POLYPUBLIE
Polytechnique Montréal

POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

Titre: Title:	Génération automatique de requêtes SPARQL à partir de questions en langue naturelle
Auteur: Author:	Rose Hirigoyen
Date:	2022
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Hirigoyen, R. (2022). Génération automatique de requêtes SPARQL à partir de questions en langue naturelle [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/10533/

Document en libre accès dans PolyPublie
Open Access document in PolyPublie

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/10533/
Directeurs de recherche: Advisors:	Amal Zouaq
Programme: Program:	Génie informatique

Ce fichier a été téléchargé à partir de PolyPublie, le dépôt institutionnel de Polytechnique Montréal
This file has been downloaded from PolyPublie, the institutional repository of Polytechnique Montréal

https://publications.polymtl.ca

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Génération automatique de requêtes SPARQL à partir de questions en langue naturelle

ROSE HIRIGOYEN
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Août 2022

© Rose Hirigoyen, 2022.

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Génération automatique de requêtes SPARQL à partir de questions en langue naturelle

présenté par **Rose HIRIGOYEN**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Michel GAGNON, président
Amal ZOUAQ, membre et directrice de recherche
Christopher J. PAL, membre

DÉDICACE

Being smart will count for nothing if you don't make the world better.
–Elisabeth Sobeck

REMERCIEMENTS

Je tiens à remercier du fond du cœur ma directrice de recherche, Prof. Amal Zouaq, qui m'a poussée à me dépasser et à donner le meilleur de moi-même. Merci pour ses conseils, son temps et son exemple.

Merci énormément à mon amoureux Thierry pour sa patience et ses encouragements tout au long de ce travail. Merci d'avoir été toujours disponible pour discuter de pistes et d'idées, et de m'avoir aidée à trouver des solutions créatives à plusieurs problèmes. Je suis infiniment reconnaissante d'avoir la chance de le côtoyer et de pouvoir compter sur lui.

Merci à Adem, que j'ai la chance de compter parmi mes meilleurs amis. Merci pour les discussions toujours enrichissantes et les conseils pleins de sagesse qui m'ont débloquée à plusieurs reprises. Merci de sa capacité à faire ressortir le meilleur de mes idées, et merci pour son amitié fidèle.

Merci à Michel Gagnon, qui m'a indirectement initiée à la recherche en traitement de la langue naturelle lors de mon premier stage en première année, et qui a aujourd'hui accepté de juger ce travail.

Finalement, je suis reconnaissante de pouvoir compter sur ma famille et mes amis précieux, qui m'ont encouragée de près et de loin tout au long de ce travail. Je tiens à leur exprimer ma gratitude la plus complète pour leur soutien infaillible.

RÉSUMÉ

Les bases de connaissances sont des graphes de stockage de données dans lesquels les nœuds sont des concepts ou des objets, et les arêtes représentent les relations entre eux. Afin de tirer des informations de ces bases de connaissances, on utilise le langage de requête SPARQL, dont l'apprentissage représente un certain défi. Or, en considérant ce langage de requête comme une langue à part entière au même titre que l'anglais ou l'allemand, on peut appliquer les techniques de traduction automatique au problème de transformer une question en langue naturelle en une requête SPARQL équivalente. Cela permet aux utilisateurs d'obtenir des informations des bases de connaissances en utilisant simplement la langue naturelle. Les modèles de traduction automatique développés pour résoudre ce problème sont en ce moment très performants pour générer des requêtes syntaxiquement correctes, mais présentent d'importantes limitations lorsqu'il s'agit d'utiliser les bons éléments de la base de connaissances (sujet, propriété, objet). Ils génèrent alors des requêtes qui ne portent pas sur les mêmes éléments que la question, et donc qui retournent les mauvaises réponses. Cette lacune passe inaperçue, car le score BLEU, qui est la principale métrique utilisée pour évaluer les modèles, n'est qu'un indicateur de la similarité entre la traduction générée automatiquement et la traduction attendue. Autrement dit, une requête qui est bien formulée retourne un score BLEU relativement élevé, même si elle ne porte pas sur les bons éléments de la base de connaissances.

Notre travail tente donc de remédier à ces limitations en introduisant de meilleures manières d'évaluer la performance des modèles de traduction de la langue naturelle vers SPARQL, ainsi qu'en proposant une manière d'améliorer la capacité des modèles à comprendre les questions portant sur des éléments jamais vus. Pour ce faire, nous proposons deux contributions.

Tout d'abord, le gros problème des architectures de type *Seq2Seq*, qui sont à la base de la traduction automatique, est qu'elles ne peuvent prédire que des mots qu'elles ont déjà rencontrés. Cela signifie qu'un utilisateur qui pose une question hors de la portée du modèle de traduction n'obtiendra jamais la bonne réponse. Or, les informations dont le modèle a besoin pour retourner la bonne réponse se trouvent souvent dans la question. Notre hypothèse est donc que donner à notre modèle la capacité de copier des mots directement de la question lui permettra de ne plus être limité par son vocabulaire et de pouvoir retourner une réponse à n'importe quelle question formulée d'une manière qu'il reconnaît. Pour ce faire, nous générons une version de nos données dite annotée, dans laquelle les informations de la requête portant sur le sujet sont rendues explicites dans la question. Nous ajoutons aussi une couche de copie

à notre modèle, ce qui lui permet d'aller récupérer ces informations (classes, propriétés de la base de connaissances) afin de générer des requêtes utilisant ces bons éléments et ainsi retourner des réponses adéquates.

Ensuite, malgré le fait que nous utilisions une architecture de traduction automatique, notre système est essentiellement utilisé comme un système de questions-réponses. Aussi, le score BLEU, bien qu'important, ne nous donne aucune indication de la performance des modèles lorsqu'ils sont confrontés à une question portant sur des éléments jamais vus. Notre hypothèse est donc qu'en évaluant la précision des réponses ainsi que la performance d'un modèle spécifiquement sur des éléments inconnus, nous aurons une bien meilleure idée de ses capacités réelles. Nous utilisons donc une métrique de précision des réponses, que nous calculons en récupérant toutes les réponses des requêtes attendues et celles des requêtes traduites, et nous calculons la précision entre ces deux ensembles. Aussi, nous générons un ensemble de test qui contient spécifiquement des éléments de la base de connaissances que le modèle n'a jamais vus, puis nous calculons le score BLEU et la précision des réponses sur ce nouvel ensemble de test.

À travers notre recherche, nous avons également remarqué un certain manque d'uniformité et de nombreuses erreurs dans les données originales. Par conséquent, nous générons également une version des données homogénéisée et corrigée pour plusieurs jeux de données standards, notamment Monument et LC-QuAD v1.0. Cela rend les ensembles de données plus simples à utiliser pour les développements futurs, ce qui représente notre troisième contribution.

Finalement, en tant que partisanes de l'*open source* et de la reproductibilité, nous avons mis beaucoup d'effort pour que tout le matériel développé soit accessible à n'importe qui ayant accès à un ordinateur, sans avoir besoin de matériel particulier ou d'une licence pédagogique. Notre quatrième et dernière contribution importante est donc une base de code documentée et facilement exécutable avec *Google Colab*.

Les résultats obtenus démontrent clairement les avantages de notre approche. Nous améliorons les performances de l'état de l'art par 30 points pour la métrique du score BLEU sur certains modèles, et obtenons de manière consistante des résultats de score BLEU et de précision des réponses supérieures à 80 %. Les performances obtenues sur les ensembles de test contenant des éléments jamais vus nous permettent d'avoir une meilleure idée de l'utilisabilité et de la flexibilité réelles des modèles, et nous permettent également de conclure que l'utilisation de la copie nous permet d'entraîner des modèles plus flexibles et réutilisables.

ABSTRACT

Knowledge bases are data storage graphs in which the nodes are concepts or objects, and the edges represent the relationships between them. In order to extract information from these knowledge bases, the SPARQL query language is used, which can be challenging to learn. However, by considering this query language as a language in its own right, just like English or German, we can apply machine translation techniques to the problem of transforming a natural language question into an equivalent SPARQL query. This allows users to obtain information from knowledge bases simply by using natural language. Machine translation models developed to solve this problem are currently very good at generating syntactically correct queries, but have significant limitations when it comes to using the correct knowledge base elements (subject, property, object). Thus, they generate queries that do not use the same elements as the question, and therefore return the wrong answers. This shortcoming often goes unnoticed, because the BLEU-score, which is the main metric used to evaluate the models, is only an indicator of the similarity between the automatically generated translation and the reference translation. In other words, a query that is well formulated returns a relatively high BLEU-score, even if it is not about the right elements of the knowledge base.

Our work therefore attempts to address these limitations by introducing better ways to evaluate the performance of natural language translation models, as well as by proposing a way to improve the models' ability to understand questions. To this end, we propose two contributions.

First, despite the fact that we use a machine translation architecture, our system is essentially used as a question answering system. Also, the BLEU-score, although important, does not give us any indication of the performance of the models when confronted with a question about never seen items. Our hypothesis is therefore that by evaluating the accuracy of the answers as well as the performance of the model especially on unknown items, we will have a much better idea of the real capabilities of the models. To verify this, we introduce an answer accuracy metric, which we compute by retrieving all the answers of expected queries and those of translated queries, and we compute the accuracy between these two sets. Also, we generate a test set that specifically contains items from the knowledge base that the model has never seen, and then compute the BLEU-score and answer accuracy on that test set.

Second, the big problem with Seq2Seq-type architectures, which are primarily used in machine translation, is that they can only predict words they have already encountered. This means that a user who asks a question outside the scope of the translation model will never

get the right answer. However, the information that the model needs to return the correct answer is often in the question. Our hypothesis is therefore that giving our model the ability to copy words directly from the question will allow it to no longer be limited by its vocabulary and to be able to return an answer to any question formulated in a way it recognizes. To do this, we generate a version of our data called *tagged*, in which the information from the subject query is inserted into the question. We also add a copy layer to our model, which allows it to retrieve this information in order to return an answer about the right subject.

Through our research, we also noticed some inconsistency and errors in the original datasets. Therefore, we also generate a homogenized and corrected version of the data. This makes the datasets easier to use for future development, and this represents our third contribution.

Finally, as proponents of open source and reproducibility, we have put a lot of effort into making all the developed material accessible to anyone with access to a computer, without the need for special hardware or an educational license. Our fourth and last important contribution is therefore a documented and easily executable code built for *Google Colab*.

The results obtained clearly demonstrate the advantages of our approach. We improve the state of the art performance by 30 points on some models, and consistently obtain BLEU-score and answer accuracy results above 80%. The performance obtained on test sets containing never-before-seen items gives us a better idea of the actual usability and flexibility of the models, and also allows us to conclude that leveraging a copy layer enables to train more flexible and reusable models.

TABLE DES MATIÈRES

DÉDICACE . . . iii

REMERCIEMENTS . . . iv

RÉSUMÉ . . . v

ABSTRACT . . . vii

TABLE DES MATIÈRES . . . ix

LISTE DES TABLEAUX . . . xiii

LISTE DES FIGURES . . . xv

LISTE DES SIGLES ET ABRÉVIATIONS . . . xvii

LISTE DES ANNEXES . . . xviii

CHAPITRE 1 INTRODUCTION . . . 1
- 1.1 Terminologie . . . 2
  - 1.1.1 Bases de connaissances . . . 2
  - 1.1.2 Resource Description Framework (RDF) . . . 3
  - 1.1.3 SPARQL . . . 3
- 1.2 Éléments de la problématique . . . 4
  - 1.2.1 Intégration du schéma insuffisante . . . 5
  - 1.2.2 Incapacité des modèles à traiter des sujets inconnus . . . 5
  - 1.2.3 Métriques d'évaluation non représentatives . . . 6
  - 1.2.4 Ensembles de test incomplets . . . 6
  - 1.2.5 Manque d'uniformité des jeux de données . . . 7
  - 1.2.6 Difficulté de reproductibilité des résultats . . . 7
- 1.3 Objectifs de recherche . . . 7
- 1.4 Plan du mémoire . . . 8

CHAPITRE 2 REVUE DE LITTÉRATURE . . . 9
- 2.1 Définitions et concepts de base . . . 9
  - 2.1.1 Systèmes de questions-réponses . . . 9

2.1.2 Traduction automatique 10
2.1.3 SPARQL intermédiaire. 10
2.1.4 Patrons SPARQL 11
2.2 Transformation de questions vers des requêtes sparql 11
2.3 Traduction automatique 14
2.4 Stratégies utilisées en traduction vers SQL 16
2.5 Intégration du schéma 17
2.6 Méchanisme de copie dans d'autres tâches 19
2.7 Limitations 19
2.7.1 Intégration du schéma insuffisante 19
2.7.2 Incapacité de gérer des éléments inconnus 20
2.7.3 Métriques non représentatives 20
2.7.4 Ensembles de tests incomplets 21
2.7.5 Besoin d'uniformité à travers les ensembles de données 21
2.7.6 Difficulté de reproductibilité des résultats 21
2.7.7 Survol 22

CHAPITRE 3 ARCHITECTURES 23
3.1 Architecture générale 23
3.2 Architectures de référence 25
3.2.1 Modèle *Seq2Seq* convolutif (CnnS2S) 26
3.2.2 Modèle *Seq2Seq Transformer* 28
3.3 Architecture de copie 29
3.3.1 Vocabulaires 30
3.3.2 Couche de copie 32

CHAPITRE 4 DONNÉES ET MÉTRIQUES 34
4.1 Ensembles de données de l'état de l'art 34
4.1.1 Survol des données 34
4.1.2 Monument 35
4.1.3 LC-QuAD 37
4.2 Ensembles de données corrigés 39
4.2.1 Défauts dans les données de l'état de l'art 40
4.2.2 Corrections apportées 41
4.2.3 Encodage en SPARQL intermédiaire uniformisé 44
4.2.4 Survol des modifications 45
4.3 Intégration des éléments de la base de connaissances 46

4.3.1 Résultat attendu 46
4.3.2 Méthodologie 47
4.3.3 Versions étiquetées 48
4.4 Considérations particulières 50
4.5 Forme finale uniformisée 51
4.6 Ensemble de données hors du vocabulaire 53
4.7 Vocabulaires 56
4.7.1 Segmentation des jetons 56
4.7.2 Identification des URIs 56
4.7.3 Symboles de base 56
4.7.4 Architectures sans copie 57
4.7.5 Architectures avec copie 58
4.7.6 Utilisation des vocabulaires 59
4.8 Métriques 61
4.8.1 Score BLEU 61
4.8.2 Précision des réponses 61

CHAPITRE 5 ÉVALUATION ET DISCUSSION 63
5.1 Résultats des expérimentations 63
5.1.1 Reproduction des résultats de base 63
5.1.2 Données corrigées 66
5.1.3 Données partiellement étiquetées - ressources 67
5.1.4 Données partiellement étiquetées - schéma 70
5.1.5 Données complètement étiquetées 72
5.2 Discussion 73

CHAPITRE 6 CONCLUSION 76
6.1 Synthèse des travaux 76
6.2 Limitations de la solution proposée 77
6.2.1 Algorithme d'étiquetage 77
6.2.2 Données hors du vocabulaire 78
6.3 Améliorations futures 79
6.3.1 Portabilité du modèle sur plusieurs bases de connaissances ou langues 79
6.3.2 Portabilité du mécanisme de copie sur d'autres architectures 79
6.3.3 Algorithme d'étiquetage neuronal 80

RÉFÉRENCES 81

ANNEXES 85

LISTE DES TABLEAUX

Tableau 1.1 Exemple d'utilisation 1
Tableau 1.2 Différents types de requêtes SPARQL 4
Tableau 3.1 Configuration de nos modèles 27
Tableau 4.1 Survol des ensembles de données utilisés 35
Tableau 4.2 Nombres d'entrées affectées par la correction 46
Tableau 4.3 Différents préfixes dans les bases de connaissances 48
Tableau 4.4 Description des composantes d'une entrée 52
Tableau 4.5 Survol des ensembles de données hors vocabulaire (HV) 54
Tableau 4.6 Symboles communs aux vocabulaires 57
Tableau 5.1 Résultats des architectures sans copie sur les données originales de LC-QuAD 64
Tableau 5.2 Résultats des architectures sans copie sur les données originales de Monument 65
Tableau 5.3 Résultats des architectures originales sur les données hors du vocabulaire (HV) 65
Tableau 5.4 Résultats des architectures sans copie sur les données corrigées de LC-QuAD 66
Tableau 5.5 Résultats des architectures sans copie sur les données corrigées de Monument 67
Tableau 5.6 Résultats des architectures sans copie sur les données hors du vocabulaire (HV) corrigées 67
Tableau 5.7 Résultats des architectures sur les données dans lesquelles seules les ressources sont étiquetées 68
Tableau 5.8 Résultats des architectures sur les données hors du vocabulaire (HV) dans lesquelles seules les ressources sont étiquetées 70
Tableau 5.9 Résultats des architectures sur les données dans lesquelles tous les éléments sauf les ressources sont étiquetés 71
Tableau 5.10 Résultats des architectures sur les données hors du vocabulaire (HV) dans lesquelles tous les éléments sauf les ressources sont étiquetés . . 72
Tableau 5.11 Résultats des architectures sur les données complètement étiquetées . 73
Tableau 5.12 Résultats des architectures sur les données hors du vocabulaire (HV) complètement étiquetées 73
Tableau A.1 Correpondances entre les jetons en SPARQL exécutable et intermédiaire 85

Tableau B.1 Tailles des vocabulaires des architectures sans copie 86
Tableau B.2 Tailles des vocabulaires des architectures avec copie 86
Tableau D.1 Résultats des architectures sans copie sur les données originales de LC-QuAD 104
Tableau D.2 Résultats des architectures sans copie sur les données originales de Monument 104
Tableau D.3 Résultats des architectures originales sur les données hors du vocabulaire (HV) 104
Tableau D.4 Résultats des architectures sans copie sur les données corrigées de LC-QuAD 105
Tableau D.5 Résultats des architectures sans copie sur les données corrigées de Monument 105
Tableau D.6 Résultats des architectures sans copie sur les données hors du vocabulaire (HV) corrigées 105
Tableau D.7 Résultats des architectures sur les données dans lesquelles seules les ressources sont étiquetées 106
Tableau D.8 Résultats des architectures sur les données hors du vocabulaire (HV) dans lesquelles seules les ressources sont étiquetées 106
Tableau D.9 Résultats des architectures sur les données dans lesquelles tous les éléments sauf les ressources sont étiquetés 107
Tableau D.10 Résultats des architectures sur les données hors du vocabulaire (HV) dans lesquelles tous les éléments sauf les ressources sont étiquetés . . 107
Tableau D.11 Résultats des architectures sur les données complètement étiquetées . 108
Tableau D.12 Résultats des architectures sur les données hors du vocabulaire (HV) complètement étiquetées 108

LISTE DES FIGURES

Figure 1.1 Exemple : Deux triplets issus d'une base de connaissances 2
Figure 1.2 Exemple syntaxe RDF 3
Figure 2.1 Un exemple d'encodage en SPARQL intermédiaire 10
Figure 2.2 Une paire générée à partir d'un patron 11
Figure 3.1 Architecture générale d'un système de question-réponse d'une base de connaissance interrogée en langue naturelle 23
Figure 3.2 Architecture générale d'un modèle *Seq2Seq* 24
Figure 3.3 Architecture détaillée d'un encodeur et d'un décodeur *Seq2Seq* de base 25
Figure 3.4 Architecture générale d'un modèle *Seq2Seq* avec copie 30
Figure 3.5 Architecture détaillée d'un encodeur et d'un décodeur *Seq2Seq* avec copie 31
Figure 4.1 Entrée dans l'ensemble de données Monument 36
Figure 4.2 Une entrée dans la version originale de LC-QuAD 38
Figure 4.3 Une entrée dans la version fournie par TNTSPA 39
Figure 4.4 Une erreur d'encodage dans TNTSPA 40
Figure 4.5 Une erreur d'encodage dans Monument80 40
Figure 4.6 Exemples de défauts dans LC-QuAD 42
Figure 4.7 Méchanisme d'étiquetage 46
Figure 4.8 Ordre des éléments référencés dans une requête, suivant les informations contenues dans le patron associé 47
Figure 4.9 Une entrée non-étiquetée 49
Figure 4.10 Une entrée dans laquelle seule les ressources sont étiquetées 49
Figure 4.11 Une entrée de Monument dans laquelle seule les ressources et les types sont étiquetées 50
Figure 4.12 Une entrée dans laquelle tous les éléments du schéma 50
Figure 4.13 Une entrée dans laquelle tous les éléments de la base de connaissances sont étiquetés 51
Figure 4.14 Entrée dans laquelle il est impossible d'étiqueter tous les éléments . . 51
Figure 4.15 Format standard d'une entrée de nos ensembles de données générés . 53
Figure 4.16 Exemple d'entrée standard générée selon le format proposé 54
Figure 4.17 Un bon résultat obtenu avec les mauvais éléments de la base de connaissances 55
Figure 4.18 Une entrée étiquetée 57
Figure 4.19 Vocabulaires complets sans copie 58

Figure 4.20 Construction des vocabulaires pour la copie 59
Figure 4.21 Une entrée préparée pour une architecture sans copie 60
Figure 4.22 Une entrée préparée pour une architecture avec copie 60
Figure 4.23 Une entrée préparée pour une architecture avec copie dans laquelle les éléments hors des vocabulaires de base sont masqués 60
Figure 5.1 Une entrée des données originales, avant nos corrections 64
Figure 5.2 Une entrée des données corrigées 66
Figure 5.3 Une entrée partiellement étiquetée avec les ressources 67
Figure 5.4 Une entrée partiellement étiquetée avec tous les éléments du schéma . 70
Figure 5.5 Une entrée complètement étiquetée 72
Figure C.1 Résultats les données originales de Monument 87
Figure C.2 Résultats les données originales de LC-QuAD 88
Figure C.3 Résultats les données corrigées de Monument 89
Figure C.4 Résultats les données corrigées de LC-QuAD 90
Figure C.5 Résultats sur Monument avec seules les ressources étiquetées 91
Figure C.6 Résultats sur Monument 50 avec seules les ressources étiquetées . . . 92
Figure C.7 Résultats sur Monument 80 avec seules les ressources étiquetées . . . 93
Figure C.8 Résultats sur les questions intermédiaires de LC-QuAD dans lesquelles seules les ressources sont étiquetées 94
Figure C.9 Résultats sur Monument avec seuls les éléments du schéma étiquetés . 95
Figure C.10 Résultats sur Monument 50 avec seuls les éléments du schéma étiquetés 96
Figure C.11 Résultats sur Monument 80 avec seuls les éléments du schéma étiquetés 97
Figure C.12 Résultats sur les questions intermédiaires de LC-QuAD dans lesquelles seuls les éléments du schéma sont étiquetés 98
Figure C.13 Résultats de la version complètement étiquetée de Monument 99
Figure C.14 Résultats de la version complètement étiquetée de Monument 50 . . . 100
Figure C.15 Résultats de la version complètement étiquetée de Monument 80 . . . 101
Figure C.16 Résultats de la version complètement étiquetée des questions intermédiaires de LC-QuAD 102

LISTE DES SIGLES ET ABRÉVIATIONS

BC	Base de connaissances
BLEU	Bilingual Evaluation Understudy
CHP	Calcul haute performance
CnnS2S	Convolutional Network Sequence-to-Sequence
DBNQA	DBpedia Neural Question Answering
GLU	Gated Linear Units
GPU	Graphics Processing Unit
HV	Hors vocabulaire
KGQA	Knowledge Graph Question Answering / Questions-Réponses sur les graphes de connaissances
LC-QuAD	Largescale Complex Question Answering Dataset
LSTM	Long Short-Term Memory
MMNMT	Massively Multilingual Neural Machine Translation / Traduction automatique neuronale massivement multilingue
NMT	Neural Machine Translation / Traduction automatique neuronale
NSpM	Neural SPARQL Machine
QALD	Question Answering over Linked Data
SDA	Smart Data Analytics
Seq2Seq	Sequence to Sequence
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
RDF	Resource Description Framework
RNN	Recurrent Neural Network
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium

LISTE DES ANNEXES

Annexe A Correspondances en SPARQL Intermédiaire 85

Annexe B Tailles des vocabulaires 86

Annexe C Courbes d'entraînement 87

Annexe D Scores de précision, rappel et F1 103

CHAPITRE 1 INTRODUCTION

Le domaine du Web Sémantique tente de codifier l'organisation de données sur le Web afin qu'un ordinateur puisse y accéder et les traverser de manière optimale. De cette manière, on s'assure que toute nouvelle information publiée puisse être intégrée dans des bases de connaissances, dont le but est de regrouper et d'organiser le plus de concepts possible. Ces bases de connaissances regroupent un ou plusieurs graphes décrits suivant le Resource Description Framework (RDF). Dans ces graphes, les nœuds sont des ressources ou des concepts, et les arêtes représentent les relations entre eux. On interroge ces bases de connaissances avec le SPARQL Protocol and RDF Query Language (SPARQL). Pour illustrer l'importance de ces données, on peut prendre en exemple le *Google Knowledge Graph*, qui contenait (en 2020) 500 milliards de faits (relations) sur 5 milliards d'entrées (concepts ou ressources) [1].

Cependant, ce ne sont pas tous les utilisateurs qui savent utiliser le SPARQL, et son apprentissage présente un défi de taille. Cette contrainte crée un biais d'accessibilité important, qui empêche la majorité des utilisateurs d'accéder à des tonnes d'informations. Or, l'un des principes de conception du World Wide Web Consortium (W3C), qui est l'organisation derrière RDF, est le *Web pour tous*. Il incarne le concept d'égalité d'accès aux ressources, quel que soit le parcours de chacun [2]. Sachant cela, il est d'autant plus important de trouver une manière de régler ce problème d'accès.

Tableau 1.1 Exemple d'utilisation

Question en langue naturelle	Qu'est-ce que Villa La Mauresque ?
Traduction SPARQL attendue	select ?a where { dbr :Villa_La_Mauresque dbo :abstract ?a }
Réponse attendue	La villa La Mauresque est une propriété privée, aménagée en 1927 au cap Ferrat (Alpes-Maritimes), pour servir de résidence principale au romancier britannique Somerset Maugham. Entourée de jardins en terrasses, cette maison a reçu nombre d'écrivains et de célébrités.

Permettre aux utilisateurs d'interroger une base de connaissances (BC) en utilisant simplement la langue naturelle pour poser des questions et obtenir la bonne réponse est une bonne manière d'enlever le besoin d'une quelconque connaissance préalable. On peut utiliser des systèmes de questions-réponses, qui servent à récupérer automatiquement des réponses d'une

base de données ou de documents. Cependant, la génération de la requête SPARQL pour récupérer la bonne réponse à une question est un défi important, notamment à cause de la syntaxe complexe du langage de requête. C'est là que le domaine de recherche de la traduction automatique vers SPARQL entre en jeu. En considérant SPARQL comme une langue à part entière, au même titre que le français ou l'anglais, on peut construire un système qui traduit une question en langue naturelle en requête SPARQL équivalente. La table 1.1 présente un exemple d'utilisation d'un tel système.

1.1 Terminologie

Cette section fait le détail des différents termes et concepts importants à comprendre afin de bien saisir la problématique

1.1.1 Bases de connaissances

Les bases de connaissances sont un moyen d'organiser un grand nombre d'éléments et leurs relations. Elles se veulent une manière de représenter toute l'information contenue sur le web dans un format facilement lisible par les machines. Au contraire des bases de données, notamment relationnelles, qui sont organisées en colonnes dans des tables, les bases de connaissances sont organisées sous la forme d'un graphe dans lequel les nœuds sont des concepts ou des objets, et les arêtes sont les relations entre eux. Par exemple, la figure 1.1 représente un graphe dans lequel le nœud *Michelle Obama* est lié au nœud *Barack Obama* par la relation *estÉpouseDe*. Un nœud peut également posséder des propriétés de type *littéral*, par exemple une chaîne de caractère, un entier ou un booléen. Dans notre exemple, cela se traduit par la propriété *seNomme* qui a comme valeur la chaîne de caractères *"Michelle Obama"*. On utilise la syntaxe RDF pour définir une base de connaissances.

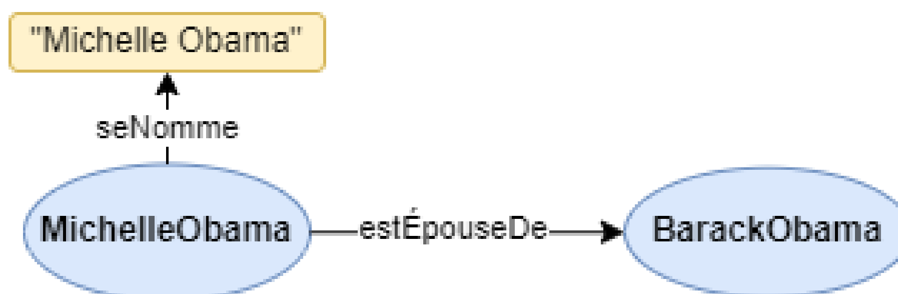


Figure 1.1 Exemple : Deux triplets issus d'une base de connaissances

1.1.2 RDF

RDF est un cadriciel développé par le W3C et qui se veut un standard pour formaliser et uniformiser la description des graphes décrivant des bases de connaissances. Tous les graphes développés avec RDF représentent des données dans un format compréhensible, facile à échanger entre applications, permettant ainsi.... de réunir toute l'information disponible sur le web simplement. Il s'agit du langage à la base des bases de connaissances, qui permet de catégoriser le type de relation entre chaque nœud d'un graphe. Une base de connaissances est composée de triplets RDF. Un triplet RDF est composé d'un sujet, d'un prédicat et d'un objet. Le sujet et l'objet sont généralement des classes (p. ex., un humain), ou des instances d'une classe (e.g., Michelle Obama). Le prédicat est généralement une propriété (p. ex., estÉpouseDe ou seNomme). Des exemples de triplets RDF dans la figure 1.2 sont (Michelle Obama, est de type, Humain), (Michelle Obama, estÉpouseDe, BarackObama), et (Michelle Obama, seNomme, "Michelle Obama"). La figure 1.2 décrit en RDF avec le language Turtle [3] le schéma et ressources représentés dans la figure 1.1

```
ex:BarackObama
     a ex:Humain;

ex:MichelleObama
     a ex:Humain;
     ex:aNom "Michelle Obama"^^^xsd:string;
     ex:estEpouseDe ex:BarackObama;
```

Figure 1.2 Exemple syntaxe RDF

1.1.3 SPARQL

Le langage de requêtes SPARQL utilise les triplets RDF pour retourner les informations demandées à une base de connaissances RDF. La question *"Quel est le nom de l'épouse de Barack Obama ?"* pourrait être traduite par la requête suivante :

```
SELECT ?nom WHERE {
     ?epouse ex:estEpouseDe ex:BarackObama .
     ?epouse ex:seNomme ?nom .
}
```

Lors de l'exécution, la requête cible d'abord tous les nœuds qui correspondent aux épouses

du nœud *Barack Obama* en utilisant le triplet (épouse, estÉpouseDe, BarackObama). Puis, on sélectionne le nom des sujets avec le triplet (épouse, seNomme, nom). Dans cet exemple, *épouse* et *nom* sont des variables puisqu'elles représentent les informations que l'on cherche. Cette requête retournera donc la chaîne de caractères *"Michelle Obama"* puisque Barack Obama n'a qu'une seule épouse, nommée *Michelle Obama*. Il y a plusieurs types de requêtes SPARQL, mais les requêtes de type COUNT, qui retournent un compte d'objets, celles de type ASK, qui retournent vrai ou faux selon la question posée, et les requêtes standards comme celle présentée plus haut sont celles représentées dans nos données. La table 1.2 présente des exemples de questions associées à ces types de requêtes.

Le langage de requête SPARQL est moins connu et moins utilisé que le langage de requête Structured Query Language (SQL), en partie parce qu'il est considéré comme un peu moins intuitif et plus difficile à maîtriser. Cela crée un biais d'accessibilité important, surtout pour les utilisateurs n'ayant aucune expérience en programmation. Il est important de rendre l'interrogation des bases de connaissances plus simple afin d'améliorer l'accès à l'information.

1.2 Éléments de la problématique

Cette section énonce les différents éléments de la problématique à laquelle nous nous attaquerons dans ce mémoire.

Afin de mieux comprendre ces limitations, il est utile de définir brièvement deux systèmes neuronaux importants, que nous détaillerons plus en profondeur au chapitre 2. Tout d'abord, les systèmes de questions-réponses prennent en entrée une question en langue naturelle et retournent la réponse trouvée dans une base de données ou dans un corpus de documents. Ensuite, les systèmes de traduction automatique prennent en entrée une phrase source en langue naturelle et retournent la phrase traduite en une langue cible, en utilisant un vocabulaire construit à partir d'exemples de traductions de référence.

Tableau 1.2 Différents types de requêtes SPARQL

Type	**Exemple**	**Type de retour**
Vanilla	Qui est l'épouse de Barack Obama	Une ressource ou une valeur
ASK	Est-ce que Michelle Obama est l'épouse de Barack Obama ?	Vrai ou Faux
COUNT	Combien d'épouses a Barack Obama ?	Un nombre entier

1.2.1 Intégration du schéma insuffisante

Les graphes RDF dans lesquels les données sont organisées contiennent des informations importantes sur la manière d'interroger les bases de connaissances. En effet, en analysant l'ontologie (la structure) sur laquelle repose une base de connaissances, on pourrait connaître les noms exacts des classes, prédicats et ressources. On pourrait également apprendre que la classe *livre* est reliée à la classe *auteur* par la propriété *est écrit par*. En bref, les informations contenues dans le graphe RDF constituent l'une des meilleures sources à partir desquelles notre modèle peut apprendre afin d'améliorer les traductions générées.

Or, dans les architectures actuelles, ces informations ne sont presque pas exploitées. En effet, les graphes RDF peuvent contenir des milliers d'éléments différents. Il est donc difficile pour une architecture neuronale de l'apprendre de manière complète, et encore plus difficile de savoir quelles connaissances utiliser pour répondre à une question particulière. Aussi, il est souvent difficile d'identifier dans les questions en langue naturelle les mots ou expressions se rapportant à des éléments de la base de connaissances. Cela rend l'intégration directe du schéma dans la question difficile.

1.2.2 Incapacité des modèles à traiter des sujets inconnus

Les modèles de traduction automatique apprennent à l'aide d'exemples. En voyant assez d'utilisations d'un mot dans son contexte, un modèle aura une idée de son sens et de la manière de l'employer.

Or, la question se pose alors de savoir comment un mot que le modèle ne connaît pas sera traité. En pratique, il sera remplacé par un symbole représentant un mot inconnu. De cette manière, le modèle comprendra qu'il ne connaît pas ce mot, et utilisera plutôt les autres mots pour essayer de comprendre la phrase. En général, le modèle traduira un mot inconnu par le mot qu'il a vu le plus souvent utilisé dans le contexte.

En traduction machine entre deux langues, cela ne représente pas un trop gros problème, puisque plusieurs traductions sont acceptables pour une même phrase, et que la suppression d'un mot ne nous empêche pas nécessairement de tout de même comprendre le sens d'une phrase. Cependant, dans le cadre d'un système de questions-réponses, il s'agit d'une lacune majeure, car cette incapacité à traiter des nouveaux mots signifie qu'un modèle ne pourra répondre à des questions que si elles portent sur un sujet qu'il a déjà vu, ce qui limite grandement sa portée et son utilité.

Aussi, un modèle a besoin de plusieurs exemples d'un mot pour apprendre son sens. Or, si l'on prend pour exemple les 4 233 000 concepts contenus dans la base de connaissances générale

DBpedia [4], il n'est pas réaliste de générer un ensemble de données avec des formulations assez variées pour couvrir plusieurs exemples de tous ces concepts.

Les modèles de l'état de l'art sont excellents pour générer des requêtes qui sont syntaxiquement correctes [5], mais sont incapables de répondre correctement à une question si elle porte sur des éléments de la base de connaissances qu'ils n'ont pas rencontrés pendant l'entraînement.

1.2.3 Métriques d'évaluation non représentatives

Dans l'état actuel du domaine, la métrique principalement utilisée pour évaluer les systèmes de traduction vers SPARQL est le score de *Bilingual Evaluation Understudy* (BLEU). Ce dernier considère le fait que plusieurs traductions sont acceptables pour une même phrase, ce qui en fait une métrique de traduction excellente.

Cependant, dans le cadre des systèmes de questions-réponses, la qualité de la traduction seule n'est pas représentative de la précision d'une réponse retournée par une requête. En effet, le score BLEU d'une traduction syntaxiquement correcte, mais qui ne fait pas référence aux bons éléments, sera tout de même relativement haut. Or, dans un système de questions-réponses, la qualité d'une réponse est alors autant - sinon plus - importante que la qualité d'une requête. Puisque le score BLEU n'évalue que la qualité de la traduction et non la réponse, les performances rapportées ne sont donc pas représentatives de la performance réelle des modèles de génération de requêtes SPARQL.

1.2.4 Ensembles de test incomplets

Dans les jeux de données de traduction de questions vers SPARQL, une limite importante vient du fait qu'il y a beaucoup d'intersection entre l'ensemble des éléments de base de connaissances utilisés dans l'ensemble d'entraînement et celui des éléments utilisés dans l'ensemble de test. Décrit plus en détail à la section 4.1.1, le taux d'intersection représente la proportion des éléments de la base de connaissances (classes, propriétés, ressources) référencés dans l'ensemble de test qui ont déjà été vus pendant l'entraînement. Cela signifie que les modèles ne sont presque pas testés sur des éléments jamais vus. Or, puisqu'on ne peut pas s'attendre à ce que les utilisateurs posent uniquement des questions sur des éléments que le modèle connaît, les ensembles de tests utilisés actuellement ne nous donnent aucune indication sur l'utilisabilité réelle de nos architectures.

1.2.5 Manque d'uniformité des jeux de données

En voulant utiliser les ensembles de données de l'état de l'art pour implémenter notre système, nous avons été confrontés au manque d'uniformité dans les données. Dans le but de développer une solution qui s'adapte pour plusieurs ensembles de données et modèles, nous considérons comme uniformes deux ensembles de données structurés de la même manière et dont les structures/formats sont identiques (c.-à-d., questions en langue naturelle, requêtes en SPARQL, patrons de requêtes trouées, etc.).

Dans leur état actuel, bien qu'ils utilisent les mêmes règles de prétraitement, il y a des différences importantes entre les ensembles de données. Il y a également plusieurs erreurs telles que des paires question-requête incorrectes ou incomplètes. Ce manque d'homogénéité crée des blocages importants pour le développement du domaine.

1.2.6 Difficulté de reproductibilité des résultats

Finalement, au cours de notre recherche, nous avons également été confrontées à la difficulté de reproductibilité des résultats. En effet, plusieurs modèles sont entraînés sur des serveurs de calcul haute performance (CHP) avec l'aide de FairSeq [6]. FairSeq est une excellente librairie d'apprentissage machine qui propose les meilleures versions de plusieurs modèles de traitement de la langue naturelle. Cependant, un chercheur qui souhaite s'initier à la traduction automatique vers SPARQL doit non seulement apprendre les rouages des modèles, mais aussi comment implémenter ses propres modèles avec FairSeq avant de pouvoir contribuer au domaine. Il est également difficile d'avoir accès à des CHP sans licence pédagogique.

1.3 Objectifs de recherche

Le but principal de ce mémoire est de donner la capacité aux modèles de traduction de la langue naturelle vers SPARQL de type *Seq2Seq* de répondre correctement à des questions portant sur des éléments de base de connaissances inconnus. Pour ce faire, nous devrons générer une représentation de la question qui est reliée aux éléments de la base de connaissances utilisés dans la requête SPARQL.

Notre première hypothèse est donc que l'ajout d'un mécanisme de copie, qui nous permettra de copier des mots de la question vers la requête générée, permettra à notre modèle de copier les éléments de la base de connaissances directement de la question au lieu de devoir apprendre à les générer. Afin de bien vérifier si cela améliore les capacités des modèles, il nous faudra également développer des métriques qui couvrent plus de facettes des architectures.

Par conséquent, notre deuxième hypothèse est que l'introduction de métriques évaluant la précision des réponses et la capacité des modèles à gérer des éléments inconnus nous permettra d'avoir une meilleure idée de leur utilisabilité réelle.

Afin de confirmer ou infirmer ces hypothèses, nous énonçons les questions de recherche suivantes :

> ***QR1.*** L'intégration du schéma RDF et des éléments de la base de connaissances dans les questions en langage naturel, couplée à un mécanisme de copie, améliore-t-elle la précision des traductions ?
>
> ***QR2.*** L'utilisation d'un mécanisme de copie permet-elle au modèle de traiter plus précisément les éléments inconnus de la base de connaissances ?
>
> ***QR3.*** L'évaluation de la précision des réponses et des performances sur des éléments de la base de connaissances inconnus sont-elles des métriques suffisantes afin d'évaluer nos architectures de manière complète ?

1.4 Plan du mémoire

Le présent mémoire est organisé comme suit. Le chapitre 2 présente l'état de la recherche actuelle dans le domaine de la traduction automatique, ainsi que dans certains domaines adjacents. Les principaux travaux de recherche en intégration des schémas, en traduction machine vers le SQL, en intégration de mécanismes de copie, et en traduction vers SPARQL y sont couverts, entre autres. Notre méthodologie est détaillée sur plusieurs chapitres. Le chapitre 3 fait le détail des architectures utilisées, et présente notre contribution principale à ce niveau. Le chapitre 4 présente tous les ensembles de données, construits à partir des données originales ou de celles que nous avons générées ainsi que les métriques utilisées. Le but de ces deux chapitres sont d'être une base solide pour quiconque souhaite reproduire nos résultats. Ces derniers sont présentés au chapitre 5, et sont suivis d'une analyse et d'une discussion détaillées.

CHAPITRE 2 REVUE DE LITTÉRATURE

Ce chapitre présente les concepts de base importants à la compréhension de la recherche, plus précisément les systèmes de questions-réponses et la traduction automatique. Une fois ces notions définies, on présente différentes solutions pour résoudre le problème de la transformation de questions en langue naturelle en requêtes SPARQL, puis plus spécifiquement la résolution de ce problème par la traduction automatique. On y détaille aussi les stratégies utilisées en SQL, les méthodes d'intégration du schéma de la base de connaissances dans les architectures développées et le rôle de la copie, pour terminer avec les limitations de ces solutions. Bien que nous n'ayons pas directement utilisé tous les articles présentés, leur compréhension nous a permis de développer une architecture complète et fonctionnelle.

2.1 Définitions et concepts de base

La présente section définit les concepts importants à la compréhension de cette recherche.

2.1.1 Systèmes de questions-réponses

Les systèmes de questions-réponses sont de plus en plus populaires à l'ère de l'information numérique. Un système de question-réponse est un système dans lequel un utilisateur pose une question, idéalement en langue naturelle, et le système retourne la bonne réponse. Les systèmes peuvent tirer la réponse de bases de données, de bases de connaissances, de communautés, etc. Les utilisateurs s'attendent à ce que les moteurs de recherche mettent à profit l'énorme quantité d'information qu'ils possèdent afin de virtuellement pouvoir répondre correctement à n'importe quelle question simple, mais une seule mauvaise réponse peut grandement affecter leur confiance envers le système. Or, une telle précision est extrêmement difficile à atteindre pour des architectures neuronales. On évalue généralement ces systèmes en utilisant la métrique de précision, qui calcule le pourcentage des bonnes réponses obtenues. Le système que nous concevons en est un de questions-réponses, car même si son architecture interne repose sur un mécanisme de traduction, il fonctionne comme un système de questions-réponses, et la précision de la réponse est beaucoup plus importante que la qualité de la requête utilisée pour la trouver.

2.1.2 Traduction automatique

Les systèmes de traduction automatique sont les systèmes qui servent à traduire des phrases d'une langue à l'autre en apprenant, à l'aide de beaucoup d'exemples, comment encoder le sens d'une phrase dans une langue machine intermédiaire, puis comment la décoder en la langue souhaitée. Le qualificatif *automatique* veut dire que tout le processus se fait sans aide ou intervention humaine. La principale métrique utilisée pour évaluer ces systèmes est le score BLEU, qui calcule la similarité entre une phrase traduite de référence et une phrase traduite générée automatiquement. Notre système en est un de traduction, car il apprend à transformer une question en langue naturelle en requête SPARQL équivalente en apprenant à représenter la signification de chaque mot et phrase, puis à la décoder vers la langue souhaitée.

2.1.3 SPARQL intermédiaire.

Bien entendu, le language de requête SPARQL n'est pas exactement équivalent à une langue naturelle. Une différence importante est que les symboles contenus dans les requêtes sont syntaxiquement importants et doivent être traités au même titre que des mots ayant un sens, au lieu de les ignorer comme on le fait souvent en traduction de langue naturelle. Par conséquent, les auteurs des Neural SPARQL Machines (NSpM) [7] proposent une manière d'encoder la requête SPARQL originale en remplaçant les symboles par des mots équivalents. Cela permet de mieux différencier les symboles et d'encoder les expressions en un seul symbole, ce qui modifie sensiblement le nombre de mots que le modèle doit apprendre.

```
{
  "question": "was ganymede discovered by galileo galilei ?",
  "pure_sparql": "ask where {
                      dbr:Ganymede_(moon)
                          dbp:discoverer dbr:Galileo_Galilei .
                  }",
  "interm_sparql": "ask where brack_open
                      dbr_Ganymede_ attr_open moon attr_close
                          dbp_discoverer dbr_Galileo_Galilei sep_dot
                   brack_close"
}
```

Figure 2.1 Un exemple d'encodage en SPARQL intermédiaire

Par exemple, on remplacera les points utilisés syntaxiquement par **sep_dot**, et on remplacera les points utilisés dans les noms des ressources de la base de connaissance par **attr_dot**. De la même manière, les accolades ouvrantes et fermantes seront encodées comme **brack_open** et **brack_close**. Le tableau complet des correspondances est représenté en annexe. La figure

2.1 présente un exemple d'une requête SPARQL exécutable encodée en requête SPARQL intermédiaire.

2.1.4 Patrons SPARQL

Dans le domaine de la traduction automatique vers SPARQL, les données du monde réel se font rares, puisque le SPARQL n'est pas utilisé couramment et que les modèles ont besoin de beaucoup de données pour être bien entraînés. Par conséquent, la majorité des ensembles de données dans le domaine de la traduction automatique de la langue naturelle vers SPARQL sont générés à partir de patrons, qui sont des paires question-requêtes trouées.

Composition. Un patron est composé d'une question en langue naturelle et de la requête SPARQL équivalente, soit dans sa forme originale ou intermédiaire. La question et la requête contiennent des trous annotés qui seront remplacés par des éléments de bases de connaissances du bon type lors de la génération des requêtes SPARQL. Dans la question, les trous seront remplacés par les étiquettes des éléments. Une étiquette est le nom en langue naturelle sous forme de chaîne de caractères qui représente un élément de base de connaissances, et on y accède par la propriété **rdfs :label**. Dans la requête, les trous seront remplacés par l'Uniform Resource Identifier (URI) de l'élément, qui est son identifiant unique dans la base de connaissances. La figure 2.2(a) est un exemple de patron, et la figure 2.2(b) est un exemple de paire question-requête générée à partir de ce patron.

```
{
  "question_template": "is <resource> a
    <class> ?",
  "interm_sparql_template": "ask where
    brack_open <resource> rdf_type
    <class> brack_close"
}
```

(a) Exemple d'un patron

```
{
  "question": "is bromley war memorial a
    place ?",
  "interm_sparql": "ask where brack_open
    dbr_Bromley_War_Memorial rdf_type
    dbo_Place brack_close"
}
```

(b) Exemple d'une paire question-requête

Figure 2.2 Une paire générée à partir d'un patron

2.2 Transformation de questions vers des requêtes sparql

Avant d'entrer dans le vif du sujet, soit les modèles de traduction automatique de la langue naturelle vers SPARQL, il est intéressant de prendre en compte toutes les manières de transformer une question en langue naturelle vers une requête SPARQL fonctionnelle, entre autres

afin de comprendre comment ces dernièrs gèrent les éléments hors du vocabulaire initial. Il existe plusieurs types de modèles que nous détaillerons dans la présente section.

Systèmes utilisant des mots-clefs. Les systèmes utilisant des mots-clefs tentent généralement d'extraire les mots-clefs d'une question posée par un utilisateur afin de construire une requête SPARQL fonctionnelle. La majorité de ces systèmes utilisent des patron de requêtes SPARQL, puisqu'ils sont une manière simple de s'assurer de toujours générer une requête fonctionnelle.

Par exemple, les auteurs de [8] commencent par prendre en entrée des paires de questions-requêtes afin d'extraire le plus de mots-clefs et d'informations possible relatives au patron avec ce qu'ils appellent un *générateur*. Puis, lors de l'étape de construction de requête, ils partent des informations extraites par le générateur et génèrent plusieurs requêtes possibles. Finalement, dans leur *comparateur*, ils utilisent un algorithme de classement afin de choisir la meilleure requête pour répondre à la question. Les auteurs de [9] ont une méthodologie similaire, mais contournent le problème difficile d'identifier quel élément de la base de connaissances est référencé par une expression de la langue naturelle en utilisant des mots-clefs entrés par un utilisateur.

Les auteurs de WDAqua-Core1 [10], attaquent la phase d'extraction des mots-clefs en utilisant les n-grammes pour faire ressortir le plus d'éléments possible, en excluant les mots d'arrêts qui n'ont pas d'importance pour comprendre le sens de la phrase (p. ex., est, sont, le).

Finalement, les auteurs de GQA [11] utilisent une méthodologie un peu différente des articles présentés jusqu'à présent, qui se base sur des règles de génération au lieu de patrons prédéfinis. Ils proposent une grammaire complexe qui utilise les informations de la base de connaissances afin d'extraire le plus d'entités possibles d'une phrase, et les utilisent pour générer des requêtes candidates selon leurs règles de génération. Le modèle de base est multilingue, et les auteurs de MuG-QA [12] proposent des règles supplémentaires qui permettent au modèle de comprendre 4 langues de plus.

Les modèles de ce type sont limités entre autres par leur capacité à identifier les bons mots-clefs dans une question en langue naturelle. Toutefois, puisqu'ils n'utilisent pas un vocabulaire prédéfini mais plutôt le schéma de la base de connaissances (et parfois même l'aide de l'utilisateur) afin de déterminer quels sont les éléments de la base de connaissances qui sont référencés par une question, ils pourront répondre correctement à n'importe quelle question dont la réponse se trouve dans la base de connaissances.

Systèmes modulaires. Les systèmes modulaires séparent le problème en étapes plus simples et utilisent différentes techniques d'apprentissage machine et de traitement de la langue pour résoudre chaque sous-problème. Par exemple, le système de question-réponse [13] est similaire à WDAqua-core1 [10], mais utilise des techniques un peu plus poussées pour chaque étape. Les auteurs de cet article construisent un système en cinq étapes. Tout d'abord, ils utilisent un algorithme d'analyse des dépendances pour construire un arbre grammatical de la question en langue naturelle. Puis, à l'aide d'un modèle d'apprentissage machine, ils classent la question soit comme une question demandant une liste de résultats, une question demandant un compte ou une question demandant une réponse booléenne (vrai ou faux). Puis, ils tirent avantage de beaucoup d'outils externes tels que Spotlight [14] ou TagMe [15] pour identifier les différents éléments de bases de connaissances dans chaque question. Ensuite, ils génèrent des requêtes candidates suivant leur propre algorithme, et finissent par une étape de classement des requêtes en utilisant des Tree-LSTMs [16].

Encore une fois, une limitation de ce types de modèles vient de leur capacité à identifier les bons mots-clefs. Cependant, puisqu'ils n'ont pas besoin d'apprendre comment utiliser spécifiquement chaque élément de la base de connaissances, ils ne rencontrent pas le problème de ne pouvoir répondre à des questions que sur des éléments vus pendant l'entraînement.

Systèmes de question-réponses sur des graphes de connaissances Les systèmes de questions-réponses sur les graphes de connaissances (KGQA) visent à reconstruire un sous-graphe du schéma RDF à partir d'une question en langue naturelle, et à utiliser ce sous-graphe afin de générer une requête correcte. L'un des aspects notables de ces architectures est qu'elles sont *zero-shot*, ce qui signifie qu'elles peuvent fournir une réponse correcte à une question sur un sujet qui n'a pas été vu lors de l'entraînement [17] (si la réponse est dans la base de connaissances). Par exemple, TeBaQ [18] est un système de KGQA qui exploite les graphes et les patrons d'autres ensembles de données KGQA. Il identifie d'abord les éléments candidats de la base de connaissances dans une question en langue naturelle, puis utilise un modèle d'apprentissage automatique afin d'extraire des caractéristiques spécifiques de la question. Le modèle renvoie ensuite la requête fonctionnelle la plus probable parmi un ensemble de requêtes SPARQL candidates construites à partir des informations recueillies lors de la première étape. L'avantage de ce système est que le modèle utilise sa connaissance des graphes pour identifier les éléments de la base de données candidats référencés dans une question. Ainsi, il n'est pas limité à une liste fixe d'éléments de la base de données qu'il a vus lors de l'entraînement, mais seulement par les informations contenues dans la base de connaissances. Un autre système KGQA intéressant est HGNet [19], dont la première étape est aussi de construire une liste d'éléments de la base de connaissances potentiels à partir de

la question en langue naturelle. Les auteurs de cette architecture identifient que dans certains graphes, certaines propriétés peuvent être dupliquées, par exemple si plusieurs éléments ont accès à la même propriété. Afin de tirer avantage de cette caractéristique, les auteurs utilisent un mécanisme de copie afin de copier les éléments d'un sous-graphe (sommets ou arêtes) qui sont dupliqués. Ce mécanisme utilise les informations générées par les réseaux LSTMs utilisés par l'architecture.

2.3 Traduction automatique

Grâce aux récentes avancées en traduction automatique, la traduction automatique neuronale (NMT) est le domaine le plus prometteur pour la transformation de questions en langue naturelle vers SPARQL. C'est pourquoi nous nous y concentrons avec plus de détail.

Modèle de base. L'architecture de base utilisée en traduction machine est l'architecture *Sequence to Sequence* (Seq2Seq), introduite par [20]. L'intuition derrière cette architecture s'énonce comme suit. Les modèles d'apprentissage machine ne prennent en entrée que des vecteurs de taille fixe. Or, dans la langue naturelle, toutes les phrases ne sont pas de la même taille. Il faut donc trouver une manière de les transformer pour que les modèles soient capables de les traiter. Pour ce faire, ces architectures utilisent des vocabulaires source et cible. Le vocabulaire source contient tous les jetons contenus dans les phrases en langue source de l'ensemble d'entraînement, et le vocabulaire cible contient tous ceux contenus dans les phrases en langue cible de l'ensemble d'entraînement. Un jeton peut être un mot complet, un préfixe, un suffixe, etc. Afin de transformer une phrase en vecteur, on remplace chaque jeton par son index dans le vocabulaire associé, et on ajoute des jetons remplissage si nécessaire afin que tous les vecteurs d'une même langue aient la même taille.

Le modèle apprend alors à comprendre le sens des phrases, et de les traduire en générant des jetons qui sont dans son vocabulaire cible. Lors de l'inférence, si la phrase source contient un jeton qui ne fait pas partie du vocabulaire, le modèle le remplace simplement par le jeton de remplacement **<unk>** qui signifie *inconnu*.

Récentes avancées. Dans les travaux originaux [20], l'encodeur et le décodeur sont des réseaux Long Short-Term Memory (LSTM). Or, plusieurs améliorations ont été apportées depuis, par exemple le remplacement des LSTMs par des LSTMs profonds [21], des transformers [22], ou même des réseaux convolutifs [23], qui sont généralement reconnus pour le traitement d'images. Récemment, les réseaux de traduction automatique neuronale massivement multilingue (MMNMT) [24, 25] ont fait leur apparition. Ces derniers sont très intéressants,

car ils utilisent des transformers légèrement modifiés pour traduire une phrase de n'importe quelle langue source vers n'importe quelle langue cible. Pour ce faire, ils sont entraînés sur plusieurs langues sources et cibles, au contraire des réseaux normaux qui comprennent une seule langue source vers une seule langue cible. Cela leur permet de transférer les apprentissages qu'ils font sur une langue aux autres qui lui ressemblent, et donc leur entraînement nécessite moins d'exemples. C'est très intéressant dans une optique d'accessibilité et d'inclusivité, car la qualité d'un modèle de traduction ne dépend plus directement du nombre de données disponibles dans cette langue. Aussi, dans le domaine de la traduction automatique, les données se font rares, notamment pour les langues dites pauvres en ressource (*low resource languages*).

Un autre exemple notable est T5 [26], qui utilise des transformateurs et l'apprentissage par transfert pour être capable de traduire trois langues à la fois. Le modèle dispose ainsi d'un vocabulaire riche de 32000 jetons et peut utiliser ses connaissances antérieures pour atteindre des performances plus élevées dans les langues pour lesquelles il y a moins de données d'entraînement. Cependant, comme indiqué dans l'article, leur modèle ne peut traiter qu'un ensemble prédéterminé et fixe de langues et il utilise un vocabulaire fixe.

Cela signifie que, bien que ces modèles puissent compter sur d'énormes vocabulaires, ils rencontrent le même problème de gestion d'éléments inconnus que les autres modèles de type *Seq2Seq*, puisqu'ils n'ont pas la capacité d'apprendre de nouveaux éléments une fois l'entraînement terminé.

Attention. Dans le domaine plus spécifique de la traduction de l'anglais vers SPARQL, la revue de littérature réalisée par [5] rapporte que les deux modèles qui performent le mieux en ce moment sont le *transformer* [22] et le *ConvSeq2Seq* [23]. Une composante importante de ces deux architectures est l'attention, introduite par [22]. Dans une phrase, ce ne sont pas tous les mots qui nous informent sur le sens. Certains mots, qu'on surnomme les *stopwords* (e.g., les, il, par), doivent être ignorés, alors que d'autres sont absolument nécessaires pour la sémantique de la phrase. L'attention est une couche de l'architecture qui permet au modèle d'identifier l'influence que chaque mot a sur la probabilité de génération d'un autre mot [22]. Dans nos architectures, nous utilisons plusieurs types d'attention.

Tout d'abord, l'auto-attention (aussi *self-attention*) sert à identifier les mots importants dans une phrase. La couche d'auto-attention compare donc une phrase à elle-même et, pour chaque mot de la phrase, évalue le poids des autres mots par rapport au mot analysé. Cela nous permet d'extraire des informations importantes sur la relation entre les mots d'une phrase.

Ensuite, l'attention encodeur-décodeur fonctionne comme l'auto-attention, mais elle sert à

identifier les relations entre les mots de deux phrases différentes, par exemple la question et la requête. Par exemple, elle comprendra peut-être que le mot **Combien** dans une question a une forte corrélation avec le mot **Count** dans une requête SPARQL.

Finalement, on utilise ce qu'on appelle l'attention à plusieurs têtes (aussi *Multi-head attention*). La faiblesse des mécanismes d'attention de base est qu'ils ne prennent pas en compte l'ordre des mots. Or, on peut tirer avantage de la composante aléatoire de l'attention en combinant plusieurs têtes (couches) d'attention de base en parallèle. On obtient alors plusieurs scores d'attention différents, qui se concentrent chacun sur quelque chose de différent. On les combine en un score d'attention final calculé grâce à plusieurs têtes d'attention. Cela nous permet de nous assurer que, malgré le fait que l'ordre ne soit pas pris en compte, toutes les nuances d'une phrase sont comprises par notre mécanisme d'attention.

Traduction en langue naturelle. Originellement développé pour le problème de traduction de l'anglais vers l'allemand, le modèle *ConvSeq2Seq* [23], basé sur l'architecture *Seq2Seq*, utilise des réseaux convolutifs - habituellement utilisés en traitement d'images - comme encodeur et décodeur. L'attention est de type multitêtes et est située dans le décodeur seulement, mais prend en entrée le résultat de l'encodeur.

De la même manière, l'architecture *Seq2Seq Transformer* [22] utilise des *transformers* comme encodeur et décodeur, et l'attention est située seulement dans le décodeur. Lorsqu'on parle de modèles, il est intéressant de mentionner la librairie FairSeq [6], utilisée par [5] pour obtenir ses résultats. FairSeq est une librairie d'apprentissage machine qui propose une implémentation des modèles de traduction les plus performants. Nous reviendrons en détail sur ces architectures dans le chapitre 3.

2.4 Stratégies utilisées en traduction vers SQL

En connaissant les particularités du langage SPARQL, il est intéressant de voir ce que nous pouvons apprendre de problèmes similaires, tels que le problème de l'analyse sémantique d'une question vers une requête SQL équivalente. SQL est un langage de requêtes utilisé pour interroger des bases de données relationnelles. Actuellement, le modèle le plus performant dans ce domaine [27] n'est pas un modèle de type *Seq2Seq*, mais plutôt un modèle de classification qui apprend à prédire 6 composants SQL différents (SELECT, AGGREGATION, nombre de WHERE, WHERE, WHERE OPERATORS, WHERE VALUE) en exploitant l'ensemble de données WikiSQL, qui est un ensemble de données largement annoté. Cette richesse des données représente un avantage dont nous ne disposons pas en traduction vers SPARQL. Par conséquent, bien qu'il ne s'agisse pas de l'architecture la plus performante, il

convient de noter la contribution du modèle Seq2SQL [28], basé sur *Seq2Seq*. Les auteurs de ce modèle utilisent une architecture de type *Seq2Seq* avec des LSTMs pour encoder et décoder les vecteurs, et ils augmentent la question en langage naturel en la concaténant à tous les noms de colonnes et au vocabulaire SQL. Le schéma de la base de données est essentiellement intégré directement dans l'entrée.

2.5 Intégration du schéma

À la lumière des stratégies utilisées pour SQL, il est intéressant de se pencher sur les manières dont le schéma RDF est utilisé dans les différentes tâches d'analyse de la langue naturelle. Inévitablement, le schéma RDF de la base de connaissances interrogée transparaît dans chaque requête SPARQL, que ce soit par les éléments référencés (ressources, éléments du schéma). Cela soulève la question de savoir si l'intégration du schéma directement dans le modèle permettrait à ce dernier d'améliorer la qualité des traductions, puisqu'il pourrait avoir une meilleure connaissance du contexte de chaque élément de la base de connaissances en sachant à quels éléments il se rattache et de quelle manière (propriétés). En dehors du problème de génération de requêtes SPARQL, plusieurs auteurs se sont penchés sur cette hypothèse. Nous détaillerons ces efforts dans la section suivante.

Représentations vectorielles. Une manière efficace d'intégrer le schéma dans un modèle est d'encoder les informations du schéma directement dans le sens des mots. Les auteurs d'ERNIE [29, 30], par exemple, tentent de créer une représentation vectorielle de chaque mot de manière analogue à BERT [31], mais enrichie par le schéma de la base de connaissances. Le modèle fonctionne avec deux encodeurs. Le premier apprend à encoder les informations lexicales et syntaxiques de la phrase, comme dans la majorité des modèles de type *Seq2Seq*. Le deuxième apprend à encoder l'information tirée des graphes de connaissance. ERNIE est utilisé dans plusieurs tâches de traitement de la langue naturelle, et le fait que la représentation d'un mot comprend de l'information du schéma lui permet d'obtenir de meilleures performances que BERT sur plusieurs tâches telles que le typage d'entité ou encore la classification de relations.

De manière similaire, les auteurs du modèle MPME [32] tentent eux aussi de créer des représentations vectorielles de mots enrichies des informations de l'élément de base de connaissances auquel ils se rattachent. Cependant, leur objectif principal est de résoudre le problème d'ambiguïté qui survient quand un mot peut prendre plusieurs sens, selon le contexte. En plus de contenir des informations du mot et de l'élément de base de connaissances, les représentations vectorielles encodent aussi une information de sens, qui permet de mieux distinguer

les mots selon leur contexte. Ces modèles sont en général très complexes, mais présentent un important avantage en matière de portabilité, puisqu'ils sont capables de tirer plein avantage des informations contenues dans une base de connaissances afin de générer des représentations vectorielles plus riches et complètes.

Aussi, les auteurs de KnowBERT [33] tentent de de créer des représentations vectorielles d'éléments de plusieurs bases de connaissances combinées, augmentées d'informations choisies par des experts. Ces deux versions de BERT sont interchangeables sur n'importe quelle base de connaissances qui respecte certains critères.

Enfin, SGPT [34], un travail récent développé parallèlement au nôtre, est construit sur GPT-2 [35] et vise à générer des requêtes SPARQL en utilisant des informations supplémentaires optionnelles de la base de connaissances pour chaque entrée, tel le niveau dans le graphe ou le type d'un élément de la BC. Cependant, les auteurs évitent le problème des mots hors du vocabulaire en masquant les URIs dans les questions et les requêtes et en générant des requêtes avec des paramètres substituables. Une fois la requête générée par l'architecture neuronale, un *post-processeur* non détaillé vient placer les bons éléments aux bons endroits dans la requête. Par contre cette architecture est particulièrement intéressante à cause de la manière assez directe dont elle intègre les informations supplémentaires.

Utilisation des patrons. En ce moment, les modèles de l'état de l'art en traduction automatique de la langue naturelle vers SPARQL sont principalement entraînés sur des ensembles de données générés à l'aide de patrons (e.g., [7, 36]), définis plus haut. Puisque les requêtes sont formulées en suivant la structure d'une certaine base de connaissances, nous considérons les patrons comme une manière d'intégrer le schéma dans le modèle, même si c'est fait de manière moins complète que les méthodes présentées plus haut.

Une manière de prendre plein avantage des patrons serait de générer des paires de questions-requêtes dans lesquelles se retrouvent directement les URIs des éléments de la base de connaissances utilisées. Le modèle pourrait apprendre à copier de la question vers la requête, au lieu de devoir apprendre la correspondance entre une expression décrivant un élément de la base de connaissances et son URI. Le mécanisme de copie, qui est une technique qui a abouti à de nombreuses avancées dans des tâches du traitement automatique des langues [37, 38] est décrit à la section suivante.

2.6 Méchanisme de copie dans d'autres tâches

Un mécanisme de copie permet à un modèle de copier directement des mots d'un document source (par exemple, la question en langue naturelle) vers un document cible (par exemple, la requête SPARQL). Le mécanisme est introduit par les auteurs de CopyNet [39], qui l'utilisent pour améliorer les tâches reliées au dialogue. Leur architecture est de type *Seq2Seq* et utilise des Recurrent Neural Networks (RNN) comme encodeur et décodeur. La copie se fait à la fin du décodeur et utilise le résultat de l'attention. Pour chaque mot, le modèle calcule une probabilité de copie, qui indique si le mot doit être copié du document source, et une probabilité de génération, qui indique si le mot doit être généré par le modèle. Si un mot de la source est inconnu du modèle (hors du vocabulaire), il a de plus grandes chances d'être copié. Cette couche de copie a été portée avec succès vers des architectures utilisant des transformers comme encodeur et décodeur, notamment pour des tâches de récapitulation [38] et de correction de grammaire [37]. Beaucoup de ces modèles s'inspirent des réseaux de pointeurs [40]. Le système de KGQA HGNet [19], décrit plus haut, utilise également un méchanisme de copie, bien que ce soit pour copier des informations d'un même sous-graphe. Cependant, à notre connaissance, la copie n'a pas encore été appliquée pour copier des éléments de questions en langue naturelle vers SPARQL.

2.7 Limitations

Dans leur état actuel, les modèles de traduction de la langue naturelle vers SPARQL présentent d'importantes limitations, brièvement présentées en introduction, que nous définissons dans cette section.

2.7.1 Intégration du schéma insuffisante

Tout d'abord, une limitation importante des modèles de type *Seq2Seq* en traduction vers SPARQL vient de la manière dont ils intègrent le schéma dans l'architecture. En ce moment, cette intégration est réalisée par les patrons servant à générer les données, qui est une manière plutôt limitée de le faire. En effet, le modèle n'a pas d'informations sur les éléments de la base de connaissances ou les relations entre eux. Aussi, au lieu d'alléger ou d'enrichir la tâche d'apprentissage, le modèle est obligé d'apprendre les correspondances entre chaque expression (e.g., *John Oliver*) et l'élément de la base de connaissances qui lui correspond (e.g., *dbr :John_Oliver*). Cela représente une partie considérable de la tâche d'apprentissage. Qui plus est, si l'utilisateur ne pose jamais de question sur les éléments appris, cet apprentissage est inutilisé, et donc superflu.

2.7.2 Incapacité de gérer des éléments inconnus

Selon les résultats rapportés par [5], qui évalue plusieurs architectures de type *Seq2Seq* sur des ensembles de données pour la traduction de l'anglais à SPARQL, les modèles atteignent un moins haut score BLEU sur l'ensemble de données Largescale Complex Question Answering Dataset (LC-QuAD) v1.0 [36]. Ce dernier est relativement petit, mais très varié sur le plan des éléments de base de connaissances.

Cela nous permet de formuler l'hypothèse que les architectures ont de la difficulté à gérer les entités inconnues en plus d'avoir du mal à relier les mots aux entités qu'ils désignent. En effet, une limitation connue des modèles de type *Seq2Seq* est qu'ils ne peuvent générer que des mots qu'ils ont déjà vus. Intuitivement, on ne peut communiquer qu'avec les mots que l'on connaît. Il en va de même pour les modèles.

2.7.3 Métriques non représentatives

Sachant cela, les très bons résultats obtenus sur l'ensemble de données Monument [7] sont plutôt surprenants. Or, au lieu de nous indiquer que le modèle est performant, ces résultats nous permettent plutôt de déterminer que les métriques en place ne sont pas suffisantes pour bien évaluer l'utilisabilité réelle des modèles.

En effet, Monument est un ensemble de données très simple, et la grande majorité des éléments de la base de connaissances utilisés dans le test ont déjà été vus par le modèle. On n'a donc aucune indication de comment ils traitent les éléments jamais vus. Le score BLEU, qui est l'une des métriques les plus répandues dans le domaine de la traduction, n'évalue que la qualité des mots utilisés dans la traduction générée automatiquement par rapport à la traduction de référence. Cette information, bien qu'importante, ne nous donne aucune indication sur la qualité des requêtes, si elles sont syntaxiquement correctes, ou si tous les bons éléments de la base de connaissances ont été utilisés.

Or, puisque notre système fonctionne fondamentalement comme un système de questions-réponses, il est crucial que nous l'évaluions comme tel. Qui plus est, si un utilisateur peut accepter quelques erreurs dans la traduction générée par un modèle de traduction d'une langue naturelle à une autre, il n'en va pas de même pour les systèmes de questions-réponses. Dans notre cas, on s'attend à une haute précision dans les réponses retournées. Une seule mauvaise réponse et surtout l'utilisation d'éléments incorrects de la base de connaissances peut décourager un utilisateur d'utiliser notre modèle.

2.7.4 Ensembles de tests incomplets

Similairement, les ensembles de tests présentement disponibles dans les jeux de données utilisés dans l'état de l'art sont "incomplets", puisque ces derniers ont un haut taux d'intersection avec les ensembles d'entraînement, tel que mentionné plus haut. Cela signifie que la majorité des entrées des ensembles de test portent sur des éléments déjà vus en entraînement, ce qui n'est pas représentatif du monde réel. En effet, alors que les ensembles d'entraînement ne couvrent qu'un sous-ensemble des sujets couverts par base de connaissances, les utilisateurs peuvent poser des questions sur n'importe lequel de ces sujets. Or, les ensembles de test fournis ne testent que si le modèle est capable de répondre à des questions sur les sous-sujets, et non s'il est utilisable sur une certaine base de connaissances. Il nous faudra donc générer un ensemble de test supplémentaire qui évalue la capacité d'un modèle à bien répondre à des questions portant sur des sujets jamais vus en entraînement.

2.7.5 Besoin d'uniformité à travers les ensembles de données

La méthode d'encodage en SPARQL intermédiaire est assez répandue à travers le domaine de la traduction machine vers SPARQL, et est notamment utilisée par Monument [7] et DBpedia Neural Question Answering (DBNQA) [41], qui sont des ensembles de données de l'état de l'art. Cependant, le manque de règle de transformations claires crée un manque d'uniformité à travers les différents jeux de données, puisqu'ils ne suivent pas tous exactement les même règles. Or, si l'encodage en SPARQL intermédiaire n'est pas bien fait, il peut être très difficile, voire impossible de retrouver la requête exécutable. Il sera donc important de générer une version uniforme et corrigée des ensembles de données, tout en explicitant la méthodologie utilisée pour le faire.

2.7.6 Difficulté de reproductibilité des résultats

Finalement, tel que décrit plus haut, il est souvent difficile de reproduire les résultats présentés dans l'état de l'art sans ressources importantes et parfois coûteuses. Dans l'esprit de réduire le plus possible les biais d'accès à l'information et à la recherche, nous voudrons rendre toutes nos données et notre code disponibles et exécutables sans prérequis, licence pédagogique, matériel coûteux, ou autre, pour quiconque voulant s'initier ou contribuer au domaine de la traduction machine vers SPARQL.

2.7.7 Survol

En résumé, les principales limitations des architectures actuelles sont qu'elles sont (1) incapables de gérer les éléments jamais vus de la base de connaissances, que (2) les métriques utilisées pour les évaluer sont souvent incomplètes et ne nous donnent pas de véritables indications de leur utilisabilité réelle, et que (3) le mécanisme d'apprentissage de nos architectures met l'accent sur la liaison des expressions aux bonnes entités au lieu de se concentrer sur l'apprentissage de la syntaxe SPARQL.

CHAPITRE 3 ARCHITECTURES

Le présent chapitre fera le détail des architectures utilisées et développées durant nos expérimentations. Pour bien comprendre le détail de notre contribution, il est intéressant de comprendre comment fonctionnent les architectures de base, puis quels sont les changements que nous venons y apporter.

3.1 Architecture générale

La figure 3.1 présente l'architecture de haut niveau d'un modèle de génération automatique de requêtes SPARQL. L'utilisateur pose une question en langue naturelle, le modèle de traduction automatique vers SPARQL traduit la question en une requête équivalente, puis on interroge la base de connaissances concernée avec la requête générée afin d'obtenir la réponse. Ainsi, contrairement aux approches de traduction neuronale originelles, notre système est également un système de type questions-réponses, puisque son utilité principale est de retourner la réponse à une question.

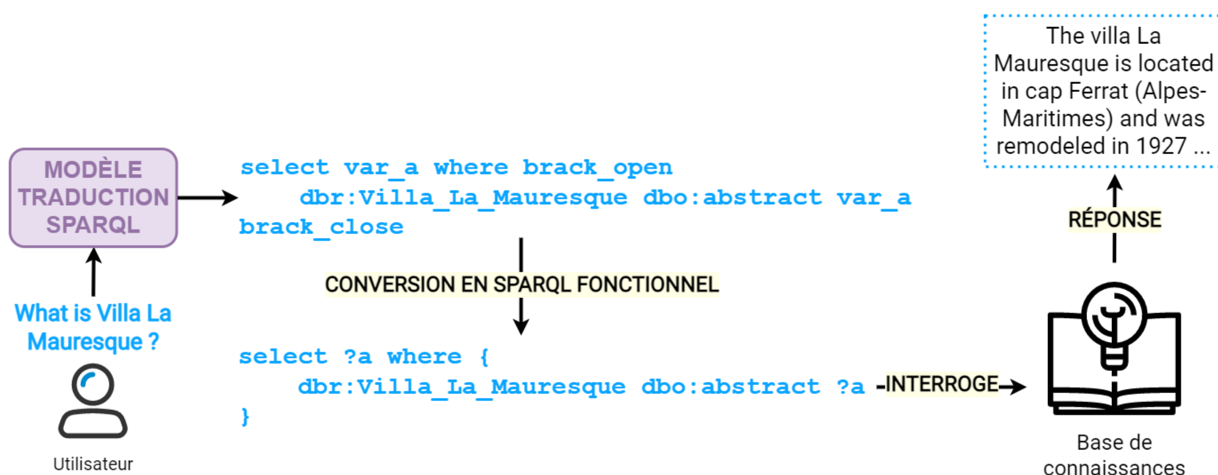


Figure 3.1 Architecture générale d'un système de question-réponse d'une base de connaissance interrogée en langue naturelle

La figure 3.2 présente l'architecture générale de nos modèles de traduction de la langue naturelle vers SPARQL. Suivant l'architecture *Seq2Seq*, ils sont composés d'un encodeur et d'une pile de décodeurs. La figure 3.3 présente le détail de la structure interne de l'encodeur et du décodeur. Nos deux architectures, décrites plus loin, suivent cette structure interne, à

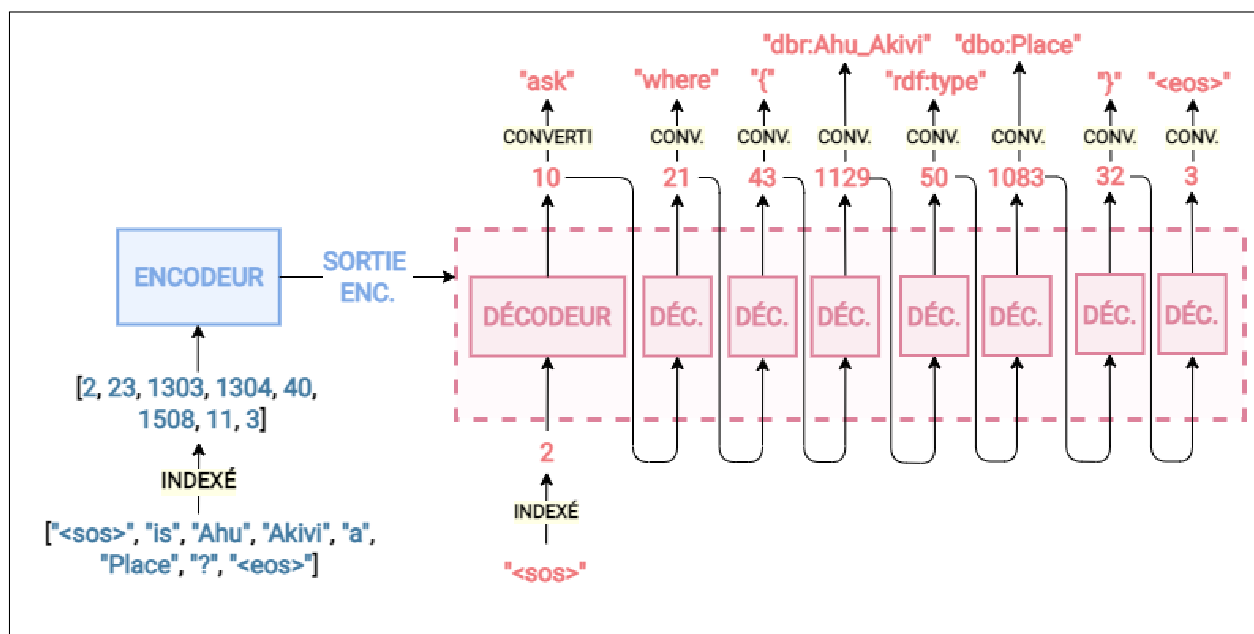


Figure 3.2 Architecture générale d'un modèle *Seq2Seq*

quelques détails près.

Encodeur. L'encodeur prend en entrée la question en langue naturelle transformée en vecteur, et retourne un tenseur représentant sa signification. Contrairement à l'implémentation originale par [20], il ne compresse pas la phrase source en un seul vecteur représentant sa signification, mais génère plutôt une séquence de vecteurs représentant chacun une partie de la phrase. Cette modification, implémentée par les auteurs de [22] et de [23], permet non seulement au jeton j d'avoir accès aux informations des jetons précédents (comme le permettent les RNNs), mais aussi aux informations des jetons suivants.

Décodeur. La sortie de l'encodeur et la requête SPARQL dans laquelle les jetons sont remplacés par des entiers (voir la section 3.3.1) sont passées au décodeur. Cela peut sembler contre-intuitif, puisque le décodeur doit apprendre à générer la requête SPARQL. Cependant, cette opération lui permet d'optimiser l'apprentissage. En effet, ce dernier fonctionne en générant un jeton à la fois, jusqu'à générer le jeton représentant la fin de la phrase. À chaque étape, il se base sur les jetons de la requête prédits jusqu'à maintenant et sur le tenseur représentant la question pour prédire le prochain jeton. Lors de l'entraînement, il est donc possible de paralléliser cette étape afin de prédire tous les prochains jetons d'un coup pour chaque sous-séquence composée des jetons 0 à N-1, où le jeton 0 est celui représentant le

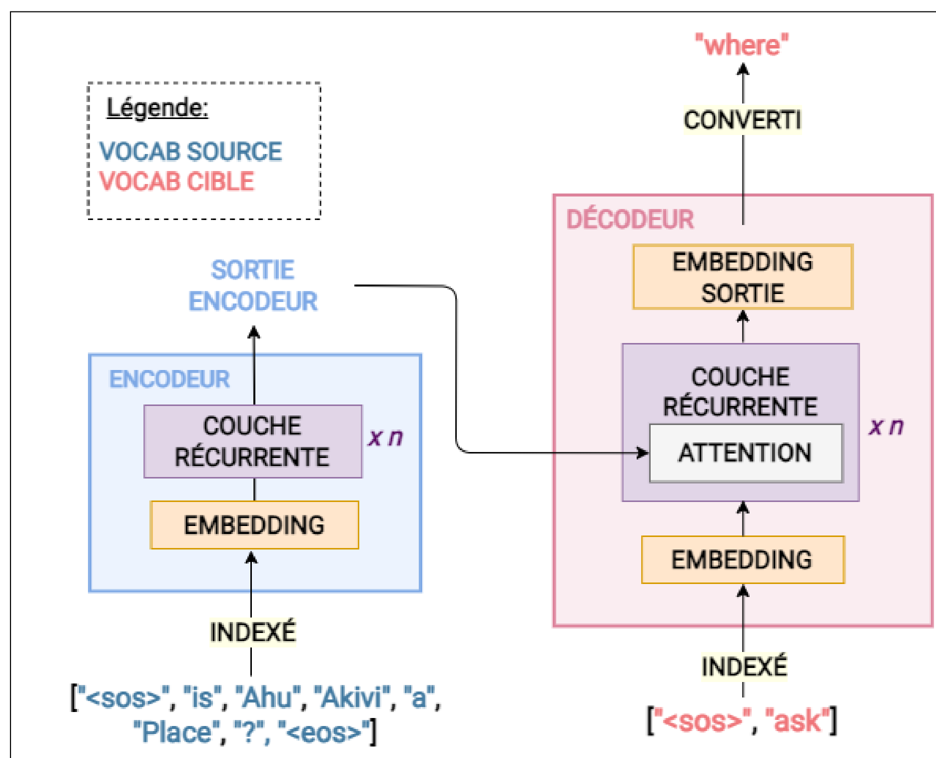


Figure 3.3 Architecture détaillée d'un encodeur et d'un décodeur *Seq2Seq* de base

début d'une phrase et le jeton N est celui représentant la fin.

Lors de l'inférence, au lieu de recevoir la requête complète, le premier décodeur de la pile ne reçoit en entrée que le jeton représentant le début d'une phrase, comme démontré à la figure 3.2, et génère le prochain jeton. Le prochain décodeur de la pile prend en entrée la requête générée jusqu'à maintenant, et ainsi de suite jusqu'à ce que le symbole **<eos>**, qui indique la fin de la phrase, soit généré.

3.2 Architectures de référence

Cette section présente les deux architectures de référence utilisées afin de déterminer l'impact de notre contribution. Nous utilisons les deux meilleures architectures sur la tâche de traduction automatique de l'anglais vers SPARQL, tel que rapporté par [5]. Au contraire des architectures plus anciennes qui utilisaient une pile d'encodeurs et de décodeurs pendant l'entraînement, les modèles utilisés sont optimisés pour faire la traduction en parallèle durant l'entraînement. On n'a donc plus directement accès aux connexions récurrentes, qui permettaient aux architectures d'avoir une sorte de mémoire des jetons et donc d'avoir une idée de leur position dans la phrase, mais l'entraînement se fait beaucoup plus rapidement.

Par contre, on utilise toujours une pile de décodeurs pour l'inférence.

3.2.1 Modèle *Seq2Seq* convolutif (CnnS2S)

Le modèle Convolutional Network Sequence-to-Sequence (CnnS2S) [23] est de type *Seq2Seq* et utilise des réseaux convolutifs comme encodeur et décodeur. Selon la revue de littérature réalisée par [5], il atteint les meilleures performances sur tous les ensembles de données testés. La table 3.1 fait le détail de la configuration de notre implémentation, qui est basée sur l'architecture *fconv_wmt_en_de*, qui est l'implémentation originale de [23] par FairSeq [6]. Il s'agit originellement d'un modèle servant à faire de la traduction automatique de l'anglais à l'allemand.

Encodeur. L'encodeur reçoit le lot contenant B questions indexées selon le vocabulaire source. Chaque phrase du lot passe à travers les opérations suivantes. On calcule d'abord un masque de remplissage M_{remp}, qui vaut 1 à toutes positions où l'indice d'un jeton correspond à l'indice du symbole de remplissage (**<pad>**). Ce masque permet de savoir quelles sont les positions des jetons qui composent la question, et quelles sont les positions des jetons qui ne servent qu'à faire en sorte que toutes les questions du lot aient la même taille.

On passe chaque jeton de la question dans une couche de plongements (embeddings) de taille E. On crée aussi un vecteur de positions ayant la forme $[0, 1 ... L_{quest} - 1]$ où L_{quest} est la taille de la question. Cela nous permet d'inclure l'information de l'ordre dans notre représentation vectorielle du jeton, puisque le modèle ne peut pas compter sur des connexions récurrentes pour le savoir comme dans les modèles utilisant des RNNs. On calcule le vecteur de plongement final pour chaque jeton en faisant la somme des plongements originaux et du vecteur de positions, ce qui nous donne une matrice de plongements T_{emb} de taille $[L_{quest} \times E]$.

On transforme T_{emb} en matrice de taille $[L_{quest} \times C]$, où C est la taille de la couche cachée. Cela nous permet de passer notre vecteur représentant le sens de la phrase à la couche récurrente, qui sera exécutée N fois. Selon la configuration définie au tableau 3.1, N = 15.

Après la dernière couche récurrente, notre matrice T_{emb} contient maintenant les informations encodant le sens de la phrase compris par le modèle. On retransforme la matrice de taille $[L_{quest} \times C]$ en une matrice de taille $[L_{quest} \times E]$ pour obtenir la matrice *convolutionnée* Enc_{conv}. On calcule aussi la matrice *combinée* Enc_{comb}, qui est la somme de la matrice Enc_{conv} et de la matrice T_{emb} calculée au début. Les matrices Enc_{conv} et Enc_{comb} sont utilisées afin de calculer l'attention dans le décodeur. Selon les auteurs du modèle [23], Enc_{conv} représente le

1. Afin de reproduire les résultats sur TNTSPA, nous avons utilisé un taux d'apprentissage de 3.5

Tableau 3.1 Configuration de nos modèles

Hyperparamètre	**Transformer**	**CnnS2S**
Taille Batch	128	128
Couches récurrentes	6	15
Dim. couche cachée	1024	[(512, 3) * 9, (1024,3) * 4, (2048, 1) * 2]
Dropout	0.5	0.2
Taux apprentissage	0.0005	0.5[1]
Taille Maximale	100	100
Optimiseur	Adam	SGD
Têtes apprentissage	4	-
Taille Positionwise Feedforward	512	-
Taille Plongements	-	768
Taille Sortie	-	512

contexte de la question en général, alors qu'Enc_{comb} contient des informations spécifiques sur les jetons de la question, ce qui est utile pour aider le décoder à prédire le prochain jeton.

Décodeur. Le décodeur reçoit le lot contenant B requêtes indexées selon le vocabulaire cible, Enc_{conv}, Enc_{comb} et M_{remp}. Il fonctionne de manière très similaire à l'encodeur. La principale différence se trouve dans la couche récurrente. À la sortie de la dernière couche récurrente, aussi exécutée N fois, on obtient pour chaque requête du lot la matrice Dec_{conv} de taille $[L_{req} \times C]$ où L_{req} est la taille de la requête. On la transforme en matrice de taille $[L_{req} \times O]$ où O est la taille du vocabulaire cible (SPARQL). Cette dernière contient les informations des probabilités de génération de chaque jeton du vocabulaire cible et sert à prédire le prochain jeton de la requête. Le décodeur retourne aussi l'attention calculée dans la couche récurrente, qui est une matrice de taille $L_{quest} \times L_{req}$ et qui retourne les scores d'incidence entre chaque jeton de la question et de la requête.

Couche récurrente. La couche récurrente est utilisée dans l'encodeur et le décodeur, avec quelques différences mineures entre les deux. Elle reçoit pour chaque phrase (question ou requête) une matrice de taille $[L \times C]$, qui contient les informations de chaque jeton de la phrase d'entrée ainsi que les informations de leur position.

On commence par faire un du remplissage sur notre matrice d'entrée. Cette étape est nécessaire afin d'éviter que la taille de notre matrice soit réduite par la couche de convolu-

tion. Dans l'encodeur, on ajoute un remplissage de $\lfloor K/2 \rfloor$ de chaque côté de notre phrase, où K est la taille du noyau de convolution utilisé, afin d'obtenir une matrice de taille $[\lfloor K/2 \rfloor + L + (\lfloor K/2 \rfloor \times C)]$. Dans le décodeur, on ajoute un remplissage de $K-1$ au début de notre phrase, afin d'obtenir une matrice de taille $[K-1+L \times C]$. Cette distinction est très importante, puisque dans l'encodeur, elle nous permet de nous assurer que la taille de la phrase reste la même tout au long des convolutions. Dans le décodeur, cela nous permet de nous assurer que le modèle utilise les jetons précédents pour prédire le jeton suivant. Si cette contrainte n'est pas respectée, le décodeur pourra, au moment de prédire le jeton i+1, utiliser ce dernier pour le prédire. Il apprendra alors simplement à le copier, au lieu d'apprendre à le générer.

On passe ensuite notre matrice d'entrée étendue à une couche de convolution qui fait doubler la taille de notre couche cachée, ce qui donne une matrice de taille $[L \times C \times 2]$. Cela nous permet d'utiliser une couche de *Gated Linear Units* (GLU) [42], qui vient diviser la dernière dimension par 2, ce qui nous donne une matrice de taille $[L \times C]$. GLU est une couche d'activation qui utilise une fonction sigmoïde afin de calculer la couche cachée après une convolution.

Dans le décodeur, la matrice résultante de la couche GLU est utilisée pour calculer l'attention, avec la matrice de plongements du décodeur T_{emb} et les matrices sortant de l'encodeur.

Finalement, puisque le modèle utilise plusieurs dimensions de couches cachées tel que défini dans le tableau 3.1, on projette si nécessaire la dimension de la couche cachée de la matrice résultante dans la prochaine dimension cachée.

3.2.2 Modèle *Seq2Seq Transformer*

Le modèle Transformer est de type *Seq2Seq* et utilise des *transfomers* [22] comme encodeur et décodeur. La table 3.1 fait le détail de la configuration de notre implémentation, qui est basée sur l'architecture *transformer_iwslt_de_en* [43] implémentée par FairSeq [6], initialement optimisé pour la tâche de traduction automatique de l'allemand à l'anglais.

Encodeur. L'encodeur reçoit le lot contenant B questions indexées avec le vocabulaire source. Il reçoit également en entrée le masque de remplissage M_{remp}, qui est calculé préalablement. Chaque phrase du lot traverse les étapes suivantes. De manière similaire à l'encodeur CnnS2S, on calcule pour chaque jeton un plongement de taille E, qu'on multiplie par un facteur de mise à l'échelle $s = \sqrt{Dim.CoucheCachée}$. Cette opération permet un entraînement plus fiable ainsi qu'une réduction de la variance dans les *plongements*. On additionne ces derniers à un vecteur de position ayant la forme $[0, 1, ...TAILLE_MAX]$, où TAILLE_MAX

est une constante prédéterminée représentant le nombre de jetons maximal qu'une requête peut avoir. Cela nous donne la matrice T_{emb}, qu'on passe à la couche récurrente de l'encodeur, exécutée N=6 fois. À chaque itération, la couche récurrente calcule d'abord l'autoattention entre chaque jeton de la question, puis fait un *dropout* sur le résultat. On additionne le résultat du *dropout* à la matrice source, puis on normalise le résultat pour obtenir la matrice $M_{quest+attn}$, qui contient les informations de position et d'plongements augmentées des résultats de l'attention.

On passe la matrice $M_{quest+attn}$ dans une couche de *PositionWise FeedForward*. On additionne la matrice résultante à $M_{quest+attn}$, avant de renormaliser le tout. La couche récurrente retourne une matrice de taille $L_{quest} \times H$, où H est la dimension de la couche cachée.

L'encodeur retourne la sortie de la dernière couche récurrente, soit une matrice de taille $L_{quest} \times H$ pour chaque phrase du lot encodant la signification de cette dernière.

Décodeur. Le décodeur reçoit le lot contenant B requêtes indexées selon le vocabulaire cible, la sortie de l'encodeur. À la différence du CnnS2S, il reçoit également un masque de remplissage pour les requêtes. Cependant, le masque de remplissage des requêtes remplit la même fonction que le remplissage dans le CnnS2S, c'est-à-dire empêcher le modèle de prédire le prochain jeton en connaissant la réponse.

La seule différence entre l'encodeur et le décodeur *transformers* est dans la couche récurrente. En effet, après avoir utilisé l'autoattention pour calculer la matrice $M_{req+attn}$, on calcule l'attention entre la question et la requête en utilisant la sortie de l'encodeur. On fait encore une fois un *dropout* sur le résultat de l'attention, qu'on additionne à la matrice $M_{req+attn}$ avant de normaliser le tout.

Après la dernière couche récurrente, qui retourne une matrice de dimension $[L_{req} \times H]$, on utilise une couche linéaire pour obtenir une matrice de dimension $[L_{req} \times O]$, où O est la taille du vocabulaire cible (SPARQL), contenant les probabilités que chaque jeton de ce vocabulaire soit le prochain jeton.

3.3 Architecture de copie

Cette section présente les modifications nécessaires pour intégrer une couche de copie permettant aux architectures de type *Seq2Seq* définies plus haut de copier des jetons directement de la phrase source (question) vers la phrase cible (requête).

La figure 3.4 montre comment le système de traduction automatique change avec la copie. Plus précisément, le décodeur reçoit maintenant la phrase source indexée en plus de la sortie

de l'encodeur. La figure 3.5 montre comment la couche de copie se place dans la structure encodeur-décodeur propre aux modèles de type *Seq2Seq*.

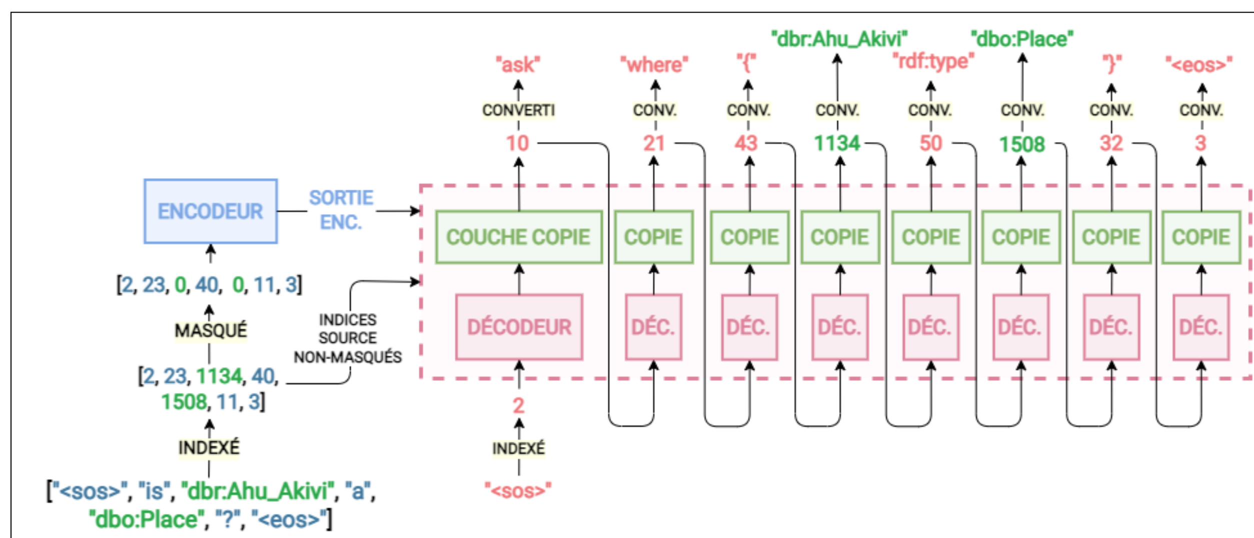


Figure 3.4 Architecture générale d'un modèle *Seq2Seq* avec copie

3.3.1 Vocabulaires

Dans les architectures de base (sans copie), le vocabulaire source (anglais) comprend chaque jeton présent dans les questions, et le vocabulaire cible (SPARQL) comprend chaque jeton présent dans les requêtes. Les jetons sont ajoutés aux vocabulaires sans ordre particulier.

Or, la couche de copie sert à donner au modèle la capacité de copier directement de la question des jetons qui seraient normalement hors du vocabulaire (et donc impossibles à générer). Dans l'ensemble d'entraînement, il faut donc implémenter un moyen de différencier les jetons qui font partie des vocabulaires de base (que le modèle apprendra à générer) et les jetons qu'on veut plutôt apprendre à copier. Par exemple, en traduction SPARQL, on veut copier les URIs référençant des éléments de la base de connaissances, c'est-à-dire les jetons qui commencent par **dbo :**, **dbr :**, **dbp :**, **dbc :**, **geo :**, **georss :** ou **dct :**.

De plus, comme le modèle reçoit des vecteurs d'indices et non de chaînes de caractères, les jetons copiés de la phrase source à la phrase cible doivent avoir le même indice dans les vocabulaires source et cible. Durant l'entraînement, les modèles ne voient que les indices des jetons, puisque la transformation du vecteur d'indices en phrase est effectuée en post-traitement. Or, si le modèle copie l'indice i de la question vers la requête, il faut que cet indice ait la même signification dans les vocabulaires source et cible afin que la transformation

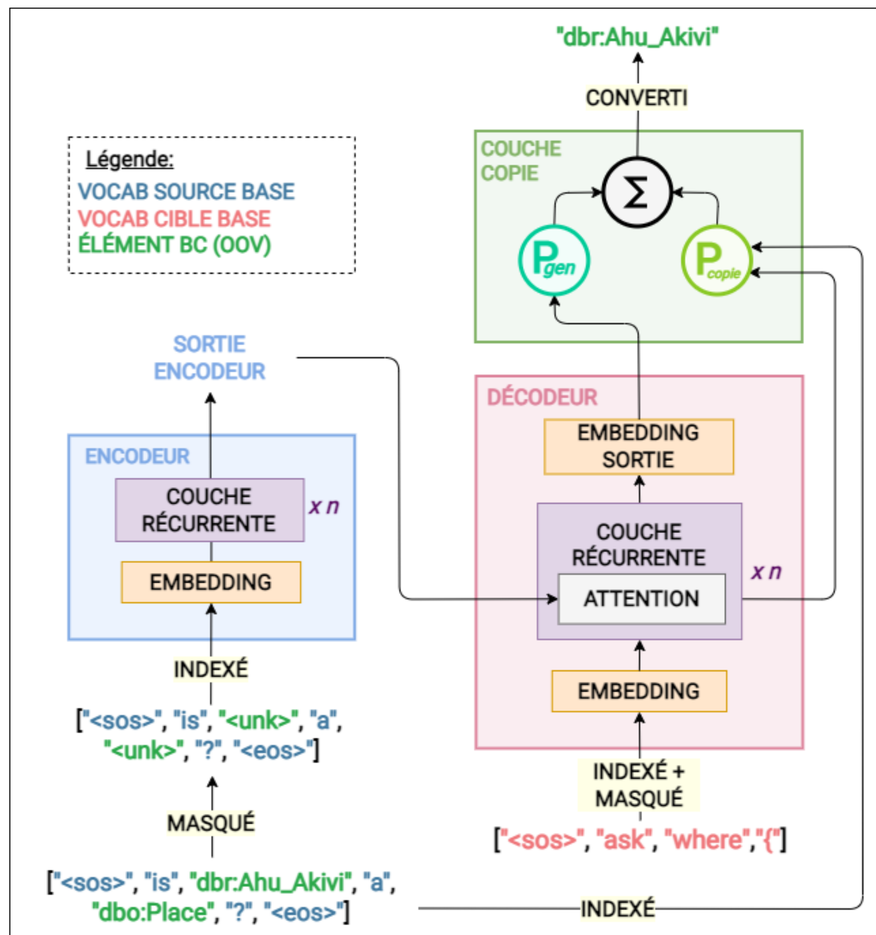


Figure 3.5 Architecture détaillée d'un encodeur et d'un décodeur *Seq2Seq* avec copie

d'indices en phrase soit faite correctement une fois la requête générée.

Afin de tenir compte de ces contraintes, nous commençons par créer un vocabulaire source de base et un vocabulaire cible de base qui ne contiennent que les jetons de base des langues source et cible, sans ordre particulier. Les vocabulaires de base ne contiennent aucun jeton que le modèle doit apprendre à copier. Pour la traduction vers SPARQL, par exemple, ils ne contiendront aucun URI.

Nous ajoutons des jetons de remplissage aux vocabulaires de base afin qu'ils aient la même taille, puis nous conservons cette taille de base en mémoire. Il s'agit de l'indice de coupure idx_{taille}, après lequel les jetons du vocabulaire sont des éléments que le modèle doit apprendre à copier de la source et non à générer.

Ensuite, nous extrayons tous les éléments que le modèle doit apprendre à copier à travers toutes les données, sans distinction de la langue. Par exemple, pour la traduction SPARQL, nous aurons la liste de tous les URIs référencés dans les questions ou les requêtes. Avec ces éléments à copier, nous créons une extension de vocabulaire, qui est concaténée à chaque vocabulaire de base afin de créer nos vocabulaires source et cible.

Ayant gardé en mémoire l'indice de coupure idx_{taille} des vocabulaires de base, on peut rapidement déterminer que chaque jeton ayant un indice au-dessus de cette valeur représente un élément à copier. Pendant l'inférence, si un nouvel élément à copier est rencontré, nous pouvons simplement l'ajouter à la fin de nos vocabulaires source et cible. Conceptuellement, il s'agit toujours de jetons hors du vocabulaire de base, que nous enregistrons temporairement afin de garder la trace de leur signification.

3.3.2 Couche de copie

Le but de la couche de copie est de donner au modèle la capacité de copier des jetons directement de la phrase source. Cela permet aux modèles de ne plus être limités par des vocabulaires de taille fixe, puisque n'importe quel jeton se trouvant dans la source peut maintenant être utilisé.

Dans une architecture augmentée avec de la copie, l'encodeur et le décodeur reçoivent des vecteurs source et cible masqués, ce qui signifie que tout jeton avec un indice supérieur à la taille des vocabulaires de base idx_{taille} est remplacé par l'indice du jeton inconnu (0).

La couche de copie est implémentée directement après le décodeur. Elle prend en entrée la phrase source indexée et non masquée ainsi que la sortie du décodeur, soit les scores d'attention et le tenseur de dimensions $[B \times L_{req} \times O]$ contenant la probabilité de générer chaque jeton du vocabulaire cible de base. Porté sur l'architecture Transformer par [37,

38], nous avons pu l'adapter à ConvS2S puisque les deux architectures génèrent des scores d'attention multitêtes.

D'abord, nous identifions s'il y a des éléments hors du vocabulaire parmi les indices des jetons de la phrase source encodée en utilisant l'indice de coupure idx_{taille}. Si c'est le cas, nous étendons le tenseur de probabilité de sortie pour inclure ces indices supplémentaires et leur attribuer initialement une probabilité de génération de 0. On a donc un tenseur de dimensions $[B \times max(src_idx) \times O]$. Ensuite, nous calculons la probabilité de génération de chaque jeton, qui est le softmax du tenseur de probabilité. En utilisant les scores d'attention, nous calculons également la probabilité que chaque jeton soit copié directement de la phrase source. En suivant l'implémentation de [37], nous calculons également un facteur $\alpha_{bal} \epsilon [0, 1]$ servant à équilibrer les probabilités de copie et de génération, définit à l'équation 3.2. Ici, Q, K et V sont les tenseurs *query*, *key* et *value* utilisés pour calculer l'attention, et W^{T} est un paramètre appris.

$$A_t = Q^T * K \tag{3.1}$$

$$\alpha_{bal} = sigmoid(W^T * (A_t^T * V)) \tag{3.2}$$

Finalement, on calcule la somme des deux probabilités équilibrée par ce facteur afin d'obtenir le tenseur final contenant la probabilité que chaque jeton (du vocabulaire cible et des jetons hors du vocabulaire présents dans la source) soit le suivant.

CHAPITRE 4 DONNÉES ET MÉTRIQUES

Ce chapitre fait le détail des ensembles de données utilisés pour l'entraînement, la validation et le test, ainsi que la manière dont ils sont générés. De plus, on y détaille les métriques utilisées pour tester les modèles.

4.1 Ensembles de données de l'état de l'art

Cette section décrit les versions originales des ensembles de données utilisés, tel qu'ils sont définis dans l'état de l'art.

4.1.1 Survol des données

La table 4.1 présente un survol des caractéristiques principales des ensembles de données Monument et LC-QuAD, qui sont décrits dans les sections suivantes. Les trois premières lignes représentent la taille des différents ensembles d'entraînement, de validation et de test. La seconde section du tableau fait le détail des patrons utilisés pour générer les ensembles de données. Tels que définis à la section 2.1.4, les patrons sont des paires question-requêtes trouées qu'on peut utiliser pour générer des ensembles de données en remplaçant les paramètres substituables par des éléments de la base de connaissances. En comptant chaque paramètre substituable comme un jeton, le *plus long patron* représente le patron de question avec le plus de jetons, et le *plus court patron* est celui avec le moins de jetons. La ligne *max params substituables* représente le nombre maximal de paramètres substituables qu'on trouve dans un même patron. Ces derniers sont les jetons qui seront remplacés par des éléments de la base de connaissances lors de la génération. La plus longue question et la plus longue requêtes sont la question et la requête comportant le plus de jetons de l'ensemble.

La ligne *Elems BC entraînement* rapporte le nombre d'éléments de la base de connaissances utilisés dans l'ensemble d'entraînement, et la ligne *Elems BC test* rapporte leur nombre dans l'ensemble de test. Finalement, le taux d'intersection calcule la proportion des éléments de la base de connaissances dans l'ensemble de test qui ont été déjà vus dans l'ensemble d'entraînement. Plus précisément, un taux d'intersection de 1 signifie que tous les éléments de l'ensemble de test ont déjà été vus au moins une fois pendant l'entraînement, tandis qu'un taux d'intersection de 0 signifie que tous les éléments de l'ensemble de test sont inconnus du modèle.

Tableau 4.1 Survol des ensembles de données utilisés

	Mon	Mon50	Mon80	TNTSPA	LC-QuAD
Taille entraînement	11831	11831	11831	4000	4000
Taille validation	1479	1479	1479	500	500
Taille test	1478	1478	1478	500	500
Plus long patron	8	8	8	15	15
Plus court patron	1	1	1	6	6
Max params substituables	2	2	2	4	4
Plus longue question	24	24	24	25	27
Plus longue requête	31	31	31	25	23
Elems BC entraînement	1797	1787	1791	4153	4150
Elems BC test	815	825	816	1045	1066
Taux d'intersection	0.928	0.925	0.925	0.704	0.713

4.1.2 Monument

L'ensemble de données Monument est produit par les auteurs des NSpMs [7]. Chaque entrée est composée d'une question et de sa traduction en requête SPARQL intermédiaire. Les données sont générées avec des patrons troués, comme décrit à la section 2.1.4. Les questions de Monument portent sur toutes les sous-classes et propriétés de la classe de DBpedia **Monument**, qui contient des informations sur différents monuments historiques, architectes, endroits, etc.

Versions

Les auteurs originaux [7] proposent 38 patrons et deux versions de leur ensemble de données : Monument 300, qui a 300 exemples par patron, et Monument 600, qui a 600 exemples par patron. Cependant, la revue de littérature réalisée par [5] propose trois versions de Monument - Monument, Monument50 et Monument80 - qui sont générées en utilisant seulement 33 des 38 patrons disponibles. C'est cette version des jeux de données que nous utilisons, puisque nous souhaitons comparer avec précision nos architectures à celles de l'état de l'art évaluées par la revue de littérature. La version originale et celle de la revue utilisent les mêmes patrons.

Composition

Les auteurs de la revue [5] indiquent qu'ils génèrent chaque version de Monument en utilisant 600 exemples par patron. Cependant, ils indiquent aussi que leur ensemble de données com-

prend 14 788 entrées, ce qui ne correspond pas au nombre attendu. Après une analyse plus approfondie, nous avons conclu que seulement la moitié des patrons génèrent 600 exemples, tandis que les autres en génèrent moins. Bien que cela n'affecte pas la reproductibilité des résultats, cette distinction n'est pas clarifiée par les auteurs. La figure 4.1 présente un exemple d'une entrée dans l'ensemble de données.

```
{
  "question": "what is kailashnath mahadev statue all about",
  "interm_sparql": "select var_a where brack_open
                         dbr_Kailashnath_Mahadev_Statue dct_subject var_a
                    brack_close"
}
```

Figure 4.1 Entrée dans l'ensemble de données Monument

Puisque les sous-ensembles d'entraînement, de validation et de test ne sont pas fournis par les auteurs de la revue [5], nous séparons aléatoirement les entrées suivant le ratio 80-10-10 proposé par la revue. Pour les trois versions de monument, nous avons donc un ensemble d'entraînement, un ensemble de validation et un ensemble de test composés respectivement de 11831, 1479 et 1478 entrées.

Utilisabilité

L'avantage de cet ensemble de données est qu'il est relativement simple. En effet, comme démontré par le tableau 4.1, les questions ne sont ni très longues, ni très complexes. De plus, elles n'utilisent pas beaucoup d'éléments de la base de connaissances. Cela en fait un excellent jeu de données pour démontrer la faisabilité d'une nouvelle implémentation. En effet, on peut s'attendre à ce que les modèles entraînés avec Monument atteignent de très hautes performances (score BLEU autour de 95%).

Cependant, sa simplicité est aussi la source de plusieurs inconvénients. Tout d'abord, le taux d'intersection entre l'ensemble d'entraînement et celui de test est très proche de 1, ce qui signifie qu'il n'y a presque aucun élément inconnu dans l'ensemble de test. Ce dernier ne nous donne donc aucune information sur la capacité des modèles entraînés avec Monument à gérer les éléments inconnus. Les questions sont également excessivement simples, et ne sont pas toujours formulées de manière naturelle. En bref, les bons résultats sur ce jeu de données indiquent surtout que l'architecture est fonctionnelle plutôt que capable d'apprendre la tâche.

4.1.3 LC-QuAD

L'ensemble de données LC-QuAD est produit par le groupe de recherche Smart Data Analytics (SDA) [44]. Chaque entrée est générée à partir d'un patron, et est composée d'une question et de sa traduction en requête SPARQL exécutable, ainsi que d'une reformulation de la question par un expert afin qu'elle soit posée de manière plus naturelle. L'ensemble de données couvre plusieurs milliers d'éléments de la base de connaissances DBpedia, sur des domaines variés.

Versions

Il existe actuellement deux versions de LC-QuAD, la v1.0 [36] et la v2.0 [45]. Les auteurs rapportent que la version 1 contient 5000 entrées générées à partir 38 patrons et couvre 5657 ressources et prédicats de la base de connaissances DBpedia. Il s'agit également de la version utilisée dans la revue produite par [5].

La version 2 contient 30000 entrées générées à partir de 22 patrons et couvre 22568 éléments de DBpedia. Lorsque possible, les entrées incluent également la requête équivalente pour tirer la réponse de la base de connaissances WikiData, ce qui est un gros avantage pour entraîner des modèles plus portables. Bien qu'évaluer nos architectures sur la v2.0 sera éventuellement intéressant pour nous, une analyse sommaire des données nous a permis de déterminer que le travail requis pour adapter notre méthodologie à cette version est hors du sujet de ce mémoire. Qui plus est, la version 2 contient plus d'entrées générées à partir de moins de patrons que la version 1, alors que nous avons besoin d'ensembles de données avec plus de formulations variées afin de tester les limites de notre architecture.

Les auteurs proposent une répartition des entrées composant LC-QuADv1.0 dans des ensembles d'entraînement et de test prédéfinis. Nous référons à cette configuration comme étant l'ensemble de données **LC-QuAD**. Cependant, pour des raisons non spécifiées dans leur article, les auteurs de la revue [5] proposent leur propre répartition des entrées. Nous référons à cette configuration comme étant l'ensemble de données **TNTSPA**, d'après le nom de leur dépôt GitHub [2].

Composition

Comme décrit plus haut, LC-QuADv1.0 contient 5000 entrées générées à partir de patrons, séparées en un ensemble d'entraînement de 4000 entrées et d'un ensemble de test de 1000 entrées. Après une analyse approfondie, il apparaît que seulement 33 des 43 patrons fournis

2. `https://github.com/xiaoyuin/tntspa`

par les auteurs sont utilisés pour générer des données, et non 38 tel que rapporté par les auteurs. Suivant le ratio 80-10-10, nous séparons l'ensemble de test fourni en 2 afin d'avoir un ensemble de validation et un ensemble de test de 500 entrées chacun.

Un aspect important de cet ensemble de données est qu'en plus de contenir la paire question-requête générée par le patron, chaque entrée contient également une question reformulée par un expert pour prendre une forme plus naturelle. On désigne comme *question intermédiaire/intermediary question* la question générée par le patron, et *question reformulée/corrected question* la question reformulée par un humain. La figure 4.2 montre un exemple d'entrée dans LC-QuAD. On remarque que les éléments de la base de connaissances sont identifiés par des chevrons dans la question intermédiaire et dans la requête, mais pas dans la question reformulée.

```
{
  "_id": "1573",
  "template_id": 2,
  "corrected_question": "What is the currency of Kerguelen Islands ?",
  "intermediary_question": "What is the <used money> of Kerguelen Islands ?",
  "sparql_query": "SELECT DISTINCT ?uri WHERE {
                     <http://dbpedia.org/resource/Kerguelen_Islands>
                     <http://dbpedia.org/ontology/currency> ?uri
                   }"
}
```

Figure 4.2 Une entrée dans la version originale de LC-QuAD

Nous utilisons par conséquent deux versions de l'ensemble de données LC-QuAD. Premièrement, nous utilisons l'ensemble **LC-QuAD Questions intermédiaires** (parfois raccourci en *Qsts interm.*), qui est composé de paires question-requêtes dans lesquelles la question est la *question intermédiaire* et la requête est la version encodée en SPARQL intermédiaire de la requête SPARQL fournie dans le jeu de données. Deuxièmement, nous utilisons l'ensemble **LC-QuAD Questions reformulées** (parfois raccourci en *Qsts reformulées*), qui est composé de paires question-requêtes dans lesquelles la question est la *question reformulée* et la requête est la version encodée en SPARQL intermédiaire de la requête SPARQL fournie dans le jeu de données.

Pour la version **TNTSPA**, les auteurs de la revue [5] fournissent un ensemble d'entraînement de 4000 entrées et un ensemble de test de 500 entrées, mais pas d'ensemble de validation. Par conséquent, nous construisons l'ensemble de validation avec les entrées de LC-QuADv1.0

n'ayant pas été utilisées dans les sous-ensembles fournis dans la version de TNTSPA. En bref, on déduit l'ensemble de validation en utilisant la différence entre LC-QuADv1.0 et TNTSPA. Les entrées de TNTSPA sont seulement composées de la question reformulée et de la requête encodée en SPARQL intermédiaire par les auteurs de la revue. La figure 4.3 montre un exemple d'entrée dans LC-QuAD. Puisqu'il s'agit d'une version non officielle de LC-QuAD et que nous l'utilisons seulement pour vérifier que nos architectures sont capables de reproduire les résultats attendus, nous ne générons pas de versions corrigées ou étiquetées de cet ensemble de données.

```
{
  "question": "what is the currency of kerguelen islands ?",
  "interm_sparql": "SELECT DISTINCT var_uri WHERE brack_open
                     <dbr_Kerguelen_Islands> <dbo_currency> var_uri
                    brack_close"
}
```

Figure 4.3 Une entrée dans la version fournie par TNTSPA

Utilisabilité

L'avantage de LC-QuAD est qu'il couvre un très large éventail d'éléments de la base de connaissances, ce qui nous permet d'évaluer la flexibilité des modèles. De plus, les questions reformulées sont un des avantages de l'ensemble de données, car elles fournissent des formulations plus variées et naturelles de chaque patron. Finalement, le taux d'intersection entre les éléments de la base de connaissances référencés dans l'ensemble d'entraînement et ceux référencés dans l'ensemble de test est plus bas que pour d'autres ensembles de données, ce qui fait que les performances obtenues sur cet ensemble de données sont un peu plus représentatives des capacités réelles des modèles.

4.2 Ensembles de données corrigés

Cette section fait le détail de la génération d'une version corrigée des ensembles de données de l'état de l'art. En effet, au cours de notre recherche, nous avons rencontré plusieurs problèmes avec l'utilisation de ces ensembles de données. Il a donc été nécessaire d'en générer une version corrigée afin d'avoir l'heure juste quant à la performance des architectures de l'état de l'art, puisqu'il y a une corrélation directe entre la qualité des données vues par un modèle et la qualité de ses prédictions.

4.2.1 Défauts dans les données de l'état de l'art

Cette section fait le détail des différents défauts présents dans les ensembles de données originaux qu'on souhaite corriger.

Encodage non-uniforme en SPARQL intermédiaire. Tel que décrit dans la section 2.7.5, le manque de règle de transformations claires entre le SPARQL exécutable et le SPARQL intermédiaire crée un certain manque d'uniformité à travers les méthodes d'encodage utilisées par les différents auteurs. Ce problème est la source de plusieurs erreurs dans les données originales.

```
{
  "question": "what is the relegious affiliations of katyayana ?",
  "interm_sparql": "SELECT DISTINCT var_uri WHERE brack_open
                    <dbr_Katyayana_ attr_open Buddhist attr_close math_gt
                      <dbp_religion> var_uri
                  brack_close",
  "pure_sparql": "SELECT DISTINCT ?uri WHERE {
                  <dbr:Katyayana_(Buddhist)>
                    <http://dbpedia.org/property/religion> ?uri
                } "
}
```

Figure 4.4 Une erreur d'encodage dans TNTSPA

Prenons par exemple la figure 4.4, qui montre une entrée de la version TNTSPA de l'ensemble LC-QuAD dans lequel l'encodage n'est pas réalisé correctement. En rouge, le chevron fermant, utilisé pour identifier les éléments de la base de connaissances, est encodé comme représentant un symbole de comparaison mathématique. En vert, on voit l'encodage des chevrons réalisé correctement.

```
{
  "question": "how many place does hisar have",
  "interm_sparql": "select count par_open wildcard par_close where brack_open
                      var_a rdf_type dbo_Place sep_dot
                      var_a dbo_location dbr_Hisar_ par_open city par_close
                   brack_close group by var_a",
  "pure_sparql": "select count ( * ) where {
                   ?a rdf:type dbo:Place .
                   ?a dbo:location dbr:Hisar_(city)
                 } group by ?a"
}
```

Figure 4.5 Une erreur d'encodage dans Monument80

Dans la figure 4.5, on voit une entrée de Monument 80 encodée de manière erronée. Lors de

l'encodage en SPARQL intermédiaire, on doit faire la différence entre les parenthèses faisant partie de la syntaxe (i.e., `par_open` et `par_close`) et celles reliées aux URIs des éléments de la base de connaissances (i.e., `attr_open` et `attr_close`). En vert, on voit un encodage correct des parenthèses, puisqu'elles sont associées à l'expression `SPARQL COUNT`. En rouge, on voit un usage incorrect des parenthèses, puisqu'elles sont utilisées pour encoder l'URI `dbr:Hisar_(city)`. Dans ce cas, on aurait dû utiliser `attr_open` et `attr_close`.

Défauts de formatage. À travers les ensembles de données, nous avons rencontré de nombreux défauts de formatage des entrées. Par exemple, certaines entrées de LC-QuADv1.0 font référence à des patrons inexistants ou qui n'ont pas été rendus disponibles par les auteurs. Dans certaines entrées, les questions en langue naturelle contiennent des liens ou des URIs. Dans d'autres, la requête n'a aucun lien avec la question, ou le champ contenant la question en contient deux. Certaines questions sont tout simplement incomplètes. Pour un certain patron, les éléments de la base de connaissances sont incomplètement identifiés dans la question intermédiaire. Certaines questions reformulées sont identiques à leur homologue intermédiaire, ou reformulées avec une syntaxe incorrecte. Enfin, il y a de nombreuses fautes de frappe dans les questions, ce qui, selon les auteurs [36], permet au modèle d'être plus flexible face à l'erreur humaine. Il s'agit d'une hypothèse raisonnable ; toutefois, nous avons corrigé les coquilles liées à des ressources de la base de connaissances (par exemple, *Ricky Grevais* au lieu de *Ricky Gervais*). La figure 4.6 présente des exemples d'erreurs dans le format des entrées.

4.2.2 Corrections apportées

Plusieurs corrections ont été apportées aux ensembles de données afin de corriger les défauts identifiés à la section précédente.

Correction du SPARQL intermédiaire. Pour les ensembles de données dont le SPARQL intermédiaire était déjà généré, nous avons appliqué une méthodologie de correction simple permettant de faciliter la transformation en SPARQL exécutable. Prenons pour exemple la requête suivante déjà encodée en SPARQL intermédiaire, qui contient toutes les erreurs (identifiées en rouge) que nous souhaitons corriger.

```
count attr_open var_s attr_close where brack_open
    dbr_Seinfeld par_open TV_show par_close dbp_seasonvar_s.
    var_e dbp_episode var_s
brack_close filter attr_open var_emath_gt10 attr_close
```

```
{
  "corrected_question": "Does the white river flow into the connecticut
                          river{u'_id': u'97e02dcf44aa43c1b7cc7a7c155b118f',",
  "intermediary_question": "Is <White River (Vermont)> the <right tributary> of
                             <Connecticut River>?",
  "pure_sparql": "ask where {
                    dbr:Connecticut_River dbo:rightTributary
                      dbr:White_River_(Vermont) }"
}
```

(a) Entrée dont la question contient du bruit

```
{
  "corrected_question": "List down the schools whose mascot is an animal from
                          the order of Even toed Ungulates?",
  "intermediary_question": "What are the <companies> whose <programming language>'s
                             <designer> is <Bjarne Stroustrup>?",
  "pure_sparql": "select distinct ?uri where {
                    ?x dbp:designer dbr:Bjarne_Stroustrup .
                    ?uri dbp:programmingLanguage ?x .
                    ?uri rdf:type dbo:School }"
}
```

(b) Entrée dont la question ne correspond pas à la requête, et dont la requête est erronée

Figure 4.6 Exemples de défauts dans LC-QuAD

Pour commencer, nous utilisons une expression régulière nous permettant d'identifier les parties de la requête qui sont de la syntaxe SPARQL et les parties qui sont des URIs représentant des éléments de la base de connaissances. Dans la partie de la syntaxe générale, nous appliquons les corrections suivantes. Tout d'abord, nous remplaçons toutes les parenthèses par `par_open` et `par_close` (démontré en rose) afin de bien les différencier des parenthèses dans les URIs. Ensuite, nous remplaçons aussi tous les points qui ne sont pas correctement remplacés par `sep_dot` (démontré en bleu). Finalement, nous corrigeons les espaces dans les filtres mathématiques et dans les variables qui sont mal encodées (démontré en vert). Après ces premières corrections, la requête est partiellement corrigée, comme suit.

```
COUNT par_open var_s par_close WHERE brack_open
    dbr_Seinfeld par_open TV_show par_close dbp_season var_s sep_dot
    var_e dbp_episode var_s
brack_close filter par_open var_e math_gt 10 par_close
```

Puis, pour chaque URI capturé avec l'expression régulière, on s'assure que les parenthèses sont encodées correctement avec `attr_open` et `attr_close` (démontré en bleu ciel). On s'assure également que, si l'URI de l'élément contient des points (par exemple, pour la ressource `dbr_Mode._Set._Clear.`), ils ne sont pas incorrectement encodés par le symbole `sep_dot`. La requête corrigée prend donc la forme finale suivante.

```
COUNT par_open var_s par_close WHERE brack_open
    dbr_Seinfeld attr_open TV_show attr_close dbp_season var_s sep_dot
    var_e dbp_episode var_s
brack_close filter par_open var_e math_gt 10 par_close
```

Format des entrées. Pour les entrées présentant des erreurs de formatage, nous avons focalisé sur la correction des erreurs relatives aux éléments de la base de connaissances de type ressource afin de faciliter la recherche future, puisqu'il est important de pouvoir les identifier dans la question. Étant donné que les ressources ont généralement un nom unique, leur URI est très proche de leur étiquette en langue naturelle, ce qui nous permet de les identifier relativement facilement dans une question en langue naturelle. Cette particularité nous permet d'identifier toutes les entrées dans lesquelles les ressources référencées dans la requête n'ont pas toutes un équivalent dans la question. Nous pouvons alors facilement faire ressortir les éléments à problème, par exemple une question mal formulée ou une faute de frappe. Nous avons dû corriger à la main plus d'une dizaine de ressources, et nous avons corrigé toutes les fautes de frappe relatives aux ressources que nous avons pu trouver.

4.2.3 Encodage en SPARQL intermédiaire uniformisé

Afin d'obtenir des requêtes en SPARQL intermédiaire uniformes à travers les ensembles de données, nous appliquons la méthodologie suivante afin de transformer une requête SPARQL exécutable en SPARQL intermédiaire. Cette méthodologie est non seulement utile pour les ensembles de données tels que LC-QuAD qui ne fournissent que les requêtes SPARQL exécutables, mais ces règles d'encodage sont également applicables aux ensembles de données qui fournissent des requêtes SPARQL intermédiaires afin de les uniformiser.

Transformations directes. Les correspondances entre les éléments du SPARQL exécutable et ceux du SPARQL intermédiaire sont détaillées dans la table A.1, présentée en annexe. Dans les règles d'encodage, nous ajoutons le symbole `sparql_quote`, représentant un guillemet simple, qui sert à différencier les guillemets et apostrophes utilisés dans les URIs de ceux utilisés dans la syntaxe SPARQL. Par exemple, la fonction `REGEX(?a, 'abc', 'i')`, une fois encodée, prend la forme suivante.

```
REGEX par_open
     var_a , sparql_quote abc sparql_quote , sparql_quote i sparql_quote
par_close.
```

Afin de nous assurer de l'uniformité des jetons de syntaxe SPARQL à travers les ensembles de données, nous mettons en minuscule toute partie des requêtes qui n'est pas un URI d'élément de la base de connaissances. Il est important de garder la casse originale des URIs, puisque la base de connaissances leur est sensible.

Encodage des ressources. Un aspect important de notre méthodologie d'encodage est que nous encodons les URIs en un seul jeton. En effet, selon l'encodage proposé par les auteurs de [7], les URI des éléments de la base de connaissances sont souvent encodés avec plusieurs jetons au lieu d'un seul. Par exemple, selon la méthodologie originale, la ressource

```
dbr:(Untitled)_Blue_Lady
```

est encodée comme

```
dbr_ attr_open Untitled attr_close _Blue_Lady
```

Ce mode d'encodage présente plusieurs problèmes. Premièrement, au lieu d'utiliser un seul jeton, on en utilise 5 : `[dbr_, attr_open, Untitled, attr_close, _Blue_Lady]`. Cela représente 5 jetons différents que notre modèle devra apprendre à comprendre individuellement,

puis à combiner correctement. Aussi, cette manière d'encoder complique énormément la tâche d'identifier quels sont les éléments de la base de connaissances référencés dans une requête. Il est facile d'exclure accidentellement le jeton `_Blue_Lady` si on consière qu'`attr_close` marque la fin de l'URI. Par conséquent, lors de l'encodage en SPARQL intermédiaire, nous n'encodons pas en SPARQL intermédiaire les symboles qui font partie d'un URI, sauf pour réduire leur préfixe.

4.2.4 Survol des modifications

Afin de résumer les modifications apportées à chaque étape, considérons la requête en SPARQL exécutable suivante.

```
COUNT(?song) WHERE {
    ?song dbp:writtenBy dbr:Primus_(band)
}
```

Selon la méthodologie originale d'encodage en SPARQL intermédiaire proposée par [7], la requête encodée prend la forme suivante.

```
COUNT par_open var_song par_close WHERE brack_open
    var_song dbp_writtenBy dbr_Primus_ attr_open band attr_close
brack_close
```

Puis, après avoir appliqué notre méthodologie d'uniformisation, la forme finale de la requête en SPARQL intermédiaire est la suivante.

```
count par_open var_song par_close where brack_open
    var_song dbp_writtenBy dbr_Primus_(band)
brack_close
```

Comme on peut le voir, nous n'utilisons pas d'étapes complexes ou de nouveaux concepts. Cependant, nous considérons qu'il est important de détailler et de formaliser les étapes d'encodage et le résultat attendu dans le but d'uniformiser les modes d'encodage en SPARQL intermédiaire. La table 4.2 démontre l'impact de nos corrections sur les questions en langue naturelle et les requêtes en SPARQL intermédiaires. Les requêtes de LC-QuAD n'ont pas besoin d'être corrigées, puisqu'elles sont en SPARQL exécutable. Nous les encodons en SPARQL intermédiaire selon la méthodologie décrite à la section suivante.

Tableau 4.2 Nombres d'entrées affectées par la correction

	Questions	Requêtes
Mon	0.29%	30.77%
Mon50	0.29%	30.77%
Mon80	0.29%	30.77%
Qsts reformulées	74.04%	-
Qsts intermédiaires	74.18%	-

4.3 Intégration des éléments de la base de connaissances

L'intuition derrière l'intégration du mécanisme de copie décrit à la section 3.3 est que, si le modèle peut directement copier les éléments de la base de connaissances depuis la question en langue naturelle dans la requête, il n'aura plus besoin d'apprendre la signification d'éléments qu'il pourrait simplement copier de la question. Or, cela signifie qu'il faut implémenter une manière d'intégrer le schéma directement dans la question.

4.3.1 Résultat attendu

On veut donc générer une version *étiquetée* de chaque ensemble de données, dans laquelle les expressions relatives à des éléments d'une base de connaissances sont remplacées par les URIs qu'elles représentent, afin que le modèle puisse les récupérer directement de la question. La figure 4.7 montre un exemple du résultat attendu.

```
{
  "question": "which show is John Oliver
    the host of ?",
  "sparql": "SELECT ?uri WHERE {
    dbr:John_Oliver dbp:hostOf ?uri . }"
}
```

(a) Exemple d'une entrée non-étiquetée

```
{
  "uri_question": "which show is
    dbr:John_Oliver the dbp:hostOf ?",
  "sparql": "SELECT ?uri WHERE {
    dbr:John_Oliver dbp:hostOf ?uri . }"
}
```

(b) Exemple d'une entrée étiquetée

Figure 4.7 Méchanisme d'étiquetage

Il est important de clarifier que le but de cette recherche n'est pas de développer une manière automatique et extensible de d'étiqueter des questions en langue naturelle, une tâche pour laquelle un réseau neuronal serait mieux adapté qu'un simple algorithme. Notre objectif est plutôt de déterminer si l'intégration d'un mécanisme de copie représente ou non une amélioration. De ce fait, la considération principale de notre implémentation est que notre algorithme

d'étiquetage remplace correctement le plus d'éléments de la base de connaissances possible sans introduire de faux positifs. Pour réaliser cela, l'utilisation des informations fournies par les patrons nous permet de remplacer tous les éléments correctement, sans exception.

4.3.2 Méthodologie

Nous commençons par ajouter manuellement à chaque patron de l'information sur l'ordre dans lequel les éléments sont référencés dans la question et dans la requête. La figure 4.8 montre un exemple d'un patron qui contient des informations sur l'ordre des éléments du patron de la requête SPARQL (clef *template*). Par exemple, l'élément *territory* occupe le premier paramètre substituable (<1>) dans la question, mais vient en deuxième dans la requête. À l'inverse, l'élément *Tonkin Campaing* occupe le second paramètre substituable (<2>) dans la question, mais est référencé en premier dans la requête.

```
{
  "id": 2,
  "question_template": "what is the <1|property> of <2|range:resource> ?",
  "template": "SELECT DISTINCT ?uri WHERE { <2> <1> ?uri } ",
  "type": "vanilla"
}
```

(a) Patron contenant des informations sur l'ordre des éléments

```
{
  "_id": "2786",
  "template_id": 2,
  "intermediary_question": "What is the <territory> of <Tonkin Campaign> ?",
  "sparql_query": "SELECT DISTINCT ?uri WHERE {
                     <dbr:Tonkin_Campaign> <dbo:territory> ?uri
                   }"
}
```

(b) Exemple d'une entrée générée selon le patron

Figure 4.8 Ordre des éléments référencés dans une requête, suivant les informations contenues dans le patron associé

Connaître la correspondance entre le placement des éléments de la base de connaissances dans la question et ceux dans la requête est un gros avantage. En effet, cela nous permet, selon l'indicateur de correspondance (dans l'exemple, <1> et <2>), de faire un simple remplacement de l'étiquette en langue naturelle par l'URI pour chaque élément de la base de connaissances référencé dans la question.

Par contre, le fait que cet algorithme d'étiquetage soit dépendant des patrons veut dire qu'il n'est pas utilisable sur les paires question-réponse qui ne suivent pas de patron. Par

conséquent, pour le jeu de données LC-QuAD, cette limitation signifie que nous pouvons générer des entrées étiquetées uniquement à partir des questions intermédiaires, qui sont générées par les patrons, et non à partir des questions reformulées.

4.3.3 Versions étiquetées

Le but des versions étiquetées est non seulement d'intégrer dans la question les URIs des éléments de la base de connaissances que l'on souhaite apprendre à copier, mais aussi de les différentier clairement de ceux qu'on souhaite apprendre à générer, s'il y en a. Pour ce faire, on utilise le symbole `:` (deux-points). En effet, chaque élément d'une base de connaissances s'inscrit dans un espace de nommage (namespace). Le préfixe de chaque URI d'une base de connaissance est son espace de nommage. Par conséquent, dans l'URI d'un élément que l'on souhaite apprendre à copier, on réduit le préfixe avec le deux-points. Au contraire, dans un élément qu'on souhaite apprendre à générer, on réduit le préfixe avec le symbole `_` (trait de soulignement). Par exemple, si un jeton est encodé comme `dbo:child`, la présence du deux-points dans le préfixe (dbo :) nous indique qu'on souhaite apprendre à le copier. S'il est plutôt encodé comme `dbo_child`, l'absence du deux-points dans le préfixe (dbo_) nous indique qu'on veut apprendre à le générer.

Tableau 4.3 Différents préfixes dans les bases de connaissances

Préfixe	**Version étendue**	**Signification**
dbo	http ://dbpedia.org/ontology	Un type (similaire à une classe)
dbc	http ://dbpedia.org/category	Une catégorie (similaire à une classe)
dbr	http ://dbpedia.org/resource	Une ressource (instance d'un type)
dbp	http ://dbpedia.org/property	Une propriété (relation entre deux objets)
dct	http ://purl.org/dc/terms/	Les métadonnées d'un élément de la BC
geo	http ://www.w3.org/2003/01/geo/wgs84_pos#	Une latitude ou une longitude
georss	http ://www.georss.org/georss/	Un point géographique

Plusieurs types d'éléments de la base de connaissances sont utilisés pour construire des requêtes SPARQL. La table 4.3 présente les différents types d'éléments référencés à travers les jeux de données utilisés. On distingue les ressources des autres types, puisqu'elles représentent une instance concrète d'un concept ou d'une classe. On les retrouve également en beaucoup plus grande proportion dans les ensembles de données.

Afin d'analyser plus en profondeur l'impact des différentes composantes de la base de connaissances sur la performance des modèles utilisant des données étiquetés, nous générons des versions des ensembles de données étiquetées de différentes manières. Prenons l'entrée non étiquetée présentée à la figure 4.9 afin de mettre en lumière les différences entre chaque version étiquetée. Les correspondances entre les URIs et les expressions en langue naturelle sont identifiées par des couleurs.

```
{
  "question": "what is the formula one racer whose relatives is ralf schumacher
               and has child is mick schumacher ?",
  "interm_sparql": "select distinct var_uri where brack_open
                      var_uri dbp_relatives dbr_Ralf_Schumacher sep_dot
                      var_uri dbo_child dbr_Mick_Schumacher sep_dot
                      var_uri rdf_type dbo_FormulaOneRacer
                    brack_close"
}
```

Figure 4.9 Une entrée non-étiquetée

Version partiellement étiquetée - ressources. Afin de comprendre l'incidence des ressources, on génère une version des données étiquetées dans laquelle seuls les éléments de type *ressource* (dbr) sont étiquetés. La figure 4.11 montre une entrée dans laquelle seules les ressources sont étiquetées.

```
{
  "uri_question_only_resources": "what is the formula one racer whose relatives is
                                 dbr:Ralf_Schumacher and has child is dbr:Mick_Schumacher ?",
  "uri_interm_sparql_only_resources": "select distinct var_uri where brack_open
                                         var_uri dbp_relatives dbr:Ralf_Schumacher sep_dot
                                         var_uri dbo_child dbr:Mick_Schumacher sep_dot
                                         var_uri rdf_type dbo_FormulaOneRacer
                                       brack_close"
}
```

Figure 4.10 Une entrée dans laquelle seule les ressources sont étiquetées

Il est important de noter que, dans Monument, les paramètres substituables dans les patrons sont uniquement des *types* (dbo) et des *ressources* (dbr). Les autres catégories d'éléments de la base de connaissances font directement partie des patrons. La version de Monument partiellement étiquetée avec les ressources affecte par conséquent les *types* en plus des *ressources*, puisqu'ils remplissent le même rôle dans les patrons. La figure **??** montre une entrée de Monument dans laquelle les *types* (dbo) sont considérés comme des ressources puisqu'ils ne décrivent pas une relation entre deux éléments, mais sont plutôt le sujet de la question.

```
{
  "uri_question_only_resources": "which dbo:Place has the most dbo:Royalty ?",
  "uri_interm_sparql_only_resources": "select var_a count par_open wildcard par_close
                                       as var_c where brack_open
                                          var_a rdf_type dbo:Place sep_dot
                                          var_b rdf_type dbo:Royalty sep_dot
                                          var_b [] var_a
                                       brack_close group by var_a
                                                   order by desc par_open var_c par_close"
}
```

Figure 4.11 Une entrée de Monument dans laquelle seule les ressources et les types sont étiquetées

Version partiellement étiquetée - schéma. Afin de comprendre l'impact des autres types d'éléments de la base de connaissances, on génère une version des données étiquetées dans laquelle tous les éléments du schéma sont étiquetés. Les éléments de type *ressource* sont exclus, puisqu'ils ne font par définition pas partie du schéma. On fait une exception pour la relation de type `rdf_type`, qui est très courante et souvent référencée de manière implicite dans les questions. Par conséquent, on considère le jeton `rdf_type` comme faisant partie de la syntaxe SPARQL de base - on ne veut pas devoir le copier de la question. La figure 4.12 montre une entrée dans laquelle tous les éléments de la base de connaissances sauf les ressources sont étiquetés.

```
{
  "uri_question_all_no_res": "what is the dbo:FormulaOneRacer whose dbp:relatives is
                              ralf schumacher and has dbo:child is mick schumacher ?",
  "uri_interm_sparql_all_no_res": "select distinct var_uri where brack_open
                                      var_uri dbp:relatives dbr_Ralf_Schumacher sep_dot
                                      var_uri dbo:child dbr_Mick_Schumacher sep_dot
                                      var_uri rdf_type dbo:FormulaOneRacer
                                   brack_close"
}
```

Figure 4.12 Une entrée dans laquelle tous les éléments du schéma

Version complètement étiquetée. Dans la version complète, on souhaite que le modèle apprenne à copier tous les éléments du schéma. Encore une fois, cela exclut le jeton `rdf_type`, qui fait plutôt partie de la syntaxe. La figure 4.13 montre une entrée complètement étiquetée.

4.4 Considérations particulières

Dans certaines paires question-requête, chaque élément référencé dans la requête n'a pas nécessairement d'équivalent dans la question. Or, si un modèle ne peut pas apprendre un

```
{
  "uri_question_all": "what is the dbo:FormulaOneRacer whose dbp:relatives is
                      dbr:Ralf_Schumacher and has dbo:child is dbr:Mick_Schumacher ?",
  "uri_interm_sparql_all": "select distinct var_uri where brack_open
                             var_uri dbp:relatives dbr:Ralf_Schumacher sep_dot
                             var_uri dbo:child dbr:Mick_Schumacher sep_dot
                             var_uri rdf_type dbo:FormulaOneRacer
                           brack_close"
}
```

Figure 4.13 Une entrée dans laquelle tous les éléments de la base de connaissances sont étiquetés

jeton par des exemples et ne peut pas le copier directement de la question, il est impossible qu'il le génère dans la traduction SPARQL. De ce fait, un élément n'est étiqueté pour la copie que s'il est présent dans la question et dans la requête de l'entrée dans laquelle il est référencé. La figure 4.14 montre un exemple d'une paire question-requête étiquetée dans laquelle l'élément de base de connaissances **dbo :TelevisionShow** n'est pas référencé dans la question. De ce fait, cet élément ne sera pas marqué pour la copie.

```
{
  "uri_question": "count the dbp:notableworks in dbr:Russell_T_Davies ?",
  "sparql": "select distinct count(?uri) where {
               dbr:Russell_T_Davies dbp:notableworks ?uri .
               ?uri rdf:type dbo_TelevisionShow . }"
}
```

Figure 4.14 Entrée dans laquelle il est impossible d'étiqueter tous les éléments

4.5 Forme finale uniformisée

En résumé, nous générons une forme flexible et uniforme des ensembles de données dans laquelle les entrées prennent la forme détaillée à la figure 4.15. La figure 4.16 présente un exemple d'entrée construite selon cette forme.

Le tableau 4.4 détaille le contenu de chaque champ d'une entrée standard. Nous avons conçu ce format afin qu'il puisse facilement être modifié pour accepter de nouvelles informations et utilisé pour uniformiser le plus de formats de jeux de données possible.

Tableau 4.4 Description des composantes d'une entrée

Nom	Description
__id	L'identifiant unique d'une entrée
template id	L'identifiant unique du patron à partir duquel une entrée est générée
question	La question corrigée en langue naturelle
interm question	La question générée à partir du patron (optionnel)
uri question only resources	La question dans laquelle seulement les ressources sont étiquetées
uri question no resources	La question dans laquelle tous les éléments du schéma sont étiquetés, sauf les ressources
uri question all	La question dans laquelle tous les éléments du schéma et les ressources sont étiquetés
interm sparql	La requête SPARQL encodée en version intermédiaire
uri interm sparql only resources	La requête SPARQL intermédiaire dans laquelle seulement les resources sont étiquetées
uri question no resources	La requête SPARQL dans laquelle tous les éléments du schéma sont étiquetés
uri interm sparql all	La requête SPARQL intermédiaire dans laquelle tous les éléments de la base de connaissances sont étiquetés
pure sparql	La requête SPARQL exécutable
set	L'ensemble dans lequel l'entrée s'inscrit, soit *train* (entraînement), *valid* (validation) ou *test*
original__data	Un dictionnaire contenant l'entrée tirée telle quelle de l'ensemble de données avant quelconque correction (optionnel)
dbpedia__result	Le résultat attendu de la requête sur DBpedia (optionnel)

```
{
    _id: str,
    template_id: Union[int, str],
    question: {
        question: str,
        interm_question: str,
        uri_question_only_resources: str,
        uri_question_no_resources: str,
        uri_question_all: str
    },
    query: {
        interm_sparql: str,
        uri_interm_sparql_only_resources: str,
        uri_interm_sparql_no_resources: str,
        uri_interm_sparql_all: str,
        pure_sparql: str
    },
    set: Str["train", "valid", "test”]
    original_data: dict,
    dbpedia_result: dict
}
```

Figure 4.15 Format standard d'une entrée de nos ensembles de données générés

4.6 Ensemble de données hors du vocabulaire

Comme mentionné plus haut, l'une des principales limitations des modèles de type *Seq2Seq* est leur incapacité à traiter les jetons hors des vocabulaires de base construits pendant l'entraînement. Afin de nous assurer de tester toutes les facettes des modèles, nous générons donc un ensemble de test supplémentaire pour chaque ensemble de données, qu'on appelle l'ensemble de test hors vocabulaire (HV).

Génération. Tout d'abord, pour chaque jeu de données, on extrait la liste des éléments de la base de connaissances référencés dans l'ensemble d'entraînement. Puis, on extrait de la base de connaissances une liste d'éléments et de triplets qui ne sont pas dans la liste des éléments déjà utilisés. Finalement, on utilise les patrons pour générer 250 requêtes, en remplissant les paramètres substituables par des éléments de la base de connaissances qui ne sont pas présents dans l'ensemble d'entraînement. La table 4.5 présente des statistiques sur les ensembles de données hors vocabulaire générés. Nous générons également des versions étiquetées des jeux de données HV.

Considérations particulières. Il est important de noter que plusieurs patrons de Monument comprennent par défaut des éléments de la base de connaissances. Par exemple, le patron suivant comprend la propriété `dct:subject`.

```
{
  _id: "4121",
  template_id: 2,
  question: {
    question: "what organisation regulates and controls the new sanno hotel ?",
    interm_question: "what is the tenant of new sanno hotel ?",
    uri_question_only_resources: "what is the tenant of dbr:New_Sanno_Hotel ?",
    uri_question_rest_no_resources: "what is the dbo:tenant of new sanno hotel ?",
    uri_question_all: "what is the dbo:tenant of dbr:New_Sanno_Hotel ?"
  },
  query: {
    interm_sparql: "select distinct var_uri where brack_open
                     dbr_New_Sanno_Hotel dbo_tenant var_uri
                   brack_close",
    uri_interm_sparql_only_resources: "select distinct var_uri where brack_open
                                         dbr:New_Sanno_Hotel dbo_tenant var_uri
                                       brack_close",
    uri_interm_sparql_rest_no_resources: "select distinct var_uri where brack_open
                                            dbr_New_Sanno_Hotel dbo:tenant var_uri
                                          brack_close",
    uri_interm_sparql_all: "select distinct var_uri where brack_open
                              dbr:New_Sanno_Hotel dbo:tenant var_uri
                            brack_close",
    pure_sparql: "select distinct ?uri where { dbr:New_Sanno_Hotel dbo:tenant ?uri }"
  },
  set: "train",
  original_data: { … },
  dbpedia_result: { … }
}
```


Figure 4.16 Exemple d'entrée standard générée selon le format proposé

Tableau 4.5 Survol des ensembles de données hors vocabulaire (HV)

	Mon HV	**Questions intermédiaires HV**
Taille	250	250
Plus longue question	14	28
Plus longue requête	31	20
Elems BC	263	502
Taux d'intersection	0.04	0.00

```
SELECT ?a WHERE { <resource> dct:subject ?a . }
```

Par conséquent, afin de pouvoir tester la capacité des modèles à gérer les jetons hors du vocabulaire tout en s'assurant qu'ils comprennent les formulations utilisées, nous laissons dans le jeu de données HV les éléments de la base de connaissances qui sont présents dans les patrons. Cependant, cela signifie que le taux d'intersection entre les ensembles d'entraînement et de test HV des versions de Monument n'est pas de 0.

Faux positifs. Lors de la génération des jeux de données HV, il est important de s'assurer que nous n'avons pas d'entrées retournant des réponses vides. En effet, si un modèle n'utilise pas les bons éléments de la base de connaissances dans la requête générée, il y a beaucoup de chances que la réponse à cette requête soit vide. Or, si la réponse attendue est également une réponse vide, cela crée une situation de *faux positif*, puisque le modèle obtient la réponse attendue sans réussir la tâche de génération de requête. La figure 4.17 montre un exemple d'une telle situation.

```
{
  "entrée": {
      "question": "how many fire eaters are there in titanic (movie) ?"
  },
  "sortie": {
      "attendue":{
          "requête": "SELECT DISTINCT COUNT(?uri) WHERE {
                          dbr:Titanic_(movie) dbo:fireEater ?uri
                      }",
          "résultat": 0,
      },
      "générée":{
          "requête": "SELECT DISTINCT COUNT(?uri) WHERE {
                          dbr:Florida_State_Road_540 dbo:soccerPlayer ?uri
                      }",
          "résultat": 0
      }
   }
}
```

Figure 4.17 Un bon résultat obtenu avec les mauvais éléments de la base de connaissances

Afin de limiter le nombre de faux positifs dans nos ensembles de données HV, nous ajoutons une condition qui vérifie que les requêtes ne retournent pas de réponses vides. Cela ne nous

permet pas d'éviter complètement les faux positifs, mais nous permet de prioriser les patrons qui retournent des réponses non vides.

4.7 Vocabulaires

La construction des vocabulaires est une étape importante de l'entraînement des modèles. La présente section décrit notre construction des vocabulaires pour chaque ensemble de données, en suivant la méthodologie décrite à la section 3.3.1.

4.7.1 Segmentation des jetons

Chaque vocabulaire est un ensemble de jetons que le modèle apprend à comprendre et à utiliser. Tel que défini plus haut, un jeton peut être un mot (exemple : quel, qui, année), un préfixe (exemple : anti-, pré-), un symbole de ponctuation (!, ?, {), etc. Dans notre architecture, nous utilisons simplement les espaces comme démarcations entre les jetons, et nous considérons les symboles de ponctuation (c.-à-d., le point d'interrogation, le point final et la virgule) comme des jetons uniques, sauf s'ils font partie d'URIs. Par exemple, la question *What is the currency of Kerguelen Islands ?* est composée des jetons suivants : `[what, is, the, currency, of, kerguelen, islands, ?]`. La requête qui lui est associée est composée des jetons suivants : `[select, distinct, var_uri, where, brack_open, dbr_Kerguelen_Islands, dbo_currency, var_uri, brack_close]`.

Afin d'éviter les problèmes liés à la casse, tout ce qui n'est pas un URI est mis en minuscule, alors que les URIs gardent leur casse originale.

4.7.2 Identification des URIs

Un URI est un lien unique pointant vers un élément de la base de connaissances et qui commence par un préfixe. La table 4.3 fait le détail des différents préfixes et de leur signification. Dans les versions originales des jeux de données, les URIs peuvent être composés de plusieurs jetons (e.g., `[dbr_Primus, attr_open, band, attr_close]`). Dans nos versions étiquetées, ils sont toujours composés d'un seul jeton (e.g., `dbr_Primus(band)`).

4.7.3 Symboles de base

Tous les vocabulaires contiennent des symboles de base nous permettant de mieux représenter les entrées. Ils sont détaillés à la table 4.6. Il s'agit majoritairement de symboles permettant de délimiter le début et la fin des questions et des requêtes. Cependant, il est important de

comprendre le rôle qu'occupe le symbole `<unk>`, représentant un jeton inconnu, dans le cadre de la traduction automatique.

Tableau 4.6 Symboles communs aux vocabulaires

Symbole	Indice	Description
<unk>	0	Utilisé pour représenter un jeton inconnu (*unknown*), hors du vocabulaire
<pad>	1	Utilisé pour faire du remplissage (*padding*) afin que toutes les phrases aient le même nombre de symboles
<sos>	2	Utilisé pour indiquer le début d'une phrase (*start of sentence*)
<eos>	3	Utilisé pour indiquer la fin d'une phrase (*end of sentence*)

De manière générale, même si les modèles de type *Seq2Seq* apprennent à comprendre des jetons d'un vocabulaire source fixe afin de générer une traduction en utilisant un vocabulaire cible fixe, on veut tout de même qu'ils génèrent une traduction d'une entrée dont ils ne connaissent pas tous les jetons utilisés. Il nous faut donc être capables d'encoder les jetons jamais vus dans l'entrée, afin que le modèle comprenne que la source contient un élément dont il ne connaît pas la signification. Le symbole `<unk>` encode donc la signification d'un jeton inconnu, indiquant au modèle qu'il doit utiliser les autres jetons déjà connus afin de comprendre une entrée source, sans pour autant le priver de savoir que la source contient un jeton inconnu.

4.7.4 Architectures sans copie

Pour résumer la section 3.3.1, dans les architectures sans copie, le vocabulaire source contient tous les jetons se trouvant dans les questions en langue naturelle, et le vocabulaire cible contient tous les jetons se trouvant dans les requêtes. La table B.1, en annexe, présente le nombre de jetons dans chaque vocabulaire source et cible des jeux de données dans les architectures sans copie. Afin de mieux illustrer notre méthodologie, supposons un jeu de données composé seulement de l'entrée démontrée à la figure 4.18.

```
Question: ["is", "dbr:Ahu_Akivi", "a", "dbo:Place", "?"]
Requête : ["ask", "where", "brack_open", "dbr:Ahu_Akivi", "rdf_type",
          "dbo:Place","brack_close"]
```

Figure 4.18 Une entrée étiquetée

La figure 4.19 montre le contenu des vocabulaires après leur construction.

VOCAB SOURCE

0: <unk>
1: <pad>
2: <sos>
3: <eos>
4: is
5: dbr:Ahu_Akivi
6: a
7: dbo:Place
8: ?

VOCAB CIBLE

0: <unk>
1: <pad>
2: <sos>
3: <eos>
4: ask
5: where
6: brack_open
7: dbr:Ahu_Akivi
8: rdf_type
9: dbo:Place
10: brack_close

Figure 4.19 Vocabulaires complets sans copie

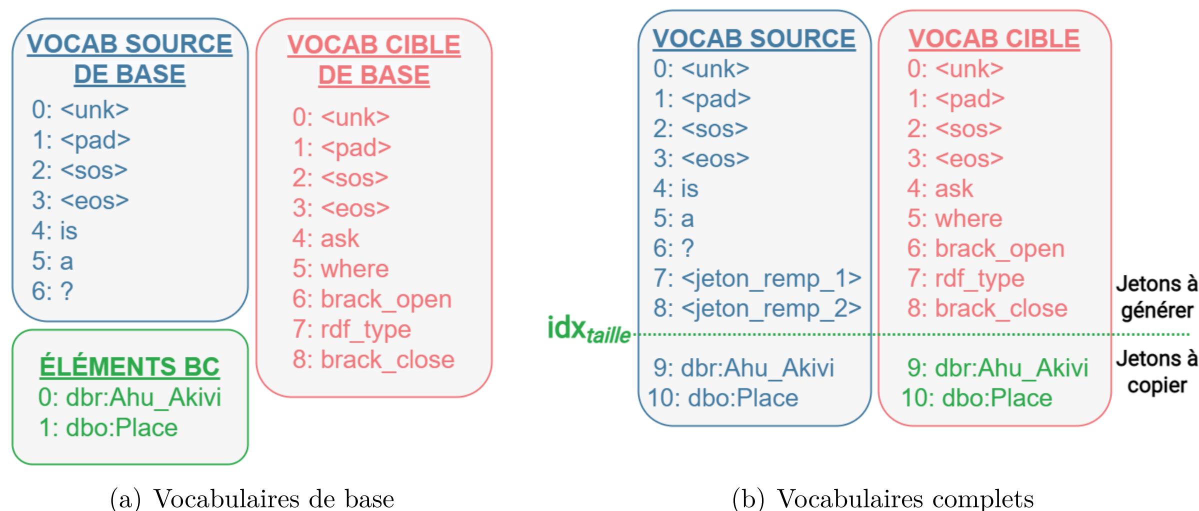
4.7.5 Architectures avec copie

En résumant encore une fois la section 3.3.1, pour les architectures avec copie, on utilise les vocabulaires de base, qui sont créés à partir de tous les jetons que le modèle devra apprendre à générer dans les questions (vocabulaire source) et dans les réponses (vocabulaire cible), et on s'assure que les vocabulaires source et cible ont la même taille en ajoutant au besoin des jetons de remplissage ($<jeton_remp_n>$).

Puis, on concatène à ces vocabulaires de base la liste des URIs des éléments de la base de connaissances que le modèle doit apprendre à copier afin d'obtenir nos vocabulaires complets. On doit s'assurer de garder en mémoire l'indice de coupure idx_{taille}, défini à la section 3.3.2, qui a comme valeur la taille des vocabulaires de base. Rappelons que, comme expliqué à la section 4.3.3, nous utilisons les préfixes et le symbole **:** (deux-points) afin de différencier les éléments de la base de connaissances que nous souhaitons générer de ceux que nous souhaitons copier.

Le tableau B.2 présente le nombre de jetons dans chaque vocabulaire source et cible de base, ainsi que le nombre de jetons dans l'extension de vocabulaire pour chaque jeu de données dans les architectures avec copie.

Reprenons en exemple le jeu de données composé seulement de l'entrée démontrée à la figure 4.18. La figure 4.20(a) montre le contenu des vocabulaires après la première étape de construction des vocabulaires initiaux, et la figure 4.20(b) montre le contenu des vocabulaires après la concaténation des jetons à copier.



(a) Vocabulaires de base (b) Vocabulaires complets

Figure 4.20 Construction des vocabulaires pour la copie

4.7.6 Utilisation des vocabulaires

Une fois les vocabulaires construits, on les utilise pour transformer nos entrées question-requêtes qui sont des chaînes de caractères en vecteurs d'entiers que nous pourrons passer à notre modèle.

Pour chaque entrée question-requête, nous commençons par segmenter la question et la requête en une liste de jetons, comme décrit à la section 4.7.1. Puis, nous utilisons le vocabulaire source pour indexer la question, et le vocabulaire cible pour indexer la requête. L'indexation est l'action de remplacer chaque jeton par l'indice lui correspondant dans le vocabulaire. Nous ajoutons également les jetons de début ({2: <sos>}) et de fin ({3: <eos>}) de phrase pour chaque question et chaque requête. Finalement, nous utilisons les jetons de remplissage ({1: <pad>}) pour remplir les questions afin qu'elles aient toutes la même taille que la question la plus longue du jeu de données. On répète le processus de remplissage pour les requêtes.

Afin d'illustrer le processus de transformation, supposons que la question la plus longue du jeu de données comprenne 8 jetons, que la requête la plus longue du jeu de données en compte 10. La figure 4.21 montre l'entrée utilisée en exemple jusqu'à présent (Figure 4.18) transformée en vecteur d'entiers pour une architecture sans copie (i.e, indexée avec le vocabulaire décrit à la figure 4.19). La figure 4.22 montre l'entrée utilisée en exemple jusqu'à présent (Figure 4.18) transformée en vecteur d'entiers pour une architecture avec copie (i.e, indexée avec le vocabulaire décrit à la figure 4.20(b)). C'est cette version de l'entrée qui est reçue par la couche de copie. Finalement, la figure 4.23 montre l'entrée masquée reçue par l'encodeur et le décodeur dans une architecture avec copie, dans laquelle tous les jetons ayant un indice

plus grand ou égal à $idx_{taille} = 9$ sont remplacés par le jeton inconnu (`{0: <unk>}`).

```
Question: [2, 4, 5, 6, 7, 8, 3, 1, 1, 1]
Requête : [2, 4, 5, 6, 7, 8, 9, 10, 3, 1, 1, 1]
```

Figure 4.21 Une entrée préparée pour une architecture sans copie

```
Question: [2, 4, 9, 5, 10, 6, 3, 1, 1, 1]
Requête : [2, 4, 5, 6, 9, 7, 10, 8, 3, 1, 1, 1]
```

Figure 4.22 Une entrée préparée pour une architecture avec copie

```
Question: [2, 4, 0, 5, 0, 6, 3, 1, 1, 1]
Requête : [2, 4, 5, 6, 0, 7, 0, 8, 3, 1, 1, 1]
```

Figure 4.23 Une entrée préparée pour une architecture avec copie dans laquelle les éléments hors des vocabulaires de base sont masqués

4.8 Métriques

La présente section décrit les métriques utilisées afin d'évaluer nos architectures.

4.8.1 Score BLEU

Le score BLEU [46] est l'une des métriques les plus utilisées pour évaluer des systèmes de traduction automatique. Elle fonctionne en comparant les n-grammes d'une phrase générée automatiquement à ceux d'une phrase de référence. Les n-grammes sont comparés sans prendre en compte l'ordre des jetons. Il s'agit d'une excellente métrique, puisqu'elle permet une certaine différence entre les deux phrases comparées. Or, s'il est acceptable pour une traduction d'une langue humaine à une autre d'utiliser des synonymes ou de traduire avec des mots un peu différents, la traduction vers SPARQL n'est pas aussi flexible. En effet, une phrase générée avec un seul mot de différence avec la phrase de référence aura un haut score BLEU, même si le mot de différence est l'élément de la base de connaissances référencé, ce qui est un gros problème dans le cadre de la traduction vers SPARQL. Par exemple, pour la question *Quelles sont les chansons écrites par John Mayer ?*, la requête attendue est la suivante.

```
SELECT ?s WHERE { ?s dbp:writtenBy dbr:John_Mayer . }
```

Or, si notre modèle a vu des exemples similaires sur John Legend, mais aucun sur John Mayer, il pourrait générer la traduction suivante :

```
SELECT ?s WHERE { ?s dbp:writtenBy dbr:John_Legend . }
```

Dans cet exemple, la traduction n'est pas bonne puisque l'élément de la base de connaissances référencé (*dbr :John_Legend*) est incorrect. Le score BLEU de cette traduction est de 66.06%, ce qui est assez haut considérant que la réponse retournée par cette requête est complètement incorrecte. Cet exemple illustre pourquoi nous avons besoin de métriques plus complètes, car même si le score BLEU nous donne une bonne indication de la syntaxe, il ne nous donne aucune indication de la capacité du modèle à fournir les bonnes requêtes et les bonnes réponses.

4.8.2 Précision des réponses

Puisque nous envisageons notre système de traduction comme un système de questions-réponses, il est important d'inclure une manière de calculer la précision des réponses retournées par les requêtes. Pour ce faire, nous utilisons la métrique de *précision des réponses*, qui

est simplement le pourcentage d'exactitude des réponses retournées par les requêtes prédites par rapport aux réponses attendues retournées par les requêtes de référence dans notre ensemble de test. Par conséquent, si on a un score BLEU de 66.06%, mais une précision de 0%, on sait que le modèle n'est probablement pas capable de choisir les bons éléments de la base de connaissances dans les requêtes générées.

CHAPITRE 5 ÉVALUATION ET DISCUSSION

Ce chapitre présente en détail les résultats obtenus durant nos expérimentations, et discute des forces et faiblesses des modèles, de leurs limitations, et des prochaines étapes dans le domaine de la traduction machine vers SPARQL.

5.1 Résultats des expérimentations

Cette section présente les résultats obtenus à travers nos expérimentations. Nous commençons par tenter de reproduire les résultats rapportés par la revue de littérature [5], dans lequel plusieurs modèles de l'état de l'art en traduction SPARQL sont entraînés en utilisant la bibliothèque d'apprentissage automatique FairSeq [6] sur un serveur CHP. Puis, nous évaluons l'impact de notre mécanisme de copie. Afin de favoriser la reproductibilité et l'accessibilité, nous utilisons nos propres implémentations des modèles entraînés sur un Graphics Processing Unit (GPU) Google Colab. Nous nous sommes concentrés sur les deux meilleures architectures, soit *Transformer* et le *CnnS2S*. Pour chaque tableau, la colonne *BLEU* est le score BLEU, qui représente la qualité des traductions. La colonne *Préc.* représente la qualité des réponses, qui est la précision du résultat des requêtes générées par rapport aux résultats des requêtes de référence.

Chaque résultat présenté est le résultat de la moyenne de trois exécutions différentes du même ensemble de données sur la même architecture. Cela donne plus de crédibilité à nos résultats et assure leur reproductibilité.

5.1.1 Reproduction des résultats de base

La table 5.1 présente les résultats des architectures sur les versions originales des jeux de données LC-QuAD, et la table 5.2 présente ces résultats sur les versions originales des jeux de données Monument. Rappelons que **TNTSPA** est la version de LC-QuAD fournie par la revue réalisée par [5], **Qsts Reformulées** est la version des données utilisant les questions reformulées de la version originale de LC-QuADv1.0, et **Qsts Intermédiaires** est la version des données utilisant les questions générées par des patrons de LC-QuADv1.0. Aussi, **Mon**, **Mon50** et **Mon80** sont trois versions différentes de Monument, générées aléatoirement en utilisant les patrons du jeu de données. La figure 5.1 illustre un exemple d'entrée des données originales.

De prime abord, on voit clairement que l'on réussit à reproduire les résultats de la revue

```
"original_data": {
  "question": "give me a count of countries whose leader name is elazibeth ii ?",
  "interm_sparql": "select distinct count attr_open var_uri attr_close where brack_open
                        var_uri dbp_leaderName dbr_Elizabeth_II.
                        var_uri rdf_type dbo_Place
                    brack_close"
}
```

Figure 5.1 Une entrée des données originales, avant nos corrections

[5] avec nos architectures implémentées à la main. Il est normal que nos résultats soient légèrement plus bas, puisque nous rapportons la moyenne de 3 exécutions, alors que la revue utilise le meilleur résultat obtenu. Aussi, à cause du caractère aléatoire de l'initialisation des poids, il peut y avoir un écart d'au plus 10 points de performance entre un bon et un mauvais entraînement. Il y a également une amélioration claire qui vient avec l'utilisation des questions dérivées des patrons, comme on peut le voir dans les performances entre les versions reformulées (**TNTSPA** et **Qsts Reformulées**) et **Qsts Intermédiaires**. Cela vient du fait que le vocabulaire pour la dernière version est un peu plus petit que pour les versions reformulées, et que les questions sont toutes formulées de la même manière.

Tableau 5.1 Résultats des architectures sans copie sur les données originales de LC-QuAD

	TNTSPA		**Qsts Reformulées**		**Qsts Intermédiaires**	
Architecture	BLEU	Préc.	BLEU	Préc.	BLEU	Préc.
Transformer	**55.98**	42.80	**53.69**	**41.20**	63.37	48.33
CnnS2S	52.24	**44.00**	51.30	40.00	**65.95**	**48.07**

On remarque aussi tout de suite les excellents résultats sur Monument (5.2), qui peuvent sembler difficiles à améliorer. La précision des réponses est très haute, alors qu'on aurait pu s'attendre à de moins bons résultats en sachant que le modèle n'est pas capable de gérer de nouveaux éléments de la BC. Or, cela s'explique par le haut taux d'intersection entre les éléments de la base de connaissances utilisés dans l'entraînement et ceux utilisés dans le test. Le modèle répond facilement à des questions sur des sujets qu'il a déjà appris.

Or l'importante différence entre les résultats sur Monument et ceux sur LC-QuAD nous laisse entrevoir le problème causé par la lourde tâche d'apprendre la signification de beaucoup d'éléments de la base de connaissances avec peu d'exemples. En effet, on sait que la différence entre Monument et LC-QuAD est que ce dernier contient une beaucoup plus grande variété

Tableau 5.2 Résultats des architectures sans copie sur les données originales de Monument

	Mon		**Mon50**		**Mon80**	
Architecture	BLEU	Préc.	BLEU	Préc.	BLEU	Préc.
Transformer	95.86	90.55	92.26	**91.72**	**96.35**	**92.69**
CnnS2S	**96.31**	**91.66**	**95.25**	88.34	94.47	82.68

d'éléments de la base de connaissances pour beaucoup moins d'exemples. Qui plus est, le taux d'intersection est plus bas pour LC-QuAD. Même pour la version **Qsts Intermédiaires**, dont les questions utilisent la formulation des patrons comme Monument, on observe une grosse différence de performances. On déduit donc que les architectures de base ont beaucoup de difficulté à traiter la grande variété d'éléments de la base de connaissances en plus de ne pas savoir gérer les éléments n'ayant pas été appris pendant l'entraînement.

Tableau 5.3 Résultats des architectures originales sur les données hors du vocabulaire (HV)

	Mon HV		**Qsts Intermédiaires HV**	
Architecture	BLEU	Préc.	BLEU	Préc.
Transformer	60.16	30.00	52.82	62.40
CnnS2S	**63.88**	**35.86**	**56.18**	**74.00**

C'est en regardant les performances sur les jeux de données hors du vocabulaire (composés seulement d'éléments de la base de connaissances jamais vus) présentées à la table 5.3 que l'on voit les réelles capacités des architectures. En effet, on observe une différence drastique entre les résultats presque parfaits sur Monument et ceux sur la version HV de Monument. La précision des réponses rapportée sur le jeu de données **Qsts Intermédiaires HV** semble très haute par rapport au score BLEU rapporté. Cela met déjà en lumière l'une des limitations de notre approche, qui est que nos jeux de données HV contiennent beaucoup de faux positifs, et ce malgré les précautions prises lors de leur génération. Les faux positifs sont des requêtes qui retournent la bonne réponse sans utiliser les bons éléments de la base de connaissances. Souvent, ce sont des réponses vides. En observant la combinaison du score BLEU et de la précision des réponses, on comprend que nos architectures génèrent des requêtes syntaxiquement correctes, mais qui n'utilisent pas les bons éléments de la base de connaissances.

5.1.2 Données corrigées

La table 5.4 présente les résultats des architectures sur les versions corrigées selon notre méthodologie des jeux de données LC-QuAD, et la table 5.5 présente ces résultats sur les versions corrigées des jeux de données Monument. La figure 5.2 illustre un exemple d'entrée corrigée.

```
{
  "question": "give me a count of countries whose leader name is elizabeth ii ?",
  "interm_sparql": "select distinct count par_open var_uri par_close where brack_open
                        var_uri dbp_leaderName dbr_Elizabeth_II sep_dot
                        var_uri rdf_type dbo_Place
                    brack_close"
}
```

Figure 5.2 Une entrée des données corrigées

Tableau 5.4 Résultats des architectures sans copie sur les données corrigées de LC-QuAD

	Qsts Reformulées		Qsts Intermédiaires	
Architecture	BLEU	Préc.	BLEU	Préc.
Transformer	**55.13**	**43.73**	63.72	46.26
CnnS2S	50.82	40.53	**64.93**	**48.47**

Les résultats sur les deux versions de LC-QuAD corrigées sont très similaires à ceux sur les données originales. Similairement, bien que les résultats sur Monument semblent augmenter minimalement, ils restent dans la marge de 5 points permise par l'initialisation aléatoire des poids. De ce fait, on ne peut affirmer avec certitude que l'utilisation d'une version corrigée permet de meilleurs résultats. Cependant, cette constance dans les résultats nous permet de déterminer que nos architectures sont tout de même résistantes aux petites erreurs et aux fautes de frappe, ce qui est un avantage en soi.

La table 5.6 présente les résultats des données hors du vocabulaire corrigées. Puisque nous n'avons pas généré ces données en y introduisant des erreurs, on obtient des résultats semblables à ceux sur les données originales, avec une petite amélioration de score BLEU qui peut être attribuée au fait que les jeux de données hors vocabulaires ont un format sensiblement plus proche des données corrigées que des données originales. Encore une fois, la haute précision des réponses sur LC-QuAD trahit la présence de requêtes retournant de faux positifs dans notre jeu de données hors vocabulaire.

Tableau 5.5 Résultats des architectures sans copie sur les données corrigées de Monument

	Mon		Mon50		Mon80	
Architecture	BLEU	Préc.	BLEU	Préc.	BLEU	Préc.
Transformer	96.46	91.81	**96.76**	92.31	**96.97**	**93.35**
CnnS2S	**97.06**	**93.66**	96.75	**92.42**	96.01	89.90

Tableau 5.6 Résultats des architectures sans copie sur les données hors du vocabulaire (HV) corrigées

	Mon HV		Qsts Intermédiaires HV	
Architecture	BLEU	Préc.	BLEU	Préc.
Transformer	66.59	22.40	54.67	67.13
CnnS2S	**67.87**	**24.53**	**56.23**	**78.80**

5.1.3 Données partiellement étiquetées - ressources

Jusqu'ici, les résultats présentés étaient sur des jeux de données non étiquetés. La table 5.7 présente les résultats des architectures sur les versions dans lesquelles seulement les ressources sont étiquetées des jeux de données LC-QuAD et Monument, et la table 5.8 présente ces résultats sur les jeux de données hors vocabulaires. Rappelons que nous avons uniquement généré des versions étiquetées pour les **Qsts Intermédiaires** de LC-QuAD, puisque notre méthodologie d'étiquetage est dépendante des patrons. Également, dans les versions étiquetées, les URIs représentant des éléments de la base de connaissances sont contenues dans un seul jeton, au contraire des versions de base dans lesquelles un URI peut être séparé en plusieurs jetons. La figure 5.3 illustre un exemple d'entrée des données partiellement étiquetées avec seulement les éléments de la base de connaissances étant des *ressources* (pour Monument, cela inclut aussi les *types*).

```
{
  "uri_question_only_resources": "give me a count of countries whose leader name is dbr:Elizabeth_II ?",
  "uri_interm_sparql_only_resources": "select distinct count par_open var_uri par_close where brack_open
                                        var_uri dbp_leaderName dbr:Elizabeth_II sep_dot
                                        var_uri rdf_type dbo_Place
                                    brack_close"
}
```

Figure 5.3 Une entrée partiellement étiquetée avec les ressources

Architectures sans copie. En ce qui concerne les architectures sans copie, on remarque tout de suite une amélioration par rapport aux données non étiquetées sur les **Qsts Intermédiaires** de LC-QuAD. Cela s'explique principalement par le fait que les URIs sont représentés par un seul jeton, et qu'une expression représentant une ressource dans la question est remplacée par un seul jeton. Cette modification diminue le vocabulaire source de plus de 1500 jetons, et le vocabulaire cible de plus de 200 jetons. De plus, au lieu de devoir apprendre à générer les différentes parties des URIs - ce qui peut introduire des erreurs - le modèle est assuré de toujours générer des URIs complets.

Pour Monument, cette amélioration est moins évidente, mais tout aussi claire, puisque les modèles atteignent systématiquement des scores BLEU au-dessus de 97%. Or, puisque ce jeu de données contient déjà beaucoup d'exemples par élément de la base de connaissances, le modèle n'avait pas autant de difficulté à associer les expressions de plusieurs jetons aux URIs de plusieurs jetons dans les données non étiquetées que pour LC-QuAD.

Cependant, on n'observe pas cette même amélioration sur les jeux de données hors du vocabulaire. En effet, on a encore une fois des performances similaires à celles observées sur les données non étiquetées. Le fait d'étiqueter nos jeux de données ne donne tout de même pas à nos architectures la capacité de comprendre les éléments jamais vus en entraînement. Cette limitation persiste dans les architectures sans copie, peu importe la manière dont on fait l'étiquetage. Par contre, les différentes manières d'étiqueter les jeux de données permettent toutes de réduire la taille des vocabulaires selon un certain degré, ce qui facilite la tâche du modèle. Par conséquent, on s'attend à des performances similaires pour tous les jeux de données étiquetés.

Tableau 5.7 Résultats des architectures sur les données dans lesquelles seules les ressources sont étiquetées

	Mon		**Mon50**		**Mon80**		**Qsts Intermédiaires**	
Données	BLEU	Préc.	BLEU	Préc.	BLEU	Préc.	BLEU	Préc.
Transf.	97.31	93.82	97.75	94.72	97.59	94.59	65.70	50.00
Transf.-copie	**100**	**100**	**100**	**100**	**100**	**100**	**85.21**	**77.40**
CnnS2S	97.82	95.31	98.35	96.62	97.93	95.56	68.50	49.07
CnnS2S-copie	**100**	**100**	**100**	**100**	**100**	**100**	**82.18**	**72.47**

Architectures avec copie. En ce qui concerne les architectures avec copie, on obtient des performances parfaites sur toutes les versions de Monument, et on observe sur les **Qsts Intermédiaires** de LC-QuAD une amélioration d'environ 20 points de score BLEU et de

précisions des réponses par rapport aux architectures sans copie. Or, en observant les tailles des vocabulaires avec et sans copie présentées en annexe dans les tables B.1 et B.2, cette amélioration n'est pas une surprise. En effet, la différence entre le nombre de jetons que le modèle doit apprendre à comprendre et à générer entre les architectures avec et sans copie est élevée. La difficulté dans les architectures avec copie n'est plus d'apprendre à générer les bons mots, puisque les vocabulaires source et cible sont beaucoup plus petits. Elles doivent plutôt apprendre à copier les éléments de la base de connaissances aux bons endroits dans les requêtes. Or, la précision des réponses parfaite sur Monument et très haute sur LC-QuAD nous permet de déterminer que la copie se fait avec succès.

En ce qui concerne les jeux de données hors du vocabulaire, les résultats obtenus sur les **Qsts Intermédiaires HV** contrastent avec ceux obtenus sur les **Qsts Intermédiaires**. En effet, on retrouve des performances similaires à celle sur les architectures sans copie au lieu de voir la même amélioration que dans la table 5.7. De plus, la précision des réponses est nettement plus basse que dans les données originales pour l'architecture **Transf.-copie**. Cela s'explique par plusieurs facteurs. Premièrement, même si les ressources sont étiquetées et qu'il est possible pour le modèle de les copier directement de la source, il reste tout de même un bon nombre d'éléments de la base de connaissances qui sont inconnus et que le modèle ne peut pas copier. Aussi, les expressions en langue naturelle désignant ces éléments inconnus sont composées de mots inconnus, ce qui fait que la question contient plusieurs jetons inconnus. Or, les modèles avec copie semblent avoir de la difficulté à comprendre les phrases contenant beaucoup de jetons inconnus, et vont parfois copier ces derniers dans la requête. La base de connaissances ne reconnaît pas le symbole **<unk>**, ce qui en fait des requêtes non exécutables.

Cela met en lumière un des avantages des modèles sans copie par rapport aux modèles avec copie. En effet, les modèles sans copie tentent de générer le mot le plus probable selon le contexte, ce qui fait qu'ils génèrent toujours des requêtes complètes (sans inconnus), même s'ils n'utilisent pas les bons éléments de la base de connaissances. Par conséquent, si la syntaxe est correctement générée, ils auront toujours une réponse, même si ce n'est pas la bonne. L'architecture **CnnS2s-copie** semble moins affectée par ce problème de copie de jetons inconnus.

Sur Monument, on obtient des performances parfaites. En effet, à cause de la manière dont le jeu de données Monument HV est construit (voir section 4.6), les éléments qui ne sont pas des ressources sont les mêmes que dans le jeu de données qui n'est pas HV. Il n'y a donc pas de jetons HV qui ne sont pas des ressources dans le jeu de données Monument HV partiellement étiqueté avec les ressources, contrairement à la version de LC-QuAD.

Tableau 5.8 Résultats des architectures sur les données hors du vocabulaire (HV) dans lesquelles seules les ressources sont étiquetées

	Mon HV		**Qsts Intermédiaires HV**	
Données	BLEU	Préc.	BLEU	Préc.
Transf.	65.68	17.87	55.80	**59.73**
Transf.-copie	**100**	**100**	**60.43**	34.80
CnnS2S	63.53	29.20	58.47	**84.00**
CnnS2S-copie	**100**	**100**	**60.04**	78.53

5.1.4 Données partiellement étiquetées - schéma

La table 5.9 présente les résultats des architectures sur les versions dans lesquelles tous les éléments sauf les ressources sont étiquetés dans les jeux de données LC-QuAD et Monument, et la table 5.10 présente ces résultats sur les jeux de données hors vocabulaires. La figure 5.4 illustre un exemple d'entrée des données partiellement étiquetées avec tous les éléments de la base de connaissances qui ne sont pas des *ressources* (ou des *types* pour Monument).

```
{
  "uri_question_rest_no_resources": "give me a count of dbo:Place whose dbp:leaderName is elizabeth ii ?",
  "uri_interm_sparql_rest_no_resources": "select distinct count par_open var_uri par_close where brack_open
                                          var_uri dbp:leaderName dbr_Elizabeth_II sep_dot
                                          var_uri rdf_type dbo:Place
                                      brack_close"
}
```

Figure 5.4 Une entrée partiellement étiquetée avec tous les éléments du schéma

Architectures sans copie. Pour Monument, on obtient sans surprise des performances similaires à celles sur les données partiellement étiquetées avec les ressources. Cela s'explique par le fait que les deux jeux de données ont des vocabulaires cibles de même taille. Même sur les données hors vocabulaires, on observe des performances très proches entre les deux jeux de données partiellement étiquetés. Encore une fois, on ne s'attend pas à des performances parfaites, puisque l'étiquetage ne donne pas au modèle la capacité de gérer des éléments inconnus. Cependant, on remarque que les résultats sur les architectures avec et sans copie sont très similaires, avec des performances sensiblement meilleures sur les architectures sans copie.

Tableau 5.9 Résultats des architectures sur les données dans lesquelles tous les éléments sauf les ressources sont étiquetés

	Mon		**Mon50**		**Mon80**		**Qsts Intermédiaires**	
Données	BLEU	Préc.	BLEU	Préc.	BLEU	Préc.	BLEU	Préc.
Transf.	**96.65**	**92.56**	**96.05**	90.64	**97.24**	**93.59**	**70.94**	**57.73**
Transf.-copie	95.10	89.96	95.95	**91.00**	97.18	93.46	68.38	52.33
CnnS2S	**97.02**	**93.37**	**96.82**	**92.78**	**97.29**	**93.66**	76.97	53.93
CnnS2S-copie	89.45	69.94	94.63	86.08	95.23	87.48	**79.38**	**59.27**

Architectures avec copie. Pour Monument, il n'y a que 11 jetons que le modèle doit apprendre à copier. Par conséquent, on s'attend aux mêmes performances que pour les modèles sans copie, puisqu'il y a presque autant d'éléments de la base de connaissances que le modèle doit apprendre à générer pour les deux types d'architectures. En ce qui concerne LC-QuAD, les tailles des vocabulaires sources et cibles sont très proches de celles des vocabulaires des architectures sans copie, puisqu'il n'y a pas beaucoup d'éléments qui ne sont pas des ressources en proportion à ceux qui le sont. Cela réduit de beaucoup l'utilité de la couche de copie.

En somme, cette manière d'étiqueter les jeux de données est beaucoup moins intéressante que les données étiquetées avec seulement les ressources, parce qu'il y a nettement plus de ressources référencées dans les données, ce qui permet à la couche de copie d'être utilisée à sa pleine capacité.

Dans les jeux de données, on observe encore une fois des performances de précision très basse par rapport aux résultats des architectures sans copie sur les données non étiquetées. La cause de cette basse précision est la même que pour les jeux de données hors vocabulaires partiellement étiquetés avec les ressources. La couche de copie semble avoir de la difficulté à gérer les questions contenant des jetons inconnus.

Tableau 5.10 Résultats des architectures sur les données hors du vocabulaire (HV) dans lesquelles tous les éléments sauf les ressources sont étiquetés

	Mon HV		**Qsts Intermédiaires HV**	
Données	BLEU	Préc.	BLEU	Préc.
Transf.	**65.85**	**18.53**	**56.79**	**58.53**
Transf.-copie	48.61	4.67	52.96	16.40
CnnS2S	**67.35**	**24.27**	60.58	**58.67**
CnnS2S-copie	36.73	13.60	**66.20**	31.07

5.1.5 Données complètement étiquetées

La table 5.11 présente les résultats des architectures sur les versions dans lesquelles tous les éléments de la base de connaissances sont étiquetés des jeux de données LC-QuAD et Monument, et la table 5.12 présente ces résultats sur les jeux de données hors vocabulaires. La figure 5.5 illustre un exemple d'entrée des données complètement étiquetées.

```
{
  "uri_question_all": "give me a count of dbo:Place whose dbp:leaderName is dbr:Elizabeth_II ?",
  "uri_interm_sparql_all": "select distinct count par_open var_uri par_close where brack_open
                             var_uri dbp:leaderName dbr:Elizabeth_II sep_dot
                             var_uri rdf_type dbo:Place
                         brack_close"
}
```

Figure 5.5 Une entrée complètement étiquetée

Architectures sans copie De toutes les versions étiquetées, c'est la version complètement étiquetée qui a le plus petit vocabulaire source (les vocabulaires cibles ont tous la même taille). On obtient d'excellents résultats sur les versions de Monument et sur les questions intermédiaires de LC-QuAD complètement étiquetées dans les architectures sans copie. Ces derniers sont similaires aux résultats obtenus sur les architectures sans copie avec les autres versions étiquetées, comme attendu.

Par contre, un rapide coup d'œil aux résultats des architectures sans copie sur les jeux de données hors vocabulaires, qui sont similaires à ceux de toutes nos architectures sans copie, nous indique que les modèles sont encore et toujours limités par leur incapacité à gérer des éléments inconnus.

Tableau 5.11 Résultats des architectures sur les données complètement étiquetées

	Mon		**Mon50**		**Mon80**		**Qsts Intermédiaires**	
Données	BLEU	Préc.	BLEU	Préc.	BLEU	Préc.	BLEU	Préc.
Transf.	97.02	92.81	97.41	94.41	97.80	94.86	70.29	51.93
Transf.-copie	**100**	**100**	**100**	**100**	**100**	**100**	**98.38**	**97.60**
CnnS2S	97.82	95.26	97.71	95.13	98.14	95.96	76.62	52.93
CnnS2S-copie	**100**	**100**	**100**	**100**	**100**	**100**	**98.35**	**97.40**

Architectures avec copie Sur les architectures avec copie, on obient des performances parfaites sur toutes les versions de Monument, et des performances presque parfaites sur LC-QuAD. Cela représente une grande réussite, considérant le nombre et la variété des éléments de la base de connaissances utilisés dans LC-QuAD. Les hautes performances de score BLEU nous montrent que, utilisé à sa pleine capacité, le modèle est capable de copier les éléments de la question vers la requête, et les hautes performances de précision des réponses nous montrent que le modèle est capable de copier les éléments dans le bon ordre.

Qui plus est, les résultats sur les données hors vocabulaire sont tout aussi encourageants. En effet, on a encore une fois des performances parfaites sur Monument, et les résultats sur LC-QuAD sont très hauts par rapport aux résultats des architectures sans copie.

Tableau 5.12 Résultats des architectures sur les données hors du vocabulaire (HV) complètement étiquetées

	Mon HV		**Qsts Intermédiaires HV**	
Données	BLEU	Préc.	BLEU	Préc.
Transf.	65.55	25.33	56.75	54.50
Transf.-copie	**100**	**100**	**85.68**	**71.07**
CnnS2S	48.31	25.60	60.98	37.87
CnnS2S-copie	**100**	**100**	**90.16**	**81.07**

5.2 Discussion

À la lumière de ces résultats, il est clair que l'utilisation d'un mécanisme d'étiquetage couplé à un mécanisme de copie représente une avenue prometteuse en traduction de la langue naturelle vers SPARQL. En effet, on observe une amélioration d'environ 30 points de score BLEU et d'environ 50 points de précision des réponses entre les architectures utilisant des

données non étiquetées sans copie et les architectures utilisant des données étiquetées avec une couche de copie sur le jeu de données **Qsts Intermédiaires** de LC-QuAD. Sur Monument, on réussit à améliorer les résultats déjà très hauts et à obtenir des performances parfaites.

En outre, la différence drastique entre les résultats rapportés sur les versions originales de Monument (tableau 5.2) et les performances obtenues sur la version hors vocabulaire des données (tableau 5.3) nous démontre l'importance d'avoir des métriques représentatives. En effet, on observe une baisse de score BLEU de 30 points ainsi qu'une baisse de précision des réponses de 60 points entre les deux versions des données. Le score BLEU est un excellent indicateur de la qualité d'une traduction, et sa flexibilité en fait une métrique très utile en traduction automatique. Cependant, il n'est pas capable d'évaluer de manière complète les capacités d'un modèle de question-réponse. C'est pourquoi on a besoin d'outils d'évaluation supplémentaires, tels la précision des réponses ou les jeux de données hors vocabulaire. Sans ces outils, les limites des architectures de l'état de l'art passent facilement inaperçues, et même les modèles avec les plus hautes performances ne sont pas adaptés pour une utilisation réelle.

Or, la couche de copie nous amène un peu plus proche d'une architecture adaptée à une utilisation réelle. En effet, comme démontré par le tableau 5.11, les architectures augmentées avec une couche de copie performent aussi bien sur des petits jeux de données que sur des plus gros. Cela représente un réel avantage, puisque les données du monde réel se font plutôt rares dans le domaine de la traduction vers SPARQL, et que les données de qualité (formulation des questions variées et naturelles, requêtes complexes) sont difficiles à générer. Qui plus est, les architectures utilisant la copie ont tendance à converger plus vite que celles qui ne l'utilisent pas, comme illustré par les graphes présentés dans l'annexe C. Cela représente une amélioration, puisque notre objectif est que cette recherche permette éventuellement une utilisation plus répandue de la traduction SPARQL. Or, un modèle qui se réentraîne rapidement est plus facilement utilisable dans des situations réelles.

D'ailleurs, bien que les performances entre les données originales (tableaux 5.1 et 5.2) et les données corrigées (tableau 5.4 et 5.5) soient similaires, la valeur des jeux de données corrigés et uniformisés vient de leur utilité pour la recherche future. En effet, notre but est que ces données soient plus facilement exploitables pour les développements futurs dans le domaine, par exemple pour la conception d'un mécanisme d'étiquetage plus complet ou l'entraînement de modèles plus portables.

C'est en ayant ce but en tête que nous avons réimplémenté notre propre version des modèles sur un *notebook* Google Colab, au lieu d'utiliser les outils rendus disponibles par la librairie FairSeq [6]. Ce choix a été motivé par la prise de conscience que la reproductibilité représente

un défi dans le domaine de la traduction vers SPARQL, et qu'il n'existe pas de point de départ accessible et central pour les personnes qui souhaitent y contribuer. Ainsi, l'entraînement de chaque modèle prend au plus une heure et demie avec les ressources de Google Colab. Tout le code [3] et les ensembles de données [4] sont disponibles sur GitHub, ainsi que beaucoup de documentation sur nos expérimentations.

À cause de la nécessité d'avoir accès à des patrons et de la limite de temps, nous n'avons pas pu tester sur plus de jeux de données dans le cadre de cette maîtrise. Cependant, des résultats préliminaires obtenus sur des sous-ensembles de l'ensemble de données DBNQA [41] sont très prometteurs. DBNQA contient 894,499 entrées générées à partir de patrons extraits de LC-QuADv1.0 [36] et de Question Answering over Linked Data v7 (QALD-7) [47], qui est un autre jeu de données de traduction SPARQL.

En résumé, grâce à nos contributions, les modèles utilisant la copie n'ont plus besoin d'apprendre les correspondances entre chaque URI et l'expression en langue naturelle lui correspondant. Ils n'ont également plus besoin d'autant d'exemples pour apprendre les formulations des questions, puisque les jeux de données utilisés dans le cadre de cette recherche n'utilisent pas beaucoup de patrons de questions. Notre travail met également en évidence que les modèles de l'état de l'art, qui sont présentés comme ayant des performances presque parfaites, ne sont en réalité pas aussi efficaces en dehors de l'ensemble de test sur lequel ils sont évalués.

3. https ://github.com/rooose/SPARQL_NMT_models
4. https ://github.com/rooose/SPARQL_NMT_data

CHAPITRE 6 CONCLUSION

6.1 Synthèse des travaux

Les principaux buts de cette recherche étaient de déterminer l'impact de l'ajout d'un mécanisme de copie aux architectures de type *Seq2Seq* les plus performantes dans le domaine de la traduction de la langue naturelle vers SPARQL. Nous avons principalement voulu déterminer si la copie était une solution viable au problème de gestion des jetons hors du vocabulaire rencontré par les architectures *Seq2Seq*, et si ce mécanisme permettait d'obtenir des requêtes plus précises. De plus, nous nous sommes questionnées sur l'efficacité des métriques actuellement utilisées en traduction automatique et avons démontré l'importance d'utiliser des métriques plus complètes.

Tout d'abord, l'étude des différents modèles de l'état de l'art en traduction automatique nous permet d'identifier une limitation importante des modèles lorsqu'ils sont utilisés comme des systèmes de question-réponse, comme c'est le cas pour la traduction de questions en langue naturelle vers des requêtes SPARQL. En effet, un tel modèle sera incapable de répondre à une question sur un sujet jamais vu en entraînement, même si la réponse se trouve dans la base de connaissances. Cela est dû au fait que les modèles de type *Seq2Seq* utilisent des vocabulaires fixes de mots qu'ils comprennent et qu'ils peuvent générer, qui ne peut être modifié après l'entraînement. Nous remarquons également que les métriques actuelles ne rendent pas adéquatement compte de cette importante limitation. Cela nous permet de cibler la copie de jetons de la question vers la requête comme une solution possible à ce problème, ainsi que de définir des méthodes d'évaluation supplémentaires. Notre intuition est que, si le modèle peut apprendre à générer des requêtes SPARQL syntaxiquement correctes, un mécanisme de copie lui permettrait de copier directement les jetons ayant rapport au sujet et les insérer au bon endroit dans la requête, sans avoir à comprendre leur signification.

Pour confirmer nos hypothèses, nous ajoutons une couche de copie aux deux meilleures architectures de traduction automatique vers SPARQL, soit le *Transformer* et le *CnnS2S*. Afin que les modèles aient des jetons à copier, il est également nécessaire d'étiqueter les jeux de données afin d'inclure les URIs directement dans les questions. Nous générons trois versions des jeux de données étiquetées différemment afin d'évaluer l'impact des différents types d'éléments de la base de connaissances sur la copie. Au passage, nous générons également une version corrigée et uniformisée des différents jeux de données, puisqu'il y a plusieurs divergences entre les différentes manières dont les jeux de données sont formatés.

Afin de bien évaluer l'impact de nos modifications, nous définissons comme métrique supplémentaire la *précision des réponses*, qui calcule la précision des réponses retournées par nos requêtes générées automatiquement par rapport aux réponses attendues retournées par les requêtes de référence. De plus, afin d'évaluer les performances des modèles sur des sujets en dehors de ceux vus en entraînement, nous générons des jeux de données hors du vocabulaire. Ces derniers sont générés à partir des patrons associés à chaque ensemble de données, dans lesquels on insère des éléments de la base de connaissances inutilisés dans les ensembles originaux.

Les résultats obtenus démontrent sans équivoque que l'ajout d'une couche de copie aux modèles de traduction automatique vers SPARQL est un avantage. En effet, nous obtenons des performances parfaites sur les différentes versions du jeu de données Monument, et des performances presque parfaites sur les questions intermédiaires de LC-QuAD (score BLEU et précision des réponses au-dessus de 97%). Nous remarquons également une amélioration dans les performances des jeux de données hors vocabulaires.

6.2 Limitations de la solution proposée

Cette section identifie les principales limitations qui subsistent dans notre recherche, et pose quelques pistes de solutions afin d'y remédier.

6.2.1 Algorithme d'étiquetage

La principale limite de notre approche réside dans l'étape d'étiquetage des questions, qui est nécessaire au bon fonctionnement de notre architecture. En effet, les jeux de données présentement disponibles ne contiennent pas cette information, et il est encore plus difficile d'étiqueter une question en langue naturelle dont la portée n'est pas définie qu'une question formulée suivant un patron dans un jeu de donné sur un domaine ou un sous-domaine particulier.

D'ailleurs, afin de limiter la portée de cette recherche, nous utilisons les patrons à notre disposition comme point central de l'algorithme d'étiquetage. Or, cela nous limite à utiliser seulement des jeux de données générés par des patrons. Cette limitation est la raison pour laquelle nous n'évaluons pas nos architectures sur les questions reformulées de LC-QuAD, même si ces dernières sont plus complexes, variées, et proches des formulations du monde réel.

Par conséquent, notre architecture est simplement inutilisable dans un cas réel, puisqu'on ne peut pas s'attendre à ce que tous les utilisateurs posent leurs questions en suivant des pa-

trons. Nous n'avons pas un algorithme suffisamment flexible pour étiqueter correctement une question posée suivant une formulation qui n'est pas comprise dans les patrons disponibles.

Bien sûr, au cours de notre recherche, nous avons évalué plusieurs avenues afin d'implémenter un algorithme plus flexible et portable, mais les solutions que nous avons trouvées créaient plus de bruit que de remplacements corrects. Après nos expérimentations, nous pensons qu'il s'agit d'une tâche plus adaptée à une architecture neuronale. Cependant, afin que notre méthodologie soit portable et utilisable à d'autres fins que la recherche, un algorithme d'étiquetage plus flexible et portable sera absolument nécessaire.

6.2.2 Données hors du vocabulaire

Une autre limitation de notre méthodologie se trouve dans notre manière d'évaluer la capacité des modèles à répondre à des questions sur des sujets inconnus. Même si notre approche est suffisante pour tirer des conclusions solides, il est encore possible de faire mieux. Comme démontré par les résultats sur les jeux de données hors du vocabulaire que nous avons générés, certains modèles produisent de très hauts résultats de précision des réponses sur les jeux de données hors vocabulaire. Or, cela est souvent causé par de faux positifs plutôt que par de bonnes réponses.

Cela nous indique que notre manière de générer les jeux de données hors vocabulaire n'est pas tout à fait au point. Idéalement, on voudrait que chaque entrée de l'ensemble HV retourne une réponse unique et complexe qui élimine la possibilité d'obtenir la bonne réponse avec les mauvais éléments de la base de connaissances. On rencontre principalement le problème des faux positifs pour la version hors vocabulaire des données de LC-QuAD. En effet, puisque cet ensemble couvre énormément d'éléments, il est difficile de trouver des sujets qui ne sont pas couverts, et sur lesquels la base de connaissances contient assez d'informations pour retourner autre chose qu'une réponse vide. C'est d'ailleurs pour cela que nos jeux de données hors vocabulaire sont assez petits. Cette méthodologie est par conséquent difficilement utilisable sur des jeux de données plus larges et variés tels que DBNQA [41].

Afin de pallier ce problème, il nous faudra tout d'abord réviser notre méthodologie de génération des jeux de données hors du vocabulaire. De plus, il sera intéressant de développer une métrique qui évalue seulement l'ordre et la présence des éléments de la base de connaissances référencés dans une requête, afin d'avoir une vue d'ensemble des différents aspects de la génération de requêtes SPARQL.

6.3 Améliorations futures

Cette recherche est particulièrement intéressante, puisqu'elle permet de surmonter une grande limitation des modèles de type *Seq2Seq* en génération de requêtes SPARQL à partir de questions en langue naturelle. De ce fait, elle ouvre la voie à une panoplie de différentes améliorations. La présente section propose quelques pistes de développement.

6.3.1 Portabilité du modèle sur plusieurs bases de connaissances ou langues

L'impact de l'ajout d'une couche de copie est que le nombre de jetons et de formulations que les modèles doivent apprendre est grandement réduit. Qui plus est, le modèle n'a plus besoin d'apprendre les noms exacts de chaque URI, puisqu'il peut simplement le copier de la question. Or, on peut se demander si cela lui permettrait de comprendre des requêtes plus variées et complexes, qui seraient potentiellement exécutables sur plusieurs bases de connaissances.

Par exemple, bien que nous ne l'ayons pas utilisé, le jeu de données LC-QuADv2.0 [45] fournit pour chaque question la requête correspondante pour DBpedia, ainsi que celle pour WikiData. Il serait donc très intéressant d'évaluer si nos architectures avec copie sont capables de copier les bons éléments aux bons endroits dans la requête, selon la base de connaissances qu'elle interroge. De plus, les récentes avancées dans les réseaux MMNMT (décrits à la section 2.3) pourraient être une excellente base pour développer un modèle fonctionnel sur plusieurs bases de connaissances, auquel on pourrait intégrer notre mécanisme de copie.

Similairement, une avenue intéressante serait de déterminer si un modèle entraîné à répondre à des questions en plusieurs langues (par exemple, en français et en anglais) atteindrait des performances similaires à nos architectures unilingues. En effet, l'utilisation d'une couche de copie permet à l'encodeur et au décodeur de se concentrer sur l'apprentissage des différentes formulations. Une autre façon d'exploiter cet aspect serait d'entraîner notre modèle sur un ensemble de données qui n'est pas généré à partir de patrons. Notre hypothèse est que nos architectures peuvent apprendre beaucoup plus de formulations, mais qu'elles sont actuellement limitées par les données disponibles.

6.3.2 Portabilité du mécanisme de copie sur d'autres architectures

La couche de copie développée dans cette recherche utilise les valeurs de l'attention afin de calculer la probabilité qu'un mot de la source soit généré. De ce fait, notre hypothèse est que la couche de copie est portable à virtuellement n'importe quel modèle de type *Seq2Seq* qui calcule les valeurs de l'attention entre la source et la cible. Il serait donc très intéressant

de tester notre méthodologie sur les modèles préentraînés de traduction automatique les plus récents, soit T5 [26], BART [48] et GPT-3 [49]. On pourrait également l'évaluer sur les modèles capables de générer du code tel Codex [50], puisque la syntaxe SPARQL est beaucoup plus proche de celle d'un langage de programmation que d'un langage naturel.

Dans le même ordre d'idées, il serait intéressant d'intégrer notre architecture à SGPT [34], afin de tirer avantage des informations supplémentaires qui sont encodées dans chaque jeton. Notre hypothèse est que cette intégration solidifierait la compréhension que le modèle a du schéma grâce à une intégration plus exhaustive, similaire à la méthodologie employée par SGPT.

6.3.3 Algorithme d'étiquetage neuronal

Finalement, bien que ce soit une limitation importante, le développement d'un algorithme d'étiquetage neuronal représente une opportunité en or de contribuer au domaine de la traduction SPARQL. En effet, notre recherche prouve qu'il est possible pour les modèles de gérer des éléments inconnus, ce qui enlève la principale limitation imposée par l'utilisation des modèles *Seq2Seq*. Par conséquent, le "seul obstacle" restant à la création d'une architecture adaptée au monde réel est un mécanisme d'étiquetage portable et flexible. Au vu des récentes avancées dans le domaine de l'annotation automatique, il est permis d'être optimiste sur la faisabilité d'un tel mécanisme [51, 52]. Cela sera définitivement le point central de notre future recherche.

RÉFÉRENCES

[1] "A reintroduction to our Knowledge Graph and knowledge panels," The Keyword, mai 2020. [En ligne]. Disponible : https://blog.google/products/search/about-knowledge-graph-and-knowledge-panels/

[2] "W3C MISSION," W3C, 2021. [En ligne]. Disponible : https://www.w3.org/Consortium/mission

[3] D. Beckett et T. Berners-Lee, "Turtle – terse rdf triple language," 01 2008. [En ligne]. Disponible : https://www.w3.org/TeamSubmission/turtle/

[4] "Resources," DBpedia Association, 2022. [En ligne]. Disponible : https://www.dbpedia.org/resources/

[5] X. Yin, D. Gromann et S. Rudolph, "Neural machine translating from natural language to sparql," 2019. [En ligne]. Disponible : https://arxiv.org/abs/1906.09302

[6] M. Ott *et al.*, "fairseq : A fast, extensible toolkit for sequence modeling," *CoRR*, vol. abs/1904.01038, 2019. [En ligne]. Disponible : http://arxiv.org/abs/1904.01038

[7] T. Soru *et al.*, "Sparql as a foreign language," 2017. [En ligne]. Disponible : https://arxiv.org/abs/1708.07624

[8] R. Cocco, M. Atzori et C. Zaniolo, "Machine learning of sparql templates for question answering over linkedspending," dans *2019 IEEE 28th International Conference on Enabling Technologies : Infrastructure for Collaborative Enterprises (WETICE)*, 2019, p. 156–161.

[9] S. Shekarpour *et al.*, "Generating sparql queries using templates," vol. 11, 01 2013.

[10] D. Diefenbach *et al.*, "Towards a question answering system over the semantic web," *CoRR*, vol. abs/1803.00832, 2018. [En ligne]. Disponible : http://arxiv.org/abs/1803.00832

[11] E. Zimina *et al.*, "Gqa : Grammatical question answering for rdf data," dans *Semantic Web Challenges*, D. Buscaldi, A. Gangemi et D. Reforgiato Recupero, édit. Cham : Springer International Publishing, 2018, p. 82–97.

[12] ——, "Mug-qa : Multilingual grammatical question answering for rdf data," dans *2018 IEEE International Conference on Progress in Informatics and Computing (PIC)*, 2018, p. 57–61.

[13] S. Liang *et al.*, "Querying knowledge graphs in natural language," 09 2020.

[14] P. Mendes *et al.*, “Dbpedia spotlight : Shedding light on the web of documents,” 09 2011, p. 1–8.

[15] P. Ferragina et U. Scaiella, “Tagme : On-the-fly annotation of short text fragments (by wikipedia entities),” vol. abs/1006.3498, 01 2010, p. 1625–1628.

[16] K. S. Tai, R. Socher et C. D. Manning, “Improved semantic representations from tree-structured long short-term memory networks,” 2015. [En ligne]. Disponible : https://arxiv.org/abs/1503.00075

[17] L. Jiang et R. Usbeck, “Knowledge graph question answering datasets and their generalizability : Are they enough for future research ?” 2022. [En ligne]. Disponible : https://arxiv.org/abs/2205.06573

[18] D. Vollmers *et al.*, “Knowledge graph question answering using graph-pattern isomorphism,” *CoRR*, vol. abs/2103.06752, 2021. [En ligne]. Disponible : https://arxiv.org/abs/2103.06752

[19] Y. Chen *et al.*, “Outlining and filling : Hierarchical query graph generation for answering complex questions over knowledge graph,” 2021. [En ligne]. Disponible : https://arxiv.org/abs/2111.00732

[20] I. Sutskever, O. Vinyals et Q. V. Le, “Sequence to sequence learning with neural networks,” *CoRR*, vol. abs/1409.3215, 2014. [En ligne]. Disponible : http://arxiv.org/abs/1409.3215

[21] M.-T. Luong, H. Pham et C. D. Manning, “Effective approaches to attention-based neural machine translation,” 2015. [En ligne]. Disponible : https://arxiv.org/abs/1508.04025

[22] A. Vaswani *et al.*, “Attention is all you need,” 2017. [En ligne]. Disponible : https://arxiv.org/abs/1706.03762

[23] J. Gehring *et al.*, “Convolutional sequence to sequence learning,” 2017. [En ligne]. Disponible : https://arxiv.org/abs/1705.03122

[24] N. Arivazhagan *et al.*, “Massively multilingual neural machine translation in the wild : Findings and challenges,” 2019. [En ligne]. Disponible : https://arxiv.org/abs/1907.05019

[25] A. Bapna, N. Arivazhagan et O. Firat, “Simple, scalable adaptation for neural machine translation,” 2019. [En ligne]. Disponible : https://arxiv.org/abs/1909.08478

[26] C. Raffel *et al.*, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *CoRR*, vol. abs/1910.10683, 2019. [En ligne]. Disponible : http://arxiv.org/abs/1910.10683

[27] T. Guo et H. Gao, “Content enhanced bert-based text-to-sql generation,” 2019. [En ligne]. Disponible : https://arxiv.org/abs/1910.07179

[28] V. Zhong, C. Xiong et R. Socher, “Seq2sql : Generating structured queries from natural language using reinforcement learning,” 2017. [En ligne]. Disponible : https://arxiv.org/abs/1709.00103

[29] Z. Zhang *et al.*, “Ernie : Enhanced language representation with informative entities,” 2019. [En ligne]. Disponible : https://arxiv.org/abs/1905.07129

[30] Y. Sun *et al.*, “Ernie 2.0 : A continual pre-training framework for language understanding,” 2019. [En ligne]. Disponible : https://arxiv.org/abs/1907.12412

[31] J. Devlin *et al.*, “BERT : pre-training of deep bidirectional transformers for language understanding,” *CoRR*, vol. abs/1810.04805, 2018. [En ligne]. Disponible : http://arxiv.org/abs/1810.04805

[32] Y. Cao *et al.*, “Bridge text and knowledge by learning multi-prototype entity mention embedding,” dans *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1 : Long Papers)*. Vancouver, Canada : Association for Computational Linguistics, juill. 2017, p. 1623–1633. [En ligne]. Disponible : https://aclanthology.org/P17-1149

[33] M. E. Peters *et al.*, “Knowledge enhanced contextual word representations,” 2019. [En ligne]. Disponible : https://arxiv.org/abs/1909.04164

[34] M. R. A. H. Rony *et al.*, “Sgpt : A generative approach for sparql query generation from natural language questions,” *IEEE Access*, vol. 10, p. 70 712–70 723, 2022.

[35] A. Radford *et al.*, “Language models are unsupervised multitask learners,” 2019.

[36] P. Trivedi *et al.*, “Lc-quad : A corpus for complex question answering over knowledge graphs,” 10 2017.

[37] W. Zhao *et al.*, “Improving grammatical error correction via pre-training a copy-augmented architecture with unlabeled data,” 2019. [En ligne]. Disponible : https://arxiv.org/abs/1903.00138

[38] A. See, P. J. Liu et C. D. Manning, “Get to the point : Summarization with pointer-generator networks,” 2017. [En ligne]. Disponible : https://arxiv.org/abs/1704.04368

[39] J. Gu *et al.*, “Incorporating copying mechanism in sequence-to-sequence learning,” 2016. [En ligne]. Disponible : https://arxiv.org/abs/1603.06393

[40] O. Vinyals, M. Fortunato et N. Jaitly, “Pointer networks,” 2015. [En ligne]. Disponible : https://arxiv.org/abs/1506.03134

[41] A.-K. Hartmann, T. Soru et E. Marx, "Generating a large dataset for neural question answering over the dbpedia knowledge base," 04 2018.

[42] Y. N. Dauphin *et al.*, "Language modeling with gated convolutional networks," *CoRR*, vol. abs/1612.08083, 2016. [En ligne]. Disponible : http://arxiv.org/abs/1612.08083

[43] N. Ng *et al.*, "Facebook fair's wmt19 news translation task submission," 2019. [En ligne]. Disponible : https://arxiv.org/abs/1907.06616

[44] "Smart Data Analytics," 2022. [En ligne]. Disponible : https://sda.tech/

[45] M. Dubey *et al.*, "Lc-quad 2.0 : A large dataset for complex question answering over wikidata and dbpedia," dans *Proceedings of the 18th International Semantic Web Conference (ISWC)*. Springer, 2019.

[46] K. Papineni *et al.*, "Bleu : A method for automatic evaluation of machine translation," dans *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL '02. USA : Association for Computational Linguistics, 2002, p. 311–318. [En ligne]. Disponible : https://doi.org/10.3115/1073083.1073135

[47] "Question Answering over Linked Data (QALD-7) – ESWC 2017," Hobbit, 2017. [En ligne]. Disponible : https://project-hobbit.eu/challenges/qald2017/

[48] M. Lewis *et al.*, "BART : denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *CoRR*, vol. abs/1910.13461, 2019. [En ligne]. Disponible : http://arxiv.org/abs/1910.13461

[49] T. B. Brown *et al.*, "Language models are few-shot learners," *CoRR*, vol. abs/2005.14165, 2020. [En ligne]. Disponible : https://arxiv.org/abs/2005.14165

[50] M. Chen *et al.*, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [En ligne]. Disponible : https://arxiv.org/abs/2107.03374

[51] M. Yani, A. A. Krisnadhi et I. Budi, "A better entity detection of question for knowledge graph question answering through extracting position-based patterns," *Journal of Big Data*, vol. 9, n°. 1, p. 80, Jun 2022. [En ligne]. Disponible : https://doi.org/10.1186/s40537-022-00631-1

[52] M. Bakhshi *et al.*, "Data-driven construction of sparql queries by approximate question graph alignment in question answering over knowledge graphs," *Expert Systems with Applications*, vol. 146, p. 113205, 2020. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/S0957417420300312

ANNEXE A CORRESPONDANCES EN SPARQL INTERMÉDIAIRE

Tableau A.1 Correpondances entre les jetons en SPARQL exécutable et intermédiaire

Exécutable	**Intermédiaire**	**Précision**
http ://dbpedia.org/ontology/ dbo :	dbo__	
http ://dbpedia.org/property/ dbp :	dbp__	
http ://dbpedia.org/category/ dbc :	dbc__	
http ://dbpedia.org/resource/ dbr :	dbr__	
dct :	dct__	
geo :	geo__	
georss :	georss__	
rdf :	rdf__	
rdfs :	rdfs__	
foaf :	foaf__	
owl :	owl__	
yago :	yago__	
skos :	skos__	
( )	par__open par__close	juste dans la syntaxe (FILTER, COUNT, ...)
( )	attr__open attr__close	juste dans les URIs d'éléments de la BC
{ }	brack__open brack__close	
.	sep__dot	juste dans la syntaxe
'	sparql__quote	notre contribution
?	var__	
*	wildcard	
<=	math__leq	
>=	math__geq	
<	math__lt	
>	math__gt	
ORDER BY ASC(?uri)	__oba__ var__uri	
ORDER BY DESC(?uri)	__obd__ var__uri	

ANNEXE B TAILLES DES VOCABULAIRES

Tableau B.1 Tailles des vocabulaires des architectures sans copie

	Original		**Corrigé**		**Étiqueté ressources**		**Étiqueté tout sauf ressources**		**Étiqueté complet**	
	Source	Cible	Source	Cible	Source	Cible	Source	Cible	Source	Cible
Mon	2413	1913	2416	1914	1826	1808	2408	1808	1812	1808
Mon50	2393	1918	2396	1919	1831	1813	2388	1813	1817	1813
Mon80	2402	1922	2405	1923	1839	1821	2397	1821	1825	1821
Qsts Interm.	6059	4420	5718	4420	4260	4167	6014	4229	4413	4229
Qsts Reformulées	6649	4420	6556	4420	-	-	-	-	-	-
TNTSPA	6409	4416	-	-	-	-	-	-	-	-

Tableau B.2 Tailles des vocabulaires des architectures avec copie

	Étiqueté ressources			**Étiqueté tout sauf ressources**			**Étiqueté complet**		
	Source	Cible	HV	Source	Cible	HV	Source	Cible	HV
Mon	60	42	1909	2565	1940	11	35	31	1920
Mon50	60	42	1909	2565	1940	11	35	31	1920
Mon80	60	42	1909	2565	1940	11	35	31	1920
Qsts Interm.	910	806	3962	5954	4063	774	297	101	4736

ANNEXE C COURBES D'ENTRAÎNEMENT

Données originales

Monument

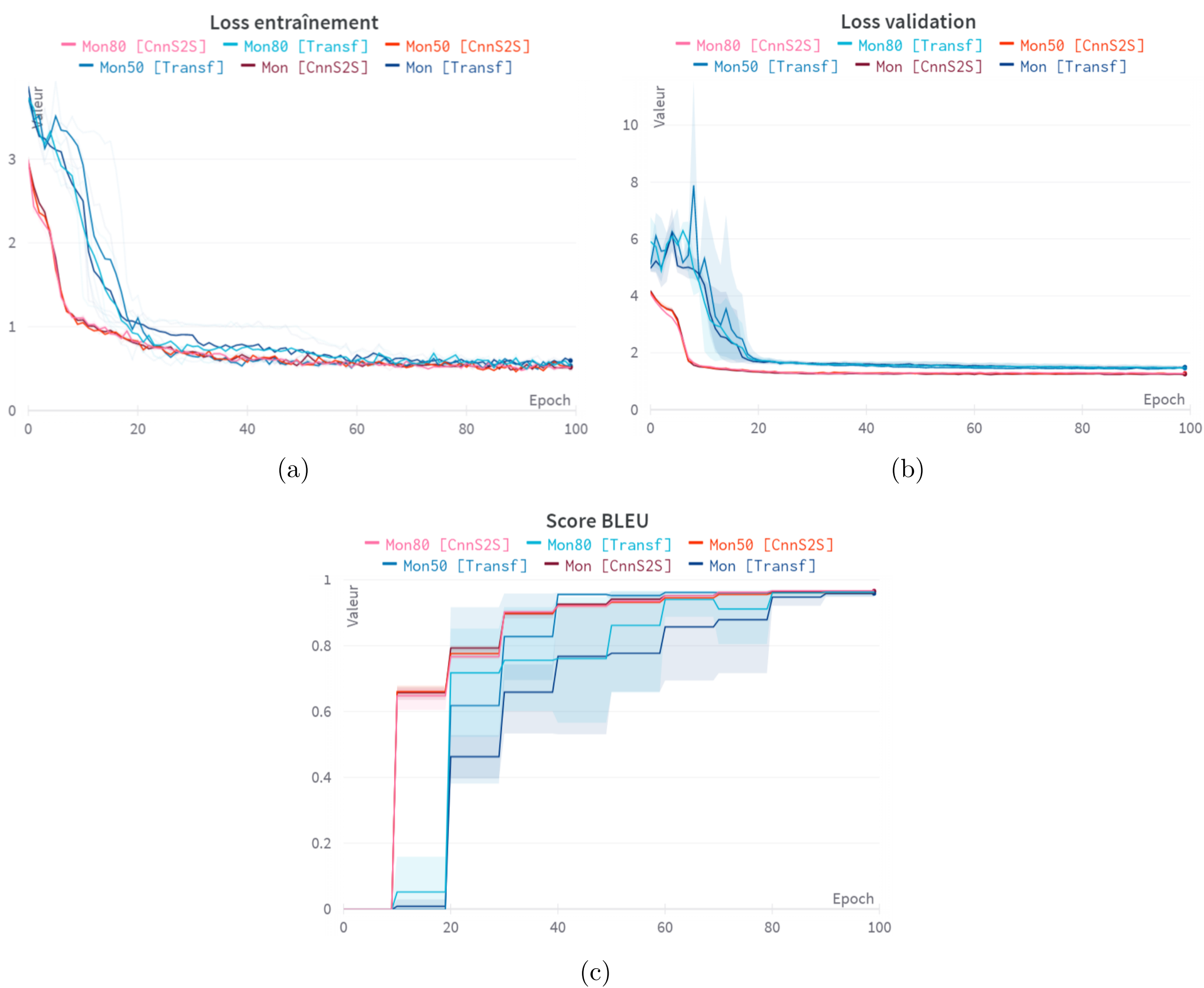


Figure C.1 Résultats les données originales de Monument

LC-QuAD

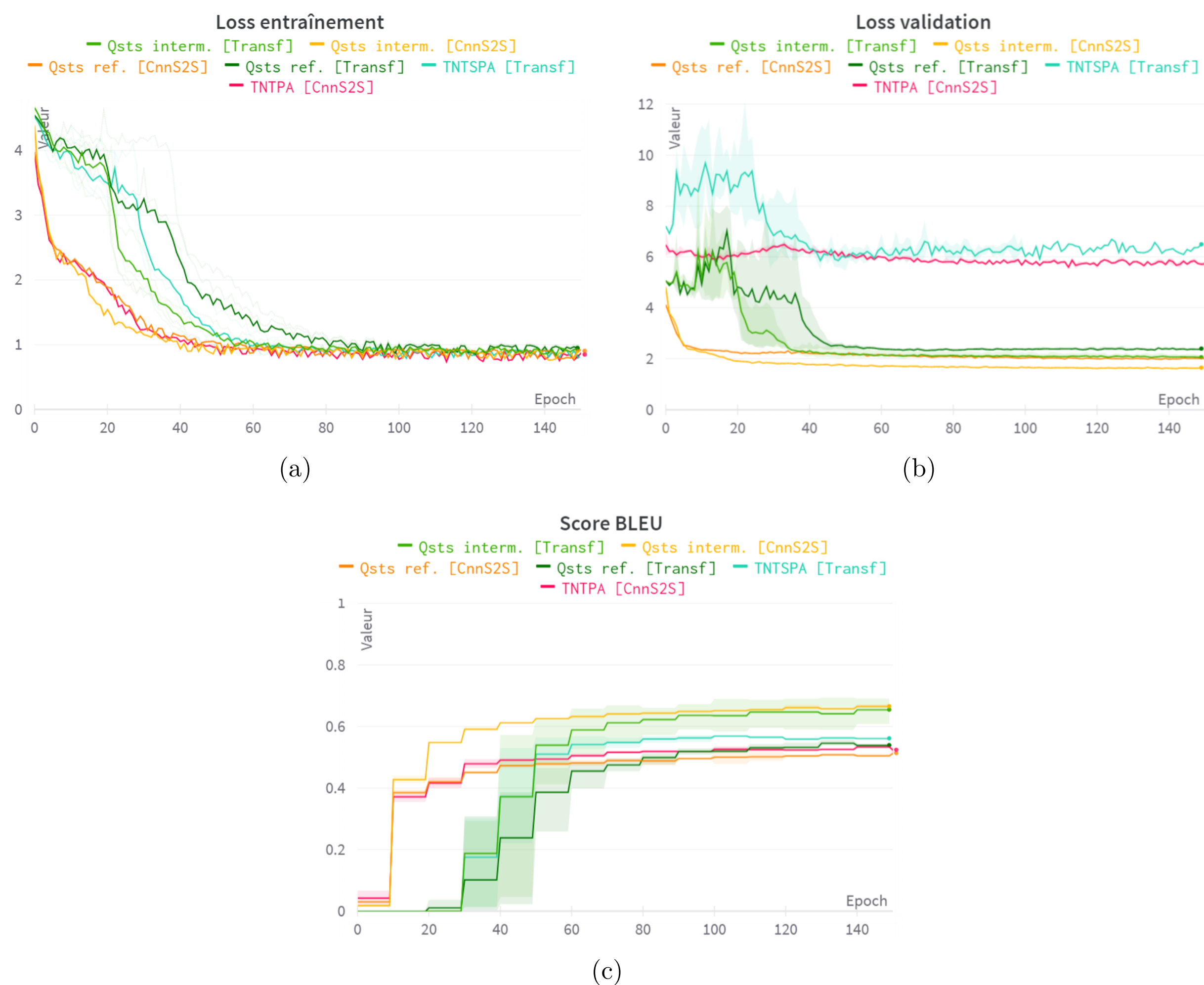


Figure C.2 Résultats les données originales de LC-QuAD

Données corrigées

Monument

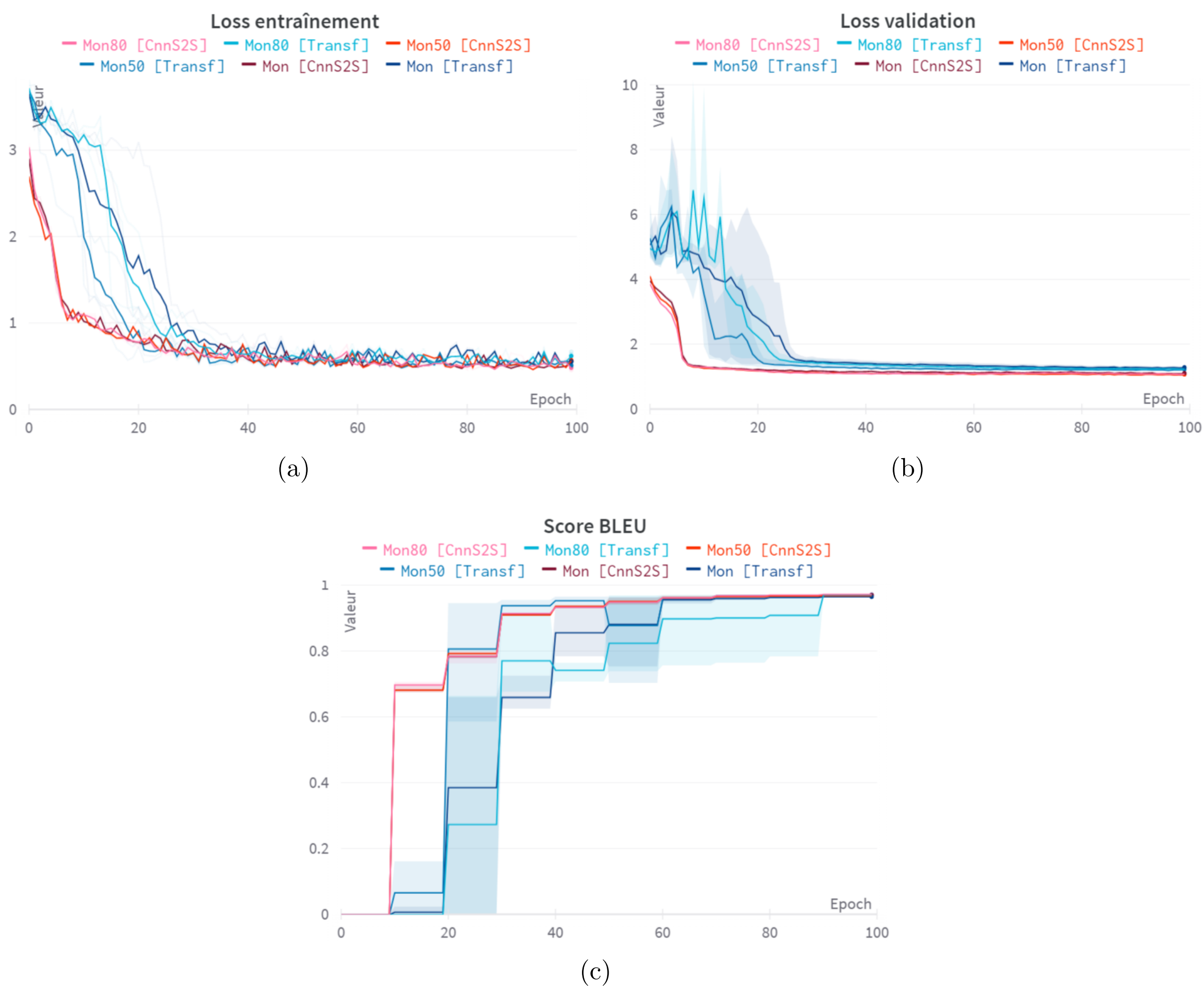


Figure C.3 Résultats les données corrigées de Monument

LC-QuAD

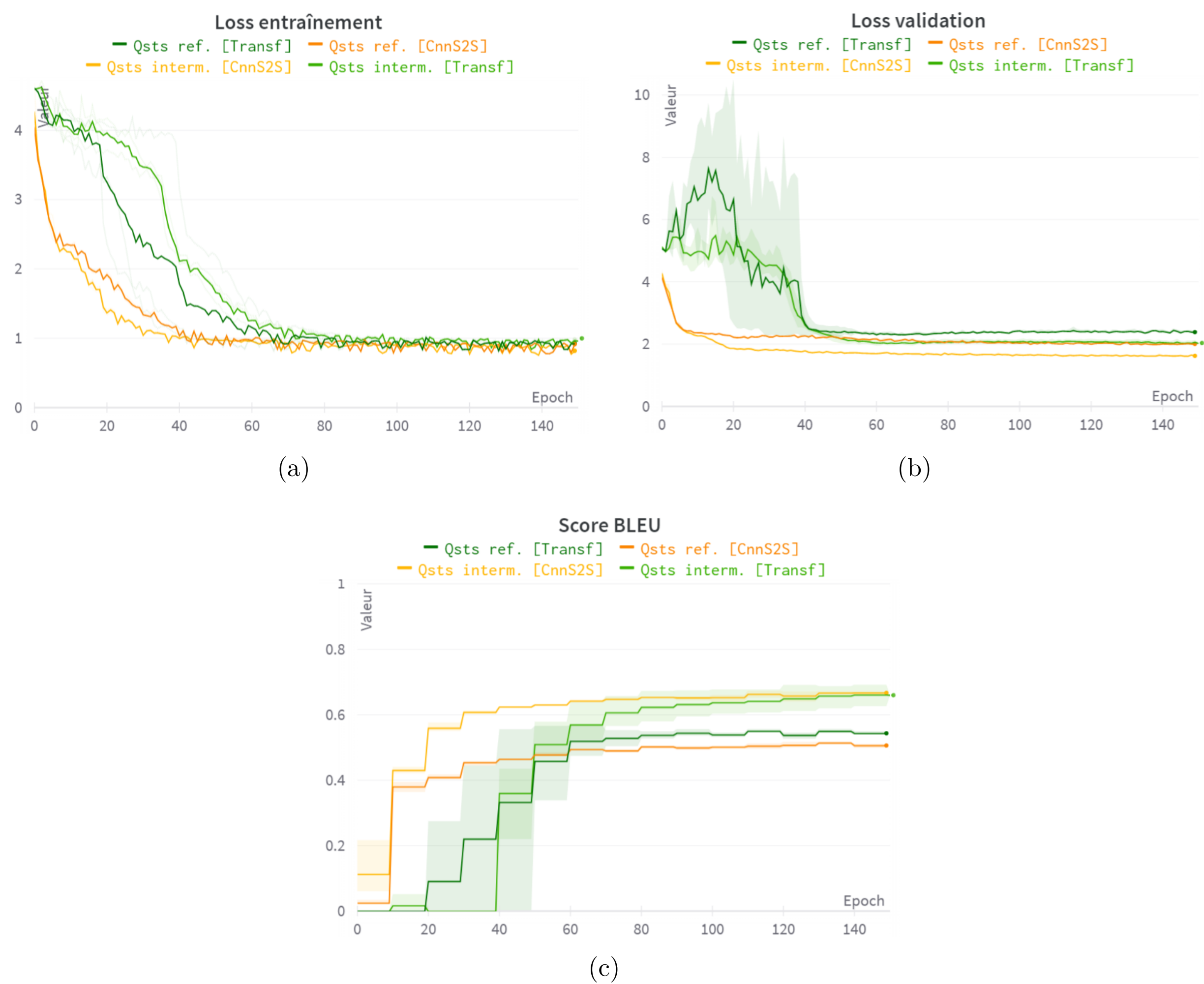


Figure C.4 Résultats les données corrigées de LC-QuAD

Données étiquetées

Versions étiquetées partiellement - ressources

Monument

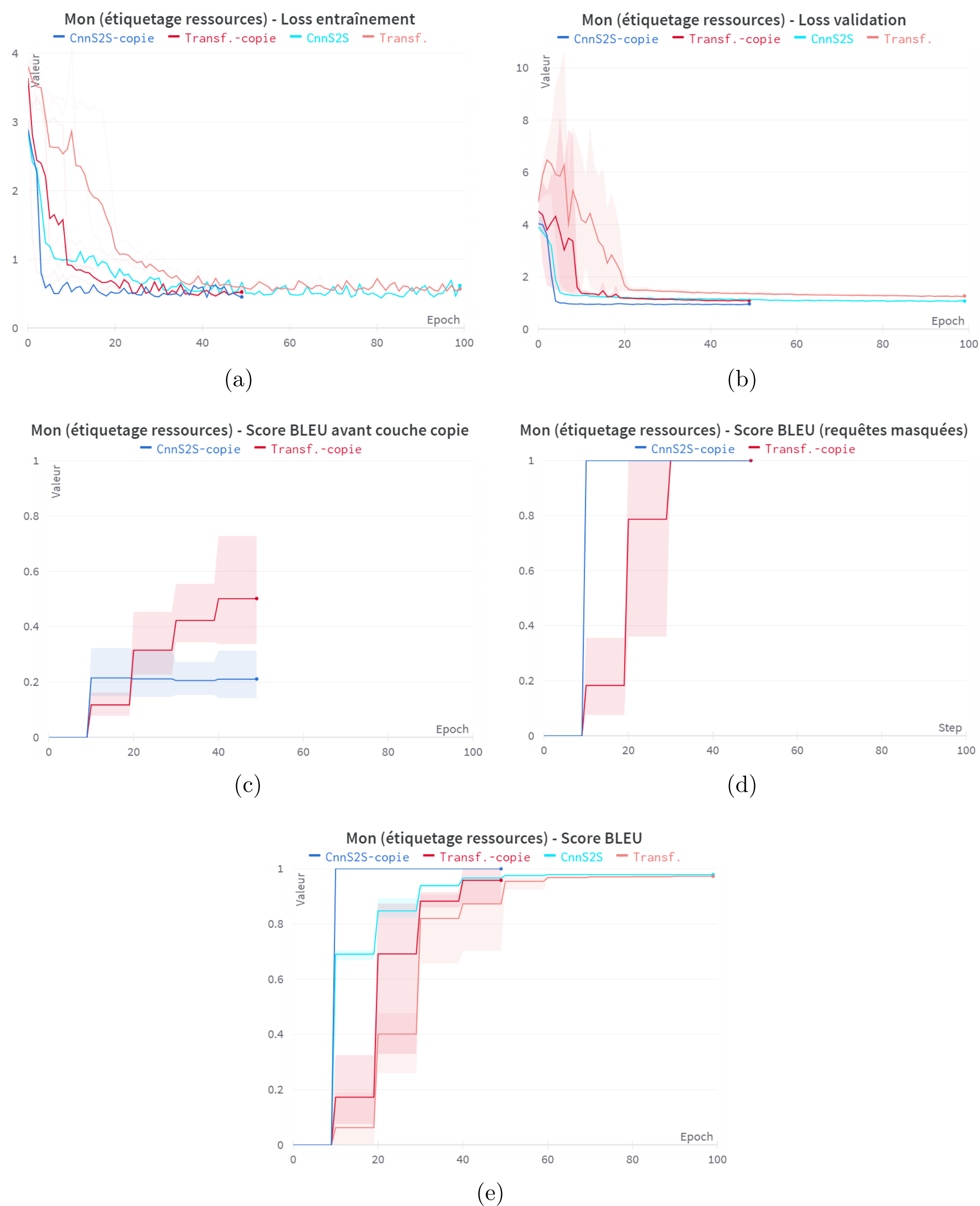


Figure C.5 Résultats sur Monument avec seules les ressources étiquetées

Monument50

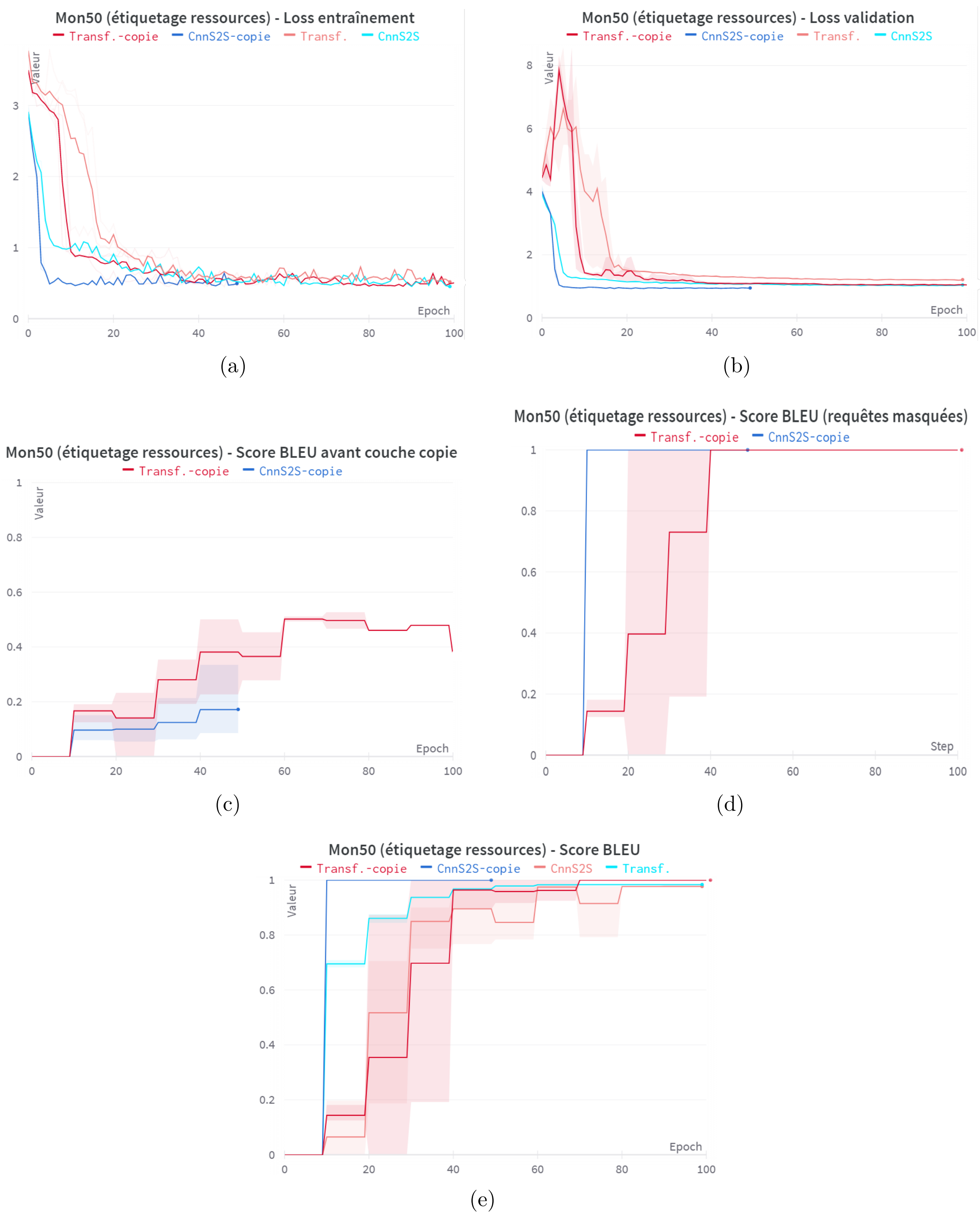


Figure C.6 Résultats sur Monument 50 avec seules les ressources étiquetées

Monument80

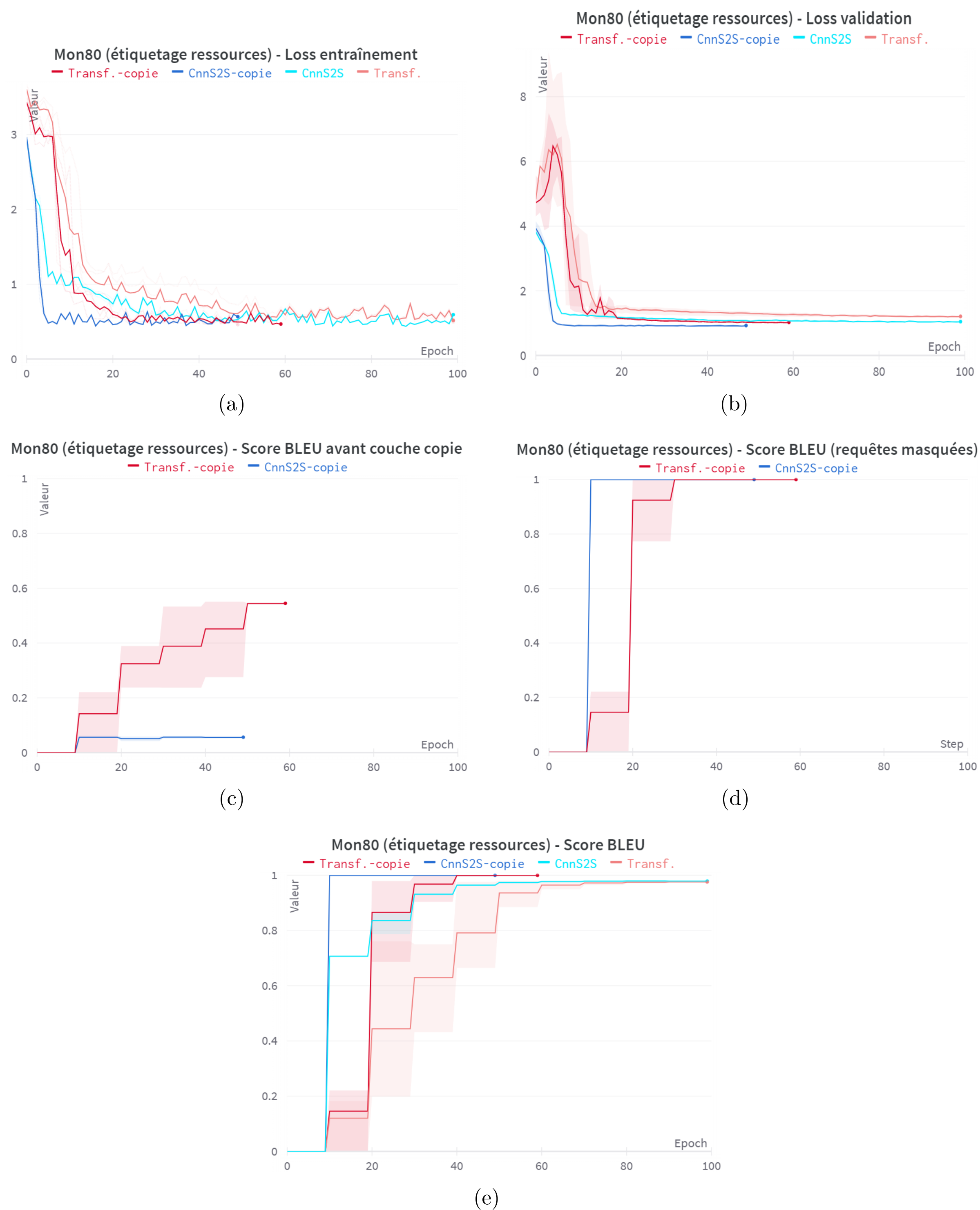


Figure C.7 Résultats sur Monument 80 avec seules les ressources étiquetées

LC-QuAD questions intermédiaires

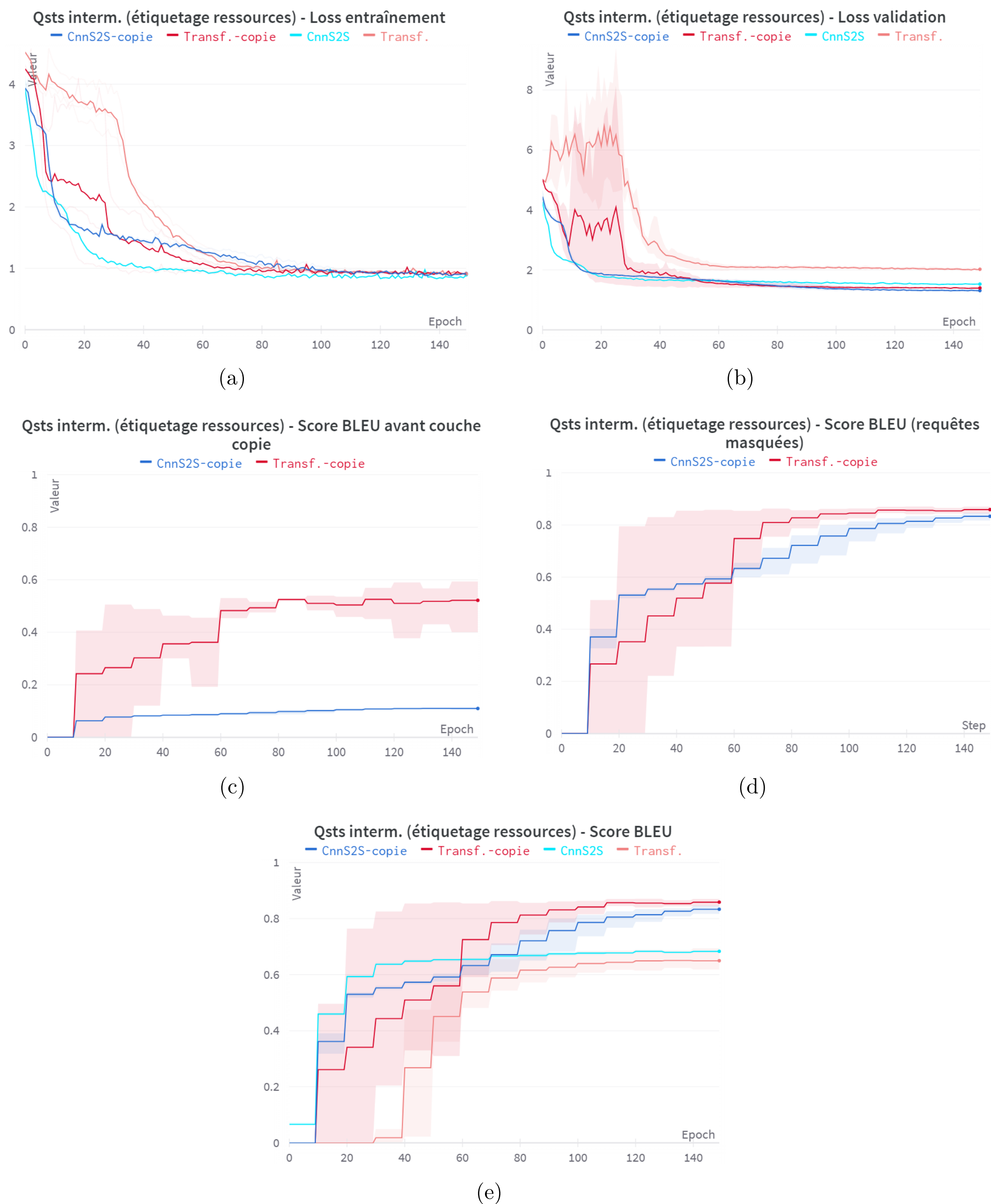


Figure C.8 Résultats sur les questions intermédiaires de LC-QuAD dans lesquelles seules les ressources sont étiquetées

Versions étiquetées partiellement - schéma

Monument

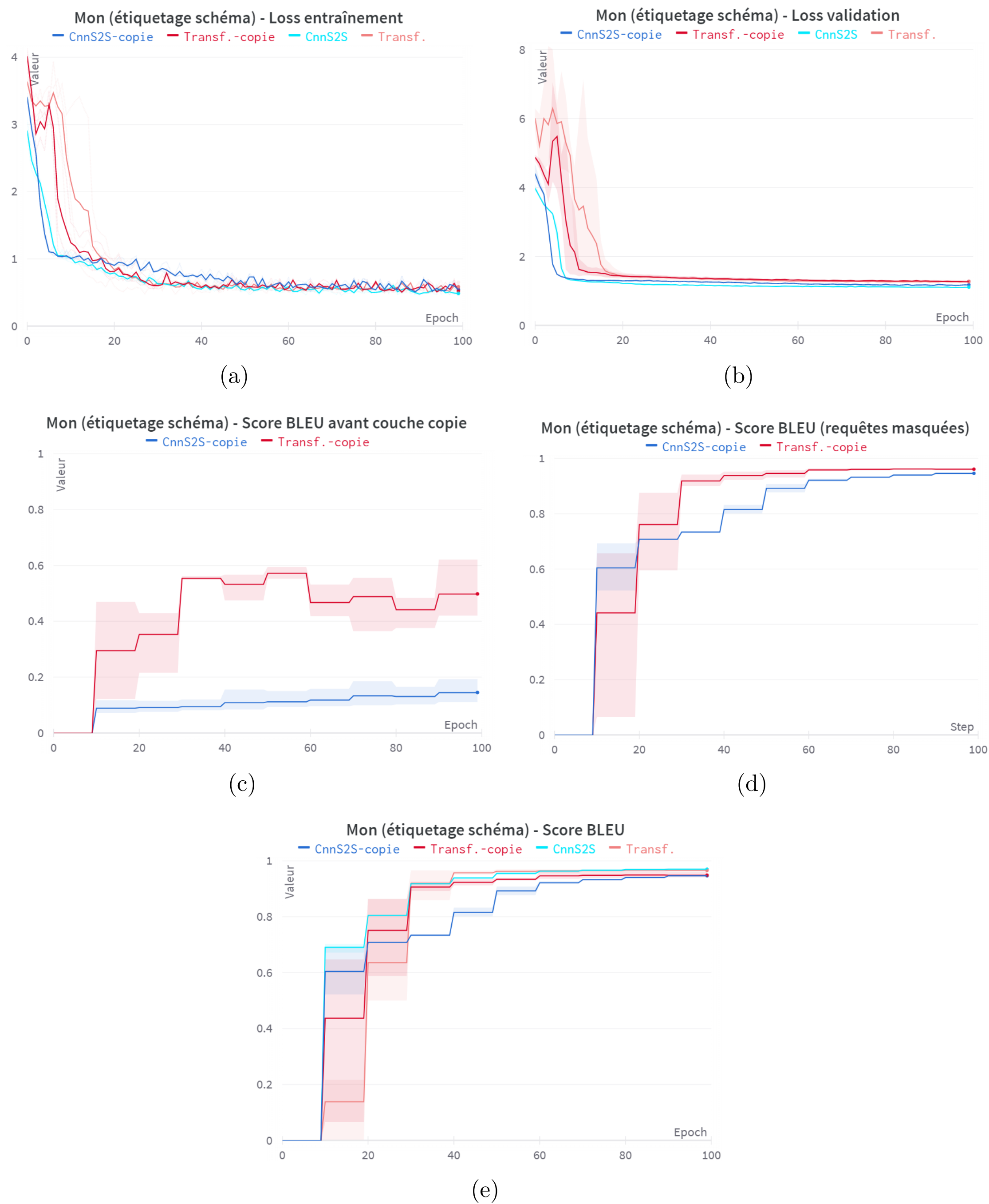


Figure C.9 Résultats sur Monument avec seuls les éléments du schéma étiquetés

Monument50

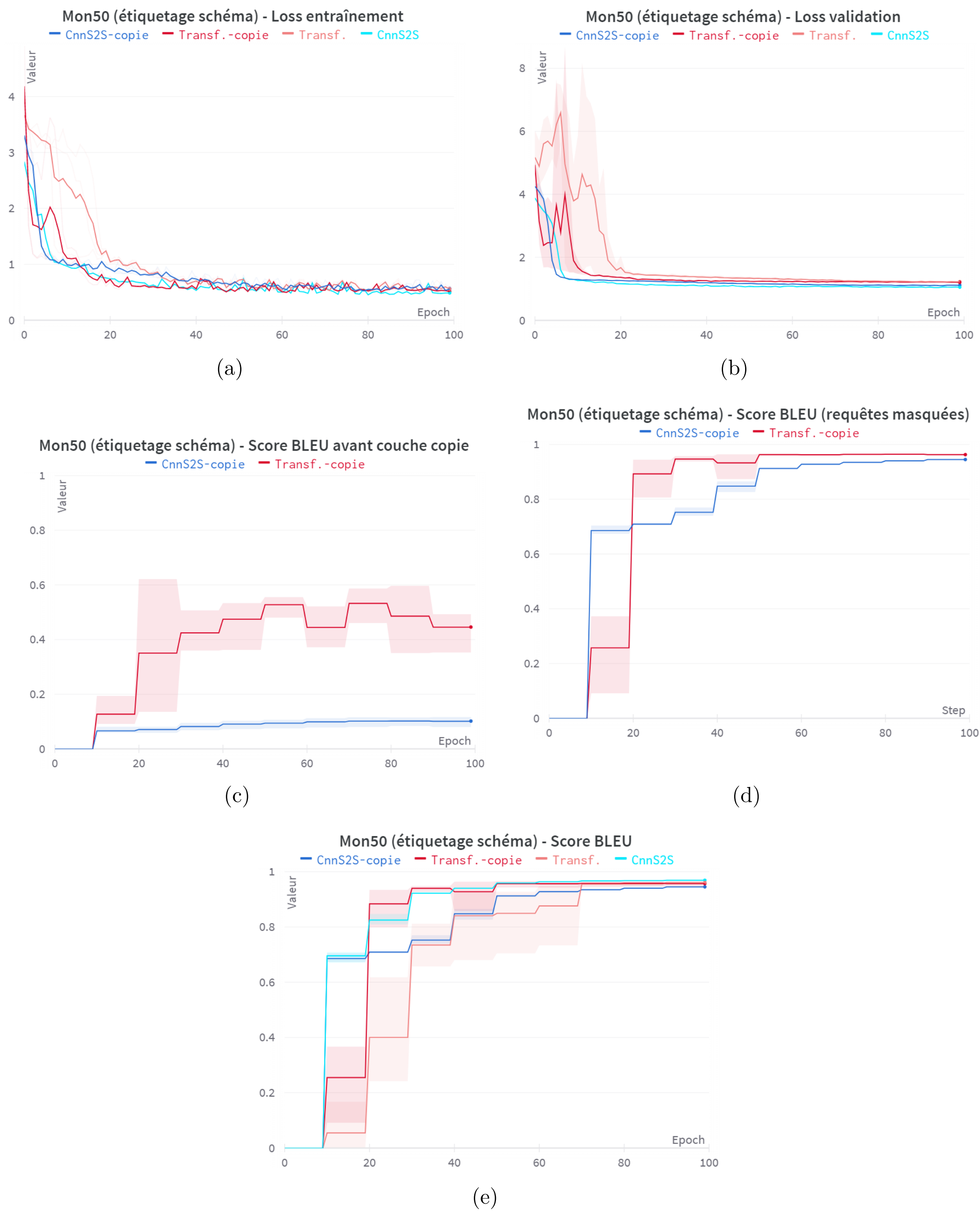


Figure C.10 Résultats sur Monument 50 avec seuls les éléments du schéma étiquetés

Monument80

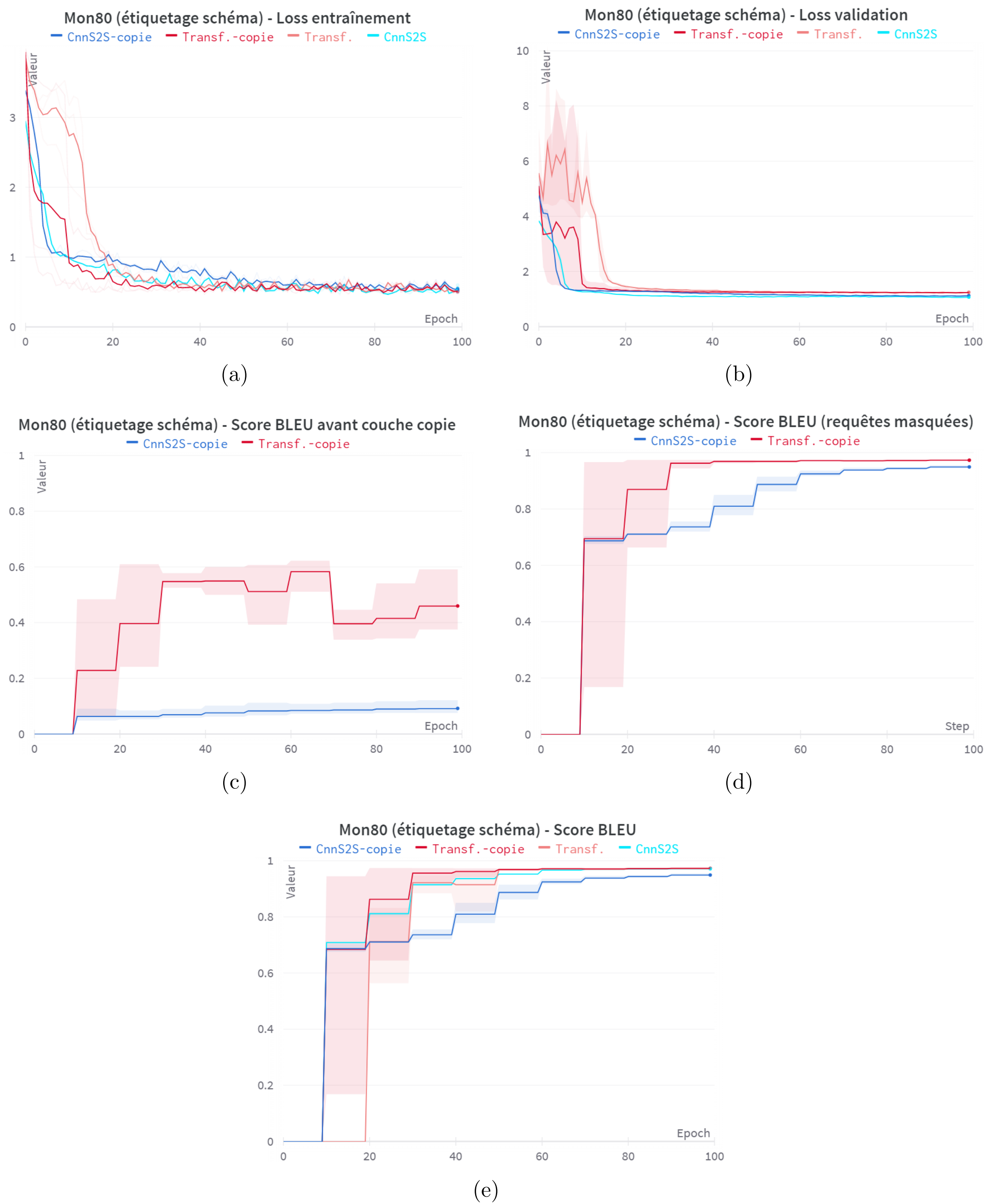


Figure C.11 Résultats sur Monument 80 avec seuls les éléments du schéma étiquetés

LC-QuAD questions intermédiaires

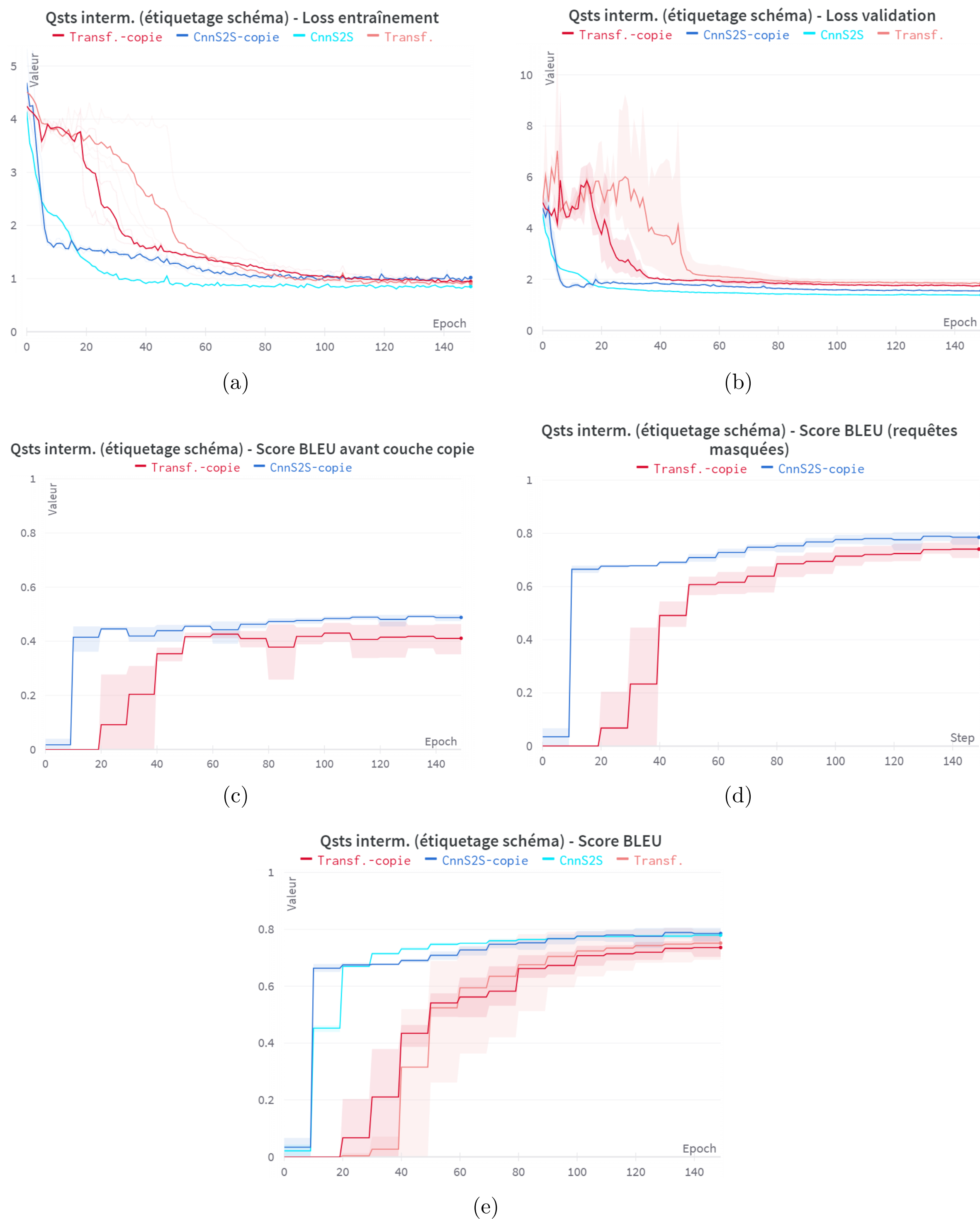


Figure C.12 Résultats sur les questions intermédiaires de LC-QuAD dans lesquelles seuls les éléments du schéma sont étiquetés

Versions complètement étiquetées

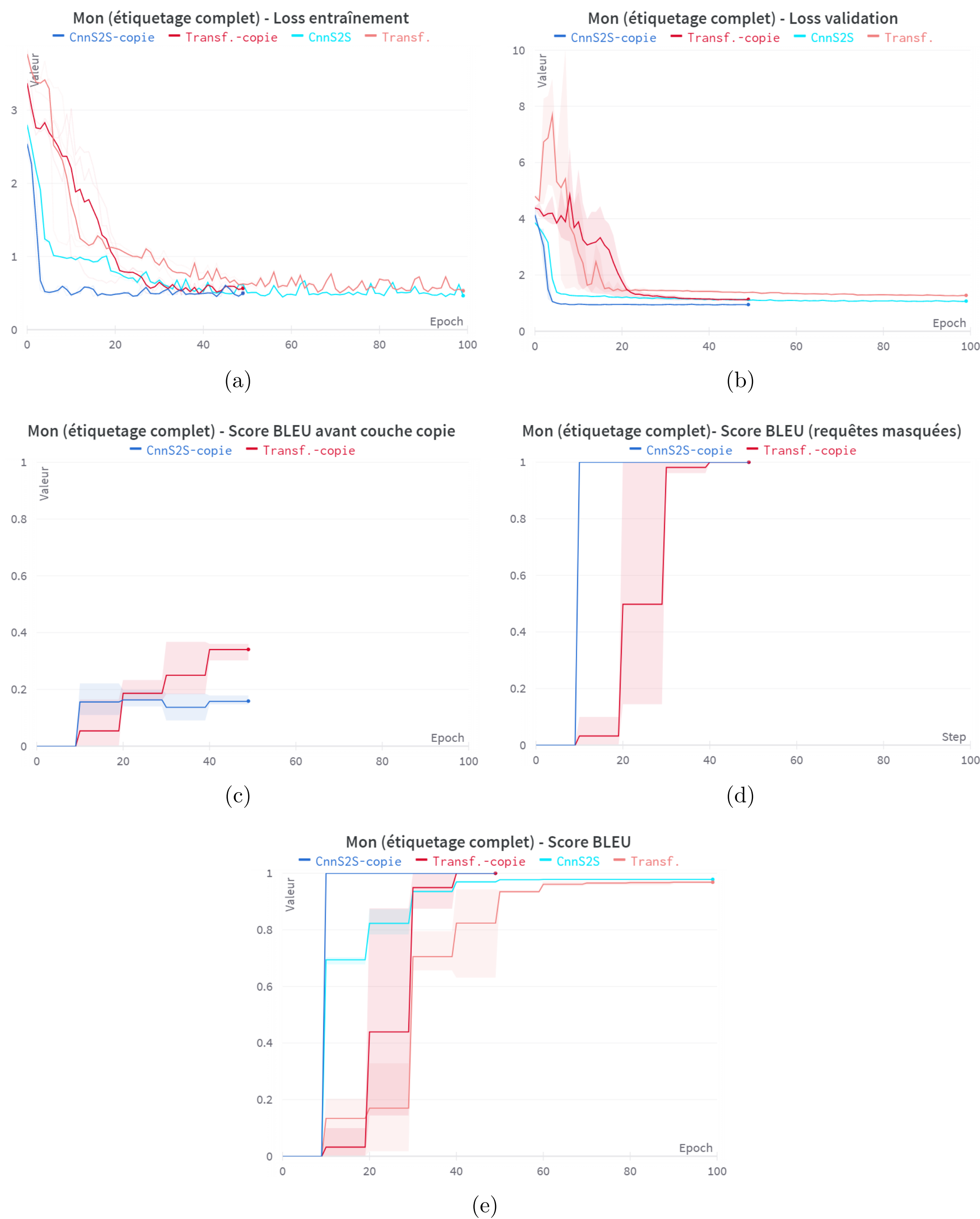


Figure C.13 Résultats de la version complètement étiquetée de Monument

Monument50

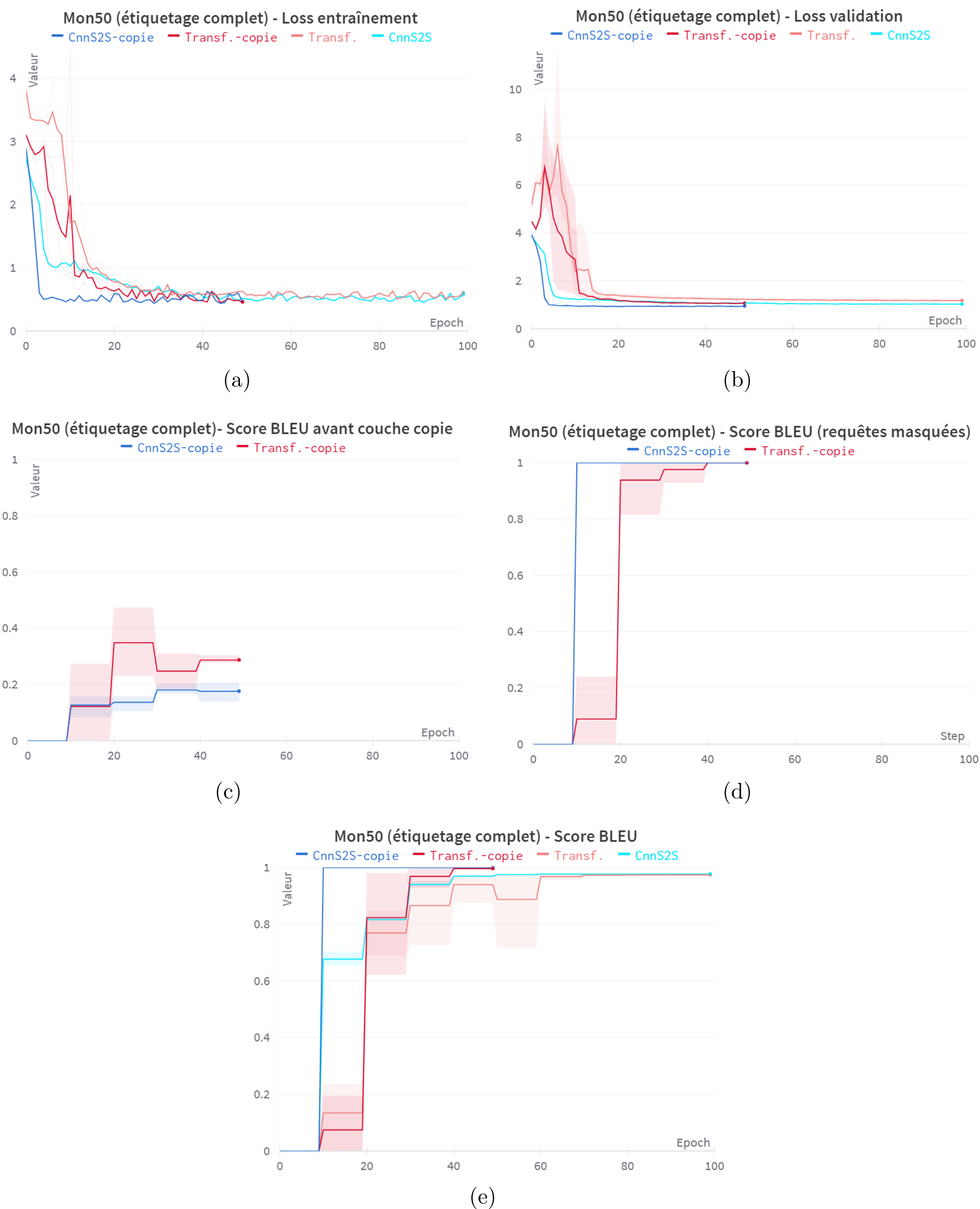


Figure C.14 Résultats de la version complètement étiquetée de Monument 50

Monument80

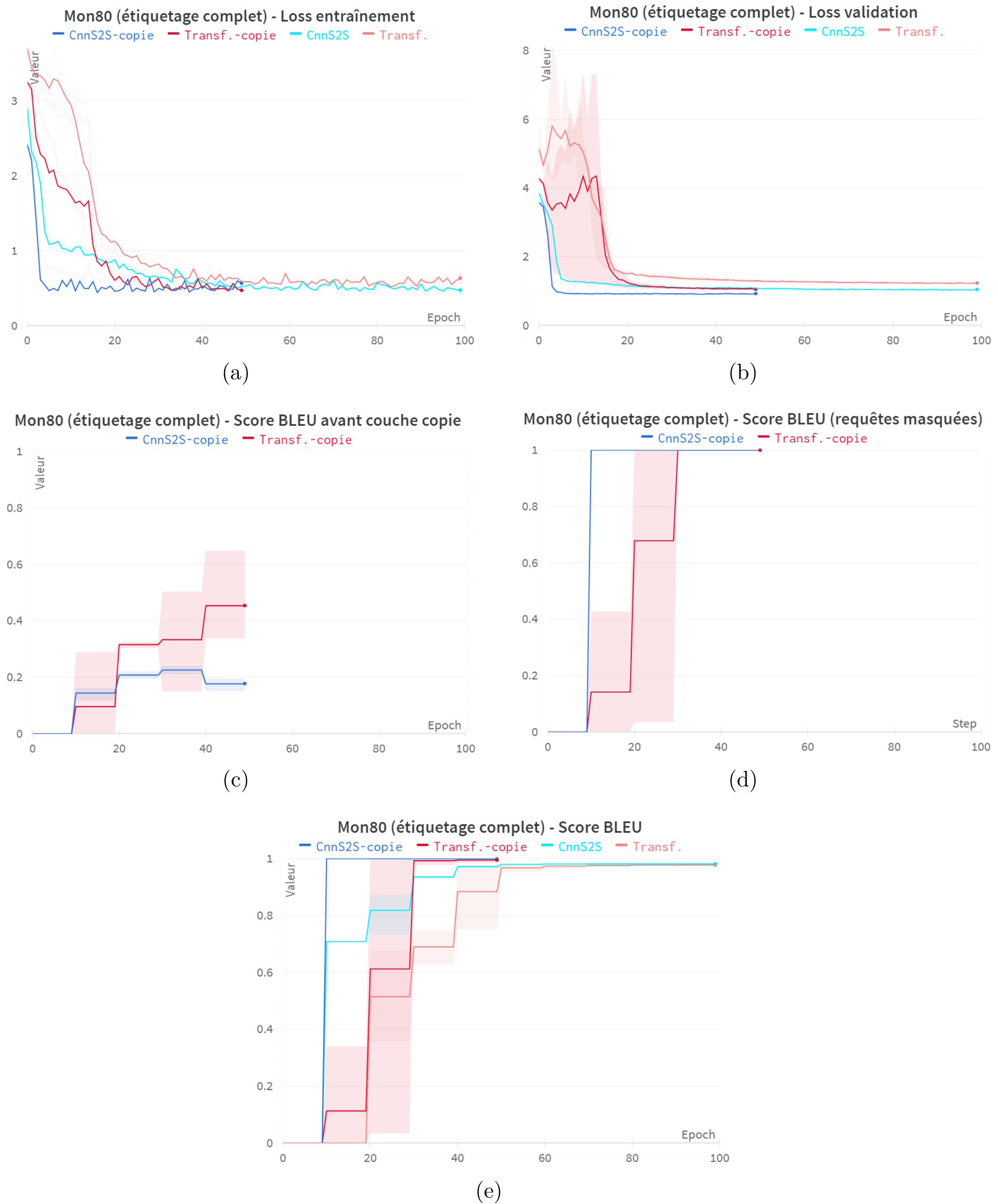


Figure C.15 Résultats de la version complètement étiquetée de Monument 80

LC-QuAD questions intermédiaires

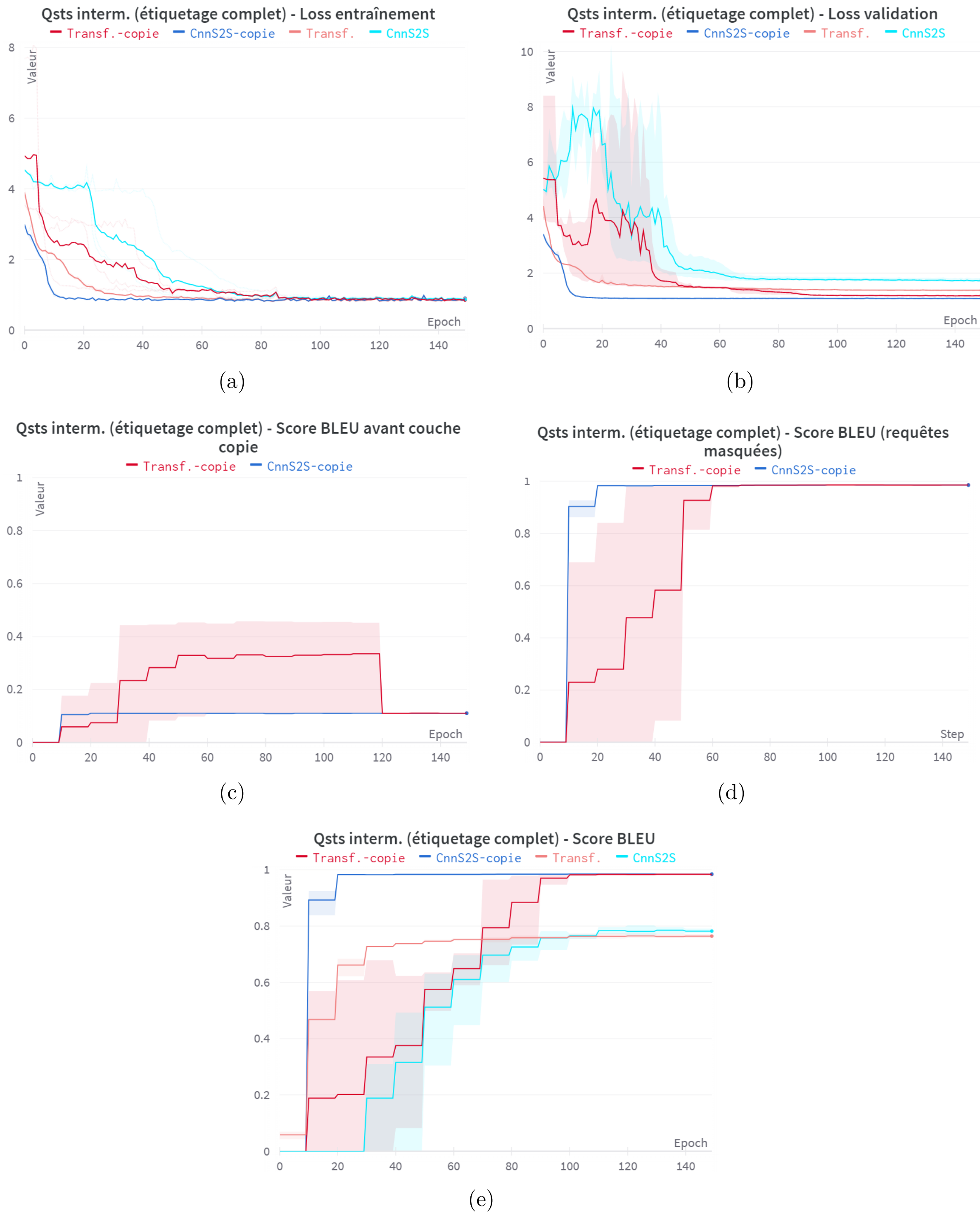


Figure C.16 Résultats de la version complètement étiquetée des questions intermédiaires de LC-QuAD

ANNEXE D SCORES DE PRÉCISION, RAPPEL ET F1

Cette annexe présente la précision, le rappel et le score F1 de chaque jeu de données avec chaque architecture. Les valeurs présentées sont les moyennes des résultats obtenus sur trois entraînements.

Pour un jeu de données, on utilise les équations D.1 et D.2 pour calculer la précision et le rappel. Pour chaque question, on connaît R_{att} le résultat retourné par la requête de référence, et R_{pred} le résultat obtenu avec la requête générée. R_{att} prend la forme d'une liste de $0...N_{att}$ réponses, et R_{pred} prend la forme d'une liste de $0...N_{pred}$ réponses. On définit une bonne réponse comme une réponse se trouvant dans R_{att} et dans R_{pred}. La variable N_B représente le nombre de bonnes réponses obtenues par une requête générée automatiquement.

$$prec_R = N_B/N_{pred} \tag{D.1}$$

$$rappel_R = N_B/N_{att} \tag{D.2}$$

On calcule la précision d'un jeu de données en faisant la moyenne des précisions obtenues pour chaque question, et on fait de même pour le rappel. On utilise la précision moyenne *prec* et le rappel moyen *rappel* afin de calculer le score F1 avec l'équation D.3.

$$F1 = 2 * \frac{prec * rappel}{prec + rappel} \tag{D.3}$$

Si le résultat obtenu ou le résultat attendu est vide (N_{pred} ou N_{att} égal à 0), la précision, le rappel et le score F1 sont tous de 0.

Données originales

Tableau D.1 Résultats des architectures sans copie sur les données originales de LC-QuAD

	TNTSPA			Qsts Reformulées			Qsts Intermédiaires		
Architecture	Préc.	Rappel	F1	Préc.	Rappel	F1	Préc.	Rappel	F1
Transformer	17.83	17.87	17.86	17.94	18.00	17.97	21.72	22.29	21.99
CnnS2S	9.81	10.17	9.99	10.12	10.27	10.19	18.05	18.63	18.34

Tableau D.2 Résultats des architectures sans copie sur les données originales de Monument

	Mon			Mon50			Mon80		
Architecture	Préc.	Rappel	F1	Préc.	Rappel	F1	Préc.	Rappel	F1
Transformer	76.38	76.55	76.47	79.17	79.22	79.19	78.76	78.90	78.83
CnnS2S	77.44	77.53	77.49	81.40	77.35	79.71	75.46	71.78	73.56

Tableau D.3 Résultats des architectures originales sur les données hors du vocabulaire (HV)

	Mon HV			Qsts Intermédiaires HV		
Architecture	Préc.	Rappel	F1	Préc.	Rappel	F1
Transformer	4.20	3.53	3.83	62.40	62.40	62.40
CnnS2S	6.11	5.28	3.83	74.09	74.09	74.09

Données corrigées

Tableau D.4 Résultats des architectures sans copie sur les données corrigées de LC-QuAD

	Qsts Reformulées			**Qsts Intermédiaires**		
Architecture	Préc.	Rappel	F1	Préc.	Rappel	F1
Transformer	17.26	17.36	17.31	21.69	22.56	22.12
CnnS2S	9.71	9.93	9.82	17.58	18.00	17.79

Tableau D.5 Résultats des architectures sans copie sur les données corrigées de Monument

	Mon			**Mon50**			**Mon80**		
Architecture	Préc.	Rappel	F1	Préc.	Rappel	F1	Préc.	Rappel	F1
Transformer	77.55	77.10	77.33	79.19	80.42	79.80	79.58	79.38	79.48
CnnS2S	78.94	79.08	79.01	80.08	80.81	80.14	78.53	77.63	78.08

Tableau D.6 Résultats des architectures sans copie sur les données hors du vocabulaire (HV) corrigées

	Mon HV			**Qsts Intermédiaires HV**		
Architecture	Préc.	Rappel	F1	Préc.	Rappel	F1
Transformer	4.65	3.96	4.27	64.27	64.27	64.27
CnnS2S	5.52	4.41	4.90	78.91	78.91	78.91

Données partiellement étiquetées - ressources

Tableau D.7 Résultats des architectures sur les données dans lesquelles seules les ressources sont étiquetées

	Mon			**Mon50**			**Mon80**			**Qsts Intermédiaires**		
Données	Préc.	Rappel	F1	Préc.	Rappel	F1	Préc.	Rappel	F1	Préc.	Rappel	F1
Transf.	80.07	78.49	79.27	82.01	79.71	80.84	81.97	81.61	81.79	23.51	23.73	23.61
Transf.-copie	82.81	82.21	82.21	84.24	84.24	84.24	83.22	83.22	83.22	47.43	47.25	47.34
CnnS2S	79.93	79.91	79.92	81.38	81.16	81.26	82.30	82.33	82.31	19.97	20.37	20.17
CnnS2S-copie	82.81	82.21	82.21	84.24	84.24	84.24	83.22	83.22	83.22	40.53	40.37	40.45

Tableau D.8 Résultats des architectures sur les données hors du vocabulaire (HV) dans lesquelles seules les ressources sont étiquetées

	Mon HV			**Qsts Intermédiaires HV**		
Architecture	Préc.	Rappel	F1	Préc.	Rappel	F1
Transf.	4.65	3.91	4.25	59.87	59.87	59.87
Transf.-copie	34.80	34.80	34.80	34.80	34.80	34.80
CnnS2S	2.96	2.39	2.59	84.00	84.00	84.00
CnnS2S-copie	34.80	34.80	34.80	78.53	78.53	78.53

Données partiellement étiquetées - schéma

Tableau D.9 Résultats des architectures sur les données dans lesquelles tous les éléments sauf les ressources sont étiquetés

	Mon			**Mon50**			**Mon80**			**Qsts Intermédiaires**		
Données	Préc.	Rappel	F1	Préc.	Rappel	F1	Préc.	Rappel	F1	Préc.	Rappel	F1
Transf.	77.89	77.97	77.93	78.18	78.92	78.55	79.12	79.28	79.20	23.42	24.08	23.74
Transf.-copie	75.74	75.99	75.86	80.19	79.55	78.86	79.45	79.59	79.52	16.24	16.33	16.29
CnnS2S	79.41	78.47	78.94	80.25	80.05	80.28	79.15	79.52	79.34	23.36	23.70	23.53
CnnS2S-copie	97.78	60.61	71.44	85.38	76.40	80.63	90.36	76.26	82.68	23.52	24.35	23.92

Tableau D.10 Résultats des architectures sur les données hors du vocabulaire (HV) dans lesquelles tous les éléments sauf les ressources sont étiquetés

	Mon HV			**Qsts Intermédiaires HV**		
Architecture	Préc.	Rappel	F1	Préc.	Rappel	F1
Transf.	4.57	3.63	4.05	58.67	58.67	58.67
Transf.-copie	0.96	0.28	0.43	16.40	16.40	16.40
CnnS2S	4.51	3.66	4.04	58.67	58.67	58.67
CnnS2S-copie	0.00	0.00	0.00	31.07	31.20	31.13

Données complètement étiquetées

Tableau D.11 Résultats des architectures sur les données complètement étiquetées

	Mon			Mon50			Mon80			Qsts Intermédiaires		
Données	Préc.	Rappel	F1	Préc.	Rappel	F1	Préc.	Rappel	F1	Préc.	Rappel	F1
Transf.	78.96	78.36	78.66	83.60	81.17	82.36	80.83	81.17	81.00	23.28	24.00	23.63
Transf.-copie	82.19	82.19	82.19	84.24	84.24	84.24	83.22	83.22	83.22	60.47	60.47	60.47
CnnS2S	80.18	80.80	80.49	82.82	82.00	82.40	81.50	82.06	81.78	22.89	23.44	23.16
CnnS2S-copie	82.18	82.18	82.18	84.24	84.24	84.24	83.22	83.22	83.22	60.28	60.27	60.27

Tableau D.12 Résultats des architectures sur les données hors du vocabulaire (HV) complètement étiquetées

	Mon HV			Qsts Intermédiaires HV		
Architecture	Préc.	Rappel	F1	Préc.	Rappel	F1
Transf.	4.82	3.57	04.09	54.27	54.27	54.27
Transf.-copie	0.96	0.28	0.43	16.40	16.40	16.40
CnnS2S	34.80	34.80	34.80	34.80	34.80	34.80
CnnS2S-copie	34.80	34.80	34.80	78.53	78.53	78.53