Table des matières

To correct
To explain more deeply

Chapitre 1

Conception du système

1.1 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.

1.2 Architecture du système

Comme montré dans la figure 1.1 et comme cité dans le chapitre précédent (voir **??**) 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 **??**). Nous pouvons séparer ces parties en deux catégories : la partie utilisateur et la partie interne du système que nous présentons ci-dessous.

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

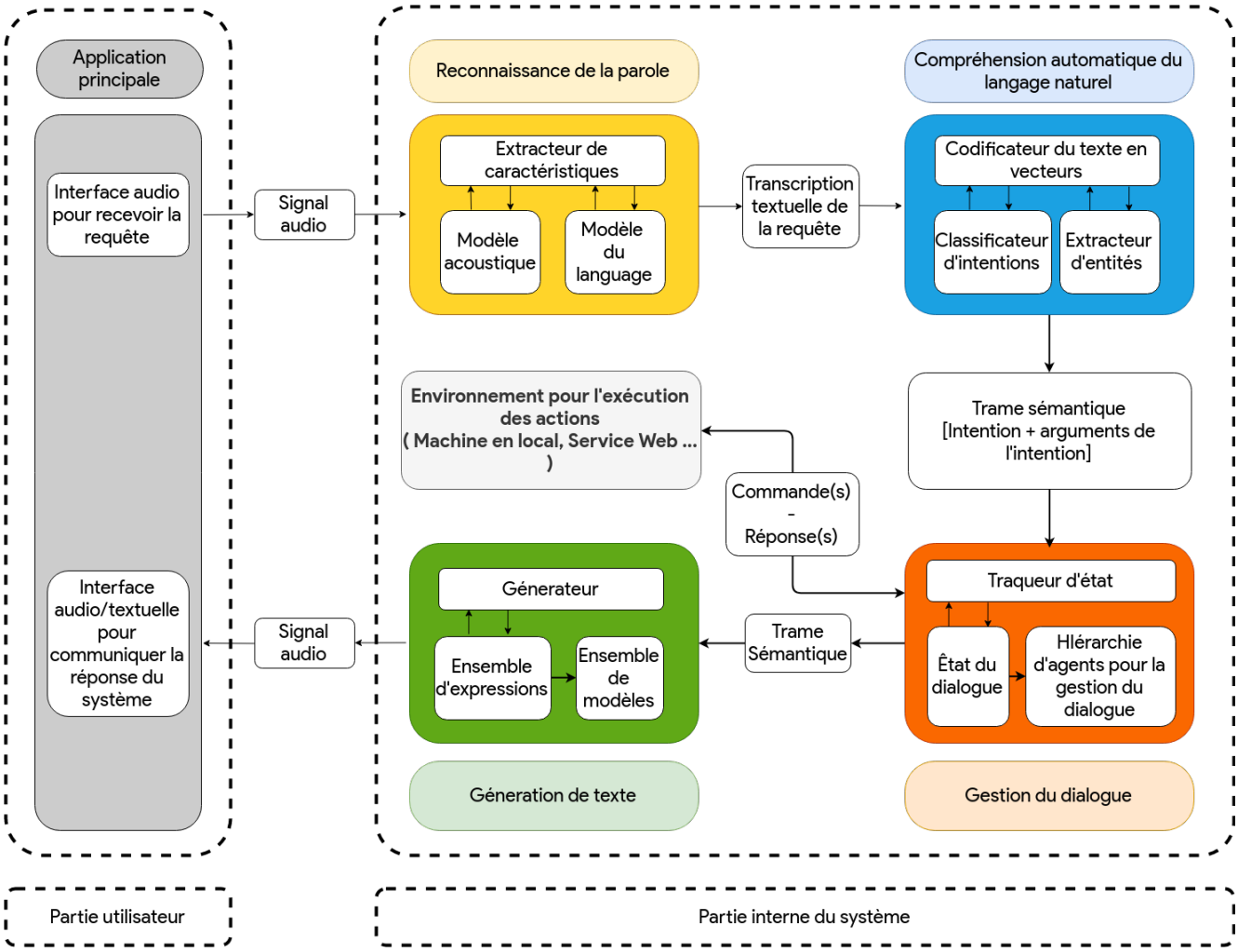

Figure 1.1 – *Architecture générale du système Bethano*

1.2.1 Partie utilisateur

Cette partie représente ce que l'utilisateur peut voir comme entrée/sortie et les interfaces qui lui sont accessibles. Puisque l'assistant est un processus qui communique majoritairement avec l'utilisateur à travers des échanges verbaux, nous avons pensé à implémenter l'interface du système comme un processus qui s'exécute en arrière plan et qui attend d'être activé (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.)

1.2.2 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 **??**), 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.

1.3 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 [5]. 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.

1.3.1 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

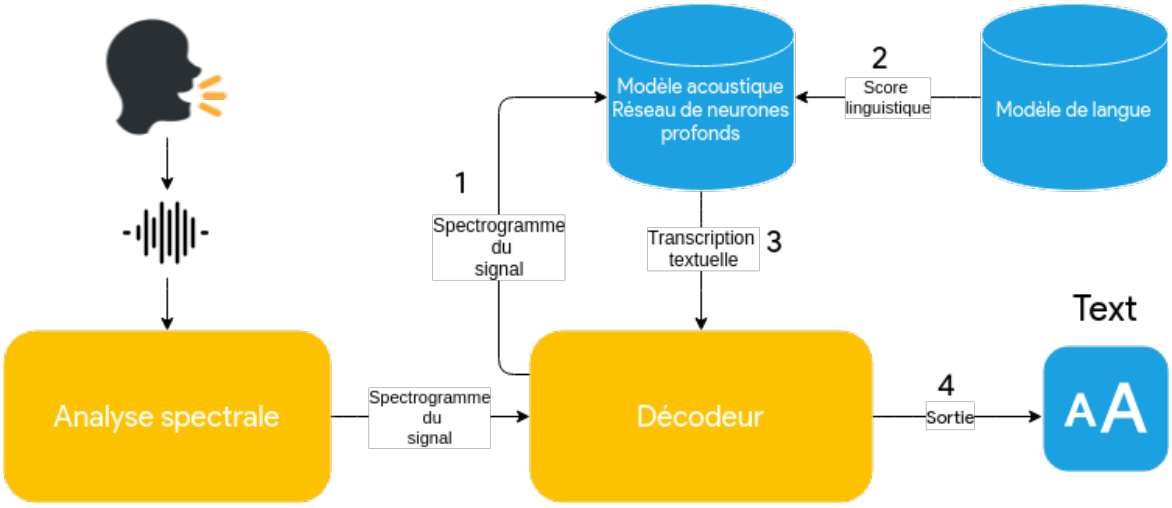

Figure 1.2 – *Architecture du module de reconnaissance de la parole (ASR)*

1.3.2 Modèle acoustique

Type du modèle

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

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

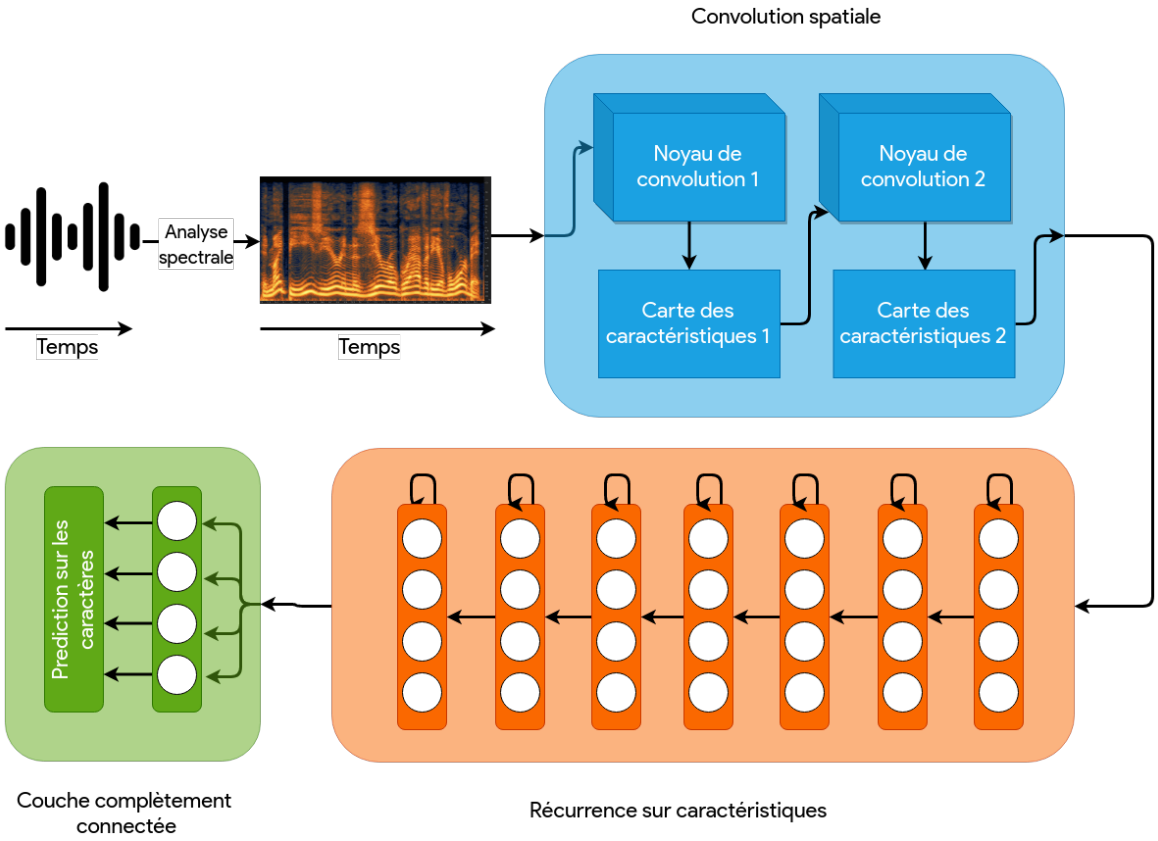

Figure 1.3 – *Architecture du modèle DeepSpeech [5]*

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 [5] 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

1.3.3 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 [5], 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 :

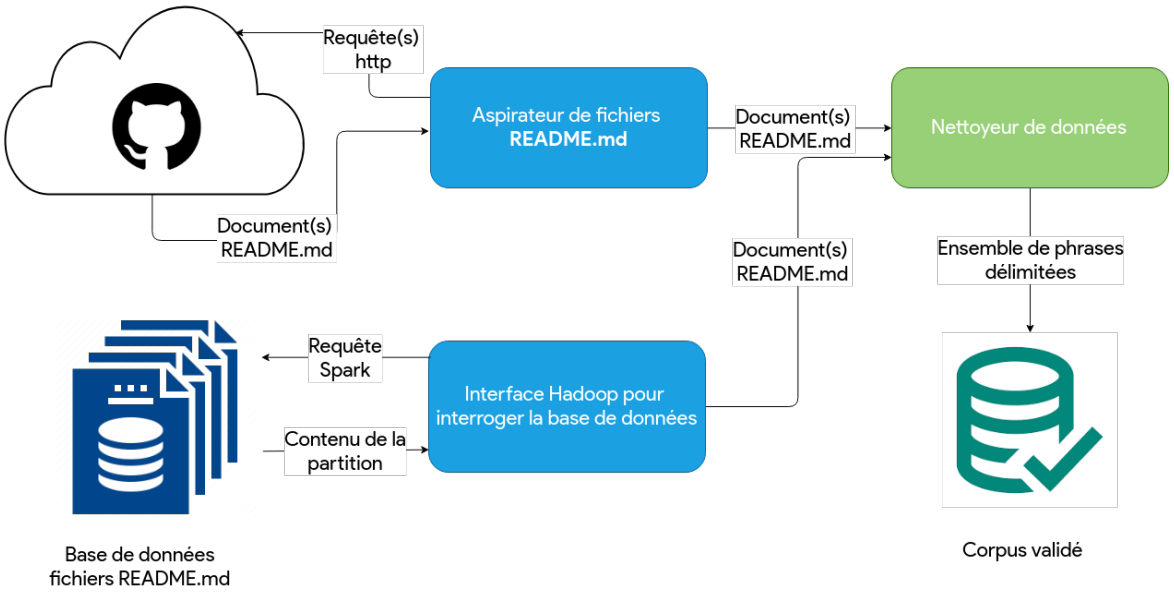

Figure 1.4 – *Processus de génération du corpus pour le modèle de langue*

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

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

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

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

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

- Les deux sources de données envoient ensuite les fichiers récoltés au nettoyeur de fichiers pour en extraire seulement les parties qui ont du sens dans le langage naturel (paragraphes, titres, instructions, etc.)
- Le corpus final est ensuite construit à partir des paragraphes extraits à l'étape précédente après les avoir 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>
```

1.4 Module de compréhension automatique du langage naturel

Second module du système, le module de compréhension automatique du langage naturel (Natural Langage Understanding, NLU) 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 ([4],[8]) et cela malgré le petit volume des données d'apprentissage. Cette option nous a paru plus abordable que la construction d'un analyseur basé règles, souvent assez rigide et dont l'exhaustivité n'est pas évidente à obtenir.

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

1.4.1 Architecture du module

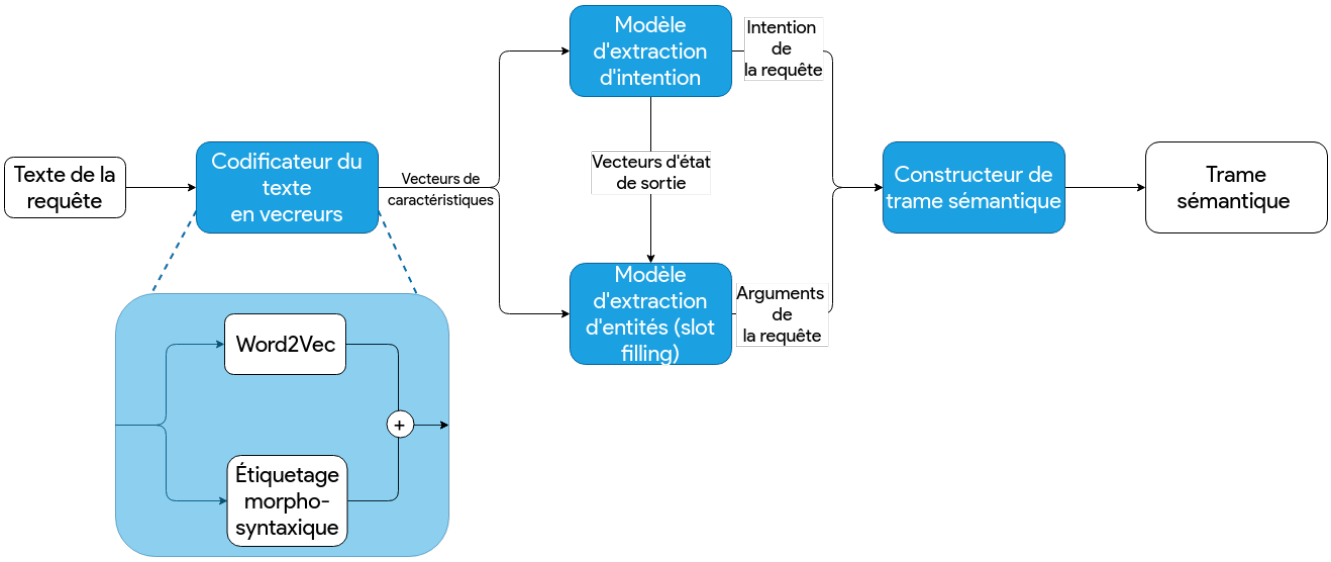

Figure 1.5 – *Architecture du module de compréhension automatique du langage naturel (NLU)*

Comme précédemment cité (voir la 1.2.2), le module NLU possède une architecture en pipeline qui reçoit en entrée le texte brut de la requête. Sa codification varie selon les approches que nous avons explorées et qui seront plus explicitées dans le chapitre suivant Réalisation et résultats. Pour mieux capturer l'aspect sémantique des mots dans le texte, nous avons décidé d'utiliser le modèle Word2Vec pré-entraîné par Google (entraîné sur 100 milliard de mots) pour produire un vecteur de taille fixe pour chaque mot. Pour encoder l'information syntaxique de la requête nous avons concaténé au vecteur de prolongement de chaque mot de la requête (Word Embedding Vector) le vecteur codifiant son étiquette morphosyntaxique. Après avoir codifié la séquence de mots, elle est envoyée aux modèles de classification d'intentions et d'extraction d'entités [9], qui sont en fait un seul modèle joint dont l'architecture est détaillée dans la section 1.4.2.1. 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.

1.4.2 Analyse sémantique avec apprentissage automatique

1.4.2.1 Modèle(s) utilisé(s)

Comme vu dans le chapitre précédent (voir **??**) 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 **??**).

1.4.2.2 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 [3] 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 [1]. 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 **??** 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 [9].
- **text** : le texte de la requête nettoyé et dont les mots sont séparés uniquement par un espace.

1.5 Module de gestion du dialogue

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

1.5.1 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'action : Celle-ci détermine l'action à prendre à partir d'un état donné.

1.5.1.1 État du dialogue

Avant de détailler les deux modules du gestionnaire. Il est nécessaire d'introduire une représentation de l'état du dialogue. Classiquement, les trames sémantiques ont été utilisées (voir **??**). Le suivi d'état se fait dans ce cas en gardant trace des emplacements remplis durant le dialogue. Nous avons opter à utiliser une représentation plus riche ; les graphes de connaissances 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[12] et des ontologies[13] pour représenter l'état du système de dialogue. À partir de ce dernier une base de règles décide quelle action à prendre directement du graphe de connaissances. L'avantage par rapport à l'utilisation des trames sémantiques apparaît dans la flexibilité et le dynamisme des graphes de connaissances. En effet, pour une tâche comme la navigation dans les fichiers, l'état de l'arborescence des fichiers est sujet à des

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

1.5.1.2 Le 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 NLU dans le graphe de connaissances. Ceci consiste à transformer la trame sémantique en un graphe qui est ensuite ajouté au graphe d'état. Plus de détails seront donnés dans la section 1.5.2 où nous construirons une ontologie pour définir un vocabulaire de dialogue et comment elle peut être utilisé pour passer du résultat du NLU en un graphe de connaissances.

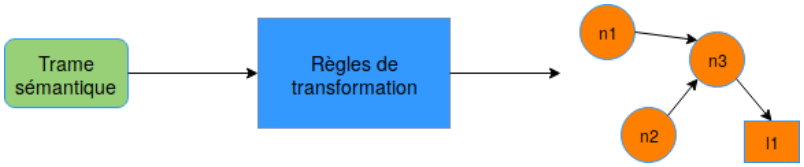

Figure 1.6 – *Schéma de transformation de trame sémqntique en graphe*

1.5.1.3 La politique d'action

La politique d'action peut être écrite manuellement, apprise à partir d'un corpus ou avec 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 basé 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.

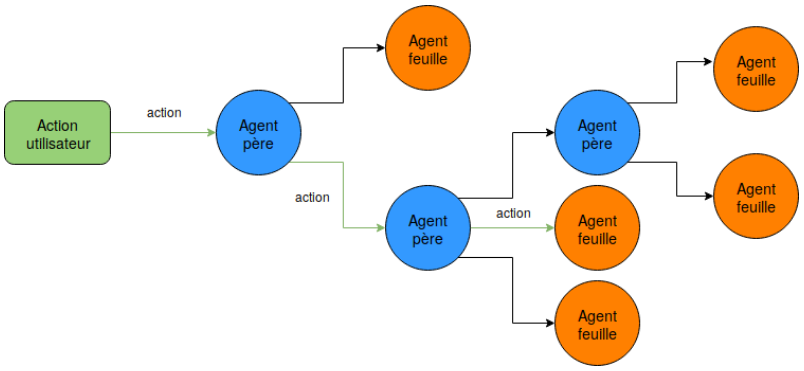

Figure 1.7 – *Schéma de l'architecture multi-agents*

L'avantage d'une telle architecture 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 capable de traiter cette 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 communique avec l'agent parent. Comme on connaît pour chaque simulateur l'agent fils qui lui correspond, il est donc possible de faire un apprentissage supervisé où les entrées sont les actions des simulateurs et l'état du système, tandis que la sortie est l'agent fils qui peut répondre à l'action.

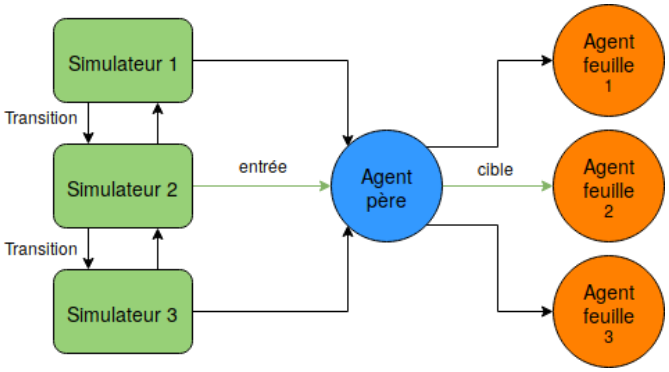

Figure 1.8 – *Schéma représentant l'apprentissage des agents parents avec les simulateurs des agents feuilles*

Pour résumer l'architecture globale du gestionnaire de dialogue, lorsque une nouvelle action utilisateur arrive au système, le traqueur d'état la reçoit. Il met à jour l'état du système en transformant l'action en un graphe de connaissances pour l'ajouter au graphe d'état. Ce nouvel graphe d'état ainsi que la dernière action reçue sont transmis à une architecture multi-agents hiérarchique qui va décider quelle action le système de dialogue doit prendre.

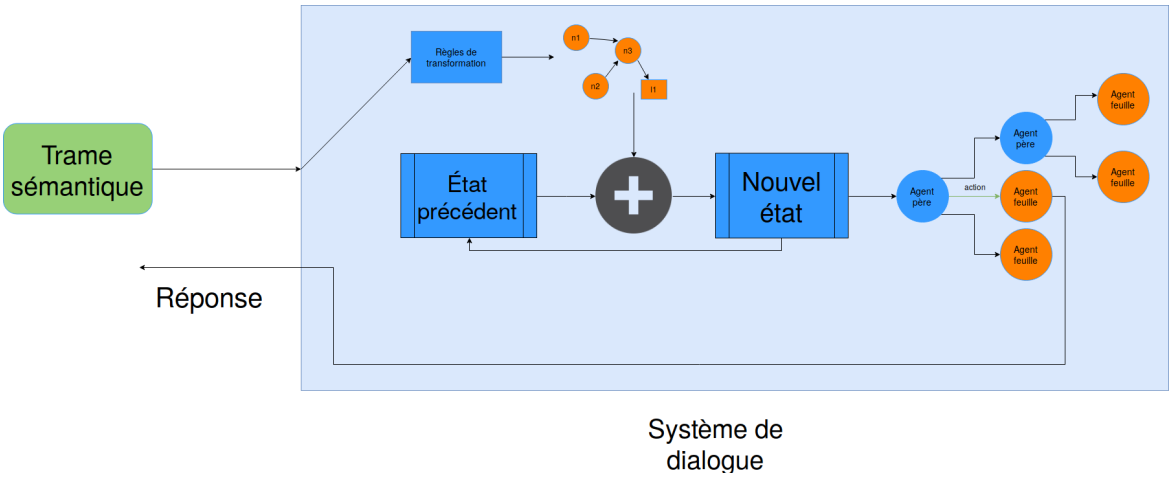

Figure 1.9 – *Schéma global du gestionnaire de dialogue*

1.5.2 Les ontologies du système

Une ontologie 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. Ce qui permettra à notre gestionnaire de comprendre le dialogue et les tâches qu'il peut réaliser.

1.5.2.1 Ontologie de dialogue

D'abord on définit une ontologie de dialogue qui contient des concepts qui peuvent aider un assistant d'ordinateur pour gérer son dialogue.

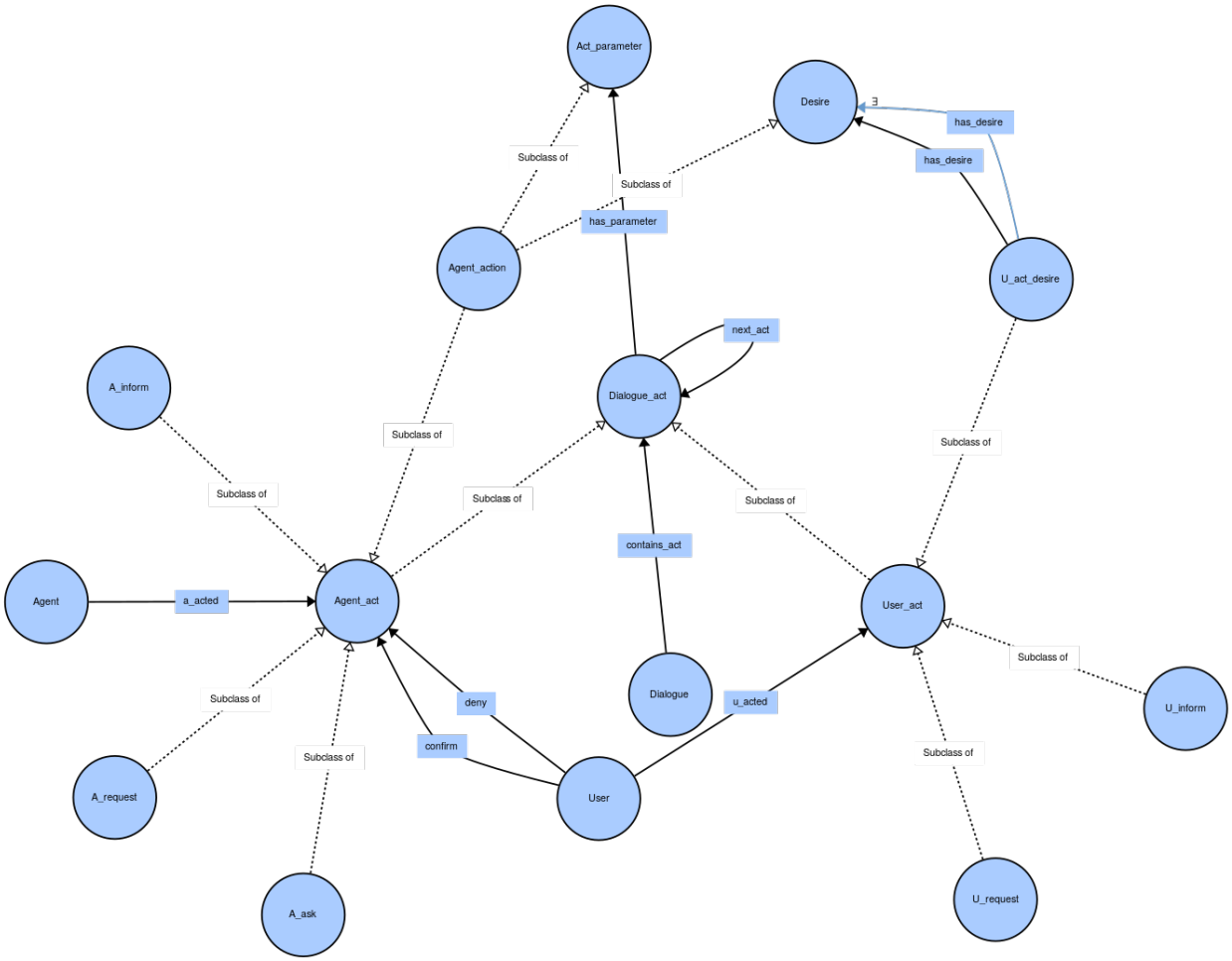

Figure 1.10 – *Graphe de l'ontologie de dialogue*

Principalement l'ontologie se compose de six classes mères :

- $Agent$ et $User$: ce sont les classes qui représentent l'utilisateur et l'agent qui participent au dialogue.
- $Dialogue$: l'agent et l'utilisateur participe à un dialogue, ce dernier contient les actions des deux participants.
- $Dialogue\_act$: 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 en 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 1.11. Une nouvelle action utilisateur est créée ainsi que ses paramètres et les relations entre eux.

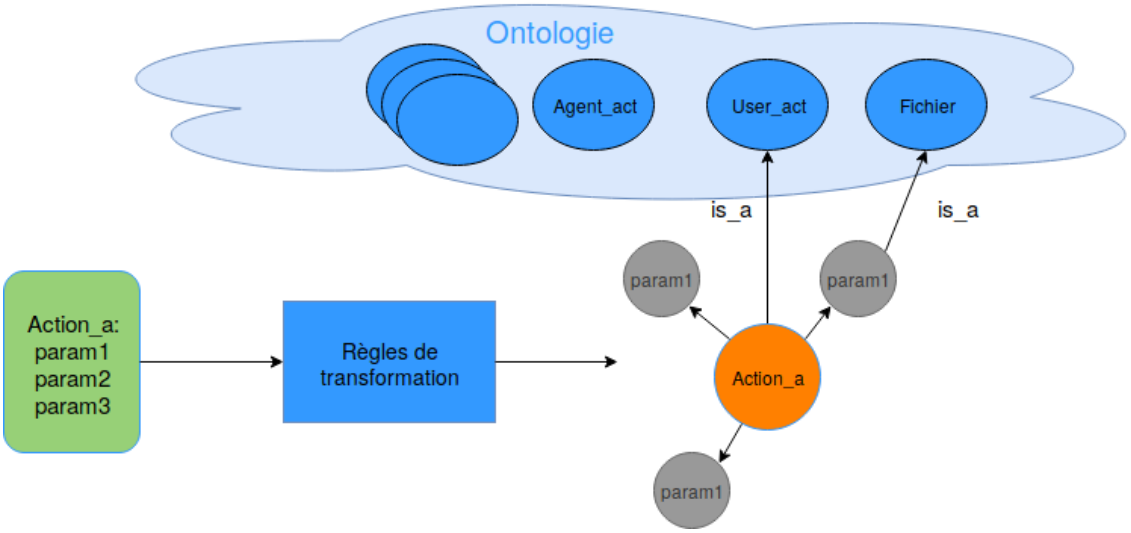

Figure 1.11 – *Schéma de transformation d'une action en graphe*

Ontologie pour l'exploration de fichiers

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

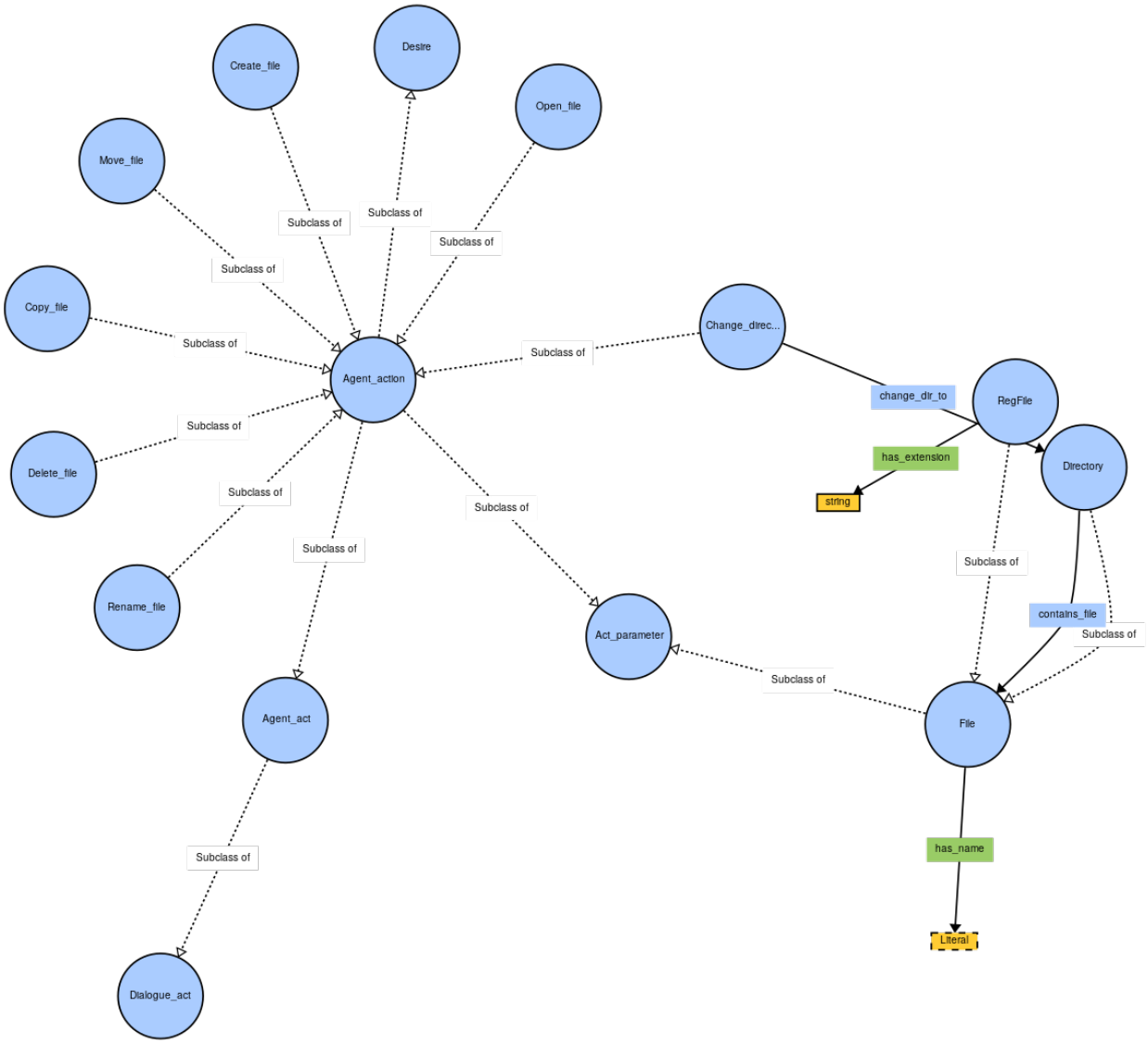

Figure 1.12 – *Graphe de l'ontologie de l'exploration de fichiers*

L'ontologie contient essentiellement :

- Des actions sur les fichiers : Créer un fichier, supprimer un fichier, changer de répertoire, etc. Ces actions sont des sous-classes de la classe $Agent\_act$ vu précédemment ainsi que les classes $Act\_parameter$ et $Desire$. Ce qui veut dire, d'un coté, que ces actions peuvent être des paramètres d'autres actions, comme demander à l'utilisateur s'il veut que l'assistant réalise une action donné, et d'un autre coté, que l'utilisateur peut demander à l'assistant de faire une de ces actions.
- Les concepts qui ont relation avec 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 ces concepts sont aussi définies comme un répertoire peut contenir des fichiers, ou une action de changement de répertoire doit avoir comme paramètre un répertoire cible.

La figure 1.13 représente l'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.

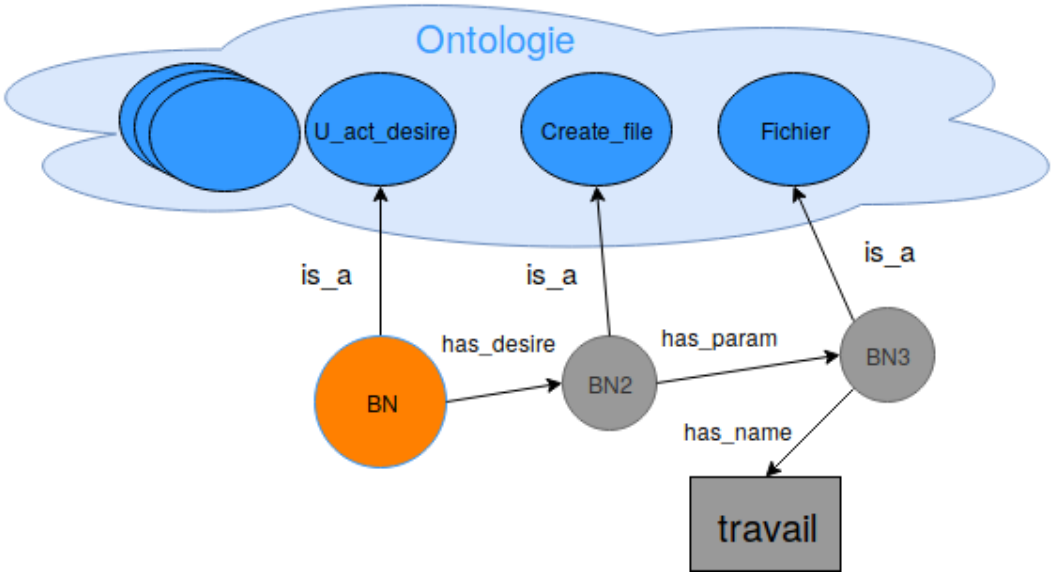

Figure 1.13 – *Schéma de transformation d'une action de demande de création de fichier en graphe*

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

1.5.3 Les simulateurs d'utilisateurs

Plusieurs méthodes peuvent être utilisées pour la création de simulateurs d'utilisateurs (voir **??**). Les simulateurs basés sur des méthodes d'apprentissage sont les plus robustes. Cependant, ils nécessitent un nombre important de données. L'alternative c'est d'utiliser des simulateurs basés règles. Nous nous sommes inspirés des simulateurs basés agenda[11] 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, une agenda est créée, celle-ci 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 qu'on peut y accéder ne sont pas fixes.
Le simulateur commence d'abord par générer une arborescence aléatoire qui représente la situation initiale du système. Ensuite, le simulateur 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 addition 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 couper 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 ; sorties : rcompense, fin, succs)
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
```

- **lignes 3-6 :** cas échec on met fin à **vrai** et on retourne une réponse de fin.
- **ligne 8 :** la fonction **maj_état** met à jour l'état du simulateur et vérifie si on est arrivé à un état de succès.
- **lignes 9-11 :** cas succès on met fin à **vrai** et on retourne une réponse de fin.
- **ligne 13 :** le simulateur décide quelle action à prendre selon l'action de l'agent. Le diagramme 1.14 résume cette décision.

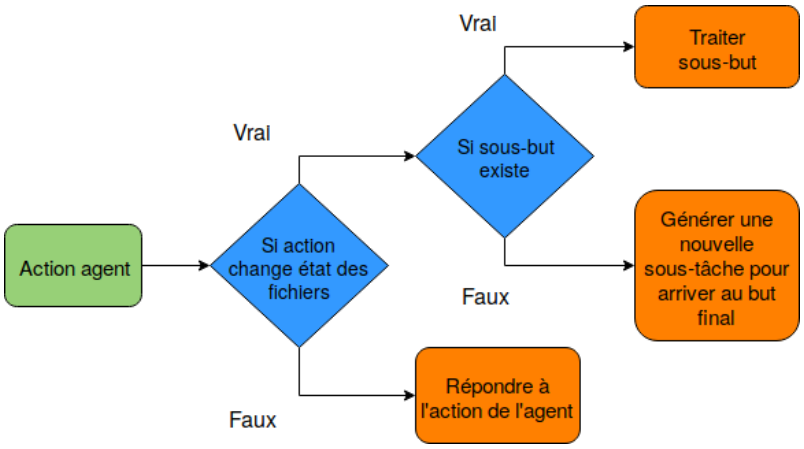

Figure 1.14 – *Diagramme de décision de l'action à prendre*

En ce qui concerne la fonction de récompense, celle-ci est évaluée à partir des changements effectués sur l'état de l'utilisateur, le tableau 1.1 associe les changements avec leurs récompenses :

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

Table 1.1 – *Tableau des récompenses*

- **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.

  Si cette valeur s'améliore, on donne une récompense positive à l'agent.
- **Succès :** c'est à dire, réaliser le but final du simulateur. Pour arriver à cet événement, l'agent ne fait qu'améliorer la similarité, 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.
- **Similarité diminuée :** la récompense est dans ce cas négative et supérieur 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.

- **Confirmation d'une question :** l'utilisateur peut confirmer une action à l'agent. La récompense est nul pour que l'agent puisse demander une confirmation quand il n'est pas sûr de ce qu'il doit faire sans diminuer le cumule des récompenses reçues.
- **Autre :** La récompense est de -1 pour éviter que le dialogue dure longtemps.

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 traité directement, sinon le simulateur initie le traitement d'une nouvelle sous-tâche. C'est à dire, Si un sous-but existe, une action qui le traite est générée, sinon une action qui traite le but final est générée.

1.5.4 Modèles d'apprentissage

Comme on l'a déjà cité dans le chapitre précédent, il existe plusieurs algorithmes d'apprentissage par renforcement comme Q-Learning ou State-Action-Reward-State-Action (SARSA)[2]. 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 état qu'ils n'ont pas vus pendant l'apprentissage. Pour pallier à ce problème, Deep Q Learning (DQL)[10] 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 difficile 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)[6] qui s'avère être des variantes des Gated Graph Neural Networks (GGNN)[7]. Ce dernier utilise des réseaux de neurones récurrent entre chaque deux nœuds reliés par une arrête pour transférer l'information d'un nœud à un autre, ce qui résulte en des vecteurs ayant un encodage de l'information associé à chaque nœud. 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, les vecteurs d'états de chaque nœud sont sommés pour produire un vecteur fixe encodant tout le graphe qui peut être relié au reste du réseau de neurones pour l'apprentissage.

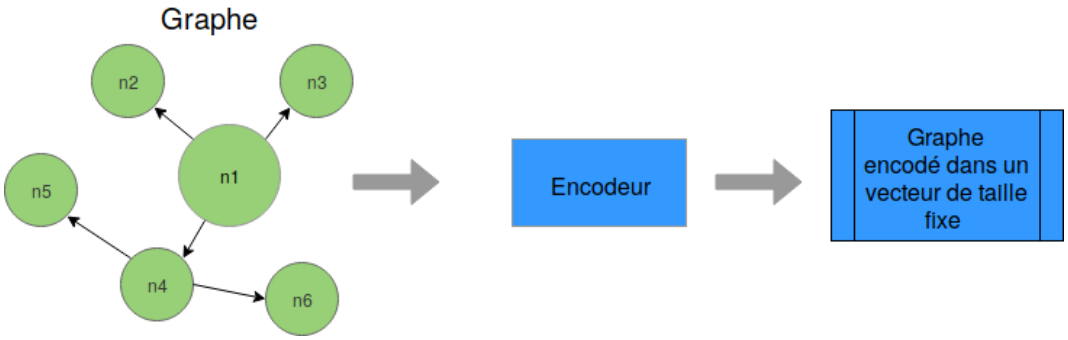

Figure 1.15 – *Schéma représentant un encodeur de graphe*

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

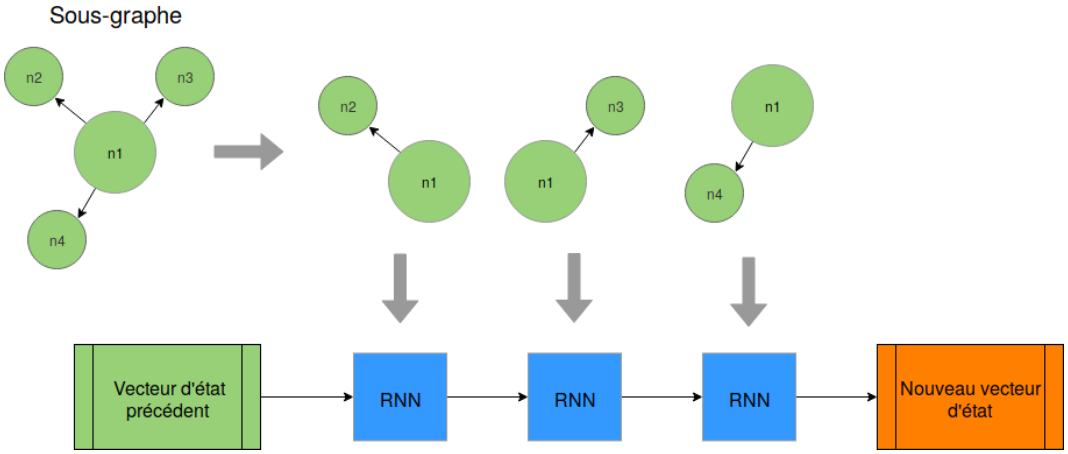

Figure 1.16 – *Schéma représentant un encodeur séquentiel de graphe*

Pour faire l'apprentissage de cet 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.

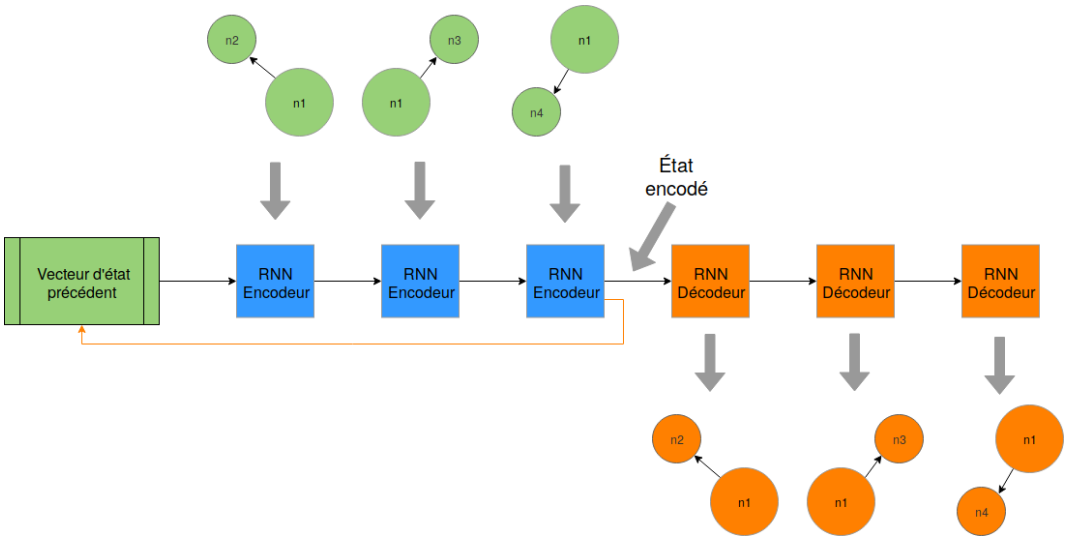

Figure 1.17 – *Schéma de l'apprentissage d'un encodeur séquentiel de graphe*

Les agents feuilles

Comme nous l'avons cité précédemment, les agents feuilles sont les agents responsable de répondre aux intentions de l'utilisateur. Pour faire l'apprentissage par renforcement d'un agent feuille, on utilise 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. On utilise pour cela un réseau de neurones profond qui prend en entrée le graphe encodé de l'agent et il produit pour chaque action la récompense correspondante. Comme le nombre d'actions est variable, il est impossible d'utiliser une architecture de réseau de neurones qui produit une sortie pour chaque action, où chaque sortie est la récompense de l'action qui y correspond. Par conséquent, il est nécessaire de donner au réseau, en addition de l'état encodé, l'action candidate.

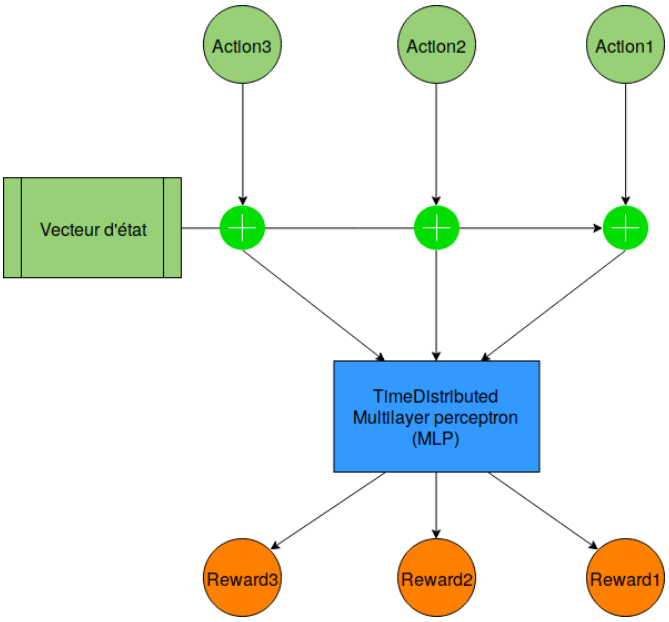

Figure 1.18 – *Schéma du réseau DQN*

L'algorithme utilisé pour l'apprentissage par renforcement est double DQN en rejouant l'expérience. En addition 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 épisode [10] l'agent reprend la mémoire pour entraîner le réseau de neurones.
- Double DQN : la fonction Q est donné par la formule $Q(s,a) = r + \alpha * max_j(Q(s', a_j))$ avec :
  - s : état de l'agent.
  - a : l'action qu'on veut estimer.
  - R : récompense immédiate.
  - α : paramétré de réduction.
  - s' : nouvel état après avoir effectuer 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[10]. La formule devient donc : $Q(s,a) = r + \alpha * Q(s', argmax_{a_j}(s', a_j))$. Cette valeur est donc utilisé pour calculer l'erreur et appliquer l'algorithme de retro-propagation pour l'apprentissage automatique.

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 de DQN est de permettre à l'encodeur de

10. un épisode est un ensemble d'interactions agent-simulateur jusqu'à finir avec un succès ou échec

contrôler quelle partie du graphe encodé à oublié. 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 porte d'oublie rend cette architecture possible.

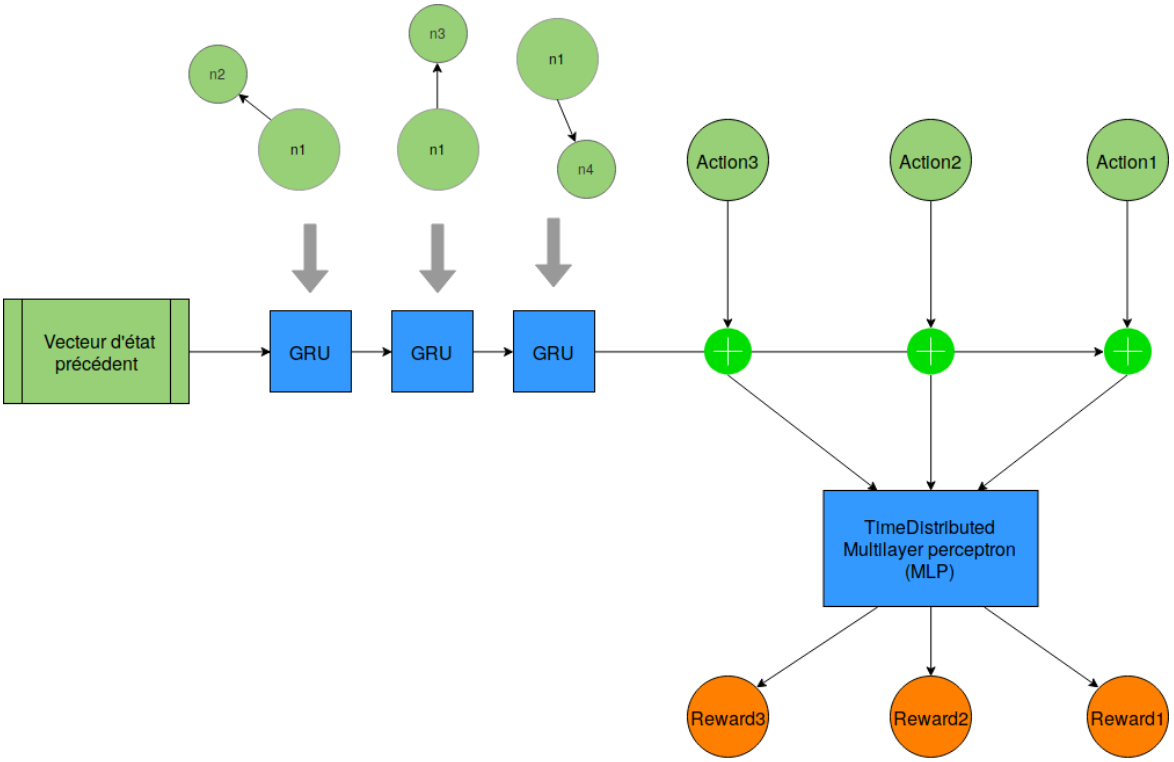

Figure 1.19 – *Schéma du réseau DQN relié avec l'encodeur directement*

L'agent coordinateur

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

1.6 Module de génération du langage naturel

Le modèle utilisé pour la génération du texte est relativement simple. Il s'agit de préparer des modèles de phrases contenant des emplacements à remplir. Chaque action de l'agent lui correspond un ensemble de modèles et chaque paramètre de l'action lui 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, de même, un modèle de phrase est choisit aléatoirement. Enfin, les emplacements vides sont remplis avec les expressions des paramètres. La figure 1.20 illustre un exemple de la génération de texte en utilisant les modèles de phrases.

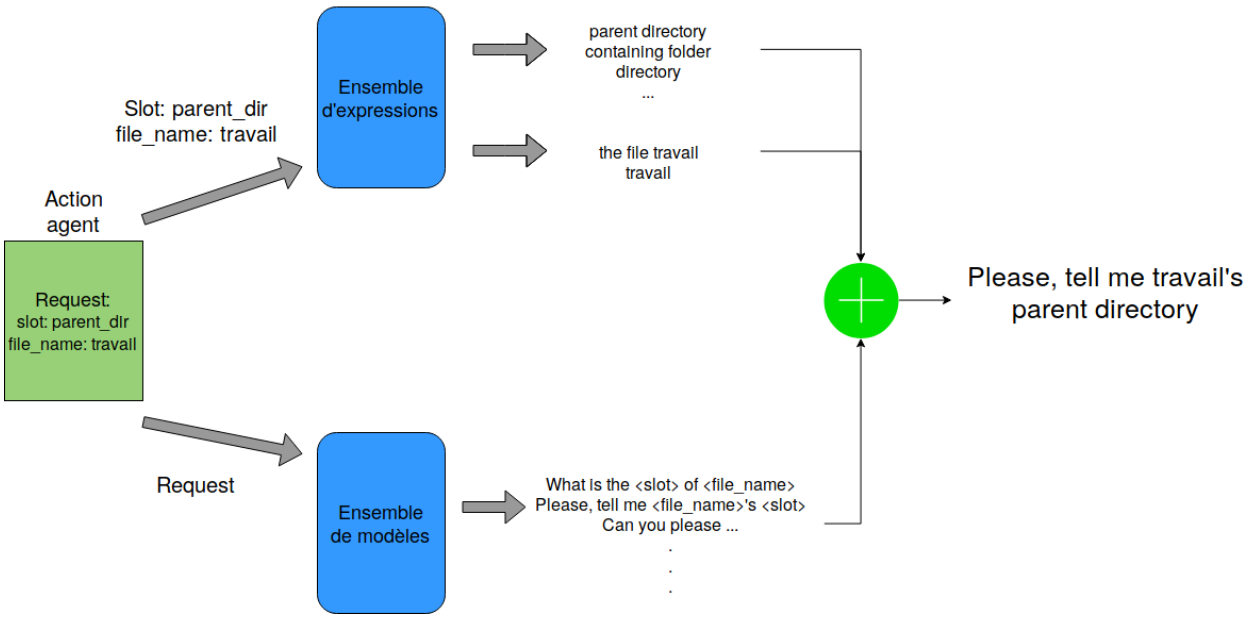

Figure 1.20 – *Schéma de fonctionnement du générateur de texte*

1.7 Conclusion

À travers 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, due à 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 ainsi que leurs améliorations et les comparer avec des standards existants.

Table des figures

Bibliographie

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

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

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

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.