Introduction générale

> *“Nous entrons dans un nouveau monde. Les technologies d’apprentissage automatique, de reconnaissance de la parole et de compréhension du langage naturel atteignent un niveau de capacités. Le résultat final est que nous aurons bientôt des assistants intelligents pour nous aider dans tous les aspects de nos vies.”*
>
> – Amy Stapleton, Opus Research

Plus de quarante ans se sont écoulés depuis la présentation du premier assistant virtuel commandé vocalement par la compagnie IBM (VanDijck, 2005). Déjà à cette époque là, ce fut présenté comme une révolution technologique. Un programme qui pouvait reconnaître 16 mots et les chiffres de 0 à 9. Quelques générations plus tard, nous nous retrouvons avec des assistants capables de reconnaître, comprendre et parler plusieurs langues. Ces assistants intelligents sont la nouvelle génération d’intelligences artificielles capables de s’intégrer dans nos vies personnelles et professionnelles.

Un domaine qui a particulièrement émergé est celui de l’aide à la manipulation d’un ordinateur personnel. Selon certains experts (Milhorat et al., 2014 ; Trappl, 2013 ; Knote et al., 2018), l’époque où nous utilisons encore le clavier, la souris et l’écran est une étape de transition. Le futur se trouve dans l’exploitation de la parole comme moyen de communication principal avec les machines. Et ce futur est proche. La course à la réalisation d’assistants qui excellent dans plusieurs domaines a commencé il y a quelques années avec l’entrée de grandes compagnies comme Google et Apple dans le secteur (Tulshan and Namdeorao Dhage, 2019). Par la suite, de très grands efforts ont été fournis dans le but d’améliorer l’expérience d’utilisation de ces assistants. Les chercheurs ainsi que les industriels se sont orientés vers cette solution très rapidement.

Nous sommes sans doute actuellement entrain de vivre une époque importante de l’intelligence artificielle. Le temps ou nous rêvions encore de converser avec une machine semble à la fois proche en terme de temps, mais loin en terme de progrès. Les utilisateurs réguliers de produits technologiques sont confrontés à une technologie nouvelle mais prometteuse. Ce secteur d’activités peut donc s’avérer très prolifique si les efforts fournis sont suffisamment conséquents.

En tant que novice dans ce secteur qu'est la personnalisation des services électroniques, l'Algérie devra rapidement se positionner pour s'incorporer dans l'évolution de ces technologies. Nous avons donc été motivés par l'envie de nous initier à ce domaine, ainsi que de contribuer aux travaux d'autres compatriotes scientifiques qui traiteraient de ce sujet. Nous pensons que les plus ambitieux des projets commencent avec des contributions à petites échelles. Notre assistant aura pour but d'améliorer l'expérience d'utilisation d'un ordinateur. Ceci en effectuant des tâches rudimentaires, efficacement et sans réel effort hormis l'énonciation de la requête. En utilisant des techniques d'intelligence artificielle d'actualité comme la reconnaissance automatique de la parole, la compréhension du langage naturel et l'apprentissage par renforcement, ce projet se veut assez ambitieux et vise à faire gagner du temps à tout utilisateur d'un ordinateur de bureau ou portable.

Dans cette optique, nous nous sommes inspirés de notre étude des travaux de la littérature sur les assistants personnels intelligents. Nous passerons en revue les aspects théoriques qui sont utilisés dans des solutions considérées comme état de l'art du domaine. Nous nous baserons sur ces techniques pour la conception des modules de notre système tout en les adaptant à nos besoins.

Ce mémoire se constitue de quatre chapitres. Le premier chapitre sera consacré à la présentation des assistants virtuels intelligents. Le deuxième chapitre se focalisera sur l'étude des travaux existants liés à la thématique de notre sujet. Le troisième chapitre traitera de l'étude conceptuelle. Le quatrième et dernier chapitre présentera notre système Speech2Act avec son évaluation, une synthèse des résultats obtenus et notre interface pour l'application. Enfin, nous clôturerons ce travail avec une conclusion générale et les perspectives envisageables.

Conclusion générale

La création d'un nouveau besoin qu'est la personnalisation des services, a été la principale source de motivation pour ce projet. Cependant, et par faute de temps ainsi que de moyens techniques (surtout en ce qui concerne le côté matériel pour l'apprentissage automatique), certains modules n'ont pas étés exploités à leur maximum. Cela ne nous a pas empêché de réaliser un travail dont nous sommes particulièrement fières. Mais, nous gardons toutefois un esprit critique, ainsi qu'une objectivité envers le travail fourni.

Tout au long de la réalisation de ce mémoire, nous avons étudié l'état actuel des assistants personnels intelligents. Nous avons dû passer une majeure partie de cette étape à comprendre les fondements théoriques et conceptuelles de chaque composants de ces systèmes, principalement à cause de la grande densité de techniques, concepts et théories qui sont nouvelles pour nous.

Après avoir assimilés la totalité des concepts, et qui font office d'état de l'art du domaine, nous sommes arrivés à certaines conclusions. Tout d'abord, développer un système en partant de rien était un travail assez massif. Dépassant de loin le cadre d'un projet de fin d'études de master. Nous avons donc mis l'accent sur certains modules, le module de compréhension du langage naturel et le module de gestion du dialogue. Ces derniers dépendaient énormément de notre problématique. Le module de reconnaissance automatique de la parole a été sujet à une amélioration spécifique à nos besoins tout en exploitant un système de base déjà existant (à savoir DeepSpeech).

Avec une idée claire du travail à réaliser, nous avons pu entamer la conception de chaque module en y incorporant nos ajouts et modifications. Beaucoup de ces modifications sont le fruit de longues séances de débat et de discussions.

En ce qui concerne le module de reconnaissance automatique de la parole, l'ajout de ce modèle de langue a amélioré les résultats. Cela s'accordait avec nos prédictions théoriques. La partie de construction du corpus pour ce modèle a été soumise à beaucoup d'optimisations incrémentales, en tombant à chaque fois sur un nouveau problème, ou bien un obstacle matériel (manque de puissance de calcul). En conséquent, les résultats n'étaient pas assez encourageants. Surtout si le but est de détrôner les systèmes propriétaires comme celui de Google qui réalise un score quasi-parfait sans apprentissage supplémentaire.

Pour le module de compréhension automatique du langage naturel, les ajouts faits au modèle d'apprentissage ont été expérimentalement validés dans le chapitre Réalisations et résultats. L'ajout de l'information morphosyntaxique a permis de donner plus de valeur sémantique à chaque mots de la requête. L'introduction d'erreurs aléatoires a quand à elle permis la gestion d'éventuelles erreurs que pourrait engendrer le module de reconnaissance automatique de la parole. La construction de l'ensemble d'apprentissage à partir de zéro était l'étape la plus longue de la réalisation de ce module. Plus l'ensemble grandissait, plus il était difficile de maintenir sa validité sans l'intervention d'un soutien externe. De plus, vu que la tâche à accomplir était relativement simple et limitée, le réel impacte de cet ajout ne peut pas être certifié et validé dans un cadre plus général. Une autre problématique est celle du manque de données d'apprentissage consacrées au domaine de la manipulation d'ordinateurs. Ces données sont généralement construites manuellement par les développeurs du système. Un autre point à soulever est celui du manque de diversité dans les tâches réalisable par l'assistant. Cet ensemble de tâches reste facilement extensible. Il suffit d'ajouter des exemples assez exhaustifs à l'ensemble de données. Cela reste néanmoins une tâche lourde et manuelle à plus grande échelle.

En ce qui concerne le gestionnaire de dialogue, nous l'avons conçu pour qu'il soit facilement ajustable et mis à l'échelle. L'utilisation d'une architecture hiérarchisée d'agents de dialogue permet de réduire grandement la complexité de développement et d'ajout d'un nouveau gestionnaire de tâches dans le système. Pour représenter l'état interne d'une telle architecture, nous avons utilisé des graphes de connaissances au lieu des trames sémantiques. En effet, leur flexibilité permet de représenter tout l'état du dialogue des différents agents de l'architecture simultanément. Pour ce qui est des agents de dialogue, ils sont entraînés en utilisant des techniques d'apprentissage par renforcement. Ils interagissent avec un simulateur d'utilisateur afin d'atteindre son but à travers l'optimisation d'une politique d'actions basée sur un système de récompenses.

Cependant, l'utilisation du graphe de connaissances et de l'apprentissage profond a un prix. En effet, la codification de la totalité du graphe en un seul vecteur de taille fixe était une tâche assez difficile. La taille nécessaire de ce dernier devrait augmenter exponentiellement avec l'ajout de nouvelles connaissances. De plus, la codification choisies pour les nœuds du graphe a eu pour effet de faire perdre de l'information sémantique. Le décodage du graphe s'en est trouvé grandement affecté.

Pour ce qui est de l'application, nous avons fait le choix d'utiliser une architecture trois tiers basée web. Ce choix fût motivé par le fait que l'utilisation d'un serveur offre une puissance de calcul considérablement plus élevée que celle d'une machine personnel en local. L'interface reste assez simple et épurée. Le but est de prioriser la parole comme moyen de communication avec le système. Cependant, cette interface a aussi pour but de montrer les fonctionnalités du système. Elle est donc plus orientée vers le développement.

En ce qui concerne les perspectives envisageables pour ce travail, nous avons longuement réfléchis à des alternatives possibles pour certains modules.

Premièrement, l'avenir de systèmes Open source pour la reconnaissance automatique de la parole est très prometteur. Ces derniers offrent un moyen libre de mener des études et contribuer au développement à grande échelle de cette discipline. Une perspective future pour ce module serait de lancer notre propre plateforme de collecte de données. Cette idée a déjà été discutée dans la partie de l'étude de l'état de l'art. Malheureusement, le temps a cruellement manqué. L'investissement de la communauté dans de telles initiatives n'en reste pas moins indéniable, comme l'a démontré le projet CommonVoice de Mozilla.

Deuxièmement, pour le module de compréhension automatique du langage naturel, faire appel à une collecte massive de données est une solution explorable. Le développement d'un outil d'aide à l'annotation d'un corpus était aussi le sujet d'un long débat. le manque de temps nous a poussé à retarder son développement. La mise à l'échelle d'une telle plateforme pourrait grandement faire avancer la tâche fastidieuse qu'est la collecte de données.

Ensuite, en ce qui concerne le module de gestion du dialogue, il est envisageable d'utiliser une méthode d'apprentissage semi-supervisée pour l'encodage des nœuds du graphe. Cette ajout pourrait permettre d'enrichir la valeur sémantique de ces nœuds, et facilitera la tâche au réseau de neurones de l'agent apprenant pour le décodage du graphe. Principalement grâce au fait que les nœuds dont les sens sont arbitrairement proches auront des codifications similaires.

En ce qui concerne le module de génération du langage naturel, une méthode plus sophistiquée comme l'utilisation de modèles d'apprentissage automatique basés encodeur-décodeur, ou bien un convertisseur de graphes de connaissances en texte pourraient être utilisés. Cependant, ces architectures requièrent un très grand volume de données d'apprentissage annotés et spécifique à notre problématique.

Enfin, pour ce qui est de l'application principale, une amélioration possible serait le déploiement du serveur dans un service de Cloud Hosting. Cela permettrait de minimiser les temps d'inférence et de post-traitement. Le développement d'une interface plus légère et plus orientée vers les cas pratiques est une amélioration possible. Garder les deux cas de figures, c.à.d. utilisation et développement, est aussi possible.

Pour conclure, nous estimons que la totalité du projet était une énorme occasion d'approfondir nos connaissances. Que ce soit celles qui nous ont été enseignées durant notre cursus, comme l'apprentissage automatique, le traitement automatique du langage naturel, le web-sémantique et la représentation de connaissances. Ou bien celles que nous avons appris au cours de notre étude de la littérature, comme l'apprentissage par renforcement, le traitement automatique de la parole, le nettoyage des données, etc. Ce projet nous a aussi initié au travail en équipe pour la réalisation d'un projet assez conséquent. Tout en étant encadrés par nos supérieurs. Finalement, nous pensons que la plus grande satisfaction vient du fait que nous avons réalisé, dans un certain délai restreint, un travail qui traite d'un sujet récent et ambitieux. Et ainsi, de poser la première pierre à l'édifice pour, idéalement, encourager les chercheurs en Algérie à s'investir dans ce domaine.

1

Conception du système

1.1 Introduction

Dans ce chapitre, nous allons présenter en détail les étapes de conception de notre système no. De prime abord une architecture générale est introduite puis décortiquée. Ensuite, chaque module du système sera détaillé du point de vue des composants qui le constituent. Une conclusion viendra ensuite clôturer ce chapitre.

1.2 Architecture du système

Comme montré dans la figure 1.1 et comme cité dans le chapitre précédent (voir **??**) le système Speech2Act se présente comme l'interconnexion de cinq parties dont une interface [1] et quatre modules internes communiquant entre eux. Chaque module forme ainsi un maillon d'une chaîne qui représente une partie du cycle de vie du système. L'architecture de Speech2Act est un pipeline (chaîne de traitement) de processus qui s'exécutent de manière indépendante mais qui font circuler un flux de données entre eux dans un format préalablement établi (voir **??**). Nous pouvons séparer ces parties en deux catégories : la partie utilisateur et la partie interne du système que nous présentons ci-dessous.

1. Par interface nous entendons le sens abstrait du terme et non obligatoirement le sens interface graphique.

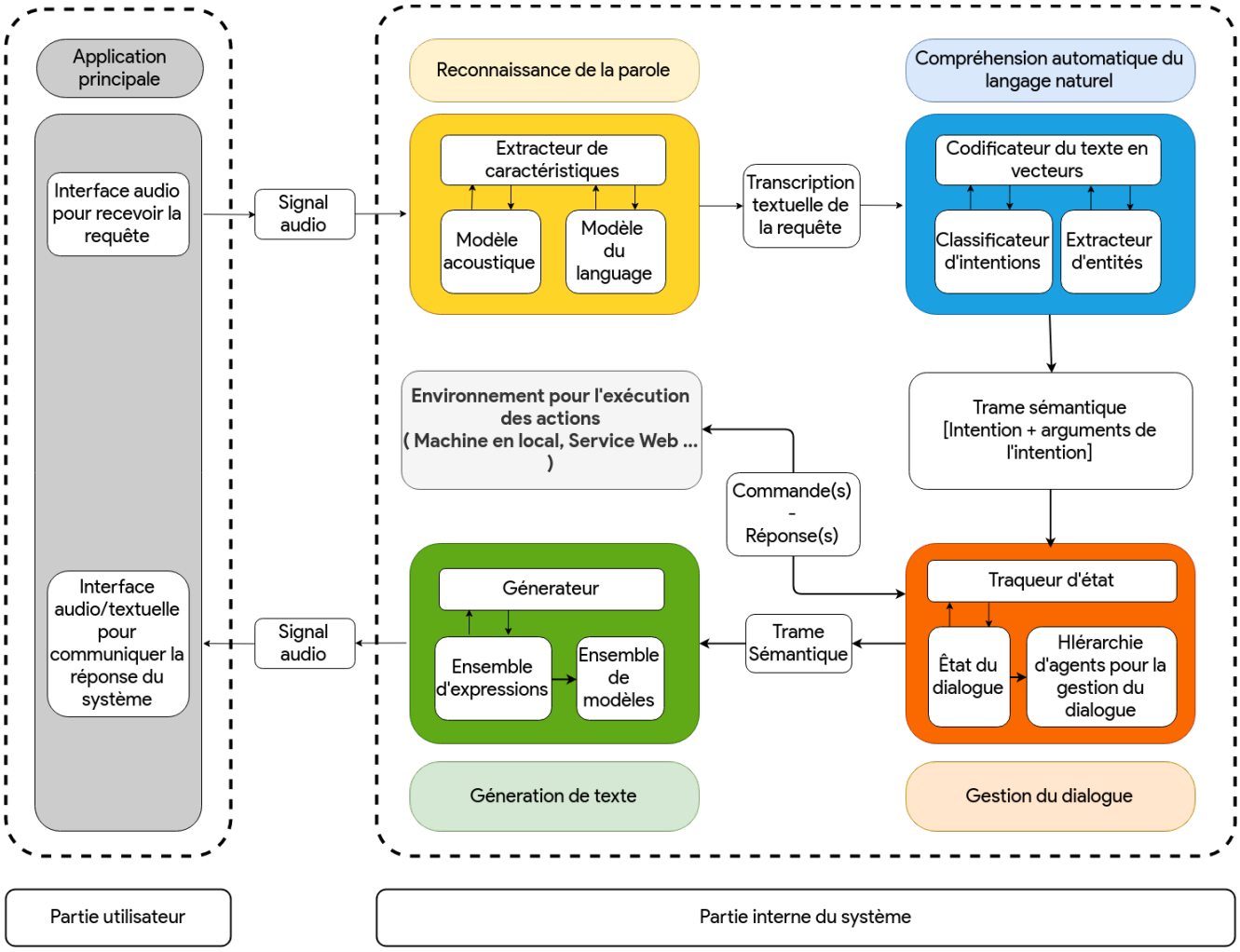

Figure 1.1 – *Architecture générale du système Speech2Act*

1.2.1 Partie utilisateur

Cette partie représente ce que l'utilisateur peut voir comme entrée/sortie et les interfaces qui lui sont accessibles. Puisque l'assistant est un processus qui communique majoritairement avec l'utilisateur à travers des échanges verbaux, nous avons pensé à implémenter l'interface du système comme un processus qui s'exécute en arrière plan et qui attend d'être activé, dans notre cas par un événement physique, c.à.d un clic sur un bouton/icône ou raccourci clavier. L'assistant pourra ensuite répondre en affichant un texte à l'écran qui sera vocalement synthétisé et envoyé à l'utilisateur via l'interface de sortie de son choix. Afficher le texte et sa transcription vocale pourrait palier à certains manques comme l'absence d'un périphérique de sortie audio.

1.2.2 Partie interne du système

Cette partie quant à elle représente ce que l'utilisateur ne voit pas. Elle fait donc partie du fonctionnement interne du système. Elle regroupe les quatre grandes étapes d'un

cycle de vie pour une commande reçue de la couche utilisateur. Comme mentionné dans le chapitre précédent (voir la section **??**), la requête passe par un module de reconnaissance de la parole, qui traduira en texte le signal audio correspondant. Le module de "compréhension du langage naturel" extrait l'intention de l'utilisateur et ses arguments (par exemple *"open the home folder"* pourrait donner une intention de type *open_file_desire[file_name="home",parent_directory="?"]*). Le gestionnaire de dialogue gardera trace de l'ensemble des échanges effectués entre l'utilisateur et l'assistant et essayera d'atteindre le but final de la requête (récente ou ancienne). Pour ce faire, il aura besoin d'interagir avec ce qu'on a appelé un environnement d'exécution, qui peut être la machine où l'assistant est installé ou bien une API [2] qui aura accès à un service à distance (sur internet par exemple) ou local (dans un réseau domestique). Finalement, une action spéciale qui servira à informer l'utilisateur sera envoyée au module de "génération du langage naturel" pour être transformée en son équivalant dans un langage naturel. Accessoirement, le texte sera vocalement synthétisé et envoyé vers l'interface de sortie de l'application.

Nous allons maintenant détailler la conception des différents modules en précisant à chaque fois le ou les procédés de sa mise en œuvre.

1.3 Module de reconnaissance automatique de la parole

Premier module du système Speech2Act, le module de reconnaissance automatique de la parole (Automatic Speech Recognition, ASR) joue un rôle clé dans le dialogue entre l'utilisateur et la machine. En effet, il doit être assez robuste et précis dans la transcription de la requête en entrée afin de minimiser les erreurs et les ambiguïtés qui peuvent survenir dans le reste du pipeline. Dans cette optique, nous avons décidé de ne pas développer entièrement un sous-système en partant de zéro ; faute de temps et par soucis de précision nous avons opté pour l'exploitation d'un outil open-source nommé DeepSpeech (Hannun et al., 2014). Naturellement, du fait que ce soit un projet open-source nous, avons pu avoir accès à différentes informations concernant le modèle d'apprentissage, d'inférence et la nature des données utilisées pour l'apprentissage et les tests.

1.3.1 Architecture du module ASR

Le module ASR possède une architecture en pipeline dont chaque composant exécute un traitement sur la donnée reçu par son prédécesseur.

2. Application Programming Interface ou interface de programmation applicative.

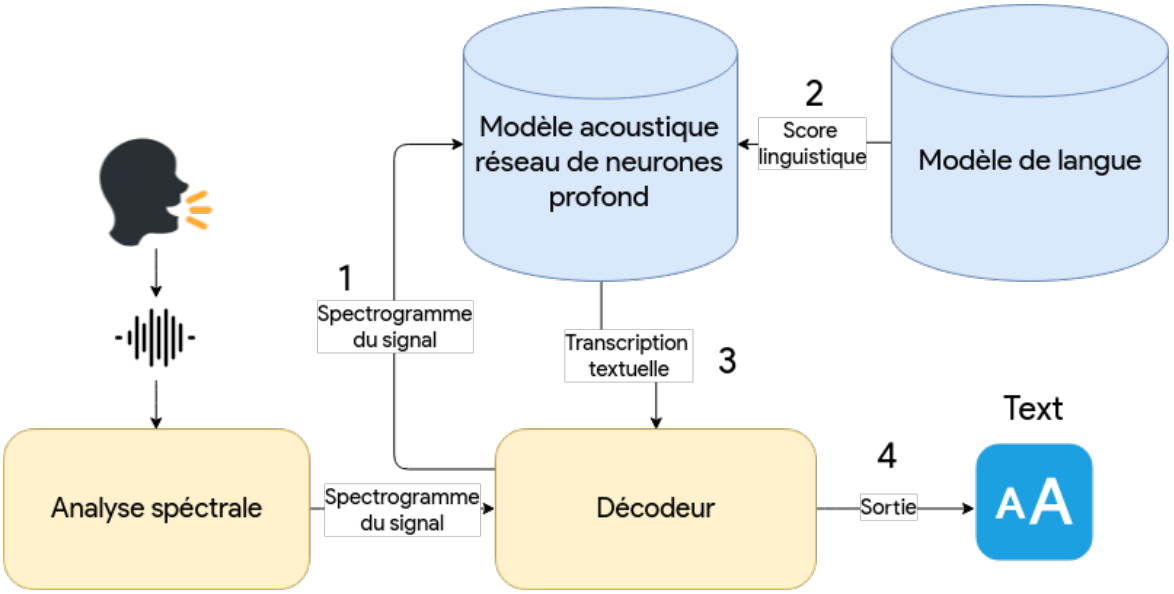

Figure 1.2 – *Architecture du module de reconnaissance de la parole (ASR)*

1.3.2 Modèle acoustique

Type du modèle

Le modèle d'apprentissage, qui est principalement le modèle acoustique à l'exception d'une partie consacré au modèle linguistique, possède une architecture en réseau de neurones avec apprentissage de bout-en-bout composé de trois parties :

- Deux couches de convolution spatiale : pour capturer les patrons dans la séquence du spectrogramme du signal audio.
- Sept couches de récurrence, réseaux de neurones récurrents, pour analyser la séquence de patrons ou caractéristiques engendrée par les couches de convolutions.
- Une couche de prédiction utilisant un réseau de neurones complètement connecté pour prédire le caractère correspondant à la fenêtre d'observation du spectrogramme du signal audio. La fonction d'erreur prend en compte la similarité du caractère produit avec le véritable caractère ainsi que la vraisemblance de la séquence produite par rapport à un modèle de langue basé sur les N-grammes.

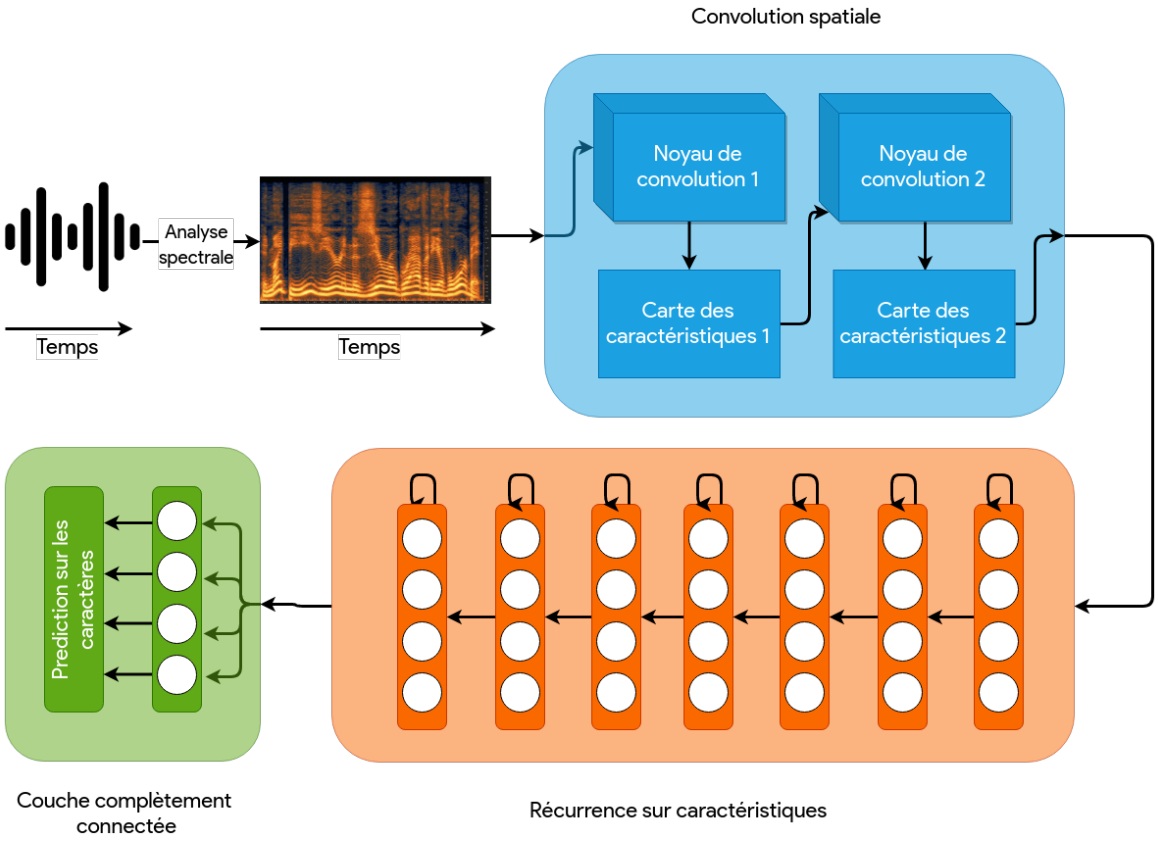

Figure 1.3 – *Architecture du modèle DeepSpeech (Hannun et al., 2014)*

Données d'apprentissage

Pour entraîner le modèle acoustique, Mozilla a lancé le projet Common Voice [3], une plateforme en ligne pour récolter des échantillons audios avec leurs transcriptions textuelles. Chaque batch (lot) de données reçu est alors manuellement validé par l'équipe de Mozilla pour l'inclure dans la banque de données d'exemples principale. À ce jour, pour la langue anglaise, la plateforme a récolté plus de 22Go de données, soit 803 heures d'enregistrements correspondant à plus de 30 000 voix différentes dont 582 heures ont été validées. Cependant, ce volume de données est relativement petit comparé à celui déjà utilisé pour l'apprentissage initial. En effet, plusieurs sources ont été combinées pour construire cet ensemble de données. Dans (Hannun et al., 2014), il a été mentionné que trois ensembles d'apprentissage existants ont été choisis dont WSJ (Wall Street Journal) [4], Switchboard [5] et Fisher [6], qui à eux trois cumulent 2380 heures d'enregistrements audios en anglais et plus de 27 000 voix différentes. Vient s'ajouter à cela, l'ensemble Baidu [7] avec 5000 heures d'enregistrements et 9600 locuteurs.

3. https://voice.mozilla.org/fr
4. http://www.cstr.ed.ac.uk/corpora/MC-WSJ-AV/
5. https://catalog.ldc.upenn.edu/LDC97S62
6. https://catalog.ldc.upenn.edu/LDC2004S13
7. https://ai.baidu.com/broad/introduction

1.3.3 Modèle de la langue

Type du modèle

C'est un modèle basé sur les N-grammes, 3-grammes pour être plus précis, qui est utilisé pour comme modèle de langue. Il permet de façon assez simple et intuitive de capturer l'enchaînement des mots dans une langue donnée, rendant ainsi la transcription finale assez proche de la façon dont les mots sont distribués dans le corpus d'apprentissage.

Données d'apprentissage

À l'origine, DeepSpeech utilise un modèle de langue dont la source n'est pas dévoilée par les chercheurs dans (Hannun et al., 2014). Mais, son volume est approximativement de 220 millions de phrases avec 495 000 mots différents. Cependant, puisque ce corpus nous reste inconnu et qu'il a probablement été construit pour reconnaître des séquence de mots en anglais assez générales, nous avons décidé de construire notre propre modèle de langue en récoltant des données depuis des dépôts sur le site Github, plus précisément les fichiers README.md des dépôts qui font office de manuels d'utilisation d'un projet hébergé sur le site. Ce type de fichiers renferme généralement des instructions de manipulation de fichiers, de lancement de commandes, etc ; ce qui offre un bon corpus pour le modèle de langue. En effet, notre système se concentre plus sur l'aspect de manipulation d'un ordinateur, donc la probabilité de trouver certaines séquences de mots qui appartiennent au domaine technique est en théorie plus élevée. La procédure suivie est la suivante :

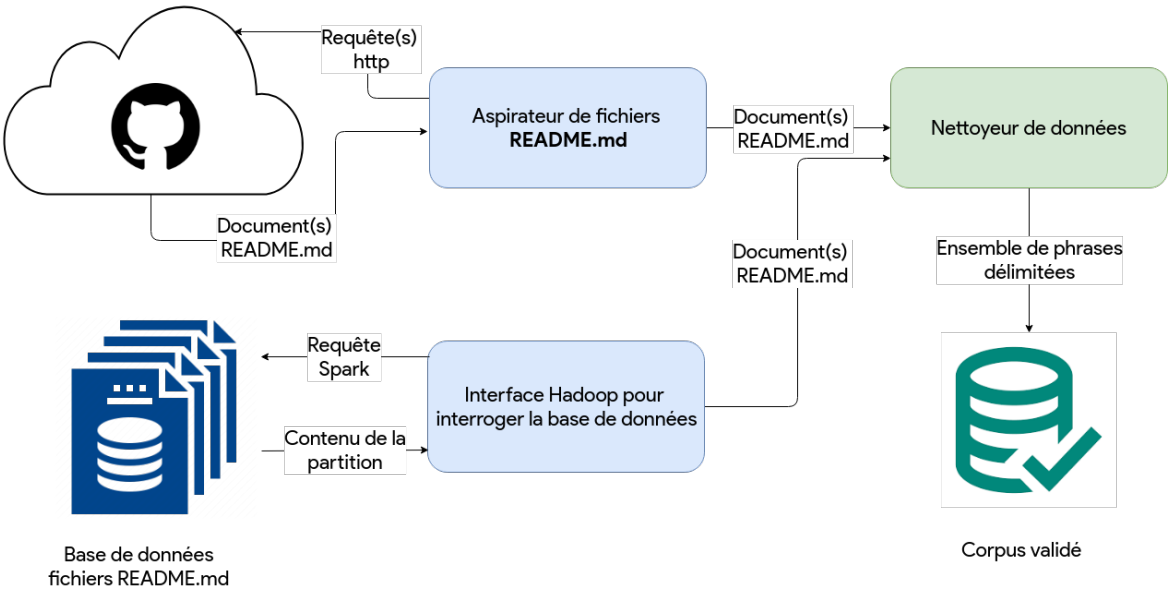

Figure 1.4 – *Processus de génération du corpus pour le modèle de langue*

- L'acquisition des données dans leur format brut **.md** (markdown) se fait de deux manières :
  - Depuis le site officiel de GitHub en faisant des requêtes http au serveur en suivant le patron suivant pour les urls :

```
'http://raw.githubusercontent.com/'+NOM_DÉPOT+'/master/README.md'
```

La liste des noms de dépôts est disponible dans un fichier[8] en free open acces (accès ouvert et libre) au format **.csv** dont les colonnes sont *Nom_Utilisateur* et *Nom_Dépot*

- En lisant une base de 16 millions de fichiers différents dont la taille totale atteint 4.5 Go

- Les deux sources de données envoient ensuite les fichiers récoltés au nettoyeur de fichiers pour en extraire seulement les parties qui ont du sens dans le langage naturel (paragraphes, titres, instructions, etc.)
- Le corpus final est ensuite construit à partir des paragraphes extraits à l'étape précédente après les avoir segmentés en phrases, à l'aide d'un modèle de segmentation prédéfini, donnant un ensemble de phrases dans le format suivant

```
<s>select and click edit</s>
<s>browse to demo on your web browser</s>
<s>you can specify these values in a file that file must be home</s>
...
```

1.4 Module de compréhension automatique du langage naturel

Second module du système, le module de compréhension automatique du langage naturel (Natural Langage Understanding, NLU) dont le rôle est de faire office de couche d'abstraction entre la requête de l'utilisateur, formulée en un langage naturel, et le fonctionnement interne du système qui communique à travers un langage plus formel. On parle ici de la construction d'une représentation sémantique de la requête. Pour ce faire nous avons opté pour l'approche par apprentissage automatique, compte tenu des bons résultats obtenus par certaines architectures (Goo et al., 2018 ; Liu and Lane, 2016) et cela malgré le petit volume des données d'apprentissage. Cette option nous a paru plus abordable que la construction d'un analyseur basé règles, souvent assez rigide et dont l'exhaustivité n'est pas évidente à obtenir.

8. https://data.world/vmarkovtsev/github-readme-files/file/top_broken.tsv

1.4.1 Architecture du module

Comme précédemment cité (voir la section 1.2.2), le module NLU possède une architecture en pipeline qui reçoit en entrée le texte brut de la requête. Sa codification varie selon les approches que nous avons explorées et qui seront plus explicitées dans le chapitre suivant "Réalisation et résultats". Pour mieux capturer l'aspect sémantique des mots dans le texte, nous avons décidé d'utiliser le modèle Word2Vec pré-entraîné par Google (entraîné sur 100 milliard de mots) pour produire un vecteur de taille fixe pour chaque mot. Pour encoder l'information syntaxique de la requête, nous avons concaténé au vecteur de prolongement de chaque mot de la requête (Word Embedding Vector) le vecteur codifiant son étiquette morphosyntaxique. Après avoir codifié la séquence de mots, elle est envoyée aux modèles de classification d'intentions et d'extraction d'entités [9], qui sont en fait un seul modèle joint dont l'architecture est détaillée dans la section 1.4.2. Ces deux informations sont ensuite décodées et passées au constructeur de trame sémantique qui structurera ces dernières en une seule entité sémantique. Un exemple d'une requête et sa trame sémantique associée est donné ci-dessous :

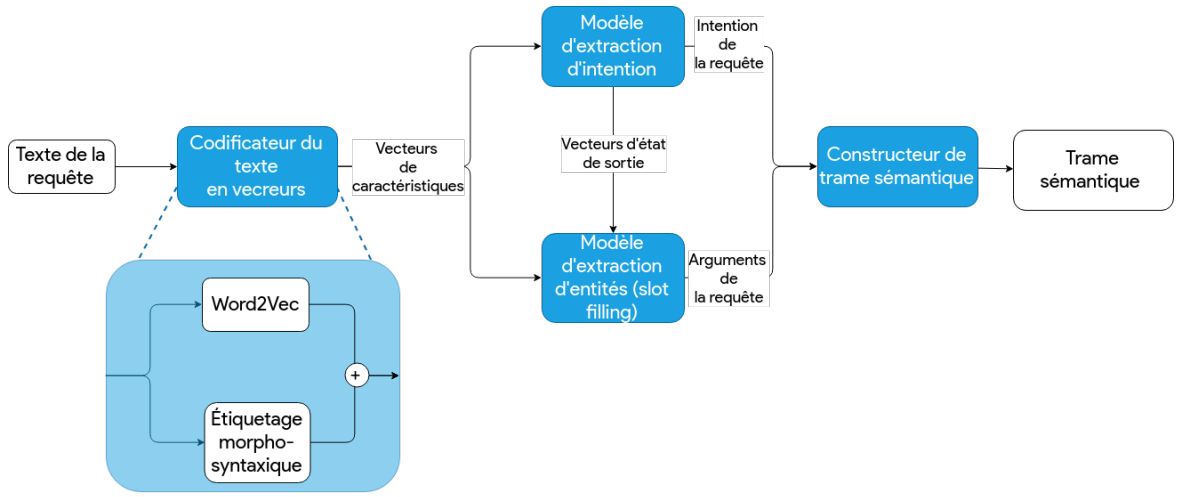

Figure 1.5 – *Architecture du module de compréhension automatique du langage naturel (NLU)*

```
{
  "query" :"could you please open the file test.py",
  "intent" :"open_file_desire",
  "entities" :[
    {
      "name" :"file_name",
      "value" :"test.py",
      "end-start" :[31,37]
    }
  ]
}
```

9. Par entité, nous entendons les arguments de l'intention.

Une trame sémantique se compose de trois parties :

- **Partie requête** : le texte de la requête énoncée par l'utilisateur.
- **Partie intentions** : l'intention extraite, à partir de la requête, se trouve dans l'entrée "$intent$" de la trame.
- **Partie arguments** : aussi appelés "entités du domaine". Ces arguments sont extraits du texte de la requête. L'entrée $entities$ contient une liste d'arguments, où chaque élément de la liste renseigne sur le nom, la valeur et l'emplacement d'une entité.

1.4.2 Modèle(s) utilisé(s)

Comme vu dans le chapitre précédent (voir la section **??**), l'architecture adoptée est une architecture mono-entrée/multi-sorties dont l'entrée est une séquence de mots codifiés et les sorties sont une séquence d'étiquettes ainsi qu'une classe associée au texte. Nous pouvons distinguer les deux parties qui sont l'encodage et le décodage de la séquence.

L'encodage sert à la fois à l'attribution de la classe (l'intention) et à l'initialisation de la séquence de décodage (pour l'attribution d'une étiquette à chaque mot). Il se fait en utilisant un réseau de neurones récurrent de type Bi-LSTM (Bidirectionall Long Short Term Memory) pour mieux capturer le contexte droit (respectivement gauche) de chaque entrée selon le sens de traitement des données dans le réseau Bi-LSTM. Le dernier vecteur en sortie est ensuite utilisé comme vecteur d'entrée pour un réseau de neurones Fully Connected (Complètement connecté) dont la dernière couche est une couche de prédiction sur une distribution de probabilités des intentions possibles.

Le décodeur est aussi un réseau de neurones récurent de type Bi-LSTM. Il prend en entrée le vecteur précédemment retourné par l'encodeur. Ainsi, à chaque étape de l'inférence une étiquette est produite en sortie pour chaque position du texte en entrée (les longueurs des séquences d'entrée et de sortie sont donc identiques). Ceci est réalisé en utilisant un autre réseau de neurones Fully Connected sur chaque vecteur d'état de sortie des cellules LSTM du décodeur (voir la figure **??**).

1.4.3 Les données d'apprentissage

Ne disposant pas d'un ensemble d'apprentissage pré-existant pour les intentions que nous avons développées, nous avons tenté d'en construire un nous-mêmes en l'enrichissant avec quelque modifications. Dans (Bocklisch et al., 2017), il a été noté que pour une tâche assez simple, comme pour notre cas, l'exploration des fichiers dans un premier temps, il n'est pas nécessaire de disposer d'une grande quantité de données (une cinquantaine d'exemples par intention approximativement) si les exemples ne sont pas facilement confondus, et surtout si l'espace des possibilités pour les requête est assez réduit et peut facilement être explicité. En jouant sur l'ordre des mots, nous avons pu générer pour les 15 intentions, 4157 patrons d'exemples au total dont 870 sont dépourvus d'arguments. Un patron d'exemple est une structure contenant des placeholders (compartiments) pouvant être remplis avec des valeurs générées programmatiquement. Par exemple :

```
delete the {file_name:} file under {parent_directory:}
```

Ces placeholders servent à la fois à générer plus d'exemples mais aussi à étiqueter le texte en choisissant les valeurs de ces variables comme valeur de l'étiquette. Un exemple d'une entrée de l'ensemble d'apprentissage avant affectation des variables est le suivant :

```
{
  "id": 6,
  "text": "I want to open the {file_name:} folder",
  "intent": "open_file_desire"
},
```

Pour remplir l'ensemble des placeholders, nous commençons d'abord par scanner le répertoire de la machine avec une profondeur maximum égale à 5, le terme profondeur désigne les niveaux de répertoires et sous-répertoires. Nous avons aussi ajouté une liste[10] des noms de fichiers et répertoires les plus populaires. Les noms des répertoires sont ensuite nettoyés à l'aide d'expressions régulières et transformés en un format universel établi à l'avance **nom_du_fichier** en choisissant "_" (le tiret du 8) comme séparateur. En bouclant sur ces noms de répertoires nous pourrons donc construire plusieurs exemples comme une entrée dans un dictionnaire dont le format est le suivant :

```
{
  'id': 79372,
  'intent': 'delete_file_desire',
  'postags': ['NN', 'VB', 'DT', 'NN', 'VBN', 'NN', 'NNS'],
  'text': 'please remove the file named platform notifications',
  'tags': 'NUL NUL NUL NUL NUL ALTER.file_name ALTER.file_name'
}
```

Une entrée est divisée en cinq champs :

- **id** : un entier qui sert d'identificateur pour l'instance.
- **intent** : l'intention (non encore codifiée) attribuée à l'instance.
- **tags** : les étiquettes de chaque mots de la requête. Une étiquette peut être soit NUL (ce n'est pas un argument) ou bien le nom de l'entité (argument) que représente le mot à la position étiquetée. La liste complète des intentions avec leurs arguments se trouve dans le tableau 2.2 du chapitre "Réalisation et résultats".
- **postags** : la liste des étiquettes morphosyntaxiques de chaque mots de la requête. L'ensemble des étiquettes utilisées est celui du Penn Treebank (Marcus et al., 1994). Dans l'exemple ci-dessus, NN signifie "Nom au singulier". NNS signifie "Nom au pluriel". VB et VBD signifient respectivement "Verbe à l'infinitif" et "Verbe au passé simple". Enfin, DT signifie "Déterminant".

10. https://github.com/xmendez/wfuzz/blob/master/wordlist/general/common.txt

- **text** : le texte de la requête nettoyé et dont les mots sont séparés uniquement par un espace.

1.5 Module de gestion du dialogue

Le but de ce module est de décider quelle action prendre à chaque instant du dialogue. De prime abord, nous allons présenter l'architecture globale de ce module notamment la représentation des informations reçues et la politique d'actions. Ensuite, nous allons détailler la conception de chaque partie.

1.5.1 Architecture du module

Comme nous l'avons déjà vu, l'architecture typique des gestionnaires de dialogue se compose de deux parties principales :

- Un module qui suit l'état du dialogue : Pour gérer le dialogue avec l'utilisateur, le gestionnaire doit représenter l'état du dialogue de façon à pouvoir répondre aux actions de l'utilisateur. Ce module sert à suivre cet état après chaque étape du dialogue.
- Une politique d'actions : Celle-ci détermine l'action à prendre à partir d'un état donné.

1.5.1.1 État du dialogue

Avant de détailler les deux modules du gestionnaire, il est nécessaire d'introduire une représentation de l'état du dialogue. Classiquement, les trames sémantiques ont été utilisées dans ce but (voir **??**). Le suivi d'état se fait, dans ce cas, en gardant trace des emplacements remplis durant le dialogue. Nous avons opté pour une représentation plus riche par l'utilisation des graphes de connaissances, qui sont une forme de représentation où les connaissances sont décrites sous forme d'un graphe orienté étiqueté. Des travaux ont déjà utilisé ce type de connaissances (Stoyanchev and Johnston, 2018) et des ontologies (Wessel et al., 2019) pour représenter l'état du système de dialogue. À partir de ce dernier, une base de règles décide quelle action prendre directement du graphe. L'avantage d'utiliser des graphes de connaissances par rapport à l'utilisation des trames sémantiques apparaît dans leur flexibilité et leur dynamisme. En effet, pour une tâche comme la navigation dans les fichiers, l'état de l'arborescence des fichiers est sujet à des changements fréquents : ajout, suppression, modification, etc. Il est difficile de faire une représentation de l'information dans ce cas avec de simples emplacements à remplir.

1.5.1.2 Suivi de l'état du dialogue

Le rôle du premier module du gestionnaire de dialogue est de mettre à jour l'état du système au cours du dialogue. Il reçoit l'action de l'utilisateur ou du gestionnaire et il produit un nouvel état. Dans notre cas, le module NLU produit toujours une trame sémantique

contenant l'intention de l'utilisateur ainsi que ses paramètres. C'est alors le travail du traqueur d'état d'injecter le résultat du module NLU dans le graphe de connaissances. Ceci consiste à transformer la trame sémantique en un graphe qui est ensuite ajouté au graphe d'états. Plus de détails seront donnés dans la section 1.5.2 où nous construirons une ontologie pour définir un vocabulaire de dialogue et comment elle peut être utilisée pour passer du résultat du module NLU en un graphe de connaissances.

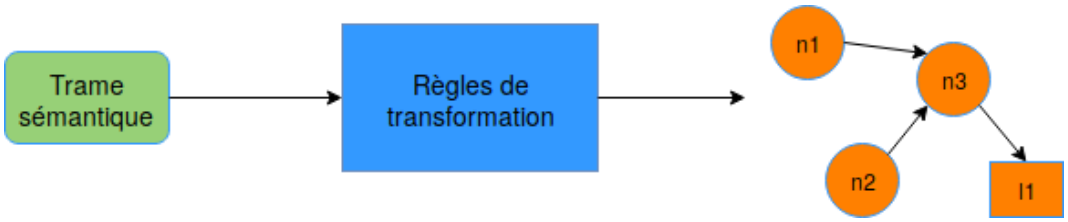

Figure 1.6 – *Schéma de transformation de trame sémantique en graphe*

1.5.1.3 La politique d'actions

La politique d'actions peut être écrite manuellement, apprise à partir d'un corpus ou à l'aide de l'apprentissage par renforcement. Dans ce dernier cas, un agent doit interagir avec un utilisateur qui évalue ses performances afin qu'il puisse apprendre. Étant donné que l'apprentissage par renforcement nécessite un nombre important d'interactions, il est primordial d'utiliser un simulateur d'utilisateur. Ce dernier peut être à base de règles, ou un modèle statistique extrait à partir d'un corpus de dialogue.

Dans les trois cas de figure, il est difficile de réaliser un modèle varié et qui peut accomplir plusieurs tâches. D'un côté, un corpus contenant des dialogues sur toutes les tâches possibles est difficile à acquérir, si ces derniers sont nombreux et spécifiques à une application précise. De l'autre côté, écrire les règles d'un système de dialogue ou d'un simulateur d'utilisateur s'avère compliqué et nécessite un travail manuel énorme pour gérer toutes les tâches possibles.

Pour pallier à cela, nous proposons d'utiliser une architecture multi-agents hiérarchique dans laquelle, les agents feuilles sont des agents qui peuvent répondre à une tâche ou une sous tâche bien précise, tandis que les agents parents sélectionnent l'agent fils capable de répondre à l'intention de l'utilisateur.

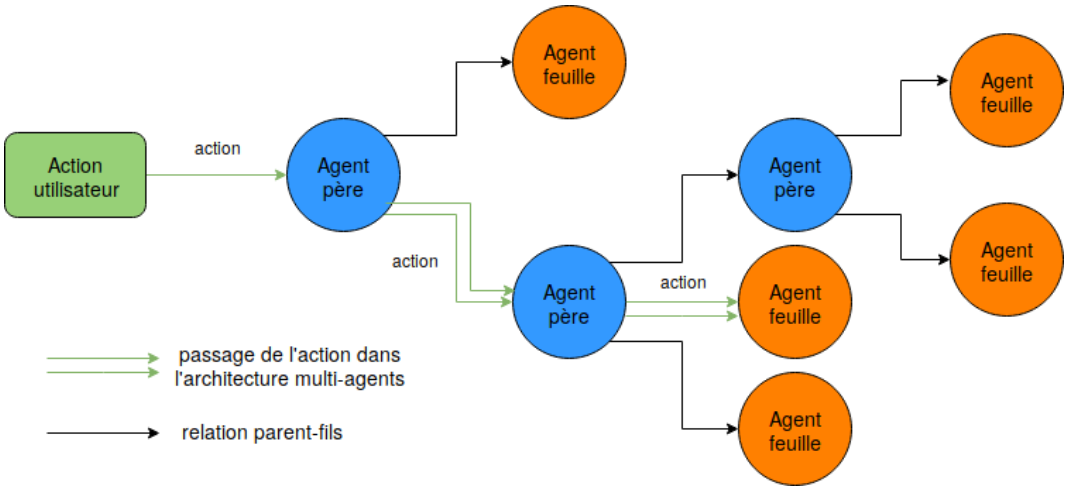

Figure 1.7 – *Schéma de l'architecture multi-agents pour la gestion du dialogue*

L'avantage de l'architecture multi-agents hiérarchique est de permettre la division du problème en plusieurs sous-problèmes indépendants. En effet, un simulateur d'utilisateur ou un corpus qui est destiné pour une seule tâche est considérablement plus abordable à créer. De plus, cette architecture permet un développement incrémental dans le sens où elle facilite l'addition d'une nouvelle tâche pour l'assistant ; il suffit d'ajouter des agents capables de traiter cette nouvelle tâche à l'architecture. Cependant, un travail supplémentaire s'avère nécessaire qui est celui des agents parents. Ce travail est relativement simple ; il suffit de faire un apprentissage supervisé des agents parents avec les simulateurs d'utilisateurs des agents fils. À tour de rôle et avec des probabilités de transition entre les simulateurs d'utilisateurs, ces derniers communiquent avec l'agent parent. Comme on connaît pour chaque simulateur l'agent fils qui lui correspond, il est donc possible de faire un apprentissage supervisé où les entrées sont les actions des simulateurs et l'état du système, tandis que la sortie est l'agent fils qui peut répondre à l'action.

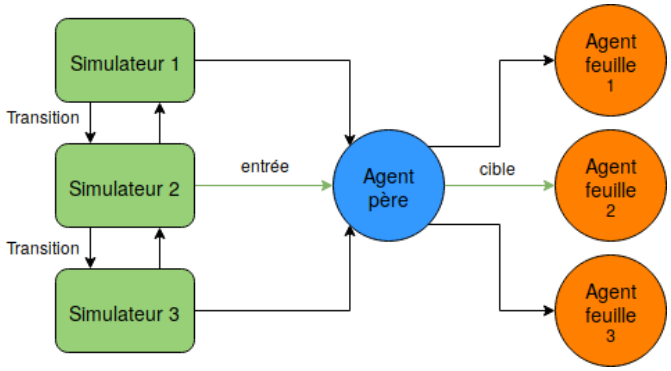

Figure 1.8 – *Schéma représentant l'apprentissage des agents parents avec les simulateurs des agents feuilles*

Pour résumer l'architecture globale du gestionnaire de dialogue, lorsqu'une nouvelle action utilisateur arrive au système, le traqueur d'état la reçoit et met à jour l'état du système en transformant l'action en un graphe de connaissances pour l'ajouter au graphe

d'état. Ce nouveau graphe d'état ainsi que la dernière action reçue sont transmis à une architecture multi-agents hiérarchique qui va décider quelle action le système de dialogue doit prendre.

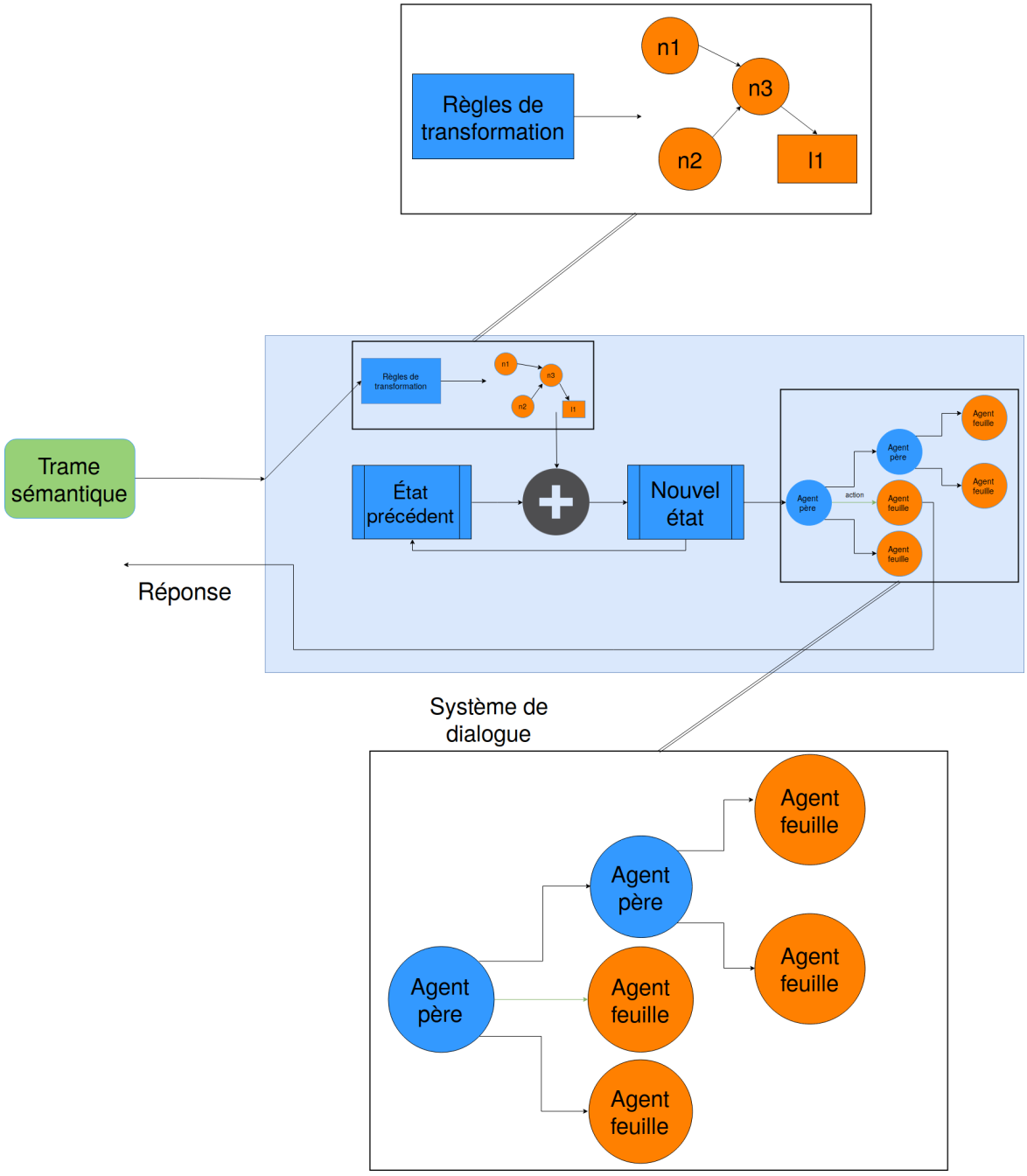

Figure 1.9 – *Schéma global du gestionnaire de dialogue*

1.5.2 Les ontologies du système

Une ontologie peut être vue comme une représentation des concepts et des relations d'un domaine donné. Elle définit un vocabulaire consensuel pour ce domaine afin de permettre

aux programmes intelligents de comprendre et de communiquer sur des données reliées à ce domaine.

Nous définissons une ontologie de dialogue ainsi que des ontologies pour chaque tâche réalisable par notre assistant. Ceci permettra à notre gestionnaire de comprendre le dialogue et les tâches qu'il peut réaliser.

1.5.2.1 Ontologie de dialogue

Nous définissons en premier une ontologie de dialogue qui contient des concepts qui peuvent aider un assistant d'ordinateur à gérer son dialogue.

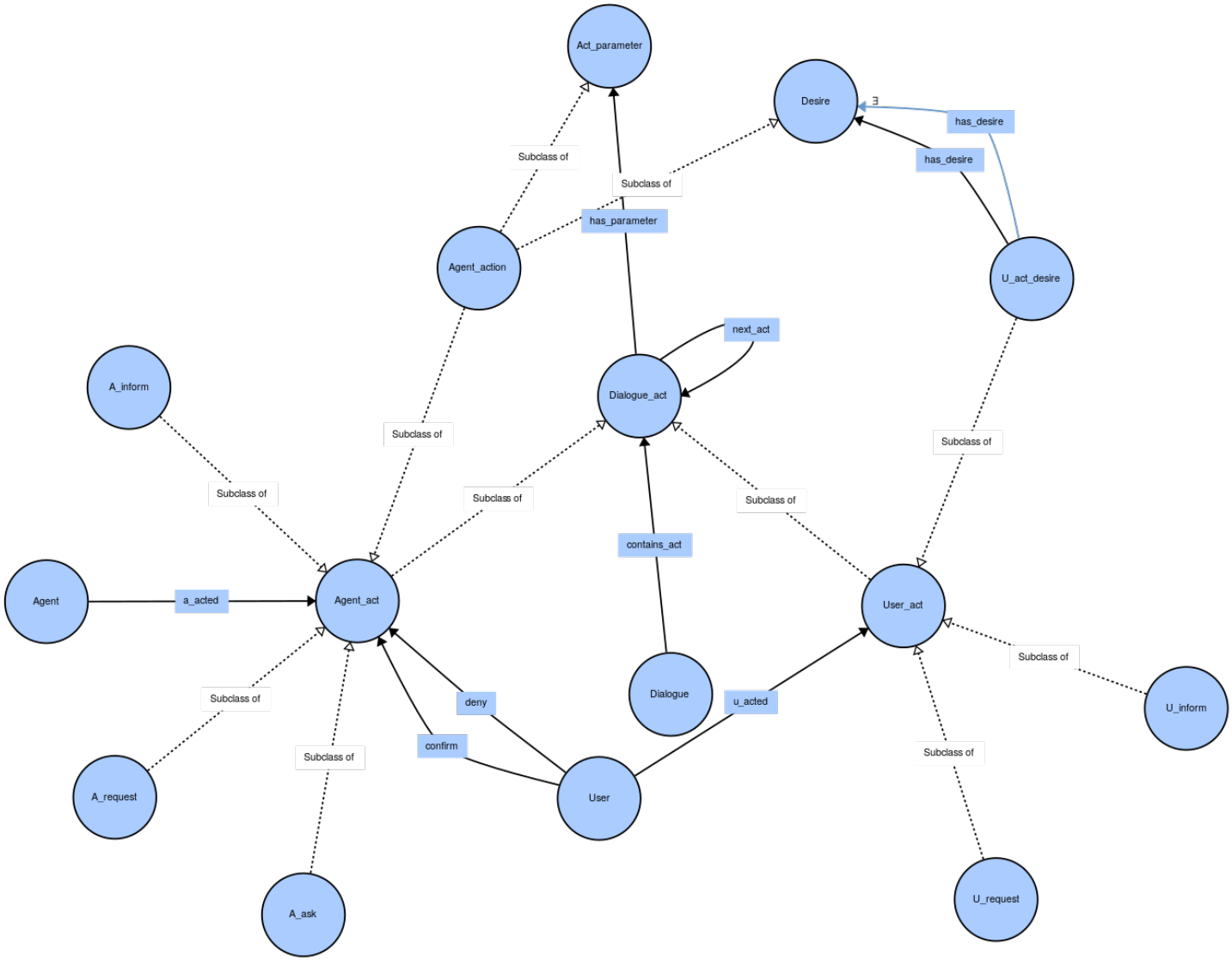

Figure 1.10 – *Graphe de l'ontologie de dialogue*

Principalement, l'ontologie se compose de six classes mères :

- $Agent$ et $User$: ce sont les classes qui représentent l'utilisateur et l'agent qui participent au dialogue.

- $Dialogue$: l'agent et l'utilisateur participent à un dialogue qui contient les actions des deux participants.
- $Dialogue\_act$: il s'agit de la classe qui représente une action du dialogue ; elle a deux sous-classes $Agent\_act$ et $User\_act$ qui représentent les actions de l'agent et de l'utilisateur respectivement.
- $Act\_parameter$: C'est la classe mère des paramètres que peuvent prendre les actions de dialogue. Par exemple, l'action d'informer peut avoir comme paramètre le nom d'un fichier.
- $Desire$: C'est la classe mère des actions de l'agent que l'utilisateur peut demander. Par exemple, il peut demander l'ouverture d'un fichier donné.

Le reste des classes sont des classes filles qui détaillent plus les concepts du dialogue agent-utilisateur.

À l'arrivée d'une nouvelle action, le traqueur d'état va créer le graphe correspondant. Un exemple abstrait de cela est représenté dans la figure 1.11. Une nouvelle action utilisateur est créée ainsi que ses paramètres et les relations entre eux.

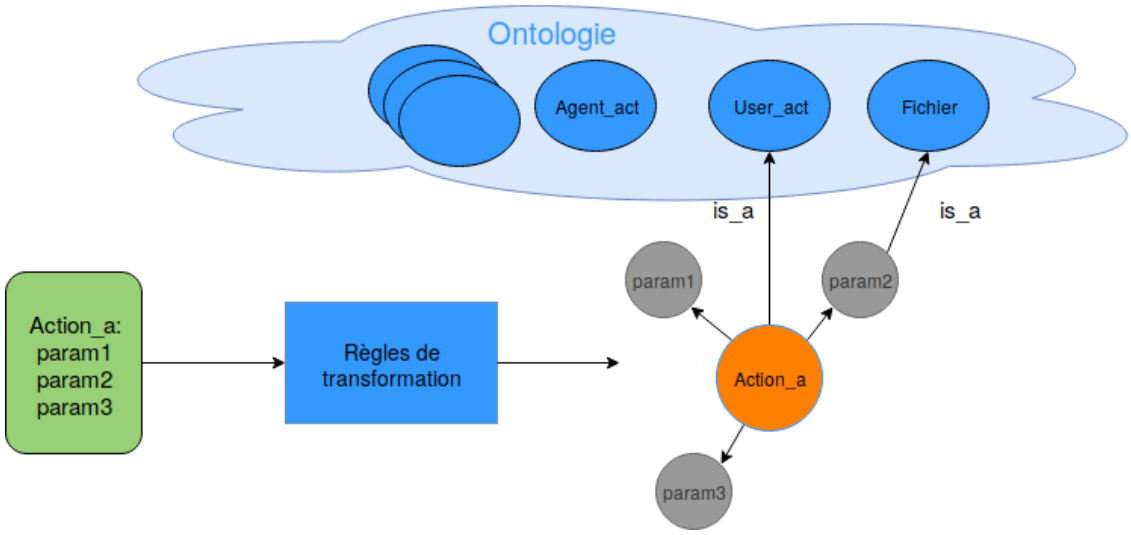

Figure 1.11 – *Schéma de transformation d'une action en graphe*

Ontologie pour l'exploration de fichiers

Un exemple d'ontologie pour la compréhension d'une tâche réalisable par l'assistant est celle de l'exploration de fichiers.

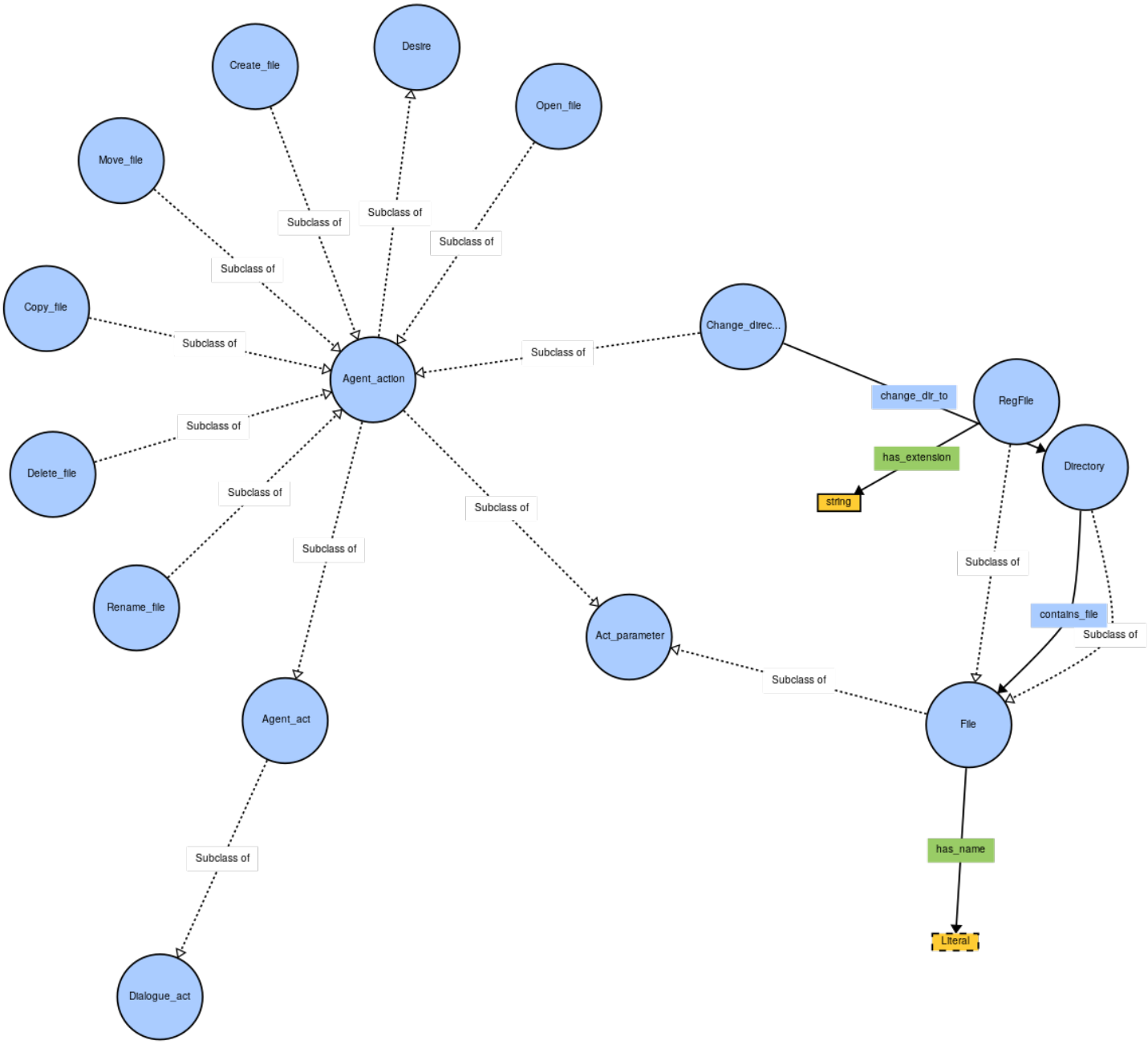

Figure 1.12 – *Graphe de l'ontologie de l'exploration de fichiers*

L'ontologie contient essentiellement :

- Des actions sur les fichiers : Créer un fichier, supprimer un fichier, changer de répertoire, etc. Ces actions sont des sous-classes de la classe $Agent\_act$ vue précédemment ainsi que les classes $Act\_parameter$ et $Desire$. Ceci veut dire, d'un côté, que ces actions peuvent être des paramètres d'autres actions, comme "demander à l'utilisateur s'il veut que l'assistant réalise une action donnée" et d'un autre côté, que l'utilisateur peut demander à l'assistant d'effectuer une de ces actions.
- Les concepts qui sont liés à l'exploration de fichiers sont des sous-classes de $Act\_parameter$ étant donné qu'ils peuvent être des paramètres d'actions. Par exemple, l'action d'ouvrir un fichier a comme paramètre un fichier.

- Des relations entre concepts sont aussi définies. Par exemple, un répertoire peut contenir des fichiers, ou une action de changement de répertoire doit avoir comme paramètre un répertoire cible.

La figure 1.13 représente l'arrivée d'une nouvelle action : "créer un fichier nommé 'travail'". L'action de l'utilisateur est donc représentée par un nouveau nœud de type U_act_desire qui désigne une action utilisateur qui demande une action de l'assistant. Cette dernière a comme paramètre un nœud de type $Create\_file$ qui est l'action de l'agent que l'utilisateur veut réaliser. Cette action à son tour a des paramètres comme, dans ce cas, le fichier qu'on veut créer.

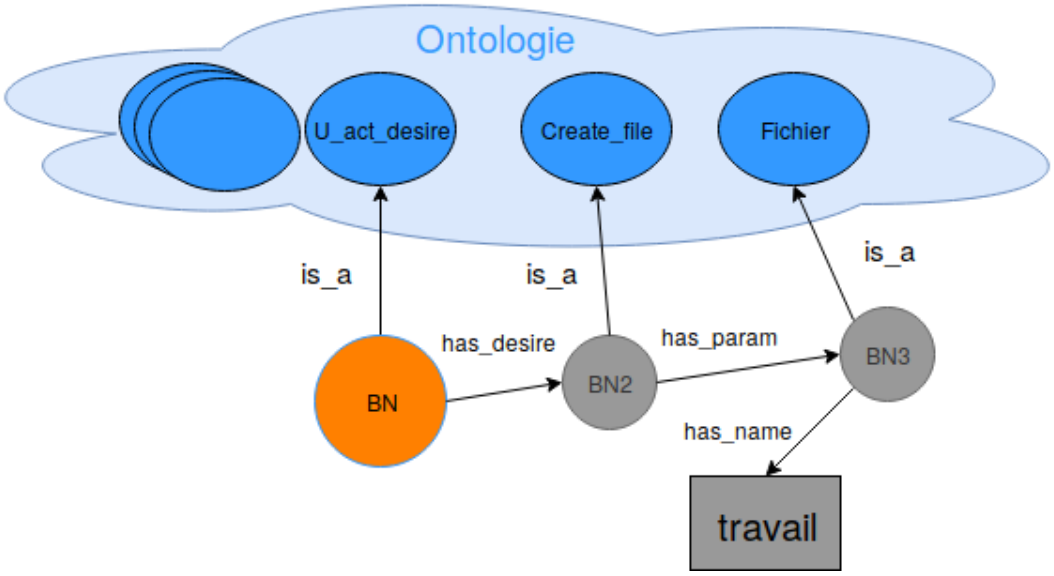

Figure 1.13 – *Schéma de transformation d'une action de demande de création de fichier en graphe*

Similairement, les graphes des autres actions sont créées en se basant sur des règles de transformation.

1.5.3 Les simulateurs d'utilisateurs

Plusieurs méthodes peuvent être utilisées pour la création de simulateurs d'utilisateurs (voir **??**). Les simulateurs basés sur des méthodes d'apprentissage sont les plus robustes. Cependant, ils nécessitent un nombre important de données. L'alternative, c'est d'utiliser des simulateurs basés règles. Nous nous sommes inspirés des simulateurs basés agenda (Schatzmann et al., 2007) qui sont des variantes des simulateurs basés règles pour créer nos propres simulateurs. Leur fonctionnement est simple. Ils commencent par générer un but. Pour y arriver, un agenda, qui contient les informations que doit convoyer le simulateur ainsi que les informations qu'il doit recevoir, est créé. Les actions sont sélectionnées en suivant des probabilités conditionnelles sur l'état de l'agenda. Enfin, les récompenses sont en fonction des informations reçues de l'agent.

Simulateur pour l'exploration de fichiers

L'exploration de fichiers ne dépend pas de simples informations à transmettre et d'autres à recevoir comme dans les cas d'envoyer un e-mail, chercher une information sur internet ou bien lancer de la musique. Il s'agit d'une tâche dynamique dont la situation de départ est variante. Pareillement, le nombre d'actions change d'un état à un autre. Par exemple, le nombre de fichiers qu'on peut supprimer ou le nombre de répertoires auxquels on peut accéder ne sont pas fixes.

Le simulateur commence d'abord par générer une arborescence aléatoire qui représente la situation initiale du système. Ensuite, il duplique cette arborescence en y introduisant des modifications pour générer une arborescence but. Enfin, le simulateur essaye de guider l'agent pour arriver au but en utilisant les actions utilisateurs possibles.

En plus des actions de création et suppression de fichiers ainsi que les changements de répertoires qui peuvent guider l'agent au but, d'autres sous-buts peuvent être créés suivant une distribution de probabilité comme copier ou changer l'emplacement d'un fichier, renommer un fichier, ouvrir un fichier, etc. Dans ce cas, le simulateur donne la priorité aux sous-buts avant de reprendre les actions menant au but final.

L'algorithme suivi par le simulateur est le suivant :

Algorithm 1 Algorithme simulateur

```
1 : procedure Step(entrées : action_agent, max_tour, tour ; sorties : recompense, fin, succes, tour)
2 :     tour ← tour + 1
3 :     if tour > max_tour then
4 :         fin ← true
5 :         succes ← echec
6 :         reponse_utilisateur ← réponse_fin()
7 :     else
8 :         succes ← maj_état(action_agent)
9 :         if succes then
10 :            fin ← true
11 :            reponse_utilisateur ← réponse_fin()
12 :        else
13 :            reponse_utilisateur ← décider_action(action_agent)
14 :    recompense ←calculer_récompense(action_agent, succes)
15 : return reponse_utilisateur
```

- **ligne 1 :** en entrée de la procédure :
  - $action\_agent$: l'action pour laquelle l'agent attend une réponse du simulateur.
  - max_tour : le nombre maximum d'échanges agent-simulateur.
  - $tour$: le numéro de l'échange actuel.

  en sortie de la procédure :

- $recompense$: la récompense que donne le simulateur suite à la dernière action de l'agent.
- fin : un booléen qui détermine si la discussion est terminée.
- $succes$: un booléen qui détermine si l'agent est arrivé à un succès.
- $tour$: la variable $tour$ est modifiée à l'intérieur de la procédure ; elle doit être donc mise en sortie aussi.

- **lignes 3-6 :** si le nombre de tours passés est supérieur au nombre maximal de tours autorisés, le simulateur retourne un état d'échec et une action indiquant à l'agent la fin de la discussion.
- **ligne 8 :** en mettant à jour l'état du simulateur, ce dernier peut savoir si l'action de l'agent permet d'arriver au but du simulateur.
- **lignes 9-11 :** si l'agent arrive au but, le simulateur le lui indique avec une action de fin et un état de succès.
- **ligne 13 :** sinon, s'il n'est toujours pas arrivé au but, le simulateur décide quelle action prendre selon l'action de l'agent.
  Le diagramme 1.14 résume cette décision.

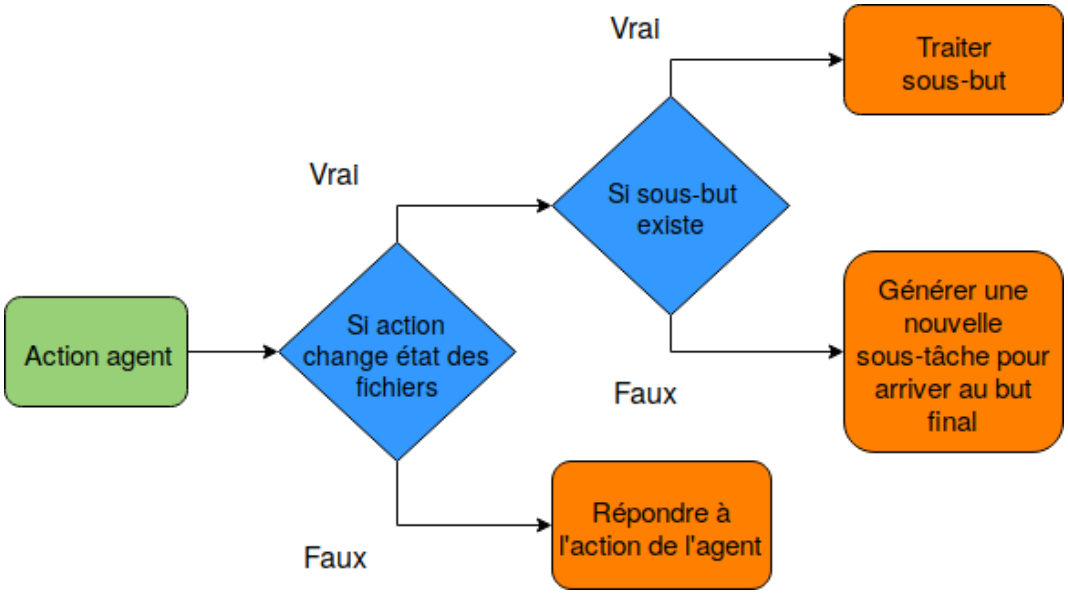

Figure 1.14 – *Diagramme de décision de l'action à prendre*

En ce qui concerne la fonction de récompense, celle-ci est évaluée à partir des changements effectués sur l'état de l'utilisateur. Il existe deux types d'actions affectant l'état :

- Les actions qui affectent l'arborescence de fichiers. Soit elles permettent d'arriver à un sous-but ou bien elles changent la similarité entre l'arborescence courante et l'arborescence du but final.
- Le reste des actions sont des échanges agent-utilisateur pour transmettre et recevoir des informations. Par exemple, l'agent peut demander le nom du répertoire parent d'un fichier.

D'où, les évènements considérés dans la fonction de récompense sont :

- La réalisation d'un sous-but.
- Le changement de similarité : soit en l'améliorant ou en la diminuant.
- La réalisation du but final.
- Les échanges d'informations : Nous distinguons de ces échanges la confirmation d'une question de l'agent qui sera traitée séparément.

Le tableau 1.1 associe les changements avec leurs récompenses :

Événements	**Récompenses**
Similarité améliorée	2
Succès	2
Sous-but réalisé	2
Similarité diminuée	-3
Confirmation d'une question	0
Autre	-1

Table 1.1 – *Tableau des récompenses*

- Nous avons choisi de donner une récompense unitaire négative pour chaque échange qui n'affecte pas l'arborescence de fichiers. La récompense négative permet d'éviter que le dialogue dure longtemps. L'agent essaye donc de minimiser ce type d'actions afin de ne pas cumuler les récompense négatives.
- **Confirmation d'une question :** la seule exception à cela c'est quand l'utilisateur confirme une action à l'agent. La récompense est nulle (= 0) pour que l'agent puisse demander une confirmation quand il n'est pas sûr de ce qu'il doit faire sans diminuer le cumul des récompenses reçues.
- **similarité améliorée :** On calcule la similarité entre l'arborescence courante et l'arborescence but. La valeur de la similarité est égale à n_sim/n_diff avec :
  - n_sim : nombre de fichiers qui existent dans les deux arborescences.
  - n_diff : nombre de fichiers qui n'existent que dans une des deux arborescences.

  Cette valeur change soit en ajoutant ou en supprimant un fichier. Dans le cas où un fichier est ajouté, si celui-ci existe dans l'arborescence but, n_sim est incrémentée et n_diff reste fixe ; la valeur de la similarité augmente. Sinon, s'il n'existe pas dans l'arborescence but, n_diff est incrémentée et n_sim reste fixe ; la valeur de la similarité diminue. Alternativement, si un fichier est supprimé, et celui-ci n'existe pas dans l'arborescence but, n_diff est décrémentée et n_sim reste fixe ; la valeur de la similarité augmente. Sinon s'il existe dans l'arborescence but, n_sim est décrémentée et n_diff reste fixe ; dans ce cas la valeur de la similarité diminue. En conséquence, si cette valeur s'améliore, on donne à l'agent une récompense positive supérieure en valeur absolue à celle donnée dans les échanges d'informations.
- **Succès :** c'est-à-dire, le but final du simulateur est réalisé. Pour arriver à cet événement, soit l'agent ajoute un fichier à l'arborescence ou bien il le supprime. Les autres actions, comme renommer un fichier ou lui changer d'emplacement, font partie des

sous-buts. L'agent ne fait donc qu'améliorer la similarité en ajoutant ou en supprimant un fichier ; c'est pourquoi cet évènement et celui qui l'a précédé ont la même valeur de récompense.

- **Sous-but réalisé :** c'est la récompense donnée quand l'agent arrive au sous-but du simulateur. Par exemple, en ouvrant un fichier, en le renommant ou en le copiant dans un autre répertoire, etc. Elle est égale à celle donnée dans le cas où la similarité est améliorée car l'exécution de l'action menant à ce sous-but ne nécessite pas plus d'efforts ou d'échanges que quand le simulateur ajoute ou supprime un fichier.
- **Similarité diminuée :** la récompense est dans ce cas négative et supérieure en valeur absolue à celle que l'agent obtient lorsqu'il améliore la similarité. Ce choix a pour but d'éviter que l'agent boucle sur des actions dont la somme des récompenses est supérieure ou égale à zero. Par exemple, il peut créer ensuite supprimer le même fichier. Si la somme de ces deux actions est supérieure ou égale à zéro, l'agent peut boucler sur ces actions indéfiniment sans pour autant recevoir des récompenses négatives.

Pour résumer le fonctionnement du simulateur, à l'arrivée d'une nouvelle action agent, celle-ci met à jour l'état du simulateur. L'état du simulateur se compose de deux parties : un simulateur d'arborescence de fichiers qui simule l'état de l'arborescence courante, et des variables d'état qui contiennent d'autres informations comme l'état des sous-buts, le fichier en cours de traitement, les informations reçues de l'agent, le répertoire courant, etc. Après la mise à jour de l'état, si l'action de l'agent nécessite une réponse immédiate, comme la demande d'une information ou la permission d'exécuter une action, celle-ci est traitée directement, sinon le simulateur initie le traitement d'une nouvelle sous-tâche. C'est-à-dire, si un but intermédiaire existe, une action qui le traite est générée ; sinon, une action qui traite le but final est générée.

1.5.4 Modèles d'apprentissage

Comme on l'a déjà cité dans le chapitre précédent, il existe plusieurs algorithmes d'apprentissage par renforcement comme Q-Learning ou State-Action- Reward-State-Action (SARSA) (Rummery and Niranjan, 1994). Cependant, ces algorithmes, en essayant d'estimer la fonction Q de récompense, traitent le problème comme un tableau état/action et essayent d'estimer pour chaque état et action la récompense résultante. Ceci implique que ces algorithmes ne peuvent pas estimer la fonction de récompense pour des états qu'ils n'ont pas vus pendant l'apprentissage. Pour pallier à ce problème, Deep Q Learning (DQL)(Mnih et al., 2015) utilise un réseau de neurones comme estimateur de la fonction Q, ce qui lui permet d'avoir une notion de similarité entre les états. Ainsi, il peut estimer la récompense pour des états jamais vus auparavant.

Encodeur de graphe

La flexibilité des graphes les rend difficiles à introduire dans un réseau de neurones vu que ce dernier n'accepte que des entrées de tailles fixes. Des méthodes ont été utilisées

pour introduire les graphes dans des réseaux de neurones notamment les convolutions sur les graphes avec Graph Convolution Networks (GCN)(Kipf and Welling, 2017) qui s'avèrent être des variantes des Gated Graph Neural Networks (GGNN)(Li et al., 2016). Ces derniers utilisent des réseaux de neurones récurrents (RNNs) entre chaque deux nœuds reliés par un arc pour transférer l'information d'un nœud à un autre. C'est-à-dire que le RNN prend en entrée les vecteurs encodant un nœud et un de ses arc pour essayer de propager l'information contenue dans ces vecteurs au nœud destination. Ce dernier met à jour le vecteur l'encodant en sommant les résultats du RNN provenant des différents nœuds voisins. Ce qui résulte en des vecteurs ayant un encodage qui tient en compte le voisinage du noeud. Cette étape est répétée k fois pour que chaque nœud ait des informations sur les nœuds qui sont à un maximum de k pas de distance, avec k un paramètre empirique. Enfin, la somme des vecteurs d'états des différents nœuds est effectuée pour produire un vecteur fixe encodant tout le graphe qui peut être relié au reste du réseau de neurones pour l'apprentissage.

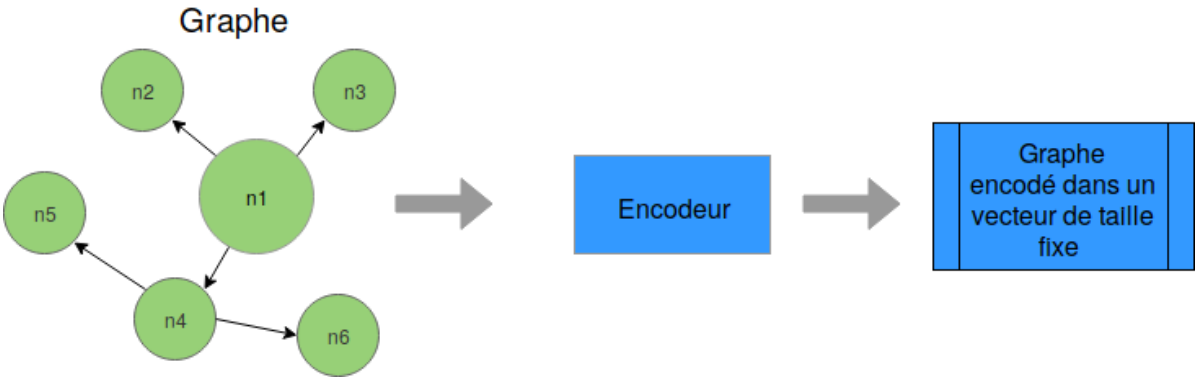

Figure 1.15 – *Schéma représentant un encodeur de graphe*

Ces méthodes nécessitent tout le graphe pour l'encoder. Cependant, dans notre cas, après chaque action, le graphe augmente de taille ce qui nécessite de refaire l'encodage à partir du début. Nous proposons de traiter le graphe comme une séquence de triplets : "nœud ; arc ; nœud" ou "sujet ; predicat ; objet" comme dans la représentation du modèle de données RDF (Resource Description Framework [11]). L'encodage se fait avec une architecture encodeur-décodeur basée sur des réseaux de neurones récurrents (RNN). Ainsi, à l'arrivée de nouveaux triplets, il suffit d'utiliser l'état précédent pour pouvoir encoder les nouveaux triplets.

11. https://www.w3.org/TR/PR-rdf-syntax/

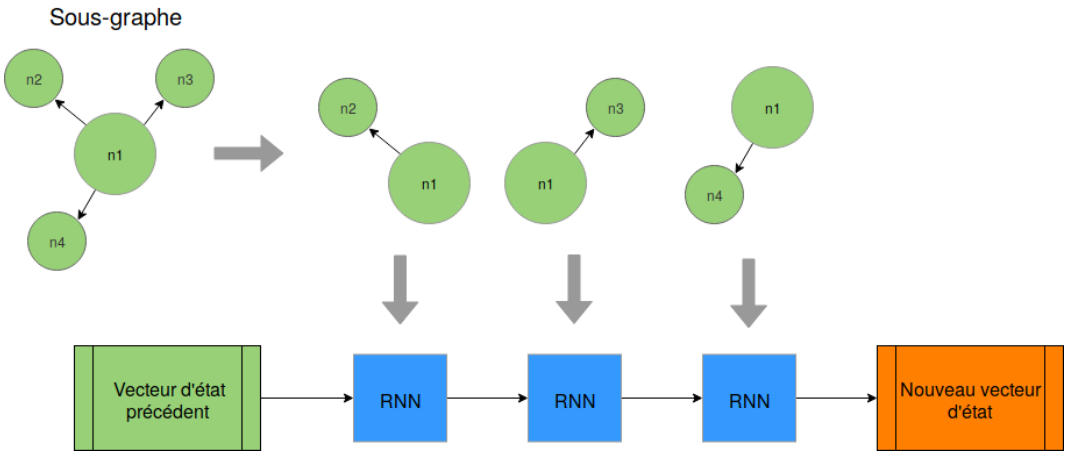

Figure 1.16 – *Schéma représentant un encodeur séquentiel de graphe*

Comme le montre la figure 1.16, lorsqu'un sous-graphe arrive, celui-ci est décomposé en triplets. Ensuite, chaque triplet est encodé séquentiellement dans le vecteur d'état précédent qui encode déjà l'ancien état du graphe. Le résultat sera un vecteur d'état encodant la totalité du nouveau graphe.

Pour faire l'apprentissage de cette architecture, il est possible de générer des graphes aléatoirement qu'on fait passer triplet par triplet par l'encodeur. Celui-ci est un RNN qui prend en entrée l'état précédent du réseau et un triplet du graphe et qui produit en sortie un nouvel état. L'état final du RNN, après avoir fait passer tous les triplets du graphe, est utilisé dans le décodeur qui est un autre RNN. Ce dernier essaye de reconstruire le graphe triplet par triplet à partir de l'état reçu comme sortie de l'encodeur. Ainsi, si on peut reconstruire le graphe, on peut dire que le dernier état encode tout le graphe dans un vecteur de taille fixe. La figure 1.17 illustre un exemple des entrées/sorties de l'apprentissage en utilisant cette architecture.

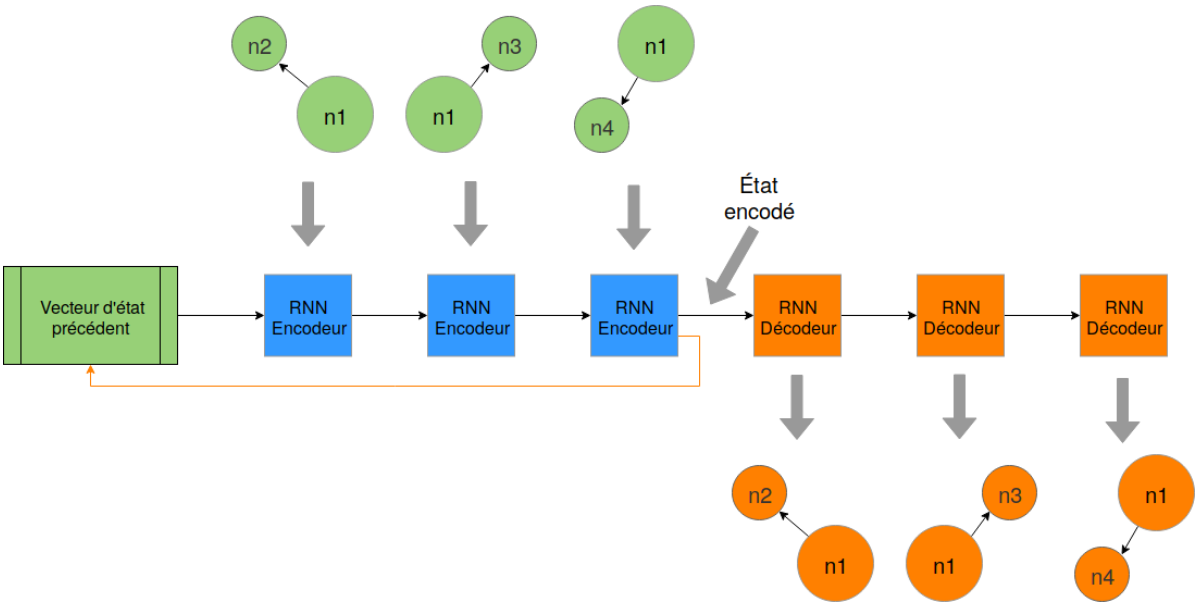

Figure 1.17 – *Schéma de l'apprentissage d'un encodeur séquentiel de graphe*

Les agents feuilles

Comme nous l'avons cité précédemment, les agents feuilles sont les agents responsables de répondre aux intentions de l'utilisateur. Pour faire l'apprentissage par renforcement d'un agent feuille, nous utilisons le simulateur d'utilisateur comme environnement de cet agent. Ce dernier interagit avec le simulateur dans le but d'estimer la correspondance des récompenses en fonction des états. Nous utilisons pour cela un réseau de neurones profond qui prend en entrée le graphe encodé de l'agent et produit pour chaque action la récompense correspondante. Comme le nombre d'actions est variable, il est impossible d'utiliser un réseau de neurones qui produit une récompense pour chaque action vu que les réseaux de neurones ont un nombre de sorties fixe. Par conséquent, il est nécessaire d'utiliser une architecture qui prend en entrée, en plus de l'état encodé, l'action candidate pour produire en sortie sa récompense.

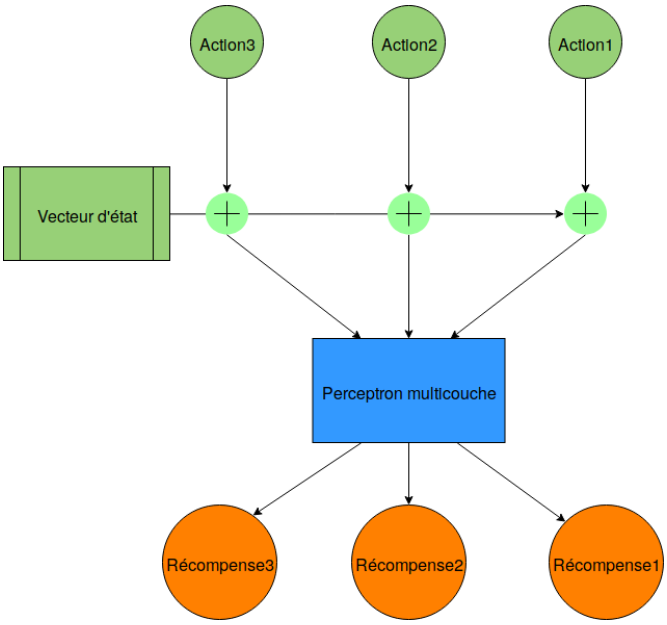

Figure 1.18 – *Schéma du réseau DQN*

L'algorithme utilisé pour l'apprentissage par renforcement est double DQN en rejouant l'expérience que nous détaillerons dans cette partie. En plus de l'utilisation des réseaux de neurones pour estimer la fonction de récompense Q, deux améliorations lui ont été ajoutées :

- Rejouer l'expérience : l'agent interagit avec le simulateur plusieurs fois en gardant dans une mémoire ses interactions. Après chaque k épisodes[12], l'agent reprend la mémoire pour entraîner le réseau de neurones.
- Double DQN : la fonction Q est donnée par la formule $Q(s,a) = r + \alpha \times max_j(Q(s', a_j))$ (Mnih et al., 2015) avec :
  - s : état de l'agent.
  - a : l'action qu'on veut estimer.

12. Un épisode est un ensemble d'interactions agent-simulateur jusqu'à l'aboutissement à un succès ou un échec

- r : récompense immédiate.
- α : paramètre de réduction.
- s' : nouvel état après avoir effectué l'action a de l'état s.
- a_j : les actions possibles à partir de l'état s'.

On remarque la récurrence dans cette formule qui nécessite la réutilisation du réseau pour estimer le terme $max_j(Q(s', a_j))$. Il a été démontré que l'utilisation d'un autre réseau qu'on fixe lors de l'apprentissage pour choisir l'action de la récompense dans le terme $max_j(Q(s', a_j))$ améliore les résultats (Mnih et al., 2015). La formule qui calcule la valeur cible du réseau de neurones au i$^{\text{ème}}$ apprentissage est donc :

$$y_i = r + \alpha \times Q_i(s', argmax_{a_j}(Q_{i-1}(s', a_j)))$$

Avec Q_i et Q_{i-1} : les fonctions de récompense données par les réseaux de neurones pendant le i$^{\text{ème}}$ apprentissage et avant le début du i$^{\text{ème}}$ apprentissage respectivement. Cette valeur y_i est donc utilisée comme cible du réseau DQN et permet de calculer l'erreur moyenne quadratique par rapport à la prédiction du réseau.

$$E = (y_i - Q_i(s, a))^2$$

Ensuite, l'algorithme de retro-propagation est appliqué pour mettre à jour les poids du réseau de neurones.

Une autre architecture possible serait de relier l'encodeur de graphe directement avec le réseau DQN pendant l'apprentissage. Ainsi, l'erreur de l'apprentissage pour l'encodeur est calculée à partir de la fonction de récompense de l'apprentissage par renforcement. L'avantage de relier l'encodeur avec le réseau DQN est de permettre à l'encodeur de contrôler quelle partie du graphe encodé est à oublier. En effet, la taille fixe du vecteur dont on encode le graphe a une limite de nombre de triplets supportable. L'utilisation des cellules de réseaux de neurones récurrents comme les LSTMs ou GRUs qui ont des portes d'oubli rend cette architecture possible. En effet, ce type de RNNs ont la possibilité de garder en mémoire (leur état) des informations qu'ils ont reçues au début de la séquence de données en sacrifiant d'autres plus récentes. En d'autres termes, ils peuvent choisir quelles sont les informations à garder dont le DQN aura besoin pour estimer la fonction de récompense.

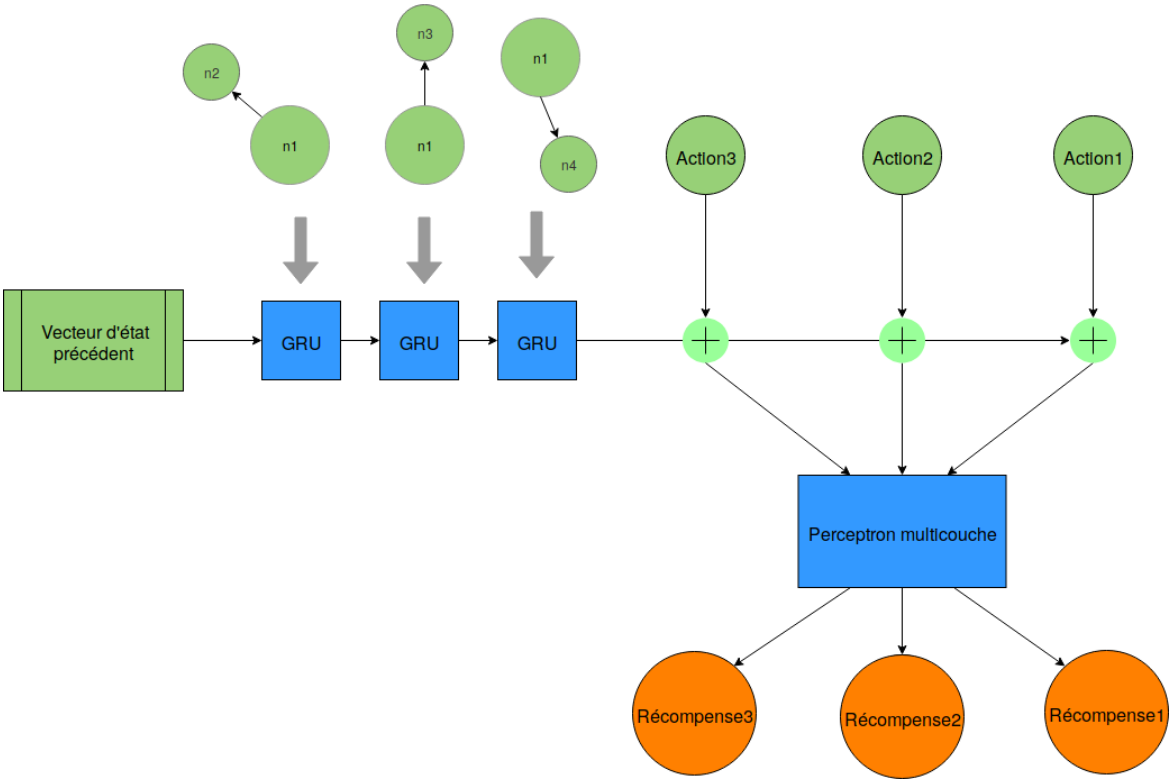

Figure 1.19 – *Schéma du réseau DQN relié avec l'encodeur directement*

L'agent coordinateur

L'agent coordinateur utilise la même architecture que celle des agents feuilles. La différence se trouve au niveau de la sortie. Dans le cas des agents coordinateurs, ils essayent de prédire quel agent fils peut répondre à la dernière action reçue.

1.6 Module de génération du langage naturel

Le modèle utilisé pour la génération du texte est relativement simple. Il s'agit de préparer des modèles de phrases contenant des emplacements à remplir. Chaque action de l'agent correspond à un ensemble de modèles et à chaque paramètre de l'action correspond un ensemble d'expressions. La génération du texte se fait en choisissant d'abord pour chaque paramètre de l'action une expression aléatoirement. Ensuite, un modèle de phrase est choisi aléatoirement. Enfin, les emplacements vides sont remplis avec les expressions des paramètres. Le module de génération de textes traite aussi les éventuelles erreurs qui peuvent se produire comme la suppression d'un fichier inexistant ou la création d'un fichier qui existe déjà. Dans ce cas, l'agent doit informer l'utilisateur que l'action demandée ne peut pas être exécutée pour qu'il résolve le problème. Le traitement des erreurs est similaire au traitement des actions. Le générateur de textes utilise le nom de l'erreur comme l'intention de l'agent et ses paramètres comme les paramètres de l'action.

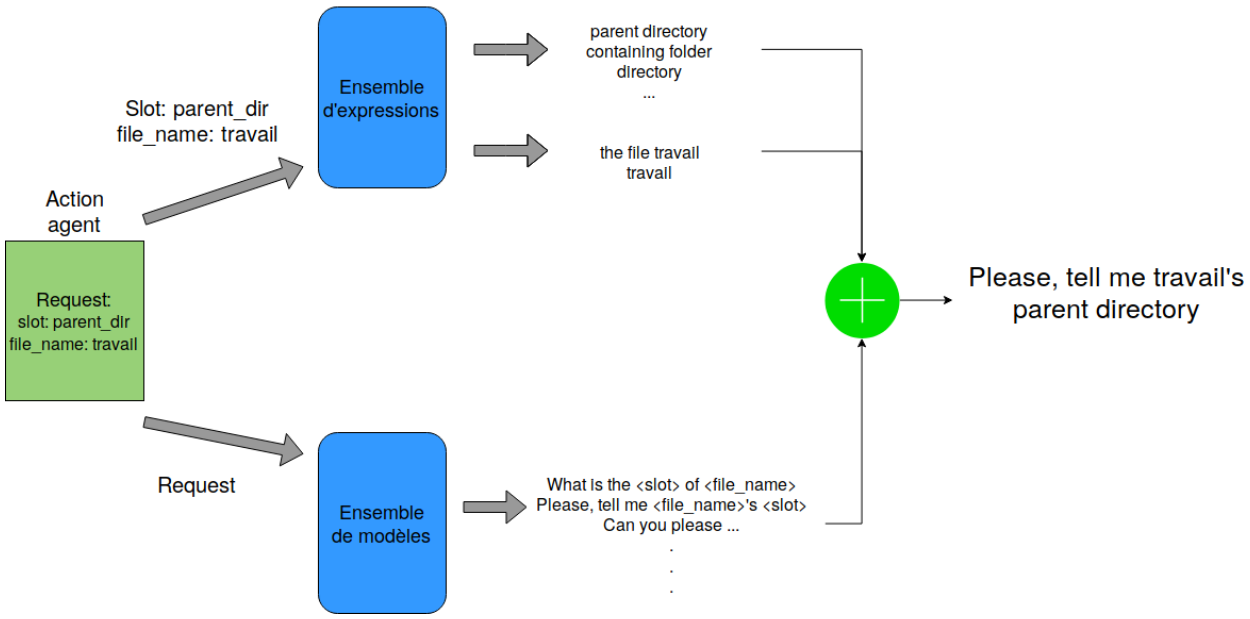

Figure 1.20 – *Schéma de fonctionnement du générateur de textes*

Dans la figure 1.20, un exemple de génération de textes en utilisant les modèles de phrases est présenté où l'agent demande à l'utilisateur de lui donner le répertoire parent d'un fichier nommé "travail". Le générateur de textes utilise l'intention de l'agent "$Request$" pour extraire les modèles de phrases qui lui correspondent. En parallèle, il utilise les paramètres de l'action pour extraire leurs expressions. Par exemple, le nom du fichier peut être transformé en "*travail*" ou "*the file travail*". Ces expressions sont enfin combinées avec le modèle de la phrase choisie pour générer une phrase complète. Par exemple, "*Please, tell me what is the **parent directory** of **travail***" ou "*Please, tell me what is the **directory** of **the file travail***".

1.7 Conclusion

Dans ce chapitre, et en nous inspirant des travaux existants, nous avons pu modéliser et conceptualiser tout les aspects du système que nous jugeons assez conforme au standard. Viennent s'ajouter les améliorations proposées pour enrichir le système de base retrouvé dans la littérature ainsi que pour faciliter l'extensibilité des fonctionnalités de bases avec un moindre effort d'intégration. Ceci est le résultat d'une conception modulaire et facilement maintenable.

Dans le chapitre suivant, nous allons passer à l'implémentation des différents modules, au développement d'une application dédiée et à une partie expérimentation pour tester les approches proposées. Nous discuterons leurs améliorations et les comparerons avec des standards existants.

2

Réalisation et résultats

2.1 Introduction

Dans la partie conception, nous avons proposé certaines modifications à apporter sur des approches existantes ainsi que des méthodes que nous avons jugées intéressantes pour notre système. Nous allons, dans la suite de ce chapitre, montrer la faisabilité de ces méthodes ainsi que leurs avantages et leurs limites.

Nous commençons par décrire l'environnement de travail. Nous détaillerons par la suite les aspects de l'implémentation des différents modules de notre application qui seront évalués et analysés. Pour clôturer avec une application d'un assistant personnel pour la manipulation de fichiers.

2.2 Environnement de développement

Dans cette section, nous allons présenter les différents outils (logiciels et matériels) qui ont été utilisés pour l'implémentation de Speech2Act.

2.2.1 Machines utilisées

Principalement, le développement se divise en deux parties :

- Apprentissage : les données sont récoltées ou construites puis nettoyées et préparées. Les modèles sont développés, entraînés puis testés.
- Les modules sont implémentés puis connectés et intégrés dans l'application.

Pour ce, faire nous avons utilisé des machines dont les spécificités sont mentionnées ci-dessous :

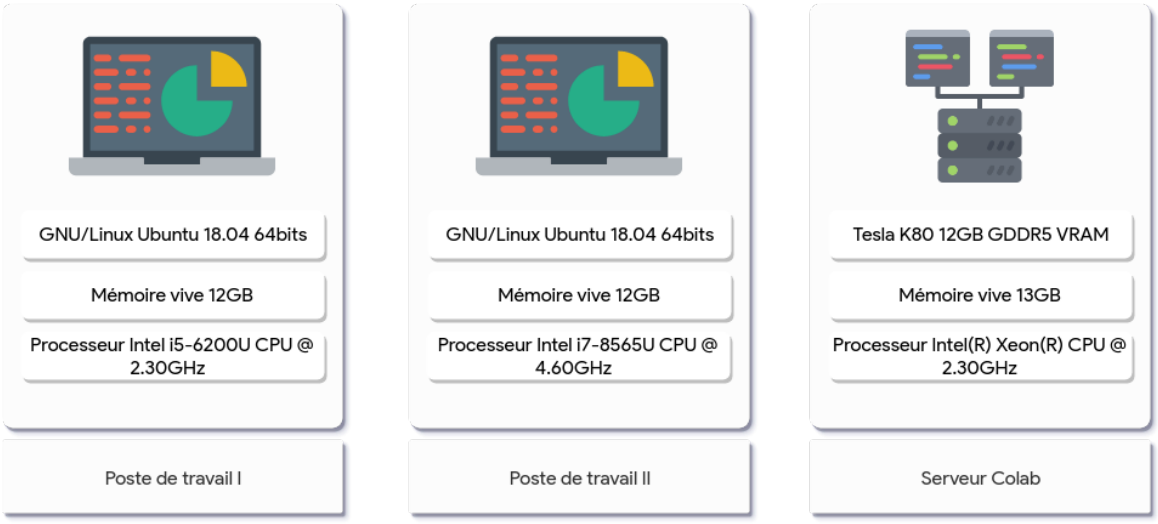

Figure 2.1 – *Caractéristiques des machines*

En ce qui concerne la partie logicielle, une liste non exhaustive est présentée ci dessous qui ne mentionne que les outils les plus utilisés et les plus exploités :

2.2.2 Langages de programmation

Python

Python[1] est un langage de programmation interprété de haut niveau, structuré et open source. Il est multi-paradigme (orienté objet, programmation fonctionnelle et impérative) et multi-usage. Il est, comme la plupart des applications et outils open source, maintenu par une équipe de développeurs un peu partout dans le monde. Il offre une grande panoplie d'extensions (packages) pour résoudre une variété de problèmes, qu'ils soient liés au développement d'applications de bureau, web ou mobiles.

Javascript

JavaScript[2] est un langage de programmation utilisé principalement par les navigateurs web pour exécuter un bout de code incorporé dans une page web, plus communément appelé script. Il permet la manipulation de tous les éléments inclus dans une page, et par conséquent permet une gestion dynamique de ces derniers. Il est beaucoup utilisé du côté client mais peut aussi être exécuté du côté serveur. Tout comme Python, il offre une grande variété dans le choix des modules qui peuvent ajouter de nouvelles fonctionnalités, le tout géré par un gestionnaire de module *npm*[3] devenu un standard.

1. https://fr.wikibooks.org/wiki/Programmation_Python/Introduction
2. https://fr.wikibooks.org/wiki/Programmation_JavaScript/Introduction
3. Node Package Manager ou Gestionnaire de packages Node

2.2.3 Librairies et bibliothèques

Figure 2.2 – *Bibliothèques et librairies les plus utilisées dans ce projet.*

DeepSpeech API

C'est un module de python qui fait office d'interface entre un script Python et la librairie de reconnaissance automatique de la parole DeepSpeech (Hannun et al., 2014). Il permet entre autres de charger différents modèles acoustiques ou modèles de langues. Il offre aussi la possibilité d'utiliser des scripts d'apprentissage prédéfinis pour peu que les données soient organisées suivant une certaine norme ; ces scripts sont notamment hautement paramétrables.

PyAudio

PyAudio[4] est un librairie python destinée à la manipulation des fichiers ou flux audios. Elle offre entre autres la possibilité d'extraire des méta-données sur un flux audio (fréquence d'échantillonnage, débit, etc.). La possibilité d'extraire le vecteur de caractéristiques d'un extrait audio est aussi présente comme fonctionnalité.

Beautiful Soup

Beautiful Soup[5] est une bibliothèque Open source permettant l'analyse de fichiers html pour en extraire ou y injecter des données. Elle est principalement utilisée pour filtrer les balises html depuis une page web.

4. https://pypi.org/project/PyAudio/
5. https://pypi.org/project/beautifulsoup4/

PySpark

PySpark[6] est un package Python utilisé comme interface pour intéragir avec un serveur Spark. Il permet entre autres de lire et écrire des données dans le nouveau format Hadoop.

Scikit-Learn

Scikit-Learn[7] est une bibliothèque Open source conçue pour rapidement développer des modèles pour l'apprentissage automatique, principalement utilisée pour ses nombreux outils de pré-traitement des données (codification, normalisation, filtrage ...).

Numpy

Numpy[8] est une bibliothèque spécialisée dans la manipulation de grands volumes de données numériques, notamment les vecteurs (tableaux) multi-dimensionnels. Les opérations sur ces derniers sont implémentées en C pour optimiser au maximum leur coût en temps de calcul ou ressources mémoires utilisées. Elle offre des structures de données compatibles avec beaucoup d'autre librairies comme Tensorflow ou Keras.

Tensorflow & Keras

Tensorflow[9] est une bibliothèque dédiée à l'apprentissage automatique, et plus particulièrement aux réseaux de neurones et l'apprentissage profond. Optimisée pour exécuter des opérations à grande échelle et massivement distribuées sur un réseau, Tensorflow offre la possibilité d'implémenter une grande variété d'architectures de modèles avec un maximum d'efficacité. Elle dispose d'un package Python permettant d'interagir avec le cœur de la bibliothèque mais reste néanmoins assez bas-niveau. Keras[10] quant-à elle propose de rajouter une couche d'abstraction à Tensorflow. C'est un package python destiné à faciliter le développement de modèles pour l'apprentissage profond tout en offrant la possibilité de rajouter et modifier un grand nombre de fonctionnalités par défaut. Sa force réside dans le fait qu'il peut utiliser au plus bas niveau plusieurs librairies autres que Tensorflow comme Theano[11] et CNTK[12].

Flask

Falsk[13] est une micro-librairie Open source dédiée au développement d'applications basées web. De base, cette librairie est très légère, mais elle offre la possibilité d'ajouter des

6. https://spark.apache.org/
7. https://scikit-learn.org/stable/
8. https://www.numpy.org/
9. https://www.tensorflow.org/
10. https://keras.io/
11. http://deeplearning.net/software/theano/
12. https://github.com/microsoft/CNTK
13. http://flask.pocoo.org/

extensions qui s'intègrent très facilement au système de base.

Vuetify

Vuetify est une librairie Open source basée sur VueJs dédié au développement d'interfaces web ou mobiles. Elle implémente le paradigme Material Design de Google et offre la possibilité d'étendre les composants de base et de créer des interfaces belles et adaptatives.

2.2.4 Outils et logiciels de développement

PyCharm

PyCharm est un environnement de développement intégré spécialisé et optimisé pour programmer dans le langage Python. Il permet l'analyse de code en continu et offre un débogueur intégré pour exécuter un code instruction par instruction. Il offre également l'intégration de logiciel de gestion de versions comme Git, et supporte le développement web avec Flask.

Git

Système décentralisé de gestion de versions. Il permet entre autres de gérer les différentes versions d'un projet durant son développement, mais aussi de garder l'historique des modifications effectuées ainsi que la possibilité de régler des conflits lors de l'intégration finale des contributions des développeurs.

Google Colaboratory

Colaboratory[14] est un outil de recherche et développement pour la formation et la recherche associées à l'apprentissage profond. C'est un environnement Python qui ne nécessite aucune configuration et offre la possibilité d'utiliser de puissantes machines rendues accessibles par Google pour accélérer la phase d'apprentissage.

Protégé

Protégé est un système dédié à la création et la modification d'ontologies. Il est développé en Java et est Open source distribué sous une licence libre (la Mozilla Public License). Il se démarque par le fait qu'il permet de travailler sur des ontologies de très grandes dimensions.

14. https://colab.research.google.com/

2.3 Reconnaissance automatique de la parole

Pour ce premier module, il a été très difficile d'effectuer les tests idéaux. En effet, nous n'avons pas pu trouver un ensemble de données qui proposait du contenu en rapport avec Speech2Act. Cependant, puisque nous avons pu construire un mini-ensemble pour tester l'apport de notre modèle de langue. Les résultats ne doivent pas être pris comme une référence absolue, mais plutôt comme une indication pour de possibles futurs tests.

2.3.1 Ensemble de test

Pour tester le modèle acoustique, les données récoltées à travers le projet CommonVoice (voir la section 1.3.2) constituent un assez bon échantillon, de par la nature des enregistrements (sur téléphone portable, par plusieurs genres et accents de locuteurs ...), mais aussi de par le volume (environs 500 heures d'enregistrements audios). Nous avons toutefois décidé de construire un mini-ensemble de test d'environ 204 enregistrements audios d'une longueur moyenne de 5 secondes chacun. Ces échantillons ont été prélevés sur trois locuteurs masculins. 20% de ces échantillons ont été prélevés dans un environnement fermé mais bruité (il s'agit d'un espace de travail pour étudiants) et sur téléphone. Le reste a été prélevé dans un environnement fermé avec peu de bruit à partir du micro d'un ordinateur portable. Chaque enregistrement est soit :

- une requête prélevée de l'ensemble de test du module de compréhension du langage naturel (voir les sections 2.4.2 et 1.4.3), ou
- une question prélevée de l'ensemble de données AskUbuntu [15] qui regroupe des questions relatives à la manipulation d'un ordinateur sous le système d'exploitation GNU/-Linux.

Pour le modèle de langue, il s'agit de celui mentionné dans la section **??**. Quelques modifications ont été rajoutées comme le filtrage des mots qui n'appartiennent pas à la langue anglaise, mais au prix du sacrifice de quelques noms propres non reconnus ou bien de séquences de mots/lettres sans réel sens.

2.3.2 Méthodologie d'évaluation

Les tests ont été effectués dans un serveur Colab pour libérer les machines locales. Les principales étapes sont les suivantes :

1. **Préparation des données** : Les données sont prélevées d'une base de données sqlite qui comprend une seule table Transcriptions. Les colonnes de la table sont :
    - **id** : identifiant de l'enregistrement
    - **path** : chemin vers le fichier audio de la requête.
    - **text** : transcription textuelle de la requête.

15. https://github.com/taolei87/askubuntu

Un script de conversion est ensuite lancé pour s'assurer que chaque enregistrement est au format .wav avec une fréquence de rafraîchissement égale à 16KHz, le modèle acoustique de DeepSpeech attend cette valeur exacte pour lancer l'inférence, sinon une erreur se produira.

2. **Métriques retenues** : À chaque instance testée, le **WER** (Word Error Rate) est calculé. Pour rappel la formule du WER est la suivante :

$$WER(y, \hat{y}) = \frac{S + D + I}{S + D + C} = \frac{S + D + I}{N}$$

où :

- $\hat{y}$ est la séquence de mots prédite appelée Hypothèse.
- y est la séquence de mots réelle appelée Référence.
- S est le nombre de substitutions (compté en mots) réalisées entre l'hypothèse et la référence.
- D est le nombre de suppressions qu'a effectué le système, donc le nombre de mots supprimés dans l'hypothèse par rapport à la référence.
- I est le nombre d'insertions effectuées par le système (c.à.d. le nombre de mots rajoutés à l'hypothèse par rapport à la référence).
- C est le nombre de mots bien placés.
- N est la longueur totale de la séquence en nombre de mots

3. **Boucle d'évaluation** : L'opération précédente est réitérée en incrémentant à chaque fois le taux d'utilisation de notre modèle de langue (de 20% à 100% avec un pas de 10%). Le WER associé est ensuite comparé à celui obtenu en utilisant le modèle de langue par défaut que propose DeepSpeech, le modèle acoustique sans modèle de langue, le résultat de l'API de Google [16] et enfin le modèle par défaut de CMU Sphinx [17]

2.3.3 Résultats

Les résultats sont décrits dans le tableau 2.1 et la figure 2.3. Nous remarquons que sur notre mini-ensemble de test, les modèles par défaut obtiennent un score très proche de 1, ce qui démontre qu'ils ont été entraînés sur des cas assez généraux, ce qui n'est pas très bon. Cependant, en changeant juste le modèle de langue par défaut en injectant des échantillons que nous avons récolté, une nette amélioration est visible (environ -20%). Ce taux d'erreur diminue en augmentant la taille du corpus utilisé pour le modèle de langue. Toutefois, après avoir pris plus de 75% du corpus, l'erreur a légèrement augmenté. Cela peut s'expliquer par la nature assez bruitée du corpus, les fichiers README.MD qui sont rédigés par des personnes, l'erreur humaine, et l'absence de processus de vérification de l'orthographe, de la grammaire ou de la syntaxe du contenu. Nous avions envisagé de choisir comme corpus des extraits de livres dédiés à la manipulation des ordinateurs sous Linux, mais nous n'avons pas trouvé de ressources gratuites ou Open source exploitables.

16. https://cloud.google.com/speech-to-text/
17. https://cmusphinx.github.io/

Il est à noter que le meilleur résultat obtenu, c.à.d un taux d'erreur de 72.6% est très loin d'être satisfaisant comparé au taux d'erreur de l'API de Google par exemple qui est d'approximativement 0.3496%. Néanmoins, l'ajout d'un modèle de langue personnalisé a pu améliorer nettement les résultats observés durant l'utilisation du modèle de base. Comme mentionné dans la section 1.3, DeepSpeech reste la meilleure option en Open Source à notre portée.

	WER (Word Error Rate)
Goole API	0,34955014
Modèle acoustique avec 50% du corpus	0,72634931
Modèle acoustique avec 75% du corpus	0,72809889
Modèle acoustique avec 100% du corpus	0,72928369
Modèle acoustique avec 25% du corpus	0,73910913
Modèle acoustique avec 10% du corpus	0,74333040
Modèle acoustique et modèle de langue de base	0,79569078
Modèle acoustique seulement	0,91092787
CMU Sphinx de base	0,94024028

Table 2.1 – *Tableau récapitulatif des résultats pour le module de reconnaissance automatique de la parole.*

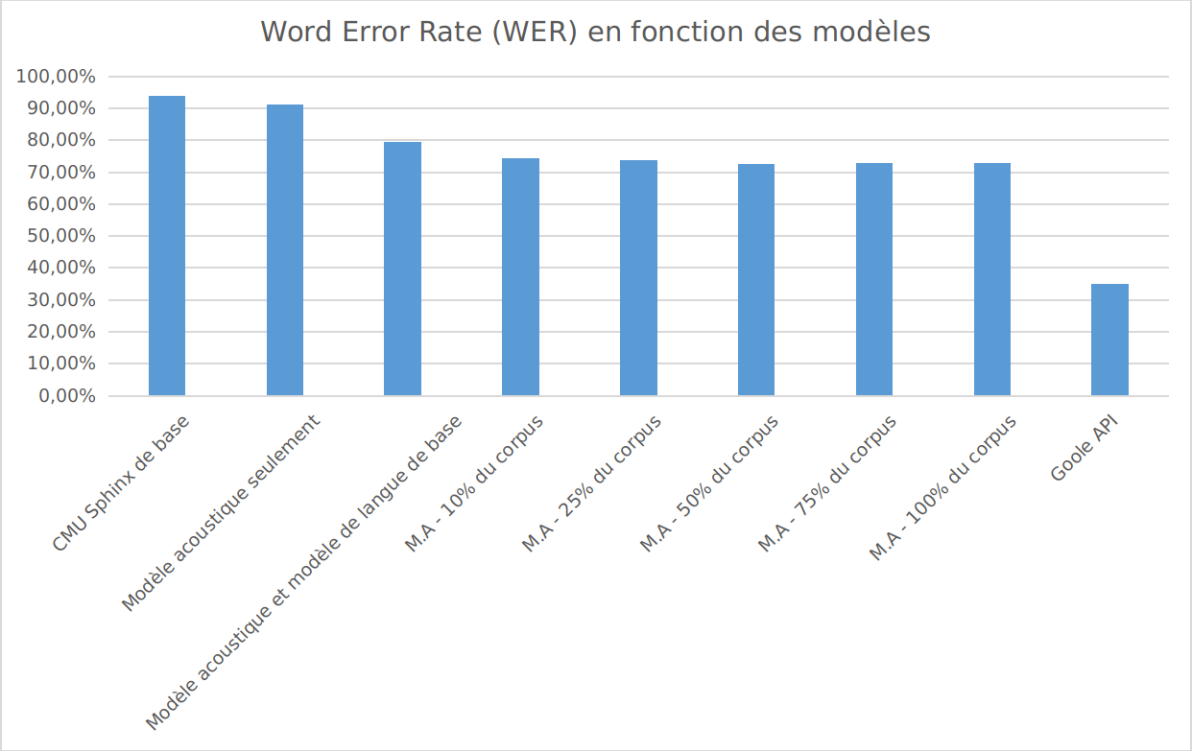

Figure 2.3 – *Graphe récapitulatif des résultats pour le a reconnaissance automatique de la parole*

2.4 Classification d'intentions et extraction d'entités de domaine

Pour ce module, une approche assez classique a été utilisée et l'ensemble de test est extrait de l'ensemble d'apprentissage (plus de détails dans la section suivante). Les métriques d'évaluation utilisées sont mentionnées dans la sous section 2.4.2. Nous commençons d'abord par détailler le contenu de l'ensemble de tests. Puis, nous présenterons la méthodologie suivie pour la réalisation de ces tests. Un tableau récapitulatif sera présenté avant la fin pour illustrer les différents résultats.

2.4.1 Ensemble de test

Comme mentionné dans le chapitre précédent (voir la section 1.4.3), nous avons nous mêmes construit un ensemble d'apprentissage relativement varié. Il regroupe pour le moment essentiellement des commandes, ou requêtes liées à l'exploration de fichiers car c'est la tâche rudimentaire que Speech2Act peut accomplir.

L'ensemble de test est dérivé de celui d'apprentissage selon une politique de découpage basée sur le taux de présence d'un intent (intention). Comme illustré dans la figure 2.4, un pourcentage de chaque groupe d'instances affilié à la même classe est utilisé à la fois pour la validation et pour le test. Ce choix est motivé par le fait que les proportions des distributions des intentions dans l'ensemble original sont non-équilibrées.

Une liste exhaustive des intentions et slots accompagnée d'une description est introduite dans le tableau 2.2 ci dessous.

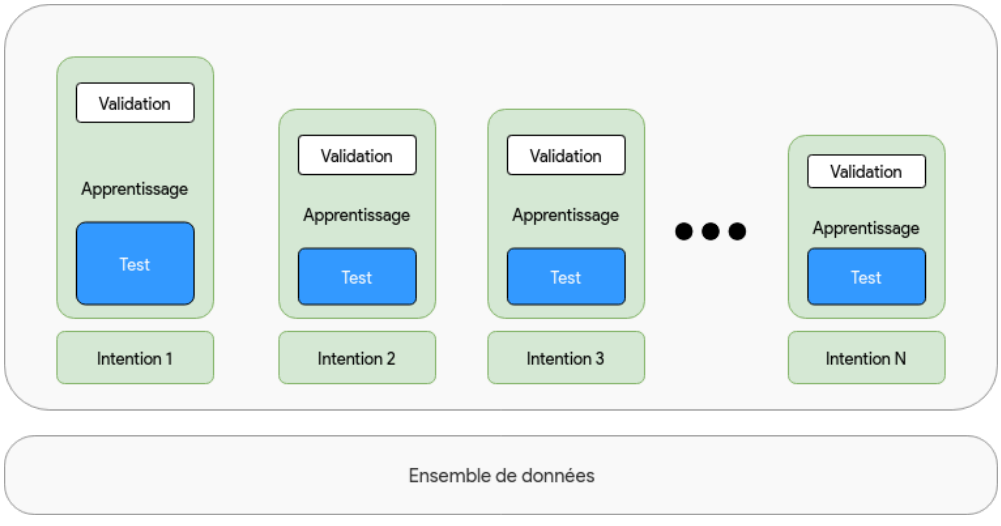

Figure 2.4 – *Schéma de découpage des données pour l'apprentissage du modèle de compréhension du langage naturel*

Intention	Description de l'intention	Groupe	Argument(s) (Entité(s))
create_file_desire	Création d'un fichier	ALTER	-file_name -parent_directory
create_directory_desire	Création d'un répertoire	ALTER	-directory_name -parent_directory
delete_file_desire	Suppression d'un fichier	ALTER	-file_name -parent_directory
delete_directory_desire	Suppression d'un répertoire	ALTER	-directory_name -parent_directory
open_file_desire	Ouverture d'un fichier	INFO	-file_name -parent_directory
close_file_desire	Fermeture d'un fichier	INFO	-file_name -parent_directory
copy_file_desire	Copie d'un fichier	ALTER	-file_name -origin -destination
move_file_desire	Déplacement d'un fichier	ALTER	-file_name -origin -destination
rename_file_desire	Renommage d'un fichier	-	-old_name -new_name
change_directory_desire	Changement du répertoire de travail courant	-	-new_directory
inform	Informer d'une intention	EXCH	-file_name -parent_directory
request	Demander une information	EXCH	-file_name -directory
deny	Réponse négative	-	-
confirm	Réponse positive	-	-
unknown	Intention inconnue	-	-

Table 2.2 – *Tableau récapitulatif de toutes les intentions avec leurs descriptions et leurs arguments.*

Les intentions sont regroupées en groupes ; chaque groupe comporte des intentions qui influent sur les documents de la même manière. Il existe bien évidemment des intentions sans groupe, on peut considérer qu'ils forment un seul groupe, ou bien que chacun constitue son propre groupe :

- **ALTER** : Groupe d'intentions qui altère l'état d'un document.
- **INFO** : Groupe d'intentions pour effectuer une opération puis informer l'utilisateur.
- **EXCH** : Groupe d'intentions dont le but est d'échanger des informations avec l'utilisateur.

2.4.2 Méthodologie d'évaluation

Après avoir construit l'ensemble de test, un parcours exhaustif des différentes combinaisons des paramètres suivants est effectué :

- **Architecture d'encodage** : C'est à dire l'utilisation ou pas d'un réseau récurrent LSTM de base ou bien BiLSTM. Le but étant de montrer que le modèle pourra mieux interpréter les données en entrée s'il capture le contexte de chaque mot.
- **Nombre de neurones pour la couche de classification d'intention** : Lorsque l'encodeur retourne le dernier vecteur d'état caché, ce dernier passera par un réseau de neurones complètement connecté et multi-couches dont nombre de couches est fixé à 3 par souci de performance, une couche d'entrée, une couche intermédiaire et une couche de sortie. Le nombre de neurones sur la couche cachée dépend grandement de la complexité de la tâche à effectuer. La classification d'intentions pour l'exploration de fichier était relativement simple ; nous avons commencé avec 32 neurones puis nous avons doublé ce nombre jusqu'à 512 pour, en théorie, donner plus de puissance au classificateur tout en évitant un sur-apprentissage par surplus de neurones.
- **Nombre d'unités d'une cellule LSTM (respectivement BiLSTM)** : Les portes d'une cellule LSTM (resp. BiLSTM) sont en vérité des réseaux de neurones denses (complètement connectés) et donc un ensemble de matrices de poids à optimiser. La capacité à "apprendre" la représentation des séquences dépend aussi du nombre de neurones dans ces mini-réseaux. Par le même raisonnement employé pour le classificateur d'intentions, nous avons commencé avec un petit nombre de neurones pour examiner d'une part où se trouverait le seuil minimal qui permettra au modèle de généraliser, et d'autre part où le seuil critique se situerait pour permettre au modèle de ne pas tomber dans un cas de sur-apprentissage. Nous partons de 128 jusqu'à 512 unités avec pas de 32 unités.
- **Fonctions d'activation** : C'est un élément essentiel qui permet d'introduire la non-linéarité dans les relations entre chaque neurones de couches voisines. Ces fonctions permettent de mieux représenter les seuils d'activation des neurones. La fonction la plus utilisée dans la littérature est actuellement ReLu (Rectified Linear Unit) car elle a expérimentalement donné de meilleurs résultats dans une grande variété de tâches et problèmes liés à l'apprentissage automatique et à la classification et étiquetage de textes. Par souci d'exhaustivité, nous avons quand même décidé de tester deux autres fonctions $tanh$ (Tangente Hyperbolique) et $sigmoid$ (Sigmoïde). Pour chaque couche de sortie, la fonction $Softmax$ a été appliquée car chaque couche traite d'un problème de classification multi-classes. Voici les équations pour chacune de ces fonctions :

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

$$softmax_i(x) = \frac{e^{x_i}}{\sum_{j=1}^{K} e_j^x} \text{ pour } i = 1, ..., K \text{ et } x = (x_1, ..., x_K) \in \mathbb{R}^K$$

- **Fonction erreur** : C'est l'élément clé pour la phase d'apprentissage, cette fonction détermine le degré d'exactitude du modèle, c'est à dire à quel point il est proche de la bonne réponse. Nous avons décidé d'utiliser comme fonction erreur la fonction $Categorical\_Crossentropy$ ou Erreur Logistique ; de par la nature de l'ensemble d'apprentissage et de test. Une version pondérée de cette fonction a été préférée, pour palier au problème du non équilibrage des classes, que ce soit pour la classification d'intentions, ou pour la reconnaissance d'entités du domaine.

  Les poids des classes sont calculés selon la formule suivante :

  $$Poids_i = \max(1, \log \frac{T}{T_i})$$

  où :

  - T est le nombre total d'instances
  - T_i est le nombre d'instances dont la classe est C_i

  La formule de la fonction erreur devient donc :

  $$Erreur(y, \hat{y}) = -\sum_{i}^{C} y_i * \log(\hat{y}_i) * Poids_i$$

  où :

  - $\hat{y}$ est le vecteur en sortie produit par le modèle à la suite d'une fonction $Softmax$.
  - y est le vecteur de classe réelle présent dans l'ensemble d'apprentissage
  - C est le nombre de classes au total.
- **Fonction d'apprentissage** : Le choix de la fonction d'apprentissage est généralement affecté par un désir de précision et de rapidité. Une fonction qui converge rapidement en un minimum local peut être parfois préférée à une autre qui prendrait un temps considérable pour soit se retrouver dans le même minimum ou un autre minimum local (donc sans garantie de minimum optimal de la fonction erreur). Les deux fonctions utilisées sont RmsProp et Adam qui sont connues pour leur rapidité de convergence.
- **Encodage des entrée-sorties** : Là encore le choix de l'encodage des données influe grandement sur la capacité du modèle à distinguer et à représenter les différentes informations qui lui sont présentées. Comme initiative de notre part, nous avons mentionné dans la section 1.4.3 l'ajout de l'étiquette morphosyntaxique de chaque mot à l'encodage. Nous avons donc lancé les tests sur un encodage avec et sans l'ajout des étiquettes pour mieux constater son impact.
- **Découpage des données** : La stratégie adoptée était de prendre aléatoirement les même proportions pour chaque sous-ensemble de chaque classe (intentions ou entités de domaine). Nous avons aussi décidé de faire varier les proportions de test et d'apprentissage en fixant celui de validation car nous avons remarqué que notre modèle n'arrivait pas à bien généraliser la relation entre les entrées et les sorties. Notre intuition portait sur le fait que le manque de données pouvait en être la cause (voir la section 1.4.3 pour plus de détails). Ainsi, nous avons varié le taux de découpage pour

les données de tests entre 25% et 75% avec un pas de 25%. Le taux de découpage pour l'apprentissage est évidement de $(100\% - T_{test}) * 90\%$.

Pour éviter que le modèle ne sur-apprenne, nous avons volontairement interchangé quelques mots dans la séquence d'entrée et celle de sortie pour introduire un certain taux d'erreur et de variété. Cet échange se fait suivant une probabilité q fixée à 20%. Bien entendu, les étiquettes morphosyntaxiques ne sont pas échangées, et ce pour garder la structure de la phrase correcte.

Pour le reste des hyper-paramètres, la plupart ont été fixés par manque de temps et de ressources. Ainsi, le nombre d'epochs (itérations) a été limité au maximum à 15 avec une politique d'arrêt anticipé si la fonction erreur d'évaluation ne diminue pas plus d'un taux $\Delta E = 2 * 10^{-3}$ pendant au moins 4 itérations successives. Les métriques employées pour évaluer les deux classificateurs sont les suivantes :

- **Précision** : il s'agit d'une métrique classique qui évalue à quel point le modèle est bon pour prédire les classes.

$$P = \frac{VP}{VP + FP}$$

où :

  - VP (Vrais Positifs) : nombre de cas où le modèle prédit correctement la classe comme étant positive.
  - FP (Faux Positifs) : nombre de cas où le modèle ne prédit pas correctement la classe comme étant positive.

- **Rappel** : Cette métrique évalue la capacité du modèle à effectuer des classifications correctes par rapport à tout l'ensemble de test. Plus formellement :

$$R = \frac{VP}{VP + FN}$$

où :

  - FN (Faux Négatifs) : nombre de cas où le modèle ne prédit pas correctement la classe comme étant négative.

- **F-Mesure** : Mesure qui combine (d'une point de vue ensembliste) la précision et le rappel. Elle ne privilégie aucune des deux et essaye de donner un aperçu plus global de l'efficacité de l'algorithme en prenant compte des résultats de ces deux mesures.

$$F - Mesure = \frac{2 * R * P}{R + P}$$

2.4.3 Résultats

Pour les résultats qui vont suivre, chaque tableau sera accompagné d'un paragraphe qui servira de commentaire aux résultats obtenus. Des remarques peuvent y être insérées pour attirer l'attention sur des détails non évidents.

Nombre de neurones	Unités LSTM	Activation	Apprentissage	Précision Intention	Rappel Intention	Précision Entités de domaine	Rappel Entités de domaine	F-Mesure moyenne
32	128	relu	rmsprop	0,9743	0,9725	0,9214	0,9128	0,9452
32	256	relu	rmsprop	0,9755	0,9741	0,9066	0,8985	0,9387
32	512	relu	rmsprop	0,9750	0,9735	0,9082	0,9019	0,9396
64	128	relu	rmsprop	0,9700	0,9689	0,8744	0,8636	0,9192
64	256	relu	rmsprop	0,9753	**0,9743**	0,9141	0,9061	0,9424
64	512	relu	rmsprop	0,9687	0,9669	**0,9293**	0,9230	**0,9470**
128	128	relu	rmsprop	0,9638	0,9626	0,8425	0,8283	0,8993
128	256	relu	rmsprop	0,9714	0,9707	0,8431	0,8345	0,9049
128	512	relu	rmsprop	0,9647	0,9635	0,8690	0,8584	0,9139
256	128	relu	rmsprop	0,9704	0,9692	0,8473	0,8342	0,9052
256	256	relu	rmsprop	0,7215	0,7141	0,7573	0,7275	0,7299
256	512	relu	rmsprop	0,9716	0,9711	0,8842	0,8761	0,9257

Table 2.3 – *Résultats des tests pour un encodage sans étiquetage morphosyntaxique avec des cellules LSTM avec découpage Apprentissage : 25%, Validation : 10%, Test : 75%.*

Nous pouvons remarquer depuis le tableau 2.3 que pour un ensemble d'apprentissage assez réduit, le modèle arrive tout de même à bien classifier la majorité des intentions avec un rappel maximum de $97,43\%$. Cependant, le slot-filling se révèle être une tâche plus ardue avec un rappel ne dépassant pas $92,30\%$. La meilleure combinaison qui équilibre les deux tâches utilise un petit nombre de neurones pour la classification d'intentions, mais en revanche demande une grande capacité de calcul pour la mémorisation des séquences en utilisant 512 unités dans les cellules LSTM.

Nombre de neurones	Unités LSTM	Activation	Apprentissage	Précision Intention	Rappel Intention	Précision Entités de domaine	Rappel Entités de domaine	F-Mesure moyenne
32	128	relu	rmsprop	0,9736	0,9721	0,8894	0,8817	0,9292
32	256	relu	rmsprop	0,9746	0,9735	0,9344	0,9298	0,9531
32	512	relu	rmsprop	0,9721	0,9714	0,9149	0,9089	0,9418
64	128	relu	rmsprop	0,9687	0,9684	0,9344	0,9286	0,9500
64	256	relu	rmsprop	0,9737	0,9729	0,9389	0,9325	0,9545
64	512	relu	rmsprop	0,9741	0,9732	0,9257	0,9188	0,9479
128	128	relu	rmsprop	0,9684	0,9674	0,9354	0,9299	**0,9503**
128	256	relu	rmsprop	0,9709	0,9703	0,9360	0,9304	0,9519
128	512	relu	rmsprop	0,9749	0,9736	0,9426	**0,9380**	0,9573
256	128	relu	rmsprop	0,9681	0,9673	0,9349	0,9286	0,9497
256	256	relu	rmsprop	0,9686	0,9679	0,9048	0,8995	0,9352
256	512	relu	rmsprop	0,9768	**0,9762**	0,9272	0,9221	0,9505

Table 2.4 – *Résultats des tests pour un encodage sans étiquetage morphosyntaxique avec des cellules BiLSTM avec découpage Apprentissage : 25%, Validation : 10%, Test : 75%.*

Dans le tableau 2.4, l'ajout de l'information du contexte pour un mot à une position donnée à travers l'utilisation d'une architecture BiLSTM affecte systématiquement les scores (Précision, Rappel et F-Mesure). Ceux-ci augmentent d'un certain taux ($\approx 10\%$) dans la majorité des cas. Il est a noter que pour le même nombre d'unités BiLSTM, le nombre de neurones pour la classification d'intentions a doublé.

Nombre de neurones	Unités LSTM	Activation	Apprentissage	Précision Intention	Rappel Intention	Précision Entités de domaine	Rappel Entités de domaine	F-Mesure moyenne
32	128	relu	rmsprop	0,9769	0,9738	0,9189	0,9115	0,9453
32	256	relu	rmsprop	0,9703	0,9693	0,8609	0,8513	0,9129
32	512	relu	rmsprop	0,9717	0,9706	0,9304	0,9238	0,9491
64	128	relu	rmsprop	0,9696	0,9672	0,8556	0,8465	0,9097
64	256	relu	rmsprop	0,969	0,9686	0,9354	**0,9287**	**0,9504**
64	512	relu	rmsprop	0,9761	**0,9750**	0,8275	0,8201	0,8997
128	128	relu	rmsprop	0,9713	0,9695	0,8565	0,8455	0,9107
128	256	relu	rmsprop	0,9689	0,9681	0,9204	0,9131	0,9426
128	512	relu	rmsprop	0,9714	0,9708	0,8756	0,8693	0,9218
256	128	relu	rmsprop	0,9707	0,9699	0,8767	0,8674	0,9212
256	256	relu	rmsprop	0,965	0,9642	0,8767	0,8707	0,9191
256	512	relu	rmsprop	0,9751	0,9739	0,8418	0,8291	0,905

Table 2.5 – *Résultats des tests pour un encodage avec étiquetage morphosyntaxique avec des cellules LSTM avec découpage Apprentissage : 25%, Validation : 10%, Test : 75%.*

Nombre de neurones	Unités LSTM	Activation	Apprentissage	Précision Intention	Rappel Intention	Précision Entités de domaine	Rappel Entités de domaine	F-Mesure moyenne
32	128	relu	rmsprop	0,9726	0,9708	0,93	0,9232	0,9491
32	256	relu	rmsprop	0,9726	0,9715	0,9481	**0,9436**	**0,9589**
32	512	relu	rmsprop	0,9683	0,9656	0,9392	0,933	0,9515
64	128	relu	rmsprop	0,9735	0,9719	0,8369	0,8278	0,9025
64	256	relu	rmsprop	0,9735	0,9731	0,937	0,9329	0,9541
64	512	relu	rmsprop	0,9651	0,9636	0,9324	0,9261	0,9468
128	128	relu	rmsprop	0,9702	0,9687	0,8721	0,867	0,9195
128	256	relu	rmsprop	0,9741	0,9732	0,8209	0,8114	0,8949
128	512	relu	rmsprop	0,9794	**0,979**	0,923	0,9163	0,9494
256	128	relu	rmsprop	0,9654	0,9645	0,9324	0,9266	0,9472
256	256	relu	rmsprop	0,9682	0,967	0,9394	0,9352	0,9525
256	512	relu	rmsprop	0,9733	0,9723	0,9357	0,9304	0,9529

Table 2.6 – *Résultats des tests pour un encodage avec étiquetage morphosyntaxique avec des cellules BiLSTM avec découpage Apprentissage : 25%, Validation : 10%, Test : 75%.*

Quant-aux deux tableaux 2.5 et 2.6, ils montrent que l'ajout de l'étiquette morphosyntaxique a amélioré les résultats pour cas de l'utilisation d'une cellule LSTM simple tout en réduisant le nombre d'unités requises, moins de calculs et plus d'efficacité. Cela se confirme encore plus pour le cas de l'utilisation de l'architecture BiLSTM, en réduisant de presque la moitié la puissance de calcul nécessaire et en augmentant un tout petit peu la qualité des prédictions.

Nombre de neurones	Unités LSTM	Activation	Apprentissage	Précision Intention	Rappel Intention	Précision Entités de domaine	Rappel Entités de domaine	F-Mesure moyenne
32	128	relu	rmsprop	0,9707	0,9691	0,8897	0,8776	0,9267
32	256	relu	rmsprop	0,974	0,9724	0,8921	0,8859	0,9311
32	512	relu	rmsprop	0,9708	0,9699	0,9293	0,9222	0,948
64	128	relu	rmsprop	0,9751	0,9729	0,8901	0,8796	0,9294
64	256	relu	rmsprop	0,9706	0,9698	0,9096	0,903	0,9383
64	512	relu	rmsprop	0,9717	0,9708	0,934	0,9262	**0,9506**
128	128	relu	rmsprop	0,965	0,9633	0,8212	0,8081	0,8894
128	256	relu	rmsprop	0,9712	0,9706	0,893	0,8817	0,9291
128	512	relu	rmsprop	0,9711	0,9698	0,9316	**0,9269**	0,9498
256	128	relu	rmsprop	0,9749	**0,9741**	0,9079	0,8988	0,9389
256	256	relu	rmsprop	0,9718	0,9712	0,8629	0,8555	0,9153
256	512	relu	rmsprop	0,9728	0,9725	0,93	0,9239	0,9498

Table 2.7 – *Résultats des tests pour un encodage sans étiquetage morphosyntaxique avec des cellules LSTM avec découpage Apprentissage : 50%, Validation : 10%, Test : 50%.*

Nombre de neurones	Unités LSTM	Activation	Apprentissage	Précision Intention	Rappel Intention	Précision Entités de domaine	Rappel Entités de domaine	F-Mesure moyenne
32	128	relu	rmsprop	0,9539	0,948	0,8215	0,8064	0,8824
32	256	relu	rmsprop	0,9649	0,9629	0,8947	0,8887	0,9278
32	512	relu	rmsprop	0,9689	0,9673	0,9456	**0,9409**	0,9557
64	128	relu	rmsprop	0,9679	0,9667	0,8941	0,8879	0,9291
64	256	relu	rmsprop	0,974	0,9732	0,9425	0,9369	**0,9567**
64	512	relu	rmsprop	0,9771	**0,9762**	0,9339	0,9302	0,9543
128	128	relu	rmsprop	0,9699	0,9693	0,9326	0,9264	0,9496
128	256	relu	rmsprop	0,9654	0,9645	0,9384	0,9328	0,9503
128	512	relu	rmsprop	0,8747	0,8657	0,7848	0,7628	0,8219
256	128	relu	rmsprop	0,971	0,97	0,7782	0,7656	0,8712
256	256	relu	rmsprop	0,9716	0,9711	0,9327	0,9266	0,9505
256	512	relu	rmsprop	0,9752	0,9745	0,939	0,9326	0,9553

Table 2.8 – *Résultats des tests pour un encodage sans étiquetage morphosyntaxique avec des cellules BiLSTM avec découpage Apprentissage : 50%, Validation : 10%, Test : 50%.*

D’après les résultats des tableaux 2.7 et 2.8, le choix de l’architecture BiLSTM à amélioré la qualité des classifications, même si ce n’est que d’un petit taux. Ces tableaux semblent aussi montrer que notre intuition sur la faible quantité de données que nous possédions soit fondée. Les scores de F-Mesure tendent en moyenne à augmenter avec l’injection de nouvelles données d’apprentissage.

Nombre de neurones	Unités LSTM	Activation	Apprentissage	Précision Intention	Rappel Intention	Précision Entités de domaine	Rappel Entités de domaine	F-Mesure moyenne
32	128	relu	rmsprop	0,9729	0,9712	0,9256	0,9182	0,9469
32	256	relu	rmsprop	0,9734	0,9716	0,8769	0,8689	0,9227
32	512	relu	rmsprop	0,9692	0,9681	0,9278	0,9226	0,9469
64	128	relu	rmsprop	0,9712	0,9693	0,8732	0,8641	0,9195
64	256	relu	rmsprop	0,9720	0,9709	0,9278	0,9219	0,9482
64	512	relu	rmsprop	0,9689	0,9671	0,9187	0,9107	0,9414
128	128	relu	rmsprop	0,9631	0,9614	0,8223	0,8090	0,8889
128	256	relu	rmsprop	0,9694	0,9686	0,8020	0,7936	0,8834
128	512	relu	rmsprop	0,9723	0,9716	0,9388	**0,9330**	**0,9539**
256	128	relu	rmsprop	0,9662	0,9655	0,8661	0,8563	0,9135
256	256	relu	rmsprop	0,9702	0,9699	0,9077	0,9016	0,9373
256	512	relu	rmsprop	0,9737	**0,9731**	0,9238	0,9193	0,9475

Table 2.9 – *Résultats des tests pour un encodage avec étiquetage morphosyntaxique avec des cellules LSTM avec découpage Apprentissage : 50%, Validation : 10%, Test : 50%.*

Nombre de neurones	Unités LSTM	Activation	Apprentissage	Précision Intention	Rappel Intention	Précision Entités de domaine	Rappel Entités de domaine	F-Mesure moyenne
32	128	relu	rmsprop	0,9906	0,9900	0,9559	0,9524	0,9722
32	256	relu	rmsprop	0,9920	0,9914	0,9650	0,9623	**0,9777**
32	512	relu	rmsprop	0,9900	0,9894	0,9603	0,9577	0,9744
64	128	relu	rmsprop	0,9903	0,9903	0,9621	0,9594	0,9755
64	256	relu	rmsprop	0,9871	0,9870	0,9570	0,9539	0,9713
64	512	relu	rmsprop	0,9893	0,9886	0,9604	0,9579	0,9741
128	128	relu	rmsprop	0,9935	**0,9931**	0,9581	0,9550	0,9749
128	256	relu	rmsprop	0,9885	0,9883	0,8658	0,8610	0,9259
128	512	relu	rmsprop	0,9896	0,9889	0,9619	0,9591	0,9749
256	128	relu	rmsprop	0,9878	0,9877	0,9518	0,9481	0,9689
256	256	relu	rmsprop	0,9882	0,9875	0,9016	0,8959	0,9433
256	512	relu	rmsprop	0,9883	0,9879	0,9652	**0,9625**	0,9760

Table 2.10 – *Résultats des tests pour un encodage avec étiquetage morphosyntaxique avec des cellules BiLSTM avec découpage Apprentissage : 50%, Validation : 10%, Test : 50%.*

Pour les tableaux 2.9 et 2.10, nous pouvons observer que l'ajout de l'information morphosyntaxique a amélioré le taux de réussite de la prédiction d'un faible taux au profit d'une moindre puissance de calculs.

Nombre de neurones	Unités LSTM	Activation	Apprentissage	Précision Intention	Rappel Intention	Précision Entités de domaine	Rappel Entités de domaine	F-Mesure moyenne
32	128	relu	rmsprop	0,9873	0,9869	0,9556	0,9518	0,9704
32	256	relu	rmsprop	0,9871	0,9869	0,9616	0,9589	0,9736
32	512	relu	rmsprop	0,9914	**0,9913**	0,9618	0,9588	0,9758
64	128	relu	rmsprop	0,9896	0,9896	0,9520	0,9477	0,9697
64	256	relu	rmsprop	0,9904	0,9904	0,9629	**0,9600**	0,9760
64	512	relu	rmsprop	0,9894	0,9894	0,9519	0,9479	0,9696
128	128	relu	rmsprop	0,9874	0,9874	0,9263	0,9204	0,9554
128	256	relu	rmsprop	0,9916	0,9897	0,9628	0,9600	**0,9760**
128	512	relu	rmsprop	0,9885	0,9883	0,9581	0,9548	0,9724
256	128	relu	rmsprop	0,9803	0,9792	0,9105	0,9014	0,9429
256	256	relu	rmsprop	0,9906	0,9906	0,9509	0,9470	0,9698
256	512	relu	rmsprop	0,9901	0,9892	0,9607	0,9576	0,9744

Table 2.11 – *Résultats des tests pour un encodage sans étiquetage morphosyntaxique avec des cellules LSTM avec découpage Apprentissage : 75%, Validation : 10%, Test : 25%.*

Nombre de neurones	Unités LSTM	Activation	Apprentissage	Précision Intention	Rappel Intention	Précision Entités de domaine	Rappel Entités de domaine	F-Mesure moyenne
32	128	relu	rmsprop	0,9889	0,9887	0,9626	0,9601	0,9750
32	256	relu	rmsprop	0,9865	0,9865	0,9611	0,9571	0,9728
32	512	relu	rmsprop	0,9862	0,9846	0,9622	0,9597	0,9732
64	128	relu	rmsprop	0,9868	0,9866	0,9627	0,9598	0,9740
64	256	relu	rmsprop	0,9869	0,9856	0,8581	0,8504	0,9202
64	512	relu	rmsprop	0,9872	0,9871	0,9650	0,9626	0,9755
128	128	relu	rmsprop	0,9779	0,9779	0,9609	0,9580	0,9687
128	256	relu	rmsprop	0,9884	0,9870	0,9547	0,9516	0,9704
128	512	relu	rmsprop	0,9880	0,9879	0,9616	0,9587	0,9741
256	128	relu	rmsprop	**0,9919**	0,9919	0,9536	0,9502	0,9719
256	256	relu	rmsprop	0,9900	0,9900	0,9658	0,9626	**0,9771**
256	512	relu	rmsprop	0,9817	0,9817	0,9655	**0,9630**	0,9730

Table 2.12 – *Résultats des tests pour un encodage sans étiquetage morphosyntaxique avec des cellules BiLSTM avec découpage Apprentissage : 75%, Validation : 10%, Test : 25%.*

Les tableaux 2.11 et 2.12 démontrent que la taille des données d'apprentissage poussent vers de meilleurs résultats. Cela reste conforme à notre intuition théorique et encourage encore plus le développement d'un corpus plus volumineux pour l'obtention probable de meilleurs résultats.

Pour mieux récapituler et visualiser les différences de qualité de prédictions des différents modèles, la figure 2.5 ci dessous est proposée.

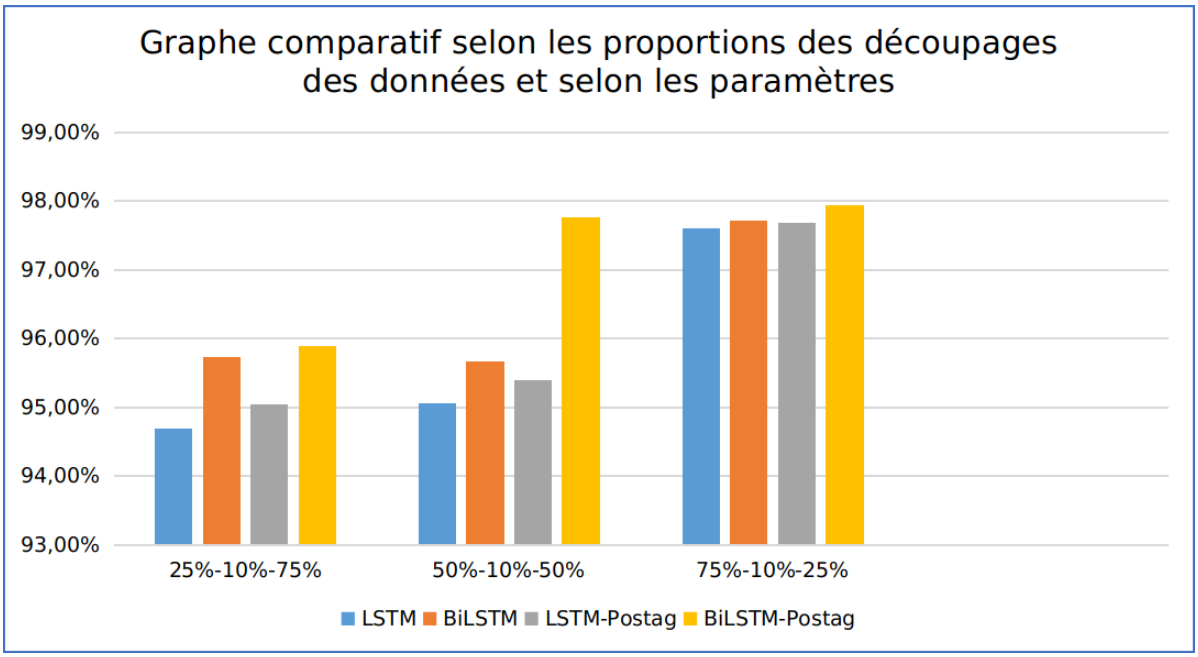

Figure 2.5 – *Graphe comparatif selon les proportions des découpages des données et selon les paramètres.*

Nous pouvons remarquer l'impact de trois facteurs :

- **La taille de l'ensemble d'apprentissage** : Systématiquement, le score F-Mesure moyen augmente avec la croissance du volume de données d'apprentissage. Cela laisse présager qu'avec plus de données le modèle pourrait mieux généraliser, ce qui éviterait de biaiser le modèle vers un type de données en particulier.
- **L'utilisation du contexte (LSTM contre BiLSTM)** : Le modèle BiLSTM donne de meilleur résultat lorsque l'information du contexte lui est accessible, ce qui confirme notre intuition.
- **L'utilisation de l'étiquetage morphosyntaxique** : L'ajout de l'information sur la syntaxe de la requête semble aider le modèle à construire une meilleure représentation interne de la requête. Il ne voit pas un mot seulement à un instant t, mais plutôt la paire (mot,rôle syntaxique du mot dans la phrase). Cet ajout est conforme à notre explication théorique dans la section 1.4.1

2.5 Ontologie et manipulation du graphe d'état

Les ontologies déjà présentées dans la section 1.5.2.1 ont été créées en utilisant Protégé. Elles ont été ensuite exportées en format Turtle [18] afin de les exploiter dans la suite de ce travail en utilisant la bibliothèque RDFLib de Python.

Les nœuds et les relations du graphe ont des identifiants entiers pour faciliter leur utilisa-

18. Turtle est une syntaxe pour l'écriture des triplets d'un graphe de connaissance avec RDF

tion avec les réseaux de neurones. Une phase de transformation des URIs [19] en identifiants entiers s'avère nécessaire. L'ontologie comprend 61 nœuds et 13 relations. Contrairement au nombre de relations, le nombre de concepts augmente au cours du dialogue ; de nouveaux nœuds sont introduits après chaque échange. Ceci nécessite de garder un ensemble d'identifiants non utilisés pour les associer pendant le dialogue. Le tableau suivant résume l'association des identifiants aux nœuds et relations des graphes.

Ressource	**Valeurs des identifiants**
Nœuds de l'ontologie	[1-61]
Relations de l'ontologie	[1-13]
Nœuds créés pendant un dialogue	[62-256]

Table 2.13 – *Tableau des identifiants*

2.6 L'agent de dialogue

Nous avons proposé dans la partie conception 1.5.4 deux architectures pour entraîner l'agent de dialogue. La première se compose de deux parties, un module pour encoder le graphe d'état et un autre pour décider l'action à prendre. Chacun est entraîné séparément. Les deux modules étant des réseaux de neurones, nous avons pensé à une deuxième architecture en les connectant pendant la phase d'apprentissage. Cette connexion permet à l'encodeur de graphe de choisir les parties du dialogue, représentées par le graphe d'état, à mémoriser afin de mieux estimer la fonction de récompense.

2.6.1 Encodeur de graphe

Dans la première méthode, l'entraînement de l'encodeur se fait avec une architecture encodeur-décodeur en passant les triplets du graphe comme entrées et sorties de cette architecture.

2.6.1.1 Implémentation

Le réseau de neurones a été implémenté en utilisant la bibliothèque Keras de Python. Nous avons utilisé des cellules GRUs (Gated Reccurent Units (Cho et al., 2014)) comme unités récurrentes pour l'encodeur et le décodeur vu leur efficacité comparable aux LSTMs tout en utilisant un seul vecteur d'état, ce qui les rend moins exigeants en mémoire. Comme notre tâche consiste à encoder un graphe dans un vecteur, la taille de ce dernier est très importante. Intuitivement, plus cette taille est grande plus le nombre de triplets qu'on peut y encoder est grand. D'où l'intérêt des cellules GRUs qui permettent d'utiliser de plus grands vecteurs d'état en moins d'espace mémoire que les LSTMs.

19. Les URIs identifient de manière unique les ressources dans le web sémantique et sont également utilisés pour identifier les concepts et les relations dans les ontologies

La génération aléatoire des graphes de taille t triplets se fait en suivant les étapes suivantes :

- choisir un nombre de nœuds nn aléatoire entre 2 et $t + 1$.
- choisir un nombre d'arcs na aléatoire entre 1 et $nn - 1$.
- choisir nn identifiants de nœuds et na identifiants d'arcs aléatoirement de l'ensemble des identifiants possibles.
- créer les triplets en choisissant pour chaque triplet deux nœuds et un arc aléatoirement des ensembles résultats de l'étape précédente.

Pendant l'apprentissage, le générateur choisit une taille pour le graphe inférieure à une taille maximale et crée un graphe en suivant les étapes sus-citées. Ce dernier est passé à l'encodeur-décodeur comme entrée et sortie désirée.

2.6.1.2 Résultats et discussion

Pour estimer la capacité de l'encodeur du graphe, nous avons varié la taille maximale du graphe ainsi que le vecteur d'état du GRU. Les résultats sont présentés dans le figure suivante.

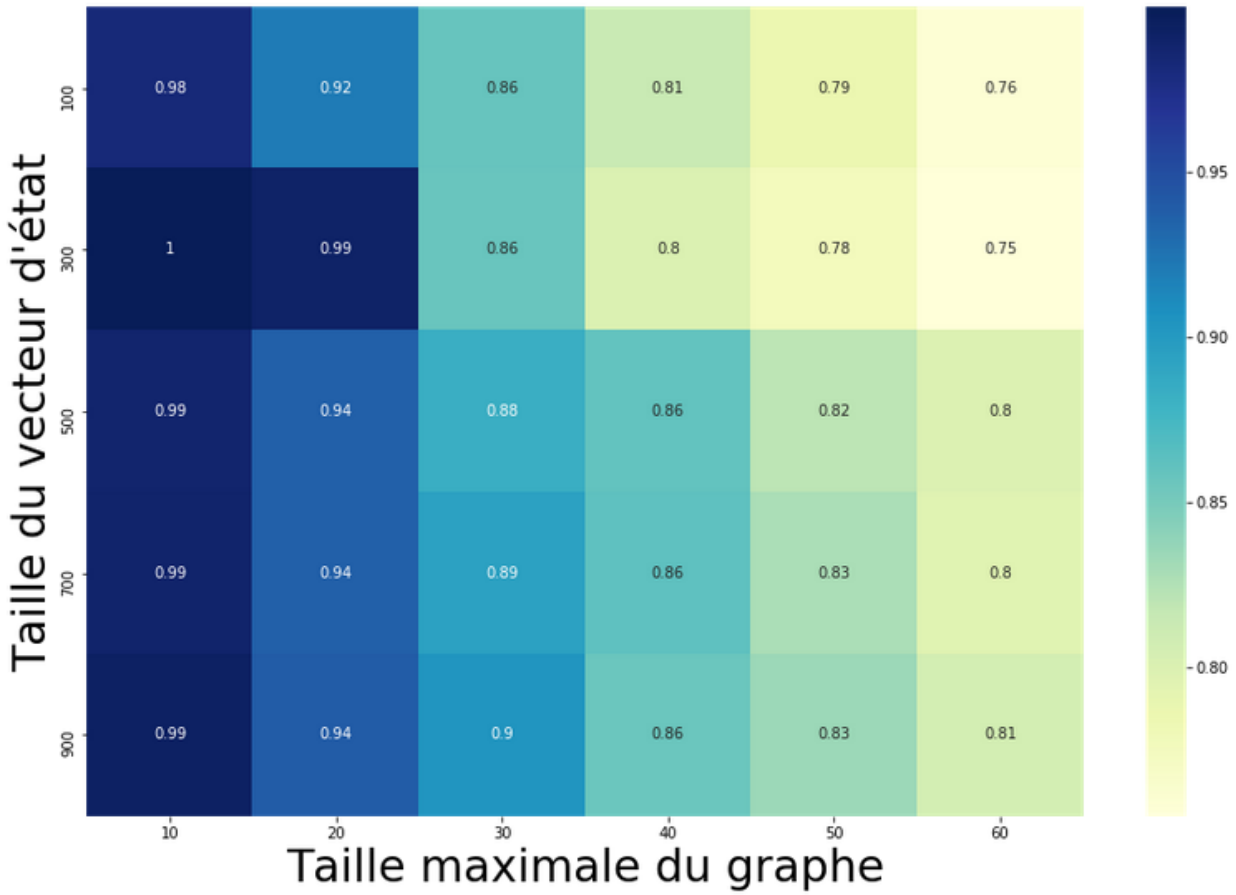

Figure 2.6 – *Variation de la précision en fonction de la taille maximale du graphe et la taille du vecteur d'état*

Nous remarquons évidemment que la précision diminue avec l'augmentation de la taille maximale du graphe. Cependant, l'augmentation de la taille du vecteur encodant le graphe n'améliore pas beaucoup les résultats. Ceci peut être dû à la nature combinatoire du problème. En effet, la taille du graphe augmente exponentiellement en fonction du nombre de triplets. nb et t représentent respectivement le nombre de triplets possibles et la fonction correspondant à la taille du graphe le nombre de graphes possible. t est définie par la fonction de récurrence suivante :

$$t(1) = nb$$

$$t(n+1) = t(n) \times nb$$

La première équation montre que le nombre de graphes possibles est égale au nombre de triplets possibles lorsque le graphe se compose d'un seul triplet. Tandis que la deuxième vient du fait que les graphes de taille $n+1$ peuvent être obtenus en combinant chaque graphe de taille n avec tous les triplets possibles. Le résultat des deux équations est :

$$t(n) = nb^n$$

Pour pallier à ce problème, le graphe d'état peut être divisé en plusieurs sous-graphes, de façon à ce que chaque sous-graphe soit assez petit pour être encodé par l'un des encodeurs déjà entraînés. Ainsi, la concaténation de ces vecteurs peut être utilisée comme entrée pour le module suivant.

2.6.2 Apprentissage par renforcement

Nous allons à présent présenter les deux méthodes citées dans la section 2.6. La première consiste à utiliser le résultat de l'encodeur pour entraîner un réseau de neurones à estimer la récompense associée à chaque action possible. La deuxième, par contre, fait l'apprentissage des deux parties à la fois.

Nous utilisons un simulateur d'utilisateur pour communiquer avec l'agent de dialogue. Afin de simuler aussi les erreurs des modules précédents (reconnaissance automatique de la parole et compréhension du langage naturel), nous ajoutons un module qui insère du bruit dans les actions du simulateur. Sachant qu'une action utilisateur se compose de l'intention et des emplacements, ce dernier module prend donc deux valeurs de probabilités en paramètre ; la probabilité de bruiter l'intention et celle de bruiter les emplacements. Dans la suite de ce travail, nous allons varier ces deux valeurs ainsi que la taille du vecteur d'état afin d'arriver à un compromis entre la robustesse face aux erreurs, la réussite des tâches utilisateur et l'utilisation de la mémoire.

Nous allons évaluer les différents résultats en comparant leur taux de succès qui est donné par le rapport entre le nombre de fois où l'agent arrive au but sur le nombre total d'essais. Un succès nécessite d'arriver au but dans un nombre d'échanges limite que nous avons estimé comme suit :

$$limite = nb_{diff} \times 4 \times (1 + (pb_i + pb_e) \times 2)$$

- limite : le nombre d'échanges maximal.

- nb_{diff} : le nombre de fichiers dans l'arborescence de départ en plus ou en moins par rapport à l'arborescence but.
- pb_i : la probabilité d'erreurs dans l'intention de l'utilisateur.
- pb_e : la probabilité d'erreurs dans les emplacements de l'action.

La première partie de l'équation, $nb_{diff} \times 4$, détermine une approximation du nombre d'actions nécessaire afin d'arriver au but ; nous avons estimé qu'il faut un maximum de 4 échanges pour ajouter ou supprimer un fichier. La deuxième partie permet de prendre en compte les erreurs du simulateur. Sachant que le nombre moyen d'actions erronées en n échanges est égal à $n \times (pb_i + pb_e)$, nous avons donc ajouté deux fois ce nombre. Ceci permet en premier lieu de ne pas compter les actions erronées dans la limite des échanges possibles, ainsi que d'ajouter des actions afin que l'agent puisse se retrouver dans la conversation après une erreur du simulateur.

2.6.2.1 Apprentissage avec DQN déconnecté

Dans cette partie, nous utilisons un des encodeurs déjà entraînés dans la partie précédente. Nous avons donc choisi d'utiliser l'encodeur de taille 300 qui a été entraîné avec des graphes de taille maximale de 20. Nous avons varié le nombre de vecteurs utilisés ainsi que les probabilités d'erreurs du simulateur. Les résultats sont présentés dans le tableau 2.14.

Nombre de vecteurs	**probabilité pb_i**	**probabilité pb_e**	**taux de réussite maximal**
4	0.25	0.3	0.25
4	0.05	0.2	0.59
4	0.01	0.1	0.69
4	0	0	0.61
5	0.25	0.3	0.27
5	0.05	0.2	0.55
5	0.01	0.1	0.61
5	**0**	**0**	**0.72**
6	0.25	0.3	0.28
6	0.05	0.2	0.63
6	0.01	0.1	0.69
6	0	0	0.7

Table 2.14 – *Taux de réussite en fonction des probabilités d'erreurs et du nombre de vecteurs d'état. Avec les probabilités pb_i et pb_e, les probabilités d'erreurs sur l'intention et les emplacements respectivement.*

Le tableau 2.14 nous permet de conclure ce qui suit :

- Évidemment, avec moins de probabilités d'erreurs, le réseau arrive à mieux reconnaître les motifs dans le vecteur d'état et leurs relations avec les récompenses du simulateur. Cependant, l'augmentation de la taille du vecteur n'améliore que légèrement les résultats ; en calculant la moyenne des taux de réussite pour chaque nombre de vecteurs, les résultats sont comme suit : 53.5% de taux de réussite pour quatre vecteurs, 53.75% pour cinq vecteurs et 57.5% pour six vecteurs.

- Le meilleur résultat a été obtenu en utilisant quatre vecteurs et sans erreurs avec un taux de réussite de 72%. En pratique, ce modèle ne s'adapte pas aux erreurs des modules précédents. Par conséquent, il se perd lorsqu'une erreur se produit et il n'arrive pas à se resituer dans la conversation.
- Avec des probabilités d'erreurs très élevées, le réseau n'arrive pas à apprendre et ne dépasse pas 28% de taux de réussite.
- Les résultats les plus prometteurs, dans ce cas, sont ceux qui sont entraînés avec des probabilités d'erreurs proches de la réalité avec des taux de réussite acceptables. Dans ce cas, nous avons obtenu un taux de réussite de 69% avec des probabilités d'erreurs de 1% et 10% sur les intentions et les emplacements respectivement.

2.6.2.2 Apprentissage avec DQN connecté

Connecter le DQN avec l'encodeur devrait permettre de réduire la taille du vecteur d'état nécessaire. Comme pour la partie précédente, nous avons varié la taille du vecteur d'état ainsi que les probabilités des erreurs pour pouvoir par la suite comparer les deux approches proposées.

Taille du vecteur d'état	**probabilité pb_i**	**probabilité pb_e**	**taux de réussite maximal**
50	0.25	0.3	0.69
50	0.05	0.2	0.77
50	0.01	0.1	0.80
50	0	0	0.87
100	0.25	0.3	0.59
100	0.05	0.2	0.81
100	0.01	0.1	0.84
100	0	0	0.89
150	0.25	0.3	0.58
150	0.05	0.2	0.82
150	0.01	0.1	0.80
150	**0**	**0**	**0.93**
200	0.25	0.3	0.59
200	0.05	0.2	0.82
200	0.01	0.1	0.85
200	0	0	0.86

Table 2.15 – *Taux de réussite en fonction des probabilités d'erreurs et de la taille du vecteur d'état. Avec les probabilités pb_i et pb_e, les probabilités d'erreurs sur l'intention et les emplacements respectivement.*

- Comme pour la méthode précédente, l'augmentation des probabilités d'erreurs diminue le taux de réussite en général.
- Le meilleur taux de réussite est toujours obtenu en faisant un apprentissage sans erreurs ce qui a donné 93% de réussite.

- Pour des probabilités d'erreurs élevées, le taux de réussite diminue considérablement par rapport aux autres valeurs de probabilités.
- Le réseau arrive à reconnaître les motifs encodés aussi bien pour de petites tailles du vecteur d'état que pour de grandes tailles. Cette méthode permet effectivement de réduire la taille du vecteur encodant le graphe d'état par rapport à son prédécesseur.

2.6.2.3 Comparaison des approches et discussion

En comparant les deux méthodes, apprentissage avec DQN déconnecté et apprentissage avec DQN connecté, il est évident que la deuxième méthode est beaucoup plus efficace. Dans cette partie, nous allons donner de potentielles explications aux résultats trouvés en analysant les forces et faiblesses de chaque méthode. Nous allons analyser les méthodes par rapport à trois aspects : le taux de réussite, la vitesse d'apprentissage et la courbe d'apprentissage.

2.6.2.4 Taux de réussite

Il est clair que connecter l'encodeur avec le réseau DQN a donné de meilleurs résultats. Ceci peut être dû aux facteurs suivants :

- La difficulté de comprendre les motifs encodés séparément : En effet, la deuxième méthode permet au réseau de neurones d'apprendre à générer des motifs qui correspondent aux poids du réseau DQN.
- Deux états de dialogue lointains peuvent être encodés dans des vecteurs très proches en utilisant un encodeur séparé. Par exemple, il se peut que deux graphes soient similaires sauf pour un nœud d'action qui est dans un graphe de type **Create_node** et dans l'autre graphe **Delete_node**. Dans ce cas, si l'encodeur les encode dans des vecteurs proches, le DQN aura des difficultés à distinguer les deux états. Par contre, en connectant l'encodeur avec le DQN, il peut observer les récompenses pour avoir l'information que certains nœuds sont plus importants que d'autres, les nœuds d'action par exemple, et qu'ils peuvent changer complètement l'état du vecteur.
- Les erreurs provenant de l'encodeur : Bien que la précision de l'encodeur obtenue était de 99%, en empilant 6 vecteurs encodés avec le même taux d'erreurs pour chacun et sachant que la probabilité d'erreur dans un vecteur est indépendante des autres, cette précision diminue à 96%.

2.6.2.5 Vitesse d'apprentissage

Nous avons comparé les vitesses d'apprentissage des deux approches. En utilisant le réseau DQN séparé, ce dernier apprend avec une moyenne de **1013.15 instances/seconde**. De l'autre côté, le réseau connecté ne fait que **197.61 instances/seconde**. La raison derrière la lenteur de la deuxième approche revient à la nécéssité de refaire l'encodage de tout le graphe d'état de l'instance sauvegardée pendant les échanges avec le simulateur. Tandis que dans la deuxième approche, on ne sauvegarde que le vecteur déjà encodé.

Les temps des échanges avec le simulateur des deux approches sont proches puisqu'ils se comportent de la même manière dans ce cas. Les deux méthodes n'encodent que les nouveaux triplets arrivant dans le vecteur d'état précédent.

2.6.2.6 Courbe d'apprentissage

Enfin, nous comparons à présent les courbes d'apprentissage des deux méthodes. Celles-ci montrent le taux de réussite par rapport au nombre d'épisodes [20]. La figure 2.7 contient quatre courbes : deux courbes pour chaque méthode, avec et sans erreurs.

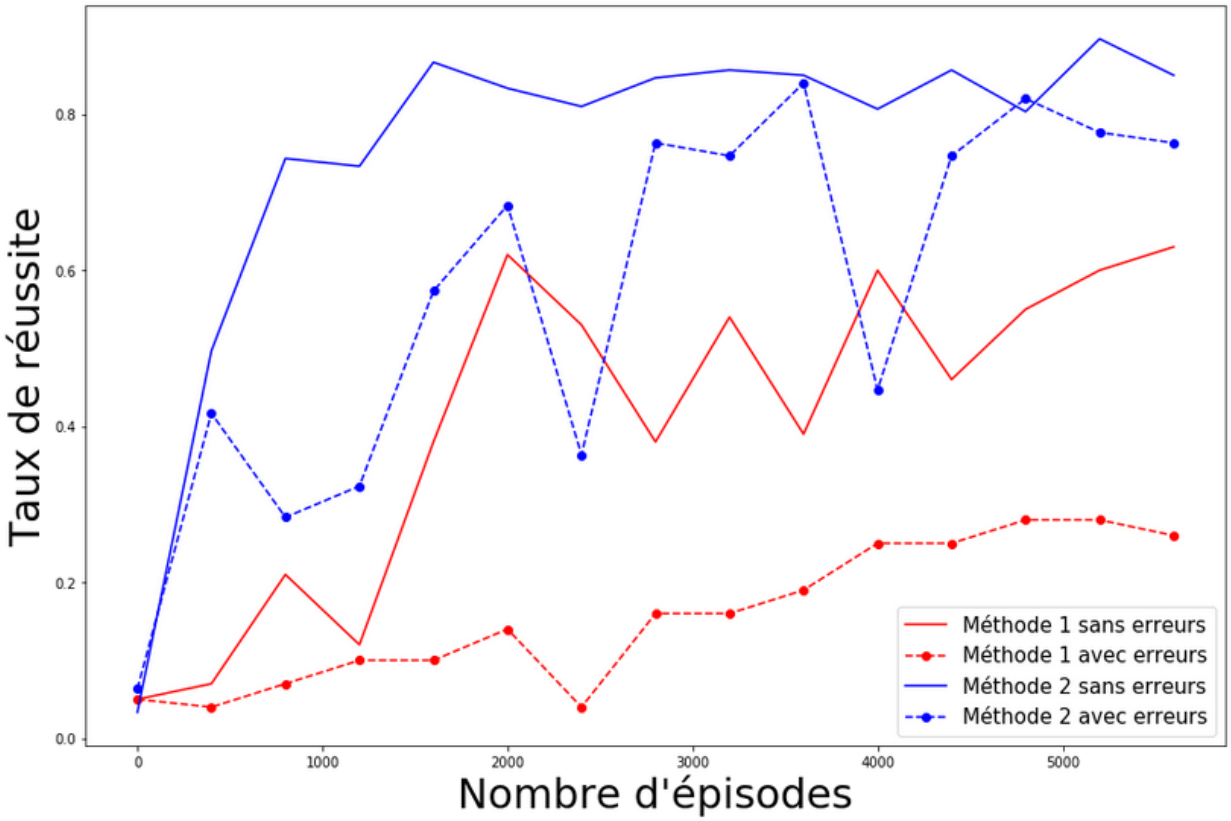

Figure 2.7 – *Les courbes d'apprentissage des deux méthodes proposées avec et sans erreurs du simulateur d'utilisateur*

Nous remarquons que la deuxième méthode arrive beaucoup plus rapidement à comprendre la fonction de récompense. Il s'avère donc que déconnecter le DQN de l'encodeur rend effectivement la tâche de trouver la relation entre les motifs des vecteurs d'états et les récompenses plus difficile.

L'apprentissage peut être amélioré dans les deux cas, et surtout en ce qui concerne le premier, en utilisant un meilleur encodage des nœuds et des arcs du graphe. En effet, les nœuds sont actuellement encodés avec des identifiants entiers seulement, perdant ainsi les riches connaissances sémantiques qu'on peut extraire des relations du graphe. Un

20. Un épisode est un ensemble d'échanges agent-simulateur qui aboutit à un succès ou un échec

meilleur encodage serait d'utiliser des méthodes d'apprentissage semi-supervisé qui permettraient de donner aux nœuds proches un encodage similaire. Cette méthode permet d'éviter quelques problèmes que nous avons cités dans 2.6.2.4, entre autres le problème d'encoder deux états lointains dans des vecteurs proches.

2.7 Application Speech2Act

L'application Speech2Act relie les modules que nous avons implémentés dans un assistant qui aide à manipuler les fichiers de l'ordinateur avec la voix. Le schéma 2.8 montre les différentes parties que comporte l'application et les communications entre elles.

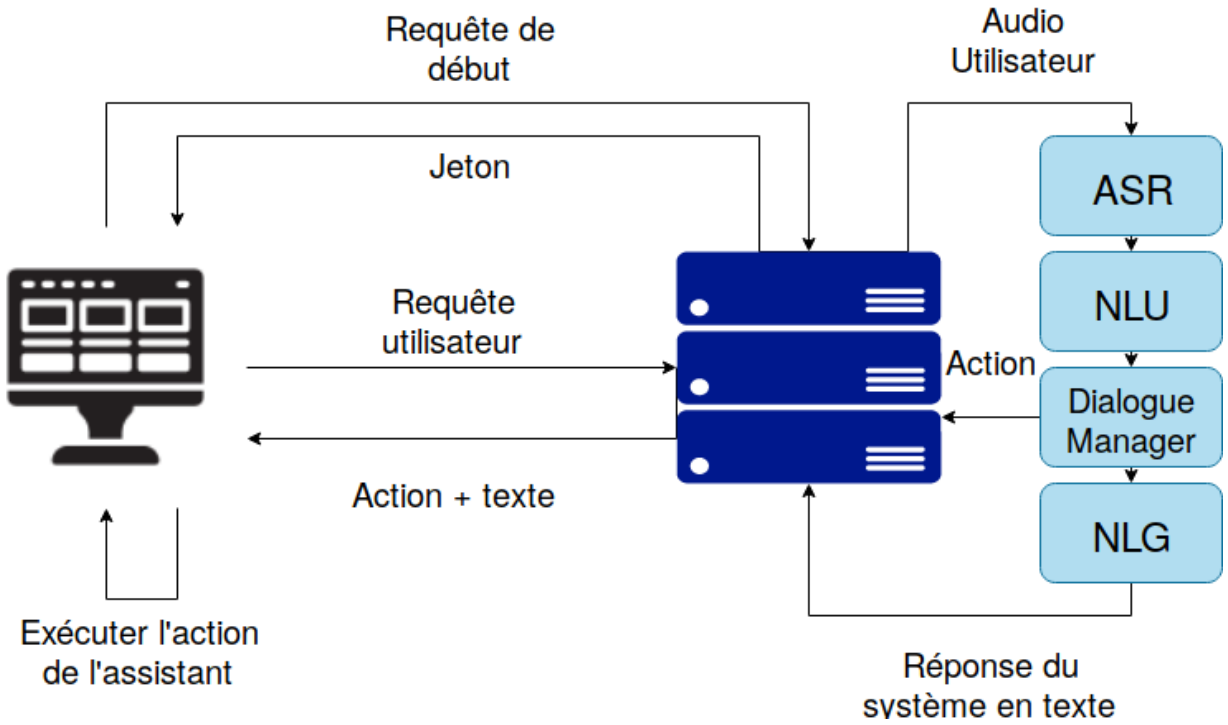

Figure 2.8 – *Schéma général de l'application Speech2Act*

L'application contient principalement deux parties : une partie frontend, avec laquelle l'utilisateur interagit, et une partie backend qui contient tous les modules que nous avons traités au cours de ce travail. Cette dernière partie se trouve dans un serveur qui répond aux requêtes de l'utilisateur. Les requêtes de l'utilisateur contiennent les données audio collectées par l'interface ainsi que les données du système sur lesquelles l'assistant peut agir. Quant-à la réponse du backend, elle est sous forme d'une action qui peut être exécutée par le côté client de l'application. Nous n'avons cependant pas encore traité les cas de permissions et les limites de ce que peut manipuler l'assistant. Néanmoins, nous avons limité l'espace des actions de l'assistant pour qu'il ne puisse agir que sur une arborescence de fichier test.

2.7.1 Backend

Pour implémenter le backend, nous avons utilisé le micro-framework Flask qui permet d'écrire en Python le côté serveur d'une application. Les communications client-serveur de notre application se font comme suit :

- Au lancement de l'application du côté client, celle-ci envoie une requête de début contenant l'état du système, dans notre cas une arborescence de fichiers.
- Le backend lui répond avec un token (jeton) qui sera l'identifiant de cet utilisateur.
- Lorsque l'utilisateur parle à l'assistant, une requête est envoyée contenant son enregistrement audio. Alternativement, l'utilisateur peut introduire directement du texte qui sera envoyé dans la requête.
- Le backend reçoit le contenu de la requête. S'il s'agit d'un enregistrement audio, il le fait passer par le module de reconnaissance de la parole pour le convertir en texte.
- Le texte passe ensuite par le module de compréhension du langage qui est directement connecté avec le gestionnaire de dialogue. Ce dernier reçoit l'action résultat du module précédent et décide quelle action prendre selon l'état du système de l'utilisateur en question.
- L'action de l'assistant est transformée en langage naturel avant qu'elle ne soit envoyée à l'utilisateur.
- Le côté client de l'application reçoit le texte et l'action de l'assistant. Il exécute l'action et affiche le texte à l'utilisateur.

2.7.2 Frontend

Pour la réalisation de l'interface de l'application, nous avons opté pour une interface basée web (facilement exportable vers Desktop). Pour cela nous avons utilisé le framework VueJS augmenté par le plugin Vuetify. le résultat est une interface épurée et qui se veut simple et légère. L'utilisation d'un framework basé web permet de facilement créer des boucles d'éventements dont l'état interne est géré entièrement par le navigateur et le moteur VueJS.

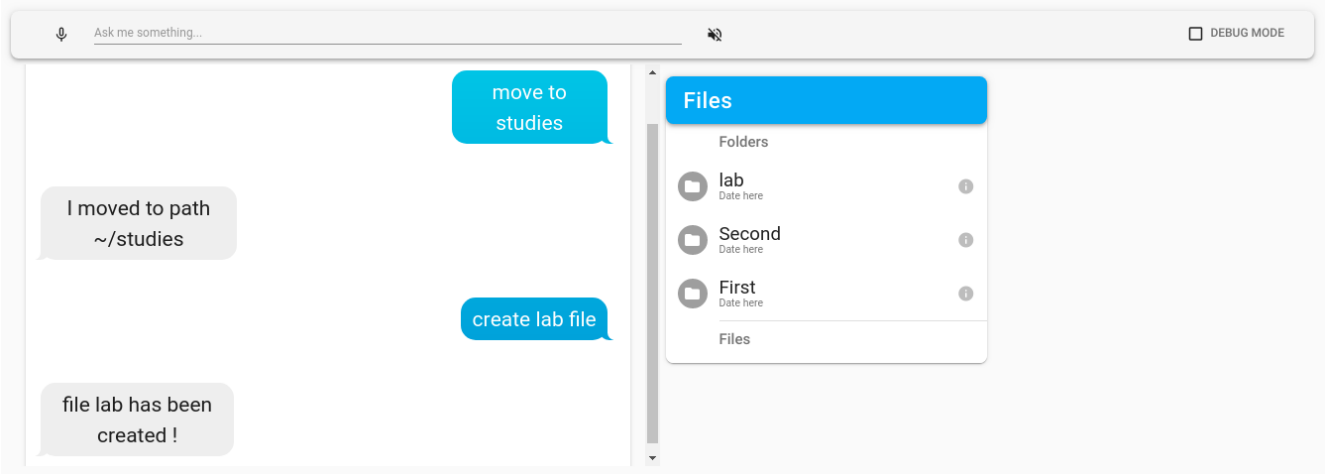

Figure 2.9 – *Interface principale de Speech2Act*

L’application se compose de quatre parties disposées sur une seule page web dynamiquement mise à jour au fur et à mesure du dialogue :

- **Champ de saisie** : C’est une petite surface qui permet à l’utilisateur de communiquer avec le système. Il lui est possible bien entendu d’utiliser du texte ou bien de cliquer sur le bouton Microphone à gauche pour lancer des commandes vocales. Pour l’instant, c’est à l’utilisateur d’arrêter l’enregistrement de la commande ou bien d’attendre une période limite fixée à 7 secondes. Le bouton de volume permet d’activer ou non la réponse vocale du système (extensions de synthèse vocale). Le bouton tout à droite est réservé au mode Développeur pour analyser l’interaction entre le système et l’utilisateur.
- **Arborescence virtuelle** : C’est une petite fenêtre pour faciliter à l’utilisateur la mémorisation de l’environnement d’interaction. Cette fenêtre est dynamiquement mise à jour à travers le déroulement du dialogue et la manipulation des fichiers/répertoires.

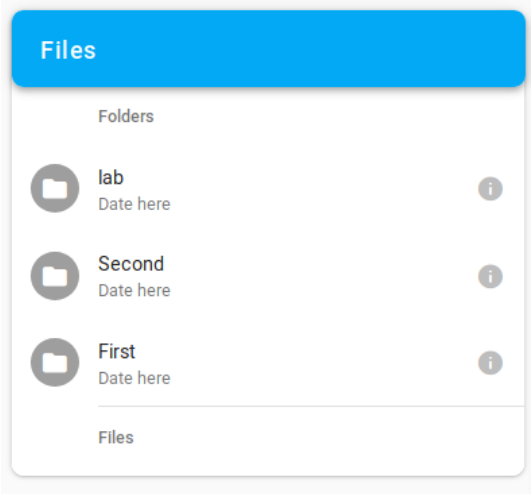

Figure 2.10 – *Fenêtre d’affichage de l’arborescence virtuelle*

- **Champ de dialogue** : C’est là que vont résider toutes les informations sur le dialogue tout au long du cycle de vie de l’application. Il est mis à jour à chaque échange entre l’utilisateur et l’assistant Speech2Act. Il permet ainsi de garder trace de tous les échanges effectués et pouvoir à tout moment vérifier ou réutiliser des messages.

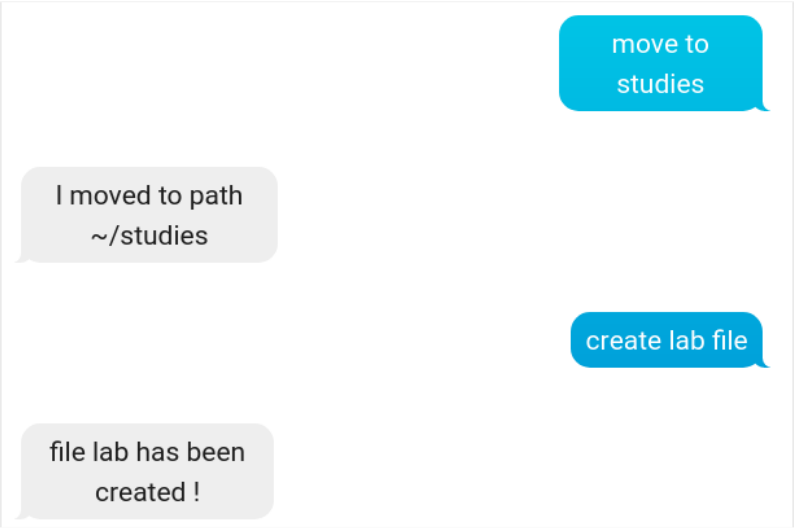

Figure 2.11 – *Fenêtre de dialogue avec l'assistant*

- **Fenêtre de Débogage** : C'est une option réservée aux développeurs pour faciliter les tests durant le développement. Toute réponse du serveur peut y être affichée pour peu qu'elle soit au format JSON sous la forme d'une réponse à une requête RESTFul. Pour le moment, il y est affiché l'intention de l'utilisateur avec son degré de confiance généré par Speech2Act, ainsi que les arguments de la requête (nom, type, positions ...).

Figure 2.12 – *Fenêtre de Débogage*

2.8 Conclusion

Au terme de ce chapitre, nous avons pu apprécier les fruits d'un long travail de conception. L'implémentation de certaines fonctionnalités a permis de mieux apprécier la complexité de la tâche qu'est le développement de Speech2Act. Chaque module a été soumis à une série de tests pour déterminer ses forces, faiblesses et limites. Une rapide analyse stipulant que pour un manque de données flagrant et un manque de ressources frustrant, Speech2Act a pu effectuer des petites tâches rudimentaires de manipulation du bureau sur un ordinateur et délivrer une mini-expérience de ce que peut être un véritable assistant virtuel intelligent.

Table des figures

Bibliographie

Bocklisch, T., Faulkner, J., Pawlowski, N., and Nichol, A. (2017). Rasa : Open source language understanding and dialogue management. *CoRR*, abs/1712.05181.

Cho, K., van Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar. Association for Computational Linguistics.

Goo, C.-W., Gao, G., Hsu, Y.-K., Huo, C.-L., Chen, T.-C., Hsu, K.-W., and Chen, Y.-N. (2018). Slot-gated modeling for joint slot filling and intent prediction. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics : Human Language Technologies, Volume 2 (Short Papers)*, New Orleans, Louisiana. Association for Computational Linguistics.

Hannun, A. Y., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., Prenger, R., Satheesh, S., Sengupta, S., Coates, A., and Ng, A. Y. (2014). Deep speech : Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567.

Kipf, T. N. and Welling, M. (2017). Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*.

Knote, R., Janson, A., Eigenbrod, L., and Söllner, M. (2018). The what and how of smart personal assistants : Principles and application domains for is research. In *Multikonferenz Wirtschaftsinformatik (MKWI)*, pages 1083–1094. Lüneburg, Germany.

Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. S. (2016). Gated graph sequence neural networks. *CoRR*, abs/1511.05493.

Liu, B. and Lane, I. (2016). Attention-based recurrent neural network models for joint intent detection and slot filling. *CoRR*, abs/1609.01454.

Marcus, M., Kim, G., Marcinkiewicz, M. A., MacIntyre, R., Bies, A., Ferguson, M., Katz, K., and Schasberger, B. (1994). The penn treebank : Annotating predicate argument structure. In *Proceedings of the Workshop on Human Language Technology*, pages 114–119, Stroudsburg, PA, USA. Association for Computational Linguistics.

Milhorat, P., Schlögl, S., Chollet, G., , B., Esposito, A., and Pelosi, G. (2014). Building the next generation of personal digital assistants.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518.

Rummery, G. A. and Niranjan, M. (1994). On-line Q-learning using connectionist systems. Technical Report TR 166, Cambridge University Engineering Department, Cambridge, England.

Schatzmann, J., Thomson, B., Weilhammer, K., Ye, H., and Young, S. J. (2007). Agenda-based user simulation for bootstrapping a pomdp dialogue system. In *HLT-NAACL*.

Stoyanchev, S. and Johnston, M. (2018). Knowledge-graph driven information state approach to dialog. In *AAAI Workshops*.

Trappl, R. (2013). *Your Virtual Butler*. Springer Berlin Heidelberg.

Tulshan, A. and Namdeorao Dhage, S. (2019). *Survey on Virtual Assistant : Google Assistant, Siri, Cortana, Alexa : 4th International Symposium SIRS 2018, Bangalore, India, September 1922, 2018, Revised Selected Papers*, pages 190–201.

VanDijck, J. (2005). From shoebox to performative agent : the computer as personal memory machine. *New Media & Society*, 7:311–332.

Wessel, M., Acharya, G., Carpenter, J., and Yin, M. (2019). *OntoVPAAn Ontology-Based Dialogue Management System for Virtual Personal Assistants : 8th International Workshop on Spoken Dialog Systems*.