Chapitre 1

Composants de base d'un assistant personnel

1.1 Introduction

Dans ce chapitre, nous allons détaillerons un peu plus l'aspect technique d'une architecture typique pour un SPA. Nous commencerons d'abord par définir des notions de base. Nous ne traiterons que celles qui ont été les plus utilisées dans les travaux que nous avons examiné. La suite du chapitre sera organisée en sections qui décrirons chacune le fonctionnement d'une partie du SPA, en citant les travaux et références qui relatent de cette dernière. Nous terminerons sur une conclusion qui introduira le chapitre suivant.

1.2 Schéma global d'un SPA

Durant nos lectures des différents travaux sur le domaine, nous avons pu dresser un schéma assez général qui englobe les principaux modules d'un SPA :

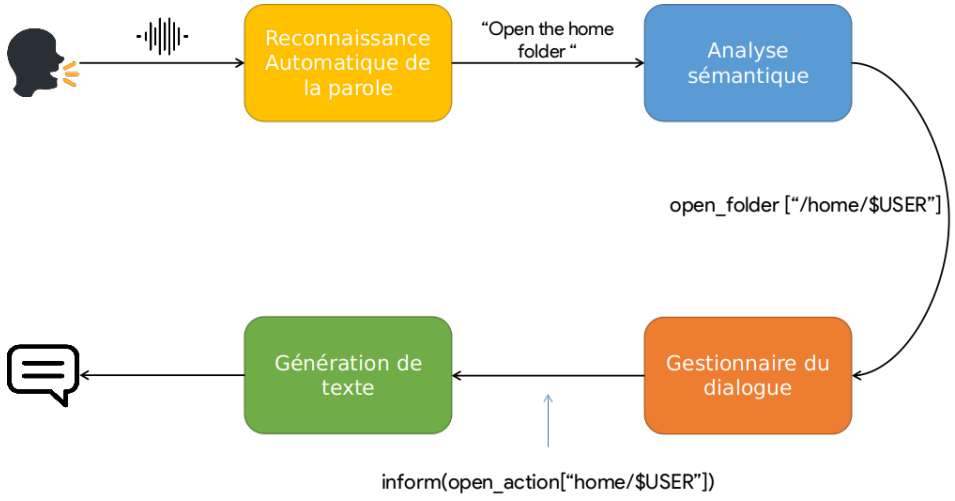

Figure 1.1 – *Schéma abstractif d'un SPA [58]*

Le processus peut être résumé en des étapes cruciales (qui seront détaillées dans les sections suivantes) :

- L'utilisateur énonce une requête en utilisant sa voix. Le signal est ensuite transformé en sa version textuelle.
- La requête étant sous un format textuel brut ne peut pas être interprétée ou comprise par la machine. Une représentation sémantique est générée qui regroupera les principales informations contenues dans la requête, à savoir l'intention de l'utilisateur et ses arguments.
- Au fur et à mesure que l'utilisateur énonce des requêtes, le système doit pouvoir être capable d'utiliser le contexte du dialogue pour interagir avec ce dernier afin d'atteindre le but final du dialogue (exécuter une commande système, trouver des informations internet ou sur la machine locale ...). 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 1.9) 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.

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.

1.3 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 utilise.

1.3.1 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 [55], l'auteur défini ce problème sous forme d'une question :

> "How can we build computer systems that automatically improve with experience, and what are the fundamental laws that govern all learning processes ?"[55]

que nous pouvons traduire par :

> “Comment pourrions nous développer un système informatique qui pourrait s’améliorer à travers l’expérience, et quelles seraient les lois qui régiraient le processus d’apprentissage ?”

D’après **(author ?)** [55], l’apprentissage automatique est un paradigme qui stipule qu’un système informatique peut apprendre à effectuer un ensemble de tâches T, sachant qu’il dispose d’un ensemble de données E, tout en améliorant sa performance P.

Il existe plusieurs sous-catégories d’apprentissage automatique. Elles diffèrent principalement par la manière dont le système apprend, du type de données sur lesquelles il apprend. ainsi que du but de son apprentissage (classification, régression,...). Nous pouvons citer les catégories suivantes :

- **Apprentissage supervisé** : Les algorithmes de cette catégories ont besoin d’une assistance externe. Les données doivent être séparées en deux parties (ensemble d’apprentissage et de test). Un *label* (ou classe) est associé à chaque instance pour permettre à l’algorithme de calculer un certain taux d’erreur qu’il essayera de minimiser au fur et à mesure qu’une nouvelle instance lui est présentée [46]. Idéalement, le système pourra apprendre une certaine fonction $\hat{f} : X \rightarrow Y$ qui liera les entrées X aux sorties Y en minimisant l’erreur E_Y
- **Apprentissage non supervisé** : Ici, les algorithmes ne disposent pas d’un étiquetage des données. Ils essayeront donc d’apprendre des *pattern* ou motifs fréquents pour grouper les données similaires. De tels algorithmes ne se préoccupent pas de la classe, mais de la similarité entre un groupe de données [9].
- **Apprentissage par renforcement** : Cette dernière catégorie regroupe des algorithmes qui apprennent par un système de *trial and error* (Essais et erreur). C’est à dire qu’en interagissant avec l’environnement. l’agent apprenant (l’algorithme) apprendra à accomplir une tâche précise en exécutant des actions qui modifierons sont état et celui de son environnement (voir la section 1.6.3.3)

Dans les sections suivantes, nous nous intéresserons à quelques type d’algorithme d’apprentissage automatique.

1.3.2 Réseaux de neurones artificiels

Un réseau de neurones artificiels est une structure d’appariement non-linéaire entre un ensemble de caractéristiques en entrée et sortie inspiré de la façon dont les systèmes nerveux biologiques fonctionnent. Ils permettent de modéliser les relations sous-jacentes des données. Ils sont composés d’un nombre arbitrairement large de petites unités de calcul interconnectées appelées neurones, qui traient l’information d’une manière parallèle dans le but de résoudre un problème bien spécifique [73]. Ils ont notamment connu un très grand succès lors des dernières années dans différents domaines comme la reconnaissance d’images [76], la reconnaissance automatique de la parole [36, 86] ou encore la classification de textes [78, 54].

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

1.3.2.1 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 1.2. Elle se compose de trois parties :

- **Une couche d'entrée :** cette couche est composée d'un ensemble de neurones ou *perceptrons* [69]. Elle représente l'information en entrée codifié en un vecteur numérique χ
- **Une ou plusieurs couches cachées :** le cur du réseau, c'est une succession de couche de neurones où chaque couche ω_i reçoit un signal sous forme d'une ou plusieurs valeurs numériques depuis une couche antérieure ω_{i-1} ou bien la couche d'entrée χ, puis envoit en sortie un autre signal de même nature qui est une combinaison non-linéaire du signal en entrée vers une couche suivante ω_{i+1} ou bien la couche de sortie
- **Une couche de sortie :** elle permet de calculer une valeur y qui peut être vu comme la prédiction du modèle Φ par rapport à son entrée χ :

$$y = f_{\Phi}(\chi) \tag{1.1}$$

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 [60]

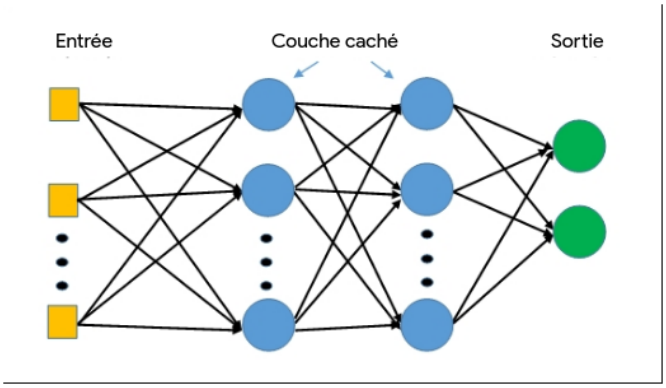

Figure 1.2 – *Architecture basique d'un réseaux de neurones multicouches*

1.3.2.2 Réseaux de neurones profonds

Les réseaux de neurones à une seule couche sont dit *shallow*. Ils présentaient l'avantage d'être assez rapide durant la phase d'apprentissage. Cependant les limites computationnelles d'antan se sont vu vite brisées avec le développement de processeurs plus puissants. De plus, avec l'explosion des données sur internet, toutes les conditions nécessaires étaient

réunies pour la mise en place d'architectures plus complexes. Les réseaux de neurones profonds sont une adaptation des réseaux multi-couches classiques avec généralement plus de 3 couches cachées.

Même si le principe reste le même, l'apprentissage profond est puissant car les architectures les plus complexes permettent d'extraire automatiquement les caractéristiques qui sont importantes mais non visibles, c'est le cas des réseaux de neurones convolutif et récurrents [49]

1.3.2.3 Réseaux de neurones récurrents

Un aspect que les réseaux de neurones (profonds ou pas) ne peuvent capturé est la notion de séquentialité. En effet beaucoup de problèmes qui sont de nature séquentielle ne peuvent être modélisés par les architectures dites classiques, comme l'analyse d'un texte. L'introduction d'une notion de séquence permet donc de capturer des dépendances entre certains états et leurs voisins, on parle ici de contexte [52].

Les réseaux de neurones récurrents (RNNs) sont des réseaux de neurones *feedforward* dont certaines connections en sortie sont réintroduite comme entrées dans le réseau durant une étape ultérieure du processus d'apprentissage. Ceci introduit la notion de temps dans l'architecture. Ainsi à un instant t, un neurone récurrent recevra en entrée la donnée $x^{(t)}$ ainsi que la valeur de l'état caché $h^{(t-1)}$ résultante de l'étape précédente du réseau, la valeur en sortie $\hat{y}^{(t)}$ est calculée en fonction de l'état caché $h^{(t)}$. Les équations suivantes montrent les calculs effectués [**?**] :

$$h^{(t)} = \tanh(W^{hx} \times x^{(t1)} + W^{hh} \times h^{(t-1)} + b_h) \tag{1.2}$$

$$\hat{y}^{(t)} = softmax(W^{hy} \times h^{(t1)} + b_y) \tag{1.3}$$

Où W^{hx} est la matrice de poids entre la couche d'entrée et l'état caché et W^{hh} est la matrice de poids de récurrence (entre l'état t et $t-1$). Les deux vecteurs b_h et b_y sont les vecteurs de biais et $\times$ est l'opération du produit matriciel[52]

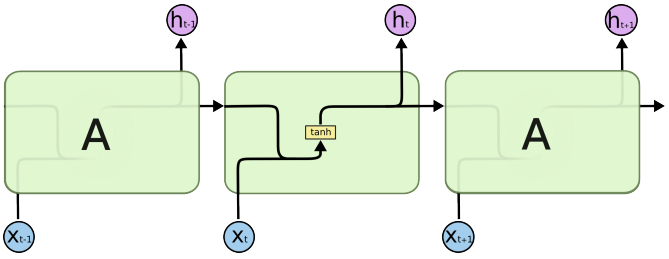

Figure 1.3 – *Architecture interne d'un réseau de neurones récurrent à un instant t [62]*

Un des principaux problèmes que rencontrent les RNNs est celui du *Vanishing gradient* traduit par le *Problème de disparition du gradient* [41]. Les relations à long terme entre les séquences ne sont donc pas capturées. Ainsi, pour remédier à ce problème, des architectures de réseaux de neurones dotées d'un module de mémoire ont étés introduites.

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

Introduite en 1997 par *Sepp Hochreiter* et al. dans [42], 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 LSTM, les composantes d'une cellule LSTM (voir 1.4) sont détaillés dans ce qui suit :

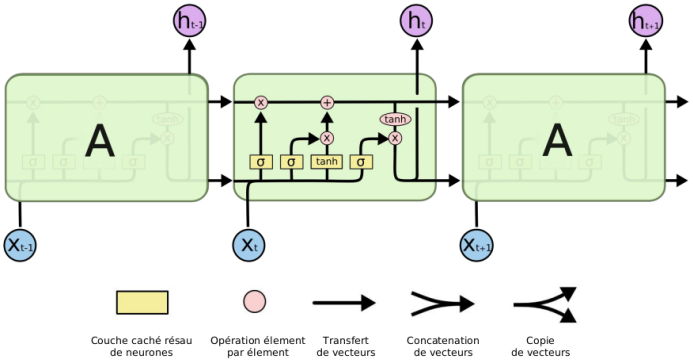

Figure 1.4 – *Architecture interne d'une cellule mémoire dans un réseau LSTM [62]*

- **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 pondérée 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 C_t [42].

$$\begin{aligned} i_t &= \sigma(W_i \times [h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= \tanh(W_C \times [h_{t-1}, x_t] + b_C) \end{aligned} \tag{1.4}$$

- **Porte d'oubli (Forget gate) :** De manière similaire, cette porte permet de spécifier au fur et à mesure de l'apprentissage les informations à oublier, qui sont donc peu importantes [42, 52].

$$f_t = \sigma(W_f \times [h_{t-1}, x_t] + b_f) \tag{1.5}$$

- **État interne de la cellule (Internal cell's state) :** C'est une sorte de convoyeur qui fait circuler l'information à travers la cellule. Cet état est mis à jour à travers la combinaison des deux valeurs précédemment filtrées par les portes f_c et i_c [42].

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{1.6}$$

- **Porte de sortie (Output gate) :** Pour renvoyer un résultat comme état interne du réseau, la cellule filtre son vecteur d'état et le combine avec la donnée en entrée et l'état du réseau précédent pour ne laisser passer que certaines informations [62, 42, 52].

$$\begin{aligned} o_t &= \sigma(W_o \times [h_{t-1}, x_t] + b_o) \\ h_t &= o_t * \tanh(C_t) \end{aligned} \tag{1.7}$$

1.3.3 Modèle de Markov caché (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 assumé 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 [30, 44].

D'un façon plus formelle, un HMM est un 5-tuples $< (S, V, \Pi, A, B >$ [64] où :

- $S = \{s_1 ... s_N\}$ est l'ensemble fini des N états du processus sous-jacent.
- $V = \{v_1 ... v_M\}$ est l'ensemble des M symboles qui constitue 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 $t = 0$, bien évidement $\sum_{i=1}^{N} \pi_i = 1$
- $A = \{a_{ij}\}$ 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 $i, j = 1 ... N$.
- $B = \{b_i(v_k)\}$ est l'ensemble des distribution 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 $i = 1 ... N$.

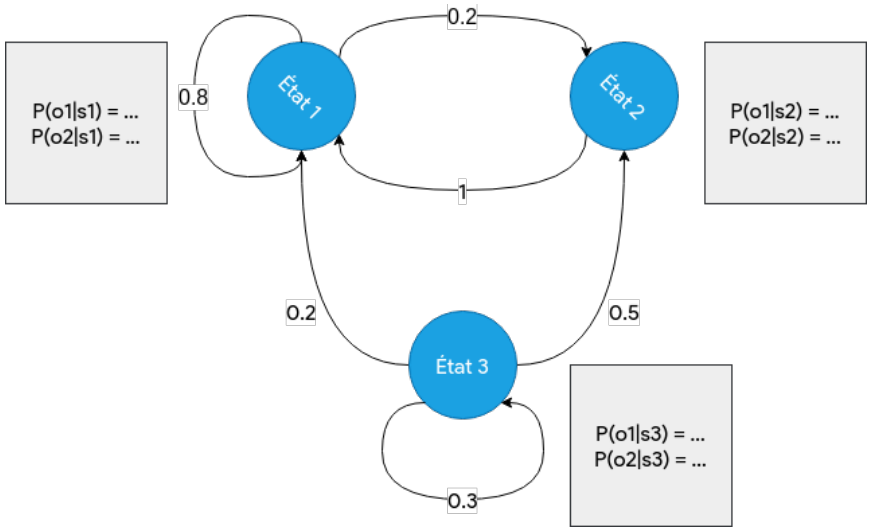

Figure 1.5 – *Exemple d'une distribution de probabilité de transition ainsi qu'une distribution de probabilité d'observations pour un HMM à 3 états et 2 observations*

Le but serait de trouver la séquence d'états $S_{1:K}$ qui maximise la probabilité [30] :

$$P(Y_{1:K}|O_{1:K}) = P(O_1)P(O_1|S_1)\prod_{t=2}^{K} P(S_t|S_{t-1})P(O_t|S_t) \tag{1.8}$$

Pour y parvenir d'une manière efficace, un décodeur qui implémente l'algorithme de viterbi peut être utilisé [27, 13]

1.4 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 [6]. Différentes approches ont été développées au cours des années. Une architecture s'est ensuite dégagé 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 [13],puis la séquence d'observations passe par le HMM et la meilleure séquence de mots est sélectionnée [86].

Une architecture assez générale s'est dégagée. Quatre modules sont impliqués dans le processus de la reconnaissance, nous les citerons dans les sections qui suivent en énumérant les modèles utilisés dans chacun.

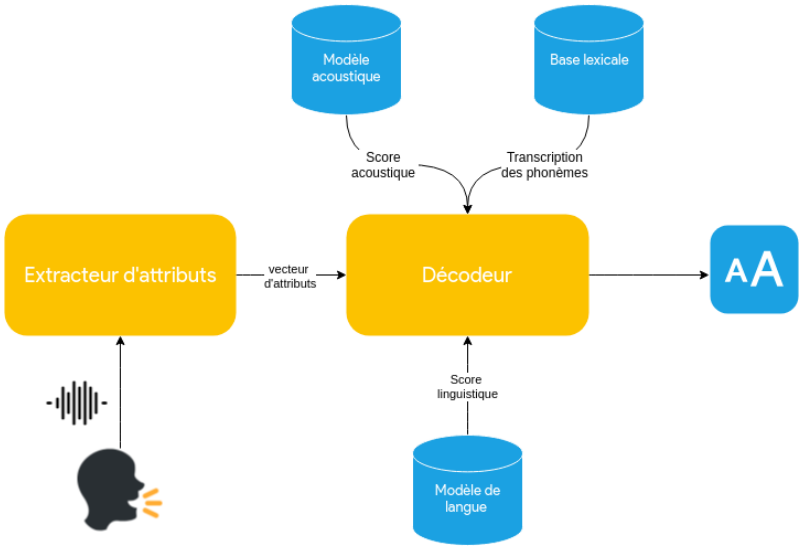

Figure 1.6 – *Architecture des systèmes de reconnaissance de la parole [86]*

1.4.1 Acquisition du signale 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 appliquée pour engendrer un spectre de magnitude qui sera ensuite passé à un module de transformation en spectre de Mel (Mel-Spectrum) qui est une sorte de changement d'échelle. Finalement, une transformation inverse de fourrier est appliquée pour engendrer le Mel-Cpestrum qui est le vecteur d'attributs que nous recherchions [61]. Un tel processus peut utiliser plusieurs techniques pour la mise à l'échelle : MFCC (Mel-frequency cepstrum) [16], LPC (Linear Predictive coding) [63] et RASTA (RelAtive SpecTrAl) [65], chacune possédant ses avantages et ses inconvénients.

1.4.2 Modélisation acoustique et modélisation du lexique

C'est le cur du système de reconnaissance. Le but est de construire un modèle permettant d'identifier une séquence de phonèmes à partir d'une séquence d'observations de vecteurs d'attributs. Ceci peut être fait en utilisant un modèle HMM et en disposant d'un corpus de fichiers audio accompagnés de leurs transcriptions en phonèmes et en texte [31, 64], ou bien un réseau de neurones profonds [86], ou encore une hybridation de ces derniers [21]. Le modèle acoustique procède après la reconnaissance de la séquence de phonèmes au décodage de ce dernier. Ceci est effectué en utilisant un dictionnaire linguistique qui transcrit chaque mots du vocabulaire en phonèmes qui le constituent. Ceci peut conduire à un cas d'ambiguïté où plusieurs mots peuvent avoir la même transcription phonétique (ou bien aucun mot ne correspond à cette séquence). Pour lever cette ambiguïté un modèle de langue est nécessaire pour décider quelle est la séquence de mots la plus probable qui coïncide avec la séquence de phonèmes observée.

1.4.3 Modélisation de la langue

Cette étape consiste à modéliser les aspects linguistiques de la langue que le système essaye de traiter. Souvent ces aspects sont spécifique au domaine d'application afin de restreindre l'espace de recherche des mots reconnaissables par le système afin de déterminer la séquence de mots en sortie la plus plausible. Les modèles utilisés sont basés soit sur des grammaires à contexte libre dans le cas où les séquences de mots reconnaissables sont peu variées et peuvent être modélisées par le biais de règles de la langue [32]. Dans un cadre plus récent, des modèles probabilistes basés sur les N-grammes sont utilisés.

Très souvent quand il est nécessaire de traiter un ensemble de mots, phrases ou bien toute entité atomique qui constitue une séquence, il est intéressant de pouvoir assigner une probabilité de vraisemblance à une séquence en particulier (par exemple une séquence de mots).

Dans un contexte textuel, un N-gramme est une suite de N mots consécutifs qui forment une sous-chaîne $S\prime$ d'une chaîne de caractères S. Un exemple d'un ensemble de bi-grammes

(2-grammes) pour la phrase "Ouvre le fichier *home*" serait donc : (Ouvre,le), (le,fichier), (fichier,*home*).

Ainsi, en disposant d'un corpus de texte assez large et diversifié et d'une méthode de comptage efficace, il serait possible de calculer la vraisemblance d'apparition d'un mot w après une certaine séquence de mots t sous forme d'une probabilité P [43].

$$P(w|t) = \frac{Comptage(t, w)}{Comptage(t)} \tag{1.9}$$

Disposant d'un volume de données textuelles assez conséquent, l'utilisation de modèle basés sur les N-grammes ont prouvé leurs utilité [43, 86, 68] .

1.5 Compréhension du langage naturel (NLU)

Après avoir obtenu la requête de l'utilisateur, le module qui prend le relais est celui de la compréhension du langage naturel. Ce domaine fait partie du traitement automatique du langage naturel (NLP). Il se focalise principalement sur les tâches qui traitent le niveau sémantique voir même pragmatique de la langue tel que la classification de texte, l'analyse de sentiments ou encore le traitement automatiques des e-mails. Dans le contexte d'un SPA, le but final d'un module de compréhension du langage naturel est de construire une représentation sémantique de la requête de l'utilisateur. Étant donné une telle requête préalablement transformé en texte brut dans un langage naturel, le module essayera d'en extraire une information primordiale qui est l'intention du locuteur.

1.5.1 L'intention

L'intention est un élément clé dans notre travail. Nous allons donc la définir et expliquer pourquoi elle a été choisis comme abstraction sémantique d'une requête d'un utilisateur.

Une intention (ou intent en anglais) est une représentation sémantique d'un but ou d'un sous-but de l'utilisateur, plusieurs requêtes de l'utilisateur peuvent avoir le même intent, ce qui rend l'utilisation d'une telle représentation essentielle pour que la machine puise mieux comprendre l'utilisateur. La motivation vient du fait que la machine ne peut pas comprendre une requête formulé dans un langage non-formel. Il a fallu trouvé un moyen de communication universel entre la machine et l'utilisateur de telle sorte que l'un puisse comprendre l'autre sans parler sa langue. L'intermédiaire entre ces deux parties est un traducteur bidirectionnel requête en langage naturel vers requête dans un langage formel et inversement.

Le choix de l'intention doit être minutieusement étudié au préalable pour ne pas trop subdiviser un type d'intention, tout en gardant un certain degré de spécificité. À titre d'exemple, si deux intentions $Ouvrir_fichier$ et $Ouvrir_document$ existent, on retrouve ici 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.

Un autre cas est celui du sur-découpage d'intentions déjà assez semblable, $Ouvrir_fichier$ et $Ouvrir_Repertoire$ peuvent être regroupés en une seul intention $Ouvrir_Entit$. Bien-sur ces choix sont purement conceptuel 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 réaliser [53, 34]

1.5.2 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 basé sur les réseaux de neurones récurrents à mémoire court et long terme (LSTM) (voir la section 1.4). Dans [53] et [34] deux architectures ont fait leurs preuves. Dans le premier travail, les auteurs ont utilisé une architecture BiLSTM (LSTM Bidirectionnel) pour capturer les contextes droit et gauche d'un mot donné [71]. Le dernier état caché retourné par la cellule LSTM est ensuite utilisé pour la classification. Une couche d'attention est ajoutée [19] pour permettre au modèle de se focaliser sur certaines parties du texte en entrée. La deuxième architecture est un peu plus simpliste. 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.

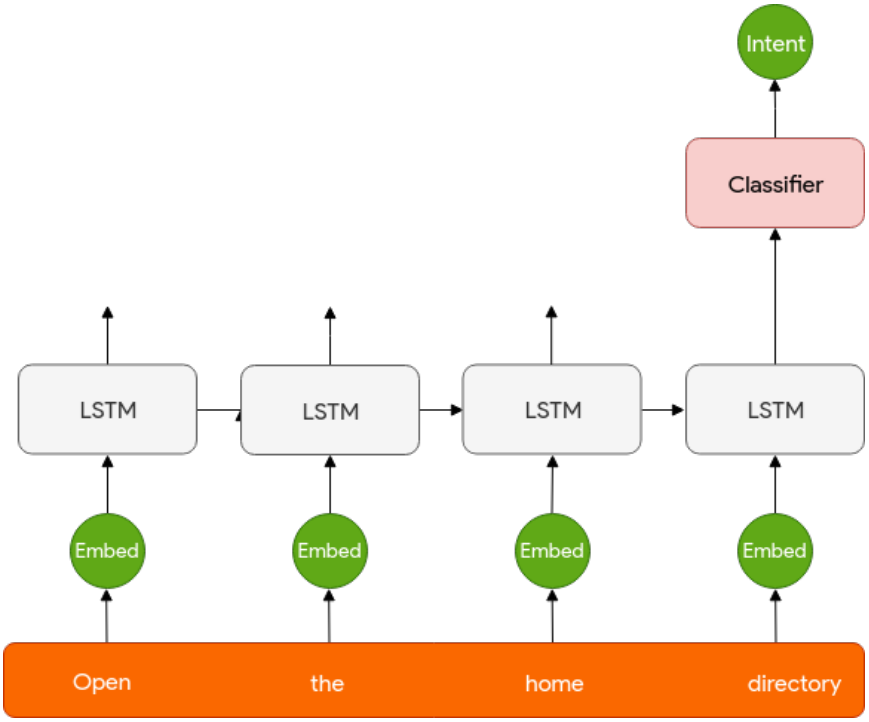

Figure 1.7 – *Architecture de base d'un classificateur d'intents [53]*

1.5.3 Extraction d'entités

Déterminer l'intention d'un utilisateur ne s'arrête pas qu'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 (nommés ou spécifiques du domaine) qui seront des arguments de l'intent identifié. Cette tâche consiste donc à, pour un intent I donné, extraire les arguments $Args$ qui lui sont appropriés depuis le texte de la requête. Plusieurs approches sont possibles, dans [34] les auteurs ont taclé le problème comme étant la traduction d'une séquence de mots en entrée en une séquence d'entités en sortie, le schéma suivant explicite le processus :

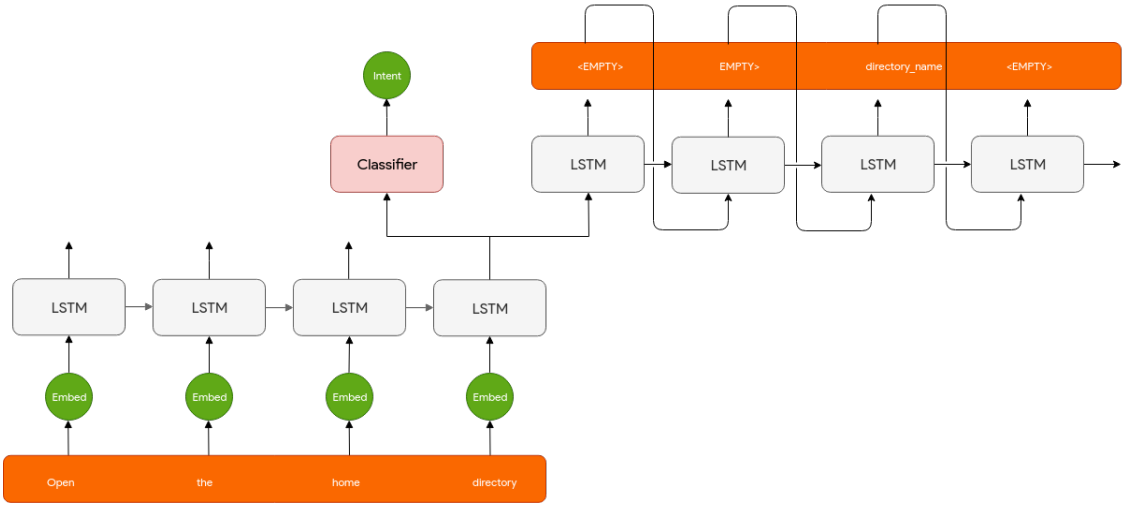

Figure 1.8 – *Architecture de base d'un classificateur d'intents doublé d'un extracteur d'entité [34]*

1.5.4 Analyse sémantique

Disposant des deux modèles précédemment cités, le module NLU peut maintenant 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'intent et les arguments (slots) qui lui sont propres [53, 34, 79].

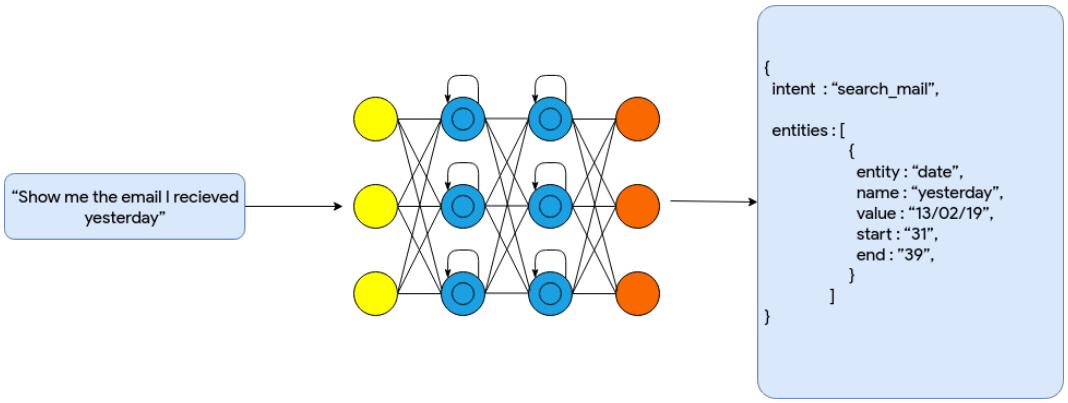

Figure 1.9 – *Exemple d'une trame sémantique pour une requête donnée*

1.6 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 1.7) pour afficher un résultat à l'utilisateur. Généralement, deux principaux modules sont présents dans les systèmes de gestion de dialogue :

- 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.

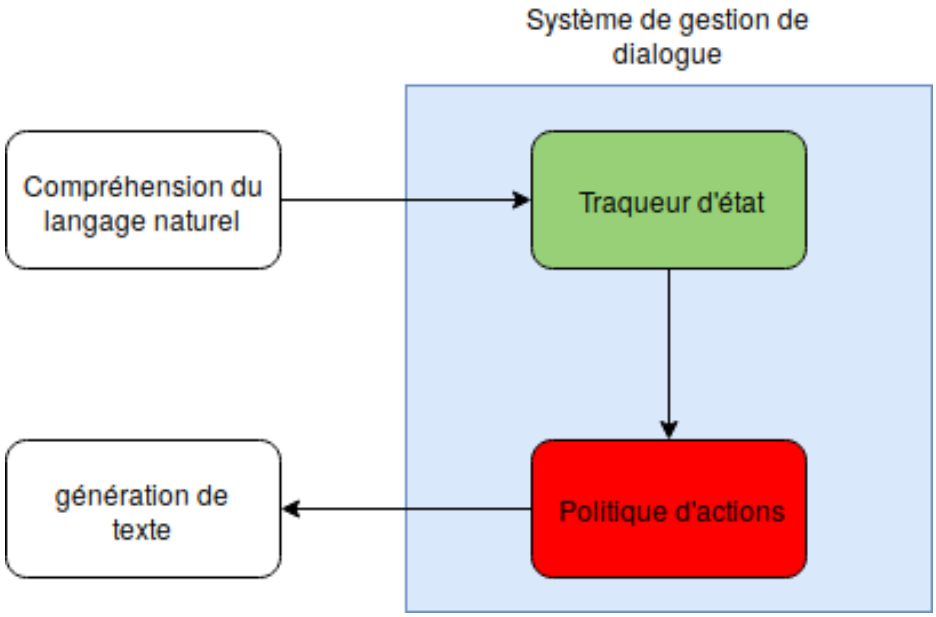

Figure 1.10 – *Schéma général d'un gestionnaire de dialogue*

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.

1.6.1 Modélisation

Pour résoudre ce problème, un gestionnaire de dialogue peut être modélisé par un processus de décision Markovien (Markovian Decision Process, MDP) [67]. 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.

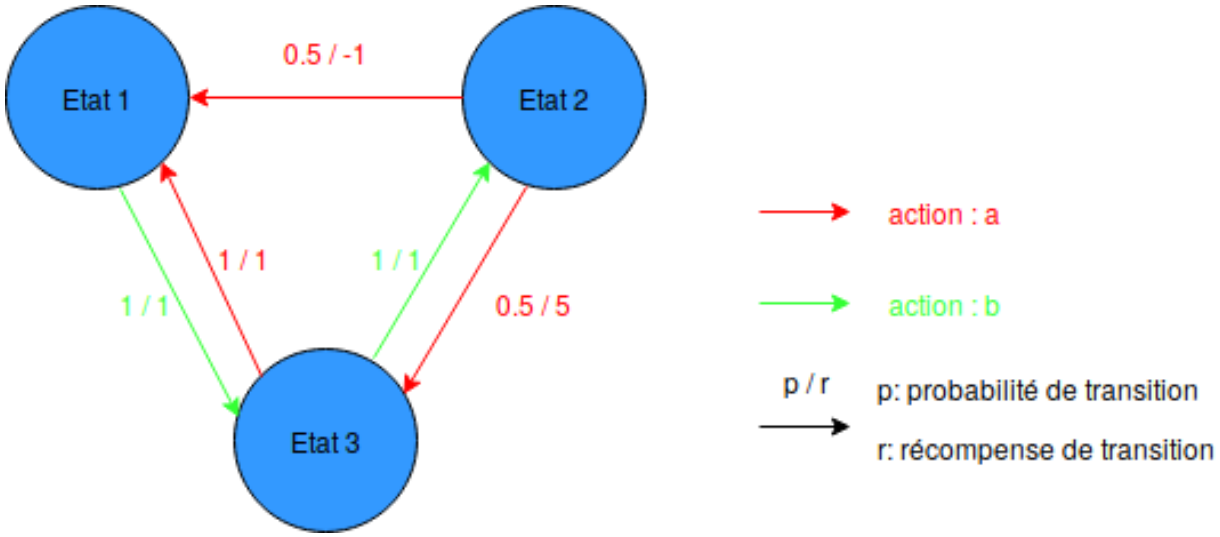

Figure 1.11 – *Schéma représentant les transitions entre états dans un MDP*

1.6.2 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

[17]. Cette structure contient des emplacements à remplir dans un domaine donné. La figure 1.12 illustre un exemple de trame sémantique.

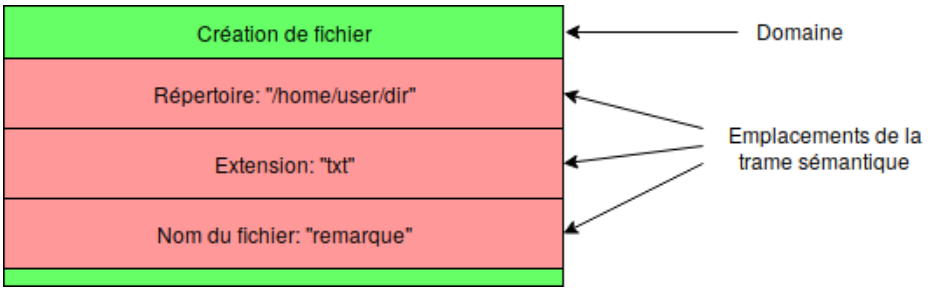

Figure 1.12 – *Schéma représentant une trame sémantique avec comme domaine : création de fichier*

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 [84].

1.6.2.1 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'arrivé d'une nouvelle information pour mettre à jour l'état [33]. Cependant, les bases de règles sont très susceptibles à faire des erreurs [17] car elles sont peu robustes face aux incertitudes.

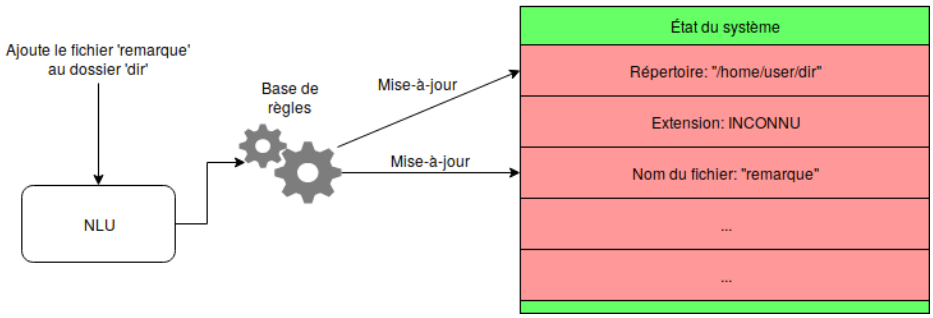

Figure 1.13 – *Schéma représentant la mise-à-jour de l'état du gestionnaire de dialogue par un système basé règles*

1.6.2.2 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) [85] 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.

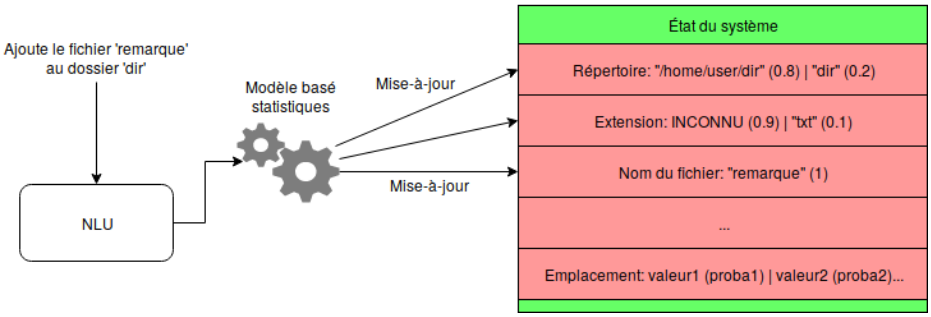

Figure 1.14 – *Schéma représentant la mise-à-jour de l'état du gestionnaire de dialogue par un système basé statistiques*

Processus de décision markovien partiellement observable (POMDP) Comme dans les processus de décision markoviens, un POMDP [8] passe d'un état à un autre en prenant une des actions possibles. Cependant, un POMDP ne connait 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 1.6.1).
- 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.

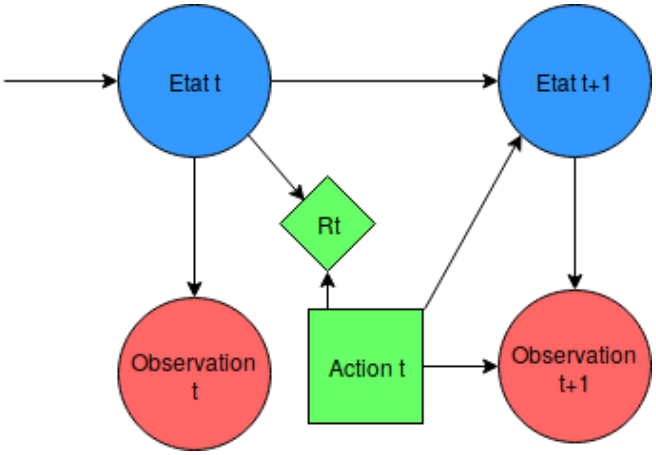

Figure 1.15 – *Diagramme d'influence dans un POMDP*

1.6.2.3 Suivi de l'état du gestionnaire de dialogue avec réseaux de neurones profonds

Récemment, des approches utilisant les réseaux de neurones profonds (voir 1.3.2.2) ont fait leur apparition. En effet, l'utilisation des architectures profondes permet de capter des

relations complexes entre les caractéristiques d'un dialogue et, ainsi, mieux estimer l'état du système. Le réseau de neurones estime les probabilités de toutes les valeurs possibles d'un emplacement de la trame sémantique [40]. En conséquence, il peut être utilisé comme modèle de suivi d'état pour un processus partiellement observable.

1.6.3 Politique de gestion de dialogue

La première partie est dédiée au module qui suit l'état du système de dialogue. Dans cette partie, nous allons présenter des approches proposées afin d'arriver au but du MDP, c'est à dire quelles actions prendre pour maximiser la somme des récompenses obtenues.

1.6.3.1 Gestion de dialogue avec une base de règles

Les premières approches utilisaient des systèmes de règles destinés à un domaine bien spécifique. Elles étaient déployées dans plusieurs domaines d'application pour leur simplicité. Cependant, le travail manuel nécessaire reste difficile à faire et, généralement, n'aboutit pas à des résultats flexibles qui peuvent suivre le flux du dialogue convenablement [50].

1.6.3.2 Gestion de dialogue par apprentissage

La résolution d'un MDP revient à trouver une estimation de la fonction de récompense afin de pouvoir choisir la meilleure action. La majorité des approches récentes utilise l'apprentissage par renforcement dans le but d'estimer la récompense obtenue par une action et un état donnés. Cette préférence par rapport aux approches supervisées revient à la difficulté de produire des corpus de dialogues [39], encore moins des corpus annotés avec les récompenses à chaque transition. Néanmoins, il existe des approches de bout en bout qui exploitent des architectures avec réseaux de neurones profonds et traitent le problème comme Seq2Seq[1] afin de produire directement une sortie à partir des informations reçues de l'utilisateur [82, 72].

1. Les modèles Seq2Seq sont des architectures de réseaux de neurones profonds qui prennent en entrée une séquence et produisent en sortie une autre séquence en utilisant les réseaux de neurones récurents (RNN).

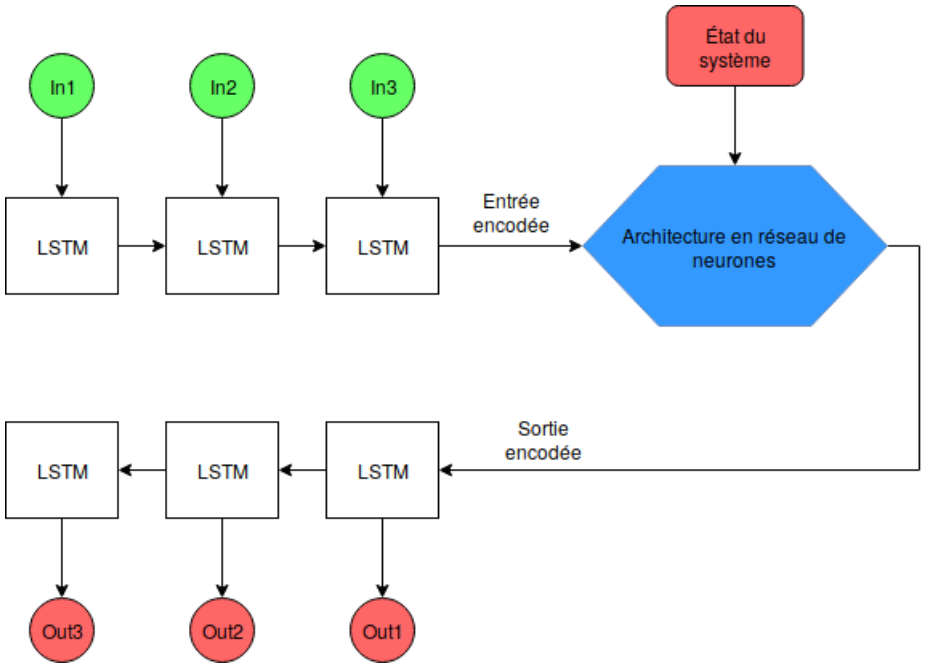

Figure 1.16 – *Schéma de gestion de dialogue de bout en bout avec architecture Seq2Seq*

1.6.3.3 Apprentissage par renforcement

L'apprentissage par renforcement est une approche qui a pour but d'estimer une politique d'actions à prendre dans un environnement donné. Cette politique doit maximiser une mesure d'évaluation qui est sous forme de récompenses obtenues après chaque action [80]. 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.

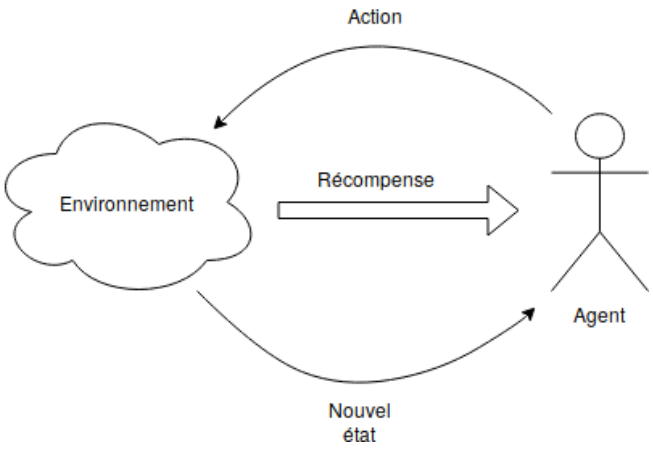

Figure 1.17 – *Schéma d'interaction agent-environnement dans l'apprentissage par renforcement*

Il existe plusieurs méthodes que l'agent peut utiliser pour estimer la fonction de récom-

pense [12], notamment Deep Q Learning (DQN) [59] qui utilise les réseaux de neurones profonds pour trouver une approximation à cette fonction. Dans cette approche, à chaque fois que l'agent prend une action, il compare la récompense reçue avec celle estimée par le réseau de neurones, et applique ensuite l'algorithme de rétro-propagation pour corriger les erreurs du réseau.

Le système de dialogue est modélisé par un MDP. Seulement, ce dernier inclut l'utilisateur comme partie de l'environnement. Par conséquent, pour appliquer l'apprentissage par renforcement, il est nécessaire qu'un utilisateur communique avec le système pour qu'il apprenne. D'où la nécessité d'utiliser un **simulateur d'utilisateur** qui est un programme capable de se comporter comme un humain interagissant avec un système de dialogue. Le simulateur doit pouvoir estimer la satisfaction de l'utilisateur après une interaction. En d'autres termes, il doit comporter une fonction de récompense. Il existe plusieurs méthodes pour créer un simulateur d'utilisateur :

- Les simulateurs d'utilisateur basés règles : dans ce cas une liste de règles est écrite manuellement et le simulateur doit les suivre pour communiquer avec le système [70].
- Les simulateurs d'utilisateur basés n-grammes : ils traitent un dialogue comme une séquence d'actions en prenant les n-1 actions précédentes pour estimer l'action la plus probable que peut prendre le simulateur à partir des statistiques tirées d'un corpus de dialogues [29].
- Les simulateurs d'utilisateur basés HMM : les états du modèle sont les états du système et les observations sont ses actions. Ainsi un HMM peut estimer l'état le plus probable du système pour prendre une action. Il existe d'autres variantes, IHMM et IOHMM, qui incluent les probabilités conditionnelles des actions de l'utilisateur sachant l'état du système ou l'action du système directement dans le modèle HMM [20].
- Les simulateurs d'utilisateur avec apprentissage par renforcement : Comme le gestionnaire de dialogue, le simulateur apprend par renforcement au même temps. Dans ce cas la fonction de récompense pour les deux agents peut être apprise à partir des dialogues humain-humain [15].

1.7 Génération du langage naturel (Natural Language Generation, NLG)

Le domaine de la génération automatique du langage naturel est l'un des domaines dont les limites sont difficiles à définir [25]. Il est vrai que la sortie d'un tel système est clairement du texte. Cependant, l'ambiguïté se trouve dans ses entrées, c'est-à-dire, sur quoi se basera le système pour générer le texte. D'après [66], la génération du langage naturel est décrite comme étant le sous domaine de l'intelligence artificielle qui traite la construction des systèmes de génération de texte à partir d'une représentation non-linguistique de l'information. Celle-ci peut être une représentation sémantique, des données numériques, une base de connaissances ou même des données visuelles (images ou vidéos). Ceci dit, d'autres travaux, comme [47], utilisent les mêmes techniques pour des entrées linguistiques. Enfin,

la génération du langage naturel peut être très proche de la gestion de dialogue [22]. En effet, le texte généré doit prendre en compte l'historique de la conversation et le contexte de l'utilisateur.
Il existe six tâches trouvées fréquemment dans les systèmes de génération de texte [66] ; nous les présentons dans cette section.

1.7.1 Détermination du contenu

Cette partie consiste à sélectionner les informations de l'entrée dont le système veut transmettre leur contenu sous forme de texte naturel à l'utilisateur. En effet, les données en entrée peuvent contenir plus d'informations que ce que l'on désire communiquer [87]. De plus, le choix de l'information peut aussi dépendre de l'utilisateur et de ses connaissances [22]. Ceci requiert de mettre au point un système qui détecte les informations pertinentes à l'utilisateur.

1.7.2 Structuration de texte

Après la détermination du contenu, le système doit ordonner les informations à transmettre. Ceci dépend grandement du domaine d'application qui peut exiger des contraintes d'ordre temporel ou de préférence par importance des idées. Les informations à transmettre en elles-mêmes sont souvent reliées par sens, ce qui implique une certaine structuration de texte à respecter.

1.7.3 Agrégation de phrases

Certaines informations peuvent être transmises dans une même phrase. Cette partie introduit des notions de la linguistique afin que le texte généré soit plus lisible et éviter les répétitions. Un exemple de cela peut être la description de la météo à Alger au cours de la matinée :

- Il va faire 16° à Alger à 7h.
- Il va faire 17° à Alger à 8h.
- Il va faire 18° à Alger à 9h.
- Il va faire 18° à Alger à 10h.

Ceci peut être agrégé en un texte plus compacte : "La température moyenne à Alger sera de 17.25°entre 7h et 10h."

1.7.4 Lexicalisation

Le système choisit les mots et les expressions à utiliser pour communiquer le contenu des phrases sélectionnées. La difficulté de cette tâche revient à l'existence de plusieurs manières d'exprimer la même idée. Cependant, certains mots ou expressions sont plus appropriés en certaines situations que d'autres. En effet, "inscrire un but" est une façon inadéquate d'exprimer "un but contre son camp" [28].

1.7.5 Génération d'expressions référentielles (REG)

Cette partie du système se focalise sur la génération d'expressions référentielles qui peuvent être entre autres : des noms propres, groupes nominaux ou pronoms et ceci a pour but d'identifier les entités du domaine. Cette tâche semble être très proche de la lexicalisation dans le sens où elle aussi a pour but de choisir les mots et les expressions ; elle s'avère néanmoins plus délicate dû à la difficulté de confier suffisamment d'information sur l'entité afin de la différencier des autres [66]. Le système doit faire un choix de l'expression référentielle en se basant sur plusieurs facteurs. Par exemple "Mohammed", "Le professeur" ou "Il" faisant référence à la même personne, le choix entre eux dépend de comment l'entité a été mentionnée auparavant, si elle l'a été, et des détails qui l'ont accompagnée.

1.7.6 Réalisation linguistique

La dernière tâche consiste à combiner les mots et expressions sélectionnés pour construire une phrase linguistiquement correcte. Ceci requiert l'utilisation des bonnes formes morphologiques des mots, les ordonner, et éventuellement l'addition de certains mots du langage afin de réaliser une structure de phrase grammaticalement et sémantiquement correcte. Plusieurs méthodes ont été proposées, principalement les méthodes basées sur des règles manuellement construites (modèles de phrases, systèmes basés grammaires) et des approches statistiques [28].

Modèles de phrases : La réalisation se fait en utilisant des modèles de phrases prédéfinis. Il suffit de remplacer des espaces réservés par certaines entrées du système. Par exemple, une application dans un contexte météorologique pourrait utiliser le modèle suivant : la température à [ville] atteint [température]° le [date].

Cette méthode est utilisée lorsque les variations des sorties de l'application sont minimales. Son utilisation a l'avantage et l'inconvénient d'être rigide. D'un côté, il est facile de contrôler la qualité des sorties syntaxiquement et sémantiquement tout en utilisant des règles de remplissage complexes [77]. Cependant, lorsque le domaine d'application présente beaucoup d'incertitude, cette méthode exige un travail manuel énorme, voire impossible à faire, pour réaliser une tâche pareille. Bien que certains travaux ont essayé de faire un apprentissage de modèles de phrases à partir d'un corpus [7], cette méthode reste inefficace lorsqu'il s'agit d'applications qui nécessitent un grand nombre de variations linguistiques.

Systèmes basés grammaire : La réalisation peut se faire en suivant une grammaire du langage. Celle-ci contient les règles morphologiques et de structures de la langue, notamment la grammaire systémique fonctionnelle (SFG) [37] qui a été largement utilisée comme dans NIGEL [56] ou KPML [10]. 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 [23] dans la figure 1.18 qui génère la phrase : "She hands the draft to the editor".

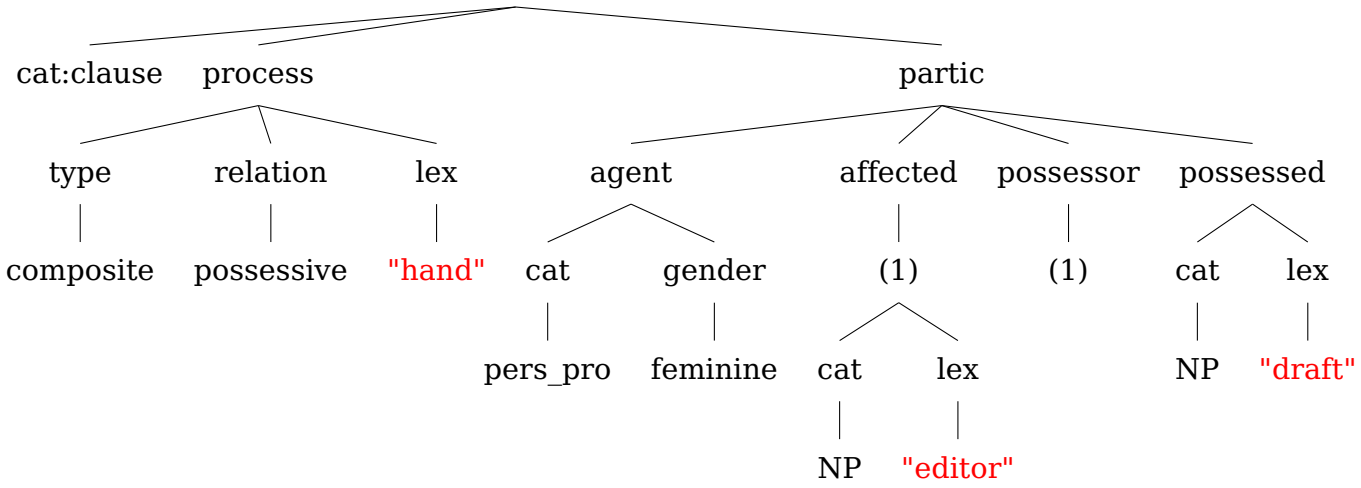

Figure 1.18 – *Un exemple d'entrée de SURGE [23]*

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 [28].

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'avan-

tage de minimiser le travail manuel tout en couvrant plus de cas de réalisation. Il existe principalement deux approches l'utilisant [28] :

- 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 par exemple [48].
- La deuxième méthode utilise les informations statistiques directement au niveau de la génération pour produire la solution optimale [11].

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 Treebanks [2] pour déduire les règles de grammaire [24].

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 voire même la possibilité qu'elle s'étend 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.

1.7.7 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 [26].

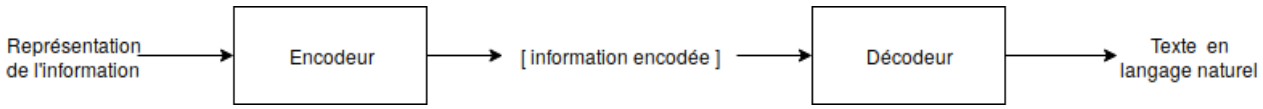

Figure 1.19 – *Schéma d'une architecture encodeur-décodeur pour NLG*

Beaucoup d'approches de génération de langage naturel en gestion de dialogue utilisent des encodeurs-décodeurs. [81] utilisent par exemple des LSTMs sémantiquement conditionnés ; ils ajoutent aux LSTMs classiques une couche contenant des informations sur

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

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 [74, 72, 35].

1.8 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.

Chapitre 2

Conception du système

2.1 Introduction

Dans ce chapitre, nous allons présenter en détail les étapes de conception de notre système Bethano. De prime abord une architecture générale est introduite puis décortiquée. Ensuite chaque module du système sera détaillé du point de vue des composants qui le constituent. Une conclusion viendra ensuite clôturer ce chapitre pour ensuite.

2.2 Architecture du système

Comme montré dans la figure 2.1 et comme cité dans le chapitre précédent (voir 1.2) le système Bethano se présente comme l'interconnexion de cinq parties dont une interface [1] et quatre modules internes communiquant entre eux. Chaque module forme ainsi un maillon d'une chaîne qui représente une partie du cycle de vie du système. L'architecture de Bethano est un pipeline (chaîne de traitement) de processus qui s'exécutent de manière indépendante mais qui font circuler un flux de données entre eux dans un format préalablement établi (voir 1.1). Nous pouvons séparer ces parties en deux catégories : la partie utilisateur et la partie interne du système que nous présentons ci-dessous.

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

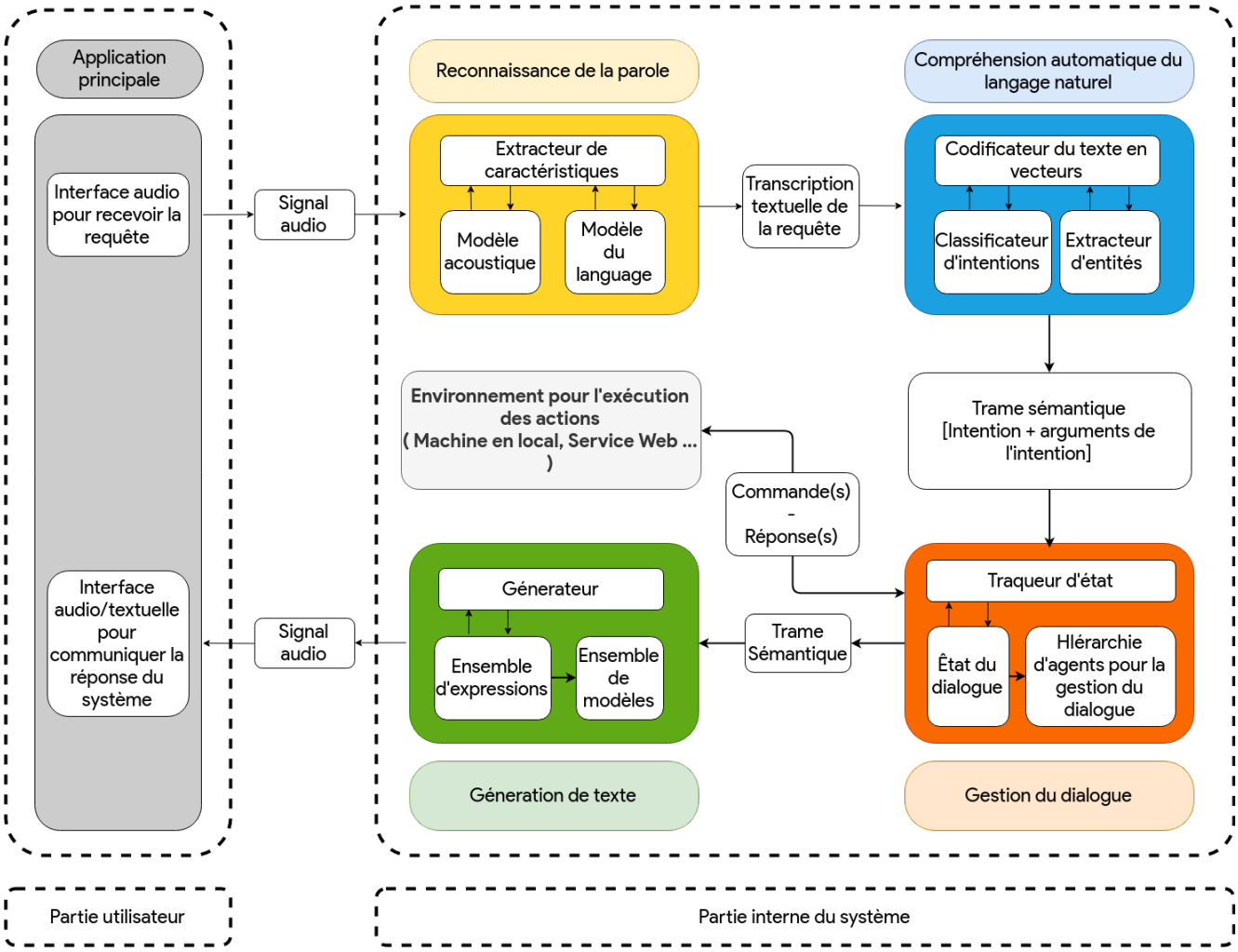

Figure 2.1 – *Architecture générale du système Bethano*

2.2.1 Partie utilisateur

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

2.2.2 Partie interne du système

Cette partie quant à elle représente ce que l'utilisateur ne voit pas et 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 1.2), la requête passe par un module de reconnaissance de la parole, qui traduira en texte le signal audio correspondant à cette dernière. Le module suivant, à savoir le module de compréhension du langage naturel, va extraire l'intention de l'utilisateur et ses arguments (par exemple *"open the home folder"* pourrait donner une intention dy 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 réside ou bien une API [2] qui aura accès à un service à distance (sur internet par exemple) ou local (dans un réseau domestique). Finalement, une action spéciale qui servira à informer l'utilisateur sera envoyée au module suivant (c.à.d le module de génération du langage naturel) pour être transformée en son équivalant dans un langage naturel, puis 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.

2.3 Module de reconnaissance automatique de la parole

Premier module du système Bethano, 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 [38]. 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 les tests.

2.3.1 Architecture du module ASR

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

2. Application Programming Interface ou interface de programmation applicative.

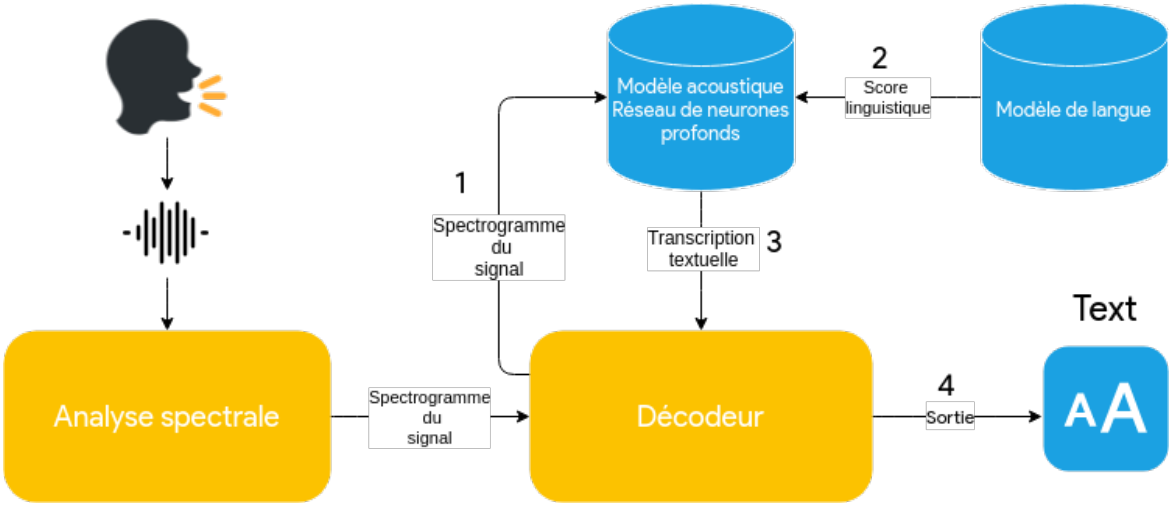

Figure 2.2 – *Architecture du module de reconnaissance de la parole (ASR)*

2.3.2 Modèle acoustique

Type du modèle

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

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

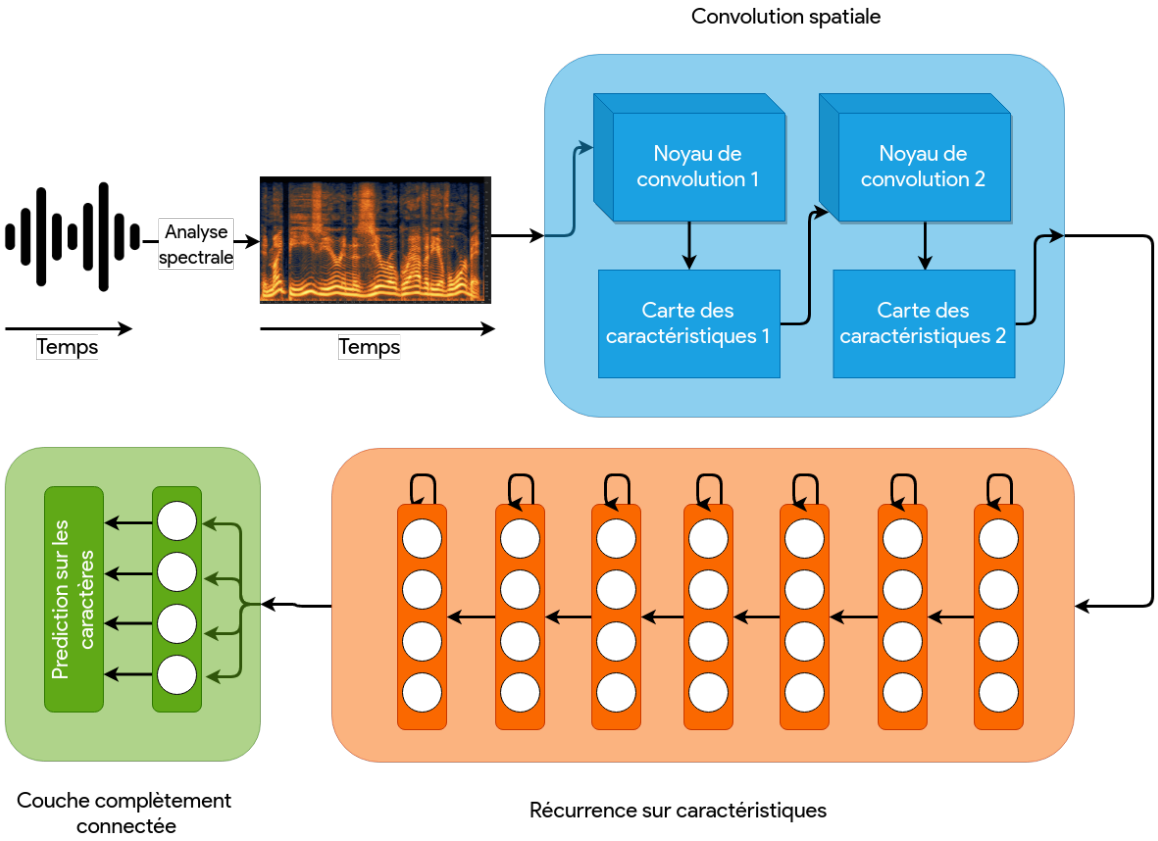

Figure 2.3 – *Architecture du modèle DeepSpeech [38]*

Données d'apprentissage

Pour entraîner le modèle acoustique, Mozilla a lancé le projet Common Voice[3], une plateforme en ligne pour récolter des échantillons audios avec leurs transcriptions textuelles. Chaque batch (lot) de données reçu est alors manuellement validé par l'équipe de Mozilla pour l'inclure dans la banque de données d'exemples principale. À ce jour, pour la langue anglaise, la plateforme a récolté plus de 22Go de données, soit 803 heures d'enregistrements correspondant à plus de 30 000 voix différentes dont 582 heures ont été validées. Cependant, ce volume de données est relativement petit comparé à celui déjà utilisé pour l'apprentissage initial. En effet, plusieurs sources ont été combinées pour construire cet ensemble de données. Dans [38] il a été mentionné que trois ensembles d'apprentissage existants ont étés choisis dont WSJ (Wall Stret Journal)[4], Switchboard[5] et Fisher[6], qui à eux trois cumulent 2380 heures d'enregistrements audios en anglais et plus de 27 000 voix différentes. Vient s'ajouter à cela l'ensemble Baidu[7] avec 5000 heures d'enregistrements et 9600 locuteurs.

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

2.3.3 Modèle de la langue

Type du modèle

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

Données d'apprentissage

À l'origine, DeepSpeech utilise un modèle de langue dont la source n'est pas dévoilée par les chercheurs dans [38], mais son volume est approximativement de 220 million de phrases avec 495 000 mots différents. Cependant, puisque ce corpus nous reste inconnu et qu'il a probablement été construit pour reconnaître des séquence de mots en anglais assez générales, nous avons décidé de construire notre propre modèle de langue en récoltant des données depuis des dépôts sur le site Github, plus précisément les fichiers README.md des dépôts qui font office de manuels d'utilisation d'un projet hébergé sur le site. Ce type de fichiers renferme généralement des instructions de manipulation de fichiers, de lancement de commandes, etc ; ce qui offre un bon corpus pour le modèle de langue. En effet notre système se concentre plus sur l'aspect de manipulation d'un ordinateur, donc la probabilité de trouver certaines séquences de mots qui appartiennent au domaine technique est en théorie plus élevée. La procédure suivi est la suivante :

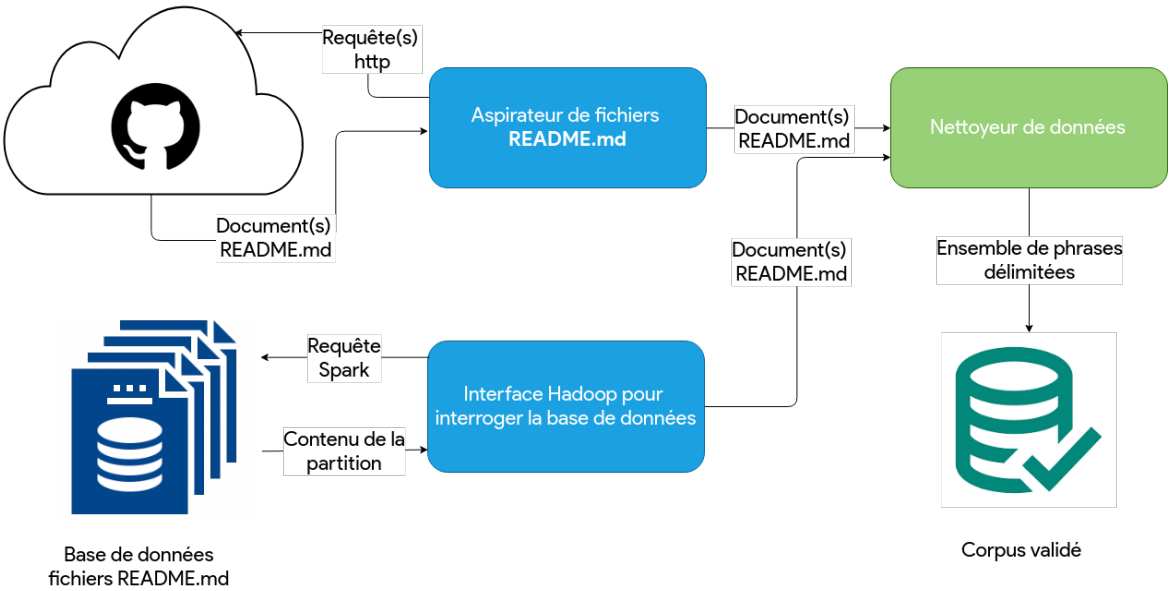

Figure 2.4 – *Processus de génération du corpus pour le modèle de langue*

- L'acquisition des données dans leur format brut **.md** (markdown) se fait de deux manières :

- Depuis le site officiel de GitHub en faisant des requêtes http au serveur en suivant le patron suivant pour les urls :

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

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

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

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

2.4 Module de compréhension automatique du langage naturel

Second module du système, le module de compréhension automatique du langage naturel (Natural Langage Understanding, NLU) à pour rôle de faire office de couche d'abstraction entre la requête de l'utilisateur (formulée dans 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 ([34],[53]) et cela malgré le petit volume des données d'apprentissage. Cette option nous a paru plus abordable que la construction d'un analyseur basé règles, souvent assez rigide et dont l'exhaustivité n'est pas évidente à obtenir.

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

2.4.1 Architecture du module

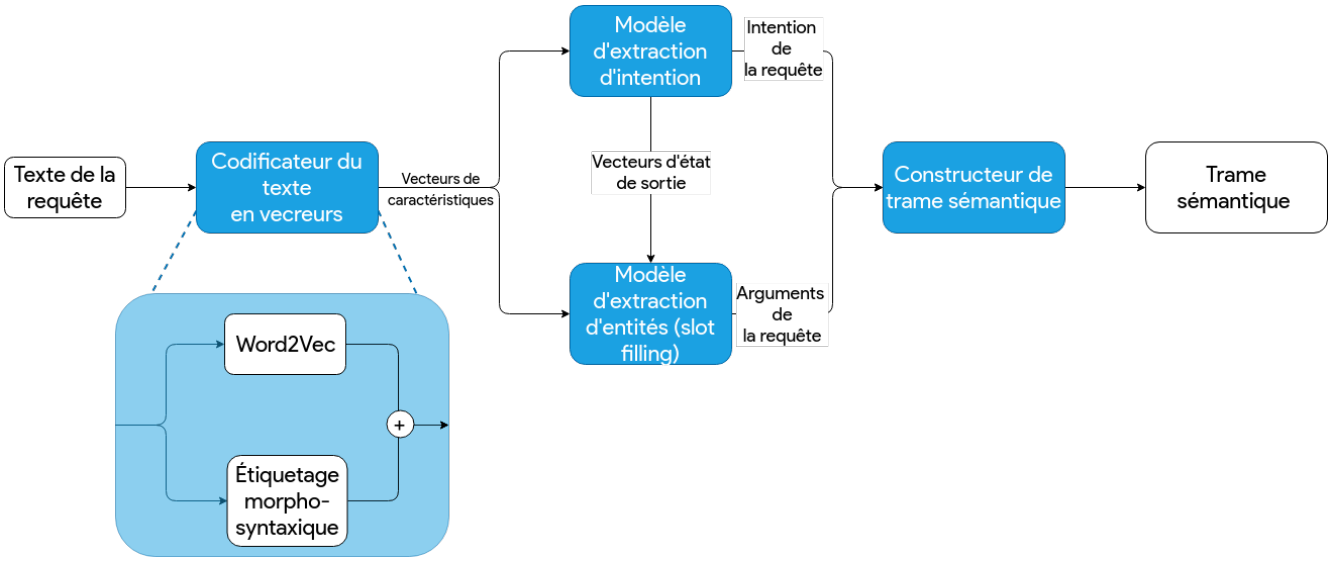

Figure 2.5 – *Architecture du module de compréhension automatique du langage naturel (NLU)*

Comme précédemment cité (voir la 2.2.2), le module NLU possède une architecture en pipeline qui reçoit en entrée le texte brut de la requête. Sa codification varie selon les approches que nous avons explorées et qui seront plus explicitées dans le chapitre suivant Réalisation et résultats. Pour mieux capturer l'aspect sémantique des mots dans le texte, nous avons décidé d'utiliser le modèle Word2Vec pré-entraîné par Google (entraîné sur 100 milliard de mots) pour produire un vecteur de taille fixe pour chaque mot. Pour encoder l'information syntaxique de la requête nous avons concaténé au vecteur de prolongement de chaque mot de la requête (Word Embedding Vector) le vecteur codifiant son étiquette morphosyntaxique. Après avoir codifié la séquence de mots, elle est envoyée aux modèles de classification d'intentions et d'extraction d'entités [9], qui sont en fait un seul modèle joint dont l'architecture est détaillée dans la section 2.4.2.1. Ces deux informations sont ensuite décodées et passées au constructeur de trame sémantique qui structurera ces dernières en une seule entité sémantique dans le format suivant :

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

```
{
  intent :"open_file_desire",
  entities :[
    {
      entity :"string",
      name :"file_name",
      value :"test.py",
      start :"31",
      end  :"37",
    }
  ]
}
```

Une trame sémantique se compose de deux parties :

- **Partie intentions** : l'intention extrait à partir de la requête. Elle se trouvera dans l'entrée "$intent$" de la trame.
- **Partie arguments** : aussi appelées entités du domaine, ces arguments sont extrait depuis le texte de la requêtes. L'entrée $entities$ contient une liste d'arguments. chaque élément de la liste renseigne sur le type, le nom, la valeur et l'emplacement d'une entité.

2.4.2 Analyse sémantique avec apprentissage automatique

2.4.2.1 Modèle(s) utilisé(s)

Comme vu dans le chapitre précédent (voir 1.5) 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 et 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 B-LSTM (Bidirectionall Long Short Term Memory) pour mieux capturer le contexte droit (respectivement gauche) de chaque entrée selon le sens de traitement des données dans le réseau B-LSTM. Le dernier vecteur en sortie est ensuite utilisé comme vecteur d'entrée pour un réseau de neurones Fully Connected (Complètement connecté) dont la dernière couche est une couche de prédiction sur une distribution de probabilités des intentions possibles.

Le décodeur est aussi un réseau de neurones récurent de type B-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) en utilisant un autre réseau de neurones Fully Connected sur chaque vecteur d'état de sortie des cellules LSTM du décodeur (voir la figure 1.8).

2.4.2.2 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 [14] 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 possibilité pour les requête est assez réduit et peut facilement être expliciter. En jouant sur l'ordre des mots nous avons pu générer pour les 15 intentions, 4157 patrons d'exemple 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 chosifiant 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 max (c.à.d les niveaux de répertoires et sous-répertoires) égale à 5. Nous avons aussi ajouté une liste des noms de fichiers et répertoires les plus populaires disponible dans [4]. 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 répertoire 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'],
  'tags': 'NUL NUL NUL NUL NUL ALTER.file_name ALTER.file_name',
  'text': 'please remove the file named platform notifications'
}
```

Une entrée est divisée en cinq champs :

- **id** : un entier qui sert d'identificateur pour l'instance.
- **intent** : l'intention (non encore codifiée) attribuée à l'instance.
- **tags** : les étiquettes de chaque mots de la requête, une étiquette peut être soit NUL (ce n'est pas un argument) ou bien le nom de l'entité (argument) que représente le mot à la position étiquetée. La liste complète des intentions avec leurs arguments se trouve dans le tableau 3.2 du chapitre Réalisation et résultats.
- **postags** : la liste des étiquettes morphosyntaxiques de chaque mots de la requête. L'ensemble des étiquettes utilisées est celui du Penn Treebank [57].
- **text** : le texte de la requête nettoyé et dont les mots sont séparés uniquement par un espace.

2.5 Module de gestion du dialogue

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

2.5.1 Architecture du module

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

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

2.5.1.1 État du dialogue

Avant de détailler les deux modules du gestionnaire. Il est nécessaire d'introduire une représentation de l'état du dialogue. Classiquement, les trames sémantiques ont été utilisées dans ce but (voir 1.6.2). Le suivi d'état se fait dans ce cas en gardant trace des emplacements remplis durant le dialogue. Nous avons opté pour l'utilisation d'une représentation plus riche, les graphes de connaissances, qui sont une forme de représentation où les connaissances sont décrites sous forme d'un graphe orienté étiqueté. Des travaux ont déjà utilisé des graphes de connaissances [75] et des ontologies [83] pour représenter l'état du système de dialogue. À partir de ce dernier une base de règles décide quelle action prendre directement du graphe. L'avantage d'utiliser des graphes de connaissances par rapport à l'utilisation des trames sémantiques apparaît dans leur flexibilité et leur dynamisme. En effet, pour une tâche comme la navigation dans les fichiers, l'état de l'arborescence des

fichiers est sujet à des changements fréquents : ajout, suppression, modification, etc. Il est difficile de faire une représentation de l'information dans ce cas avec de simples emplacements à remplir.

2.5.1.2 Suivi de l'état du dialogue

Le rôle du premier module est de mettre à jour l'état du système au cours du dialogue. Il reçoit l'action de l'utilisateur ou du gestionnaire et il produit un nouvel état. Dans notre cas, le module NLU produit toujours une trame sémantique contenant l'intention de l'utilisateur ainsi que ses paramètres. C'est alors le travail du traqueur d'état d'injecter le résultat du module NLU dans le graphe de connaissances. Ceci consiste à transformer la trame sémantique en un graphe qui est ensuite ajouté au graphe d'états. Plus de détails seront donnés dans la section 2.5.2 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.

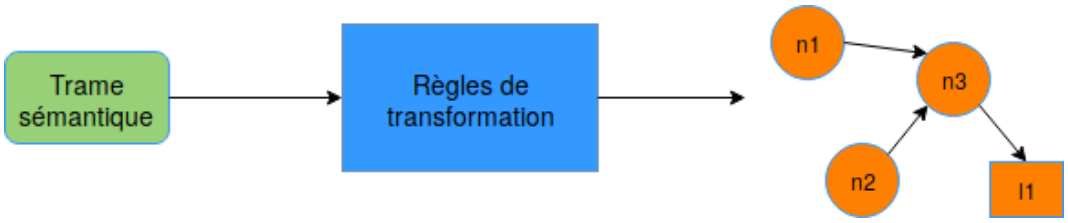

Figure 2.6 – *Schéma de transformation de trame sémantique en graphe*

2.5.1.3 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 coté, 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 coté, écrire les règles d'un système de dialogue ou d'un simulateur d'utilisateur s'avère compliqué et nécessite un travail manuel énorme pour gérer toutes les tâches possibles.

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

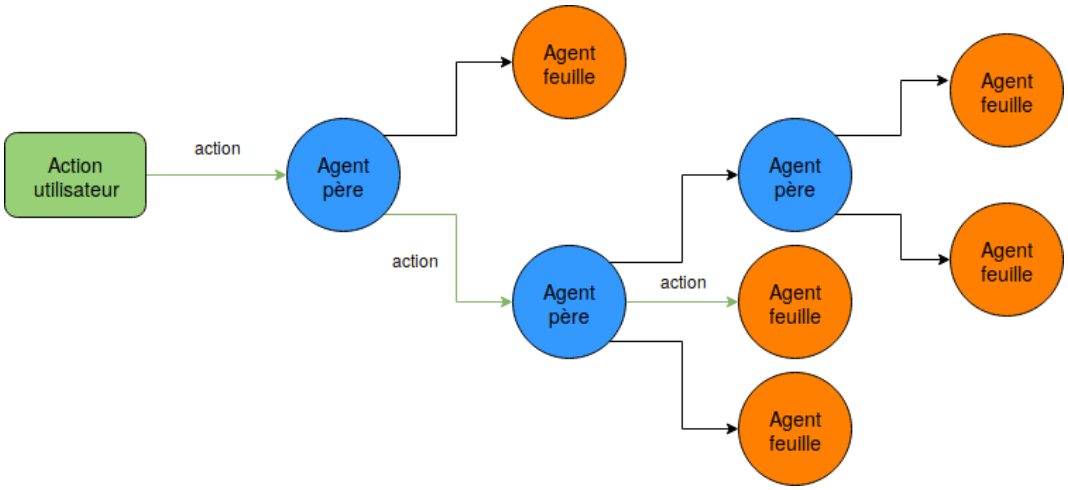

Figure 2.7 – *Schéma de l'architecture multi-agents pour la gestion du dialogue*

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 sont destinés pour une seule tâche sont considérablement plus abordables 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 transitions entre les simulateurs d'utilisateurs, ces derniers communiquent avec l'agent parent. Comme on connaît pour chaque simulateur l'agent fils qui lui correspond, il est donc possible de faire un apprentissage supervisé où les entrées sont les actions des simulateurs et l'état du système, tandis que la sortie est l'agent fils qui peut répondre à l'action.

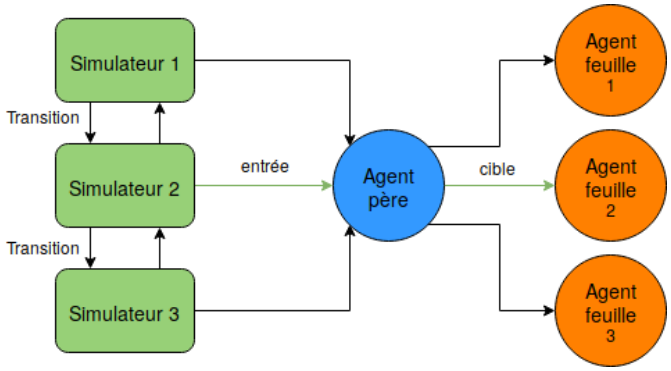

Figure 2.8 – *Schéma représentant l'apprentissage des agents parents avec les simulateurs des agents feuilles*

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

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

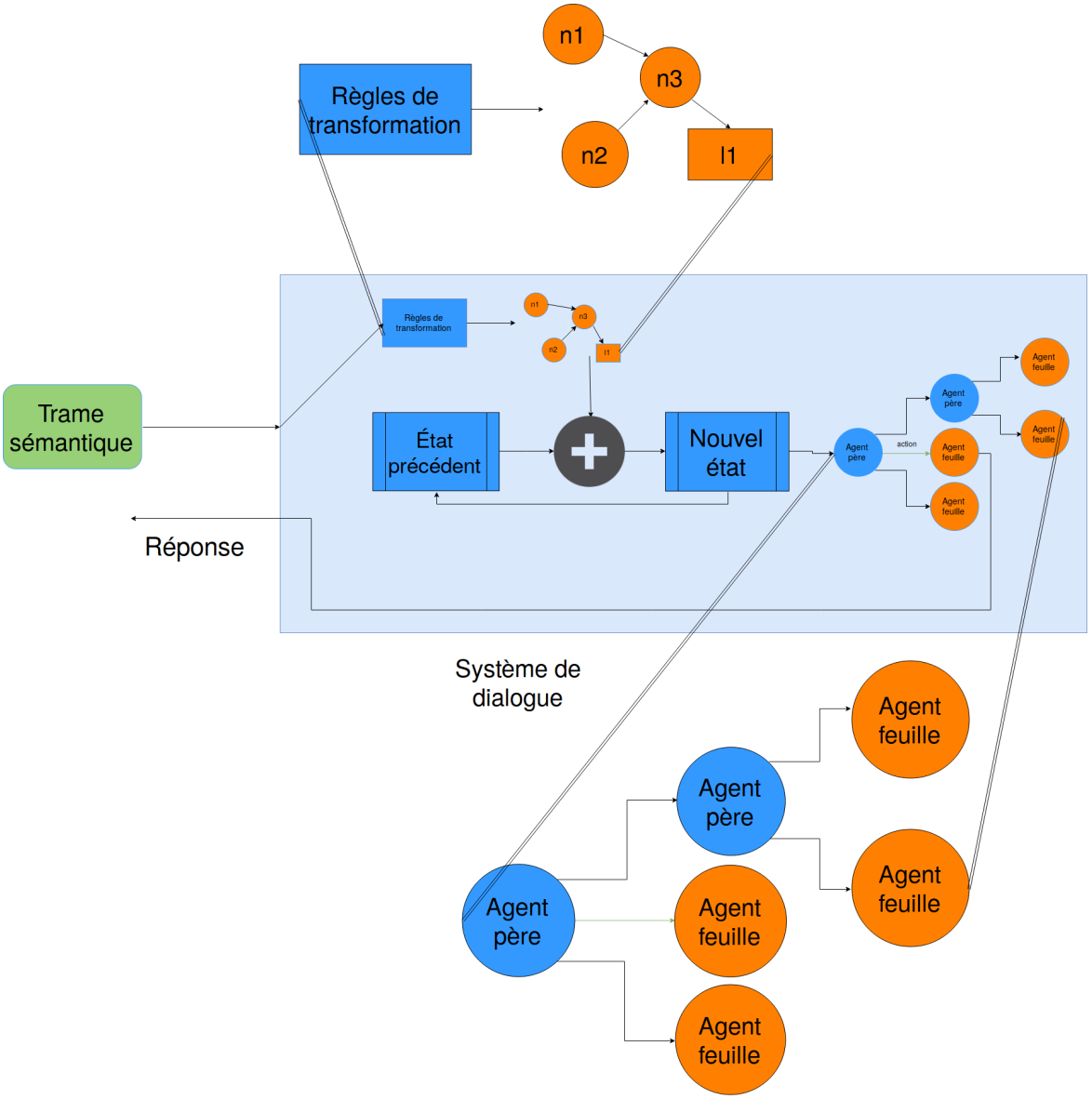

Figure 2.9 – *Schéma global du gestionnaire de dialogue*

2.5.2 Les ontologies du système

Une ontologie est une représentation des concepts et des relations d'un domaine donné. Elle définit un vocabulaire 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.

2.5.2.1 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.

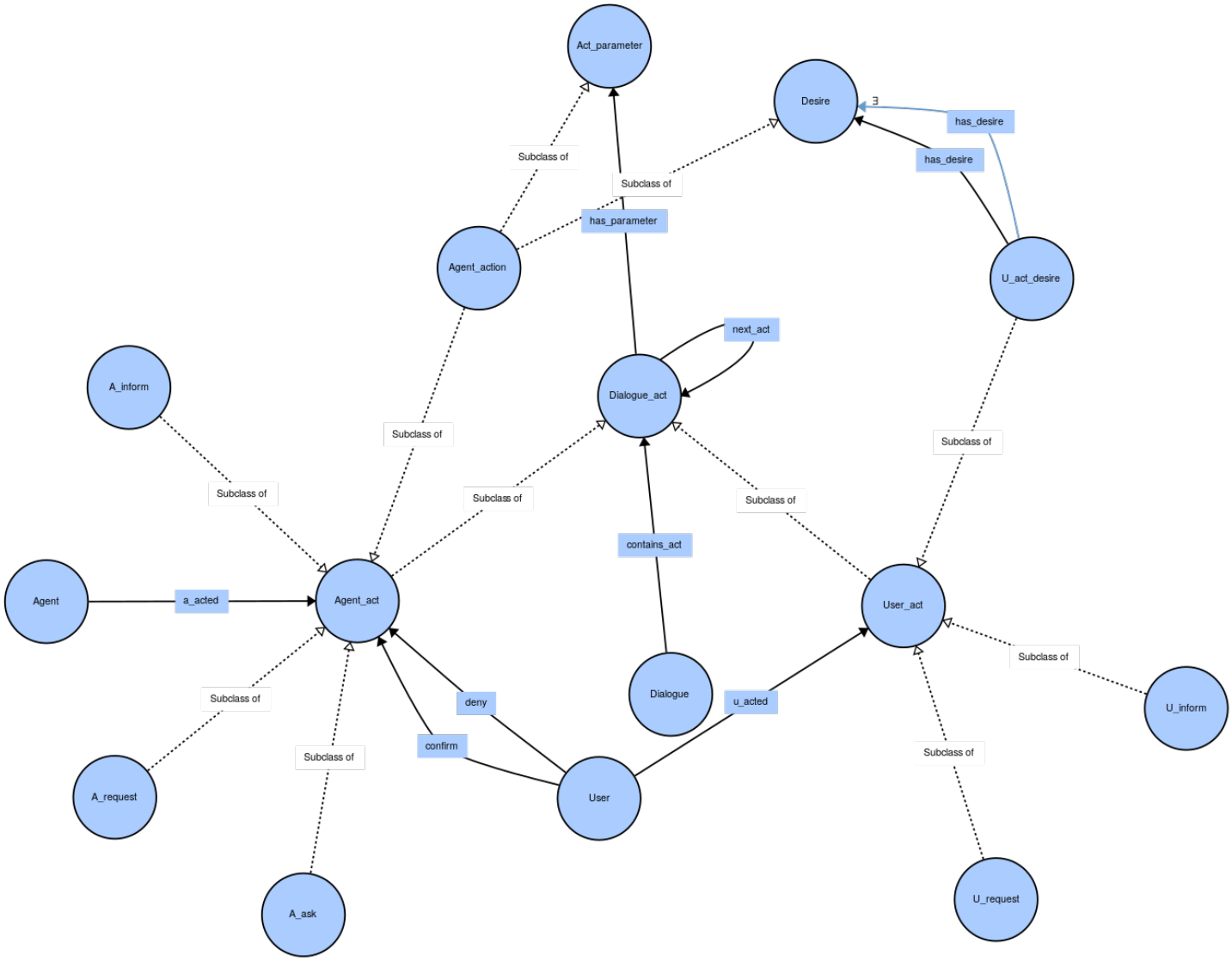

Figure 2.10 – *Graphe de l'ontologie de dialogue*

Principalement l'ontologie se compose de six classes mères :

- $Agent$ et $User$: ce sont les classes qui représentent l'utilisateur et l'agent qui participent au dialogue.
- $Dialogue$: l'agent et l'utilisateur participent à un dialogue ; ce dernier 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'arriver d'une nouvelle action, le traqueur d'état va créer le graphe correspondant. Un exemple abstrait de cela est représenté dans la figure 2.11. Une nouvelle action utilisateur est créée ainsi que ses paramètres et les relations entre eux.

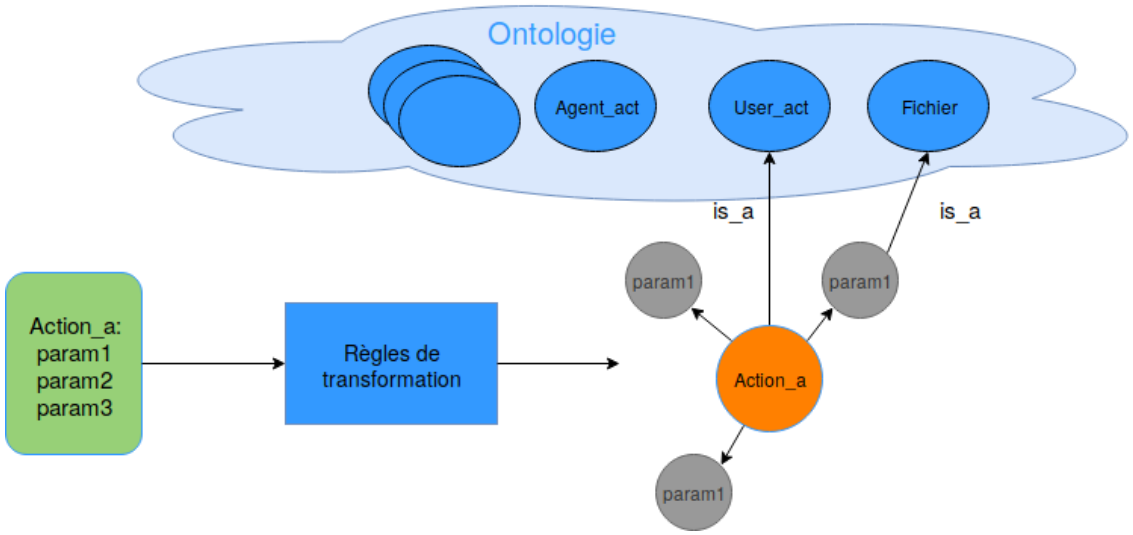

Figure 2.11 – *Schéma de transformation d'une action en graphe*

Ontologie pour l'exploration de fichiers

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

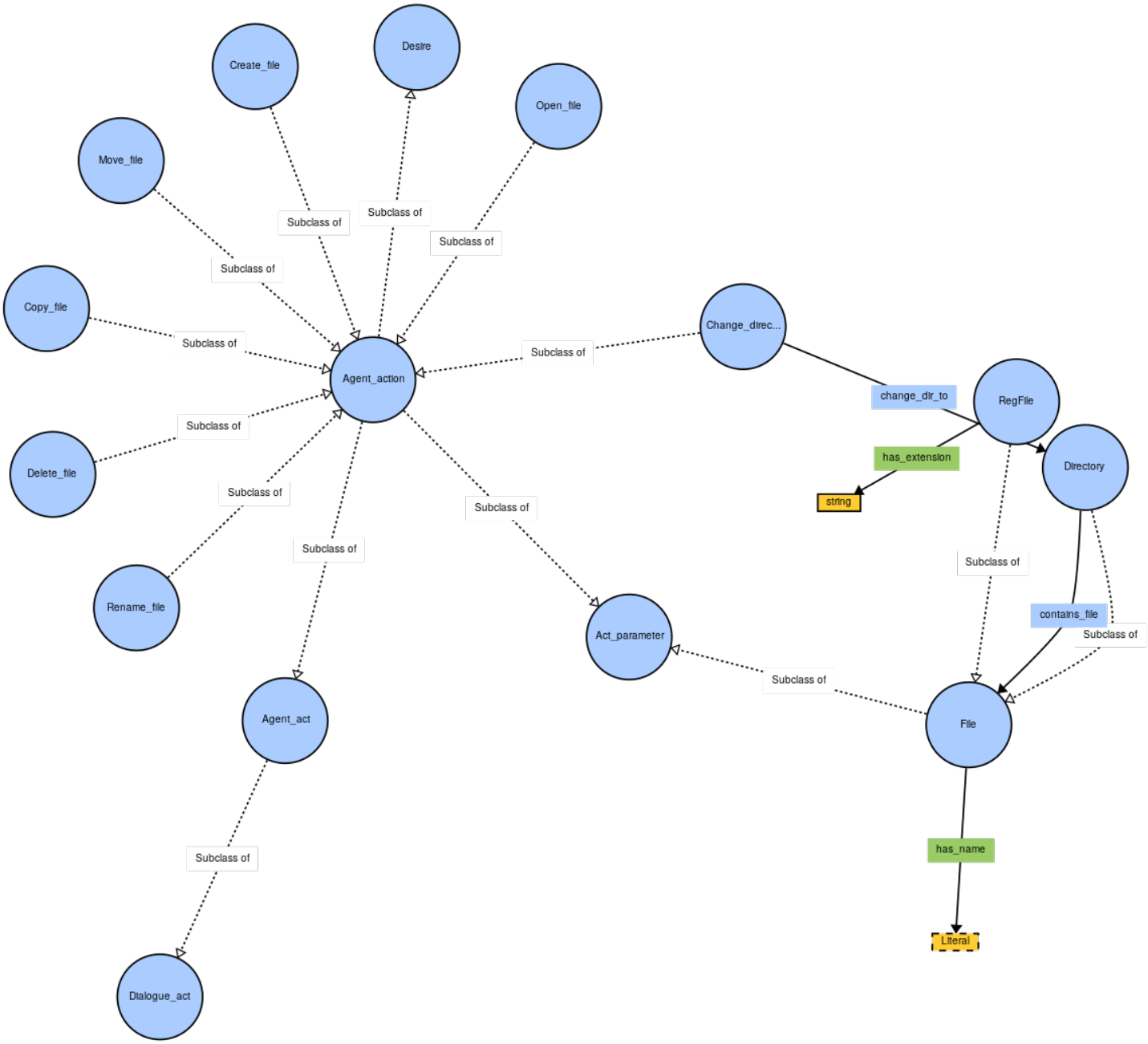

Figure 2.12 – *Graphe de l'ontologie de l'exploration de fichiers*

L'ontologie contient essentiellement :

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

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

La figure 2.13 représente l'arrivé 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.

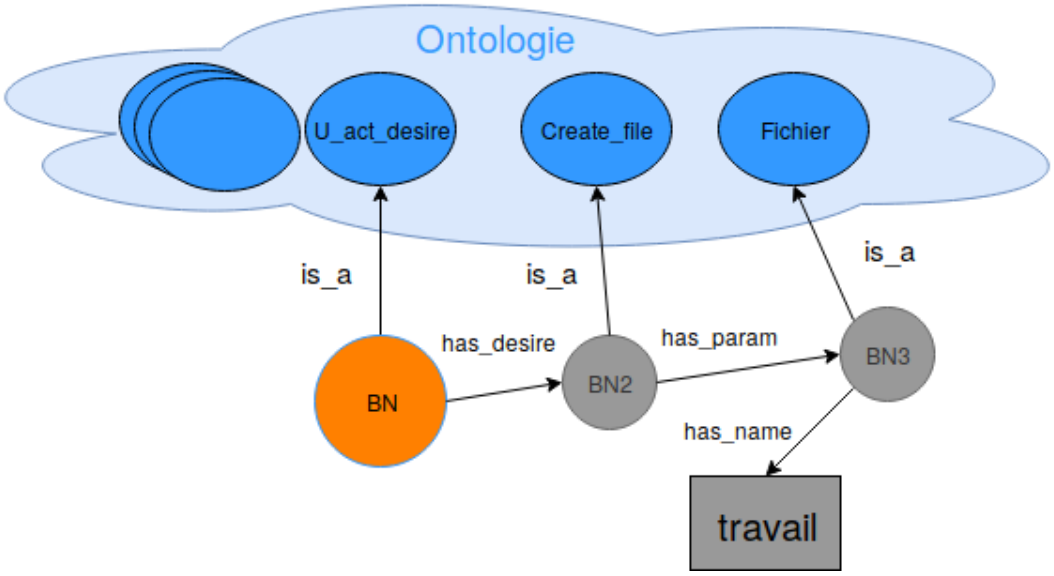

Figure 2.13 – *Schéma de transformation d'une action de demande de création de fichier en graphe*

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

2.5.3 Les simulateurs d'utilisateurs

Plusieurs méthodes peuvent être utilisées pour la création de simulateurs d'utilisateurs (voir **??**). Les simulateurs basés sur des méthodes d'apprentissage sont les plus robustes. Cependant, ils nécessitent un nombre important de données. L'alternative c'est d'utiliser des simulateurs basés règles. Nous nous sommes inspirés des simulateurs basés agenda [70] 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 est créé, qui contient les informations que doit convoyer le simulateur ainsi que les informations qu'il doit recevoir. 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 nombres d'actions change d'un état à un autre. Par exemple, le nombre de fichiers qu'on peut supprimer ou le nombre de répertoires auxquels on peut accéder ne sont pas fixes.

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

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

L'algorithme suivi par le simulateur est le suivant :

Algorithm 1 Algorithme simulateur

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

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

  en sortie de la procédure :

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

- **lignes 3-6 :** si le nombre de tours passés est supérieur au nombre maximal de tours autorisés, le simulateur retourne un état d'échec et une action indiquant la fin de la discussion à l'agent.
- **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 c'est le cas, le simulateur l'indique à l'agent avec une action de fin et un état de succès.
- **ligne 13 :** sinon, le simulateur décide quelle action prendre selon l'action de l'agent. Le diagramme 2.14 résume cette décision.

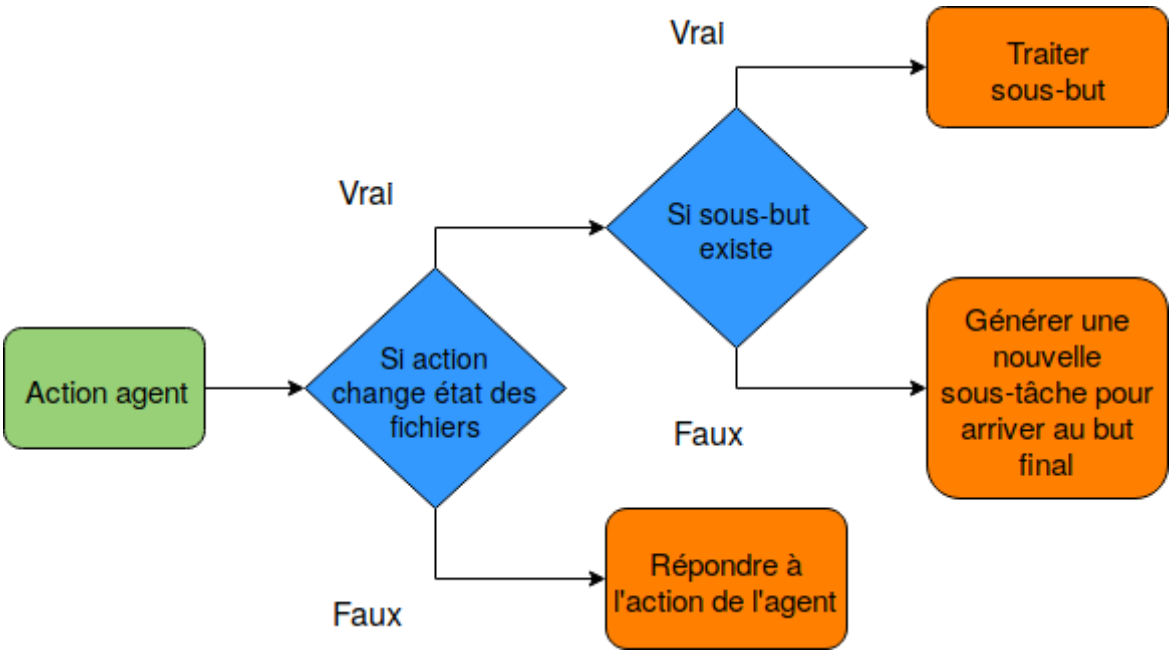

Figure 2.14 – *Diagramme de décision de l'action à prendre*

En ce qui concerne la fonction de récompense, celle-ci est évaluée à partir des changements effectués sur l'état de l'utilisateur. Le tableau 2.1 associe les changements avec leurs récompenses :

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

Table 2.1 – *Tableau des récompenses*

Il existe deux types d'actions :

- 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.

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

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

des sous-buts. L'agent ne fait donc qu'améliorer la similarité qu'on a vue précédemment en ajoutant ou en supprimant un fichier, c'est pourquoi les deux événements ont la même valeur de récompense.

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

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

2.5.4 Modèles d'apprentissage

Comme on l'a déjà cité dans le chapitre précédent, il existe plusieurs algorithmes d'apprentissage par renforcement comme Q-Learning ou State-Action- Reward-State-Action (SARSA) [5]. Cependant ces algorithmes, en essayant d'estimer la fonction Q de récompense, traitent le problème comme un tableau état/action et essayent d'estimer pour chaque état et action la récompense résultante. Ceci implique que ces algorithmes ne peuvent pas estimer la fonction de récompense pour des états qu'ils n'ont pas vus pendant l'apprentissage. Pour pallier à ce problème, Deep Q Learning (DQL)[59] utilise un réseau de neurones comme estimateur de la fonction Q, ce qui lui permet d'avoir une notion de similarité entre les états. Ainsi, il peut estimer la récompense pour des états jamais vus auparavant.

Encodeur de graphe

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

graphes avec Graph Convolution Networks (GCN)[45] qui s'avèrent être des variantes des Gated Graph Neural Networks (GGNN)[51]. Ces derniers utilisent des réseaux de neurones récurrents (RNNs) entre chaque deux nœuds reliés par un arc pour transférer l'information d'un nœud à un autre. C'est à dire que le RNN prend en entrée les vecteurs encodant un nœud et un de ses arc pour essayer de propager l'information contenue dans ces vecteurs au nœud destination. Ce dernier met à jour le vecteur l'encodant en sommant les résultats du RNN provenant des différents nœuds voisins. Ce qui résulte en des vecteurs ayant un encodage qui tient en compte le voisinage du noeud. Cette étape est répétée k fois pour que chaque nœud aie des informations des nœuds qui sont à un maximum de k pas de distance, avec k un paramètre empirique. Enfin, la somme des vecteurs d'états des différents nœuds est effectuée pour produire un vecteur fixe encodant tout le graphe qui peut être relié au reste du réseau de neurones pour l'apprentissage.

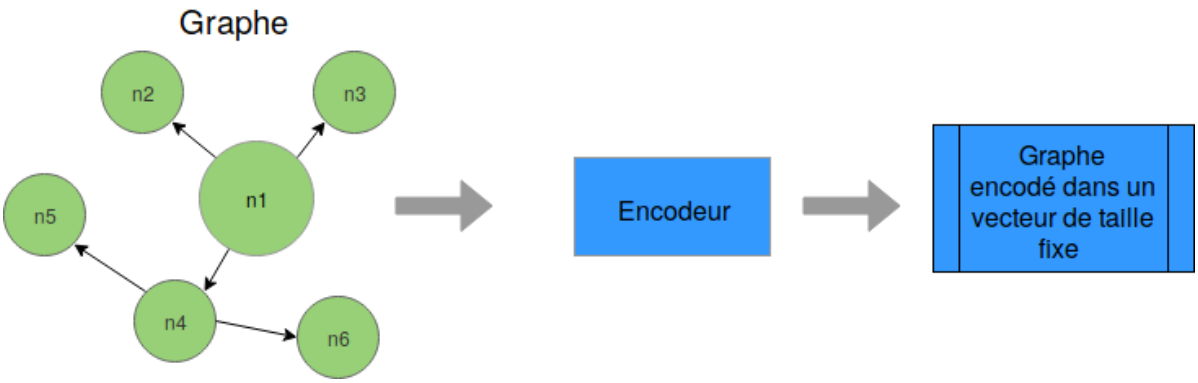

Figure 2.15 – *Schéma représentant un encodeur de graphe*

Ces méthodes nécessitent tout le graphe pour l'encoder. Cependant, dans notre cas, après chaque action, le graphe augmente de taille ce qui nécessite de refaire l'encodage à partir du début. Nous proposons de traiter le graphe comme une séquence de triplets : "nœud ; arc ; nœud" ou "sujet ; predicat ; objet" comme dans la représentation Resource Description Framework (RDF). 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.

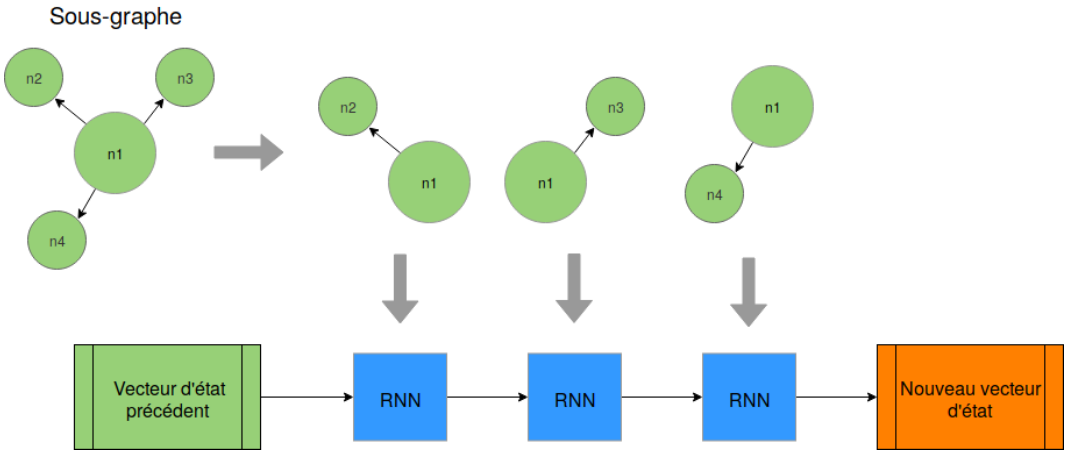

Figure 2.16 – *Schéma représentant un encodeur séquentiel de graphe*

Comme le montre la figure 2.5.4, lorsque un sous-graphe arrive, celui-ci est décomposé en triplets. Ensuite, chaque triplet est encodé séquentiellement dans le vecteur d'état précédent qui encode déjà l'ancien état du graphe. Le résultat sera un vecteur d'état encodant la totalité du nouveau graphe.

Pour faire l'apprentissage de cette architecture, il est possible de générer des graphes aléatoirement qu'on fait passer triplet par triplet dans 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.

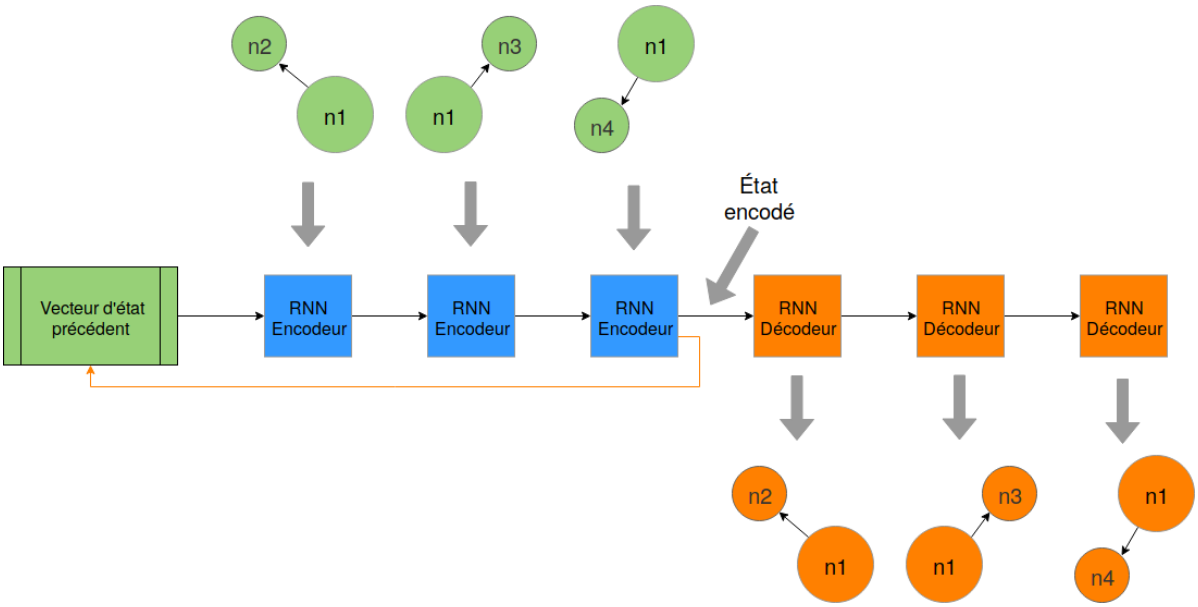

Figure 2.17 – *Schéma de l'apprentissage d'un encodeur séquentiel de graphe*

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 pour but d'estimer la correspondance des récompenses en fonction des états. Nous utilisons pour cela un réseau de neurones profond qui prend en entrée le graphe encodé de l'agent et produit pour chaque action la récompense correspondante. Comme le nombre d'actions est variable, il est impossible d'utiliser un réseau de neurones qui produit une récompense pour chaque action vu que les réseaux de neurones ont un nombre de sorties fixe. Par conséquent, il est nécessaire d'utiliser une architecture qui prend en entrée, en plus de l'état encodé, l'action candidate pour produire en sortie sa récompense.

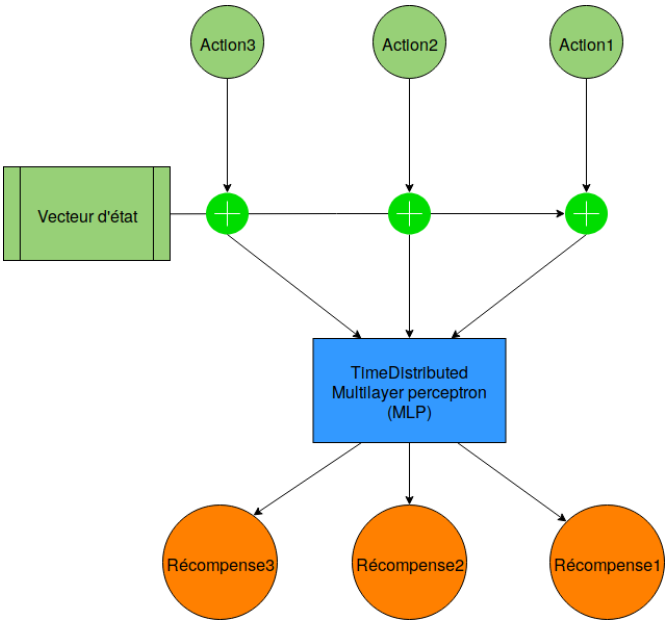

Figure 2.18 – *Schéma du réseau DQN*

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

- Rejouer l'expérience : l'agent interagit avec le simulateur plusieurs fois en gardant dans une mémoire ses interactions. Après chaque k épisodes[10] l'agent reprend la mémoire pour entraîner le réseau de neurones.
- Double DQN : la fonction Q est donnée par la formule $Q(s,a) = r + \alpha \times max_j(Q(s', a_j))$ [59] 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 l'évaluation de la récompense dans ce terme améliore les résultats [59]. La formule devient donc :

  $$Q(s,a) = r + \alpha \times Q(s', argmax_{a_j}(s', a_j))$$

  Cette valeur est donc utilisée comme cible du réseau DQN et permet de calculer l'erreur moyenne quadratique par rapport à sa prédiction. Ensuite l'algorithme de retro-propagation est appliqué pour l'apprentissage automatique.

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

Une autre architecture possible serait de relier l'encodeur de graphe directement avec le réseau DQN pendant l'apprentissage. Ainsi l'erreur de l'apprentissage pour l'encodeur est calculée à partir de la fonction de récompenses 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 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.

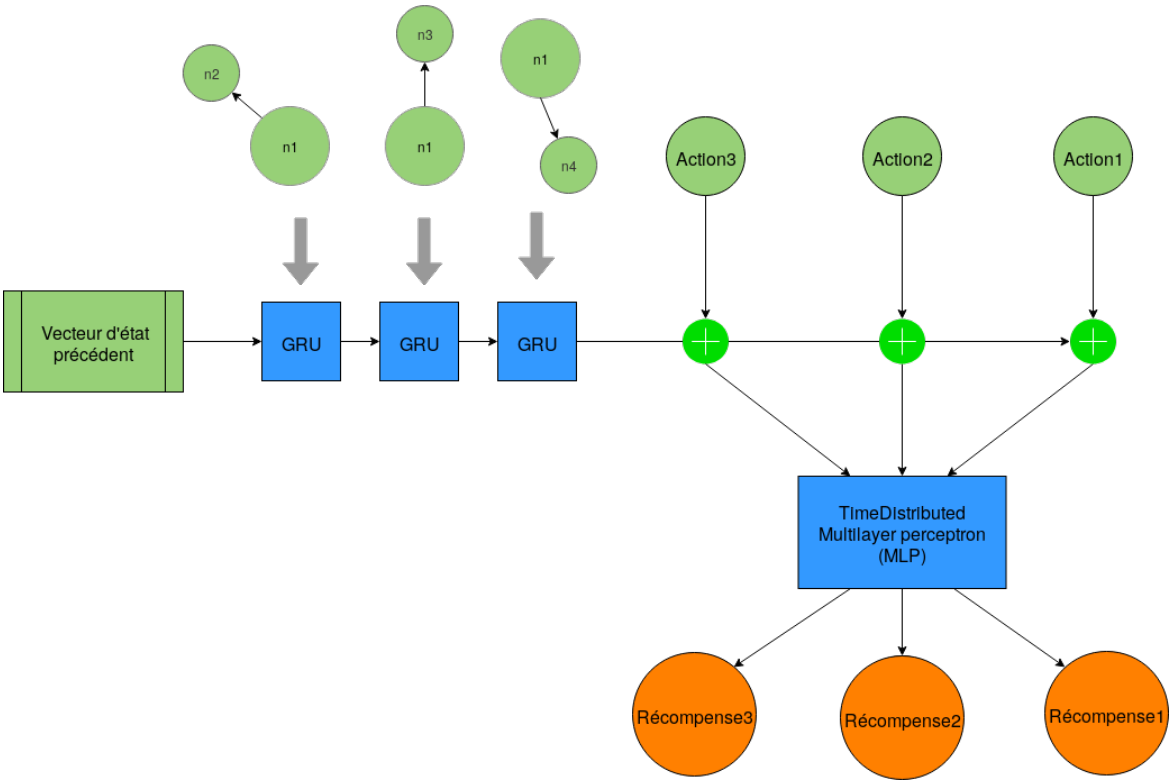

Figure 2.19 – *Schéma du réseau DQN relié avec l'encodeur directement*

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.

2.6 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 texte 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 texte utilise le nom de l'erreur comme l'intention de l'agent et ses paramètres comme les paramètres de l'action.

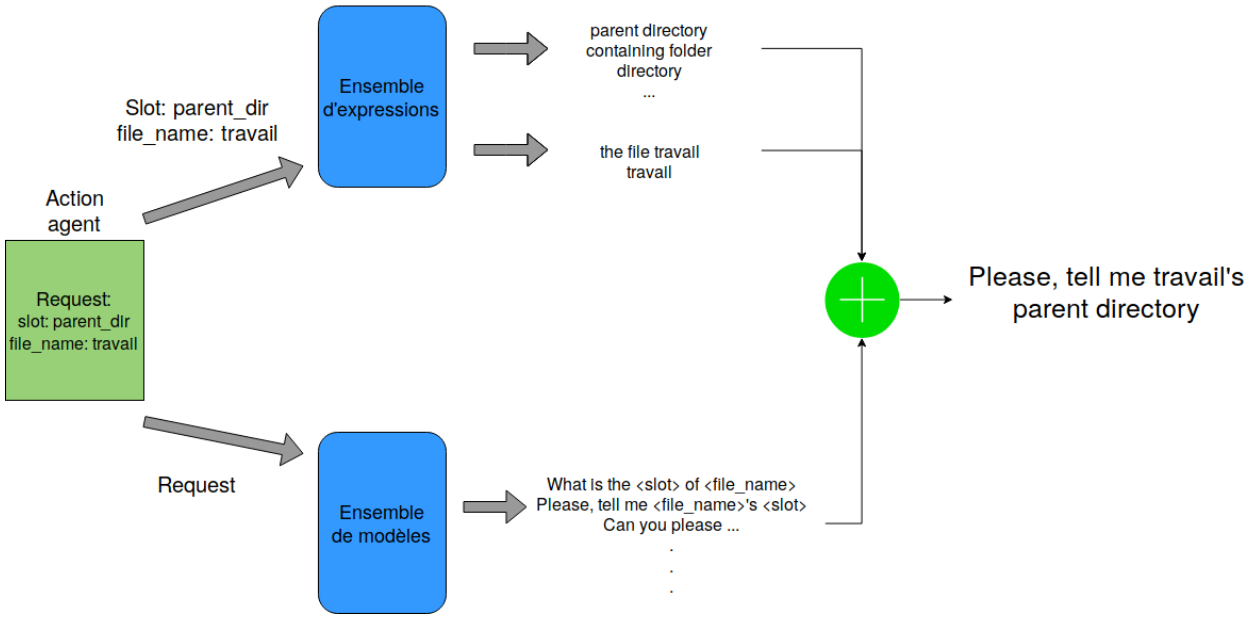

Figure 2.20 – *Schéma de fonctionnement du générateur de texte*

Dans la figure 2.20 un exemple de génération de texte 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 texte 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 choisi 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***".

2.7 Conclusion

Dans ce chapitre, et en nous inspirant des travaux existants, nous avons pu modéliser et conceptualiser tout les aspects du système que nous jugeons assez conforme au standard. Viennent s'ajouter les améliorations proposées pour enrichir le système de base retrouvé

dans la littérature ainsi que pour faciliter l'extensibilité des fonctionnalités de bases avec un moindre effort d'intégration. Ceci est le résultat d'une conception modulaire et facilement maintenable.

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

Chapitre 3

Réalisation et résultats

3.1 Introduction

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

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

3.2 Environnement de développement

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

3.2.1 Machines utilisées

Principalement, le développement se divise en deux parties :

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

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

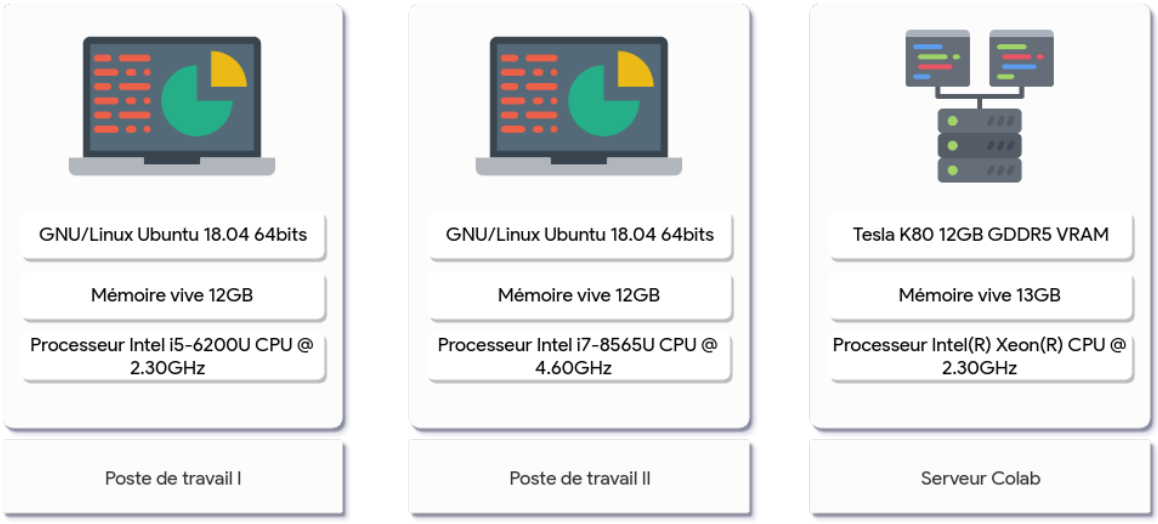

Figure 3.1 – *Caractéristiques des machines*

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 :

3.2.2 Langages de programmation

Python

C'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 reliés au développement d'applications de bureau, web ou mobiles [3].

Javascript

JavaScript 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 coté client mais peut aussi être exécuté du coté 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*[1] devenu un standard [2].

1. Node Package Manager ou Gestionnaire de packages Node

3.2.3 Librairies et bibliothèques

Figure 3.2 – *Bibliothèques et librairies les plus utilisées dans ce projet.*

DeepSpeech API

Package python qui fait office d'interface entre un script Python et la librairie de reconnaissance automatique de la parole DeepSpeech [38]. 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 [1].

PyAudio

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 ...). La possibilité d'extraire le vecteur de caractéristiques d'un extrait audio est aussi présente comme fonctionnalité [**?**].

Beautiful Soup

Bibliothèque Open source permettant l'analyse de fichiers html pour en extraire ou y injecter des données. Principalement utilisée pour filtrer les balises html depuis une page web [**?**].

PySpark

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

Scikit-Learn

Bibliothèque Open source conçue pour rapidement développer des modèles pour l'apprentissage automatique, principalement utilisée pour ses nombreux outils de pré-traitement des données (codification, normalisation, filtrage ...) [**?**].

Numpy

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

Tensorflow & Keras

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

Flask

Micro-librairie Open source dédiée au développement d'applications basées web. De base, cette librairie est très légère, mais elle offre la possibilité d'ajouter des extensions qui s'intègrent très facilement au système de base [**?**].

Vuetify

Librairie Open source basée sur VueJs dédié au développement d'interfaces web ou mobiles. Elle implémente le paradigme Material Design de Google et offre la possibilité d'étendre les composants de base et de créer des interfaces belles et adaptatives [**?**].

3.2.4 Outils et logiciels de développement

PyCharm

PyCharm est un environnement de développement intégré spécialisé et optimisé pour programmer dans le langage Python (voir 3.2.2 ci dessus). Il permet l'analyse de code en continu et offre un débogueur intégré pour exécuter un code instruction par instruction. Il offre également l'intégration de logiciel de gestion de versions comme Git [**?**], et supporte le développement web avec Django et Flask [**?**].

Git

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

Google Colaboratory

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

Protégé

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

3.3 Reconnaissance automatique de la parole

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

3.3.1 Ensemble de test

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

- une requête prélevée de l'ensemble de test du module de compréhension du langage naturel (voir les sections 3.4.2 et 2.4.2.2), ou
- une question prélevée de l'ensemble de données AskUbuntu [2] qui regroupe des questions relatives à la manipulation d'un ordinateur sous le système d'exploitation GNU/-Linux.

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

3.3.2 Méthodologie d'évaluation

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

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

   Un script de conversion est ensuite lancé pour s'assurer que chaque enregistrement est au format .wav avec une fréquence de rafraîchissement égale à 16KHz (le modèle acoustique de DeepSpeech attend cette valeur exacte pour lancer l'inférence, sinon une erreur se produira).
2. **Métriques retenues** : À chaque instance testée, le **WER** (Word Error Rate) est calculé. Pour rappel la formule du WER est la suivante :

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

   où :

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

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

3. **Boucle d'évaluation** l'opération précédente est réitérée en incrémentant à chaque fois le taux d'utilisation de notre modèle de langue (de 20% à 100% avec un pas de 10%). Le WER associé est ensuite comparé à celui obtenu en utilisant le modèle de langue par défaut que propose DeepSpeech, le modèle acoustique sans modèle de langue, le résultat de l'API de Google [3] et enfin le modèle par défaut de CMU Sphinx [4]

3.3.3 Résultats

Les résultats sont décrits dans le tableau 3.1 et la figure 3.3. Nous remarquons que sur notre mini-ensemble de test, les modèles par défaut obtiennent un score très proche de 1, ce qui démontre qu'ils ont été entraînés sur des cas assez généraux, ce qui n'est pas très bon. Cependant, en changeant juste le modèle de langue par défaut en injectant des échantillons, une nette amélioration est visible (environ -20%). Ce taux d'erreur diminue en augmentant la taille du corpus utilisé pour le modèle de langue. Toutefois, après avoir pris plus de 75% du corpus, l'erreur a légèrement augmenté. Cela peut s'expliquer par la nature assez bruitée du corpus, les fichiers README.MD qui sont rédigés par des personnes, l'erreur humaine, et l'absence de processus de vérification de l'orthographe, de la grammaire ou de la syntaxe du contenu. Nous avions envisagé de choisir comme corpus des extraits de livres dédiés à la manipulation des ordinateurs sous Linux, mais nous n'avons pas trouvé de ressources gratuites ou Open source exploitables.

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 2.3, DeepSpeech reste la meilleure option en Open Source à notre portée.

3. https://cloud.google.com/speech-to-text/
4. https://cmusphinx.github.io/

	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 - 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 3.1 – *Tableau récapitulatif des résultats pour le module de reconnaissance automatique de la parole.*

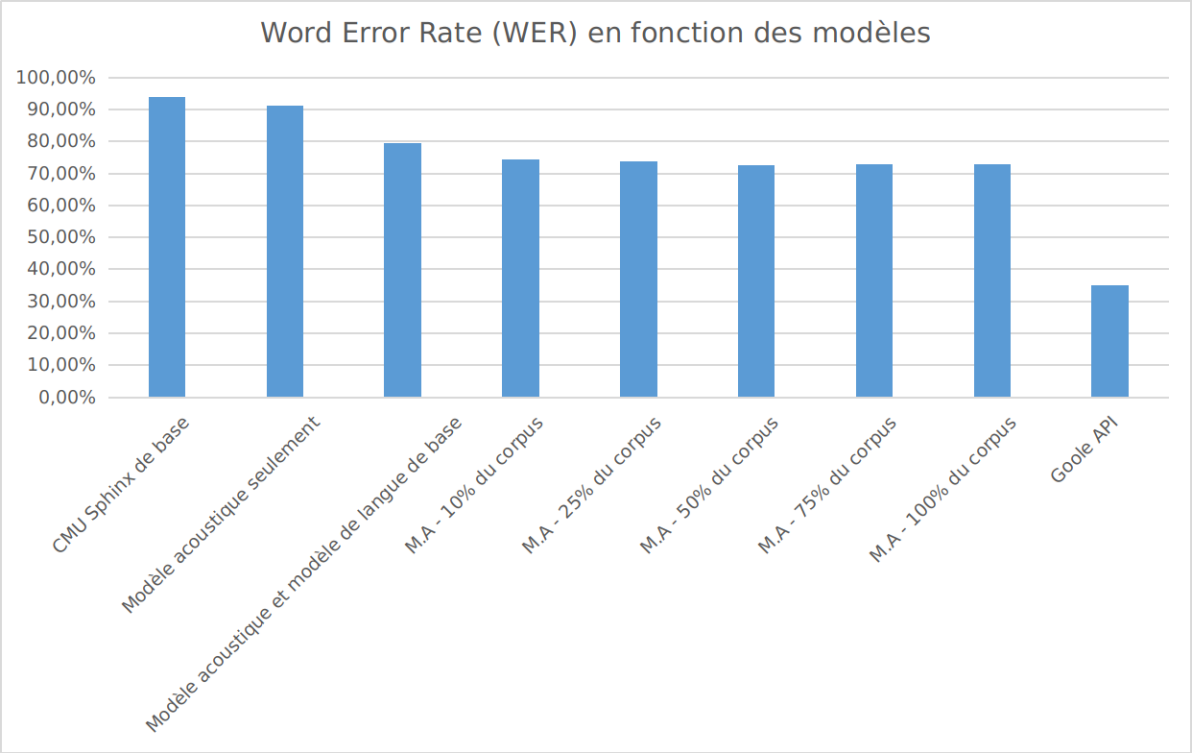

Figure 3.3 – *Graphe récapitulatif des résultats pour le a reconnaissance automatique de la parole*

3.4 Classification d'intentions et extractions 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 mé-

triques d'évaluation utilisées sont mentionnées dans la sous section 3.4.2. 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.

3.4.1 Ensemble de test

Comme mentionné dans le chapitre précédent (voir la section 2.4.2.2), nous avons nous mêmes construit un ensemble d'apprentissage relativement varié. Il regroupe essentiellement des commandes, ou requêtes liées à l'exploration de fichiers pour le moment car c'est la tâche rudimentaire que Bethano 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 **??**, 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 3.2 ci dessous.

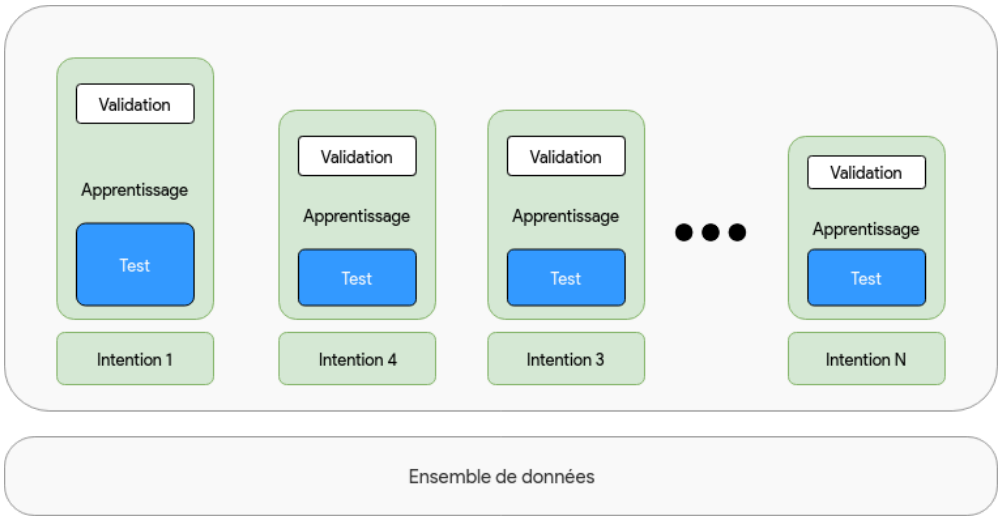

Figure 3.4 – *Schéma de découpage des données pour l'apprentissage du modèle de compréhension du langage naturel*

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

Table 3.2 – *Tableau récapitulatif de toutes les intentions avec leurs descriptions et leurs arguments.*

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

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

3.4.2 Méthodologie d'évaluation

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

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

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

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

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

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

  Les poids des classes sont calculés selon la formule suivante :

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

  où :

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

  La formule de la fonction erreur devient donc :

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

  où :

  - $\hat{y}$ est le vecteur en sortie produit par le modèle à la suite d'une fonction $Softmax$.
  - y est le vecteur de classe réelle présent dans l'ensemble d'apprentissage
  - C est le nombre de classes au total.
- **Fonction d'apprentissage** : Le choix de la fonction d'apprentissage est généralement affecté par un désir de précision et de rapidité. Une fonction qui converge rapidement en un minimum local peut être parfois préférée à une autre qui prendrait un temps considérable pour soit se retrouver dans le même minimum ou un autre minimum local (donc sans garantie de minimum optimal de la fonction erreur). Les deux fonctions utilisées sont RmsProp et Adam qui sont connues pour leur rapidité de convergence.
- **Encodage des entrée-sorties** : Là encore le choix de l'encodage des données influe grandement sur la capacité du modèle à distinguer et à représenter les différentes informations qui lui sont présentées. Comme initiative de notre part, nous avons mentionné dans la section 2.4.2.2 l'ajout de l'étiquette morphosyntaxique de chaque mot à l'encodage. Nous avons donc lancé les tests sur un encodage avec (respectivement sans) l'ajout des étiquettes pour mieux constater son impact.
- **Découpage des données** La stratégie adoptée était de prendre aléatoirement les même proportions pour chaque sous-ensemble de chaque classe (intentions ou entités de domaine). Nous avons aussi décidé de faire varier les proportions de test et d'apprentissage en fixant celui de validation car nous avons remarqué que notre modèle n'arrivait pas à bien généraliser la relation entre les entrées et les sorties. Notre intuition portait sur le fait que le manque de données pouvait en être la cause (voir la section 2.4.2.2 pour plus de détails). Ainsi, nous avons varié le taux de découpage

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

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

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

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

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

où :

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

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

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

où :

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

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

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

3.4.3 Résultats

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

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

Table 3.3 – *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 3.3 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 3.4 – *Résultats des tests pour un encodage sans étiquetage morphosyntaxique avec des cellules BiLSTM avec découpage Apprentissage : 25%, Validation : 10%, Test : 75%.*

Dans le tableau 3.4, l'ajout de l'information du contexte pour un mot à une position donnée à travers l'utilisation d'une architecture BiLSTM affecte systématiquement les scores (Précision, Rappel et F-Mesure). Ceux-ci augmentent d'un certain taux ($\approx 10\%$) dans la majorité des cas. Il est a noter que pour le même nombre d'unités BiLSTM, le nombre de neurones pour la classification d'intentions a doublé.

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

Table 3.5 – *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 3.6 – *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 3.5 et 3.6, ils montrent que l'ajout de l'étiquette morphosyntaxique a amélioré les résultats pour cas de l'utilisation d'une cellule LSTM simple tout en réduisant le nombre d'unités requises, moins de calculs et plus d'efficacité. Cela se confirme encore plus pour le cas de l'utilisation de l'architecture BiLSTM, en réduisant de presque la moitié la puissance de calcul nécessaire et en augmentant un tout petit peu la qualité des prédictions.

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

Table 3.7 – *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 3.8 – *Résultats des tests pour un encodage sans étiquetage morphosyntaxique avec des cellules BiLSTM avec découpage Apprentissage : 50%, Validation : 10%, Test : 50%.*

D’après les résultats des tableaux 3.7 et 3.8, le choix de l’architecture BiLSTM à amélioré la qualité des classifications, même si ce n’est que d’un petit taux. Ces tableaux semblent aussi montrer que notre intuition sur la faible quantité de données que nous possédions soit fondée. Les scores de F-Mesure tendent en moyenne à augmenter avec l’injection de nouvelles données d’apprentissage.

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

Table 3.9 – *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 3.10 – *Résultats des tests pour un encodage avec étiquetage morphosyntaxique avec des cellules BiLSTM avec découpage Apprentissage : 50%, Validation : 10%, Test : 50%.*

Pour les tableaux 3.9 et 3.10, 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 3.11 – *Résultats des tests pour un encodage sans étiquetage morphosyntaxique avec des cellules LSTM avec découpage Apprentissage : 75%, Validation : 10%, Test : 25%.*

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

Table 3.12 – *Résultats des tests pour un encodage sans étiquetage morphosyntaxique avec des cellules BiLSTM avec découpage Apprentissage : 75%, Validation : 10%, Test : 25%.*

Les tableaux 3.11 et 3.12 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 différent modèles, la figure 3.5 ci dessous est proposée.

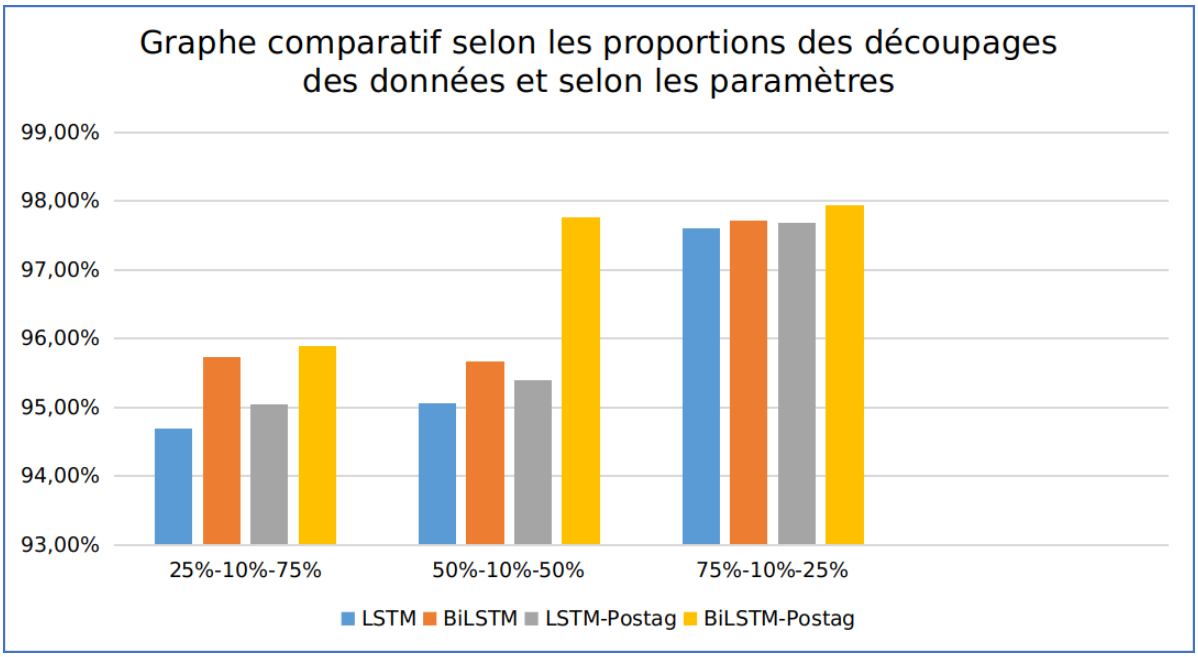

Figure 3.5 – *Graphe comparatif selon les proportions des découpages des données et selon les paramètres.*

Nous pouvons remarquer l'impact de trois facteurs :

- **La taille de l'ensemble d'apprentissage** : Systématiquement, le score F-Mesure moyen augmente avec la croissance du volume de données d'apprentissage. Cela laisse présager qu'avec plus de données le modèle pourrait mieux généraliser, ce qui éviterait de biaiser le modèle vers un type de données en particulier.
- **L'utilisation du contexte (LSTM contre BiLSTM)** : Le modèle BiLSTM donne de meilleur résultat lorsque l'information du contexte lui est accessible, ce qui confirme notre intuition.
- **L'utilisation de l'étiquetage morphosyntaxique** : L'ajout de l'information sur la syntaxe de la requête semble aider le modèle à construire une meilleure représentation interne de la requête. Il ne voit pas un mot seulement à un instant t, mais plutôt la paire (mot,rôle syntaxique du mot dans la phrase). Cet ajout est conforme à notre explication théorique dans la section 2.4.1

3.5 Ontologie et manipulation du graphe d'état

Les ontologies déjà présentées dans la section 2.5.2.1 ont été créées en utilisant Protégé. Elles ont été ensuite exportées en format Turtle [5] afin de les exploiter dans la suite de ce travail en utilisant la bibliothèque RDFLib de Python.

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

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

tion avec les réseaux de neurones. Une phase de transformation des URIs [6] en identifiants entiers s'avère nécessaire. L'ontologie comprend 61 nœuds et 13 relations. Contrairement au nombre de relations, le nombre de concepts augmente au cours du dialogue ; de nouveaux nœuds sont introduits après chaque échange. Ceci nécessite de garder un ensemble d'identifiants non utilisés pour les associer pendant le dialogue. Le tableau suivant résume l'association des identifiants aux nœuds et relations des graphes.

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

Table 3.13 – *Tableau des identifiants*

3.6 L'agent de dialogue

Nous avons proposé dans la partie conception 2.5.4 deux architectures pour entraîner l'agent de dialogue. La première se compose de deux parties, un module pour encoder le graphe d'état et un autre pour décider l'action à prendre. Chacun est entraîné séparément. Les deux modules étant des réseaux de neurones, nous avons pensé à une deuxième architecture en les connectant pendant la phase d'apprentissage. Cette connexion permet à l'encodeur de graphe de choisir les parties du dialogue, représentées par le graphe d'état, à mémoriser afin de mieux estimer la fonction de récompense.

3.6.1 Encodeur de graphe

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

3.6.1.1 Implémentation

Le réseau de neurones a été implémenté en utilisant la bibliothèque Keras de Python. Nous avons utilisé des cellules GRUs (Gated Reccurent Units [18]) comme unités récurrentes pour l'encodeur et le décodeur vu leur efficacité comparable aux LSTMs tout en utilisant un seul vecteur d'état, ce qui les rend moins exigeants en mémoire. Comme notre tâche consiste à encoder un graphe dans un vecteur, la taille de ce dernier est très importante. Intuitivement, plus cette taille est grande plus le nombre de triplets qu'on peut y encoder est grand. D'où l'intérêt des cellules GRUs qui permettent d'utiliser de plus grands vecteurs d'état en moins d'espace mémoire que les LSTMs.

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

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

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

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

3.6.1.2 Résultats et discussion

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

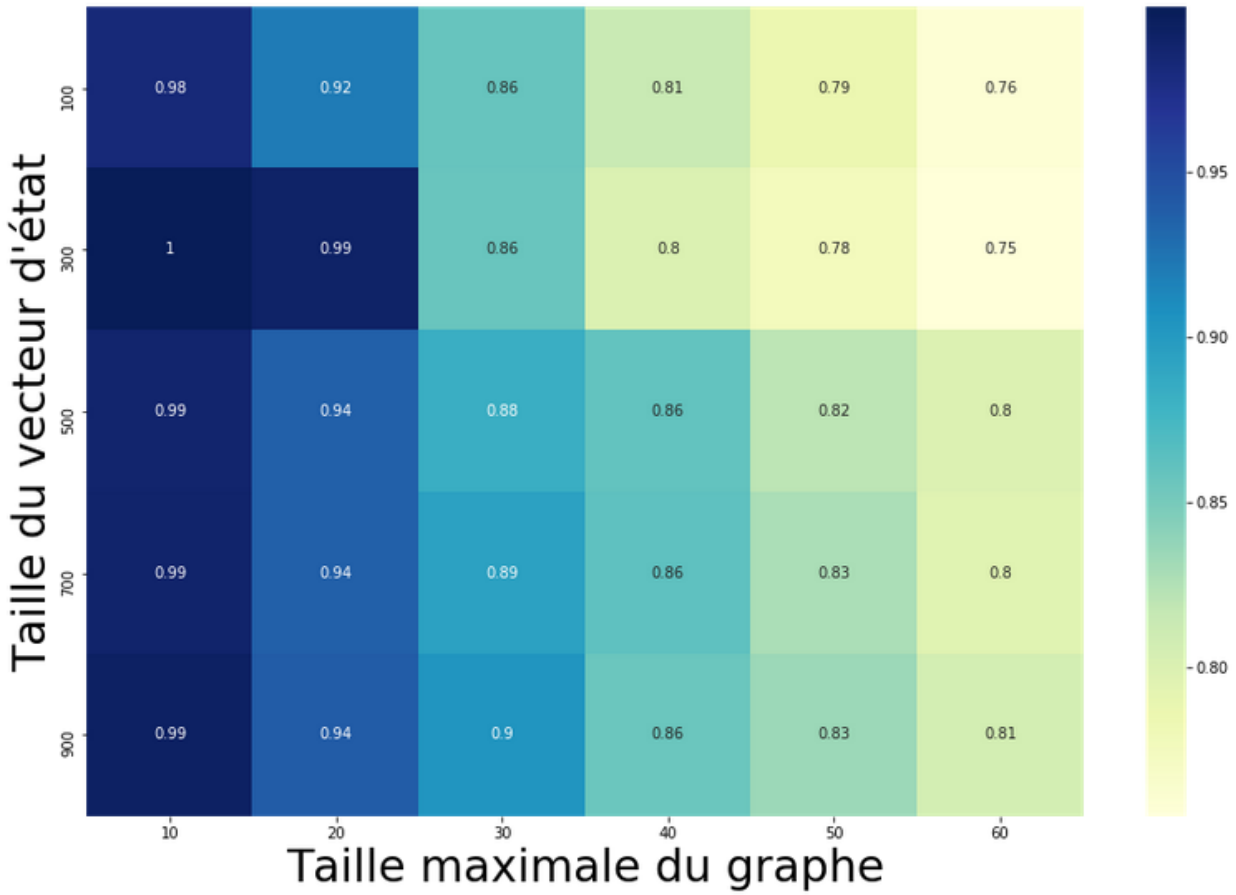

Figure 3.6 – *Variation de la précision en fonction de la taille maximale du graphe et la taille du vecteur d'état*

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

$$t(1) = nb$$

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

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

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

Pour essayer de 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.

3.6.2 Apprentissage par renforcement

Nous allons à présent présenter les deux méthodes citées dans la section 3.6. La première consiste à utiliser le résultat de l'encodeur pour entraîner un réseau de neurones à estimer la récompense associée à chaque action possible. La deuxième, par contre, fait l'apprentissage des deux parties à la fois.

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

Nous allons évaluer les différents résultats en comparant leur taux de succès qui est donné par le rapport entre le nombre de fois où l'agent arrive au but sur le nombre total d'essais.

Un succès nécessite d'arriver au but dans un nombre d'échanges limite que nous avons estimé comme suit :

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

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

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

3.6.2.1 Apprentissage avec DQN déconnecté

Dans cette partie, nous utilisons un des encodeurs déjà entraînés dans la partie précédente. Nous avons donc choisi d'utiliser l'encodeur de taille 300 qui a été entraîné avec des graphes de taille maximale de 20. Nous avons varié le nombre de vecteurs utilisés ainsi que les probabilités d'erreurs du simulateur. Les résultats sont présentés dans le tableau 3.14.

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

Table 3.14 – *Taux de réussite en fonction des probabilités d'erreurs et du nombre de vecteurs d'état. Avec les probabilités pb_i et pb_e : les probabilités d'erreurs sur l'intention et les emplacements respectivement.*

Le tableau 3.14 nous permet de conclure ce qui suit :

- Évidemment, avec moins de probabilités d'erreurs, le réseau arrive à mieux reconnaître les motifs dans le vecteur d'état et leurs relations avec les récompenses du simulateur. Cependant, l'augmentation de la taille du vecteur n'améliore que légèrement les résultats ; en calculons la moyenne des taux de réussite pour chaque nombre de vecteurs, les résultats sont comme suit : **53.5%** de taux de réussite pour quatre vecteurs, **53.75%** pour cinq vecteurs et **57.5%** pour six vecteurs.
- Le meilleur résultat a été obtenu en utilisant quatre vecteurs et sans erreurs avec un taux de réussite de **72%**. En pratique, ce modèle ne s'adapte pas aux erreurs des modules précédents. Par conséquent, il se perd lorsqu'une erreur se produit et il n'arrive pas à se resituer dans la conversation.
- Avec des probabilités d'erreurs très élevées, le réseau n'arrive pas à apprendre et ne dépasse pas 28% de taux de réussite.
- Les résultats les plus prometteurs dans ce cas sont celles qui sont entraînées 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.

3.6.2.2 Apprentissage avec DQN connecté

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

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

Table 3.15 – *Taux de réussite en fonction des probabilités d'erreurs et de la taille du vecteur d'état. Avec les probabilités pb_i et pb_e : les probabilités d'erreurs sur l'intention et les emplacements respectivement.*

- Comme pour la méthode précédente, l'augmentation des probabilités d'erreurs diminue le taux de réussite en général.
- Le meilleur taux de réussite est toujours obtenu en faisant un apprentissage sans erreurs ce qui a donné **93%** de réussite.
- Pour des probabilités d'erreurs élevées, le taux de réussite diminue considérablement par rapport aux autres valeurs de probabilités.
- Le réseau arrive à reconnaître les motifs encodés aussi bien pour de petites tailles du vecteur d'état que pour de grandes tailles. Cette méthode permet effectivement de réduire la taille du vecteur encodant le graphe d'état par rapport à son prédécesseur.

3.6.2.3 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 potentielle 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.

3.6.2.4 Taux de réussite

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

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

3.6.2.5 Vitesse d'apprentissage

Nous avons comparé les vitesses d'apprentissage des deux approches. En utilisant le réseau DQN séparé, ce dernier apprend avec une moyenne de **1013.15 instances/seconde**.

De l'autre côté, le réseau connecté ne fait que **197.61 instances/seconde**. La raison derrière la lenteur de la deuxième approche revient à la nécéssité de refaire l'encodage de tout le graphe d'état de l'instance sauvegardée pendant les échanges avec le simulateur. Tandis que dans la deuxième approche, on ne sauvegarde que le vecteur déjà encodé.

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

3.6.2.6 Courbe d'apprentissage

Enfin, nous comparons à présent les courbes d'apprentissage des deux méthodes. Celles-ci montrent le taux de réussite par rapport au nombre d'épisodes [7]. La figure 3.7 contient quatre courbes : deux courbes pour chaque méthode, avec et sans erreurs.

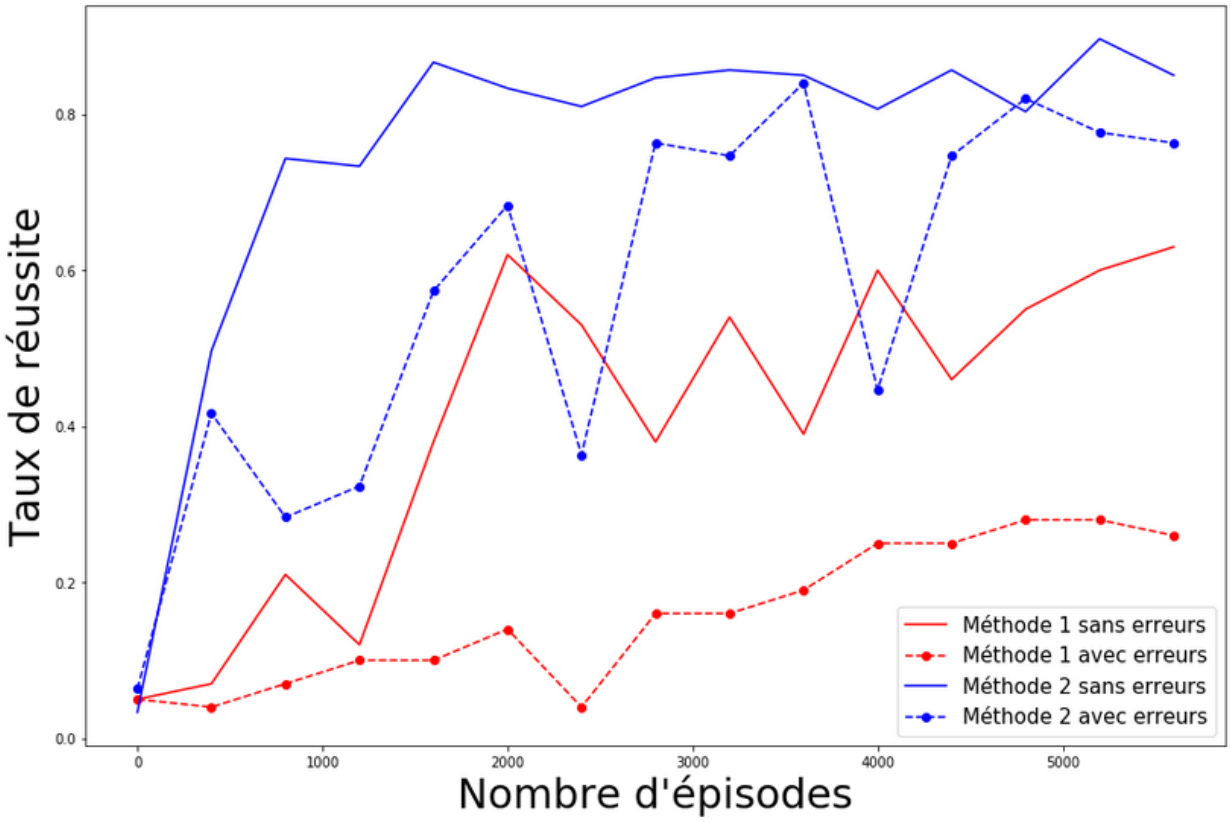

Figure 3.7 – *Les courbes d'apprentissage des deux méthodes proposées avec et sans erreurs du simulateur d'utilisateur*

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

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

L'apprentissage peut être amélioré dans les deux cas, et surtout en ce qui concerne le premier, en utilisant un meilleur encodage des nœuds et des arcs du graphe. En effet, les nœuds sont actuellement encodés avec des identifiants entiers seulement, perdant ainsi les riches connaissances sémantiques qu'on peut extraire des relations du graphe. Un meilleur encodage serait d'utiliser des méthodes d'apprentissage semi-supervisé qui permettraient de donner aux nœuds proches un encodage similaire. Cette méthode permet d'éviter quelques problèmes que nous avons cités dans 3.6.2.4, entre autres le problème d'encoder deux états lointains dans des vecteurs proches.

3.7 Application PCS2

L'application PCS2 relie les modules que nous avons implémentés dans un assistant qui aide à manipuler les fichiers de l'ordinateur avec la voix. Le schéma 3.8 montre les différentes parties que comporte l'application et les communications entre elles.

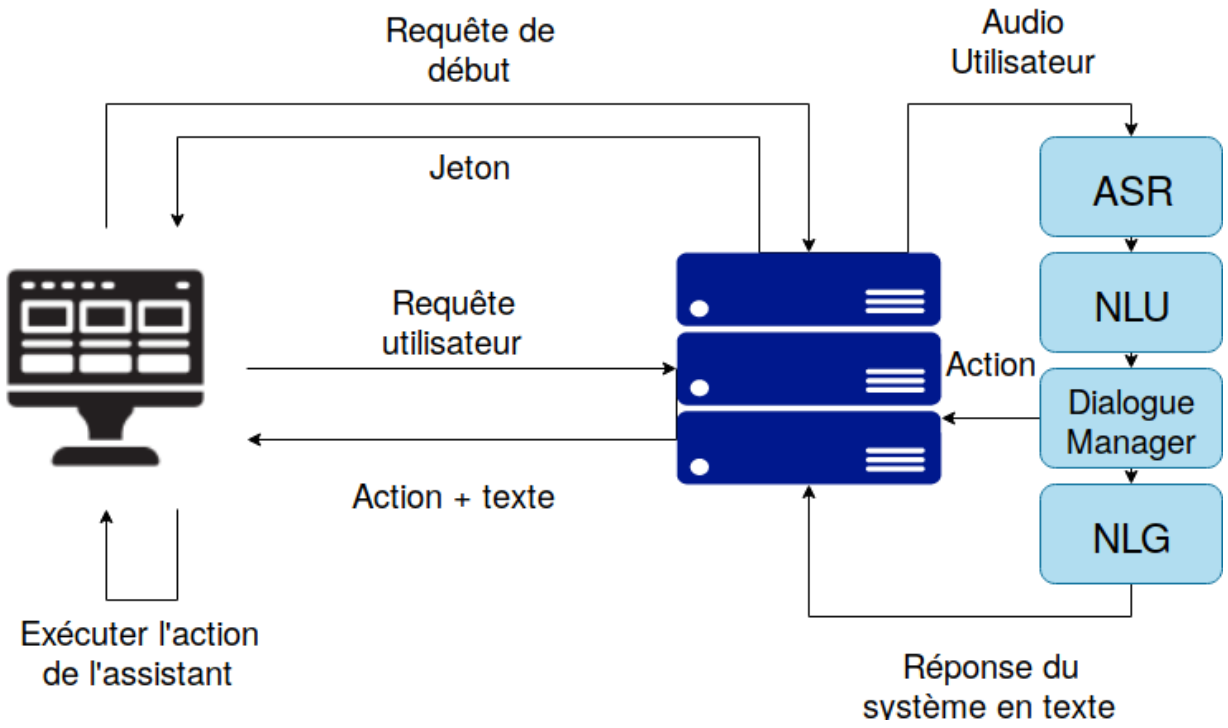

Figure 3.8 – *Schéma général de l'application PCS2*

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.

3.7.1 Backend

Pour implémenter le backend, nous avons utilisé le 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 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 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.

3.7.2 Frontend

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

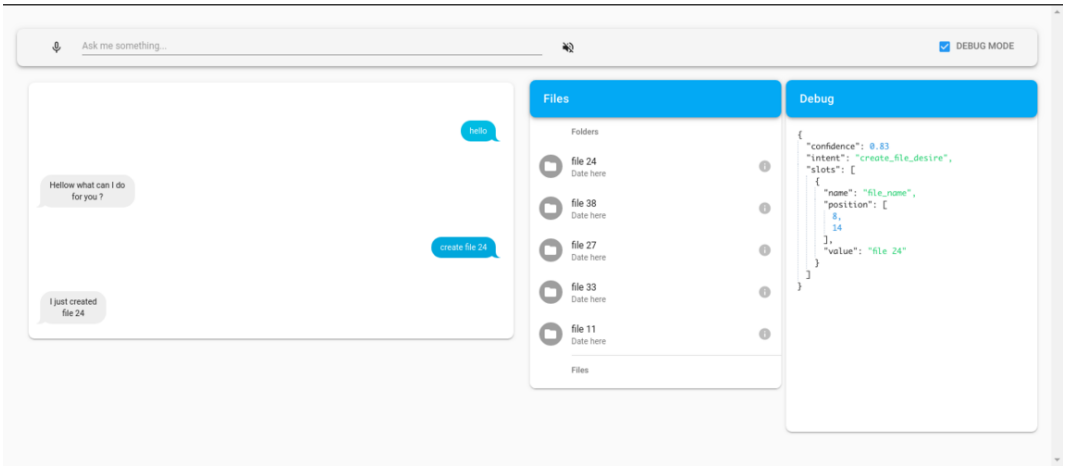

Figure 3.9 – *Interface principale de Bethano*

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 ai 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.

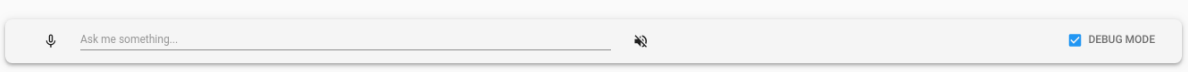

Figure 3.10 – *Champ de saisie de l'application*

- **Arborescence virtuelle** : C'est une petite fenêtre pour faciliter à l'utilisateur la mémorisation de l'environnement d'interaction. Cette fenêtre est dynamiquement mise à jour à travers le déroulement du dialogue et la manipulation des fichiers/répertoires.

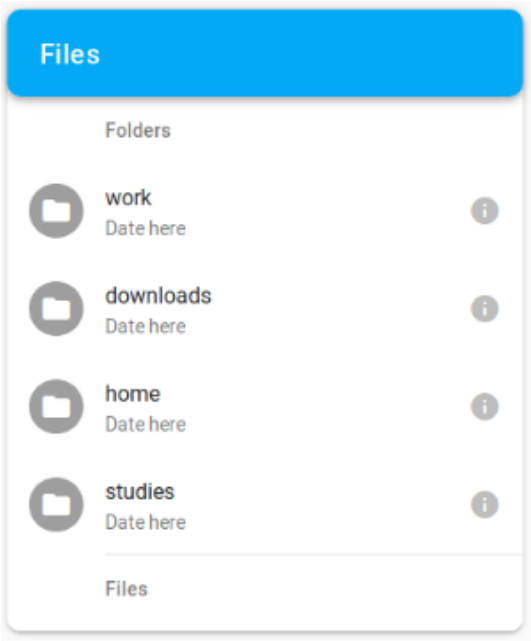

Figure 3.11 – *Fenêtre d'affichage de l'arborescence virtuelle*

- **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 avec les messages échangés par l'utilisateur et l'assistant Bethano. Il permet ainsi de garder trace de tous les échanges effectués et pouvoir à tout moment vérifier ou réutiliser des messages.

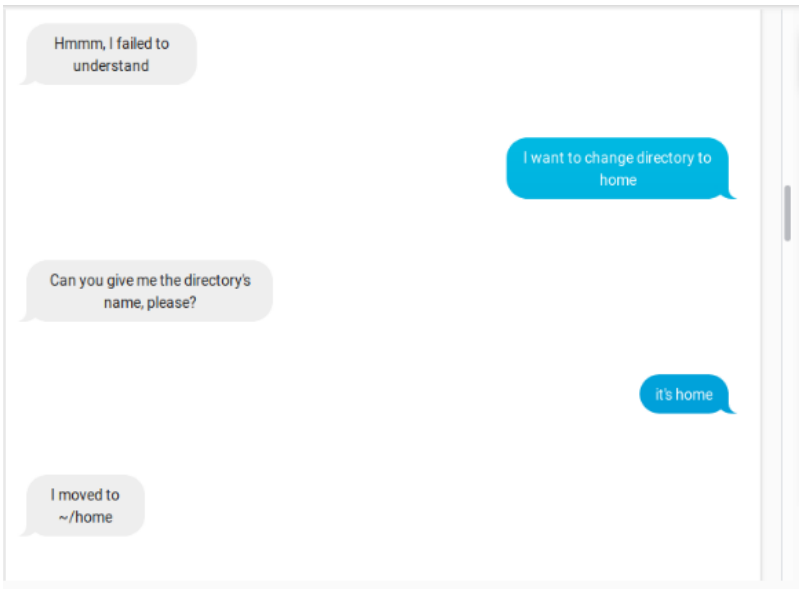

Figure 3.12 – *Fenêtre de dialogue avec l'assistant*

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

Figure 3.13 – *Fenêtre de Débogage*

3.8 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 Bethano. 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, Bethano a pu effectuer des petites tâches rudimentaires de manipulation du bureau sur un ordinateur et délivrer une mini-expérience de ce que peut être un véritable assistant virtuel intelligent.

Table des figures

Bibliographie

[1] mozilla/deepspeech : A tensorflow implementation of baidu's deepspeech architecture. https://github.com/mozilla/DeepSpeech. (Consulté le 10/06/2019).

[2] Programmation javascript/introduction wikilivres. https://fr.wikibooks.org/wiki/Programmation_JavaScript/Introduction. (Consulté le 10/06/2019).

[3] Programmation python/introduction wikilivres. https://fr.wikibooks.org/wiki/Programmation_Python/Introduction. (Consulté le 10/06/2019).

[4] wfuzz. https://github.com/xmendez/wfuzz/blob/master/wordlist/general/common.txt. (Consulté le 14/06/2019).

[5] G A. Rummery and Mahesan Niranjan. On-line q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166*, 11 1994.

[6] Fawaz Al-Anzi and Dia AbuZeina. Literature survey of arabic speech recognition. pages 1–6, 03 2018.

[7] Gabor Angeli, Christopher D. Manning, and Daniel Jurafsky. Parsing time : Learning to interpret time expressions. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics : Human Language Technologies*, NAACL HLT '12, Stroudsburg, PA, USA, 2012. Association for Computational Linguistics.

[8] K. J. Åström. Optimal control of Markov Processes with incomplete state information. *Journal of Mathematical Analysis and Applications*, 10:174–205, January 1965.

[9] H.B. Barlow. Unsupervised learning. *Neural Computation*, 1(3):295–311, 1989.

[10] John A. Bateman. Enabling technology for multilingual natural language generation : The kpml development environment. *Nat. Lang. Eng.*, 3(1), March 1997.

[11] Anja Belz. Automatic generation of weather forecast texts using comprehensive probabilistic generation-space models. *Nat. Lang. Eng.*, 14(4), October 2008.

[12] Dimitri Bertsekas. *Dynamic Programming & Optimal Control, Vol II : Approximate Dynamic Programming*. Athena Scientific, 01 2012.

[13] Julien Bloit and Xavier Rodet. Short-time viterbi for online hmm decoding : Evaluation on a real-time phone recognition task. pages 2121 – 2124, 05 2008.

[14] Tom Bocklisch, Joey Faulkner, Nick Pawlowski, and Alan Nichol. Rasa : Open source language understanding and dialogue management. *CoRR*, abs/1712.05181, 2017.

[15] Senthilkumar Chandramohan, Matthieu Geist, Fabrice Lefèvre, and Olivier Pietquin. User simulation in dialogue systems using inverse reinforcement learning. In *INTERSPEECH*, page 10251028, 2011.

[16] P. M. Chauhan and N. P. Desai. Mel frequency cepstral coefficients (mfcc) based speaker identification in noisy environment using wiener filter. In *2014 International Conference on Green Computing Communication and Electrical Engineering (ICGCCEE)*, pages 1–5, March 2014.

[17] Hongshen Chen, Xiaorui Liu, Dawei Yin, and Jiliang Tang. A survey on dialogue systems : Recent advances and new frontiers. *SIGKDD Explor. Newsl.*, 19(2):25–35, November 2017.

[18] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[19] Jan Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, KyungHyun Cho, and Yoshua Bengio. Attention-based models for speech recognition. *CoRR*, abs/1506.07503, 2015.

[20] Heriberto Cuayáhuitl, Steve Renals, Oliver Lemon, and Hiroshi Shimodaira. Human-computer dialogue simulation using hidden markov models. *IEEE Workshop on Automatic Speech Recognition and Understanding, 2005.*, pages 290–295, 2005.

[21] li Deng, Geoffrey Hinton, and Brian Kingsbury. New types of deep neural network learning for speech recognition and related applications : An overview. pages 8599–8603, 10 2013.

[22] Nina Dethlefs. Context-sensitive natural language generation : From knowledge-driven to data-driven techniques. *Language and Linguistics Compass*, 8, 03 2014.

[23] Michael Elhadad and Jacques Robin. An overview of surge : a reusable comprehensive syntactic realization component. In *International Natural Language Generation Workshop*, 1996.

[24] Dominic Espinosa, Michael White, and Dennis Mehay. Hypertagging : Supertagging for surface realization with ccg. In *ACL*, 2008.

[25] Roger Evans, P. Piwek, and Lynne Cahill. What is nlg ? In *Proceedings of the Second International Conference on Natural Language Generation*, 2002.

[26] Thiago Castro Ferreira, Iacer Calixto, Sander Wubben, and Emiel Krahmer. Linguistic realisation as machine translation : Comparing different mt models for amr-to-text generation. In *INLG*, 2017.

[27] G. D. Forney. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, March 1973.

[28] Albert Gatt and Emiel Krahmer. Survey of the state of the art in natural language

generation : Core tasks, applications and evaluation. *J. Artif. Int. Res.*, 61(1), January 2018.

[29] Kallirroi Georgila, James Henderson, and Oliver Lemon. Learning user simulations for information state update dialogue systems. In *INTERSPEECH*, pages 893–896, 2005.

[30] Zoubin Ghahramani. Hidden markov models. chapter An Introduction to Hidden Markov Models and Bayesian Networks, pages 9–42. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2002.

[31] Wiqas Ghai and Navdeep Singh. Literature review on automatic speech recognition. *International Journal of Computer Applications*, 41(8):42–50, March 2012. Full text available.

[32] R Glass and Michael Kyle Mccandless. Automatic acquisition of language models for speech recognition. 10 1994.

[33] D Goddeau, Helen Meng, Joseph Polifroni, Stephanie Seneff, and S Busayapongchai. A form-based dialogue manager for spoken language applications. volume 2, pages 701 – 704, 11 1996.

[34] Chih-Wen Goo, Guang Gao, Yun-Kai Hsu, Chih-Li Huo, Tsung-Chieh Chen, Keng-Wei Hsu, and Yun-Nung Chen. Slot-gated modeling for joint slot filling and intent prediction. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics : Human Language Technologies, Volume 2 (Short Papers)*, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[35] Raghav Goyal, Marc Dymetman, and Eric Gaussier. Natural language generation through character-based rnns with finite-state prior knowledge. In *COLING*, pages 1083–1092, 2016.

[36] A. Graves, A. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649, May 2013.

[37] M. A. K. Halliday and Christian M. I. M. Matthiessen. *Introduction to Functional Grammar*. Hodder Arnold, London, 3 edition, 2004.

[38] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. Deep speech : Scaling up end-to-end speech recognition. *CoRR*, abs/1412.5567, 2014.

[39] James Henderson, Oliver Lemon, and Kallirroi Georgila. Hybrid reinforcement/supervised learning of dialogue policies from fixed data sets. *Comput. Linguist.*, 34(4):487–511, December 2008.

[40] Matthew Henderson, Blaise Thomson, and Steve J. Young. Deep neural network approach for the dialog state tracking challenge. In *SIGDIAL Conference*, pages 467–471, 2013.

[41] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural

nets and problem solutions. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 6(2):107–116, April 1998.

[42] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[43] Daniel Jurafsky and James H. Martin. *Speech and Language Processing, 2nd Edition*. Prentice Hall, 2008.

[44] John G. Kemeny and J. Laurie Snell. Markov processes in learning theory. *Psychometrika*, 22(3):221–230, Sep 1957.

[45] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[46] Sotiris Kotsiantis, I Zaharakis, and P Pintelas. Machine learning : A review of classification and combining techniques. *Artificial Intelligence Review*, 26:159–190, 11 2006.

[47] Cyril Labbé and François Portet. Towards an abstractive opinion summarisation of multiple reviews in the tourism domain. *CEUR Workshop Proceedings*, 917, 01 2012.

[48] Irene Langkilde-Geary. Forest-based statistical sentence generation. In *ANLP*, 2000.

[49] Yann LeCun, Y Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521:436–44, 05 2015.

[50] Cheongjae Lee, Sangkeun Jung, Kyungduk Kim, Donghyeon Lee, and Gary Geunbae Lee. Recent approaches to dialog management for spoken dialog systems. *JCSE*, 4:1–22, 2010.

[51] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. Gated graph sequence neural networks. *CoRR*, abs/1511.05493, 2016.

[52] Zachary Chase Lipton. A critical review of recurrent neural networks for sequence learning. *CoRR*, abs/1506.00019, 2015.

[53] Bing Liu and Ian Lane. Attention-based recurrent neural network models for joint intent detection and slot filling. *CoRR*, abs/1609.01454, 2016.

[54] Pengfei Liu, Xipeng Qiu, and Xuanjing Huang. Recurrent neural network for text classification with multi-task learning. *CoRR*, abs/1605.05101, 2016.

[55] Tom M Mitchell. The discipline of machine learning. 01 2006.

[56] William C. Mann and Christian M. I. M. Matthiessen. Nigel : A systemic grammar for text generation. 1983.

[57] Mitchell Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. The penn treebank : Annotating predicate argument structure. In *Proceedings of the Workshop on Human Language Technology*, pages 114–119, Stroudsburg, PA, USA, 1994. Association for Computational Linguistics.

[58] Pierrick Milhorat, Stephan Schlögl, Gerard Chollet, Boudy , Anna Esposito, and G Pelosi. Building the next generation of personal digital assistants. 03 2014.

[59] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518, 2015.

[60] Fionn Murtagh. Multilayer perceptrons for classification and regression. *Neurocomputing*, 2(5-6):183–197, jul 1991.

[61] Shreya Narang and Ms. Divya Gupta. Speech feature extraction techniques : A review. 2015.

[62] Christopher Olah. Understanding lstm networks – colah's blog. `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`, Août 2015. (Accessed on 02/19/2019).

[63] Douglas D. O'Shaughnessy. Linear predictive coding. *IEEE Potentials*, 7:29–32, 1988.

[64] R. Rabiner and B. H. Juang. An introduction to hidden markov models. *IEEE ASSP Magazine*, 3:4–16, 1986.

[65] Hajer Rahali, Zied Hajaiej, and Noureddine Ellouze. Robust features for speech recognition using temporal filtering technique in the presence of impulsive noise. *International Journal of Image, Graphics and Signal Processing*, 6:17–24, 10 2014.

[66] Ehud Reiter and Robert Dale. Building applied natural language generation systems. *Natural Language Engineering*, 3(1), March 1997.

[67] title = "A Markovian Decision Process Richard Bellman". *Indiana Univ. Math. J.", fjournal = "Indiana University Mathematics Journal*, 6:679–684, 1957.

[68] Brian Roark, Murat Saraclar, and Michael Collins. Discriminative n-gram language modeling. *Comput. Speech Lang.*, 21(2), April 2007.

[69] F. Rosenblatt. *Perceptron simulation experiments (Project Para)*. 1959.

[70] Jost Schatzmann, Blaise Thomson, Karl Weilhammer, Hui Ye, and Steve J. Young. Agenda-based user simulation for bootstrapping a pomdp dialogue system. In *HLT-NAACL*, 2007.

[71] M. Schuster and K.K. Paliwal. Bidirectional recurrent neural networks. *Trans. Sig. Proc.*, 45(11), November 1997.

[72] Iulian V. Serban, Alessandro Sordoni, Yoshua Bengio, Aaron Courville, and Joelle Pineau. Building end-to-end dialogue systems using generative hierarchical neural network models. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI, pages 3776–3783. AAAI Press, 2016.

[73] B.Maind. Sonali. Research paper on basic of artificial neural network. page 1, 2014.

[74] Alessandro Sordoni, Michel Galley, Michael Auli, Chris Brockett, Yangfeng Ji, Margaret Mitchell, Jian-Yun Nie, Jianfeng Gao, and William B. Dolan. A neural network approach to context-sensitive generation of conversational responses. In *HLT-NAACL*, pages 196–205, 2015.

[75] Svetlana Stoyanchev and Michael Johnston. Knowledge-graph driven information state approach to dialog. In *AAAI Workshops*, 2018.

[76] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

[77] M. Theune, E. Klabbers, J. R. De Pijper, E. Krahmer, and J. Odijk. From data to speech : A general approach. *Natural Language Engineering*, 7(1), March 2001.

[78] Marc Velay and Fabrice Daniel. Seq2seq and multi-task learning for joint intent and content extraction for domain specific interpreters. *CoRR*, abs/1808.00423:3–4, 2018.

[79] Yu Wang, Yilin Shen, and Hongxia Jin. A bi-model based rnn semantic frame parsing model for intent detection and slot filling. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics : Human Language Technologies, Volume 2 (Short Papers)*, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[80] Gellért Weisz, Pawex0142 Budzianowski, Pei hao Su, and Milica Gax0161ix0107. Sample efficient deep reinforcement learning for dialogue systems with large action spaces. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 26:2083–2097, 2018.

[81] Tsung-Hsien Wen, Milica Gasic, Nikola Mrksic, Pei hao Su, David Vandyke, and Steve J. Young. Semantically conditioned lstm-based natural language generation for spoken dialogue systems. In *EMNLP*, 2015.

[82] Tsung-Hsien Wen, Milica Gasic, Nikola Mrksic, Lina Maria Rojas-Barahona, Pei hao Su, Stefan Ultes, David Vandyke, and Steve J. Young. A network-based end-to-end trainable task-oriented dialogue system. In *EACL*, pages 438–449, 2017.

[83] Michael Wessel, Girish Acharya, James Carpenter, and Min Yin. *OntoVPAAn Ontology-Based Dialogue Management System for Virtual Personal Assistants : 8th International Workshop on Spoken Dialog Systems*. 01 2019.

[84] J. D. Williams and S. Young. Scaling pomdps for spoken dialog management. *Trans. Audio, Speech and Lang. Proc.*, 15(7):2116–2129, September 2007.

[85] Steve Young, Milica Gašić, Simon Keizer, François Mairesse, Jost Schatzmann, Blaise Thomson, and Kai Yu. The hidden information state model : A practical framework for pomdp-based spoken dialogue management. *Comput. Speech Lang.*, 24(2):150–174, April 2010.

[86] Dong Yu and Li Deng. *Automatic Speech Recognition*. Springer London, 2015.

[87] Jin Yu, Ehud Reiter, Jim Hunter, and Chris Mellish. Choosing the content of tex-

tual summaries of large time-series data sets. *Natural Language Engineering*, 13(1), March 2007.