

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des Sciences et de la Technologie Houari Boumediene
Faculté d'électronique et d'informatique
Département d'informatique



Mémoire de Master

Domaine : Informatique

Spécialité : Systèmes Informatiques Intelligents

Thème

Exploration de la reconnaissance de la parole pour le développement d'aspects d'un assistant de manipulation de l'ordinateur

Présenté par :

- BOURAHLA Yasser
- BENHADDAD Wissam

Proposé et dirigé par :

- Pr. GUESSOUUM Ahmed
- Pr. AKLI ASTOUATI Karima

Devant le jury composé de :

- | | |
|-------------------|------------|
| - Mme. F. KHELLAF | Présidente |
| - Mme. I. AMROUS | Membre |

Projet N° : 061/2019

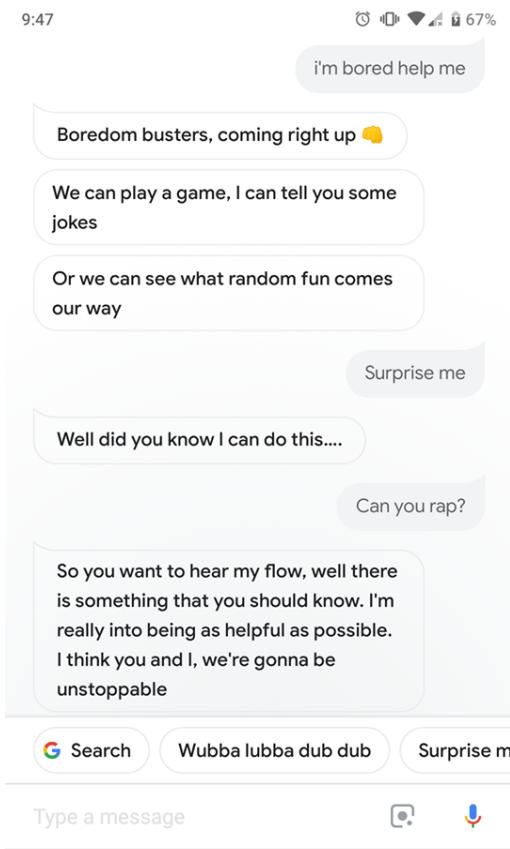


Figure 1.1 – Conversation aléatoire avec Google Assistant

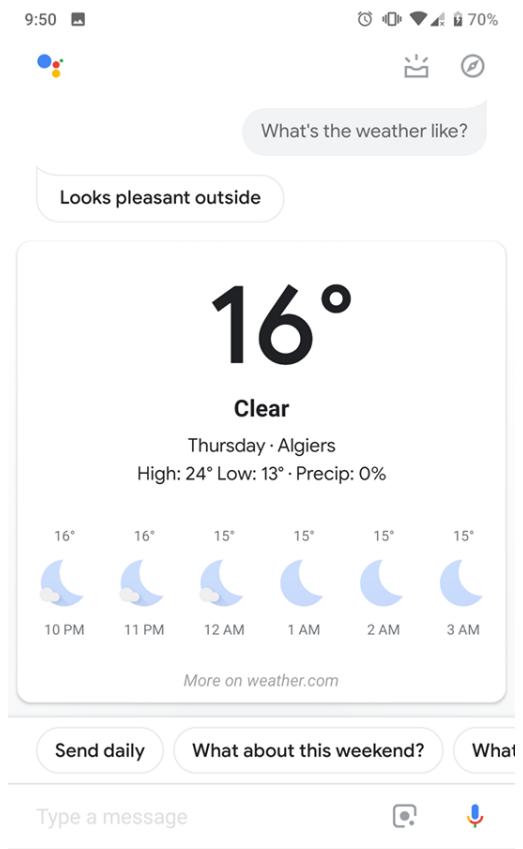


Figure 1.2 – Requête simple formulée à Google Assitant

1.5.2 Apple Siri

Siri est le SPA développé par Apple. Contrairement aux autres assistants virtuels qui existaient avant sa sortie, Siri proposait une nouvelle façon de communiquer avec l'utilisateur, à travers une interface de requêtes vocales, et une façon d'entretenir une conversation très humanoïde, satisfaisant ainsi le critère d'anthropomorphisme vu dans la section 1.3.4. Siri est capable de proposer des recommandations, déléguer la requête à des services web ou d'autres applications (voir figures 1.3), de répondre à des questions précises (voir figure 1.4). Il a l'avantage d'être uniquement disponible que sur les appareils qui composent l'écosystème d'Apple (MacBook, iPhone, iWatch, etc). Ceci assure une prise en charge et un support total de la part d'Apple. Cependant, pour utiliser l'ensemble de ses fonctionnalités, Apple restreint la liste des appareils compatibles à ceux que la compagnie fabrique uniquement. Il est ainsi quasi mono-plateforme et difficile à utiliser sur des alternatives aux produits de la firme.



Figure 1.3 – *Intégration aux applications (Zibreg, 2016)*

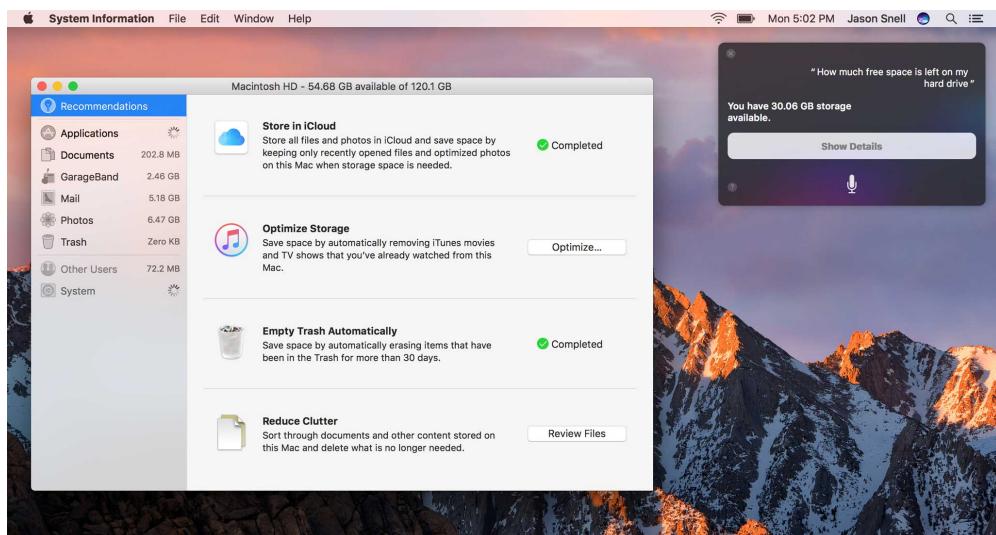


Figure 1.4 – *Siri sur un laptop (Snell, 2016)*

1.5.3 Amazon Alexa

Amazon Alexa est un assistant exclusivement intégré au dispositif Amazon Echo, un haut-parleur portatif. À l'instar de Siri, il est capable de communiquer avec l'utilisateur par le biais de la parole, pouvant ainsi exécuter diverses commandes, comme jouer de la musique, réciter des livres audios, annoncer des news en temps réel (résultats sportifs, tendances politiques, etc.). Son atout majeur est sa capacité à s'intégrer à plusieurs appareils-connectés (contrôleur de thermostat ou de lumières ambiante dans une Smart-House) ainsi que la possibilité d'ajout des Skills (ou compétences) de la part des développeurs tiers pour enrichir la panoplie de services que peut offrir Alexa.

d'atteindre le but final du dialogue (exécuter une commande système, trouver des informations internet ou sur la machine locale, etc.). Le gestionnaire du dialogue communique avec son environnement en utilisant la même représentation sémantique produite par l'analyse précédente.

- Puisqu'il est préférable de cacher à l'utilisateur tout comportement interne au système, il serait préférable de transformer la trame sémantique (voir la figure 2.9) en texte naturel pour l'afficher en sortie.

Il est à noter que chacun des modules seront indépendants pour ce qu'il est de leur fonctionnement interne. Ainsi, aucun n'assumera un fonctionnement arbitraire des autres modules. Seul le format des informations qui circulent entre eux devra être établi au préalable pour un fonctionnement durable et robuste au changement.

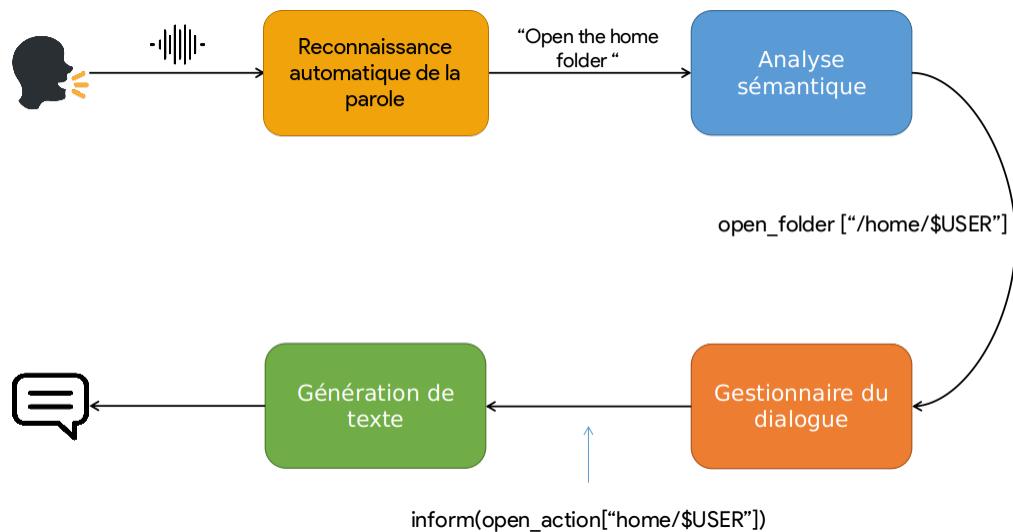


Figure 2.1 – Schéma abstractif d'un SPA (Milhorat et al., 2014)

Pour ce qui est de la suite du chapitre, nous allons présenter les différents modules ainsi que les techniques et architectures qui sont considérées comme étant l'état de l'art du domaine.

2.3 Notions et aspects théoriques

Dans cette section, nous présenterons différentes notions liées au domaine de l'apprentissage automatique, pour ensuite les citer dans les modules qui les utilisent.

2.3.1 Apprentissage automatique

Le plus souvent, un domaine scientifique peut être défini à travers le ou les types de problèmes dont il essaye de trouver une solution. D'après (Mitchell, 2006), l'auteur définit ce problème sous forme d'une question :

domaines comme la reconnaissance d'images (Szegedy et al., 2015), la reconnaissance automatique de la parole (Graves et al., 2013; Yu and Deng, 2015) ou encore la classification de textes (Velay and Daniel, 2018 ; Liu et al., 2016).

Il existe une variété d'architectures de réseaux de neurones. Nous traiterons dans cette section trois d'entre elles :

- Réseaux de neurones multicouches denses
- Réseaux de neurones profonds
- Réseaux de neurones récurrents et leurs variantes

2.3.2.1 Réseaux de neurones multicouches denses

La forme la plus basique que peut avoir un réseau de neurones est celle d'un réseau multicouches comme montré dans la figure 2.2. Elle se compose de trois parties :

- **Une couche d'entrée** : Elle reçoit l'information en entrée codifiée en un vecteur numérique χ .
- **Une ou plusieurs couches cachées** : le cœur du réseau, c'est une succession de couches de neurones où chaque couche C_i reçoit un signal sous forme d'une ou plusieurs valeurs numériques depuis une couche antérieure C_{i-1} ou bien la couche d'entrée χ , puis envoie en sortie un autre signal de même nature qui est une combinaison non-linéaire du signal en entrée vers une couche cachée suivante C_{i+1} ou bien la couche de sortie.
- **Une couche de sortie** : elle permet de calculer une valeur \hat{y} qui peut être vue comme la prédiction du modèle Φ par rapport à son entrée χ :

$$\hat{y} = f_{\Phi}(\chi) \quad (2.1)$$

Le réseau calcule une erreur $e(y, \hat{y})$ en fonction de sa valeur en sortie et de la valeur exacte puis corrigera cette erreur au fur et à mesure du parcours des données d'apprentissage (Murtagh, 1991).

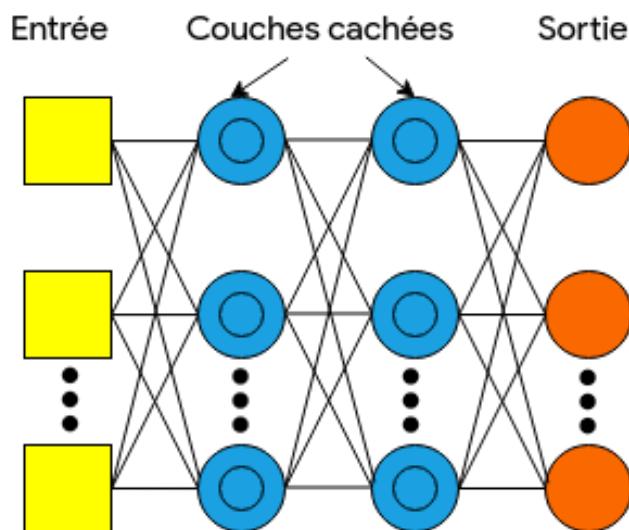


Figure 2.2 – Architecture basique d'un réseau de neurones multicouches

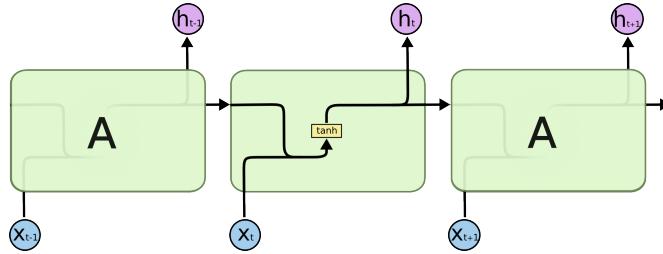


Figure 2.3 – Architecture interne d'un réseau de neurones récurrent à un instant t (Olah, 2015)

Réseaux de neurones récurrents à mémoire à court et long terme (LSTM)

Introduite en 1997 dans (Hochreiter and Schmidhuber, 1997), cette architecture de réseaux de neurones récurrents est dotée d'un système de portes qui filtrent l'information qui y passe ainsi que d'un état interne de la cellule mémoire d'un réseau LSTM. Les composants d'une cellule LSTM sont détaillés dans la figure 2.4.

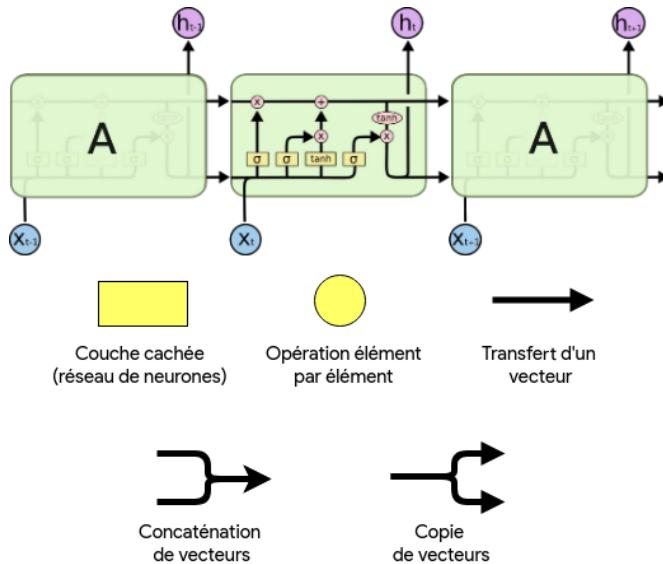


Figure 2.4 – Architecture interne d'une cellule mémoire dans un réseau LSTM (Olah, 2015)

- **Porte d'entrée (Input gate) :** C'est une unité de calcul qui laisse passer certaines informations en entrée en utilisant une fonction d'activation sigmoïde pour pondérer les composants du vecteur d'entrée à une étape t et celles du vecteur d'état interne à l'étape $t - 1$ (1 : laisser passer, 0 : ne pas laisser passer) et ainsi générer un vecteur candidat \tilde{C}_t (Hochreiter and Schmidhuber, 1997).

$$\begin{aligned} i_t &= \sigma(W_i \times [h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= \tanh(W_C \times [h_{t-1}, x_t] + b_C) \end{aligned} \tag{2.4}$$

où :

- h_{t-1} et x_t sont respectivement le vecteur d'état caché à l'étape $t - 1$ et le vecteur de données en entrée à l'étape t .
- σ est la fonction Sigmoïde.

2.3.3 Modèles de Markov cachés (Hidden Markov Models HMM)

Informellement, un modèle de Markov caché (HMM) est un outil de représentation pour modéliser la distribution de probabilité d'une séquence d'observations. Il est dit *caché* pour deux raisons. Premièrement, il est supposé qu'une observation O à un instant t (dénotée O_t) est le résultat d'un certain processus (souvent stochastique) dont l'état S_t est caché à l'observateur. Deuxièmement, l'état S_t du processus caché ne dépend uniquement que de son état à l'instant $t - 1$. Il est alors dit que ce processus est Markovien ou qu'il satisfait la propriété de Markov (Ghahramani, 2002 ; Kemeny and Laurie Snell, 1957). La figure 2.5 illustre graphiquement un HMM.

De façon plus formelle, un HMM est un 5-tuple $\langle S, V, \Pi, A, B \rangle$ (Rabiner and Juang, 1986) où :

- $S = \{s_1, \dots, s_N\}$ est l'ensemble fini des N états du processus sous-jacent.
- $V = \{v_1, \dots, v_M\}$ est l'ensemble des M symboles qui constituent un certain vocabulaire.
- $\Pi = \{\pi_i\}$ est une distribution initiale des probabilité d'états tel que π_i est la probabilité de se trouver à l'état i à l'instant $t = 0$, avec $\sum_{i=1}^N \pi_i = 1$.
- $A = \{a_{ij}\}$ est une matrice $N \times N$ dont chaque entrée a_{ij} est la probabilité de transition d'un état i à un état j avec $\sum_{j=1}^N a_{ij} = 1$ pour tout $i = 1, \dots, N$.
- $B = \{b_i(v_k)\}$ est l'ensemble des distributions de probabilités d'émission (ou d'observation), donc $b_i(v_k)$ est la probabilité de générer un symbole v_k du vocabulaire étant donné un certain état i avec $\sum_{k=1}^M b_i(v_k) = 1$ pour tout $i = 1, \dots, N$.

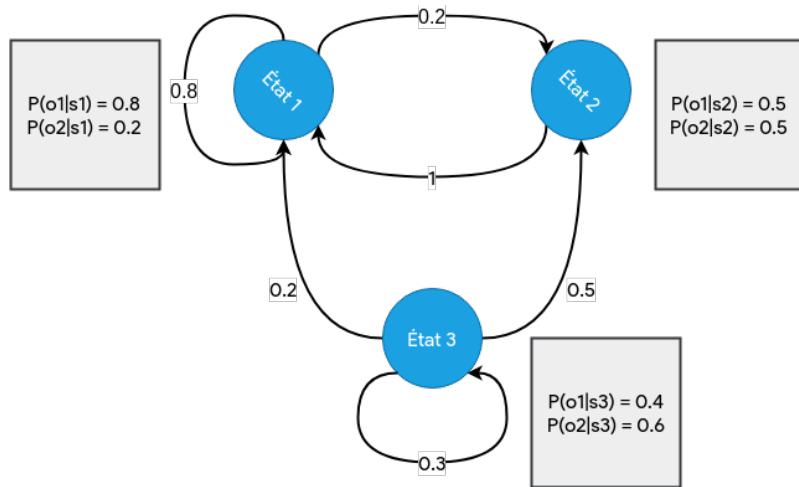


Figure 2.5 – Exemple d'une distribution de probabilités de transitions ainsi qu'une distribution de probabilités d'observations pour un HMM à 3 états et 2 observations

Le but est de trouver la séquence d'états $S_{1:K}$ qui maximise la probabilité (Ghahramani, 2002) :

$$P(Y_{1:K}|O_{1:K}) = P(O_1)P(O_1|S_1) \prod_{t=2}^K P(S_t|S_{t-1})P(O_t|S_t) \quad (2.8)$$

Pour y parvenir d'une manière efficace, un décodeur qui implémente l'algorithme de viterbi peut être utilisé (Forney, 1973 ; Bloit and Rodet, 2008).

2.4 Reconnaissance automatique de la parole (ASR)

Le premier module qui compose le système est celui de la reconnaissance automatique de la parole (ASR). Le but d'un tel système (ou sous-système dans notre cas) est de convertir un signal audio correspondant à la locution d'un utilisateur en un texte qui peut être interprété par la machine (Al-Anzi and AbuZeina, 2018). Différentes approches ont été développées au cours des années. Une architecture s'est ensuite dégagée où les systèmes passent par deux phases : la phase d'apprentissage et la phase de reconnaissance. La première consiste à collecter les données qui constituent le corpus d'apprentissage, un ensemble de fichiers audios avec leurs transcriptions en texte et en phonèmes. Le signal est ensuite traité pour en extraire des vecteurs de caractéristiques (ou attributs). La suite de l'apprentissage consiste à initialiser le HMM, lui passer les vecteurs précédemment extraits puis l'enregistrer pour la phase suivante qui est la reconnaissance. Dans la deuxième phase, le signal audio passe par le même procédé d'extraction des attributs, en utilisant un algorithme de décodage approprié (Bloit and Rodet, 2008), puis la séquence d'observations passe par le HMM et la meilleure séquence de mots est sélectionnée (Yu and Deng, 2015).

Comme le montre la figure 2.6, une architecture assez générale s'est dégagée. Quatre modules sont impliqués dans le processus de la reconnaissance de la parole ; nous les citerons dans les sections qui suivent en énumérant les modèles utilisés dans chacun.

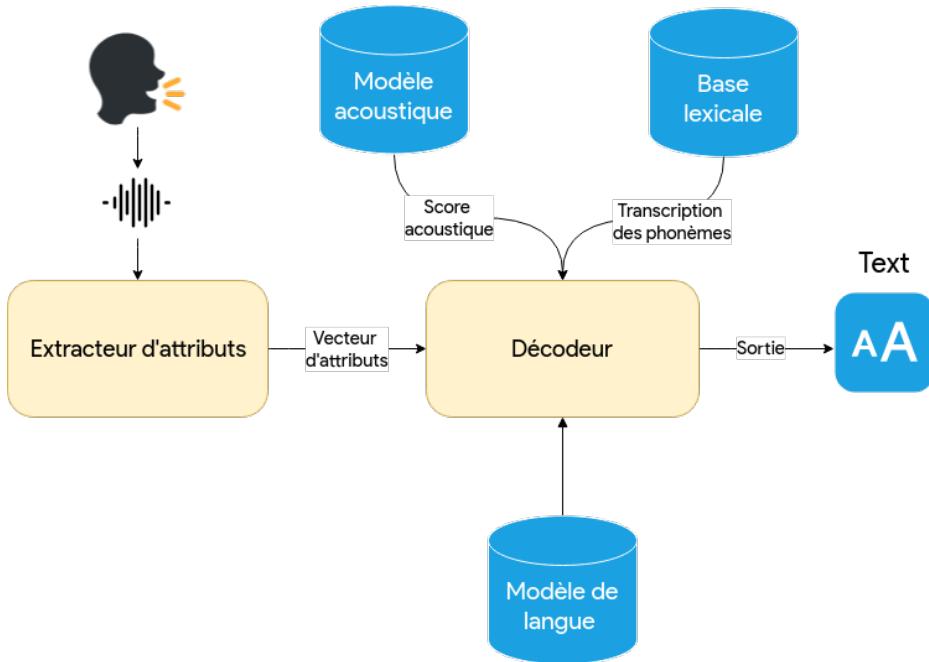


Figure 2.6 – Architecture d'un système de reconnaissance de la parole (Yu and Deng, 2015)

2.4.1 Acquisition du signal et extraction d'attributs

L'étape consiste à extraire une séquence de vecteurs caractéristiques à partir du signal audio, fournissant une représentation compacte de ce dernier. Le processus démarre par la segmentation du signal en trames. Une transformation de fourrier est ensuite ap-

une répétition de l'information *fichier* qui est aussi un *document*. Dans ce cas il suffit d'éliminer un synonyme pour ne garder qu'un seul cas d'intention.

Un autre cas est celui du sur-découpage d'intentions déjà assez semblable, *Ouvrir_fichier* et *Ouvrir_Repertoire* peuvent être regroupés en une seule intention *Ouvrir_Entite*. Bien-sur ces choix sont purement conceptuels et dépendent fortement du problème traité par le système. Choisir une représentation à la place d'une autre est lié à la nature de la tâche que le module doit accomplir (Liu and Lane, 2016 ; Goo et al., 2018).

2.5.2 Classification d'intentions

La classification d'intentions à partir d'un texte est une tâche réalisable avec des techniques récentes d'apprentissage automatique, plus précisément en utilisant des modèles basés sur les réseaux de neurones récurrents à "mémoire à court et long terme" (LSTM) (voir la section 2.4). Dans (Liu and Lane, 2016) et (Goo et al., 2018) deux architectures qui ont fait leurs preuves sont présentées. Dans le premier travail, les auteurs ont utilisé une architecture Bi-LSTM (LSTM Bidirectionnel) pour capturer les contextes droit et gauche d'un mot donné (Schuster and Paliwal, 1997). Le dernier état caché retourné par la cellule LSTM est ensuite utilisé pour la classification. Une couche d'attention est ajoutée (Chorowski et al., 2015) pour permettre au modèle de se focaliser sur certaines parties du texte en entrée. La deuxième architecture , qui est illustrée dans la figure 2.7, est un peu plus simple. En effet, elle utilise des cellules LSTM basiques avec une couche de classification sur le dernier état. L'avantage par rapport à la première architecture est le temps d'apprentissage assez réduit au détriment d'une erreur de classification plus accrue mais toujours dans les normes.

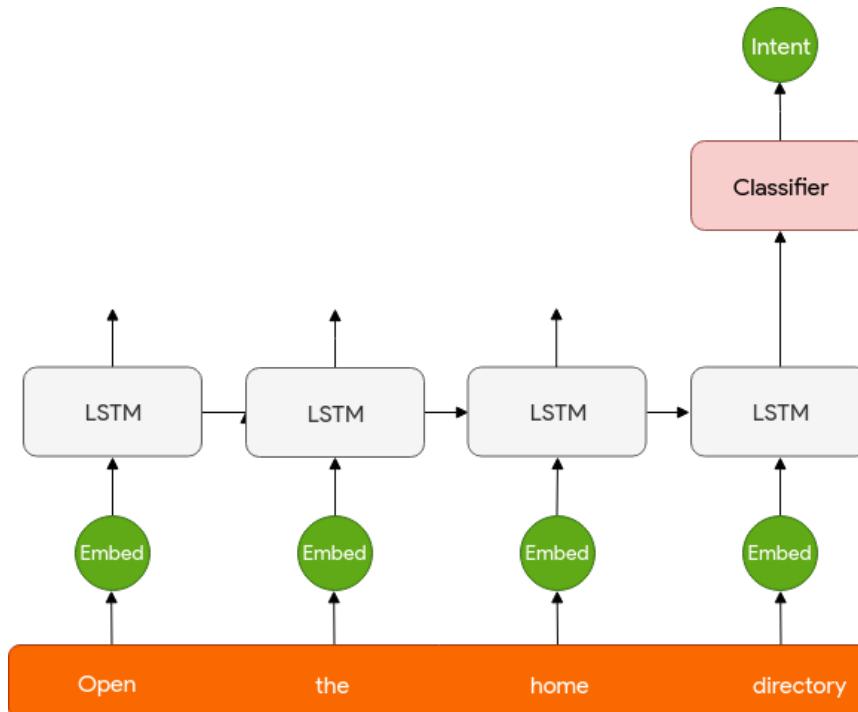


Figure 2.7 – Architecture de base d'un classificateur d'intentions (intents) (Liu and Lane, 2016)

2.5.3 Extraction d'entités

Déterminer l'intention d'un utilisateur ne s'arrête pas au stade de l'identification de l'*intent*. En effet, pour mieux représenter l'information récoltée depuis la requête, il faut en extraire des entités (nommées ou spécifiques du domaine) qui seront des arguments de l'*intent* identifié. Cette tâche consiste donc, pour une intention I donné, à extraire les arguments $Args$ qui lui sont appropriés depuis le texte de la requête. Plusieurs approches sont possibles. Dans (Goo et al., 2018) les auteurs ont attaqué le problème comme étant la traduction d'une séquence de mots en entrée en une séquence d'entités en sortie ; le figure 2.8 explicite le processus.

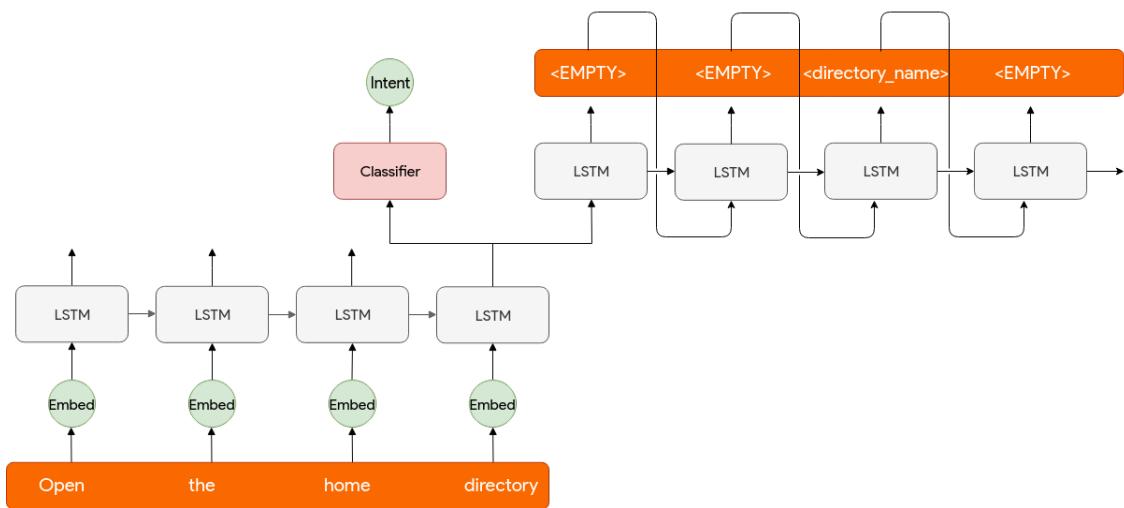


Figure 2.8 – Architecture de base d'un classificateur d'intentions doublé d'un extracteur d'entités (Goo et al., 2018)

2.5.4 Analyse sémantique

Disposant des deux modèles précédemment cités, le module NLU peut construire une représentation sémantique adéquate qui va être transmise au module suivant qui sera le gestionnaire du dialogue. L'avantage d'avoir en sortie une structure sémantique universelle est la non-dépendance par rapport à la langue de l'utilisateur. En effet, le système pourra encore fonctionner si une correspondance entre la nouvelle langue et le format choisi pour la représentation sémantique peut être trouvée. Le module donnera donc en sortie une trame sémantique (Semantic frame) qui comprendra les informations préalablement extraites, à savoir l'intention et les arguments (slots) qui lui sont propres (Liu and Lane, 2016 ; Goo et al., 2018 ; Wang et al., 2018).

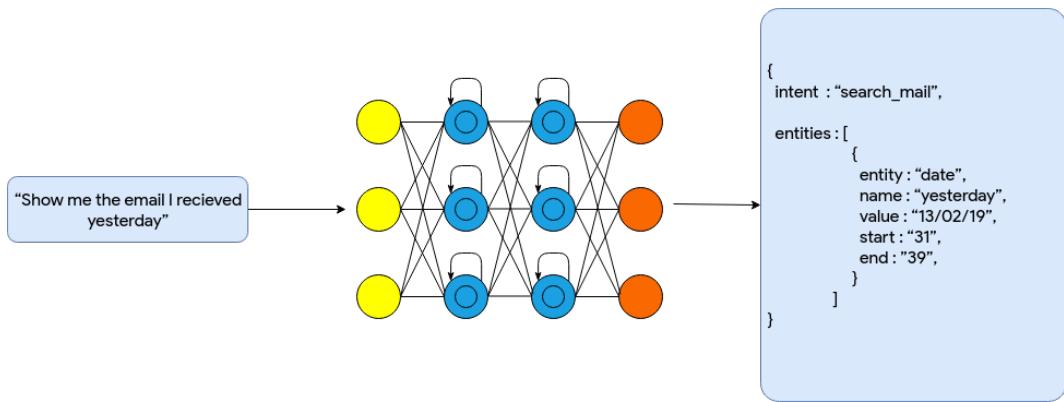


Figure 2.9 – Exemple d'une trame sémantique pour une requête donnée

2.6 Gestion du dialogue

La compréhension du langage naturel permet de transformer un texte en une représentation sémantique. Afin qu'un système puisse réaliser un dialogue aussi anthropomorphe que possible, il doit décider, à partir des représentations sémantiques reçues au cours du dialogue, quelle action prendre à chaque étape de la conversation. Cette action est transmise au générateur du langage naturel (voir 2.7) pour afficher un résultat à l'utilisateur. Généralement, deux principaux modules sont présents dans les systèmes de gestion de dialogue comme le montre la figure 2.10 :

- Un module qui met à jour l'état du gestionnaire de dialogue : le traqueur d'état.
- Un module qui détermine la politique d'action du gestionnaire de dialogue.

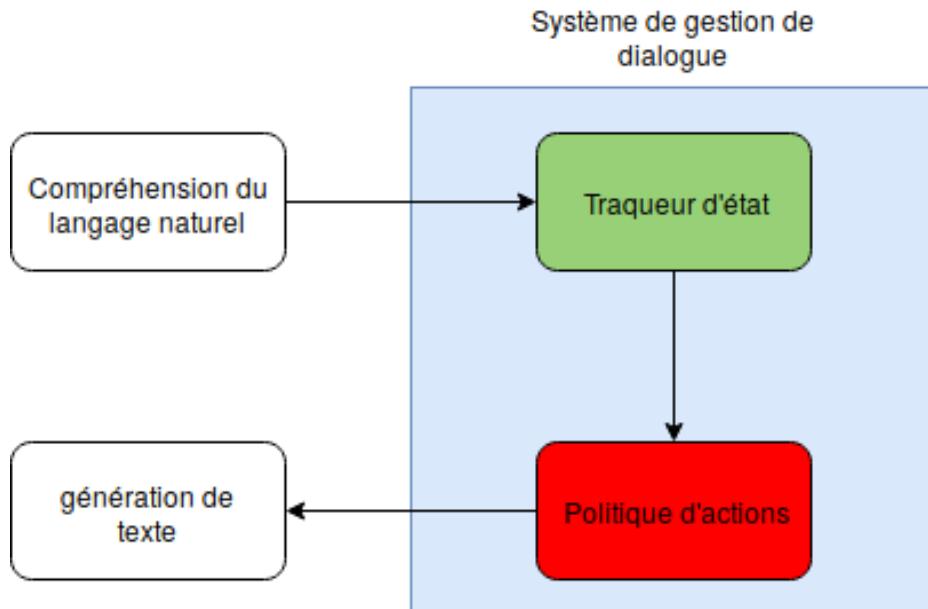


Figure 2.10 – Schéma général d'un gestionnaire de dialogue

Ainsi, le gestionnaire de dialogue passe d'un état à un autre après chaque interaction

avec l'utilisateur. Le problème à résoudre est donc de trouver la meilleure action à prendre en étant dans un état donné.

2.6.1 Modélisation

Pour résoudre ce problème, un gestionnaire de dialogue peut être modélisé par un processus de décision Markovien (Markovian Decision Process, MDP) (Bellman, 1957). Ce dernier est modélisé par un 4-tuple (S, A, P, R) :

- S : ensemble des états du système.
- A : ensemble des actions du système.
- P : distribution de probabilités de transitions entre états sachant l'action prise. $P(s'/s, a)$ est la probabilité de passer à l'état s' sachant qu'on était à l'état s après avoir réalisé l'action a .
- R : est la récompense reçue immédiatement après avoir changé l'état avec une action donnée. $R(s'/s, a)$ est la récompense reçue après être passé à l'état s' sachant que le système était à l'état s après avoir réalisé l'action a .

Un MDP, à tout instant t , est dans un état s . Dans notre cas c'est l'état du gestionnaire de dialogue. Il peut prendre une action a afin de passer à un nouvel état s' . De ce fait, il reçoit une récompense qui, dans notre cas, est une mesure sur les performances du système de dialogue. Le but est de maximiser les récompenses reçues. La figure 2.11 illustre un exemple de MDP à trois états avec les probabilités de transitions entre eux ainsi que la récompense associée à chaque transition.

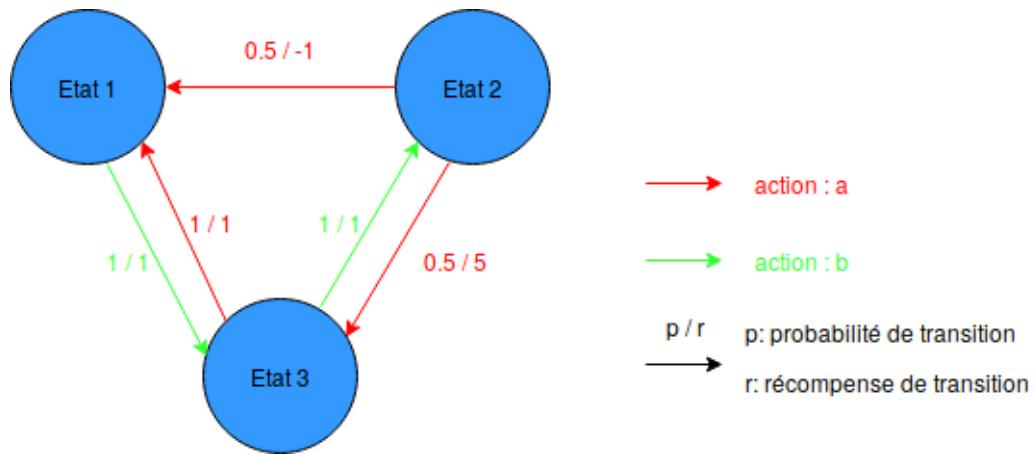


Figure 2.11 – Schéma représentant les transitions entre états dans un MDP

2.6.2 État du gestionnaire de dialogue

L'état d'un système de dialogue est une représentation sémantique qui contient des informations sur le but final de l'utilisateur ainsi que l'historique de la conversation. La représentation souvent utilisée dans les systèmes de dialogue est celle de la trame sémantique (Chen et al., 2017). Cette structure contient des emplacements à remplir dans un domaine

donné. La figure 2.12 illustre un exemple de trame sémantique.

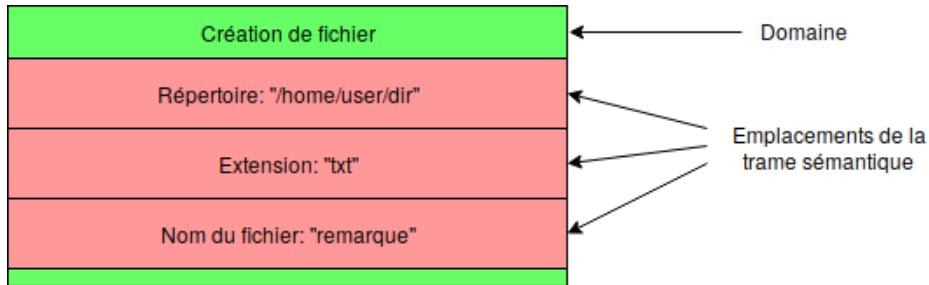


Figure 2.12 – Schéma représentant une trame sémantique avec comme domaine : création de fichier

À l'arrivée d'une nouvelle information, un module dédié met à jour l'état du gestionnaire de dialogue. Comme l'action du système de dialogue est décidée à partir de son état, cette tâche est donc essentielle au bon fonctionnement du système. Plusieurs méthodes ont été proposées pour gérer le suivi de l'état du gestionnaire de dialogue (Williams and Young, 2007).

2.6.2.1 Suivi de l'état du gestionnaire de dialogue avec une base de règles

La méthode traditionnelle utilisée consiste à écrire manuellement les règles à suivre lors de l'arrivée d'une nouvelle information pour mettre à jour l'état (Goddeau et al., 1996). Cependant, les bases de règles sont très susceptibles à faire des erreurs (Chen et al., 2017) car elles sont peu robustes face aux incertitudes. La figure 2.13 est un exemple de mise à jour d'une trame sémantique à partir d'une commande utilisateur avec les bases de règles.

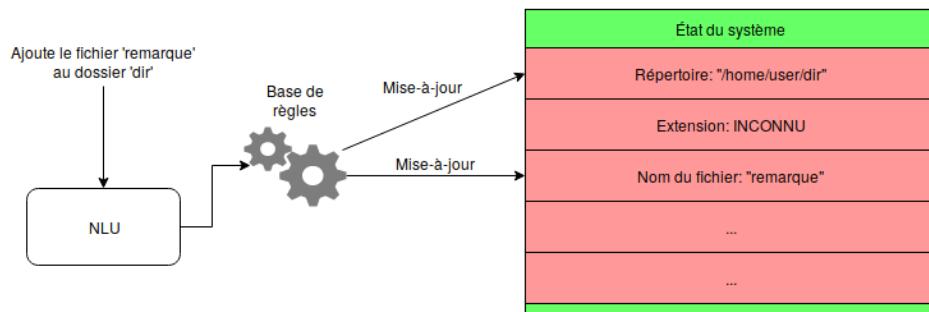


Figure 2.13 – Schéma représentant la mise-à-jour de l'état du gestionnaire de dialogue par un système basé règles

2.6.2.2 Suivi de l'état du gestionnaire de dialogue avec des méthodes statistiques

Le suivi dans ce cas se fait en gardant une distribution de probabilités sur l'état du système, ce qui nécessite une nouvelle modélisation non déterministe du problème. Les processus de décision markoviens partiellement observables (Partially Observable Markov

Decision Process POMDP) (Young et al., 2010) sont une variante des MDP capable de répondre à ce besoin que nous introduirons par la suite. Dans ce cas, le système garde une distribution de probabilités sur les valeurs possibles des différents emplacements de la trame sémantique comme le montre la figure 2.14.

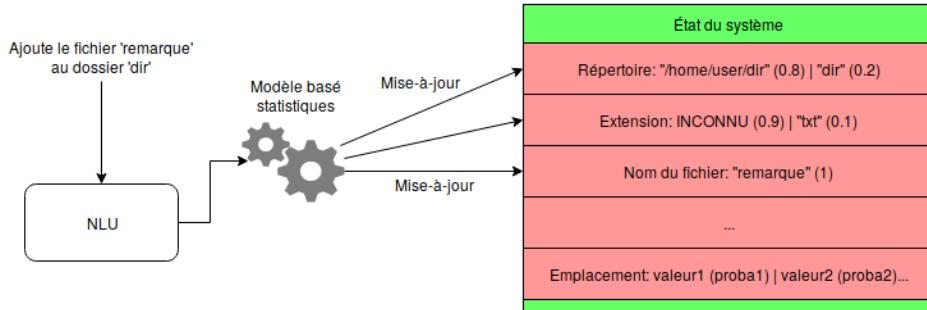


Figure 2.14 – Schéma représentant la mise-à-jour de l'état du gestionnaire de dialogue par un système basé statistiques

Processus de décision markovien partiellement observable (POMDP) Comme dans les processus de décision markoviens, un POMDP (Åström, 1965) passe d'un état à un autre en prenant une des actions possibles. Cependant, un POMDP ne connaît pas l'état exact dans lequel le système se trouve à un instant t . Il reçoit par contre une observation qui est, dans notre cas, l'action de l'utilisateur, à partir de laquelle il peut estimer une distribution de probabilités sur l'état actuel. Pour résumer, un POMDP est un 6-tuple (S, A, P, R, M, O) où :

- Les 4 premiers composants sont les mêmes que ceux d'un MDP (voir 2.6.1).
- M : l'ensemble des observations.
- O : la distribution de probabilités sur les observations o en connaissant l'état s et l'action a prise pour y arriver. $O(o|s, a)$ est la probabilité d'observer o sachant que le système se trouve à l'état s et qu'il a pris l'action a pour y arriver.

La figure 2.15 est une partie du diagramme d'influence d'un POMDP montrant les influences d'une transition entre deux états.

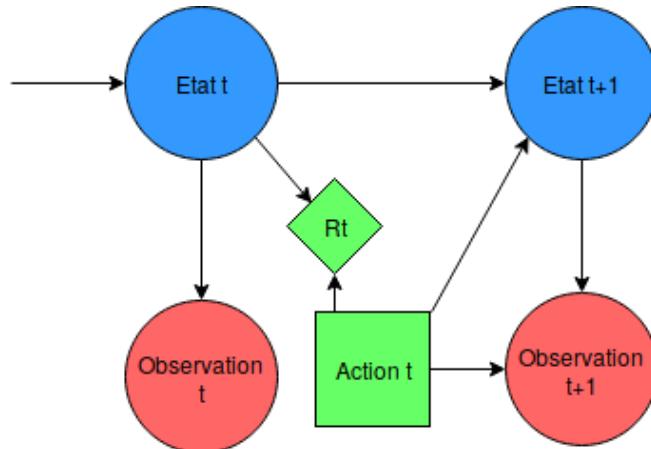


Figure 2.15 – Diagramme d'influence dans un POMDP

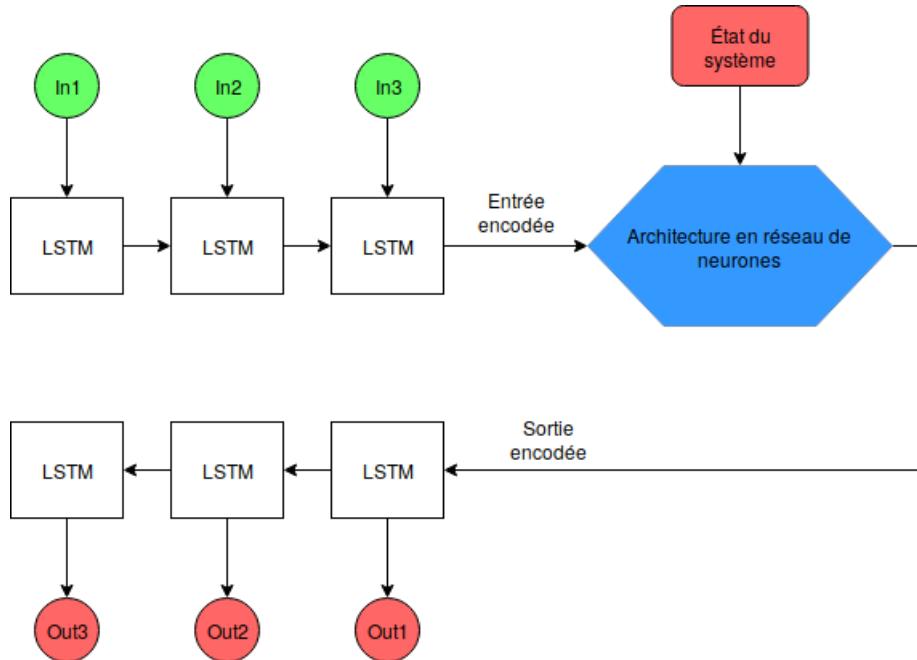


Figure 2.16 – Schéma de gestion de dialogue de bout en bout avec architecture Seq2Seq

une mesure d'évaluation qui est sous forme de récompenses obtenues après chaque action (Weisz et al., 2018). L'environnement est souvent modélisé comme un MDP, ou éventuellement POMDP. L'agent d'apprentissage passe donc d'un état à un autre en prenant des actions dans cet environnement. L'apprentissage se fait dans ce cas en apprenant par l'expérience de l'agent, à savoir les récompenses obtenues par les actions prises dans des états donnés. Il peut ainsi estimer la fonction de récompense pour pouvoir faire le choix d'actions optimales. La figure 2.17 montre les relations entre l'agent et son environnement dans un apprentissage par renforcement.

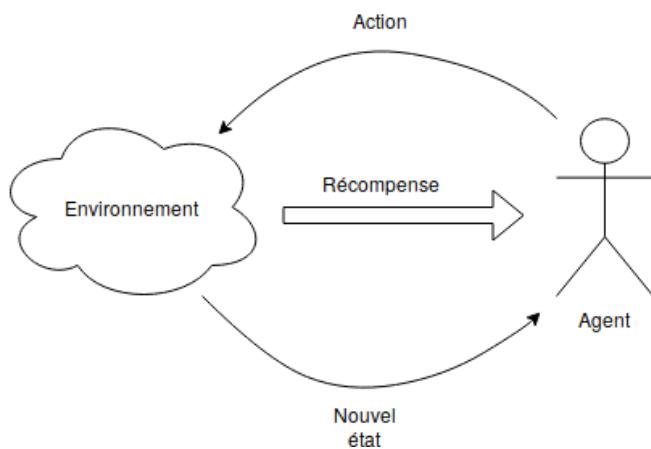


Figure 2.17 – Schéma d'interaction agent-environnement dans l'apprentissage par renforcement

Il existe plusieurs méthodes que l'agent peut utiliser pour estimer la fonction de récompense (Bertsekas, 2007), notamment Deep Q Learning (DQN) (Mnih et al., 2015) qui utilise les réseaux de neurones profonds pour trouver une approximation à cette fonction. Dans

tamment la grammaire systémique fonctionnelle (SFG) (Halliday and Matthiessen, 2004) qui a été largement utilisée comme dans NIGEL (Mann and Matthiessen, 1983) ou KPMI (Bateman, 1997). L'exploitation des grammaires dans la génération du texte nécessite généralement des entrées détaillées. En plus des composantes du lexique sélectionnées, des descriptions de leurs rôles ainsi que leurs fonctions grammaticales sont souvent exigées. Un exemple d'entrée d'un système basé grammaire est celui de SURGE (Elhadad and Robin, 1996) dans la figure 2.18 qui génère la phrase : "She hands the draft to the editor".

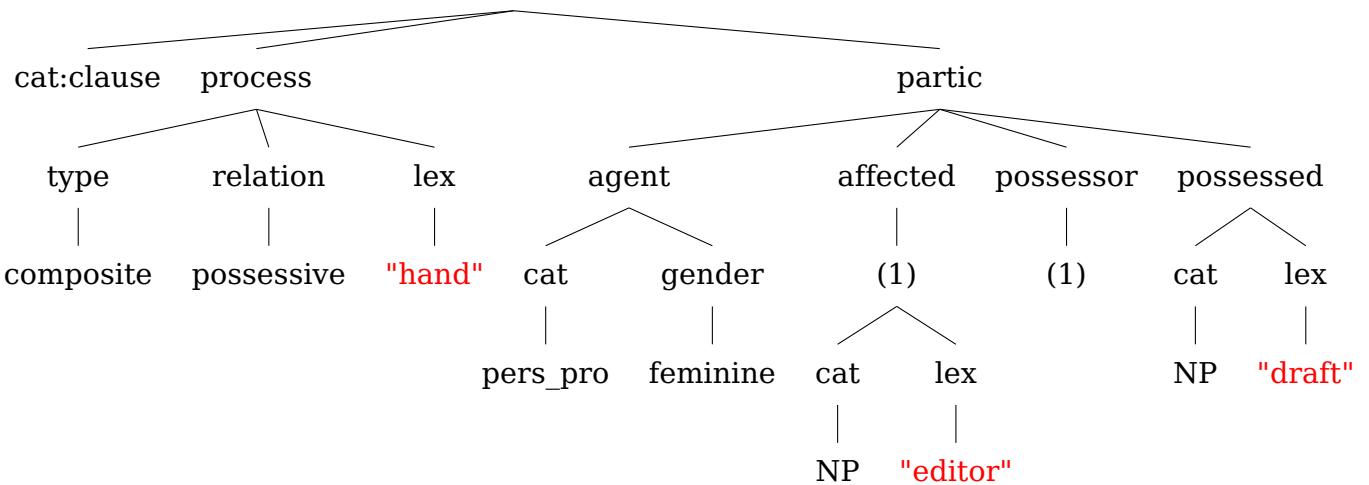


Figure 2.18 – Un exemple d'entrée de SURGE (Elhadad and Robin, 1996)

Comme les modèles de phrases, les systèmes basés grammaire nécessitent un énorme travail manuel. En particulier, il est difficile de prendre en compte le contexte en définissant les règles de choix entre les variantes possibles du texte résultat à partir des entrées (Gatt and Krahmer, 2018).

Approches statistiques : Il existe plusieurs méthodes basées sur des statistiques pour la tâche de réalisation. Certains se basent sur des grammaires probabilistes, qui ont l'avantage de minimiser le travail manuel tout en couvrant plus de cas de réalisation. Il existe principalement deux approches l'utilisant (Gatt and Krahmer, 2018) :

- La première se base sur une petite grammaire qui génère plusieurs alternatives qui sont ensuite ordonnées selon un modèle statistique basé sur un corpus pour sélectionner la phrase la plus probable comme par exemple (Langkilde-Geary, 2000).
- La deuxième méthode utilise les informations statistiques directement au niveau de la génération pour produire la solution optimale (Belz, 2008).

Dans les deux méthodes sus-citées la grammaire de base peut être manuellement développée. Dans ce cas, les informations statistiques aideront à la détermination de la solution optimale. Elle peut être aussi extraite à partir des données, comme l'utilisation des Treebanks³ pour déduire les règles de grammaire (Espinosa et al., 2008).

3. un Treebank est un texte analysé qui contient des informations syntaxiques ou sémantiques sur les structures de phrases

D'autres approches statistiques n'utilisent pas des grammaires mais se basent sur des classificateurs. Ces derniers peuvent être cascadés de telle sorte à décider quel constituant utiliser dans quelle position ainsi que les modifications nécessaires pour générer un texte correct. À noter qu'une telle approche, ne nécessitant pas l'utilisation de grammaire, utilise des entrées plus abstraites et moins détaillées linguistiquement. À voir même la possibilité qu'elle s'étende aux autres tâches de NLG, c'est-à-dire un système qui accomplit plusieurs tâches de NLG en parallèle jusqu'à la réalisation linguistique en utilisant les entrées initiales. Par exemple, les systèmes encodeur-décodeur sont des systèmes de bout-en-bout qui peuvent, directement à partir des entrées, générer du texte sans passer explicitement par les étapes citées précédemment. Dans la suite de ce travail, nous allons présenter ces systèmes basés encodeur-décodeur qui sont récemment plus utilisés.

2.8 Systèmes basés encodeur-décodeur

Une architecture souvent utilisée dans le traitement du langage naturel est l'encodeur-décodeur. En particulier, son utilisation dans les tâches seq2seq permet de mettre en correspondance une séquence de taille variable en entrée avec une autre séquence en sortie. Les modèles seq2seq peuvent être adaptés pour convertir une représentation abstraite de l'information en langage naturel (Ferreira et al., 2017) comme le montre la figure 2.19.

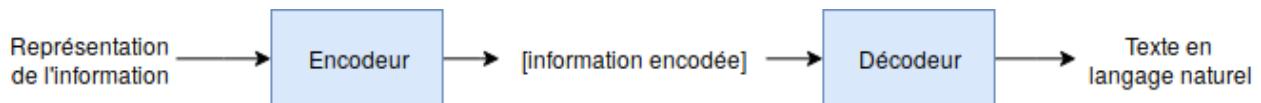


Figure 2.19 – Schéma d'une architecture encodeur-décodeur pour NLG

Beaucoup d'approches de génération de langage naturel en gestion de dialogue utilisent des encodeurs-décodeurs. Wen et al. (2015) utilisent par exemple des LSTMs sémantiquement conditionnés ; ils ajoutent aux LSTMs classiques une couche contenant des informations sur l'action prise par le gestionnaire de dialogue pour assurer que la génération représente le sens désiré. D'autres travaux utilisent des réseaux de neurones récurrents pour encoder l'état du gestionnaire de dialogue et l'entrée reçue, suivie par un décodeur pour générer le texte de la réponse (Sordoni et al., 2015 ; Serban et al., 2016 ; Goyal et al., 2016).

2.9 Conclusion

Au terme de ce chapitre et après avoir étudié l'état de l'art sur les systèmes existants, nous avons pu construire un bagage théorique assez complet dans le domaine des SPAs. En assimilant les différentes approches, il est maintenant temps de passer à la conception de notre propre système. Le chapitre suivant introduira nos propositions pour le développement de notre propre SPA qui sera destiné à la manipulation d'un ordinateur.

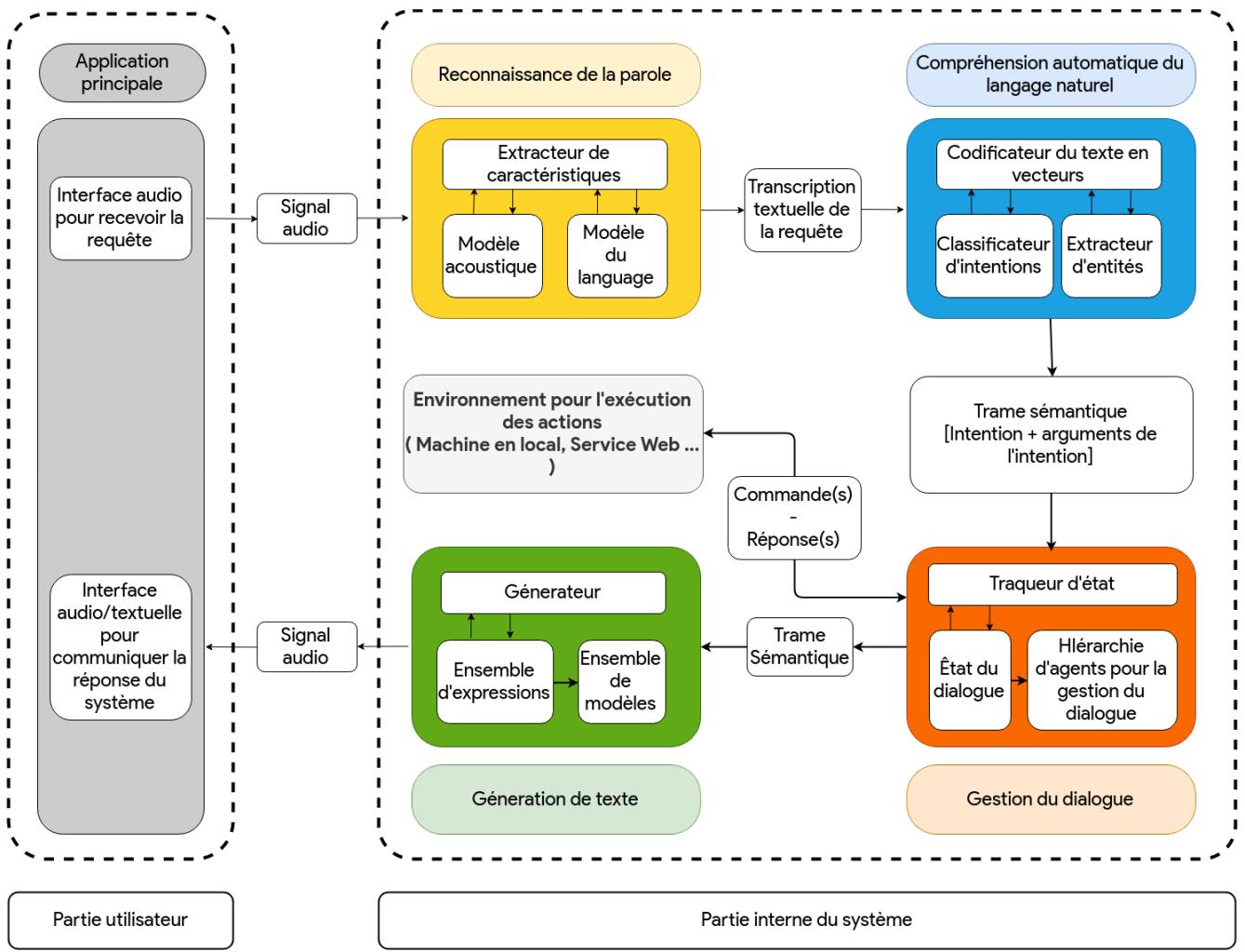


Figure 3.1 – Architecture générale du système Speact

son choix. Afficher le texte et sa transcription vocale pourrait palier à certains manques comme l'absence d'un périphérique de sortie audio.

3.2.2 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 2.2), 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 API² 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 équivalent 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.

3.3 Module de reconnaissance automatique de la parole

Premier module du système Speact, 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.

3.3.1 Architecture du module ASR

Comme illustré par la figure 3.2, le module ASR possède une architecture en pipeline dont chaque composant exécute un traitement sur la donnée reçue par son prédecesseur.

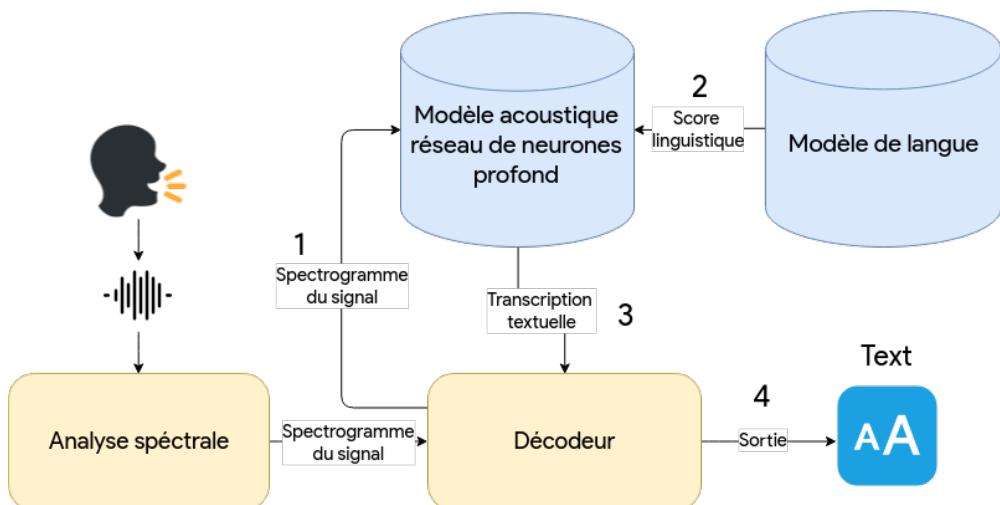


Figure 3.2 – Architecture du module de reconnaissance de la parole (ASR)

2. Application Programming Interface ou interface de programmation applicative.

3.3.2 Modèle acoustique

Type du modèle

Le modèle d'apprentissage, qui est principalement le modèle acoustique à l'exception d'une partie consacrée au modèle linguistique, possède une architecture en réseau de neurones avec apprentissage de bout-en-bout composé de trois parties (voir la figure 3.3) :

- 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.

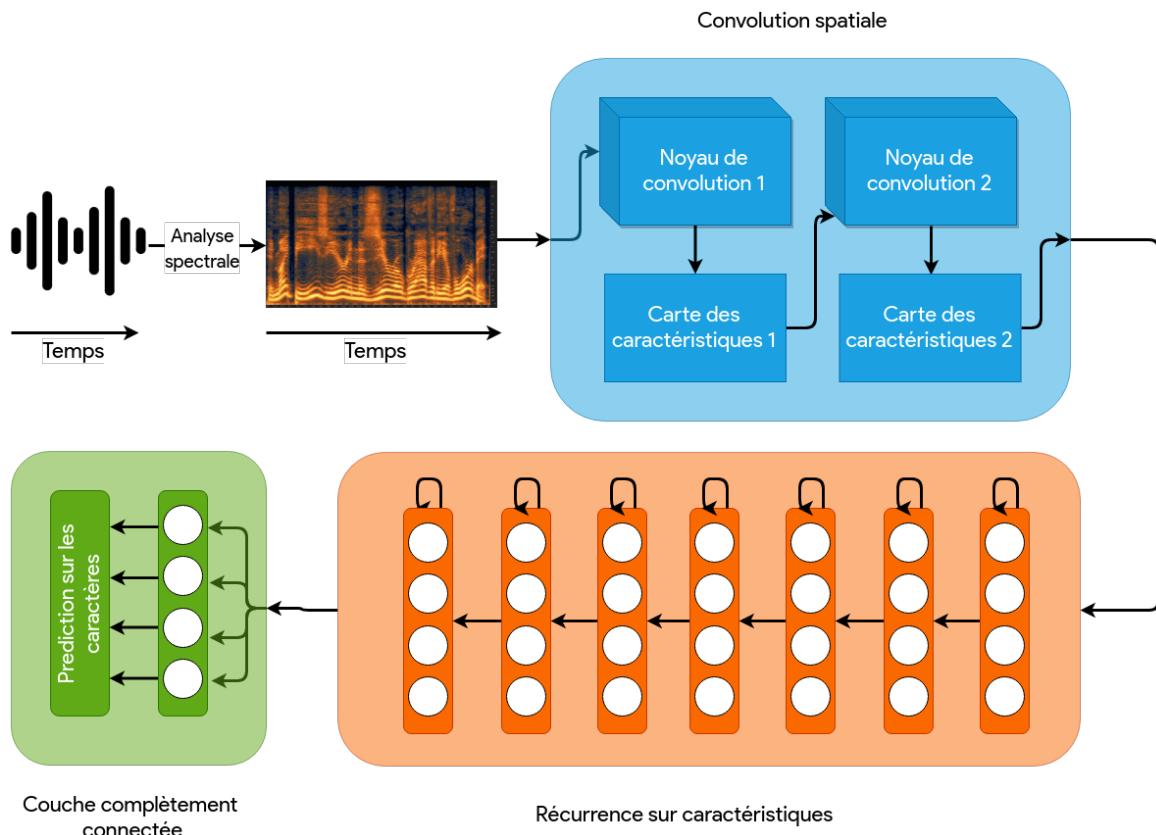


Figure 3.3 – Architecture du modèle DeepSpeech (Hannun et al., 2014)

Données d'apprentissage

Pour entraîner le modèle acoustique, Mozilla a lancé le projet Common Voice³, une plateforme en ligne pour récolter des échantillons audio avec leurs transcriptions textuelles.

3. <https://voice.mozilla.org/fr>

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

La liste des noms de dépôts est disponible dans un fichier⁸ 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>
...

```

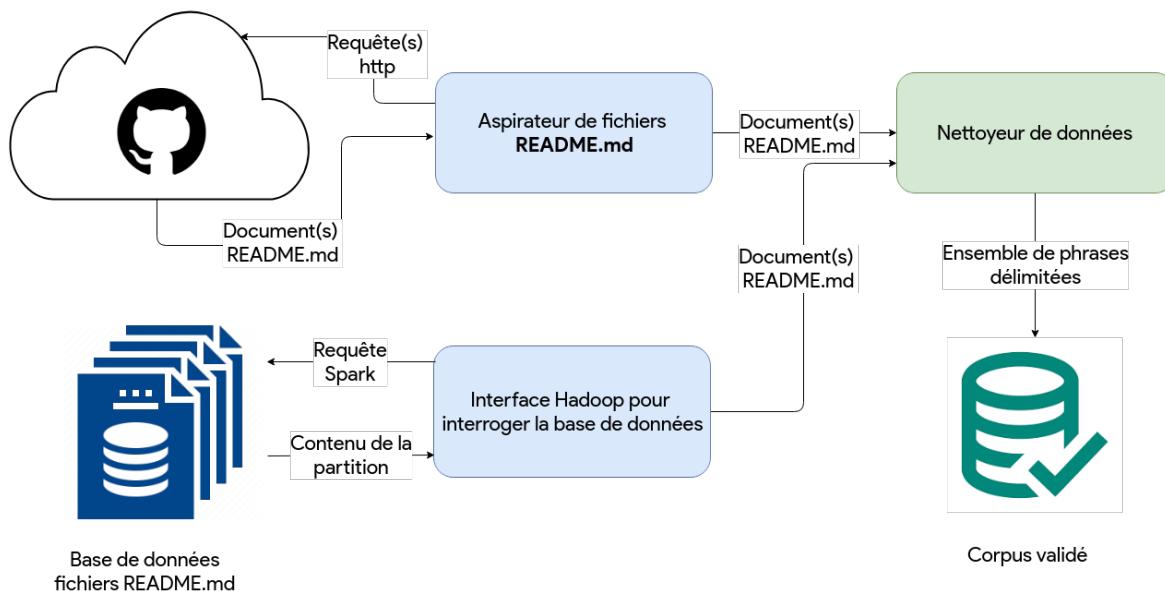


Figure 3.4 – Processus de génération du corpus pour le modèle de langue

3.4 Module de compréhension automatique du langage naturel

Second module du système, le module de compréhension automatique du langage naturel (Natural Language Understanding, NLU) dont le rôle est de faire office de couche d'abstraction.

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

tion 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.

3.4.1 Architecture du module

Comme illustré dans la figure 3.5, 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'entités⁹, qui sont en fait un seul modèle joint dont l'architecture est détaillée dans la section 3.4.2. 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 :

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

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.

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

- **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.

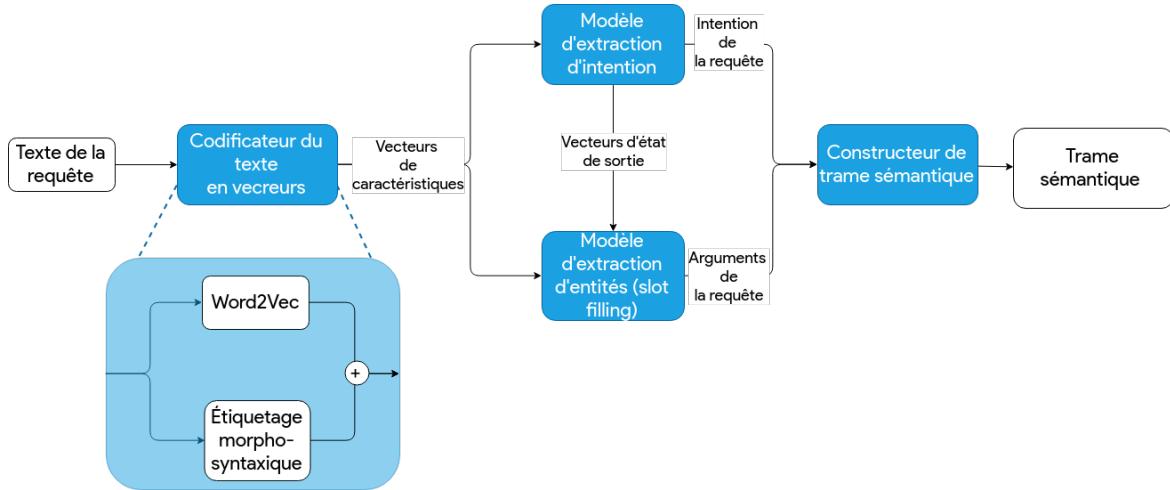


Figure 3.5 – Architecture du module de compréhension automatique du langage naturel (NLU)

3.4.2 Modèle utilisé

Comme vu dans le chapitre précédent (voir la section 2.5), 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 (Bidirectional 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 récurrent 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 2.8).

3.4.3 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 liste¹⁰ 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 :

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

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.

3.5.1.2 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 (voir figure 3.6) qui est ensuite ajouté au graphe d'états. Plus de détails seront donnés dans la section 3.5.2 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.

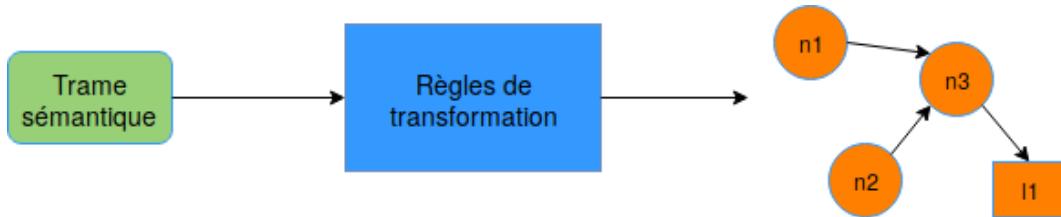


Figure 3.6 – Schéma de transformation de trame sémantique en graphe

3.5.1.3 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 comme c'est illustré dans la figure 3.7.

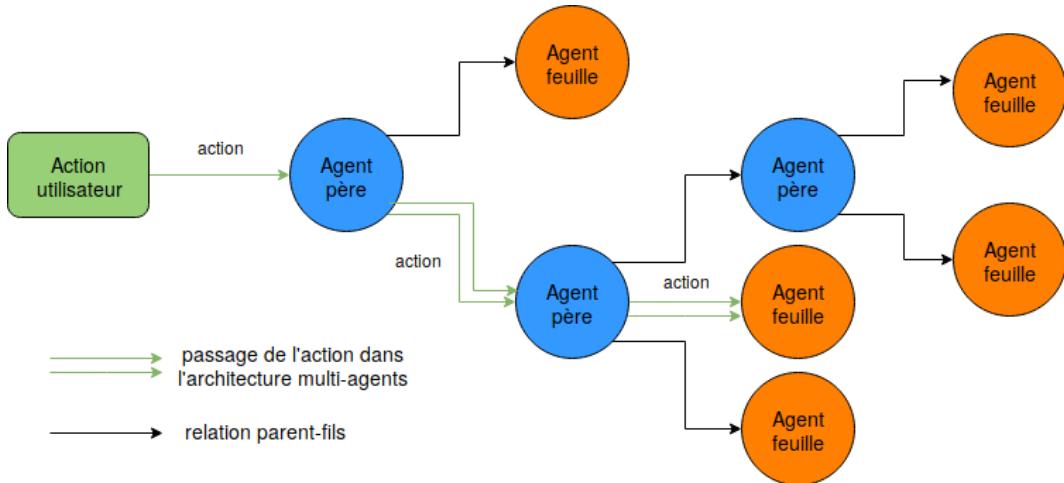


Figure 3.7 – Schéma de l’architecture multi-agents pour la gestion du dialogue

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. La figure 3.8 montre l’association des agents feuilles aux simulateurs pendant l’apprentissage.

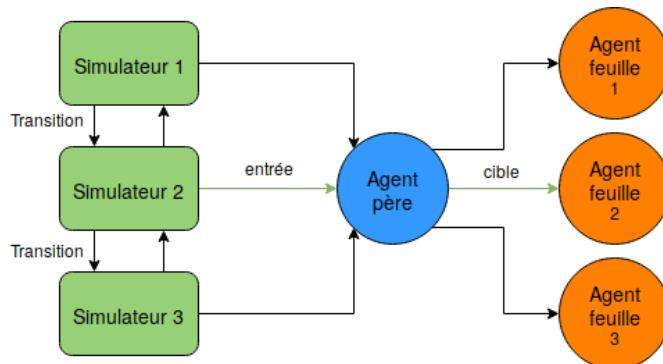


Figure 3.8 – Schéma représentant l’apprentissage des agents parents avec les simulateurs des agents feuilles

Pour résumer l’architecture globale du gestionnaire de dialogue (voir figure 3.9), 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.

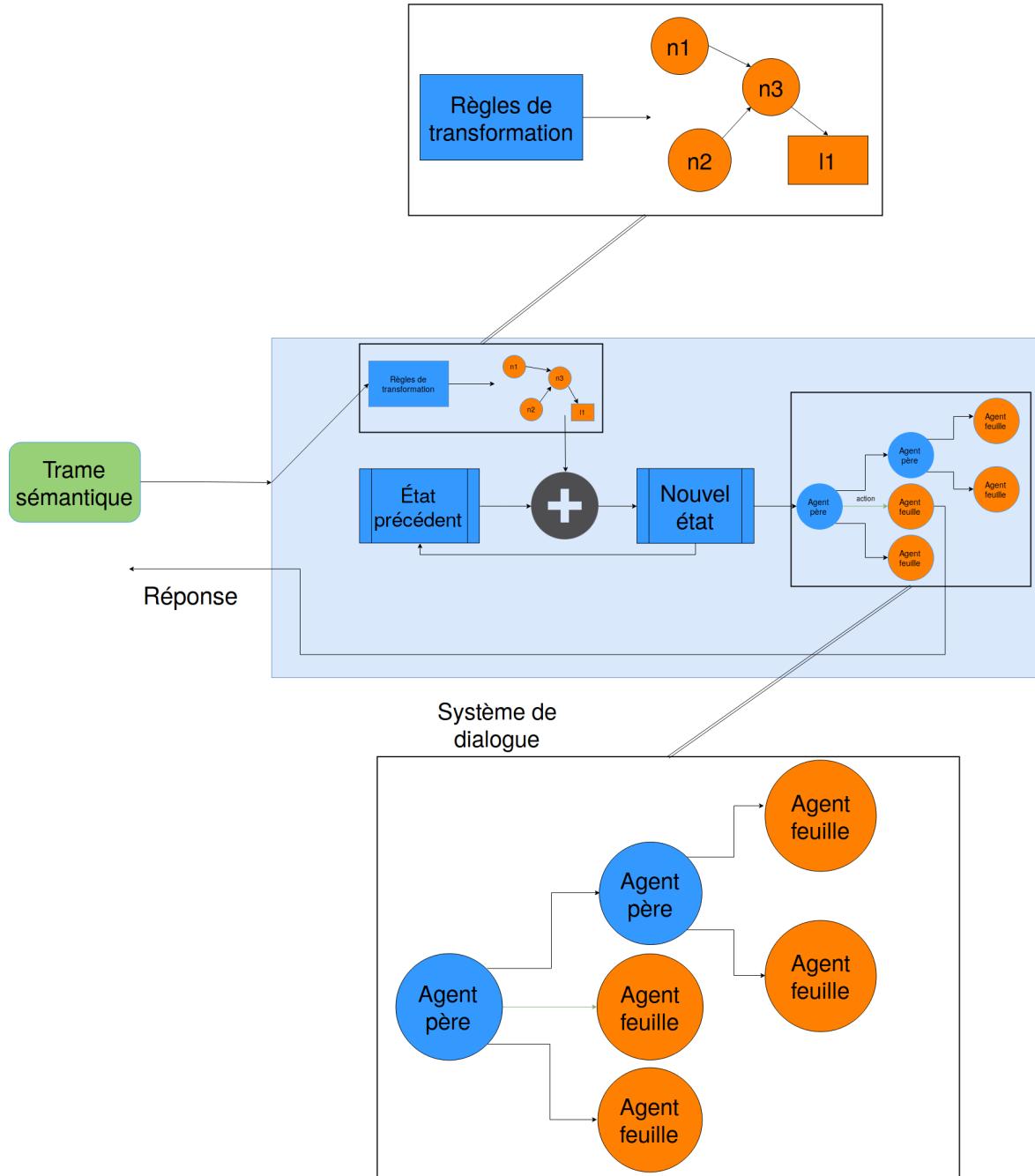


Figure 3.9 – Schéma global du gestionnaire de dialogue

3.5.2 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.

3.5.2.1 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.

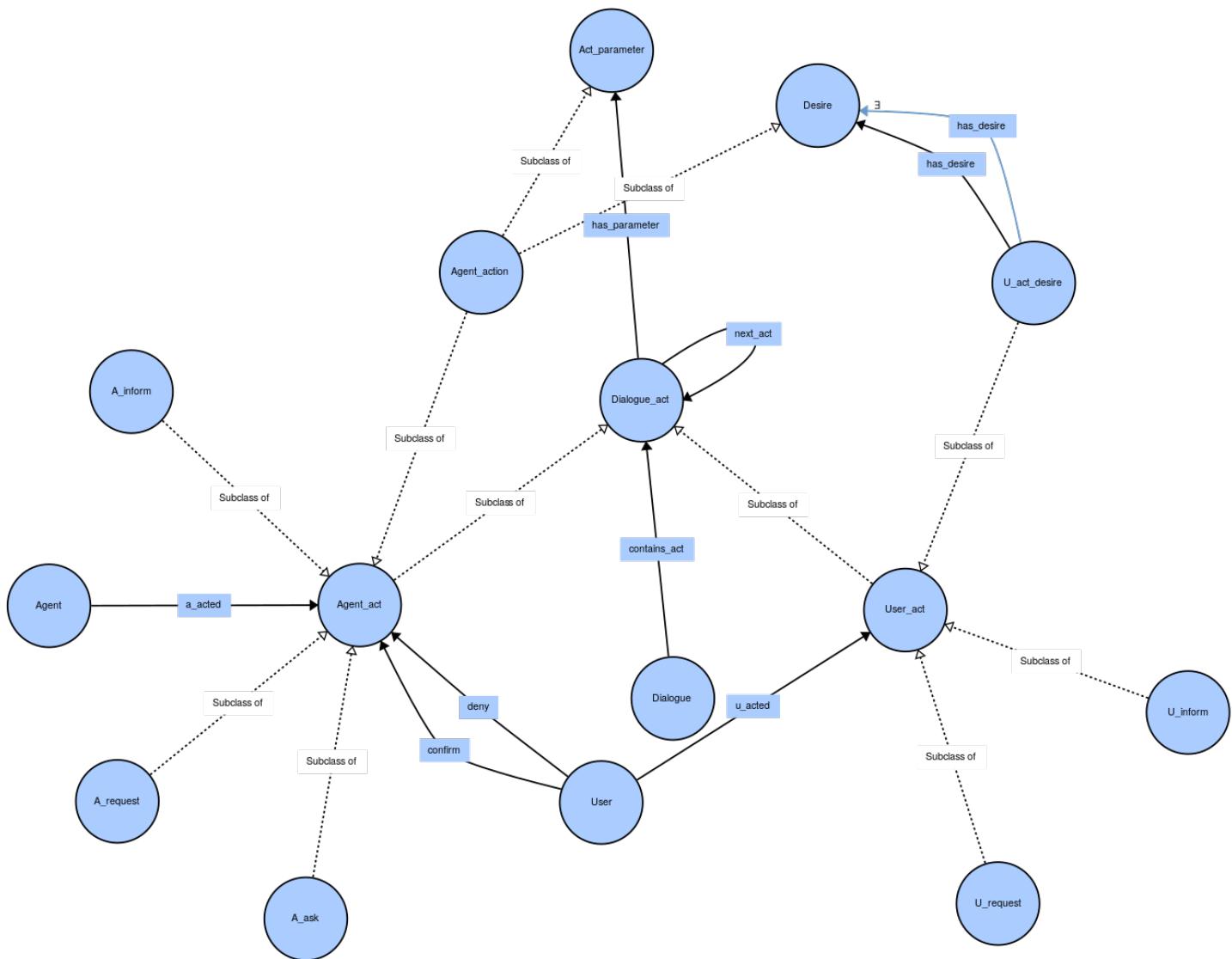


Figure 3.10 – Graphe de l'ontologie de dialogue

Principalement, l'ontologie, illustrée dans la figure 3.10, se compose de six classes mères :

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

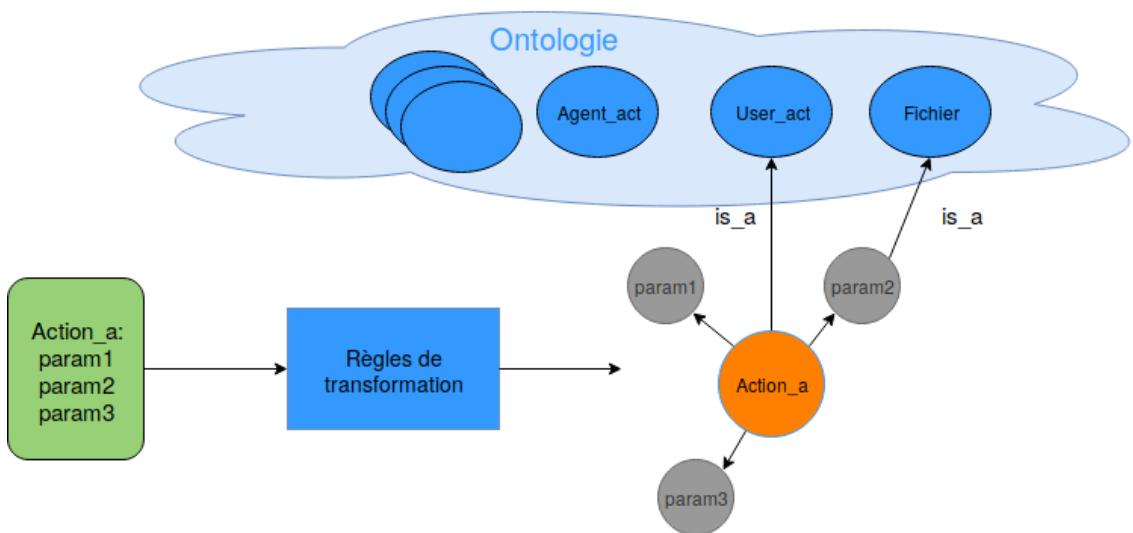


Figure 3.11 – Schéma de transformation d'une action en graphe

- *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 3.11. Une nouvelle action utilisateur est créée ainsi que ses paramètres et les relations entre eux.

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 qui est illustrée dans la figure 3.12.

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.

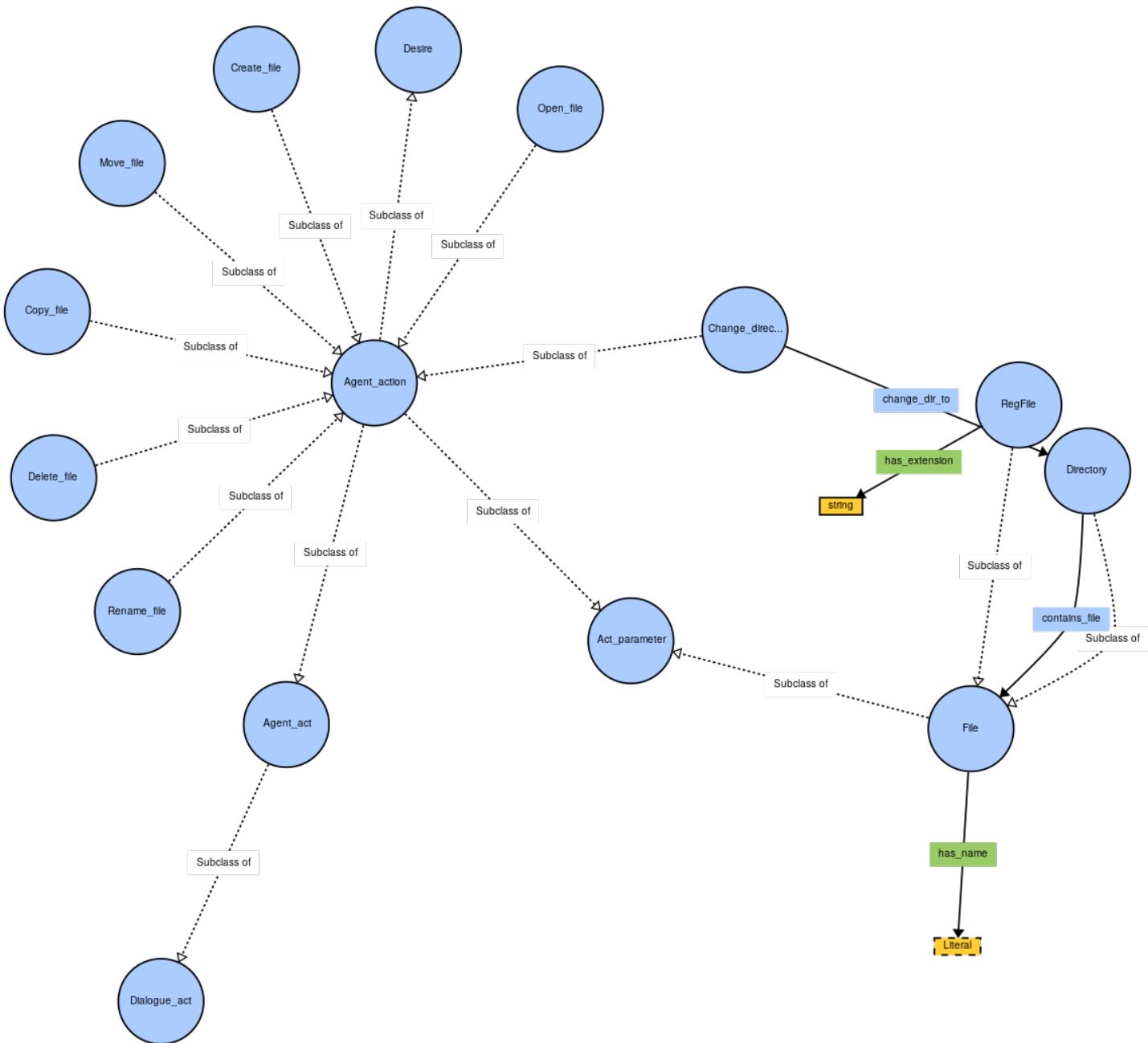


Figure 3.12 – Graphe de l'ontologie de l'exploration de fichiers

La figure 3.13 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.

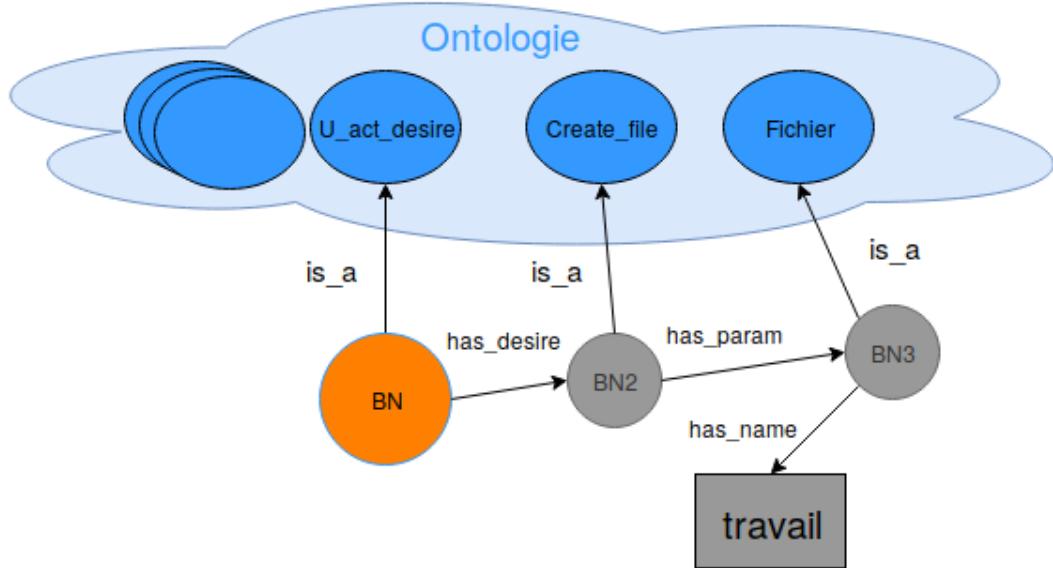


Figure 3.13 – Schéma de transformation d'une action de demande de création de fichier en graphe

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

3.5.3 Les simulateurs d'utilisateurs

Plusieurs méthodes peuvent être utilisées pour la création de simulateurs d'utilisateurs (voir 2.6.3.3). 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,

- **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 3.14 résume cette décision.

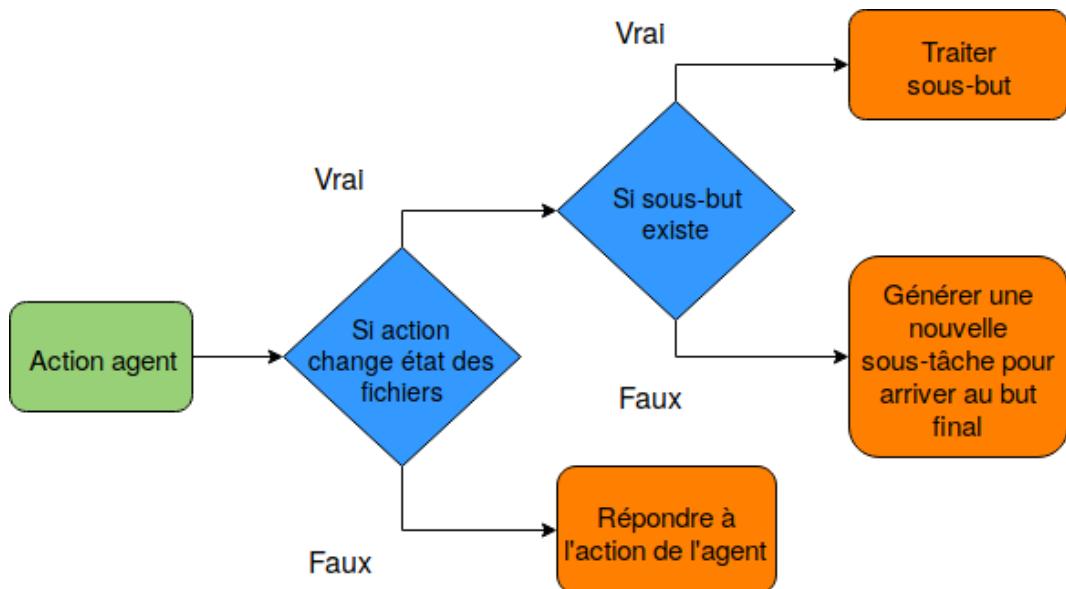


Figure 3.14 – Diagramme de décision de l'action à prendre

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.

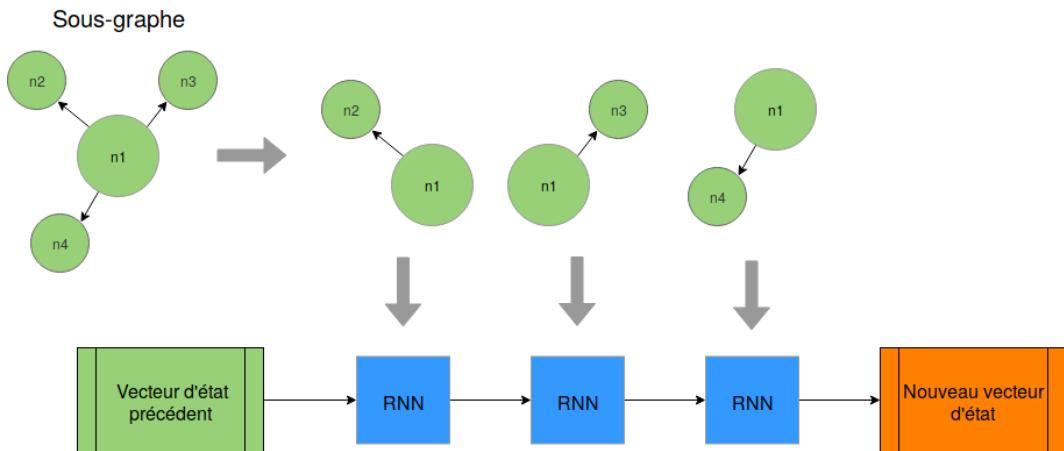


Figure 3.16 – Schéma représentant un encodeur séquentiel de graphe

étape est répétée k fois pour que chaque noeud ait des informations sur les noeuds qui sont à un maximum de k pas de distance, avec k un paramètre empirique. Enfin, la somme des vecteurs d'états des différents noeuds est effectuée pour produire un vecteur fixe encodant tout le graphe qui peut être relié au reste du réseau de neurones pour l'apprentissage.

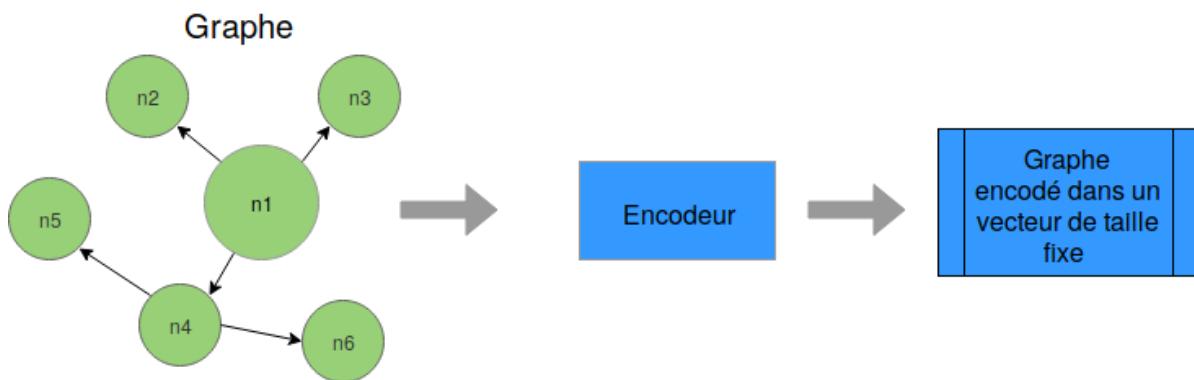


Figure 3.15 – Schéma représentant un encodeur de graphe

Ces méthodes nécessitent tout le graphe pour l'encoder comme l'illustre la figure 3.15. 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 : "noeud ; arc ; noeud" ou "sujet ; predicat ; objet" comme dans la représentation du modèle de données RDF (Resource Description Framework¹¹). 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.

Comme le montre la figure 3.16, 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.

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

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 3.17 illustre un exemple des entrées/sorties de l'apprentissage en utilisant cette architecture.

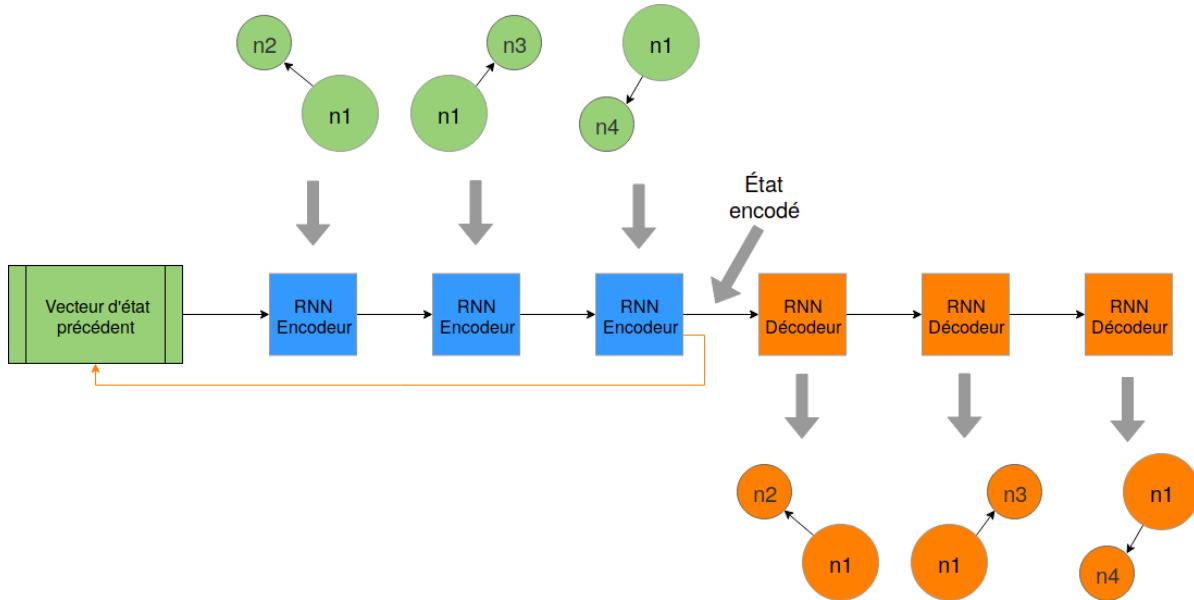


Figure 3.17 – Schéma de l'apprentissage d'un encodeur séquentiel de graphe

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 comme il est montré dans la figure 3.18.

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 :

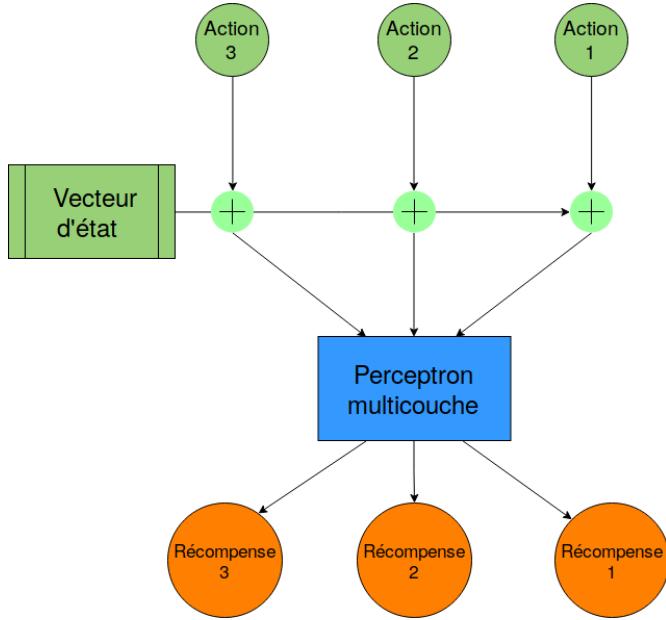


Figure 3.18 – Schéma du réseau DQN

- Rejouer l'expérience : l'agent interagit avec le simulateur plusieurs fois en gardant dans une mémoire ses interactions. Après chaque k épisodes¹², 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.
 - 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^{ème} apprentissage est donc :

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

Avec Q_i et Q_{i-1} : les fonctions de récompense données par les réseaux de neurones pendant le i^{ème} apprentissage et avant le début du i^{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.

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

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

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 (voir figure 3.19). 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.

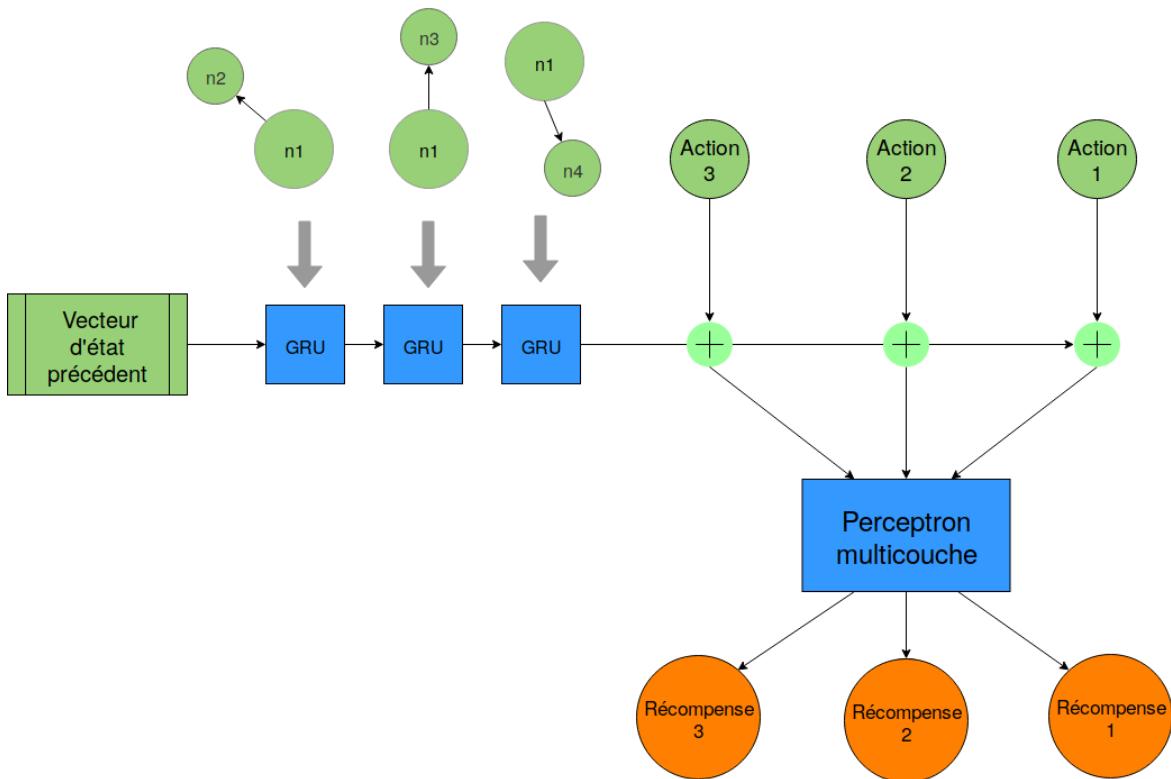


Figure 3.19 – Schéma du réseau DQN relié avec l'encodeur directement

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.

3.6 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.

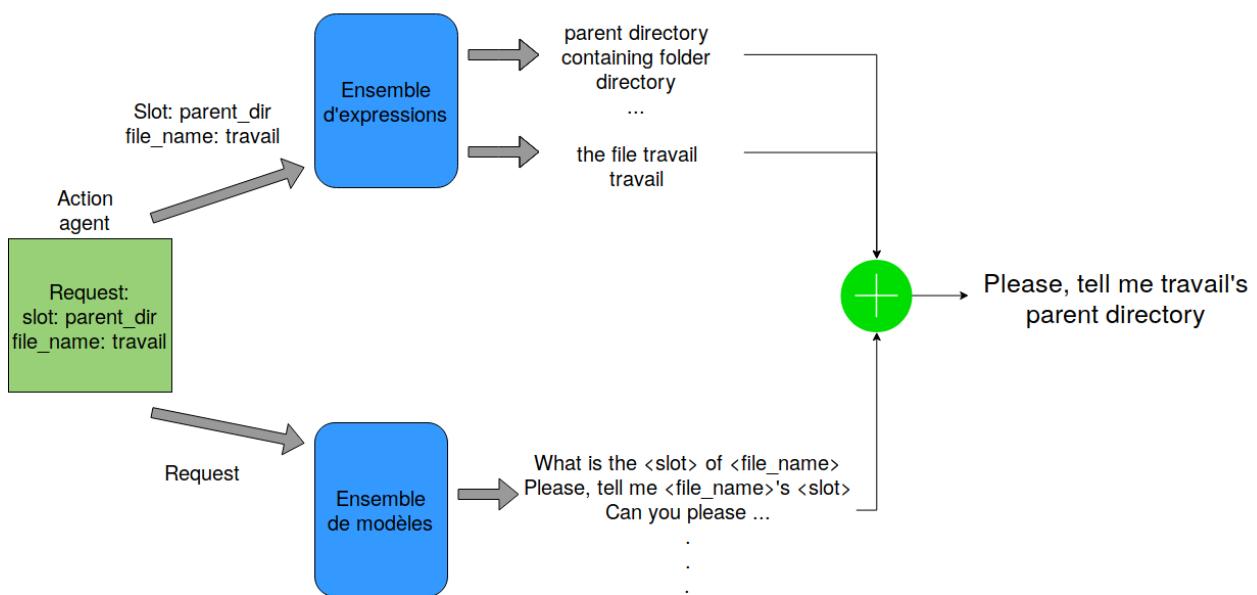


Figure 3.20 – Schéma de fonctionnement du générateur de textes

Dans la figure 3.20, 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".



Figure 4.1 – 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 :

4.2.2 Langages de programmation

Python

Python¹ 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

JavaScript² 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

4.2.3 Librairies et bibliothèques

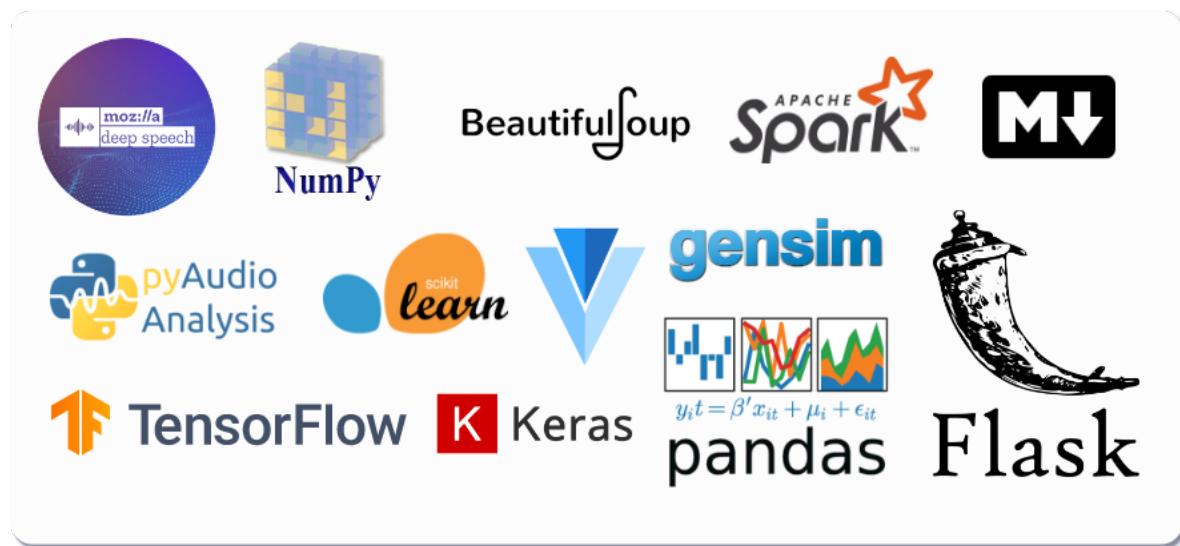


Figure 4.2 – Bibliothèques et librairies les plus utilisées dans ce projet.

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

PyAudio⁴ 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 Soup⁵ 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/>

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 3.3, 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 4.1 – Tableau récapitulatif des résultats pour le module de reconnaissance automatique de la parole.

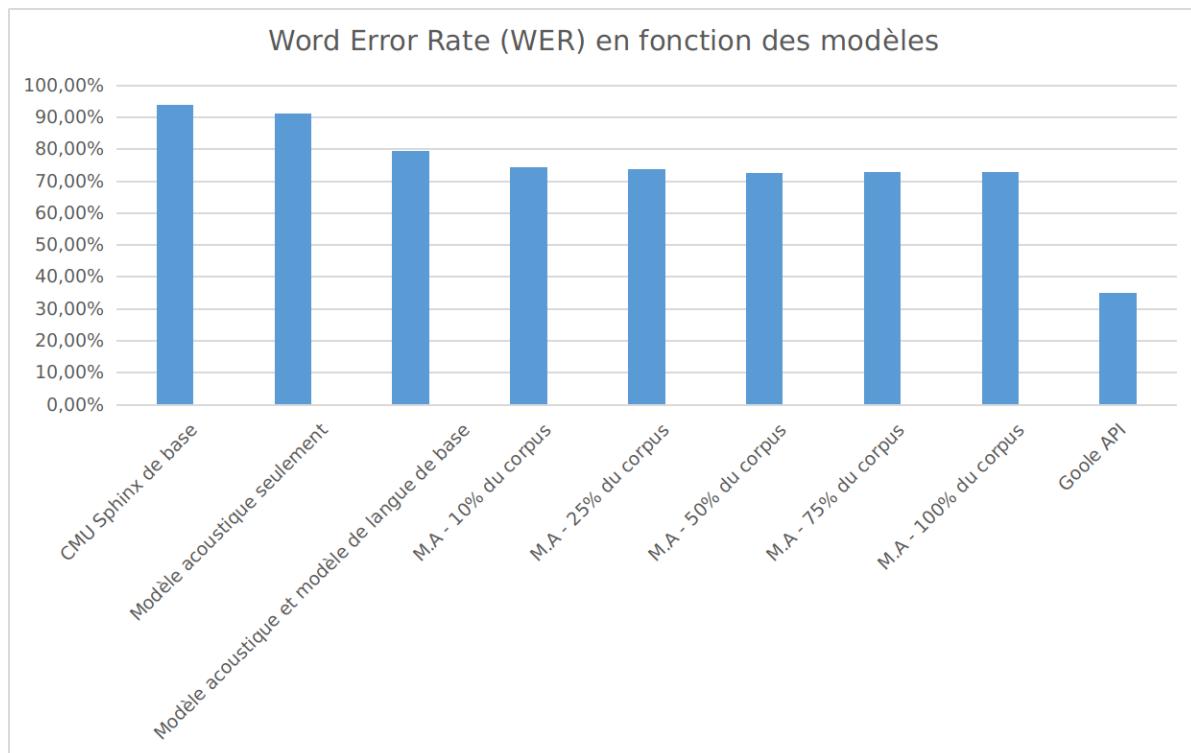


Figure 4.3 – Graphe récapitulatif des résultats pour le module de reconnaissance automatique de la parole

4.4 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 4.4.2. 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.

4.4.1 Ensemble de test

Comme mentionné dans le chapitre précédent (voir la section 3.4.3), 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 Speact 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 4.4, 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 4.2 ci dessous.

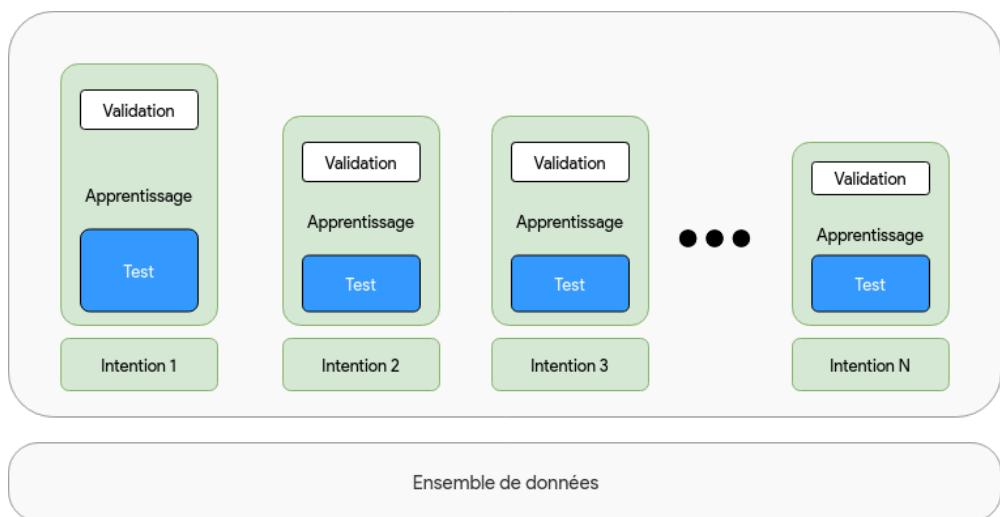


Figure 4.4 – Schéma de découpage des données pour l'apprentissage du modèle de compréhension du langage naturel

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 4.3 – 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 4.3 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 4.4 – Résultats des tests pour un encodage sans étiquetage morphosyntaxique avec des cellules BiLSTM avec découpage Apprentissage : 25%, Validation : 10%, Test : 75%.

Dans le tableau 4.4, 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 à 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 4.5 – 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 4.6 – 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 4.5 et 4.6, 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.

D'après les résultats des tableaux 4.7 et 4.8, le choix de l'architecture BiLSTM a 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	0,9707	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 4.7 – 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 4.8 – Résultats des tests pour un encodage sans étiquetage morphosyntaxique avec des cellules BiLSTM 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	0,9729	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 4.9 – 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 4.10 – Résultats des tests pour un encodage avec étiquetage morphosyntaxique avec des cellules BiLSTM avec découpage Apprentissage : 50%, Validation : 10%, Test : 50%.

Pour les tableaux 4.9 et 4.10, 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 4.11 – Résultats des tests pour un encodage sans étiquetage morphosyntaxique avec des cellules LSTM avec découpage Apprentissage : 75%, Validation : 10%, Test : 25%.

Les tableaux 4.11 et 4.12 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 4.5 est proposée.

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

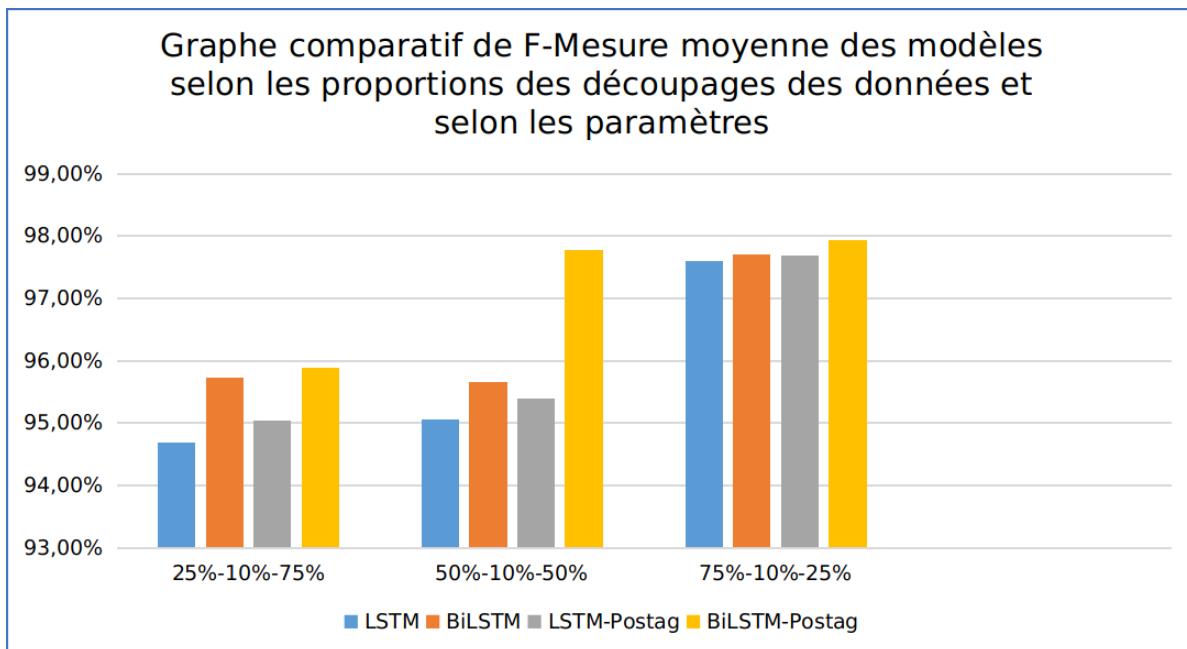


Figure 4.5 – Graphe comparatif de F-Mesure moyenne des modèles selon les proportions des découpages des données et selon les paramètres.

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.

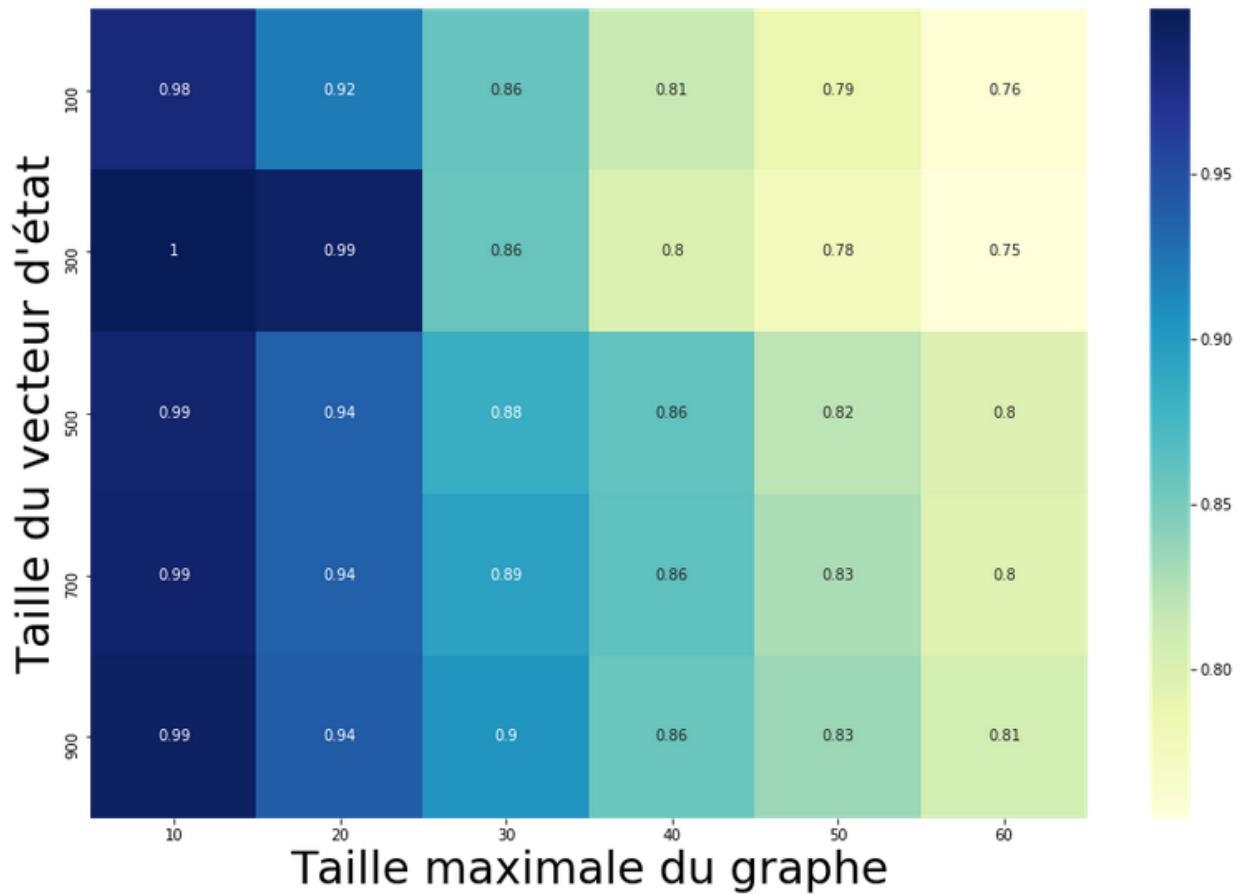


Figure 4.6 – Variation de la précision en fonction de la taille maximale du graphe et la taille du vecteur d'état

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.

4.6.2 Apprentissage par renforcement

Nous allons à présent présenter les deux méthodes citées dans la section 4.6. 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.

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 4.14 – 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 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 résigner 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.

4.6.2.2 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. Les résultats sont présentés dans le tableau 4.15.

- 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 prédecesseur.

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 4.15 – 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.

4.6.2.3 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.

4.6.2.4 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'in-

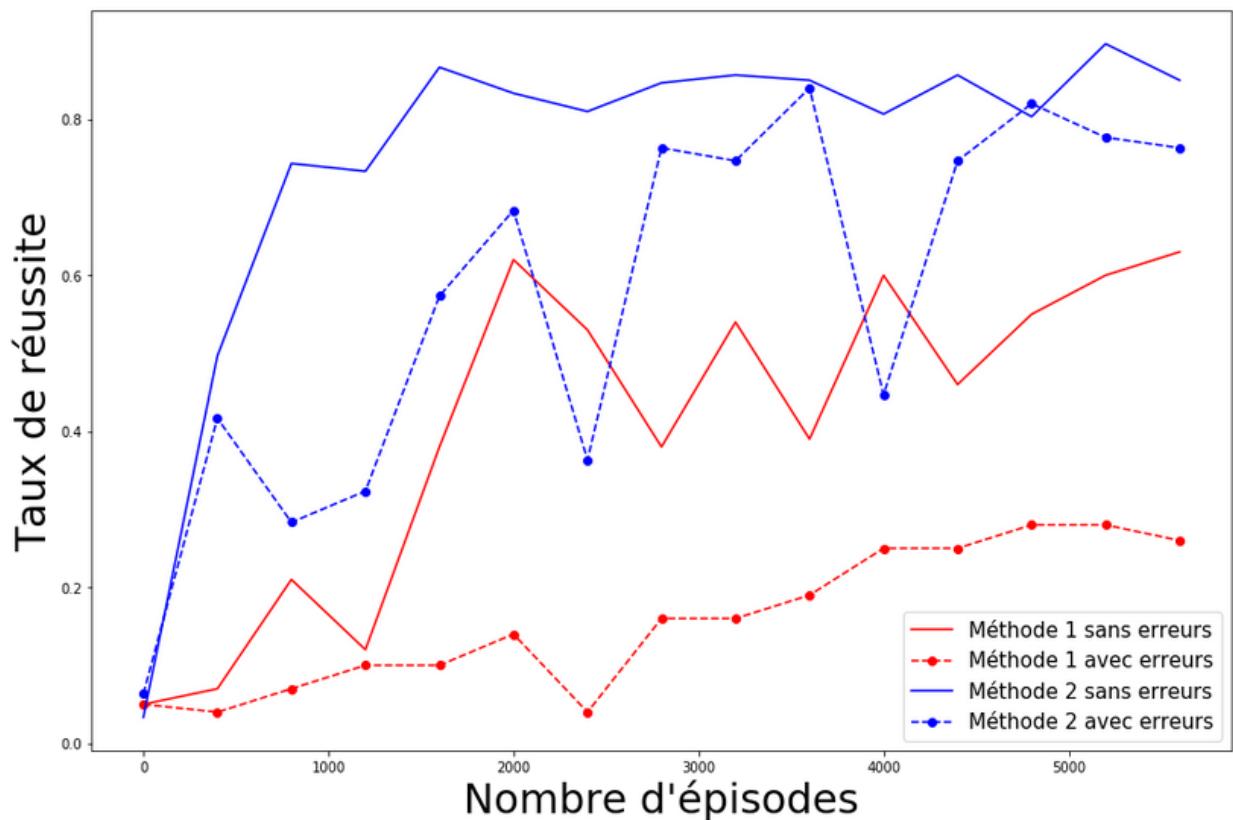


Figure 4.7 – Les courbes d'apprentissage des deux méthodes proposées avec et sans erreurs du simulateur d'utilisateur

4.7 Application Speact

L'application Speact relie les modules que nous avons implémentés dans un assistant qui aide à manipuler les fichiers de l'ordinateur avec la voix. Le schéma 4.8 montre les différentes parties que comporte l'application et les communications entre elles.

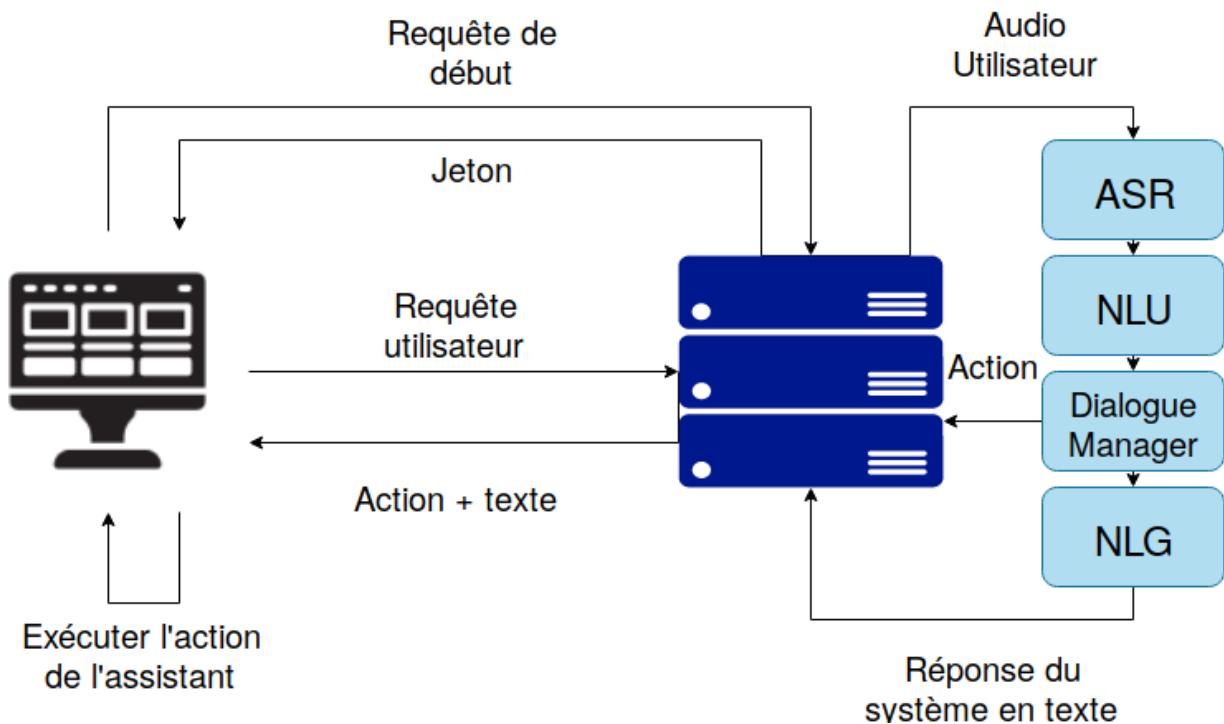


Figure 4.8 – Schéma général de l'application Speact

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.

4.7.1 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.

4.7.2 Frontend

Pour la réalisation de l'interface de l'application, qui est présentée dans la figure 4.9, 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'événements dont l'état interne est géré entièrement par le navigateur et le moteur VueJS.

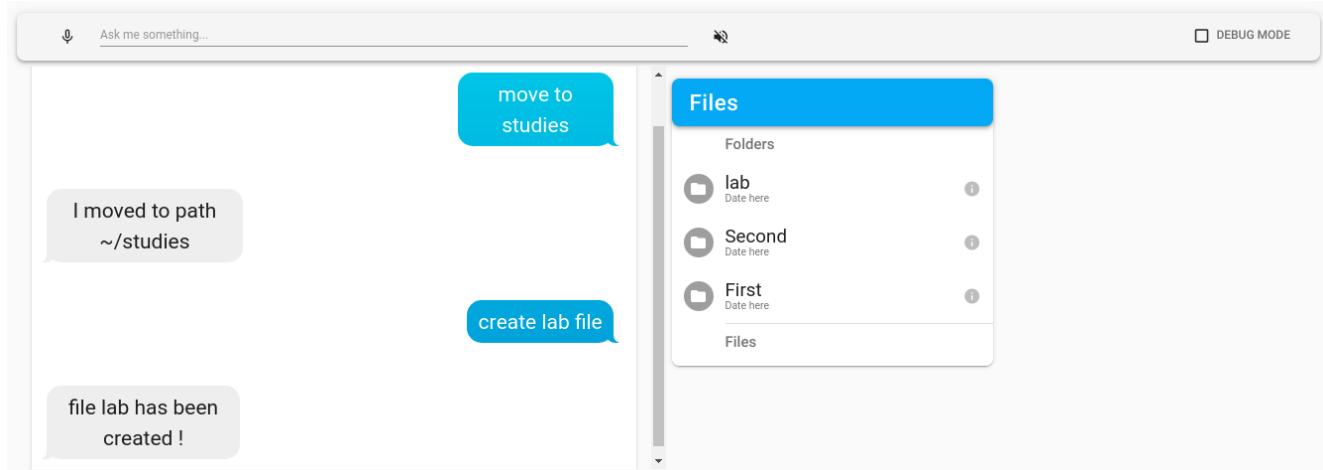


Figure 4.9 – Interface principale de Speact

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** : Comme le montre la figure 4.10, il s'agit d'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.

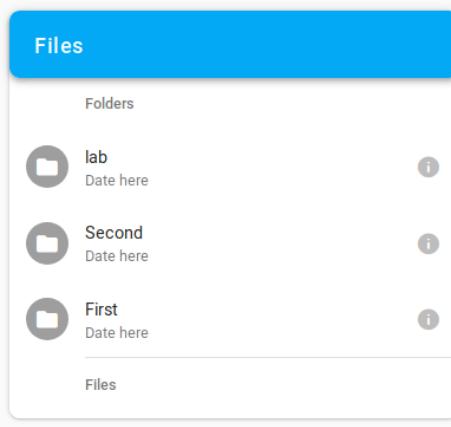


Figure 4.10 – Fenêtre d'affichage de l'arborescence virtuelle

- **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 Speact (voir la figure 4.11). Il permet ainsi de garder trace de tous les échanges effectués et pouvoir à tout moment vérifier ou réutiliser des messages.

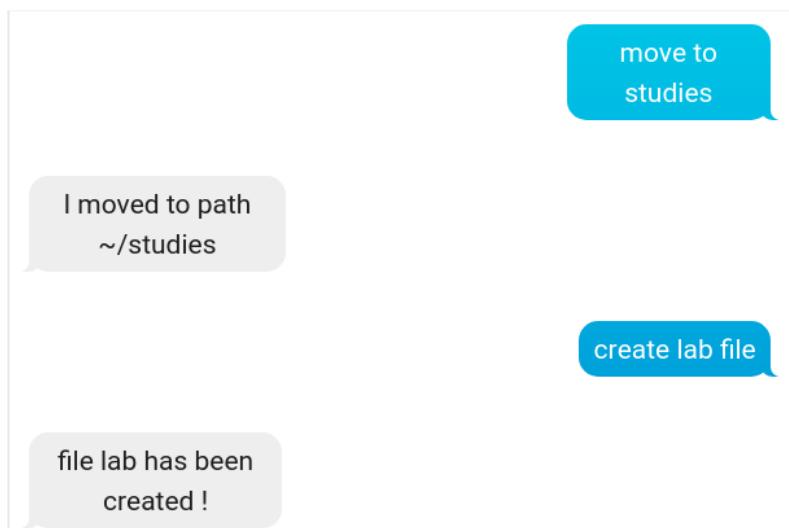


Figure 4.11 – Fenêtre de dialogue avec l'assistant

- **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, et comme le montre la figure 4.12, il y est affiché l'intention de l'utilisateur avec son degré de confiance généré par Speact, ainsi que les arguments de la requête (nom, type, positions ...).



The screenshot shows a window titled "Debug" containing a JSON object. The JSON structure is as follows:

```
{
  "confidence": 0.98,
  "intent": "create_file_desire",
  "slots": [
    {
      "name": "alter.file_name",
      "position": [
        7,
        9
      ],
      "value": "lab"
    }
  ]
}
```

Figure 4.12 – Fenêtre de Débogage

4.8 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 Speact. 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, Speact 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.