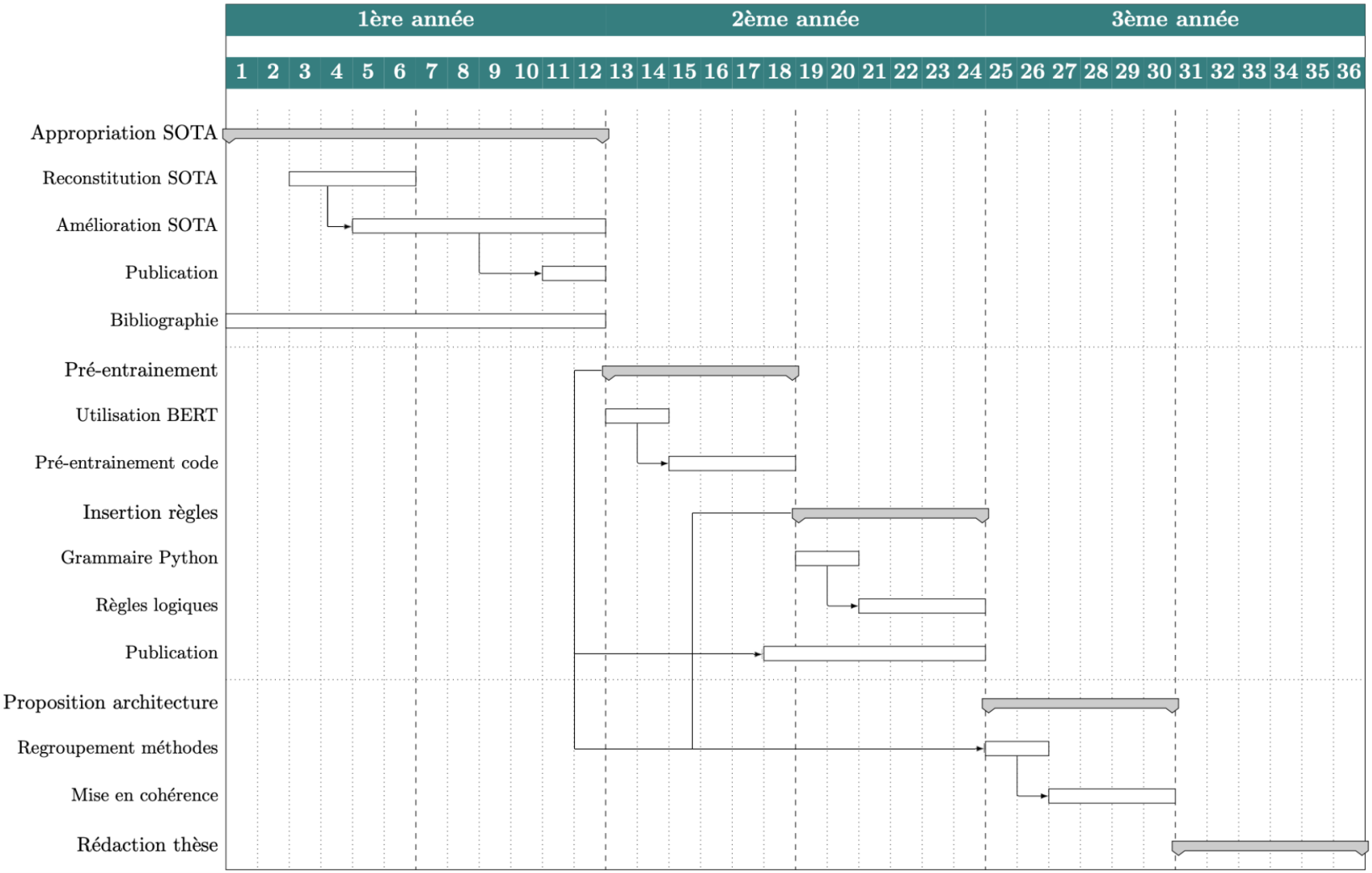
— Mise en place des techniques imaginées en amont de la thèse durant la seconde année :

1. utilisation de BERT pour obtenir une meilleure représentation du langage naturel
2. pré-entraînement (comme BERT) sur des tâches spécifiques au code (sur des arbres par exemple)
3. utilisation de règles à intégrer au réseau de neurones (par exemple pour obtenir du code dit "Pythonic")
4. synthèse des résultats dans un article

— Proposition d'une architecture globale reprenant les résultats obtenus et rédaction de la thèse durant la troisième année :

1. synthèse des résultats obtenus précédemment
2. réflexion sur la possibilité d'associer les différentes méthodes
3. mise en place de l'architecture finale
4. rédaction de la thèse

Ces points sont illustrés dans le diagramme de Gantt présenté à la page suivante.


1ère année
2ème année
3ème année
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
Appropriation SOTA
Reconstitution SOTA
Amélioration SOTA
Publication
Bibliographie
Pré-entrainement
Utilisation BERT
Pré-entrainement code
Insertion règles
Grammaire Python
Règles logiques
Publication
Proposition architecture
Regroupement méthodes
Mise en cohérence
Rédaction thèse

Bibliographie

[1] A. M. Turing. Computing machinery and intelligence. *Mind 49 : 433-460.*, 1950.

[2] Nan Duan Ming Zhou Daya Guo1, Duyu Tang and Jian Yin. Dialog-to-action : Conversational question answering over a large-scale knowledge base. *Advances in Neural Information Processing Systems 31, pages 2946–2955*, 2018.

[3] Chang Kenton Jacob Devlin, Ming-Wei and Lee Kristina Toutanova. Bert : Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv :1810.04805*, 2018.

[4] Niki Parmar Jakob Uszkoreit Llion Jones Aidan N Gomez Lukasz Kaiser Ashish Vaswani, Noam Shazeer and Illia Polosukhin. Attention is all you need. *In Advances in Neural Information Processing Systems, pages 6000–6010.*, 2017.

[5] Mitchell Stern Maxim Rabinovich and Dan Klein. Abstract syntax networks for code generation and semantic parsing. *In Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2017.

[6] Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. *In The 55th Annual Meeting of the Association for Computational Linguistics*, 2017.

[7] Edgar Chen Bogdan Vasilescu Pengcheng Yin, Bowen Deng and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. *In International Conference on Mining Software Repositories, MSR, pages 476–486.*, 2018.

[8] Tiferet Gazit Miltiadis Allamanis Hamel Husain, Ho-Hsiang Wu and Marc Brockschmidt. Codesearchnet challenge : Evaluating the state of semantic code search. *arXiv preprint arXiv :1909.09436*, 2019.

[9] Junxian He Pengcheng Yin, Chunting Zhou and Graham Neubig. Tree-structured latent variable models for semi-supervised semantic parsing. *In Proceedings of ACL.*, 2018.

[10] Kaushik Chakrabarti Pengcheng He, Yi Mao and Weizhu Chen. X-sql : reinforce context into schema representation. *Microsoft Dynamics 365 AI*, 2019.

[11] Caiming Xiong Victor Zhong and Richard Socher. Seq2sql : Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv :1709.00103*, 2018.

[12] Surya Bhupatiraju Rishabh Singh Abdel-rahman Mohamed Jacob Devlin, Jonathan Uesato and Pushmeet Kohli. Robustfill : Neural program learning under noisy i/o. *arXiv preprint arXiv :1703.07469*, 2017.

[13] Udi Naveh Amir Globerson Omer Goldman, Veronica Latcinnik and Jonathan Berant. Weakly supervised semantic parsing with abstract examples. *In Proceedings of the Annual Meeting of the Association for Computational Linguistics, pages 1809–1819.*, 2018.

[14] Tao Li and Vivek Srikumar. Augmenting neural networks with first-order logic. *arXiv preprint arXiv :1906.06298*, 2019.

[15] Duyu Tang Nan Duan Xiaocheng Feng Ming Gong Linjun Shou Bing Qin Ting Liu Daxin Jiang Zhangyin Feng, Daya Guo and Ming Zhou. Codebert :

A pre-trained model for programming and natural languages. *arXiv preprint arXiv :2002.08155*, 2020.

[16] Pengcheng Yin and Graham Neubig. Tranx : A transition-based neural abstract syntax parser for semantic parsing and code generation. *In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing : System Demonstrations, pages 7–12. Association for Computational Linguistics.*, 2018.