Department of Information Engineering and Computer Science

Bachelor's Degree in
Computer Science

FINAL DISSERTATION

Supervised learning vs Reinforcement learning in task completion dialogues

Supervisor

Prof. Giuseppe Riccardi

Co-Supervisor

Dr. Evgeny A. Stepanov

Student

Guidolin Davide

Academic year 2020/2021

Acknowledgements

Thanks to Giuliano Tortoreto and Giovanni De Toni who helped me during my internship at VUI and during the development of my thesis.

Contents

Abstract

Task-completion dialogue is a complex problem in which the system has to complete dialogues with a real user to satisfy the user's requests and to perform some tasks asked by the user, e.g. reserving a restaurant table or booking a hotel room. This is a complex problem because firstly the system needs to translate the user sentences from natural language to a representation that the machine can understand, secondly, the number of sentences that the user can utter is infinite and the system has to understand what the user wants, thirdly the user or the speech-to-text module could generate some errors in the sentences and the system has to deal with these errors.

In this thesis, we compare two of the main methods used to train an agent to solve the problem of task-completion dialogues: supervised learning and reinforcement learning. In supervised learning, the agent uses a set of training dialogues to learn how to reply to the user while in reinforcement learning the agent learns by trial and error and receives a positive or negative reward at the end of each dialogue. For the supervised learning part, we used an agent developed by the company VUI inc. while for the reinforcement learning part we used some agents trained using the *PyDial*[6] framework and we used two popular algorithms: GP-SARSA[1] and Deep Q networks[3]. To train the supervised learning agent and to evaluate the various agents we used the Schema-Guided Dialogue (SGD) dataset [4] that consists of over 20k annotated multi-domain, task-oriented conversations between a human and a virtual assistant spanning 20 domains. For our experiments, we used four domains: Banks, Buses, Calendar and Restaurants. In the reinforcement learning part, since the agent learns by trial and error and it does not use training dialogues, we used a user simulator that simulates the behavior of a real user and permits to train the agents without the need for a real user that interacts with the system. This user simulator has been developed during my internship at VUI and generates dialogues similar to the ones found in SGD. To train the reinforcement learning agents we used *PyDial*, a framework created to train reinforcement learning dialogue agents that contains the implementation of popular algorithms like GP-SARSA and DQN. We integrated our user simulator into this framework and we created a goal generator, to generate user goals, and an action evaluator, to evaluate the agents. We also modified the belief tracker to make it work with SGD actions. As evaluation metrics, we used the success rate, that was already implemented in PyDial and measures the percentage of successful dialogues, and the action accuracy that was used in the VUI agent, and measures the accuracy of the agent's actions against a test set. The success rate is used to compare the two reinforcement learning algorithms, while the action accuracy is used to compare reinforcement learning and supervised learning. We also trained some agents using a mixed approach, that is reinforcement learning with some training dialogues used in the training phase. With this method, the agent gets a reward or penalty after every action and not only after the entire dialogue.

We show that GP-SARSA, with an average success rate of 88%, is better than DQN that has an average success rate of 53%. The action accuracy is similar in the two algorithms with an average of 24% in both GP-SARSA and DQN. We show that supervised learning, with an average action accuracy of 79%, is better than reinforcement learning. In the end, we show that using training dialogues with the reinforcement learning agents raises the average action accuracy to 31% for GP-SARSA and 40% for DQN.

Introduction

In the first section, we will start by providing a general overview of how a dialogue system works, which are the main modules, and how the information flow from the user to the system and back to the user. In the second section, we will explore the dataset and we will report some statistics like the number of training and test dialogues for the used domains, the number of slots, and the number of values for every slot. In the third section, we will show how the user simulator works and how it decides which action to take based on the system response. Then, in the fourth section, we will explore the PyDial framework, we will show how we used our user simulator with it, how we implemented a custom goal generator, how we modified the belief tracker and how we implemented the class to evaluate the system performances. In the fifth section, we will talk about the evaluation metrics used to measure the performances, we will explain the *success rate* used in the PyDial framework and the *action accuracy* used in VUI. Finally, in the sixth section, we will report the results, we will show that for reinforcement learning GP-SARSA is better than DQN and we will show that supervised learning will perform better than reinforcement learning and better than the mixed approach.

1 Dialogue systems

1.1 General architecture of a dialogue system

A dialogue system is composed of three main modules: the natural language understanding module (NLU), the dialogue management module (DM) and the natural language generation module (NLG). If the user's input comes as a sound wave we can have an additional speech-to-text or automatic speech recognition (ASR) module that converts that sound wave into text and if the system is supposed to reply with an audio signal we can have a text-to-speech (TTS) module that converts the system response to a sound wave. (Figure 1.1)

When the user speaks the sound wave is converted to text by the ASR module. Here the first problems arise because the ASR module could generate some errors during the conversion and the resulting text may not match what the user said. Then the user's utterance in text format goes through the NLU module in which the meaningful information is extracted and converted into a dialogue action. In the DM module a belief state, that contains all the information needed to represent the current state of the dialogue, is created from the dialogue actions. The belief state is then converted to numbers so that it can be used as input for the machine learning algorithm that selects the next action to take. To get the information needed to reply to the user the DM module communicates with a knowledge base. When the agent has selected the next action to take it updates the belief state and the new action is sent to the NLG module that converts it to an understandable utterance. Finally, the system utterance is sent to the TTS module that generates audio. As the ASR module, the NLU, NLG, and TTS modules could introduce errors before the final output.

In our experiments, we will not use the ASR, NLU, NLG, and ASR modules since we will work only with dialogue actions.

1.2 Natural language understanding and dialogue actions

The NLU module takes as input a text sentence, extracts the user's intent and the useful slots, and outputs a dialogue action. In some cases, the dialogue action is a direct substitute for intent and in others, a mapping is made from the intent and slot labels to the action. The intents categorize the user's intentions while the slots are the useful information that is present in the sentence.

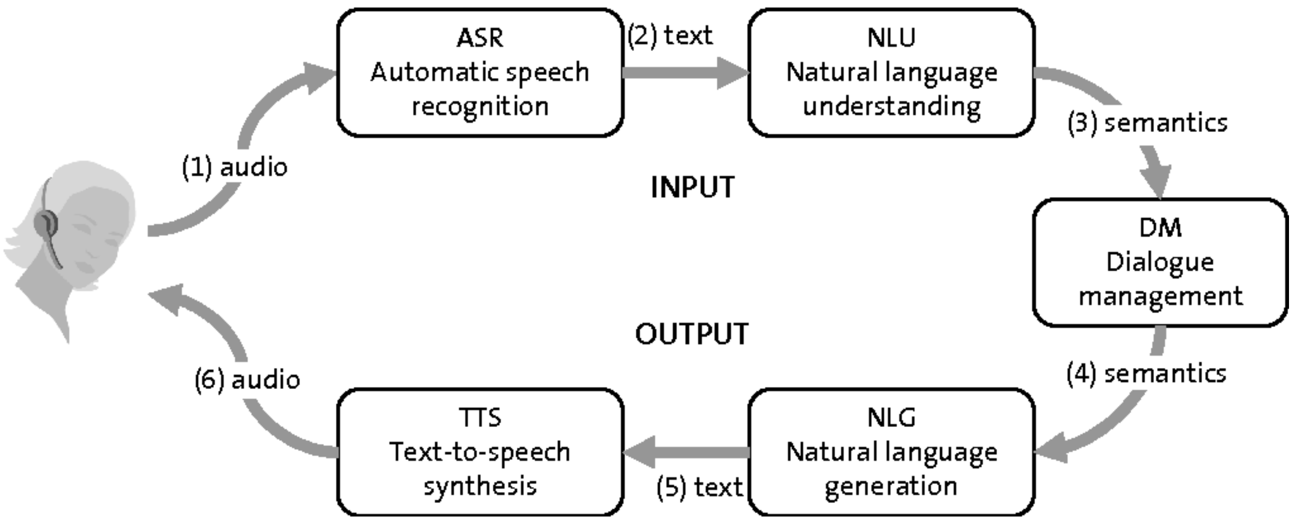

Figure 1.1: Architecture of a spoken dialogue system

The NLU module can be implemented with classical machine learning approaches, such as SVM, K-nearest neighbors and random forest but in recent years, with the success of deep learning in other areas, neural networks, especially RNNs, started to be widely used for this task. [7]

1.3 Belief tracker and belief state

The dialogue action needs to be converted to a representation that can be used as input for the machine learning algorithm. To do so, in the DM module there is a belief tracker that uses the dialogue action to update the belief state. The belief state is used to track the state of the dialogue, i.e. the slots that the user requested, the constraints that the user informed, and other information like what the user did in the current turn, e.g. issued a constraint, requested a slot, asked about a specific venue by his name, ended the dialogue[2]. A belief state is usually a dictionary in which the keys are the information just listed and the values are the probabilities that the dialogue action contains that information. We use probabilities because the NLU module is not perfect so along with the dialogue action it outputs a probability to tell how much confidence it has that the extracted information is correct. The belief state is then used as input for the machine learning algorithm that uses it to predict the next action to take.

In our experiments, we will work only with dialogue actions so that we will not have the errors introduced by the NLU and NLG modules and the comparison between the different algorithms will be easier.

1.4 Ontology

The ontology contains all the information needed by the DM to understand the incoming dialogue action and reply with a new one, e.g. it contains all the possible slot values for every slot and all the possible user intents for a domain. For example, the ontology for a Restaurant domain could contain all the restaurants' names, addresses, and phone numbers of these restaurants, along with the intents that the user could perform, e.g. find a restaurant or reserve a table. In our experiments we don't care about the slot values used by the system, however, in a real system, the ontology should contain all the information that the users could require.

2 Dataset

2.1 Description of the dataset

For this work, we used the Schema-Guided Dialogue (SGD) dataset [4] that consists of over 20k multi-domain, task-oriented conversations between a human and a virtual assistant, and every turn of every dialogue is annotated with useful information like the dialogue actions and slots. The SGD dialogues span across 20 domains, however in our work, we used 4 domains: Banks, Buses, Calendar, Restaurants. In table 2.1 we report the total number of dialogues for the used domains, however, some of those dialogues are multi-domain, e.g. the dialogue starts with the Restaurants domain and then changes to the Buses domain, but for our experiments, we decided to use only single domain dialogues. In table 2.2 we report the number of single-domain dialogues along with the number of slots for every domain.

Domain	**Number of dialogues**
Banks	727
Buses	2280
Calendar	1602
Restaurants	2419

Table 2.1: Total number of dialogues for the used domains

Domain	**Number of dialogues**	**Number of slots**
Banks	206	5
Buses	310	18
Calendar	169	6
Restaurants	367	11

Table 2.2: Number of single-domain dialogues and number of slots for the used domains

In appendix A we reported the full list of slots along with the number of values for every slot. Since we used only the single domain dialogues, some slot values could be missing in our experiments.

In SGD there is a set of dialogue actions that are used in the dialogues annotations. We used the same set of user actions in our user simulator. The actions are:

- **SYSTEM ACTIONS**:
  - **INFORM** - Inform the value for a slot to the user.
  - **REQUEST** - Request the value of a slot from the user.
  - **CONFIRM** - Confirm the value of a slot.
  - **OFFER** - Offer a certain value for a slot to the user.
  - **NOTIFY_[SUCCESS/FAILURE]** - Inform the user that their request was successful/-failed.
  - **OFFER_INTENT** - Offer a new intent to the user.
  - **REQ_MORE** - Asking the user if they need anything else.

- **GOODBYE** - End the dialogue.

- **USER ACTIONS**:
  - **INFORM_INTENT** - Express the desire to perform a certain task to the system.
  - **[NEGATE/AFFIRM]_INTENT** - Negate/Agree to the intent which has been offered by the system.
  - **INFORM** - Inform the value of a slot to the system.
  - **REQUEST** - Request the value of a slot from the system.
  - **AFFIRM** - Agree to the system's proposition.
  - **NEGATE** - Deny the system's proposal.
  - **SELECT** - Select a result being offered by the system.
  - **REQUEST_ALTS** - Ask for more results besides the ones offered by the system.
  - **THANK_YOU** - Thank the system.
  - **GOODBYE** - End the dialogue.

Note that in a single turn there can be more than one dialogue action, this is because in SGD there is only one slot for every action, so if in the same turn the system request two slots we will have two actions `REQUEST(slot1)` and `REQUEST(slot2)`.

Now we report an example of an SGD dialogue and the same dialogue but in the dialogue actions format:

```
======================== Dialog: 1_00000 ========================
SERVICES: Restaurants_1

USER: I am feeling hungry so I would like to find a place to eat.
SYSTEM: Do you have a specific which you want the eating place to be
   located at?
USER: I would like for it to be in San Jose.
SYSTEM: Is there a specific cuisine type you enjoy, such as Mexican,
   Italian or something else?
USER: I usually like eating the American type of food.
SYSTEM: I see that at 71 Saint Peter there is a good restaurant which
   is in San Jose.
USER: Can you give me the address of this restaurant.
SYSTEM: If you want to go to this restaurant you can find it at 71
   North San Pedro Street.
USER: Can you give me the phone number that I can contact them with?
SYSTEM: If you want to phone them you can at 408-971-8523.
USER: Is there some other restaurant which you can suggest?
SYSTEM: How would you like Bazille restaurant which is situated in San
    Jose.
USER: Do you have another restaurant matching my needs? For example a
   restaurant which is economical and is located in Palo Alto.
SYSTEM: I see that 7 restaurants suit to what you requested. Bird Dog
   seems as a good restaurant and is located in Palo Alto.
USER: Alright, that seems good. I would like to make a booking at this
    restaurant.
SYSTEM: For which time do you want the booking to be?
USER: I will be eating there at 11:30 am so make it for then.
SYSTEM: Can you please confirm that you want to book a table for 2 at
   11:30 am at the Bird Dog restaurant in Palo Alto for today.
USER: That suits me well. Can you tell me if they feature live music?
```

```
SYSTEM: Your booking has been made without errors, but unfortunately
   they do not have live music.
USER: Will I be able to find liquor there? Can you give me the address
    of their location?
SYSTEM: The restaurant is located at 420 Ramona Street. Unfortunately
   they do not serve alcohol at the restaurant.
USER: I appreciate it very much. That would be all.
SYSTEM: Have a good time!
```

Same dialogue with dialogue actions:

```
=============================== Dialog:  1_00000 ================================
SERVICES:  Restaurants_1
USER:
- INFORM_INTENT(intent=['FindRestaurants'])
SYSTEM:
- REQUEST(city)
USER:
- INFORM(city=['San Jose'])
SYSTEM:
- REQUEST(cuisine=['Mexican', 'Italian'])
USER:
- INFORM(cuisine=['American'])
SYSTEM:
- OFFER(restaurant_name=['71 Saint Peter'])
- OFFER(city=['San Jose'])
USER:
- REQUEST(street_address)
SYSTEM:
- INFORM(street_address=['71 North San Pedro Street'])
USER:
- REQUEST(phone_number)
SYSTEM:
- INFORM(phone_number=['408-971-8523'])
USER:
- REQUEST_ALTS
SYSTEM:
- OFFER(restaurant_name=['Bazille'])
- OFFER(city=['San Jose'])
USER:
- INFORM(price_range=['moderate'])
- INFORM(city=['Palo Alto'])
- REQUEST_ALTS
SYSTEM:
- OFFER(restaurant_name=['Bird Dog'])
- OFFER(city=['Palo Alto'])
- INFORM_COUNT(count=['7'])
USER:
- INFORM_INTENT(intent=['ReserveRestaurant'])
- SELECT
SYSTEM:
- REQUEST(time)
USER:
- INFORM(time=['11:30 am'])
```

```
SYSTEM:
- CONFIRM(restaurant_name=['Bird Dog'])
- CONFIRM(city=['Palo Alto'])
- CONFIRM(time=['11:30 am'])
- CONFIRM(party_size=['2'])
- CONFIRM(date=['today'])
USER:
- REQUEST(has_live_music)
- AFFIRM
SYSTEM:
- INFORM(has_live_music=['False'])
- NOTIFY_SUCCESS
USER:
- REQUEST(serves_alcohol)
- REQUEST(street_address)
SYSTEM:
- INFORM(street_address=['420 Ramona Street'])
- INFORM(serves_alcohol=['False'])
USER:
- THANK_YOU
- GOODBYE
SYSTEM:
- GOODBYE
```

2.2 Training and test sets

To create the training set we used 85% of the dialogues, the other 15% are used as the test set. The training set is used only in the supervised learning and in the supervised reinforcement learning approaches while the test set is used to test the action level accuracy of every agent. In table 2.3 we report the number of training and test dialogues for every domain.

Domain	Number of training dialogues	Number of test dialogues
Banks	175	32
Buses	263	47
Calendar	143	26
Restaurants	311	56

Table 2.3: Number single-domain dialogues for the used domains

3 User simulator

The user simulator is very important for the training of a reinforcement learning agent, in fact, since we can't use training dialogues because the agent learns by trial and error, we would need a real user that interacts with the system and this would cost money and a lot of time. To solve this problem it's a common practice to develop a user simulator that acts like a real user but it interacts with the system in a faster way.

During my internship at VUI, I developed an agenda-based user simulator that had to generate dialogues similar to the ones found in the SGD dataset. A user simulator is composed of two main modules: the dialogue manager (DM) that selects the action that the user should take and the natural language generator (NLG) that translates the chosen dialogue action into natural language. However, since we will work only with dialogue actions we did not implement the NLG module. In an agenda-based user simulator, the DM module takes the system action and chooses the next user action using a set of hand-crafted rules and probabilities based on a goal that is selected at the beginning of every dialogue [5].

3.1 User goal

The user goal contains information that the user simulator has to obtain from the system and the constraints that it can inform. Usually a goal contains a set of request slots that the system has to fill and a set of inform slots that the user simulator can provide to the system, however, in our simulator we also have a set of intents that the user has to perform to complete the dialogue. For example, a goal for the restaurants domain could have as request slots `street_address, phone_number, restaurant_name`, as inform slots `city, price_range` and as user intents `FindRestaurant, ReserveTable`. The user simulator will end the dialogue when every slot will be filled with a value issued by the system and when the two intents are finished. An intent can be considered finished in two ways: if the intent has request slots associated to it then the intent will finish when all the associated request slots will be filled, e.g. the `FindRestaurant` intent has the request slots `street_address, phone_number, restaurant_name` and it will finish when all these slots will be filled. If the intent does not have request slots associated to it then the intent will finish when the system will notify the user simulator that the user request has been completed, e.g. the `ReserveTable` intent will finish when the system will reply with a `notify()` (`NOTIFY_SUCCESS` or `NOTIFY_FAILURE`) action, meaning that it tried to perform that task, regardless of the result. In every goal there is a request slot that is always present and is the `name` slot. This slot is needed because it was used by the PyDial researchers and since there were a lot of classes that checked the presence of this slot we decided to add it to the goals instead of changing the PyDial code. In some domains it was already present, like in the restaurants domain, in the other domain we added it with fake values.

3.2 Basic working of the user simulator

The input to our user simulator is the agent's dialogue action, however, to make the user simulator usable with different agents, we added a layer that converts the agent's action to an SGD action. In this way, we only need to specify a mapping between the set of agent actions and the SGD actions. The same applies to the user simulator output, it outputs an SGD action that will be converted to an action understandable by the agent. For example, if we have an agent that uses an action for every slot, e.g. `request_cuisine`, `request_city`, etc. we can convert these actions to `REQUEST(cuisine)` and `REQUEST(city)` so that the user simulator can understand them. After the conversion, the user simulator will decide the next action to take by following a set of rules. The first thing it does is checking if the dialogue has just started and we are at turn 0, if so it replies with an `INFORM_INTENT` action to express what he wants. After the first turn, the user response depends on the system action. If the system requests a slot the user will reply with an `INFORM` action if the slot is present in the inform slots in the user goal, otherwise, it will reply with a `NEGATE` action. The same applies for a `CONFIRM` user action: if the user already informed the slot that the system asks confirmation for then it will reply with an `AFFIRM` action, otherwise with a `NEGATE` action. If the system offers a slot then the user will save it in a copy of the user goal, when all the slots in this copy are filled then the user will select the offer with a `SELECT` action. If the system offers an intent the user will accept it (`AFFIRM_INTENT` action) only if the current intent is finished and the offered intent is not finished yet, otherwise, it will refuse it (`NEGATE_INTENT` action). The `REQUEST` action will be used after a `REQ_MORE` system action if some slots in the request slots are not filled yet. Finally, the `GOODBYE` action will be used when the user goal is completed.

These are the basic rules, then, to make the user simulator more realistic, we added some variability to the responses, for example by replying with a wrong slot or with a `REQUEST` action instead of a `NEGATE` action.

4 PyDial framework

The PyDial framework is a multi-domain statistical spoken dialogue system toolkit that was originally developed by the Dialogue Systems Group at Cambridge University Engineering Department (CUED) and it is actively used for research by the Dialogue Systems and Machine Learning Group at the Heinrich-Heine University, Düsseldorf. This framework is easily extensible with custom modules and permits to customize every agent by changing its parameters in a configuration file.

In our work, we integrated our user simulator with PyDial, and we created a custom goal generator to select the user goals. We modified the belief tracker to make it usable with the SGD actions and we developed an action evaluator to measure the action accuracy of the agents.

4.1 User simulator

We used our user simulator to train the reinforcement learning agents. To use it we had to extend the PyDial class `UMSimulator` that provided two methods to override: the `receive()` method used to processes the new system action and the `respond()` method used to get the user's response action. In the `receive()` method the only thing we do is saving the system action in an array. In the `respond()` method we pop the last system action from the array, we convert it to an SGD action, we select the next action that the user has to take and we convert the user action from the SGD format to the system format. To use our new user simulator class with PyDial we only need to specify the class in the configuration file.

4.2 Goal generator

The user goals are generated by a goal generator. The PyDial goal generator, however, generates goals by selecting a set of random constraints and requests but we want goals that are as similar as possible to the goals for the SGD dialogues so we decided to implement a goal generator. To do this PyDial provided the `GoalGenerator` class that we extended, as we did with the user simulator. The only method to override is the `init_goal()` method that has to return a `UMGoal` that contains the user constraints and requests. In our goal generator, the goals are loaded from a goal file and simply converted to `UMGoal`. The goal file has previously been generated with a script that goes through every dialogue in the SGD dataset and extracts the slots informed by the user that become the constraints or inform slots in the user goal, the slots requested by the user (request slots), and the intents expressed by the user.

4.3 Belief tracker

PyDial has not been designed to use SGD dialogue actions so the belief tracker was not able to recognize the SGD user actions, except for some common actions like `request` or `inform`. PyDial offered the possibility to use a custom tracker by inheriting the `BeliefTracker` class, however, since we only needed to change the action names, we decided to inherit from an already implemented tracker, the `RuleBasedTracker`, and re-implement only the methods that contained hard-coded action names and the methods that parsed the dialogue actions. It's important to note that we only modified the parsing and tracking of the user actions. Regarding the agent actions, we decided to use the ones that were present in PyDial since almost all of them matched with the SGD agent actions. We modified the `SummaryAction` class to remove unnecessary system actions and add the missing ones, like `OFFER` and `NOTIFY`.

4.4 Action evaluator

To evaluate the action accuracy we had to implement another module because PyDial only measures the success rate. We created a class that loads the agent and the test set from the files passed as command-line arguments. For every test dialogue, the user simulator will follow the user role in the test dialogue and the agent responses will be evaluated against the agent actions in the test dialogue. Finally, the action accuracy will be calculated and the results will be printed.

5 Evaluation metrics

To compare the different reinforcement learning approaches we used the dialogue accuracy (or success rate) that was used in the PyDial framework and the action accuracy that was used in VUI.

5.1 Success rate

The success rate measures the percentage of successful dialogues. A dialogue is successful when all the request slots in the user goal are filled with a value. Here we only care that the request slots are filled, in a real system we should check that the slots values used by the agent are correct, e.g. the restaurant's street address should be the correct address for that restaurant. To calculate the success rate we use the following formula:

$$successRate = 100 \cdot \frac{\sum_{i=1}^{N} \begin{cases} 1 & \text{if } \forall \text{ request slot } s \in \text{UserGoal}[i],\ s.value \neq \text{None} \\ 0 & \text{otherwise} \end{cases}}{N}$$

where N is the number of dialogues and $UserGoal[i]$ is the user goal of the ith dialogue.

The success rate is only used to compare the reinforcement learning agents because in supervised learning we don't have user goals and so we can't decide when a dialogue is completed.

5.2 Action accuracy

The action accuracy measures the percentage of correct actions taken by the agent against a test set.

The action accuracy takes into account only the actions and not the slots. The system action is correct if the same action is present in the test turn, i.e.

$$\exists \text{ action} \in \text{test\_actions} \mid \text{action} == \text{agent\_action}$$

where $test\_actions$ are the actions in the current turn of the test dialogue and $agent\_action$ is the action taken by the agent.

6 Experiments

In this section we will describe the experiments that we did and we will report the results. In appendix B we reported also an example of a successful dialogue and a fail dialogue.

6.1 Supervised learning

To train the supervised learning agents we used the VUI tools. First of all we had to convert the training dialogues to a format understandable by the VUI tools. Then we had to define the agent actions, so, in this case, the agent used an action for every action-slot pair, e.g. the action `REQUEST(city)` becomes `action_request_city`, etc.

In the training phase we could modify the number of epochs and the augmentation factor that is used to increase the number of training samples. In the results you will find a different value for the augmentation factor for the various domains, this is because after a certain value the system crashed since we had too many training samples in memory, so the augmentation factor depends on the number of available training dialogues. We used 20 as the number of epochs for the four domains.

In table 6.1 we report the results of the four agents on training and testing.

Domain	Augmentation factor	Action accuracy on training	Action accuracy on testing
Banks	6	92%	81%
Buses	5	91%	83%
Calendar	9	91%	81%
Restaurants	5	91%	72%

Table 6.1: Supervised learning results

6.2 Reinforcement learning

In reinforcement learning the agent learns by receiving a reward or a penalty at the end of every dialogue. In the following section, along with the results, we will report the values of the reward (`success reward`) for a correct dialogue and the penalty (`fail penalty`) for a wrong dialogue that we used for the training of every agent.

In every agent we penalize the number of turns to encourage the agent to complete the dialogue as fast as possible. For every turn the agent will have a -1 penalty that will be added to the final reward of the dialogue. During training we set a number of maximum turns to avoid too long dialogues.

In the results we report also the success rate on testing. Note that to calculate this value we don't use test dialogues, but we use the user simulator to evaluate the agent, so the accuracy on testing should be similar to the accuracy on training. The test dialogues are used only for action accuracy.

6.2.1 GP-SARSA

GP-SARSA is famous for improving the learning rate and reaching convergence faster than other algorithms[1]. We noticed this behavior in our experiments, in fact, we trained these agents only for 1000 dialogues and then we moved to DQN since the results were good. In the end, we trained GP-SARSA for 5000 dialogues and the success rate is very good.

We report the results of the best GP-SARSA agents for the four domains, along with the success reward and the fail penalty used to train them. We report also the average number of turns needed by the agent to complete a dialogue and the plot of the reward and success rate values. In the plots every iteration corresponds to 100 dialogues.

Banks domain:

Domain	Success reward	Fail penalty	Success rate on training	Success rate on testing	Action accuracy	Avg. number of turns
Banks	10	5	90%	81%	27%	9

Table 6.2: Best agent for the Banks domain

Figure 6.1: Best agent reward and success rate for the Banks domain

Buses domain:

Domain	Success reward	Fail penalty	Success rate on training	Success rate on testing	Action accuracy	Avg. number of turns
Buses	10	15	100%	98%	27%	11

Table 6.3: Best agent for the Buses domain

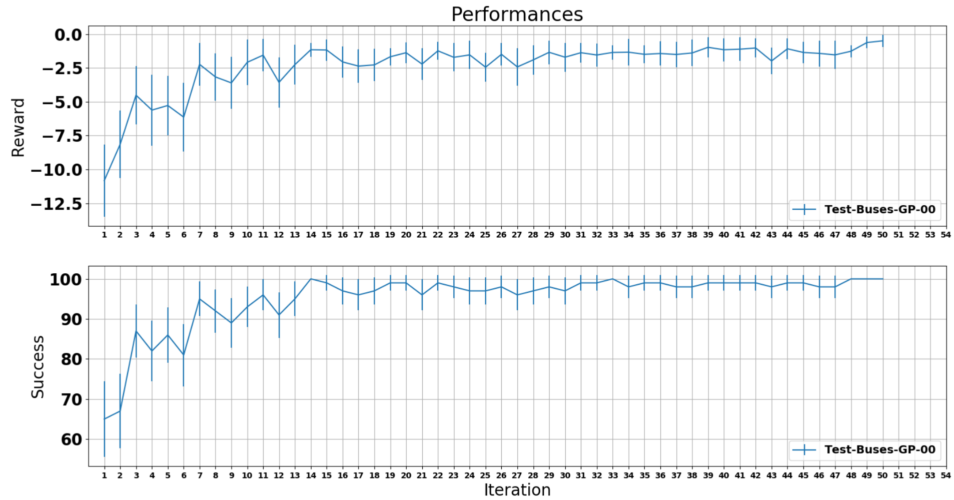

Figure 6.2: Best agent reward and success rate for the Buses domain

Calendar domain:

Domain	Success reward	Fail penalty	Success rate on training	Success rate on testing	Action accuracy	Avg. number of turns
Calendar	10	15	84%	78%	13%	13

Table 6.4: Best agent for the Calendar domain

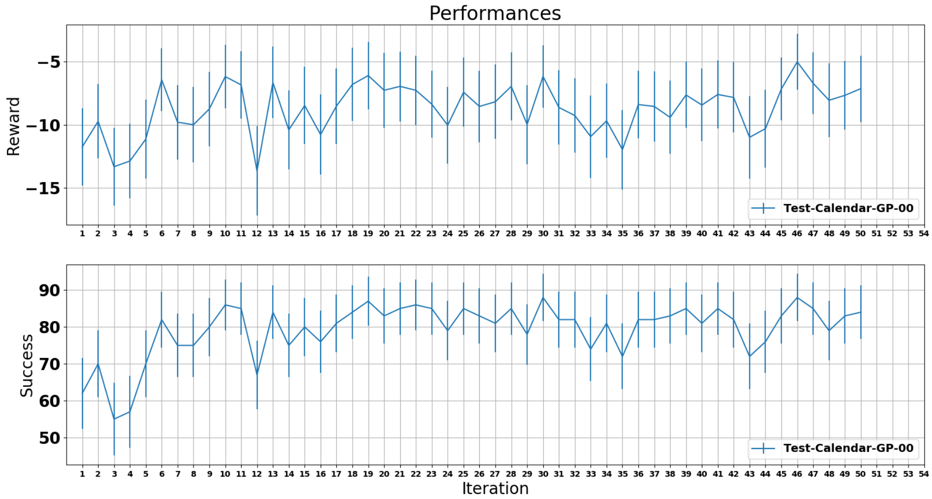

Figure 6.3: Best agent reward and success rate for the Calendar domain

Restaurants domain:

Domain	Success reward	Fail penalty	Success rate on training	Success rate on testing	Action accuracy	Avg. number of turns
Restaurants	10	15	93%	94%	29%	10

Table 6.5: Best agent for the Restaurants domain

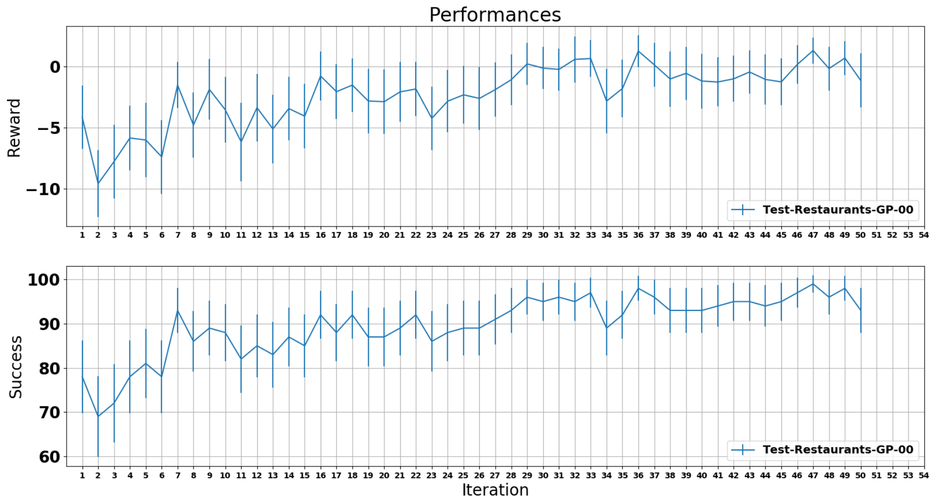

Figure 6.4: Best agent reward and success rate for the Restaurants domain

6.2.2 DQN

Deep-Q-Networks were the hardest agents to train. We started by using the same configurations of the best GP-SARSA agents, however, the agents were not able to learn and, even after 12000 dialogues the success rate was 0%. In figure 6.5, that shows the plot of the reward value and the success rate, we can see that these values have some spikes but in the end the success rate comes back to zero.

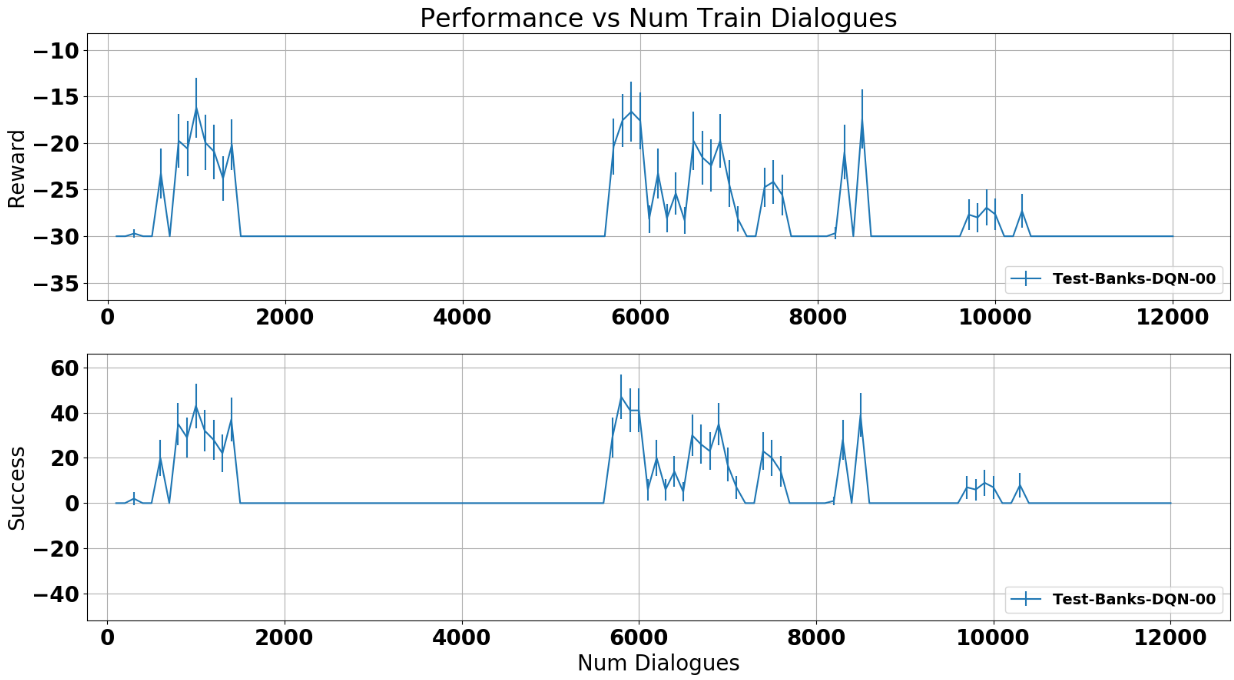

Figure 6.5: The agent is not able to learn after 12000 dialogues

We started to investigate the dialogues that failed and we noticed that sometimes the agent got stuck repeating the same actions until the end of the dialogue and this could be caused by the belief state. If the belief state does not contain enough information, the agent can't distinguish between different statuses and it will end up repeating the same action. To solve this problem we decided to modify the belief state and the belief state tracker by adding new information that could help the system. Since it was repeating the same actions we added the last system action, the last user action, and a counter that tracked how many times the system repeated the same action. With this new information the system was able to learn. Finally, we changed the number of maximum turns for a dialogue from 25 to 15 to force the agent to finish the dialogues before 15 turns.

In the following tables we report the results of the best DQN agents for the four domains, along with the success reward and the fail penalty used to train them. We report also the average number of turns needed by the agent to complete a dialogue and the plot of the reward and success rate values. In the plots every iteration corresponds to 100 dialogues.

Banks domain:

Domain	Success reward	Fail penalty	Success rate on training	Success rate on testing	Action accuracy	Avg. number of turns
Banks	10	0	50%	50%	26%	12

Table 6.6: Best agent for the Banks domain

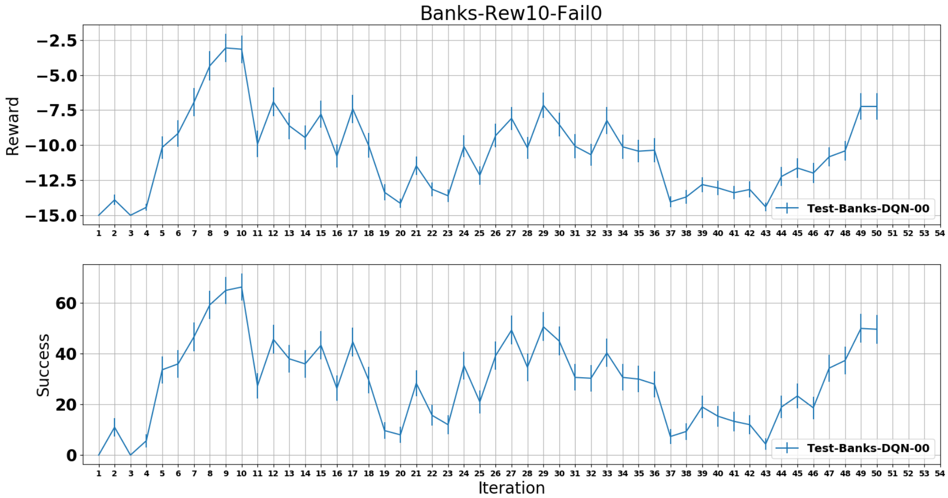

Figure 6.6: Best agent reward and success rate for the Banks domain

Buses domain:

Domain	Success reward	Fail penalty	Success rate on training	Success rate on testing	Action accuracy	Avg. number of turns
Buses	20	0	52%	52%	28%	14

Table 6.7: Best agent for the Buses domain

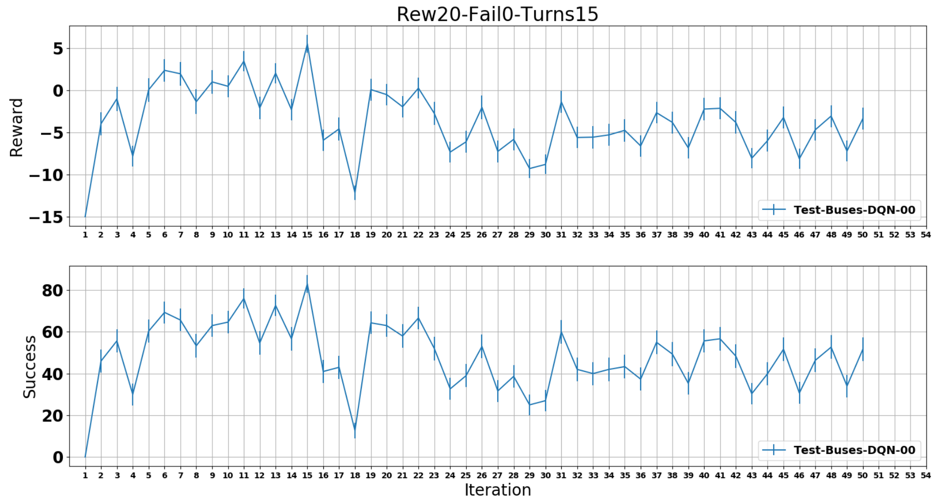

Figure 6.7: Best agent reward and success rate for the Buses domain

Calendar domain:

Domain	Success reward	Fail penalty	Success rate on training	Success rate on testing	Action accuracy	Avg. number of turns
Calendar	10	0	48%	47%	15%	12

Table 6.8: Best agent for the Calendar domain

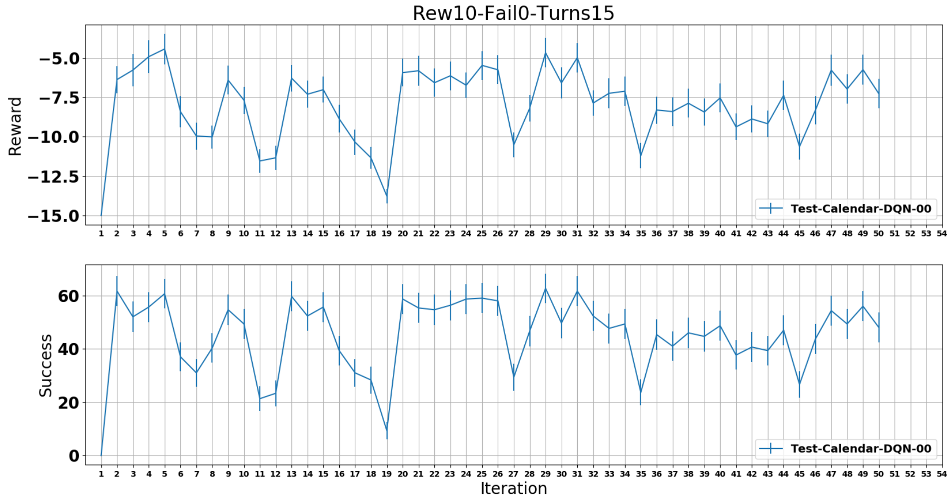

Figure 6.8: Best agent reward and success rate for the Calendar domain

Restaurants domain:

Domain	Success reward	Fail penalty	Success rate on training	Success rate on testing	Action accuracy	Avg. number of turns
Restaurants	10	0	62%	64%	27%	12

Table 6.9: Best agent for the Restaurants domain

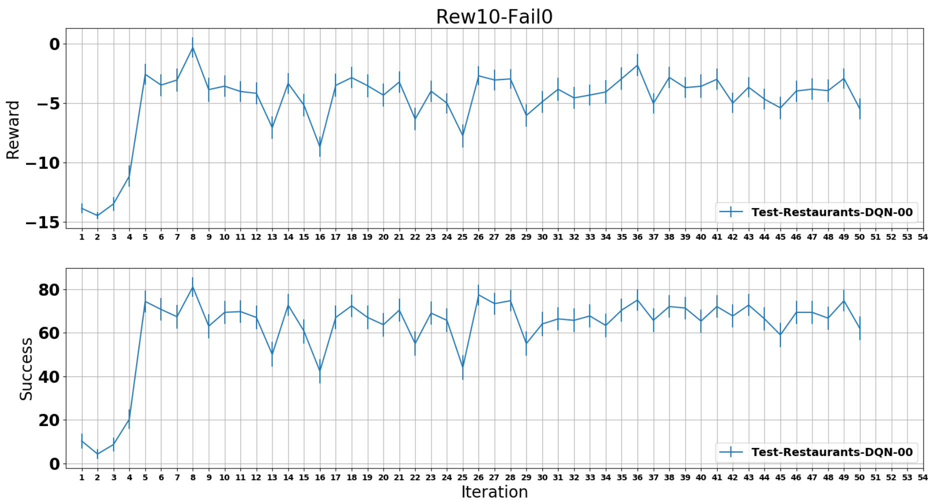

Figure 6.9: Best agent reward and success rate for the Restaurants domain

We can see that the Restaurants agent performs better than the other agents and the reward and success rate curves are the most stable between the four agents. This is because the Restaurants domain had the bigger belief state (see table 6.10) so the agent had more details for every turn with respect to the other agents.

Domain	Belief size (#of values)
Banks	306
Buses	683
Calendar	496
Restaurants	1443

Table 6.10: Belief state size for every domain

6.3 Supervised reinforcement learning

In supervised reinforcement learning we mix supervised learning and reinforcement learning. The base approach is reinforcement learning, however, instead of using only the user simulator we use also training dialogues and for every iteration of 100 dialogues, we use a configurable number of training dialogues. In our experiments, unless specified, we used 70 training dialogues for every iteration.

The goal of this mixed approach is to increase the action accuracy without lowering the success rate so much.

To implement this approach we modified the user simulator so that we can load the user turns of a training dialogue and it will follow the dialogue regardless of the system response. We modified also the reward function. At every turn of a training dialogue the agent action will be compared against the training dialogue agent action and we have 3 cases:

- If both the action and the slot are correct, the reward will be +10
- If only the action is correct and the slot is incorrect, the reward will be +5
- If both the action and the slot are incorrect, the reward will be 0

In this way, when we have a training dialogue, the agent will have the highest reward only if it chooses the correct action and the correct slot in every turn.

With this method, the agent was able to learn the correct action to take for the current belief state instead of focusing only on completing the dialogue successfully.

In the results the success rate on testing will be higher than the success rate on training. This is because during testing we don't have dialogues taken from the training set, but we use only the user simulator, so the agent can still take some wrong actions during the dialogue but in the end complete it successfully.

6.3.1 GP-SARSA

As in pure reinforcement learning, GP-SARSA is very good in completing dialogues successfully, in fact, the success rate on testing is still very high. However, the action accuracy is very low with respect to supervised learning.

Banks domain:

Domain	Success reward	Fail penalty	Success rate on training	Success rate on testing	Action accuracy	Avg. number of turns
Banks	10	5	25%	77%	28%	10

Table 6.11: Best agent for the Banks domain

Figure 6.10: Best agent reward and success rate for the Banks domain

Buses domain:

Domain	Success reward	Fail penalty	Success rate on training	Success rate on testing	Action accuracy	Avg. number of turns
Buses	10	15	30%	95%	32%	10

Table 6.12: Best agent for the Buses domain

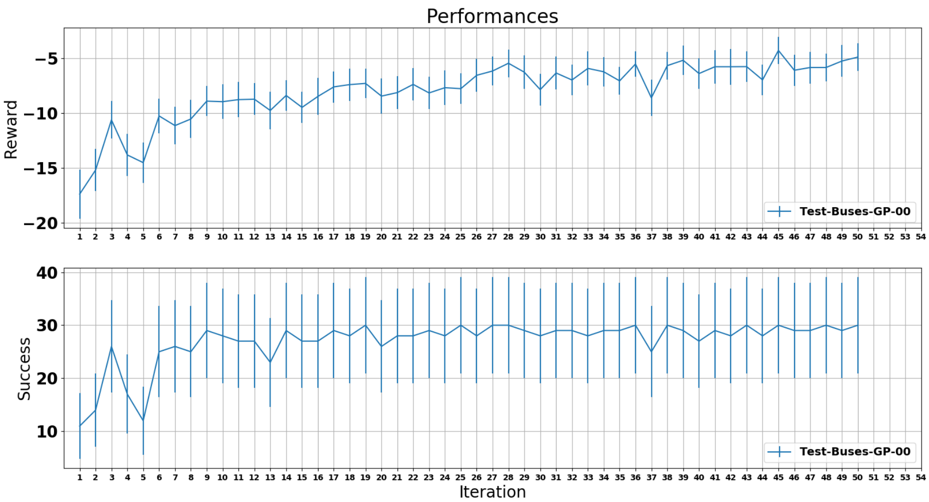

Figure 6.11: Best agent reward and success rate for the Buses domain

Calendar domain:

Domain	Success reward	Fail penalty	Success rate on training	Success rate on testing	Action accuracy	Avg. number of turns
Calendar	10	15	25%	77%	25%	12

Table 6.13: Best agent for the Calendar domain

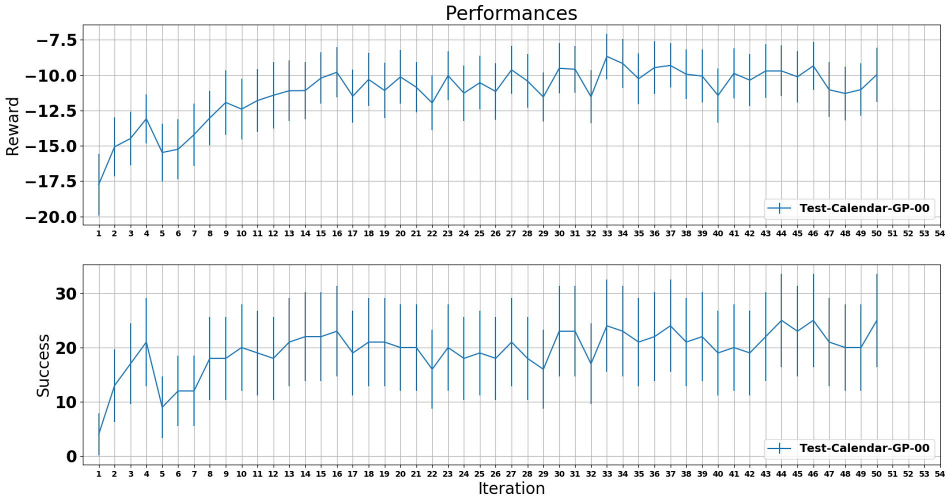

Figure 6.12: Best agent reward and success rate for the Calendar domain

Restaurants domain:

Domain	Success reward	Fail penalty	Success rate on training	Success rate on testing	Action accuracy	Avg. number of turns
Restaurants	10	15	23%	79%	39%	11

Table 6.14: Best agent for the Restaurants domain

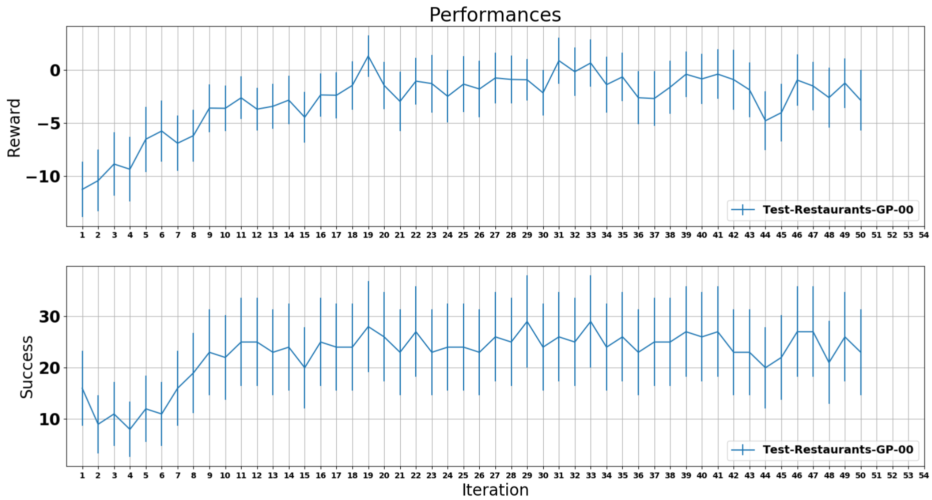

Figure 6.13: Best agent reward and success rate for the Restaurants domain

6.3.2 DQN

DQN has a very low success rate, this is probably because the training dialogues have a big impact on the optimization of the policy. However, the action accuracy is higher than GP-SARSA, but still far from supervised learning.

Banks domain:

Domain	Success reward	Fail penalty	Success rate on training	Success rate on testing	Action accuracy	Avg. number of turns
Banks	15	0	30%	28%	41%	12

Table 6.15: Best agent for the Banks domain

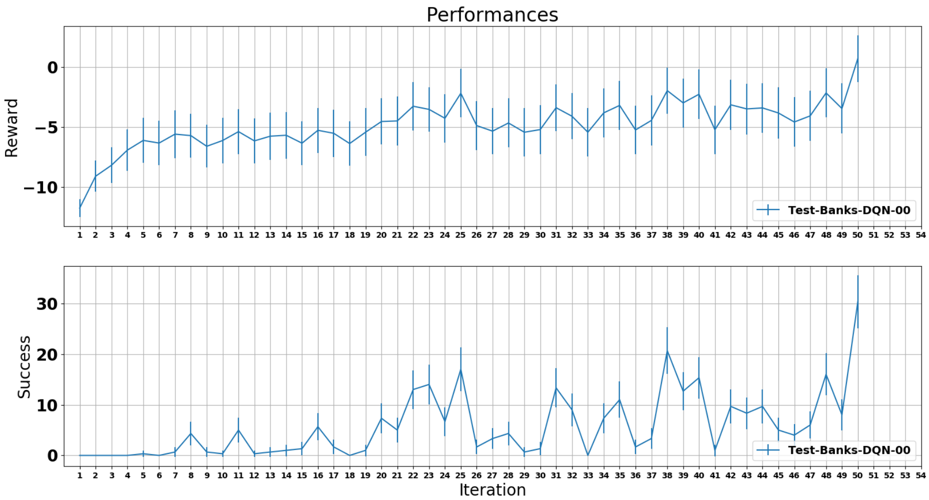

Figure 6.14: Best agent reward and success rate for the Banks domain

Buses domain:

Domain	Success reward	Fail penalty	Success rate on training	Success rate on testing	Action accuracy	Avg. number of turns
Buses	10	0	30%	29%	37%	14

Table 6.16: Best agent for the Buses domain

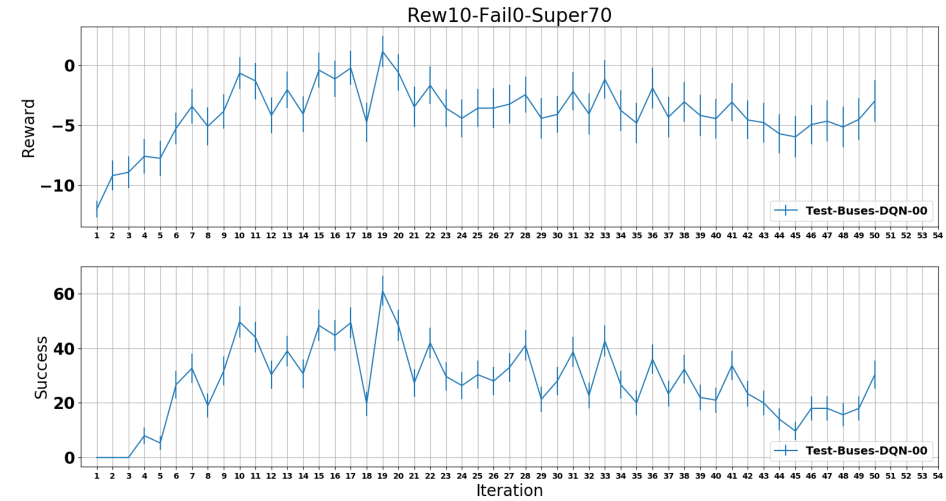

Figure 6.15: Best agent reward and success rate for the Buses domain

Calendar domain:

Domain	Success reward	Fail penalty	Success rate on training	Success rate on testing	Action accuracy	Avg. number of turns
Calendar	10	0	20%	26%	32%	14

Table 6.17: Best agent for the Calendar domain

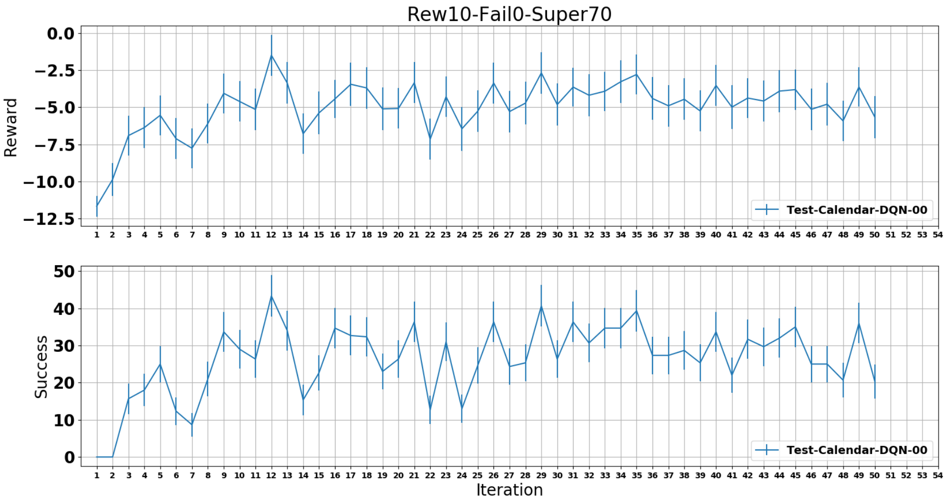

Figure 6.16: Best agent reward and success rate for the Calendar domain

Restaurants domain:

Domain	Success reward	Fail penalty	Success rate on training	Success rate on testing	Action accuracy	Avg. number of turns
Restaurants	10	0	35%	40%	50%	13

Table 6.18: Best agent for the Restaurants domain

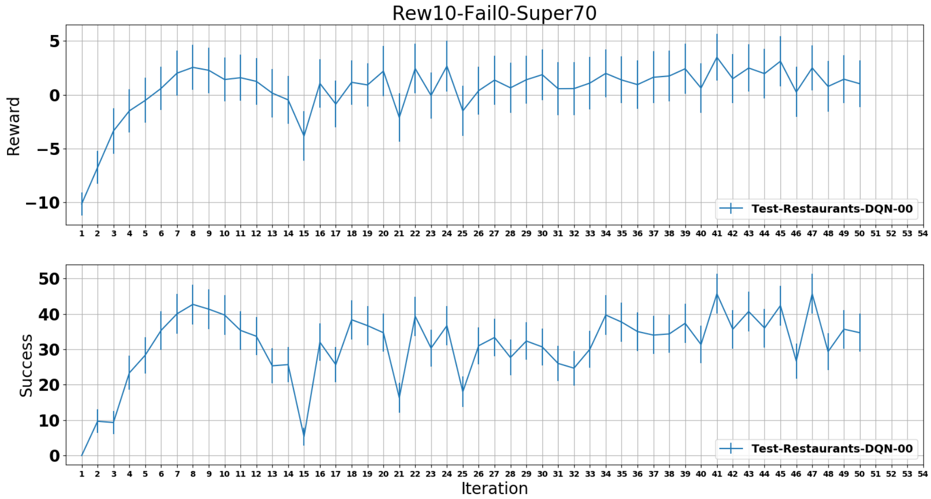

Figure 6.17: Best agent reward and success rate for the Restaurants domain

6.4 Comparison

In this section, we compare the various approaches. In table 6.19 we compare the success rate of the two reinforcement learning algorithms.

6.4.1 GP-SARSA vs DQN

Domain	GP-SARSA success rate on training	DQN success rate on training	GP-SARSA success rate on testing	DQN success rate on testing
Banks	90%	50%	81%	50%
Buses	100%	52%	98%	52%
Calendar	84%	48%	78%	47%
Restaurants	93%	62%	94%	64%

Table 6.19: GP-SARSA success rate vs DQN success rate

We can see that GP-SARSA is the best RL algorithm since it completes successfully a big percentage of dialogues during testing, arriving at 98% in the Buses domain.

In table 6.20 we report the action accuracy of the reinforcement learning agents for every domain, for the two algorithms.

Domain	GP-SARSA action accuracy	DQN action accuracy
Banks	27%	26%
Buses	27%	28%
Calendar	13%	15%
Restaurants	29%	27%

Table 6.20: GP-SARSA action accuracy vs DQN action accuracy

We can see that the two approaches have a similar action accuracy, and it is very low. This is because the agent receives a reward only at the end of every dialogue and not at every action, so it will focus on completing the dialogue successfully.

6.4.2 Pure-RL vs Supervised-RL

We start by comparing GP-SARSA and DQN and see which algorithms works better for Supervised-RL.

Domain	GP-SARSA success rate on testing	DQN success rate on testing	GP-SARSA action accuracy	DQN action accuracy
Banks	77%	28%	28%	41%
Buses	95%	29%	32%	37%
Calendar	77%	26%	25%	32%
Restaurants	79%	40%	39%	50%

Table 6.21: GP-SARSA success rate vs DQN success rate in Supervised-RL

In this case, GP-SARSA is still better if we look at the success rate, however, DQN is better if we look at the action accuracy. Now we compare the RL agents against the Supervised-RL agents. Since in Supervised-RL GP-SARSA is better in the success rate and DQN is better in the action accuracy we will compare by algorithm, so GP-SARSA RL with GP-SARSA Supervised-RL and the same with DQN.

Domain	GP-SARSA RL success rate on testing	GP-SARSA Supervised-RL success rate on testing
Banks	81%	77%
Buses	98%	95%
Calendar	78%	77%
Restaurants	94%	79%

Table 6.22: RL success rate vs Supervised-RL success rate using GP-SARSA

Domain	DQN RL success rate on testing	DQN Supervised-RL success rate on testing
Banks	50%	28%
Buses	52%	29%
Calendar	47%	26%
Restaurants	64%	40%

Table 6.23: RL success rate vs Supervised-RL success rate using DQN

In both the algorithms the success rate is lower in Supervised-RL. This is because 70% of the dialogues used in the training phase are training dialogues that are used to improve the action accuracy, and only 30% of the dialogues let the agent explore the best actions to complete the dialogue successfully and thus to improve the success rate.

Domain	GP-SARSA RL action accuracy	GP-SARSA Supervised-RL action accuracy
Banks	27%	28%
Buses	27%	32%
Calendar	13%	25%
Restaurants	29%	39%

Table 6.24: RL action accuracy vs Supervised-RL action accuracy using GP-SARSA

Domain	DQN RL action accuracy	DQN Supervised-RL action accuracy
Banks	26%	41%
Buses	28%	37%
Calendar	15%	32%
Restaurants	27%	50%

Table 6.25: RL success rate vs Supervised-RL action accuracy using DQN

The action accuracy is better in Supervised-RL with DQN having the best improvement.

In the end, from the success rate perspective, we can state that pure-RL is better than Supervised-RL, but from the action accuracy perspective, Supervised-RL is better. Overall we can't have a winner, however, we can say that the goal of Supervised-RL, which was improving the action accuracy without lowering so much the success rate has been reached successfully.

6.4.3 Supervised learning vs Reinforcement learning vs Supervised reinforcement learning

Finally, we compare supervised learning, reinforcement learning, and supervised-reinforcement learning. Since in supervised learning we measure only the action accuracy we will compare the agents using that metric. For reinforcement learning and Supervised-RL, we selected the best action accuracy between GP-SARSA and DQN.

Domain	SL action accuracy	RL action accuracy	Supervised-RL action accuracy
Banks	81%	27%	41%
Buses	83%	28%	37%
Calendar	81%	15%	32%
Restaurants	72%	29%	50%

Table 6.26: RL success rate vs Supervised-RL action accuracy using DQN

7 Conclusions

From the experiments section we can conclude that, if we use the action accuracy as a metric, supervised learning is the best approach for solving the problem of task-completion dialogues. An agent trained using supervised learning can learn from a set of training dialogues, and, for this reason, it can act as an agent that is already used in production or as a real human that talks with a user. Moreover, by using recurrent neural networks, the agent could learn to understand the context of the dialogue from the first sentences and ask and give the user relevant information about the task that he wants to complete.

Reinforcement learning, and, in particular, GP-SARSA, is still a good approach since it completes successfully a lot of dialogues. Reinforcement learning has also the advantage that we don't need training data, but we only need a user simulator and this user simulator could be customized to mimic every type of user, e.g. an angry user that does not provide the requested information. The problem with reinforcement learning is that it learns by trial and error and the reward is given only at the end of the dialogue. This is a problem because the agent could complete the dialogue successfully but, inside the dialogue, it could ask for information that is not inherent to what the user wants.

Supervised reinforcement learning turned out to be a good approach. Using DQN we managed to raise the action accuracy, however, it would be nice to have a higher success rate.

7.1 Future work

All the reinforcement learning agents have been trained using the default architectures that were present in *PyDial*. We could try to modify some parameters, e.g. for DQN we could try to add more neurons or more layers to the neural network and see if the agents improve.

To improve the supervised-rl agents it would be interesting to try to train the agent using only the supervised dialogues for a certain number of iterations, until the action accuracy is good, and then switch to pure reinforcement learning to increase the success rate.

For both RL and supervised-rl we should find the best parameters for the success rate, the fail penalty, and for the reward for every correct action/slot in supervised-rl. In our work, we tried some parameters and we stopped when the results were good, however, we should try all the combinations and see what works best. Since the training of an agent takes 3 to 5 hours, this work will take a lot of time, but it could improve our results.

Bibliography

[1] Milica Gašić and Steve Young. Gaussian processes for pomdp-based dialogue manager optimization. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22(1):28–40, 2014.

[2] Matthew Henderson, Blaise Thomson, and Jason D Williams. The second dialog state tracking challenge. In *Proceedings of the 15th annual meeting of the special interest group on discourse and dialogue (SIGDIAL)*, pages 263–272, 2014.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*, 2013.

[4] Abhinav Rastogi, Xiaoxue Zang, Srinivas Sunkara, Raghav Gupta, and Pranav Khaitan. Towards scalable multi-domain conversational agents: The schema-guided dialogue dataset. *arXiv preprint arXiv:1909.05855*, 2019.

[5] Weiyan Shi, Kun Qian, Xuewei Wang, and Zhou Yu. How to build user simulators to train rl-based dialog systems, 2019.

[6] Stefan Ultes, Lina M. Rojas Barahona, Pei-Hao Su, David Vandyke, Dongho Kim, Iñigo Casanueva, Paweł Budzianowski, Nikola Mrkšić, Tsung-Hsien Wen, Milica Gasic, and Steve Young. PyDial: A Multi-domain Statistical Dialogue System Toolkit. In *Proceedings of ACL 2017, System Demonstrations*, pages 73–78, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[7] H. Weld, X. Huang, S. Long, J. Poon, and S. C. Han. A survey of joint intent detection and slot-filling models in natural language understanding, 2021.

Appendix A Slot values

In the following table we report the slot names for every domain and the number of values for every slot.

Domain	slot	Number of values
Banks	account_type	2
	recipient_account_type	3
	balance	1024
	amount	365
	recipient_account_name	27
Buses	from_location	71
	to_location	65
	from_station	44
	to_station	39
	leaving_date	119
	leaving_time	238
	fare	382
	travelers	5
	transfers	2
	origin	111
	destination	1437
	origin_station_name	43
	destination_station_name	37
	departure_date	158
	price	616
	departure_time	270
	group_size	5
	fare_type	4
Calendar	event_date	127
	event_time	264
	event_location	592
	event_name	1252
	available_start_time	63
	available_end_time	38

Domain	**slot**	**Number of values**
Restaurants	restaurant_name	1777
	date	146
	time	232
	serves_alcohol	2
	has_live_music	2
	phone_number	1629
	street_address	1321
	party_size	6
	price_range	5
	city	368
	cuisine	234

Appendix B Example of generated dialogues

Successfull dialogue

```
 ** Turn 0 **
| Sys > hello()
| User > INFORM_INTENT(intent="findrestaurants")
** Turn 1 **
| Sys > request(date)
| User > INFORM(date="the 7th")
** Turn 2 **
| Sys > inform(date="the 7th",intent="findrestaurants",name="dynasty")
| User > INFORM_INTENT(intent="reserverestaurant")
** Turn 3 **
| Sys > notify()
| User > REQUEST(phonenumber)
** Turn 4 **
| Sys > inform(name="dynasty",phonenumber="925-244-1559")
| User > REQUEST(streetaddress)
** Turn 5 **
| Sys > inform(name="dynasty",streetaddress="711 w. broadway @ heather street
| User > GOODBYE()
** Turn 6 **
| Sys > bye()
```

In the first three turns the user wants to find a restaurant for the 7th. The agent informs the name of the restaurant (”dynasty”) and the user informs the intent `reserverestaurant`. Then the agent replies with a `notify()` action, meaning that he tried to reserve the restaurant. Finally the user asks the phone number and the street address of the restaurant and the agent replies successfully.

Fail dialogue

```
** Turn 0 **
| Sys > hello()
| User > INFORM_INTENT(intent="findrestaurants")
** Turn 1 **
system summary act: request_date.
| Sys > request(date)
| User > INFORM(date="1st of march")
** Turn 2 **
system summary act: request_pricerange.
| Sys > request(pricerange)
| User > INFORM(city="los altos")
** Turn 3 **
```

```
system summary act: request_pricerange.
| Sys > request(pricerange)
| User > INFORM(pricerange="moderate")
** Turn 4 **
system summary act: inform.
| Sys > inform(name="none",date="1st of march",city="los altos",pricerange="moder
| User > REQUEST(streetaddress)
** Turn 5 **
system summary act: request_partysize.
| Sys > request(partysize)
| User > NEGATE()
** Turn 6 **
system summary act: confirm_partysize.
| Sys > confirm(partysize="dontcare")
| User > AFFIRM()
** Turn 7 **
system summary act: confirm_date.
| Sys > confirm(date="1st of march")
| User > AFFIRM()
** Turn 8 **
system summary act: confirm_partysize.
| Sys > confirm(partysize="dontcare")
| User > AFFIRM()
** Turn 9 **
system summary act: confirm_partysize.
| Sys > confirm(partysize="dontcare")
| User > AFFIRM()
** Turn 10 **
system summary act: confirm_date.
| Sys > confirm(date="1st of march")
| User > AFFIRM()
** Turn 11 **
system summary act: confirm_partysize.
| Sys > confirm(partysize="dontcare")
| User > AFFIRM()
** Turn 12 **
system summary act: confirm_partysize.
| Sys > confirm(partysize="dontcare")
| User > AFFIRM()
** Turn 13 **
system summary act: confirm_partysize.
| Sys > confirm(partysize="dontcare")
| User > AFFIRM()
** Turn 14 **
system summary act: confirm_partysize.
| Sys > confirm(partysize="dontcare")
| User > AFFIRM()
** Turn 15 **
system summary act: confirm_partysize.
| Sys > confirm(partysize="dontcare")
| User > AFFIRM()
** Turn 16 **
system summary act: confirm_date.
```

```
| Sys > confirm(date="1st of march")
| User > AFFIRM()
** Turn 17 **
system summary act: confirm_date.
| Sys > confirm(date="1st of march")
| User > AFFIRM()
** Turn 18 **
system summary act: confirm_date.
| Sys > confirm(date="1st of march")
| User > AFFIRM()
** Turn 19 **
system summary act: confirm_date.
| Sys > confirm(date="1st of march")
| User > AFFIRM()
** Turn 20 **
system summary act: confirm_partysize.
| Sys > confirm(partysize="dontcare")
| User > AFFIRM()
** Turn 21 **
system summary act: confirm_partysize.
| Sys > confirm(partysize="dontcare")
| User > AFFIRM()
** Turn 22 **
system summary act: confirm_date.
| Sys > confirm(date="1st of march")
| User > AFFIRM()
** Turn 23 **
system summary act: confirm_partysize.
| Sys > confirm(partysize="dontcare")
| User > AFFIRM()
** Turn 24 **
system summary act: request_partysize.
| Sys > request(partysize)
| User > INFORM(partysize="dontcare")
** Turn 25 **
system summary act: inform.
| Sys > inform(name="none",date="1st of march",city="los altos",pricerange="
| User > REQUEST(streetaddress)
** Turn 26 **
Ending dialog due to MAX TURNS being reached: 26
```

Here we can see that until turn 5 the agent choses the correct actions. In turn 2 we can see that the user is not collaborating and replies with another slot, but the agent asks the previous slot again in turn 3 and the user replies correctly. From turn 6, till turn 23 the agent gets stuck asking confirmation for a slot and after 25 turns the dialogue ends because the maximum number of turns has reached.

Appendix C References for the replicability of the experiments

In this repository we pushed the modified version of PyDial along with the data used for training and testing. In the `README` file we reported the commands used to train and test the reinforcement learning agents. Here there is a zip file that contains all the trained RL agents along with their results.

Full link to the repository:
https://github.com/Dagu9/Reinforcement-learning-SGD
Full link to the zip file:
https://drive.google.com/file/d/1uxEdwYMT8gbmOoQI7nl-JMyl5l00eIru/view?usp=sharing

Since the supervised learning agent is trained using the private tools of a company, we cannot share the code or the agent files.