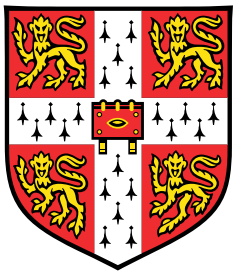
Sample efficient deep reinforcement learning for dialogue systems with large action spaces



Gellért Weisz

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
Master of Philosophy

Churchill College

10 August 2017

Declaration

I, Gellért Weisz of Churchill College, being a candidate for the M.Phil in Machine Learning, Speech and Language Technology, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Word count: 14902

Signed

Date: Thursday 10th August, 2017

Acknowledgements

I would first like to express my gratitude to my supervisor, Dr Milica Gašić, whose time and advice was indispensable when undertaking this piece of work. I would also like to thank Paweł Budzianowski and Eddy Pei-Hao Su for the advice they gave during this year, and for their assistance in implementing the ideas in practice. I further want to thank all the MLSALT professors, lecturers and examiners who have been there to answer my questions and ready to provide insights over the course of the year. Their teaching was excellent and their explanations of the intricacies of machine learning and speech technology were an invaluable resource.

Abstract

In Statistical Dialogue Systems, we aim to deploy Artificial Intelligence to build automated dialogue agents that can converse with humans. A part of this effort is the policy optimisation task, which attempts to find a policy describing how to respond to humans, in the form of a function taking the current state of the dialogue and returning the response of the system. In this project, we investigate Reinforcement Learning approaches to this problem. Particular attention is given to Deep Reinforcement Learning, Actor-Critic methods, off-policy Reinforcement Learning with Experience Replay, the Natural Policy Gradient and its approximations, and various methods aimed at reducing the bias and variance of estimators. When combined, these methods result in an algorithm called ACER, that beats the current state of the art in Statistical Dialogue Systems. This not only leads to a more sample efficient algorithm that can train faster, but also allows us to apply the algorithm in more difficult environments than before. We thus experiment with learning in the *master action space*, which has two orders of magnitude more actions. After optimising ACER to the *master action space*, it trains significantly faster and reaches a superior final performance than the current state of the art.

Table of contents

1 Introduction **1**

2 Preliminaties **3**

2.1 Spoken Dialogue Systems . . . 3
2.1.1 Speech recognition . . . 4
2.1.2 Ontology . . . 4
2.1.3 Semantic decoding . . . 4
2.1.4 Action spaces . . . 5
2.1.5 Dialogue management . . . 8
2.1.6 Natural language generation . . . 10
2.1.7 Speech synthesis . . . 11
2.2 Training the policy with Simulation . . . 11
2.3 Reinforcement Learning . . . 12
2.3.1 Model-based planning . . . 13
2.3.2 Model-free tabular reinforcement learning . . . 14
2.3.3 Function approximation . . . 18

3 Method **29**

3.1 Actor-critic with Experience Replay . . . 29
3.2 Lambda returns . . . 32
3.3 Retrace . . . 33
3.3.1 Computational cost . . . 35
3.4 Architecture of our actor-critic Neural Networks . . . 35
3.5 Importance Weight Truncation with Bias Correction . . . 37
3.6 Trust Region Policy Optimisation . . . 39
3.7 Summary of ACER . . . 43
3.8 Master actions for ACER . . . 46
3.9 Master actions for GP . . . 47

4 Evaluation **49**
4.1 Testing method 49
4.2 Performance of ACER 50
4.3 Contribution of TRPO 52
4.4 Effect of execution mask 53
4.5 Hyperparameter tuning 55
4.6 Master action space 58
4.7 Resilience against errors 60

5 Summary and Conclusions **63**
5.1 Future work 64
5.1.1 Supervised pre-training 64
5.1.2 Expanding the action space 65
5.1.3 Off-policy eNAC 65

References **67**

Appendix A Example dialogue **73**

Chapter 1

Introduction

Traditionally, computers are operated by either a keyboard and a mouse or touch. They provide feedback to the user primarily via visual clues on a display. This Human-computer Interaction model can be unintuitive to a human user at first, but it allows the user to express its intent clearly, as long as their goal is supported and they are equipped with sufficient knowledge to operate the machine. A Spoken Dialogue System (SDS) aims to make the Human-computer Interaction more intuitive by equipping computers with the ability to translate between human and computer language, thereby relieving humans of this burden and creating an intuitive interaction model. More specifically, the objective of an SDS is to help a human user achieve their goal in a specific domain (eg. hotel booking), using speech as the form of communication. Recent advances in Artificial Intelligence and Reinforcement Learning established the necessary technology to build the first generation of commercial Spoken Dialogue Systems deployable as regular household items. Examples of such systems are Amazon's Echo, Google's Home and Apple's Siri and HomePod.

Spoken Dialogue Systems are complex as they have to solve many challenging problems at once, under significant uncertainty. They have to recognise spoken language, decode the meaning of natural language, understand the user's goal while keeping track of the history of a conversation, determine what information to convey to the user, convert that information into natural language, and synthesise the sentences into speech that sounds natural. This work focuses on one particular step in this pipeline: devising a policy that determines the information to convey to the user, given our belief of their goal.

This policy has been traditionally planned out by hand using flow-charts. This was a manual and inflexible process with many drawbacks that ultimately lead to systems that were unable to converse intelligently. To overcome this, the policy optimisation problem has been formulated as a reinforcement learning problem. In this formulation, the computer takes actions and gets rewards. An algorithm aims to learn a policy that maximises the rewards

through learning to take the best actions based on the state of the dialogue. Since the number of possible states can be very large (potentially infinite), complex and universal function approximators such as Neural Networks have been deployed as the policy.

Using Neural Networks for policy optimisation is challenging for two reasons. First, there is often little training data available for an SDS as the data often comes from real humans. The system should be able to train quickly in an on-line setting while the training data is being gathered from users, to make the data to be gathered useful. Neural Networks often exhibit too much bias or high variance when the volume of training data is small, making it difficult to quickly train them in a stable way. Second, the success or failure of a dialogue may be the only information available to the system to train the policy on. Dialogue success depends crucially on most actions in the dialogue, making it difficult to determine which individual actions contributed to the success, or lead to the failure of a dialogue. This problem is exacerbated by the large size of the *state space*: the system will potentially never be in the same state twice.

This project deploys Neural Networks for policy optimisation, aiming to solve the related problems and derive a quick and stable learning algorithm, relying on recent innovations in the field. The final algorithm, called ACER, achieves the best results seen so far on Neural Network-based Spoken Dialogue Systems in the PyDial framework. An extension of this project investigates how both algorithms applied, GP and ACER, can be optimised tor the *master action space*. The performance of these algorithms beat the relevant state of the art reported in Spoken Dialogue System.

Chapter 2

Preliminaties

2.1 Spoken Dialogue Systems

A Spoken Dialogue System needs to solve many incredibly complex problems: it has to understand your speech, understand the logic and meaning behind it, use its knowledge base to come up with the right answer and phrase it in human language, then play it to the human. The current state of the art attacks this complex problem by splitting it into modules. In this *modular approach*, modules are eg. converting human speech to words, phrasing natural language from the system's internal language, etc. These modules form a *pipeline* illustrated in Figure 2.1. Even with this modular approach, a hard Machine Learning problem needs to be solved for each module. Due to the high amount of uncertainty they have to deal with, such as decoding a noisy recording of human speech into words, none of these problems have been solved optimally so far and they all correspond to active research fields in Artificial Intelligence. This highlights the benefit of the modular approach: since the modules have well-defined input and output and are trainable in isolation, an improved solution to a subproblem can be swapped in the pipeline without any change required for other parts of the system. For this reason, research efforts can be focused on specific modules in isolation.

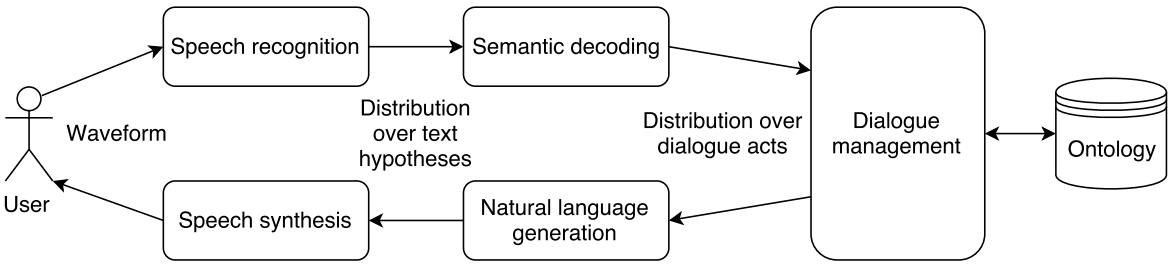


Fig. 2.1 Architecture of an Spoken Dialogue System.

Other approaches to Spoken Dialogue Systems include (1) hand-coding the entire pipeline with a flow-chart-based dialogue model, which is a less general approach, and (2) utilising Neural Networks to solve all of the involved problems at once. Approach (2), while possible in theory, is far beyond the scope of current AI technology.

2.1.1 Speech recognition

Humans have the benefit of having substantial general world knowledge and context knowledge that they use to distinguish between similarly sounding words when performing speech recognition. Even so, speech recognition is a challenging problem even for humans. Currently, computers cannot recognise speech with near-human certainty or accuracy [36]. Computers perform speech recognition even worse when significant noise is mixed with the speech signal. However, Dialogue Systems need to be designed such that they could be used in noisy environments too, such as in a car or on the street. Decoding errors will happen, but we can build resilience against such errors. Rather than a one-best hypothesis, the decoder outputs a set of most likely hypotheses for each utterance spoken by the user. This is conveyed in a lattice or confusion network form, along with the probabilities associated to the hypotheses. This method allows errors to be corrected further down the pipeline, on the basis of which decoding makes sense in the current context. Arguably, this method is similar to how a human would recognise speech by thinking about what makes sense in the context.

2.1.2 Ontology

The domain of a Spoken Dialogue System is defined by the ontology. The ontology is a structured representation of the database, or the knowledge base, of the system. It defines the type of entities users can interact with (eg. restaurants), their *requestable* slots, which are properties the user can ask about (eg. name, address, phone number), and the informable slots that users can specify (eg. price range, area). These are listed together with their possible corresponding values (eg. cheap, moderate and expensive for pricerange). The database consists of a set of entities and their corresponding properties. In this way, the ontology restricts the type and content of interaction between the user and the system.

2.1.3 Semantic decoding

Semantic decoding turns natural language into a domain-specific well-defined language that the back-end of an SDS can work with. It efficiently encodes the *user intent* or the *system reply* in a *dialogue act*. A *dialogue act* consists of a dialogue act type, and pairs of semantic

slots and values, as illustrated in the example below.

"Could you find me a cheap place in the city centre?" $\rightarrow$

$$\underbrace{\text{inform}}_{\text{dialogue act type}} \; (\; \underbrace{\text{price}}_{\text{semantic slot 1}} \; = \; \underbrace{\text{cheap}}_{\text{semantic value 1}} \; , \; \underbrace{\text{area}}_{\text{semantic slot 2}} \; = \; \underbrace{\text{centre}}_{\text{semantic value 2}} \;)$$

The set of possible slot-value pairs are defined by the ontology. For limited ontologies, applying Keyword Spotting with the ontology's vocabulary thus lends itself naturally to the semantic decoding problem. Support Vector Machines, Conditional Random Fields and Recurrent Neural Networks with Attention have been applied to this problem with varying levels of success [25]. It is important to note that the output of this step is a distribution over the user's possible *dialogue acts* along with their probabilities, with errors accumulating both from the semantic decoder and speech recogniser.

2.1.4 Action spaces

Before investigating further components of the SDS, we introduce the set of possible *system actions*, the *dialogue acts* that the system can give as a response. This is called the *action space*. We will investigate the *action space* in the restaurants domain, noting that most domains have a similar overall architecture. The set of possible actions are:

- **request + slot**

  Here, *slot* is an informable slot such as area, food, or pricerange. This action prompts the user to specify their criteria on a slot. Example textual representation: *"Which area are you interested in?"*.

- **confirm + slot**

  Here, *slot* is an informable slot. This action prompts the user to confirm their criteria on a slot that they may or may not have already mentioned. Due to errors accumulating during the decoding pipeline (speech recognition, semantic decoding, belief tracking), the system has to deal with considerable uncertainty, but it can attempt to increase its certainty in the user's criteria by using a confirm action. Example textual representation: *"Did you say you want an expensive restaurant?"*.

- **select + slot**

  Here, *slot* is an informable slot. This action prompts a user to select a value for the slot from a specified list of values. This is less open-ended than a request action and more open-ended than a confirm action. It be used by the system to increase its certainty in

a user criterion. Example textual representation: *"Would you like Indian or Korean food?"*.

- **inform + method + slots**

  This action provides information on a restaurant. The associated *method* specifies how the restaurant to give information on should be chosen. The exact choice of restaurant is not part of the action specification; it is derived by code when converting from the action into the *dialogue act*. This conversion process reads the action and the *belief state* and communicates with the database in the *ontology*, selecting a specific restaurant by applying heuristics.

  The *standard method* is to choose the first result in the *ontology* that matches the user criteria specified so far. The *method* can also be *byname*, in which case the system believes that the user asked about a specific restaurant by referring to its name, and information on that restaurant should be provided. If the method is *requested*, we inform on the same restaurant we informed on last, if it is *alternatives* then we pick another restaurant that matches the user's criteria (if possible).

  There are several properties of a restaurant, with a binary choice for each of them on whether the system wants to inform on it in a dialogue turn or not. The *informable* slots for restaurants are:

  - area
  - food type
  - description
  - phone number
  - price-range
  - address
  - postcode
  - signature

  We note that some of these slots are also *requestable*, allowing a user to query a restaurant based on those slots. These slots are *area, food type* and *price-range*. A restaurant also has a *name*, which we will always inform on. Together with the 7 slots listed above, the system has a choice between $2^8 = 256$ different ways it can inform on a restaurant. A specific choice is referred to as the *payload* of an inform action.

- **reqmore**

  This is a simple action that prompts the user to provide more input.

- **bye**

  This action is used to end the call, normally only as a response to the user's intention to end the call.

For the restaurants domain, there are $4 \cdot 2^8 = 1024$ inform actions and $3 \cdot 3 + 2 = 11$ other actions. We call this action space the *master action space*. Due to its large size, training a dialogue policy in this action space is increasingly difficult. Some algorithms do not converge to the optimal policy, converge very slowly, or, in rare cases, have prohibitive computational demands to converge [1]. To alleviate this problem, we introduce the *summary action space*. In this space, the inform actions do not specify which slots to inform on, leaving only 4 separate inform actions, and 15 actions in total. An example of a master action and corresponding summary action is:

$$\begin{aligned} \text{Master action:} \quad & \text{inform}(\text{price} = \text{cheap}, \text{area} = \text{centre}) \\ \text{Summary action:} \quad & \text{inform\_requested} \end{aligned}$$

If a policy is trained on the *summary action space*, the action selected by the policy needs to be converted to a *master action*. The conversion is a set of heuristics that attempts to find the optimal slots to inform on given the belief state. For example, if the *user intent* is to ask for the phone number of a restaurant, the heuristics would understand this from the belief state and derive that the phone number should be informed on. An example dialogue, including conversion between summary and master action space is included in Appendix A.

Using the *summary action space* comes with the clear benefit of a simpler dialogue policy optimisation task. On the other hand, the necessary heuristics to map to the *master action space* need to be manually constructed for each domain. Furthermore, since the belief state is read by hard-coded heuristics, the meaning of the representation has to be pre-agreed. This limits the applicability of a Neural Network as the *belief tracker* with an output vector trained to encode the belief in the most useful form.

Execution mask Not every system action is appropriate in every situation. For example, *inform* is not a valid action at the very beginning of the dialogue, when the system has not yet received any information on what kind of entity the user is looking for. Thus, we can

[1] Since the training has to be on-line, ie. happening while user input is acquired, training is limited in computation time to prevent the user from having to wait for the system to reply. However, the training step is rarely the bottleneck.

simplify the action space further by restricting the system actions that could be selected to the actions that the system is able to produce a corresponding sensible response for.

2.1.5 Dialogue management

The job of the dialogue manager is to take the user's *dialogue acts* and determine the appropriate response to them in the format of *dialogue acts*. Traditionally, this could be implemented as a large flow-chart with all interaction models carefully hand-crafted. The required manual work and the inflexibility of supported interactions heavily limit the success of this approach. Instead, our aim is to build a more intelligent dialogue manager out of two components, the belief tracker and the policy optimiser.

2.1.5.1 Belief tracking

We call the user's overall goal for a dialogue the *user goal*. This could be something like finding and booking a restaurant with some specified criteria. The user works towards this goal in every dialogue turn. In each dialogue turn, the short-term goal of the user is called the *user intent*. Examples of *user intent* are: *confirm* what the system said, *inform* the system on some criteria, and *request* more information on something.

The belief tracker is the memory unit of the SDS, with the aim to track the *user goal*, the *user intent* and the *dialogue history*. We call this tuple the *dialogue state*, and the output of the belief tracker is our current *belief* of this state.

Tracking *dialogue history* ensures we maintain the Markov Property - the assumption that the next state depends only on the current state and current action, and is conditionally independent of past actions and states. This is motivated by the idea that all relevant dialogue history should be incorporated in the *dialogue state*. The tracking is slightly more complex however, as we only have a Partially Observable Markov Decision Process (POMDP). This is illustrated in Figure 2.2, where the grey variables are not observable. a_t is the current action, s_t and s_{t+1} are the current and next state, o_t and o_{t+1} are the current and next observations (*belief state*) and r_t is the current reward, an indirect measure of how successful taking action a_t was in state s_t (more on rewards in Section 2.3).

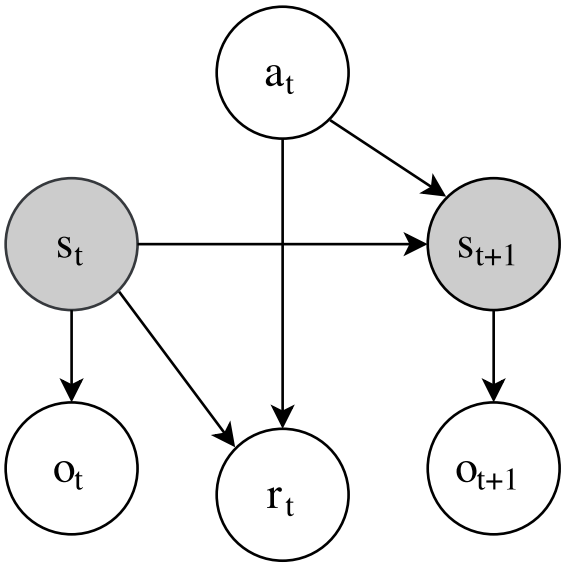


Fig. 2.2 Partially Observable Markov Decision Process in the Dialogue System.

Inference using this formulation is intractable due to the large (possibly infinite) state space [14]. We could alleviate the problem by breaking the state into the *user goal*, *user intent* and *dialogue history* tuple and further assuming certain conditional independencies between these items, but this would be specifically engineered to an SDS task and would just be an approximation. According to Kaelbling et al. [14], we could view the problem as a continuous-space Markov Decision Process (MDP), where the states are the *belief states* (Figure 2.3). We adopt this method in this work.

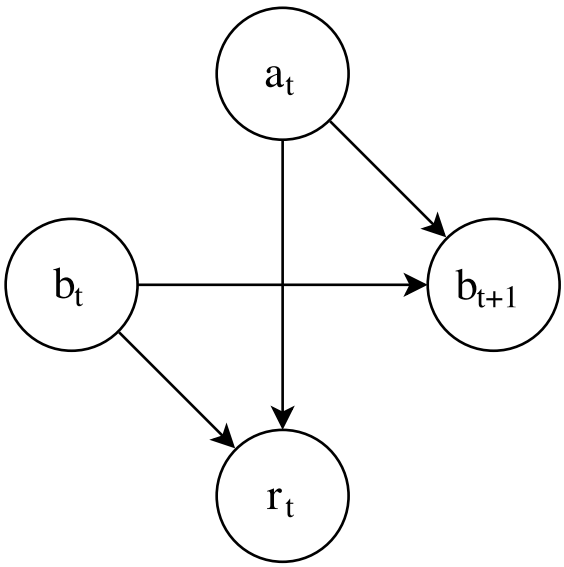


Fig. 2.3 Markov Decision Process in the Dialogue System.

In the PyDial framework, the *belief state* is represented as a dictionary that holds the probability distributions of every value for every slot in the ontology. This could also be represented as a long vector of individual probabilities.

2.1.5.2 Policy optimisation

A *policy* is a probability distribution of possible *user actions* given the current *belief state*, and is commonly written as $P(a|b,\pi) = \pi(a|b)$. Here, π is the policy, a is the action and b is the output of the belief tracker, which is interpreted as a vector of probabilities [2].

In order to define the optimal policy, we need to introduce a utility function (*reward*) that describes how good taking action a is in state b. The reward for a complete dialogue depends on whether the user was successful in reaching their goal and the length of the conversation, such that short successful dialogues are preferred. Thus, the last dialogue interaction gains a reward based on whether the dialogue was successful, and every other interaction loses a small constant reward, penalising for the length of the dialogue. The cumulative reward of a dialogue is calculated as:

$$R(b_0, a_0, ..., b_T, a_T) = \sum_{t=0}^{T} r(b_t, a_t),$$

where T is the length of the dialogue. The task of policy optimisation is to maximise the expected cumulative reward in any state b when following policy π, by choosing the optimal action a from the set of possible actions $\mathbb{A}$:

$$\pi^*(b_i) = \arg\max_{a_i \in \mathbb{A}} \mathop{\mathbb{E}}_{b_{i+1} \sim P(a_i|b_i), a_{i+1} \sim \pi^*(b_{i+1}), ...} \big(R(b_i, a_i, ..., b_T, a_T)\big).$$

2.1.6 Natural language generation

The output of the Dialogue Manager is a *dialogue act*, the *system response* to the user. Whether the summary or the master action space is used in the Dialogue Manager, its output will be converted to a canonical form, with all the values for the slots filled in according to the *ontology*. This *dialogue act* is converted into textual natural language by the Natural Language Generation (NLG) module. An example for such a conversion is:

$$\text{request + area} \rightarrow \text{Which area are you interested in?}$$

[2] In our case, the action space is discrete, but we note that in general, the action space can also be continuous.

The goal is for the textual output to be easy to understand by the human and convey the same meaning as the machine-representation of the *system response*.

2.1.7 Speech synthesis

The textual output of the Natural Language Generation unit is converted into waveform. The goal is to derive an audio representation of the exact textual input that sounds natural and is easy to understand to a human. The output of this unit is played to the user.

2.2 Training the policy with Simulation

Due to the modular nature of Spoken Dialogue Systems, we are able to isolate the *dialogue management* part of the pipeline and train it separately with a *simulated user*. The key advantage of this approach is its support for automated training of the policy without a human component. The simulation pipeline is illustrated in Figure 2.4. The simulation runs on the semantic level, but it is also possible to run it on speech and and textual transcript level, involving multiple stages of the original SDS pipeline. After every dialogue turn, the simulator feeds back a *reward* signal to the optimisation process. This typically encodes how efficient and successful a dialogue is.

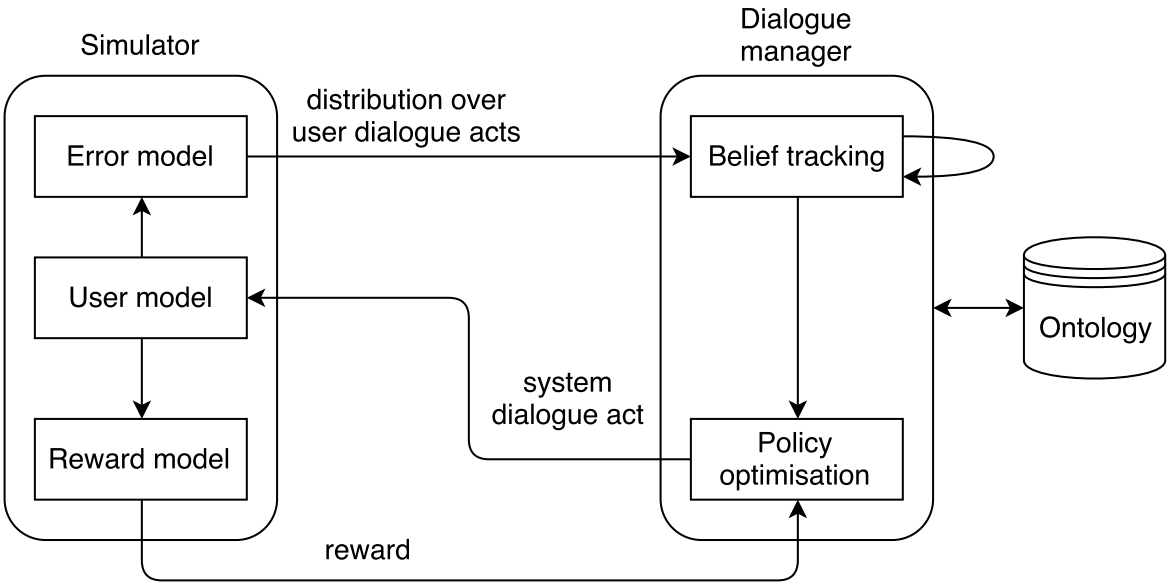


Fig. 2.4 Simulation pipeline used to train the policy.

For this project, we use a fairly simple *belief tracker* for the simulated training of the policy called the *focus tracker*. The focus tracker updates its current belief based on the certainty of incoming data (probabilities of being in state s expressed as $o(s)$) and the previous belief:

$$b(s_t = s) = o(s) + \left(1 - \sum_{s'} o(s')\right) b(s_{t-1} = s).$$

Alternatively, the *dialogue tracker* could be the output of a Neural Network trained separately [12].

The implementation of the *user simulator* carries further choices. Ideally, the simulator's behaviour is as close as possible to the target human user. The *data-driven* approach learns human behaviour from training data acquired from human test users – another Machine Learning problem in itself. This approach can learn both user actions and user satisfaction (reward) with a dialogue. Instead, the simulator used in this project is *agenda-driven*: the simulated user has a particular goal, eg. to find a restaurant based on some particular criteria, and it will take actions to reach this goal. These actions are selected by some code that relies on *hyperparameters* controlling its behaviour, such as patience and informativeness. At the end of a dialogue, the reward is based on whether the user goal was reached. This simulator also has an *error model* that simulates uncertainty coming from the semantic decoder by outputting a distribution over user dialogue acts with some added noise. This makes the training process more difficult, but leads to a more stable policy that can operate with uncertainty in the input.

The starting point of this dissertation is this entire simulation pipeline implemented in the *PyDial* framework, as well as some policy optimisation algorithms. This project builds on these and implements novel algorithms with the goal that they converge to the optimal policy quickly, in a stable way, using few sample dialogues. The training will be run with the pipeline described above, on the Cambridge Restaurants domain. While these choices are specific, the modular design of SDSs means that the training algorithms implemented are generally applicable to any set of choices.

2.3 Reinforcement Learning

In Reinforcement Learning, there is an agent interacting with the environment in discrete time steps. In each time step, the agent observes the environment as a belief state vector b_t and chooses an action a_t out of the action space $\mathbb{A}$. We will consider discrete action spaces but in general they can be continuous. After performing action a_t, the agent observes a reward r_t produced by the environment. The interaction model is illustrated in Figure 2.5.

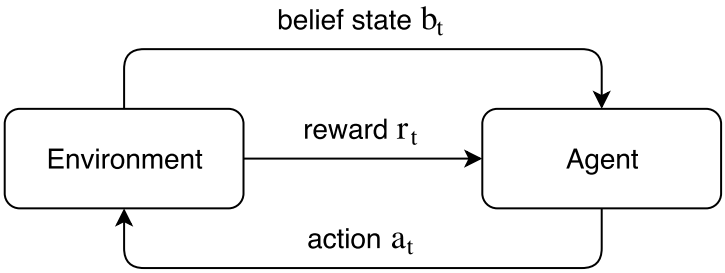


Fig. 2.5 Reinforcement Learning interaction model between agent and environment.

We need a concept of the total value of an episode of interactions. We call this the cumulative discounted return. For the t^{th} timestep, we calculate this as

$$R_t = \sum_{i \geq 0} \gamma^i r_{t+i}.$$

The *discount factor* γ trades-off the importance of immediate and future rewards. The goal of the agent is find a policy that maximises the expected discounted cumulative return for every state. We define the value of a state-action pair under policy π to be the Q-function:

$$Q_\pi(b_t, a_t) = \mathop{\mathbb{E}}_{b_{t+1:T}, a_{t+1:T}} \big(R_t | b_t, a_t\big),$$

and the value of a state is the V-function:

$$V_\pi(b_t) = \mathop{\mathbb{E}}_{a_t} \big(Q_\pi(b_t, a_t) | x_t\big).$$

In both definitions, the expectation is taken over the states the environment could be in after performing the actions, and the actions selected by policy π.

2.3.1 Model-based planning

The state transition and reward probabilities of the *environment* $p(b', r|b, a)$ are described by the *model*. If this model is available, we apply *planning* to find the optimal policy. To derive this, we start from the Bellman optimality equation. Assuming we have the optimal value

function $V_*(b)$ and optimal policy π^*:

$$\begin{aligned} V_*(b) &= \max_a \overset{*}{\underset{\pi}{\mathbb{E}}}\big(R_t|b_t,a_t\big) \\ &= \max_a \overset{*}{\underset{\pi}{\mathbb{E}}}\big(r_{t+1}+\gamma\sum_{k=0}^{T-t-1}\gamma^k r_{t+k+2}|b_t,a_t\big) \\ &= \max_a \overset{*}{\underset{\pi}{\mathbb{E}}}\big(r_{t+1}+\gamma V_*(b_{t+1}\big) \\ &= \max_a \sum_{b',r} p(b',r|b,a)\big(r+\gamma V_*(b')\big) \end{aligned}$$

If starting from a baseline value function (eg. all values are zero) for V_0, we can apply the following iterative update process based on the Bellman equation:

$$V_{k+1}(b) = \max_a \sum_{b',r} p(b',r|b,a)\big(r+\gamma V_k(b')\big) \tag{2.1}$$

It can be proved that the update process leads to convergence: $\lim_{k\to\infty} V_k(b) = V_*(b)$. This leads to the *Value iteration algorithm*:

Algorithm 1 Value iteration

1: Initialise V_0 arbitrarily
2: **repeat**
3: Improve V_{k+1} using the estimate of V_k (Equation 2.1)
4: **until** convergence

2.3.2 Model-free tabular reinforcement learning

As the reinforcement learning scenarios become more challenging, the model of the environment is often not available to the agent. In such cases, the agent learns from trial and error by interacting with a simulated or real environment. Monte Carlo methods can be used to estimate transition and reward probabilities of the environment. These estimates are accurate only in the limit of infinite observations for each state-action pair, thus a requirement for the *behaviour policy* is to maintain exploration, ie. to keep visiting all state-action pairs with nonzero probability. The *behaviour policy* is the policy used to generate the data during learning. For **on-policy methods**, the behaviour policy is the same as the learned policy, in other words, we evaluate and improve the same policy that is used to make decisions. In contrast, **off-policy methods** evaluate and improve a policy different from that used to generate the data, ie. the behaviour policy and the learned policy can be different. The

advantage of off-policy methods is that the optimal policy can be learned while the behaviour policy can be set to explore sub-optimal actions too, so that the algorithm can keep exploring the state-space. Thus, the purpose of the behaviour policy is to *soften* the optimal policy; we will see a concrete example of how this is achieved.

2.3.2.1 On-policy Monte Carlo control

An example of an on-policy method is the On-policy Monte Carlo control (Algorithm 2). This algorithm iterates through the following steps: first, it gathers a full episode based on the current policy π. Then, it recalculates the average cumulative rewards for every state, action pair based on the observations. This is used as the Monte Carlo approximation to the Q-function. Finally, it carries out an update step, where the policy is selected based on the approximated Q-function.

To maintain exploration, instead of following the best action based on the policy π, we use ε-greedy exploration, which selects the best action (according to π) with probability $1-\varepsilon$, and selects a random action with probability ε for $\varepsilon > 0$. The choice of ε represents the exploration-exploitation trade-off: A large ε will lead to the agent exploring more of the environment, but a small ε allows the exploitation of the policy learned so far, which helps the agent learn faster. A common choice is to start off with a high ε and gradually decrease it as time progresses and the agent's predictions become more reliable.

Algorithm 2 On-policy Monte Carlo control

Initialise Q and π arbitrarily $Returns \leftarrow$ empty list$\forall b \in \mathbb{B}, a \in \mathbb{A}$
repeat
  for $b \in \mathbb{B}$ and $a \in \mathbb{A}$ **do**
    Generate an episode using ε-greedy π starting with b, a
    for b, a in the episode **do**
      $Returns \leftarrow$ append return reveived for action a in state b
      $Q(b,a) \leftarrow average(Returns(b,a))$
    for b in the episode **do**
      $\pi(b) \leftarrow \arg\max_a Q(b,a)$
until convergence

2.3.2.2 Off-policy Monte Carlo control

For the off-policy version of Monte Carlo control, the target policy π is greedy with respect to the approximated Q, and we generate the behaviour based on a different policy μ. We require for convergence that μ visit all states and actions with nonzero probability. To correct

for this discrepancy we need to introduce Importance Sampling (IS) weights. Otherwise we would add bias and the expected value of the approximated Q function would no longer be the optimal Q_*. The IS weights are $\rho = \frac{\pi(a,b)}{\mu(a,b)}$. Due to π being zero for every non-optimal action according to Q, this means that all traces will be cut as soon as the generated episode disagrees with π (Algorithm 3).

Algorithm 3 Off-policy Monte Carlo control

Initialise Q arbitrarily, $N(b,a) = D(b,a) = 0 \ \forall b \in \mathbb{B}, a \in \mathbb{A}$ $\pi \leftarrow$ greedy with respect to Q
repeat
    Generate episode $\{b_{0:T}, a_{0:T}\}$ using behaviour policy π
    $R \leftarrow 0, W \leftarrow 1$
    for $t = T$ down to 0 **do**
        $R \leftarrow \gamma R + r_{t+1}$
        $N(b_t, a_t) \leftarrow N(b_t, a_t) + W \cdot R$ ▷ numerator of Q
        $D(b_t, a_t) \leftarrow D(b_t, a_t) + W$ ▷ denominator of Q
        $Q(a_t, a_t) \leftarrow \frac{N(b_t, a_t)}{D(s_t, a_t)}$
        $\pi(b) \leftarrow \arg\max_a Q(b,a)$
        if $a_t \neq \pi(b_t)$ **then**
            Exit for loop ▷ cut trace as IS weight becomes 0
        $W \leftarrow W \frac{1}{\mu(b_t, a_t)}$ ▷ calculate IS weight
until convergence

Both off-policy and on-policy Monte Carlo control can be proved to converge, however, for off-policy Monte Carlo control, this relies on sufficient exploration of the state-space. The problem with these methods is their slow convergence, sample inefficiency and high variance. Both on-policy [6] and off-policy [13] methods have been applied to SDS.

2.3.2.3 Temporal Difference learning

Monte Carlo control approximates the episodic cumulative reward for each state and action. These updates depend very much on how successful the episode has been in the future and thus the Monte Carlo updates introduce high variance. To reduce the variance, the function approximations can be updated every time step rather than per episode, based on the current approximation and the reward received. These updates are called the Temporal Difference (TD) error; they are multiplied by the learning rate α and added to the current approximation.

SARSA updates the current Q-function estimate based on the state, action, reward, next state, next action (b, a, r, b', a') tuple with the TD-error:

$$\delta = r + \gamma Q(b', a') - Q(b, a)$$

Algorithm 4 shows the full algorithm.

Algorithm 4 SARSA

Initialise Q arbitrarily, $Q(terminal, \cdot) = 0$
repeat
  Initialise b, choose a ε-greedily
  $R \leftarrow 0, W \leftarrow 1$
  repeat
    Take action a, observe r, b'
    Choose a' ε-greedily
    $Q(b, a) \leftarrow Q(b, a) + \alpha(r + \gamma Q(b', a') - Q(b, a))$
    $b \leftarrow b', a \leftarrow a'$
  until b is terminal
until convergence

Q-learning updates the current Q-function esimate based on the state, action, reward, next state (b, a, r, b') tuple, dropping the next action. Instead, the update is based on the a' that maximises the Q-value estimate. In this way, Q-learning is off-policy as the policy improved is greedy with respect to Q and may be different from the behaviour policy. The TD-error is:

$$\delta = r + \gamma \max_{a'} Q(b', a') - Q(b, a)$$

Algorithm 5 shows the full algorithm.

Both Q-learning and SARSA can be shown to converge if ε tends to 0 and every state-action pair is visited infinitely often. Compared to Monte Carlo control, these methods have lower variance but higher bias, which can be attributed to the difference in the update rule. Q-learning has been applied to Dialogue Systems by Levin et al. [17], and SARSA has been applied by Henderson et al. [11].

A fundamental shortcoming of standard tabular methods is that since approximate function values for different state-action pairs are separated, the system's knowledge of a state-action pair does not generalise to other state-action pairs.

Algorithm 5 Q-learning

Initialise Q arbitrarily, $Q(terminal, \cdot) = 0$
repeat
    Initialise b
    repeat
        Choose a ε-greedily
        Take action a, observer r, b'
        $Q(b,a) \leftarrow Q(b,a) + \alpha(r + \gamma \max_{a'} Q(b',a') - Q(b,a))$
        $b \leftarrow b'$
    until b is terminal
until convergence

2.3.3 Function approximation

So far we have been considering tabular solutions, with discrete action and state spaces such that $Q(b,a)$ can be approximated with independent values for every (b,a) pair. In more realistic scenarios, the set of states or actions are either continuous or large enough for tabular approximations to be inefficient. One could quantise the state space and apply a tabular algorithm, but this may introduce large errors with the quantisation levels necessary for the performance to be acceptable. Another method is to apply function approximation for Q, V and π.

Non-parametric Gaussian Process-based approximation The Q-function can be approximated as a Gaussian process with zero mean and kernel function $k(\cdot,\cdot)$. This means that for any set of points $P_{1:n}$, they are distributed according to a zero mean Gaussian and the covariance matrix is given by $K_{ij} = k(P_i, P_j)$. Here, a point is a belief-action pair. Since covariances are always positive, the covariance matrix K needs to be positive semidefinite. k is only a kernel function if this is satisfied. Examples of kernel functions are:

- The linear belief kernel:

$$k((b,a),(b',a')) = \langle b, b' \rangle \delta_a(a'),$$

  where δ is the Kronecker delta function and $\langle \cdot, \cdot \rangle$ is the scalar product.

- The Gaussian belief kernel:

$$k((b,a),(b',a')) = p^2 \exp\left(-\frac{||b-b'||_2^2}{2l^2}\right) \delta_a(a'),$$

  where p and l are hyperparameters that are fixed before training.

After every iteration, the Q function is updated by the posterior given all state-action (b,a) pairs and rewards observed so far [29]. A problem with Gaussian Processes is that the computational cost of inference grows cubically with the number of datapoints. To combat this, we set a sparcification threshold that ensures we only save datapoints whose belief states are "distant" enough from the ones we already saved [7]. Even though this introduces an approximation, Gaussian Processes generally enjoy quick convergence, good performance and sample efficiency for small action spaces. Compared to other function approximation methods, the main advantages of GP are that it 1) allows the incorporation of the prior knowledge via the kernel function and 2) it provides an estimate of the uncertainty. GP has been applied to Dialogue Systems by Gašić et al. [8].

Value-function approximation We can approximate the *value function* according to policy π, $V_\pi(b)$ as

$$V_\pi(b) \approx \hat{V}_\pi(b,\theta).$$

To define $\hat{V}$, it is common to introduce feature functions that map to feature vectors of length N: $\phi(b) : B \to \mathbb{R}^N$, where $\mathbb{B}$ is the set of states. Provided the parameter set θ is a similar N-length vector of rationals, we can define $\hat{V}$ to be a linear function of θ:

$$\hat{V}_\pi(b,\theta) = \theta^T \cdot \phi(b).$$

Linear function approximation has been applied to Dialogue Systems by Chandramohan et al. [2], but more complex definitions are also possible. Another commonly used definition feeds the input state through a Neural Network, making $\hat{V}$ the output of the network. For what follows, we only require $\hat{V}$ to be differentiable with respect to its parameters θ.

With the aim of bringing our approximation $\hat{V}_\pi(b)$ as close as possible to the true value $V_\pi(b)$, we introduce a measure off (Mean Squared Value Error) error to minimise:

$$\text{MSVE} = \sum_b d(b)\big(V_\pi(b) - \hat{V}_\pi(b,\theta)\big)^2,$$

where $d(b)$ is the fraction of time spent in b under policy π. [3]

[3] Intuitively, the more time we spend in a state, the more accurate we want our value approximation of it to be. This is expressed by weighting the errors by $d(b)$.

In order to minimise the error, the weights θ can be trained with Stochastic Gradient Descent. Assuming that we receive samples b_t and $V_\pi(b_t)$ with distribution $\pi(b)$,

$$\begin{aligned}\theta_{t+1} &= \theta_t - \frac{1}{2}\alpha\nabla\big(V_\pi(b_t) - \hat{V}_\pi(b_t,\theta)\big)^2 \\ &= \theta_t + \alpha\big(V_\pi(b_t) - \hat{V}_\pi(b_t,\theta_t)\big)\nabla\hat{V}_\pi(b_t,\theta_t),\end{aligned}$$

where α is the learning rate. Of course we normally do not have access to the true value of $V_\pi(b_t)$ in a model-free environment, so we approximate it, relying on observations. Common, previously mentioned approximations are:

- **Monte Carlo back-up**: $V_\pi(b_t) \approx R_t$. Suffers from high variance, but its approximations do not depend on previous estimates of V, ie. does not bootstrap.
- **TD back-up**: $V_\pi(b_t) \approx r_t + \gamma\hat{V}_\pi(b_{t+1},\theta)$. This method bootstraps, ie. it bases its approximations of V on previous approximations.

As an example, Algorithm 6 presents the *value-function approximation* method with *Monte Carlo back-up*.

Algorithm 6 Gradient Monte Carlo Algorithm for Value-function approximation

Input: policy π and differentiable function $\hat{V}(b,\theta) : B \times \mathbb{R}^N \to \mathbb{R}$
Initialise θ_0
repeat
    Generate episode $\{b_{0:T}, r_{0:T}\}$ according to π
    for $t = 0, 1, ..., T$ **do**
        $\theta_{t+1} = \theta_t + \alpha\big(R_t - \hat{V}(b,\theta_t)\big)\nabla\hat{V}(b_t,\theta_t)$
until convergence

Policy-based approximation A small change in the value function can result in erratic changes in the policy, making the Value-based approximation method somewhat unstable [29]. To provide stronger convergence guarantees, we aim to learn a parametrised policy directly, without the need for computing a value function.

We parametrise the policy with $\omega \in \mathbb{R}^N$ to get $\pi(a|b,\omega)$. A possible way to define π is

$$\pi(a|b,\omega) = \frac{\exp(\omega^T\phi(b,a))}{\sum_{a'} exp(\omega^T\phi(b,a'))},$$

where $\phi(b,a)$ are the features extracted from a state-action pair. This is called the *softmax function* and it produces a valid probability distribution for any input: the output probabilities

fall between 0 and 1, and sum up to 1. Similarly to *value-based approximation*, π can also be eg. the output of a Neural Network. We require π to be a valid probability distribution and to be differentiable with respect to ω.

If $J(\omega_t)$ is a measure of the success of π parametrised with ω_t, then the gradient update rule is

$$\omega_{t+1} = \omega_t + \alpha \nabla J(\omega_t).$$

Policy Gradient Theorem. *[19, 30] If*

$$J(\omega) = V_{\pi(\omega)}(b_0), \textit{ then}$$

$$\nabla J(\omega) = \sum_b d_\pi(b) \sum_a Q_\pi(b,a) \nabla_\omega \pi(a|b,\omega),$$

where $d_\pi(b)$ is the fraction of time spent in b under policy π.

Reinforce Using the Policy Gradient Theorem and the update rule, we need a method to approximate the gradient of J. In order to approximate

$$\sum_b d_\pi(b) \sum_a Q_\pi(b,a) \nabla_\omega \pi(a|b,\omega),$$

we notice that $d_\pi(b)$ is the state-distribution under the policy. So if we follow π, we get the required distribution of states. However, we need to correct for the distribution of actions as they are not weighted in our target. Next, we notice that $Q_\pi(b,a)$ can be approximated with the discounted cumulative reward R. Finally, we have

$$\begin{aligned}
\nabla J(w) &= \mathop{\mathbb{E}}_{\pi}\left[\gamma^t R_t \frac{\nabla_\omega \pi(a|b,\omega)}{\pi(a|b,\omega)}\right] \\
&= \mathop{\mathbb{E}}_{\pi}\left[R_t \nabla_\omega \log \pi(a|b,\omega)\right] \\
\omega_{t+1} &= \omega_t + \alpha R_t \nabla \log \pi(a_t|b_t,\omega_t).
\end{aligned}$$

This leads to Algorithm 7 by Williams [35].

Actor-critic methods We can simultaneously estimate the policy and the value function (or Q-function) in an iterative manner.

1. **Actor:** improves the current policy based on the Q-function estimated by the critic.
2. **Critic:** improves the current Q-function based on the new policy.

Algorithm 7 REINFORCE

Input: policy $\pi(a|b,\omega)$, learning rate $\alpha > 0$
Initialise ω
repeat
    Generate episode $\{b_{0:T}, r_{0:T}\}$ according to $\pi(\cdot|\cdot,\omega)$
    for $t = 0, 1, ..., T$ **do**
        $R_t \leftarrow$ cumulative return at step t
        $\omega \leftarrow \omega + \alpha R_t \nabla \log \pi(a|b_t,\omega)$
until convergence

This is a combination of *value-based approximation* and *policy-based approximation*, in that we can apply an approximators of both types for the actor and the critic respectively. The advantage of this method compared to *REINFORCE* is that here, $Q_\pi(b,a)$ is approximated by function approximation rather than R. This greatly reduces the variance and stabilises learning. To further reduce the variance of this method, we observe that

$$\begin{aligned}\nabla J(\omega) &= \mathop{\mathbb{E}}_{\pi}[R_t \nabla_\omega \log \pi(a|b,\omega)] \\ &= \mathop{\mathbb{E}}_{\pi}[Q_\pi(b,a) \nabla_\omega \log \pi(a|b,\omega)] \\ &= \mathop{\mathbb{E}}_{\pi}[(Q_\pi(b,a) - V_\pi(b)) \nabla_\omega \log \pi(a|b,\omega)],\end{aligned}$$

where in the last step we made use of the fact that

$$\sum_a (V_\pi(b) \nabla_\omega \pi(a|b,\omega)) = V_\pi(b) \cdot \sum_a (\nabla_\omega \pi(a|b,\omega)) = 0,$$

as $\pi(\cdot|b)$ needs to sum to 1. See Sutton et al. [30] for full proof. Let us define the advantage function

$$A_\pi(b,a) = Q_\pi(b,a) - V_\pi(b).$$

The variance of the estimation of Q in the policy gradient can thus be further reduced if the advantage function is used:

$$\nabla J(\omega) = \mathop{\mathbb{E}}_{\pi}[A_\pi(b,a) \nabla_\omega \log \pi(a|b,\omega)].$$

The critic could estimate both V_π and Q_π, or only the advantage function $A_\pi(b,a)$ directly.

Natural Actor Critic (NAC) A problem with the *vanilla* gradient ascent presented in Section 2.3.3 is that small changes in the parameter space in the course of an update step can

lead to large differences in the policy. Even with a low learning rate (α), this could lead to unstable learning due to the policy changing erratically. When α is low enough to avoid this, it can be too low for learning to converge in a reasonable time frame.

One way to solve this is to restrict the step size in the policy space rather than the parameter space [31, 21]. More specifically, the distance in the parameter space is defined using a *distance tensor* G_ω:

$$|d\omega|^2 = d\omega^T G_\omega d\omega.$$

Gradient ascent updates the parameters ω in the direction of the steepest ascent, such that the difference between the old and new ω is within the learning rate α. Traditional (vanilla) gradient ascent uses the Euclidean distance measure by defining G_ω as the identity matrix. Instead, *natural gradient* defines G_ω as:

$$(G_\omega)_{ij} = \mathbb{E}\left[\frac{\delta \log p(x|\omega)}{\delta \omega_i}\frac{\delta \log p(x|\omega)}{\delta \omega_j}\right],$$

For a distribution p that depends on the parameters ω. This is the Fisher Information Matrix. Intuitively, since this distance measure operates in the distribution space of p, it is invariant to the scaling of the parameters ω (Amari [1]). For a general distance, Amari [1] also shows that the steepest direction of ascent is

$$G_\omega^{-1}\nabla_\omega J(\omega).$$

Calculating the inverse of such Fisher Information Matrices tends to be prohibitively costly in general. Natural Actor Critic (NAC) fixes this by using *compatible function approximation*. It defines $\pi_\omega(a|b)$ freely, with the requirement that it be differentiable with respect to ω. This can be the output of a neural network. However, the definition of the critic estimating the advantage function is more restricted. Parametrised with w, it is defined as

$$A_w(b,a) = \nabla_\omega \log \pi(a|b,\omega)\cdot w.$$

Thus, the parameters w are weights for a linear combination. Peters et al. [23] shows that under this definition, if w minimises the squared approximation error, ie.

$$w = \arg\min_w \mathbb{E}\left(A_\pi(b,a) - A_w(b,a)\right)^2,$$

then the natural gradient is

$$G_\omega^{-1}\nabla_\omega J(\omega) = w.$$

Thus the natural gradient for the actor update is recovered by solving the minimisation problem for w during the critic update. A version of this algorithm, Episodic Natural Actor Critic (eNAC), relies on the following (for complete derivation see Thomson [31]):

$$\begin{aligned}&\sum_{n=1}^{N}\left|\sum_{t=0}^{T_n-1}\left(A_\pi(b_{n,t},a_{n,t})-A_w(b_{n,t},a_{n,t})\right)\right|^2\\&=\sum_{n=1}^{N}\left|\sum_{t=0}^{T_n-1}\left(r_{n,t}-\nabla_\omega\log\pi_\omega(a_{n,t}|b_{n,t})\cdot w-J(\omega)\right)\right|^2,\end{aligned}$$

where n and t are reference the indices of the episodes and the turns within the episodes, respectively. Having collected N episodes, eNAC applies Least Squares to find w that minimises this expression. Since we do not have the value of $J(\omega)$, it needs to be approximated too. However, it is just one constant that is the same throughout the episodes, so we only introduce one additional degree of freedom. Algorithm 8 displays the complete algorithm.

Algorithm 8 Episodic Natural Actor Critic

1: Input: policy $\pi(a|b,\omega)$, $A_w(b,a)$ and learning rate $\alpha>0$
2: Initialise ω
3: **repeat**
4: Gather N episodes according to π_ω with states $b_{n,t}$, actions $a_{n,t}$ and total rewards R_n
5: **Critic evaluation** Apply least squares to find w and $J(\omega)$ to minimise

$$\sum_{n=1}^{N}\left(\sum_{t=0}^{T_n-1}\left(R_n-\nabla_\omega\log\pi_\omega(a_{n,t}|b_{n,t})\cdot w-J(\omega)\right)\right)^2$$

6: **Actor update** $\omega\leftarrow\omega+\alpha\cdot w$
7: **until** convergence

NAC has been applied to SDS by Thomson and Young [32].

Deep Reinforcement Learning (DRL) is a branch of Reinforcement Learning focusing on *function approximation* using *deep Neural Networks*. A Neural Network is a collection of connected neurons, where the output of a neuron is a linear combination of its inputs and the *weights* associated to the neuron, fed into a nonlinear differentiable function such as tanh, called *activation*. A deep Neural Network is made out of several (m) layers of neurons, where information flows from input x to output y according to the following equations, where

h_i is the hidden layer (NN outputs) for layer i and g_i are the activations:

$$\begin{aligned} h_0 &= g_0(W_0 x^T + b_0) \\ h_i &= g_i(W_i h_{i-1}^T + b_i),\ 0 < i < m \\ y &= g_m(W_m h_{m-1}^T + b_m). \end{aligned}$$

The choice of activation functions often reflects the function of a neuron. Some examples are listed below.

- **Hyperbolic tangent function**: $f(x) = \tanh(x)$. Produces an output between -1 and 1 with a soft transition.
- **Logistic**: $f(x) = \frac{1}{1+\exp(-x)}$. Produces an output between 0 and 1 with a soft transition.
- **Rectified Linear Unit (ReLU)**: $f(x) = [x]_+$. In other words, $f(x) = 0$ for negative x, and is $f(x) = x$ for non-negative x. The derivative of ReLU is simpler, making it quicker to train.
- **Softmax**: $f_i(x) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$. This produces a valid probability distribution with soft transitions.
- **Identity function:** $f(x) = x$. This does not restrict or condense the output into any specific region.

Deep Reinforcement Learning methods have been applied to SDS by Li et al. [18] and Williams et al. [34].

Deep Q-network (DQN) is a *value-based approximation* Deep Reinforcement Learning method. The value of $Q(b,a)$ is approximated using weights θ. It uses the Mean Squared Value Error for measuring the prediction error, which is minimised by Stochastic Gradient Descent. The MSVE is calculated after observing a reward r after performing action a in belief state b:

$$\text{MSVE} = \left(r + \gamma \max_{a'} Q_\theta(b', a') - Q_\theta(b, a) \right)^2.$$

There are two problems with a vanilla implementation of this method that make it diverge:

- **States are correlated:** the successive states we encounter in an episode are correlated with each other, pushing the network to learn about currently observed kinds of states while forgetting previous kinds.

- **Targets are non-stationary:** arising from the previous problem, the Q-value target of successive positions is correlated, pushing the network to be biased towards currently observed targets.

We want the network to learn the Q-value purely based on the state-action pair without any temporal bias. To achiever this, we introduce Experience Replay.

Instead of updating the network as experience is gathered, the **Experience Replay (ER)** method puts new experiences in an *experience memory*. Targets from this memory are randomly sampled and used for training after each iteration. This is similar to the idea of the Dyna framework [28], where an internal model is trained from all the observed interactions.

Another improvement that stabilises learning is to only update the network weights after multiple targets have been considered. The update will be the sum of the gradients. Despite these improvements however, the underlying issue with purely *value-based approximations* is that a small change in the values can lead to significant changes in the corresponding greedy policy [22]. This makes Deep Q-network a relatively unstable algorithm in the SDS domain, as explored by Fatemi et al. [4]. DQN is also inefficient in the sense that it requires many dialogues to converge. We will strive to find more stable and sample efficient algorithms.

Advantage Actor Critic (A2C) approximates both the policy function π (actor) and the value function V (critic) with deep Neural Networks. The critic is parametrised with θ and is updated with SGD based on the loss function:

$$L(\theta) = (R_t - V_\theta(s_t))^2 .$$

The policy is parametrised with ω. For an objective function $J(\omega)$, we established before that the update rule for ω is:

$$\nabla J(\omega) = \mathop{\mathbb{E}}_{\pi}[A_\pi(b,a)\nabla_\omega \log \pi(a|b,\omega)] ,$$

where A is the advantage function. We use an estimate of the advantage function:

$$\mathbb{E}A_\pi(a|b) = R_t - V_\theta(s_t),$$

and the objective is defined, as usual, as

$$J(\omega) = V_\omega(s_0).$$

This leads to Algorithm 9.

Algorithm 9 Advantage Actor Critic

Input: policy $\pi(a|b,\omega)$, $V_\theta(b,a)$, learning rate α
Initialise θ, ω, and $V_\theta(\text{terminal}) = 0$
repeat
    Generate episode $\{b_{0:T}, a_{0:T}, r_{0:T}\}$ according to $\pi(\cdot|\cdot,\omega)$
    $R_T \leftarrow 0$
    for $t = T$ downto 0 **do**
        $R_{t-1} \leftarrow r_t + \gamma V_\theta(b_t,\theta)$
        $\nabla J = \nabla J + (R_t - V_\theta(b_t,\theta))\nabla_\omega \log \pi(a_t|b_t,\omega)$
        $\nabla L = \nabla L + \nabla_\theta (R_t - V_\theta(b_t,\theta))^2$
    $\omega \leftarrow \omega + \alpha \nabla J$
    $\theta \leftarrow \theta + \alpha \nabla L$
until convergence

Chapter 3

Method

This project builds on recent breakthroughs in Deep Reinforcement Learning. In particular, we investigate improvements to the actor-critic method. The goal is a stable and sample efficient learning algorithm that performs well on challenging policy optimisation tasks in the Spoken Dialogue Systems domain. Recent advances in DRL apply several methods, including experience replay, truncated importance sampling with bias correction [33], the off-policy Retrace algorithm [20] and trust region policy optimisation [26] to the domain of Atari games. The core of this project is to investigate to which extent these advances are applicable to the dialogue policy optimisation task. To this end, we introduce the Actor Critic with Experience Replay (ACER) algorithm presented in Wang et al. [33], explaining and investigating the steps needed to apply it to SDS. We thus achieve stable and efficient learning, which ultimately allows larger action spaces to be considered. ACER proves to be surprisingly effective, beating the state of the art Neural Network-based dialogue optimiser [27]. This success motivated to extend the project to consider the more challenging domain of the *master action space*. This chapter concludes with a description of modifications to ACER and GP that have been devised to optimise the algorithms for master action space.

3.1 Actor-critic with Experience Replay

In Section 2.3.3, we investigated DRL algorithms Deep Q-network (DQN) and Advantage Actor Critic (A2C). DQN samples its experience from a memory (Experience Replay), thus overcoming the correlated states and targets problem. However, it only estimates the Q function, leading to unstable learning. On the other hand, A2C is an actor-critic method and it estimates both the *value function* and the *policy*. Its targets are calculated from an unbiased, low-variance estimate, leading to stable learning. A2C does not have Experience Replay however, which means that only one update step can be made per iteration, leading

to slower learning. The question arises whether Experience Replay could be added to A2C. For this, we have to derive an off-policy version of A2C, because the policy with which the experience has been collected, μ, is different from the current policy π.

To derive this mathematically, we revisit the definition of the objective function. According to the original definition, we want to maximise the value of the initial state:

$$J(\omega) = V_{\pi(\omega)}(b_0).$$

Another way of expressing the same objective is to maximise the cumulative reward received from the average state [3]. For policy μ, let the occupancy frequency d^μ be defined as:

$$d^\mu(b) = \lim_{t\to\infty} P(b_t = b|b_0, \mu).$$

According to the new definition of $J(\omega)$, V is weighted by d^μ because μ was used to collect the experience. We have

$$J(\omega) = \sum_{b\in\mathbb{B}} d^\mu(b) V_\pi(b).$$

The off-policy version of the Policy Gradient Theorem is used to derive the gradients:

$$\begin{aligned}\nabla_\omega J(\omega) &= \nabla_\omega \left[\sum_{b\in\mathbb{B}} d^\mu(b) \sum_{a\in\mathbb{A}} \pi(a|b) Q_\pi(b,a)\right] \\ &= \sum_{b\in\mathbb{B}} d^\mu(b) \sum_{a\in\mathbb{A}} [\nabla_\omega \pi(a|b) Q_\pi(b,a) + \pi(a|b) \nabla_\omega Q_\pi(b,a)]\end{aligned}$$

The second term is difficult to estimate accurately. However, we can estimate it by zero and omit it. Degris et al. [3] provide justification for this. We have $\nabla_\omega J(\omega) \approx g(\omega)$, where

$$g(\omega) = \sum_{b\in\mathbb{B}} d^\mu(b) \sum_{a\in\mathbb{A}} \nabla_\omega \pi(a|b) Q_\pi(b,a)$$

We encounter the states in proportions according to d^μ just by sampling from the experience memory, so we do not need to estimate it explicitly. Estimating Q_π however is more difficult: the off-policy interactions are gathered according to μ, and we are interested in the Q-function under a different policy, the current policy π. We will look at different methods to estimate Q_π.

Naïvely, we can estimate $Q_\pi(b,a)$ with an importance sampled version of the discounted cumulative reward R, sampled from the replay memory:

$$\nabla J(\omega) \approx \mathbb{E}\left[\left(\prod_{t=0}^{T}\rho_t\right)\left(\sum_{i=0}^{T}\gamma^i r_{t+i}\right)\nabla_\omega \pi(a|b,\omega)|b\sim d^\mu\right].$$

where $\rho_t = \frac{\pi(a_t|b_t)}{\mu(a_t|b_t)}$ are the Importance Sampling (IS) weights. This estimation is unbiased, but suffers from very high variance [33]. This is because it multiplies potentially unbounded IS weights for an entire episode. This multiplication either results in a very small value (*vanishing weight*) or a very large value (*exploding weight*). To achieve stable learning, we need a different method that considers state-action pairs in isolation, applying only one IS weight for each.

Starting again from the off-policy version of the Policy Gradient Theorem:

$$\begin{aligned}
g(\omega) &= \sum_{b\in\mathbb{B}} d^\mu(b)\sum_{a\in\mathbb{A}}\nabla_\omega\pi(a|b)Q_\pi(b,a)\\
&= \mathbb{E}\left[\sum_{a\in\mathbb{A}}\nabla_\omega\pi(a|b)Q_\pi(b,a)|b\sim d^\mu\right]\\
&= \mathbb{E}\left[\sum_{a\in\mathbb{A}}\mu(a|b)\frac{\pi(a|b)}{\mu(a|b)}\frac{\nabla_\omega\pi(a|b)}{\pi(a|b)}Q_\pi(b,a)|b\sim d^\mu\right]\\
&= \mathbb{E}\left[\rho(a|b)\nabla_\omega\log\pi(a|b)Q_\pi(b,a)|b\sim d^\mu, a\sim\mu(\cdot|b)\right],
\end{aligned}$$

where $\rho(a|b) = \frac{\pi(a|b)}{\mu(a|b)}$ are the Importance Sampling (IS) weights. As with the on-policy gradient, we can again use the advantage function in place of the Q-function for an unbiased estimate with a lower variance:

$$g(\omega) = \mathop{\mathbb{E}}_{\mu}\left[\rho(a|b)\nabla_\omega\log\pi(a|b)A_\theta\right].$$

The advantage function is approximated in the vanilla version of A2C as $R_t - V(b_t,\theta)$. We cannot use this here as the cumulative reward R has been gathered according to the old policy μ and may not be representative of the current cumulative reward we can obtain following π. Instead, we approximate the advantage function as $r_t + \gamma V(b_{t+1},\theta) - V(b_t,\theta)$. This also applies to the loss of the critic, which used to be $(R_t - V(b_t,\theta))^2$. A2C with ER uses $(r_t + \gamma V(b_{t+1},\theta) - V(b_t,\theta))^2$ instead. Algorithm 10 shows the complete algorithm.

Algorithm 10 Advantage Actor Critic with Experience Replay

Input: policy $\pi(a|b,\omega)$, $V_\theta(b,a)$, learning rate α
Initialise $\theta, \omega, V_\theta(\text{terminal}) = 0$
repeat
    Generate an episode according to $\pi(\cdot|\cdot,\omega)$ and save to replay memory
    for $i = 0, 1, ..., \text{batch\_size}$ **do**
        Sample episode $\{b_{0:T}, a_{0:T}, r_{0:T}\}$ from replay memory, with old policy $\mu(a|b)$
        for $t = T$ downto 0 **do**
            $R_{t-1} \leftarrow r_t + \gamma V(b_t, \theta)$
            $\rho(a_t|b_t) \leftarrow \frac{\pi(a_t|b_t)}{\mu(a_t|b_t)}$
            $\nabla J = \nabla J + \rho(a_t|b_t)\left(r_t + \gamma V(b_{t+1}, \theta) - V(b_t, \theta)\right)\nabla_\omega \log \pi(a_t|b_t, \omega)$
            $\nabla L = \nabla L + \nabla_\theta \left(r_t + \gamma V(b_{t+1}, \theta) - V(b_t, \theta)\right)^2$
    $\omega \leftarrow \omega + \alpha \nabla J$
    $\theta \leftarrow \theta + \alpha \nabla L$
until convergence

3.2 Lambda returns

We saw how using the unbiased estimator

$$Q_\pi(b_t, a_t) \approx \sum_{i=t}^{T} \gamma^i r_i$$

resulted in high variance, due to the IS weight that has to be calculated for the entire episode. The estimation

$$Q_\pi(b_t, a_t) \approx r_t + \gamma V(b_{t+1}, \theta)$$

only requires a single IS weight. However, this estimation is biased: the value function update of the current state is based on the current estimate of the value function for the next state. For example, if an interaction results in a higher final reward than expected by the function approximators, then the final state will be updated to reflect this. All other states however only update based on the estimate of the next state. For this reason, a similar interaction needs to happen again for this update rule to propagate the update to two states before the final state. Continuing with this, we get that an interaction needs to happen as many times as its length to propagate the update to the initial state. This leads to slow convergence or no convergence at all.

It is possible to combine both methods and create an estimator that trades off bias and variance according to a parameter λ. Degris et al. [3] estimate Q_π as:

$$Q_\pi(b_t, a_t) \approx R_t^\lambda, \text{ where}$$

$$R_t^\lambda = r_t + (1-\lambda)\gamma V(b_{t+1}) + \lambda\gamma\rho_{t+1}R_{t+1}^\lambda.$$

Setting λ to 0 results in an equivalent estimation to what we had before: $Q_\pi(b_t, a_t) \approx r_t + \gamma V(b_{t+1}, \theta)$, with a low variance but high bias. Conversely, setting λ to 1 results in high variance as many IS weights will be producted. This has the advantage of propagating the final reward further to the starting state so suffers from lower bias. A carefully hand-selected λ could bring the best of both worlds.

It is important to note that this approach has some shortcomings. First, it is required to set λ ahead of time to represent a good trade-off. Second, even when λ is 0 to reduce variance as much as possible, occasional large IS weights introduce the variance, and they can still cause instability [33].

3.3 Retrace

The Retrace algorithm (Munos et al. [20]) attempts to estimate the current Q-function from off-policy interactions in a safe and efficient way, with small variance. Throughout this discussion, we call a method *safe* if its estimate of Q^π can be proven to converge to Q^π. We introduce Retrace together with a set of related methods to illustrate the benefits of Retrace. These methods all fit a general framework in which we derive our updated estimate of the Q-function, Q^{ret}, from our current baseline estimate, Q, according to an error term that we compute based on state-action trajectories sampled from the replay memory.

$$Q^{ret}(b,a) = Q(b,a) + \mathop{\mathbb{E}}_{\mu}\left[\sum_{t\geq 0}\gamma^t\left(\prod_{s=1}^{t}c_s\right)(r_t + \gamma V(b_{t+1}) - Q(b_t,a_t))\right]. \tag{3.1}$$

The methods that stem from this framework differ only in their definition of c_s, which will be given later.

This framework introduces changes to the actor-critic model. Instead of approximating V and π with Neural Networks and estimating Q in a closed-form equation to compute the update targets, we estimate π and Q with Neural Networks. In other words, we change the critic from estimating V to estimating the Q-function with a Neural Network. We compute V

from π and Q:

$$V(b) = \mathop{\mathbb{E}}_{\pi} Q(b,\cdot) = \sum_a \pi(a|b)Q(b,a).$$

The framework thus defines the update targets Q^{ret} from the output of the critic Neural Network Q, with the help of coefficients c_s. We define $(\prod_{s=1}^{t})c_s = 1$ for $t = 0$. Following the idea presented in Sutton and Barto [29], we call c_s the *eligibility traces*. Intuitively, they control the weight with which prediction errors errors are considered: the immediate prediction error $r_0 + \gamma V(b_1) - Q(b_0,a_0)$ is considered with a weight of 1. The further away in the state-action trajectory an error occurs, the lower the weight with which we consider it.

If the current baseline estimation is $Q(b,a) = 0$ for every state-action pair and $c_s = \rho(b_s,a_s)$, then Q_{ret} will be equivalent to the importance sampled cumulative reward estimator. On the other hand, if $c_s = 0$, then Q^{ret} is equivalent to the TD-error estimation. This framework can thus be seen as a method that bridges the TD-error estimation and the cumulative return estimation methods. In this way, it is similar to the *Lambda returns* method: its two extremes are the same high-variance and low-bias cumulative return, and the low-variance and high-bias TD error estimation. However, the bias-variance trade-off is now controlled by the *eligibility traces* c_s instead of a single constant λ. This will give us greater flexibility. We will present three methods of setting c_s.

Importance Sampling (IS): $c_s = \rho(a_s|b_s)$ [24, 9]. When IS is used, the estimation will have no bias and it will yield Q^{π} for Q' in expectation. As seen before, if the baseline $Q = 0$, then this is the same as the cumulative return estimation method. Setting Q can be seen as a variance reduction method [20]. Still, the product of IS weights introduces too much variance.

Off-policy $Q^{\pi}(\lambda)$: $c_s = \lambda$ [10]. This approach uses no IS, and only discounts traces according to a single constant. As a results, it has low variance. Surprisingly, this method converges, but only under a strict requirement that π and μ are sufficiently close to each other. We cannot guarantee this so this method is not safe for us to use.

Tree-backup: $c_s = \lambda\pi(a_s|b_s)$ [24]. In addition to the *off-policy $Q^{\pi}(\lambda)$ method*, this method also multiplies the traces by the target policy probabilities. This method still has low variance and is also safe as the estimation converges to Q^{π}. However it is not efficient in the near on-policy case where π and μ become similar. This usually happens further in the training process, as a result of eg. ε being reduced in the ε-greedy exploration. In such a case, multiplying all the traces with the target policy probabilities cuts them unnecessarily, reducing their contribution to the update target.

Retrace(λ): $c_s = \lambda \min(1, \rho(a_s|b_s))$. Ideally, we need a method that is safe, has low variance and is as efficient as possible. Intuitively, safety and efficiency have been a trade-off in the previous methods, in the sense that a safe method (*tree-backup*) cut the traces too much in the near on-policy case to be efficient, and an efficient method (*off-policy* $Q^\pi(\lambda)$) was only safe under strict conditions. Retrace solves this trade-off by setting the traces "dynamically", based on the Importance Sampling weights. In the near on-policy case, it is efficient as IS weights will be about 1, preventing the traces from vanishing. It has low variance because the IS weights are clipped at 1. It is also safe for any π and μ. The goal of this discussion is limited to conveying the intuition behind *Retrace*, but a full proof of safety is available in Munos et al. [20].

3.3.1 Computational cost

Let us investigate the computational cost of deriving Q^{ret} from Q in a naïve way. For each episode sampled from the replay memory, and for each state-action pair, we need to visit the remaining part of the episode to calculate the expectation of errors under μ according to Equation 3.1. This quadratic element of the computational cost can be reduced to a linear one by deriving Q^{ret} in a recursive way. For an episode trajectory $b_{1:T}, a_{1:T}$ sampled from the replay memory, Equation 3.1 becomes:

$$\begin{aligned}
Q^{ret}(b_i,a_i) &= Q(b_i,a_i) + \sum_{t\geq 0}^{T-i} \gamma^t \left(\prod_{s=1}^{t} c_{i+s}\right)\left(r_{i+t} + \gamma V(b_{t+i+1}) - Q(b_{i+t},a_{i+t})\right) \\
&= Q(b_i,a_i) + r_i + \gamma V(b_{i+1}) - Q(b_i,a_i) \\
&\quad + \gamma c_{i+1} \sum_{t\geq 0}^{T-i-1} \gamma^t \left(\prod_{s=1}^{t} c_{i+1+s}\right)\left(r_{i+1+t} + \gamma V(b_{t+i+2}) - Q(b_{i+1+t},a_{i+1+t})\right) \\
&= r_i + \gamma V(b_{i+1}) + \gamma c_{i+1}\left(Q^{ret}(b_{i+1},a_{i+1} - Q(b_{i+1},a_{i+1})\right).
\end{aligned}$$

We will use this more computationally efficient, recursive formulation of Q^{ret}.

3.4 Architecture of our actor-critic Neural Networks

Now that we introduced all the prerequisites, we direct our attention at designing the Neural Networks for our actor-critic. On top of the input of the belief state, we build two hidden layers, h_1 and h_2. The NN outputs functions π and Q; these will be two "heads" of the NN. This means that both hidden layers h_1 and h_2 are shared between the predictor of Q and the predictor of π. Weight sharing can be beneficial as it reduces the number of parameters to

train. In our example, we expect there to be a strong correlation between π and Q: the more likely we are to choose an action according to π, the higher we expect its Q-value to be. This situation lends itself well for weight sharing. The architecture is illustrated in Figure 3.1.

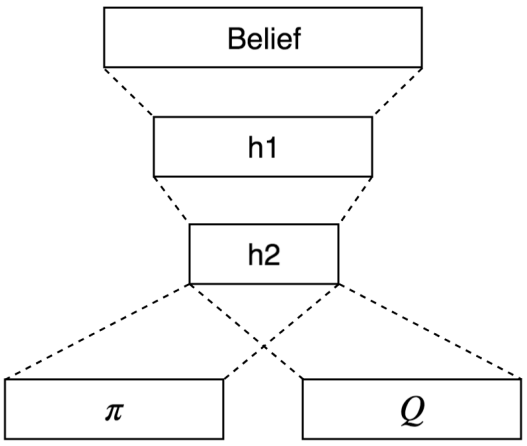


Fig. 3.1 Architecture of our actor-critic neural network [33].

The layers are fully-connected between the input (belief) layer and h_1, as well as between h_1 and h_2. The activation function for theses layers is the Rectified Linear Unit (ReLU). h_2 is fully-connected to π and Q too. The activation function for π is *softmax*, which converts the inputs to a probability distribution with values between 0 and 1, summing up to 1. There is no activation function for the output Q, as we want it to have an unlimited range, both from above and below (as rewards can be negative). Q is thus a linear combination of h_2 and the weights, plus a baseline constant per action. For the *Cambridge Restaurants* domain, the *belief state* is represented by a 268-dimensional vector, as illustrated in Figure 3.2. This is the input of the Neural Network. Layer h_1 consists of 130 neurons and h_2 has 50 neurons. The idea is to force the NN to encode all information about the *belief state* relevant to π and Q in 50 neurons, thereby learning a more stable mapping. The output vectors π and Q have the dimensionality of the action space. Initially, we experiment with the summary action space, which has 15 actions.

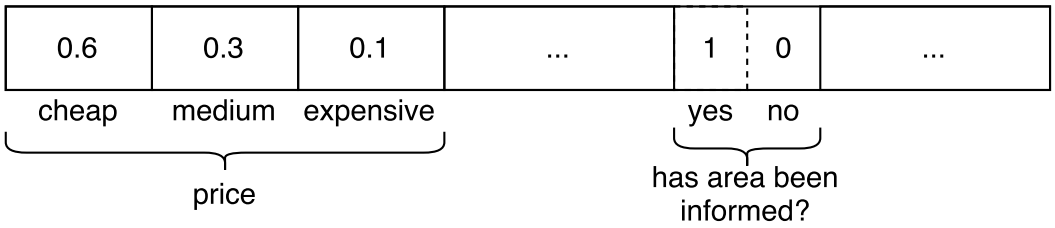


Fig. 3.2 Example of *belief state* representation. Contains *continuous* fields such as *area* are represented as a probability distribution, while *discrete* fields such as *has are been informed* are represented as a binary value and its negated form.

As before, we will denote the weights used by π and Q as ω and θ, respectively. We note however that most weights are shared between ω and θ. The training process follows the following steps:

1. Sample a batch of random experiences from experience memory.
2. Compute Importance Sampling weights.
3. Calculate Q^{ret} according to Retrace.
4. The advantage function is calculated as $A_\pi(b,a) = Q^{ret}(b,a) - \sum_a \pi(a|b)Q(b,a)$.
5. Calculate the actor gradient as $g(\omega) = \sum_{a,b} \left(\rho(a|b)\nabla_\omega \log \pi(a|b)A_\pi(b,a)\right)$.
6. Calculate the critic gradient as $\nabla_\theta \sum_{a,b}(Q^{ret}(b,a) - Q_\theta(b,a))^2$.
7. Update weights θ and ω based on the gradients calculated and the step-size α.

We can see from this training pipeline that Q^{ret} two serves two purposes simultaneously. First, it is used as the update target of the critic. Training the critic will provide an improved baseline for future calculations of Q^{ret}, as well as more accurate calculations of the V-function (derived from π and Q). Second, Q^{ret} is used to estimate the advantage function for the actor gradient calculation. While we could also use the output of the critic, Q, to perform this estimation, using Q^{ret} compares favourably to that approach, as it is updated by the sampled returns. Q^{ret} being an efficient estimator that has low bias and variance helps both of these use cases.

3.5 Importance Weight Truncation with Bias Correction

Currently, we calculate the policy gradient as:

$$g(\omega) = \mathop{\mathbb{E}}_{b_t \sim d^\mu, a_t \sim \mu} \left[\rho(a_t|b_t)\nabla_\omega \log \pi(a_t|b_t)A_\pi(b_t,a_t)\right],$$

where the expectation is taken over the replay memory, and $\rho(a_t|b_t) = \frac{\pi(a_t|s_t)}{\mu(a_t|s_t)}$. An issue with this approximation is that the Importance Sampling weights $\rho(a_t|b_t)$ are potentially unbounded, introducing some variance. To solve this problem, we clip the IS weights from above by a constant c: $\overline{\rho}(a_t|b_t) = \min\{c, \rho((a_t|b_t))\}$. We can split the equation into two parts, one involving the truncated IS weight, and the other the residual. We need to also estimate the residual, otherwise we introduce bias in the gradient estimation. We call the residual the *bias correction* term.

$$\begin{aligned} g(\omega) &= \mathop{\mathbb{E}}_{b_t\sim d^{\mu}, a_t\sim\mu}[\rho(a_t|b_t)\nabla_\omega \log\pi(a_t|b_t)A_\pi(b_t,a_t)] \\ &= \mathop{\mathbb{E}}_{b_t\sim d^{\mu}}\Big[\mathop{\mathbb{E}}_{a_t\sim\mu}\overline{\rho}(a_t|b_t)\nabla_\omega \log\pi(a_t|b_t)A_\pi(b_t,a_t) \\ &\qquad + \mathop{\mathbb{E}}_{a_t\sim\mu}[\rho(a_t|b_t)-c]_+\nabla_\omega \log\pi(a_t|b_t)A_\pi(b_t,a_t)\Big], \end{aligned}$$

where $[\cdot]_+ = \max(0,\cdot)$. The weight of the bias correction term, $[\rho(a_t|b_t)-c]_+$, can still be unboundedly large. This can be solved by sampling the action from the distribution π rather than μ [33]. In this case we multiply by $\frac{\mu(a_t|b_t)}{\pi(a_t|b_t)} = 1/\rho(a_t|b_t)$ to correct for the different sampling. The equation of the gradient becomes:

$$\begin{aligned} g(\omega) = \mathop{\mathbb{E}}_{b_t\sim d^{\mu}}\Bigg[&\mathop{\mathbb{E}}_{a_t\sim\mu}\overline{\rho}(a_t|b_t)\nabla_\omega \log\pi(a_t|b_t)A_\pi(b_t,a_t) \\ &+ \mathop{\mathbb{E}}_{a\sim\pi}\left[\frac{\rho(a|b_t)-c}{\rho(a|b_t)}\right]_+\nabla_\omega \log\pi(a|b_t)A_\pi(b_t,a)\Bigg]. \end{aligned}$$

There are two key advantages of this formulation:

- The bias correction term ensures that the estimate of the gradient remains unbiased.
- The bias correction term is only active when $\rho(a|b) > c$, and otherwise the formulation is equivalent to what we had before.

This means we can tune c to a high enough value to only modify the handling of belief-action pairs with a high-variance Importance Sampling. For these cases, when $\rho(a|b) > c$, the variance of the estimation is significantly reduced: the base term's weight is clipped by c, and the bias correction weight, $\left[\frac{\rho(a|b_t)-c}{\rho(a|b_t)}\right]_+$, falls between 0 and 1, both being bounded.

To apply this method, called the *truncation with bias correction trick* by Wang et al. [33], we have to overcome a problem with the advantage function estimation. Before, we estimated $A_\pi(b,a) = Q^{ret}(b,a) - \sum_a \pi(a|b)Q(b,a)$ for belief-action pairs that we sampled from the replay memory. For the bias correction term however, only the belief is sampled

from the memory, and all the actions are considered and weighted by the current policy π. Due to the way Q^{ret} is formulated, it learns from rewards, and only learns belief-action pairs that have been visited and sampled from the replay memory. Thus the estimation is not available for the bias correction term, so we use the output of the Neural Network, Q, to estimate the advantage function for that term: $A'_\pi(b,a) = Q(b,a) - \sum_a \pi(a|b)Q(b,a)$.

3.6 Trust Region Policy Optimisation

As discussed in Section 2.3.3, restricting the training step-size according to the Eucledian distance metric on the parameter space has some shortcomings. More specifically, small changes in the parameter space can lead to erratic changes in the output policy. This could lead to unstable learning or a learning rate too small for quick convergence. We discussed Natural Actor Critic, and the computation of the *natural gradient* that restricts the step size in the policy space. We saw that its training procedure was very different from the off-policy gradient estimation methods presented here. We also saw that it required *compatible function approximation* to avoid an expensive Fisher information matrix computation. For these reasons, we cannot directly compute the natural gradient for ACER. We can, however, attempt to modify NAC to work off-policy, which has been attempted as a side project, with little success compared to ACER.

Instead of computing the exact natural gradient, we can approximate it. For the natural gradient, the distance metric tensor is the Fisher information matrix:

$$(G_\omega)_{ij} = \mathbb{E}\left[\frac{\delta \log p(x|\omega)}{\delta \omega_i}\frac{\delta \log p(x|\omega)}{\delta \omega_j}\right].$$

It can be shown [16] that

$$d\omega^T G_\omega d\omega \approx \mathrm{KL}(\pi(\cdot|b,\omega)||\pi(\cdot|b,\omega+d\omega)),$$

Where KL is the Kullback–Leibler divergence. It is defined for discrete probability distributions as

$$\mathrm{KL}(P||Q) = \sum_i P(i)\log\frac{P(i)}{Q(i)}.$$

Thus, instead of directly restricting the learning step-size with the *natural gradient* method, we can approximate the same method by restricting the Kullback–Leibler divergence between the current policy π parametrised by ω, and the updated policy parametrised by $\omega + \alpha \cdot \nabla_\omega J$. More specifically, we will update the parameters with a value as close as possible to $\omega + \alpha \cdot \nabla_\omega J$, such that the new policy is within a constant distance of the old policy,

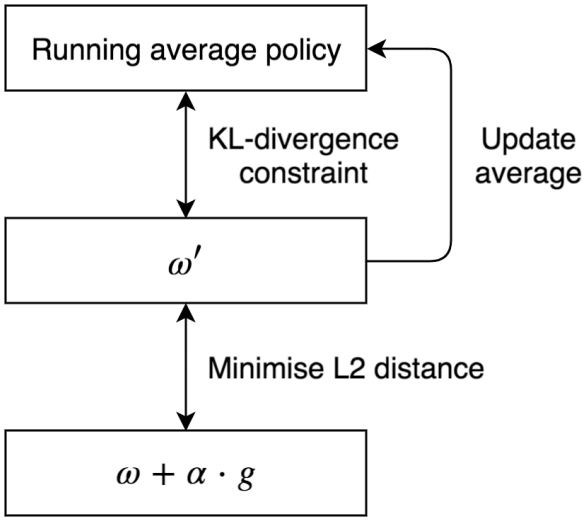


Fig. 3.3 Trust Region Policy Optimisation.

where the distance metric is KL divergence. This method is called *Trust Region Policy Optimisation (TRPO)*, introduced by Schulman et al. [26]. Their method however relies on repeated computations of Fisher-vector products for each update, which can be prohibitively expensive. Wang et al. [33] introduce an efficient TRPO method that we will adopt instead. Our description of the method will largely follow theirs with additional explanations, however, it has been adapted to our discrete action-space SDS domain, and unnecessary steps have been removed.

To begin with, Wang et al. [33] propose that the KL-divergence to the updated policy should be measured not from the current policy, but from a separate average policy instead. This stabilises the algorithm by preventing it from gaining momentum in a specific direction. Instead, it is restricted to stay around a more stable average policy π_a. TRPO with the average policy is illustrated in Figure 3.3. The average policy is parametrised with ω_a, where ω_a represents a running average of all previous policy parameters. It is updated *softly* after each learning step as:

$$\omega_a \leftarrow \beta\omega_a + (1-\beta)\omega.$$

β is a hyperparameter that controls the amount of history to maintain in the average policy. A too low value close to zero makes the average policy forget the history very quickly, reducing the effect of calculating the distances from the average policy instead of the current one. A too high value close to one will prevent the average policy to adjust to the current policy, or slows this adjustment process down. However, the weight given to the history is reduced exponentially as the learning steps progress. A setting of β between 0.95 and 0.99 turned out to work best in our case.

Next, we define our goal for TRPO. We work with our previous definition of the policy gradient:

$$g(\omega) = \mathop{\mathbb{E}}_{b_t \sim d^{\mu}} \Bigg[\mathop{\mathbb{E}}_{a_t \sim \mu} \overline{\rho}(a_t|b_t) \nabla_{\omega} \log \pi(a_t|b_t) \left(Q^{ret}(b,a) - \sum_a \pi(a|b) Q(b,a) \right) + \mathop{\mathbb{E}}_{a \sim \pi} \left[\frac{\rho(a|b_t) - c}{\rho(a|b_t)} \right]_+ \nabla_{\omega} \log \pi(a|b_t) \left(Q(b,a) - \sum_a \pi(a|b) Q(b,a) \right) \Bigg].$$

TRPO can be formulated as an optimisation problem, where we aim to find z than minimises the L2-distance between z and the vanilla gradient g. This is a quadratic minimisation. In addition, for reasons that will become clear later, our aim is for the divergence constraint to be formulated in a linear way. Since z will be used for the parameter update, we have that $\omega' = \omega + \alpha z$, where ω' denotes the updated parameters. We can approximate the KL divergence after the policy update using a first-order Taylor expansion:

$$\mathrm{KL}\left[\pi(\cdot|\omega_a)||\pi(\cdot|\omega')\right] = \mathrm{KL}\left[\pi(\cdot|\omega_a)||\pi(\cdot|\omega)\right] + \nabla_{\omega}\mathrm{KL}\left[\pi(\cdot|\omega_a)||\pi(\cdot|\omega)\right]^T \cdot \alpha z.$$

So the increase in KL divergence in this step is

$$\nabla_{\omega}\mathrm{KL}\left[\pi(\cdot|\omega_a)||\pi(\cdot|\omega)\right]^T \cdot \alpha z.$$

We can constrain this increase to be small by setting δ, such that

$$\nabla_{\omega}\mathrm{KL}\left[\pi(\cdot|\omega_a)||\pi(\cdot|\omega)\right]^T \cdot z \leq \delta,$$

where the learning rate α is left out, since it is a constant and can be incorporated into δ. Letting $k = \nabla_{\omega}\mathrm{KL}\left[\pi(\cdot|\omega_a)||\pi(\cdot|\omega)\right]$, the optimisation problem with linearised KL divergence constrain is [33]:

$$\begin{aligned} \underset{z}{\text{minimize}} \quad & \frac{1}{2}||g(\omega) - z||_2^2 \\ \text{subject to} \quad & k^T z \leq \delta \end{aligned}$$

Since the constraint is linear, the overall optimisation problem reduces to a simple quadratic programming problem. We separate it into two cases. If $k^T g \leq \delta$, that is, the KL divergence constraint is satisfied by g itself, then the solution is $z = g$. Otherwise, the solution will be at the edge of the convex solution space. In other words, if $k^T g > \delta$, we can rewrite the

constraint as an equality:

$$\begin{aligned} \underset{z}{\text{minimize}} \quad & \frac{1}{2}||g(\omega)-z||_2^2 \\ \text{subject to} \quad & k^T z = \delta \end{aligned}$$

In this case, since $k^T z - \delta = 0$, we can define the objective slightly differently, by subtracting λ times this value. The objective becomes

$$\mathbb{L}(\lambda, z) = \frac{1}{2}||g(\omega)-z||_2^2 - \lambda(k^T z - \delta).$$

We call this function the *Lagrangian* of the constrained problem, and λ the *Lagrangian multiplier*. At a solution, the derivative of $\mathbb{L}$ with respect to z will be 0, as it is a local minimum. The derivative with respect to λ will also be 0, as $k^T z - \delta = 0$. These conditions are known as the Karush–Kuhn–Tucker (KKT) conditions, and they are necessary conditions for the optimum of the constraint satisfaction problem. We thus have

$$\begin{aligned} \frac{\partial L}{\partial \lambda} &= -k^T z + \delta &&= 0 \\ \frac{\partial L}{\partial z} &= (z-g) - \lambda k &&= 0, \end{aligned}$$

from which we can derive

$$\begin{aligned} \delta &= k^T z \\ z &= -\lambda k + g \\ \delta &= -\lambda ||k||_2^2 + k^T g \\ \lambda &= \frac{k^T g - \delta}{||k||_2^2} \\ z &= g - \frac{k^T g - \delta}{||k||_2^2} k. \end{aligned}$$

In case $k^T g \leq \delta$, $\frac{k^T g - \delta}{||k||_2^2} \leq 0$, so for the general case,

$$z = g - \max\{0, \frac{k^T g - \delta}{||k||_2^2} k\}.$$

As Wang et al. [33] point out, this closed-form solution to the problem has an intuitive interpretation. If the KL constraint is satisfied, there is no change to the gradient. Otherwise, the gradient is scaled back in the direction of k, which reduces the rate of change of the KL divergence between the updated policy and the average policy.

3.7 Summary of ACER

Actor Critic with Experience Replay (ACER) is the result of all methods presented in this chapter. When on-policy, it is a modified version of A2C. Both use Experience Replay and sample from their memories to achieve high sample efficiency. The difference between them is that ACER additionally employs Trust Region Policy Optimisation, and that it uses a Q-function estimator instead of a V-function estimator as the critic. When off-policy, it uses Truncated Importance Sampling with Bias Correction [33] to reduce the variance of IS weights without adding bias. The Retrace algorithm is used to compute the targets based on the observed rewards in a safe, efficient way, with low bias and variance. When complete, the training pipeline consists of the steps below. We will investigate the effect of various hyperparameters and how to set them in Section 4.5.

1. Sample a batch M of random dialogues from experience memory.
2. Compute Importance Sampling weights.
3. Calculate Q^{ret} according to Retrace: for each sampled dialogue,

$$Q^{ret}(b_i,a_i) = r_i + \gamma V(b_{i+1}) + \gamma c_{i+1}\left(Q^{ret}(b_{i+1},a_{i+1} - Q(b_{i+1},a_{i+1})\right).$$

4. The advantage function is calculated as $A_\pi(b,a) = Q^{ret}(b,a) - \sum_a \pi(a|b)Q(b,a)$, the advantage function for the bias correction term is $A'_\pi(b,a) = Q(b,a) - \sum_a \pi(a|b)Q(b,a)$.
5. Calculate the actor gradient as

$$g(\omega) = \sum_{(a_t,b_t)\in M}\Bigg[\overline{\rho}(a_t|b_t)\nabla_\omega \log\pi(a_t|b_t)A_\pi(b_t,a_t) + \sum_{a\in\mathbb{A}}\pi(a|b_t)\left[\frac{\rho(a|b_t)-c}{\rho(a|b_t)}\right]_+ \nabla_\omega\log\pi(a|b_t)A'_\pi(b_t,a)\Bigg],$$

   for a constant c defining the truncation threshold for IS weights.
6. Apply TRPO on the gradient by calculating the closed-form solution z to the optimisation problem. For $k = \nabla_\omega \mathrm{KL}[\pi(\cdot|\omega_a)||\pi(\cdot|\omega)]$,

$$z = g - \max\{0, \frac{k^T g - \delta}{||k||_2^2}k\},$$

   for a constant δ controlling the allowed rate of KL divergence.

7. Calculate the critic gradient as $\nabla_\theta \sum_{a,b}(Q^{ret}(b,a) - Q_\theta(b,a))^2$.

8. Update weights ω and θ based on z and the critic gradient, respectively, based on the learning rate α.

9. Update the average policy weights $\omega_a \leftarrow \beta\omega_a + (1-\beta)\omega$, for a constant β controlling the forget rate.

This training algorithm is presented in pseudocode (Algorithm 12), and is called from the master ACER algorithm (Algorithm 11). A hyperparameter batch_size controls the number of dialogues considered for a training step, and n controls the number of training steps for each new dialogue gathered.

Algorithm 11 ACER master algorithm

Input: policy $\pi(a|b,\omega)$, $Q_\theta(b,a)$, hyperparameters $\alpha,\beta,\gamma,\delta$
Initialise θ,ω,ω_a, and $Q_\theta(\text{terminal})=0$
repeat
    Generate episode $\{b_{0:T},a_{0:T},r_{0:T}\}$ according to ε-greedy using $\pi(\cdot|\cdot,\omega)$
    Save generated episode, along with values of $\pi(\cdot|\cdot,\omega)$ (to be used for IS weights)
    for $i=1$ to n **do**
        Sample a subset of replay memory, M, of size batch_size
        Call training algorithm (Algorithm 12) with $\{M,\theta\omega,\omega_a,\pi,Q,\alpha,\beta,\gamma,\delta\}$
until convergence

Algorithm 12 ACER training algorithm

Input: $\{M,\theta\omega,\omega_a,\pi,Q,\alpha,\beta,\gamma,\delta\}$
Initialise $g=0,\ d\theta=0$
for each dialogue $\{b_{1:N},a_{1:N},r_{1:N},\mu\}$ in M **do**
    for $1=N$ to 1 **do**
        $\rho_t \leftarrow \frac{\pi(a_t|b_t,\omega)}{\mu(a_t|b_t)}$
        $V(b_t)\leftarrow\sum_a Q_\theta(b_t,a)\pi(a|b_t,\omega)$
        $Q^{ret}\leftarrow r_t+\gamma Q^{ret}$
        $A_\pi(b_t,a_t)\leftarrow Q^{ret}-V(b_t)$
        $A'_\pi(b_t,a_t)\leftarrow Q_\theta(b_t,a_t)-V(b_t)$
        $\overline{\rho}_t\leftarrow min(r,\rho_t)$
        $B\leftarrow\sum_{a\in\mathbb{A}}\pi(a|b_t,\omega)\left[\frac{\rho(a|b_t,\omega)-c}{\rho(a|b_t,\omega)}\right]_+\nabla_\omega\log\pi(a|b_t,\omega)A'_\pi(b_t,a)$
        $g\leftarrow g+\overline{\rho}_t\nabla_\omega\log\pi(a_t|b_t,\omega)A_\pi(b_t,a_t)+B$ ▷ IS truncation plus bias correction
        $d\theta\leftarrow d\theta-\nabla_\theta(Q^{ret}-Q_\theta(b_t,a_t))^2$
        $Q^{ret}\leftarrow\overline{\rho}_t(Q^{ret}-Q_\theta(b_t,a))+V(b_t)$
$k\leftarrow\nabla_\omega\text{KL}[\pi(\cdot|\omega_a)||\pi(\cdot|\omega)]$
$z\leftarrow g-\max\{0,\frac{k^Tg-\delta}{||k||_2^2}k\}$
$\omega\leftarrow\omega+\alpha\cdot z$
$\theta\leftarrow\theta+\alpha\cdot d\theta$
$\omega_a\leftarrow\beta\omega_a+(1-\beta)\omega$

3.8 Master actions for ACER

In Section 3.4, we discussed the architecture for ACER NNs when applied to the summary action space. The outputs π and Q were produced directly from the two hidden layers h_1 and h_2. As an extension to this project, because we found that ACER substantially increased the learning rate, we also applied it on the master action space. However, to make this efficient, the NNs need to be redesigned.

As discussed in Section 2.1.4, there are 8 informable slots of an entity, each with a binary choice on whether we inform on it. Thus, a single inform action makes up $2^8 = 256$ separate master actions, only differing in what they inform on. We want to incorporate the fact that these actions are very similar into the design of our NNs. We achieve this by breaking the policy π into a *summary* policy π_s, corresponding to the 15-dimensional summary action space, and a *payload* policy π_p, corresponding to the 256 choices of the *payload* of an inform action. We break the Q function up similarly into Q_s and Q_p. We reconstruct the 1035-dimensional master policy π and master Q-function Q as follows: for each summary action A,

- If A does not have a payload (ie. is not an inform action), append the corresponding summary values from π_s and Q_s onto π and Q.
- Otherwise, for each payload P of the 256 possible choices, append $\pi_s(A) \cdot \pi_p(P)$ to π. This is because the probability of choosing action A with payload P is modelled as the product of the probability of choosing A and that of choosing P. For each P, we also append $Q_s(A) + Q_p(P)$ to Q, allowing the payload network to learn an offset of Q achieved by choosing a particular payload.

The complete NN architecture is illustrated in Figure 3.4. It is important to note that only the architecture of the NNs is changed and the training algorithm is unchanged. In fact, the NNs are treated as a black box by ACER. These outputs being 1035-dimensional for master action space, training ACER in this setting will be informative about the applicability of this algorithm on larger action spaces.

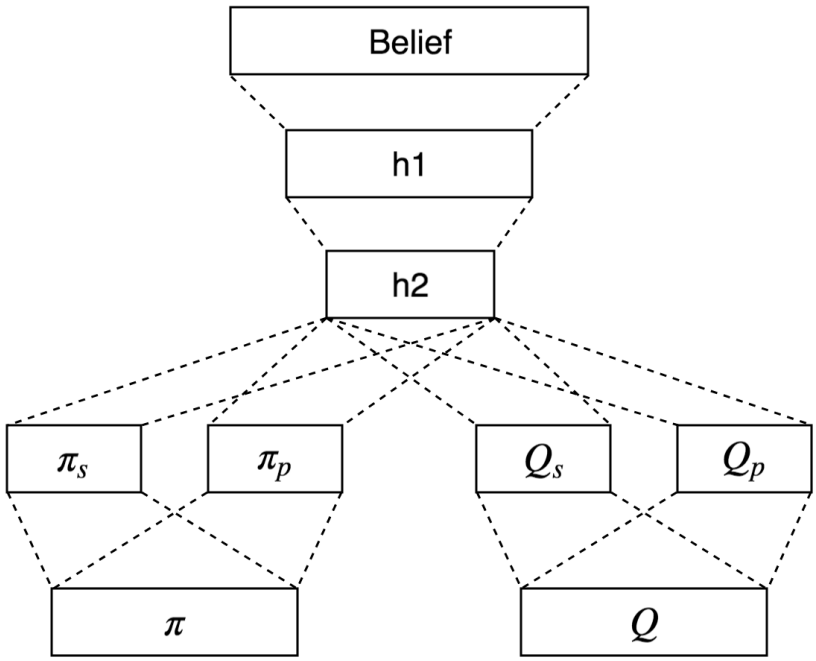


Fig. 3.4 Architecture of the actor-critic neural network for the master action space.

3.9 Master actions for GP

Similarly to ACER, the GP method needs to be adjusted before we deploy it on master action space. In Section 2.3.3, we introduced the kernel function:

$$k((b,a),(b',a')) = \langle b,b' \rangle \delta(a,a').$$

We recall that the kernel function defines our a priori belief of the covariance between any two belief-action pairs. This kernel is a multiplication of a scalar product of the beliefs and a Kronecker delta on the actions. The latter has the effect that any two different actions are considered completely independent. While this might be a good approximation for summary actions, a more elaborate action kernel is required for master actions. This could introduce the idea that two inform actions with slightly different payloads are expected to have similar results on the same belief state, thus showing higher covariance.

Our new action kernel returns 0 for actions a and a' that stem from different *summary* actions. Otherwise, a and a' are the same inform action with differing payloads. In this case, we calculate the kernel based on the cosine similarity of the two payloads, treating the payloads as vectors describing the sets of slots to inform on. Let us call a^s and a^p the summary action and the payload corresponding to a. a^p is represented as a vector where

each entry is either 0 or 1, depending on whether the corresponding slot is informed on. Writing $\overline{a^p} = \frac{a^p}{||a^p||^{\frac{1}{2}}}$ for the normalised version of the payload vector, the kernel becomes $k((b,a),(b',a')) = \langle b,b'\rangle \delta(a^s,a'_s)\langle \overline{a^p}, \overline{a'^p}\rangle$. Furthermore, the Kronecker delta function can be implemented as a scalar product between one-hot vector representations of the summary actions a^s. The final kernel is

$$k((b,a),(b',a')) = \langle b,b'\rangle \langle a^s,a'^s\rangle \langle \overline{a^p}, \overline{a'^p}\rangle.$$

To show that this is a kernel function, we show that resulting kernel matrices are positive semidefinite. For this, we show that for any set of (b_i,a_i) pairs and vectors x_i of rationals of the right dimensionality,

$$\sum_{i,j} x_i K_{ij} x_j = \sum_{i,j} x_i k((b_i,a_i),(b_j,a_j)) x_j \geq 0$$

This is because

$$\begin{aligned}\sum_{i,j} x_i k((b_i,a_i),(b_j,a_j)) x_j &= \sum_{i,j}\sum_k x_i x_j \cdot a^s_{ik} a^s_{jk} \cdot \overline{a^p}_{ik}\overline{a^p}_{jk} \cdot b_{ik} b_{jk} \\ &= ||\sum_{i,k} x_i \cdot a^s_{ik} \cdot \overline{a^p}_{ik} \cdot b_{ik}||^2 \geq 0.\end{aligned}$$

As in the case of ACER on master action space, the training algorithm is unchanged. Only the kernel function is adjusted to incorporate the idea of similarity between master actions. The GP can thus be trained on this 1035-dimensional master action space.

Chapter 4

Evaluation

In this chapter, we aim to quantitatively evaluate the performance of our implementation of ACER, described in the previous chapter. We find that ACER delivers the best performance and fastest convergence among all Neural Network-based algorithms implemented in PyDial. We follow up this observation with an investigation of the contribution of TRPO. We also deploy the algorithm in a more challenging setting without the *execution mask* aiding action selection. Next, we investigate the effect of different hyperparameter selections, and the algorithm's stability against it. As an extension to the core project, we then turn our attention to master action space, and deploy ACER and GP on it. Finally, we investigate how resilient different algorithms are to semantic errors and changing testing conditions.

4.1 Testing method

During training, the algorithm the algorithm explores the state space. As training progresses, this exploration is generally decreasing and the algorithm exploits what it learned more and more. In testing, we aim to measure the performance of the algorithm *as if it stopped training* after a certain number of episodes. Thus, testing experiments are run as follows. First, the total number of *dialogues* or *iterations* (here, usually 4000) is broken down into *milestones* (here, usually 20 milestones of 200 iterations each). As the training over the total number of iterations progresses, a snapshot of the state of the training (all NN weights, hyperparameters, and replay memory) is saved at each milestone. A separate run of 4000 iterations is then performed without any training steps, where each of the saved snapshots are tested for 200 iterations each. In addition to no training being performed during this testing phase, it also performs *no exploration*. More specifically, in our case, the greedy policy with respect to π, instead of ε-greedy, is used to derive the next action. This informs us on the performance of

the system *as if it stopped training* at a specific milestone, allowing us to observe the speed of convergence and the performance of early milestones, discounting for the exploration.

We use the PyDial user simulator as described in Section 2.2, with the *focus* belief tracker for all experiments. The number of maximum turns is limited to 25, after which, if the user did not achieve their goal, the dialogue is deemed unsuccessful. The discount factor γ is set to 0.99 for all algorithms where it is applicable. For NN-based algorithms, the size of a *minibatch*, on which the training step is performed, is 64. For algorithms employing Experience Replay, the replay memory has a capacity of 2000 interactions. Where ε-greedy is used, ε is linearly reduced from 0.95 down to 0 over the training process. For NN-based algorithms, the size of the hidden layers is 130 for h_1 and 50 for h_2

4.2 Performance of ACER

To test ACER with other algorithms implemented in PyDial, we introduce the initial environment. The simulated semantic error rate is 0% both for training and testing. The learning rate for the actor and critic are 0.001. Instead of a simple Gradient Descent on the loss function, we use the Adam Optimiser, which associates *momentum* to the gradient [15]. To discourage the algorithm from learning a trivial policy, we subtract a small multiple (0.01) of the policy entropy. This way, saying eg. *bye* every time is discouraged, as the entropy of such a policy would be low, thus the loss would be higher. The ACER-specific hyperparameters are: $c = 5, \delta = 1, \beta = 0.99, n = 1$. The batch size is 64. We perform training and testing on 4000 iterations split into 20 steps. We compare our implementation of ACER to existing implementations of eNAC, A2C and GP. We run the experiment 5 times and average the results, to reduce the variance arising from different random initialisations. We compare the average per-episode reward obtained by the agent, the average number of turns in a dialogue and the percentage of successful dialogues. For every turn in the dialogue, there is a reward of -1 to incentivise shorter dialogues. For every dialogue where the simulated user achieved their goal, there is a reward of 20. The algorithms optimise this reward metric, rather than the success rate. It is thus important to include it in the comparisons. As the training progresses, the success rate generally increases (Figure 4.1), while the average reward increases (Figure 4.2) and the number of turns decreases (Figure 4.3).

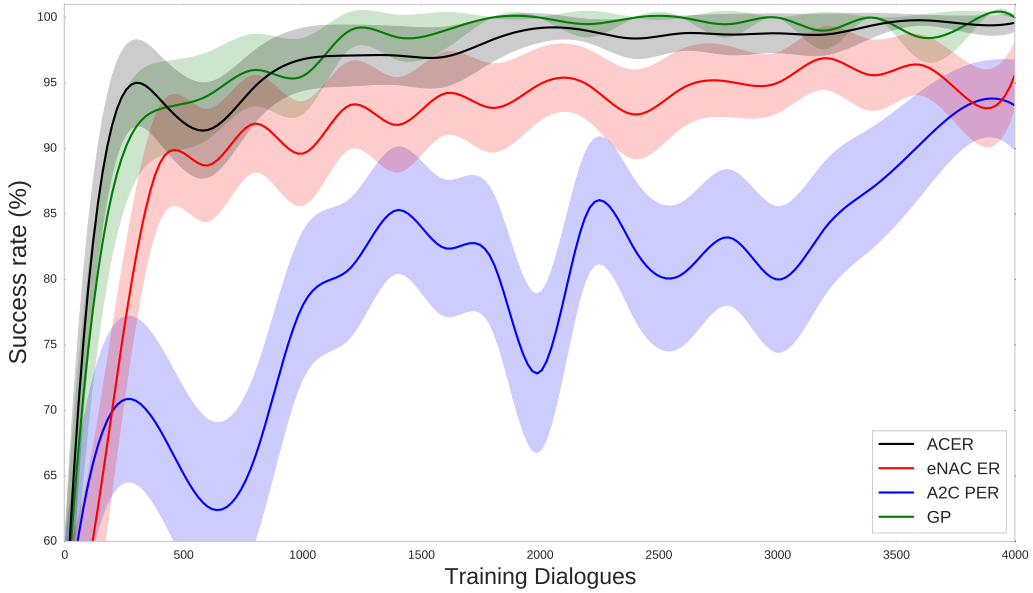


Fig. 4.1 Success rate of ACER compared to other RL methods.

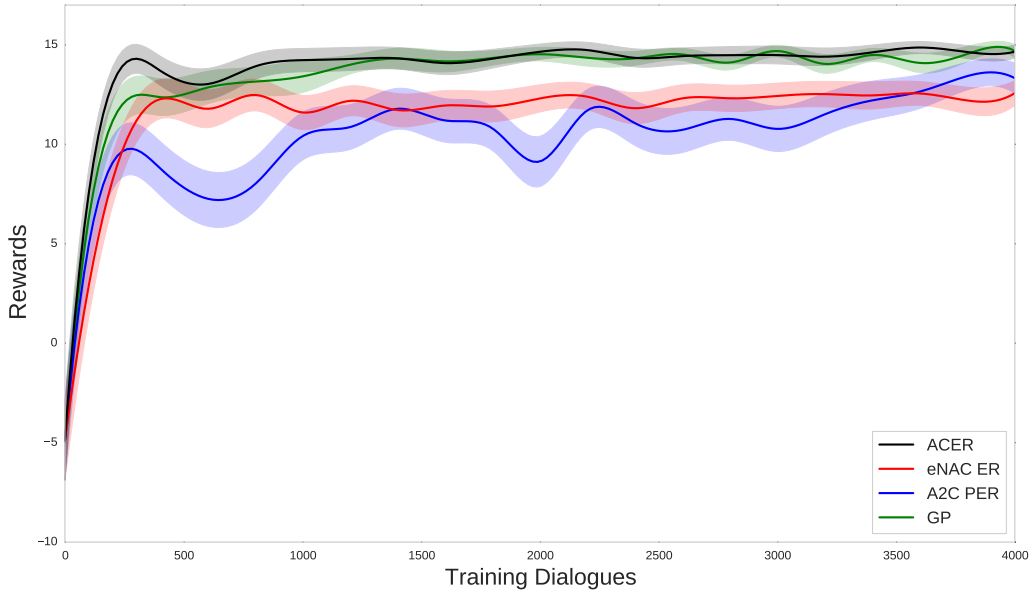


Fig. 4.2 Rewards of ACER compared to other RL methods.

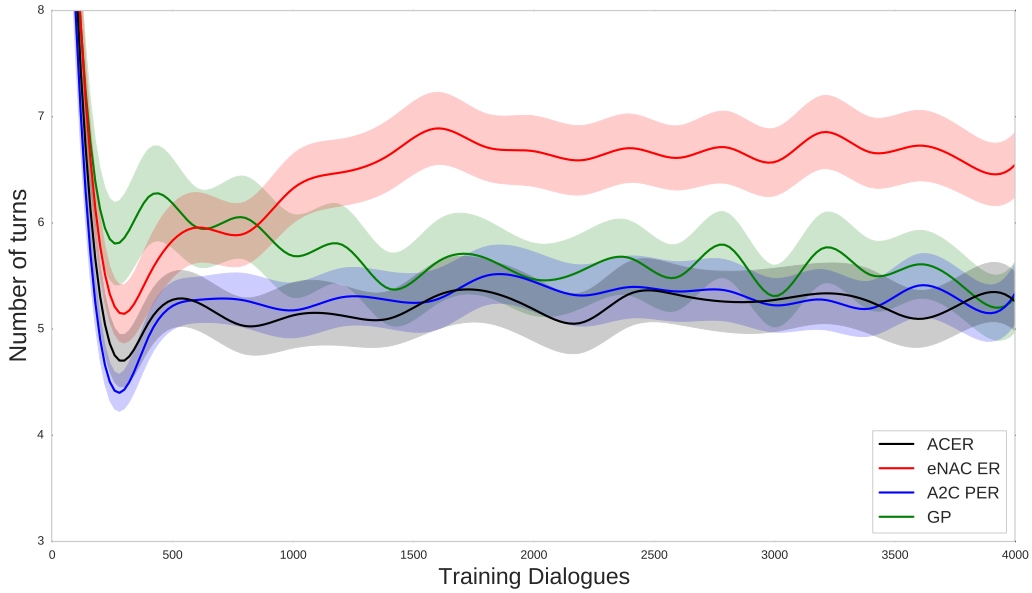


Fig. 4.3 Number of turns of ACER compared to other RL methods.

We observe that ACER is comparable to GP in terms of speed of convergence, sample efficiency, success rate, rewards and turns. While the success rate of ACER remains one or two percentage points below that of GP, ACER requires fewer dialogue turns and ultimately obtains somewhat higher rewards than GP. This suggests that the slightly worse success rate of ACER presents a shortcoming of the reward function rather than the algorithm, as the algorithm only optimises the reward function.

We also observe that ACER far exceeds the performance of other Neural Network-based methods in terms of all of speed of convergence, sample efficiency, success rate, rewards and turns. These results far exceed our expectations.

4.3 Contribution of TRPO

As ACER is an elaborate algorithm including an efficient implementation of TRPO. This is a modular part of ACER, ie. it serves to stabilise the learning but can be removed completely. To investigate the advantage that TRPO provides to the whole algorithm, we removed it and compared the performance of ACER with TRPO to ACER without TRPO (Figure 4.4). While the algorithm converges to a similar performance without TRPO, we can see that applying TRPO is beneficial in the early stages of learning, as it stabilises the updates. As a result, TRPO improves the rate of convergence and sample efficiency.

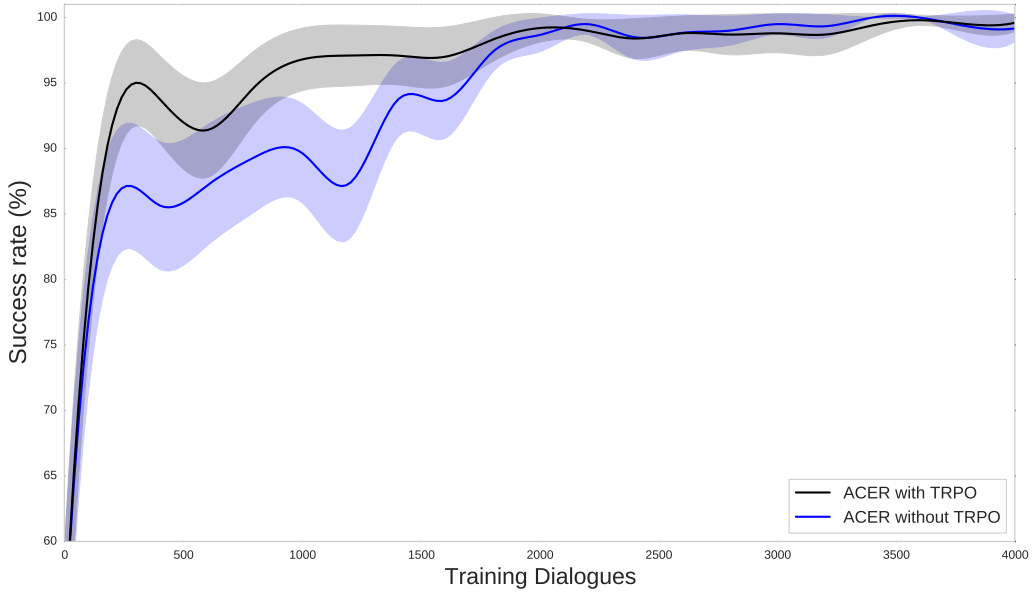


Fig. 4.4 Success rate of ACER with TRPO, compared to ACER without TRPO.

4.4 Effect of execution mask

As introduced in Section 2.1.4, not every system action is appropriate in every situation. For example, *inform* is not a valid action at the very beginning of the dialogue, when the system has not yet received any information on what kind of entity the user is looking for. As a result, if the policy were to select such an action, the system would fail to convert it to a valid response [1]. To fix this, an *execution mask* is constructed by the system that ensures that only valid actions are selected: the probability of invalid actions is set programmatically to zero. Removing this mask inherently complicates the task of policy learning, as the policy then has to learn not to select the inappropriate actions based on the belief state. To test how ACER tackles this problem, we repeat our experiments without the execution mask. Figure 4.5 compares success rates, while Figure 4.6 compares rewards.

[1] In this case, an empty response is constructed during the conversion process.

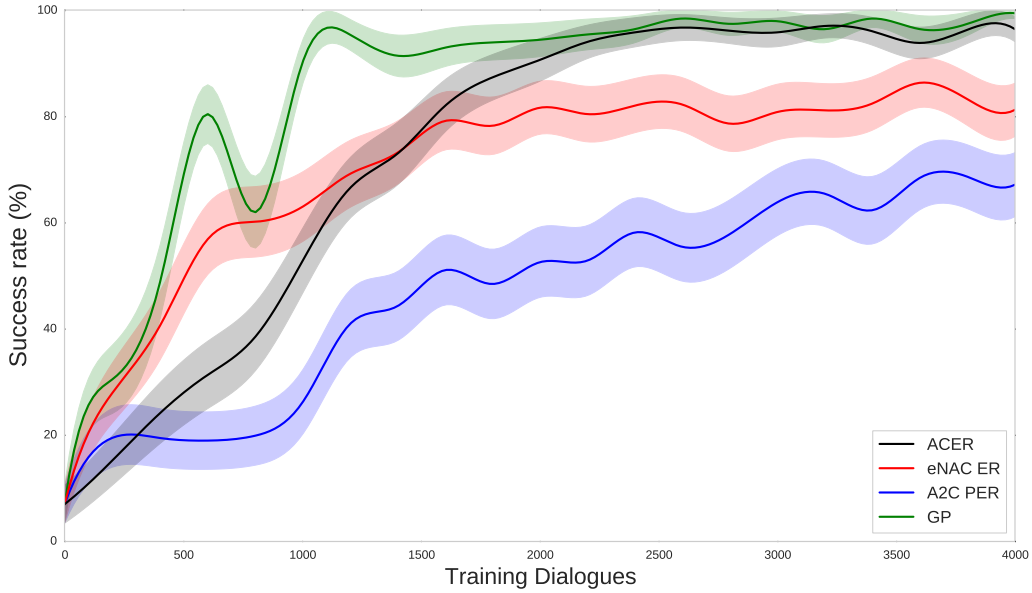


Fig. 4.5 Success rate of ACER compared to other RL methods, without the execution mask.

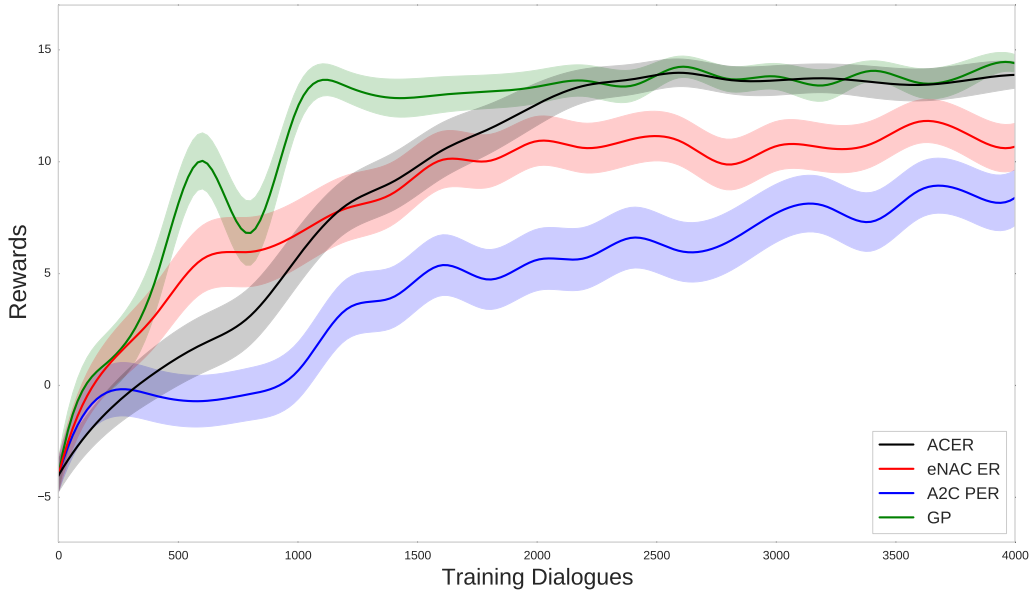


Fig. 4.6 Rewards of ACER compared to other RL methods, without the execution mask.

We can see that in general, algorithms converge slower under this more difficult setting, as expected. The final performance of GP and ACER remain somewhat below their performances with the mask. This is also expected as a mapping learned by RL is rarely as precise as a hard-coded solution to a problem (execution mask). Still, the difference in final

performances is surprisingly low. GP shows faster initial convergence than ACER, as the latter shows a more steady progress. They remain comparable in every other regard.

4.5 Hyperparameter tuning

ACER has several additional hyperparameters compared to more traditional algorithms. We investigate the effect of hyperparameters $c, \delta, \beta,$ and n on the algorithm's performance. To better illustrate the differences, we run the tests in a more challenging setting, without the execution mask.

Importance Weight threshold c This value is the upper bound of IS weight; weights higher than c are truncated. Setting this value to too high diminishes the effect of weight truncation, while a value too low will rely more on the less accurate bias correction term. From Figure 4.7, we see that $c = 5$ delivers the highest convergence rate and a good final performance. We also see that for the wide range of values from $c = 1$ to $c = 20$, there is no big difference in final performance, suggesting that the algorithm is stable in face of varying this hyperparameter.

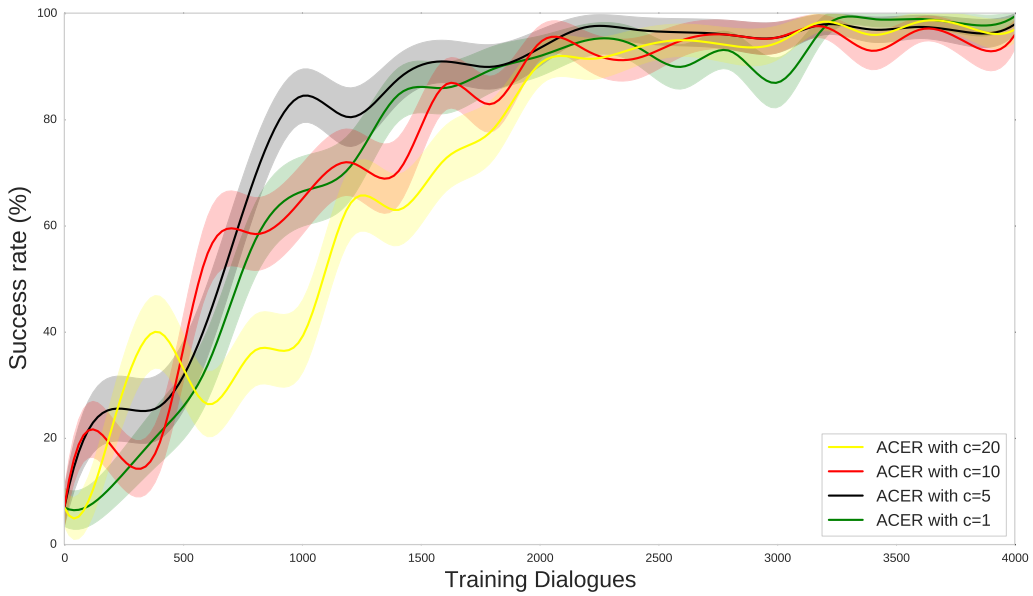


Fig. 4.7 Success rate of ACER with varying hyperparameter c.

KL divergence constraint δ This value constrains the KL divergence between an updated policy and the running average policy. Setting it too high allows radical jumps, setting it too low slows the convergence down. Figure 4.8 shows results for different settings. We can see

that a setting of $\delta = 10$ or $\delta = 50$ results in erratic changes in the performance of ACER, while $\delta = 0.5$ and $\delta = 1$ are sensible choices.

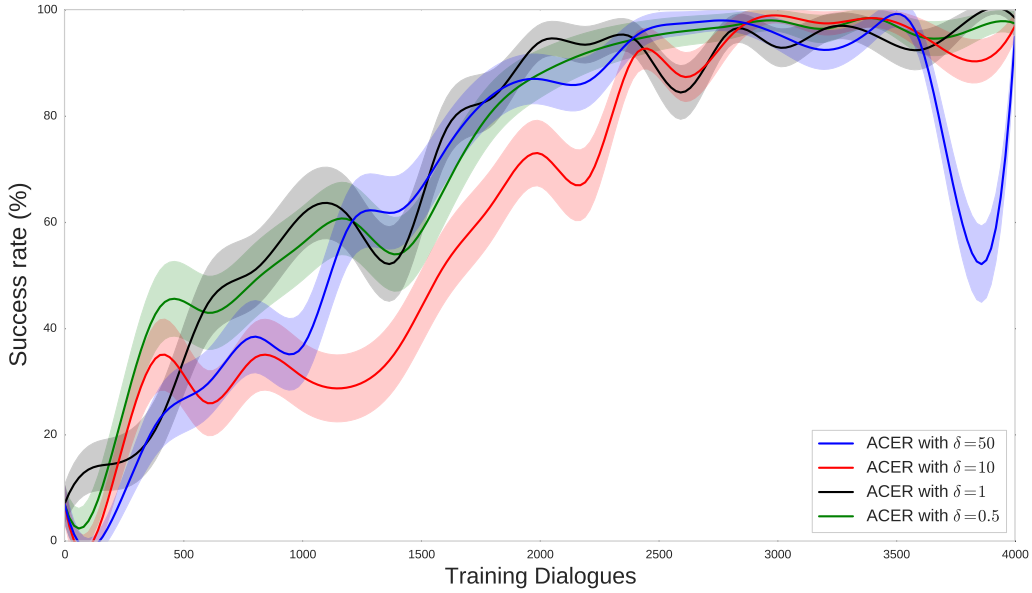


Fig. 4.8 Success rate of ACER with varying hyperparameter δ.

Average policy update weight β As discussed in Section 3.6, a too low value makes the average policy forget the history very quickly, but a too high value will prevent the average policy to adjust to the current policy. Figure 4.9 shows the comparisons. We can see that for $\beta \leq 0.9$, the average policy forgets the history too quickly, allowing the policy to gain momentum in any direction and thus preventing it from converging to a good performance. For $\beta = 0.95$, the policy converges quickly, while $\beta = 0.99$ results in a somewhat conservative algorithm, where the KL divergence constraint keeps the policy near a slowly changing average. $\beta = 0.99$ still converges to a good result, but does so somewhat slower than $\beta = 0.95$.

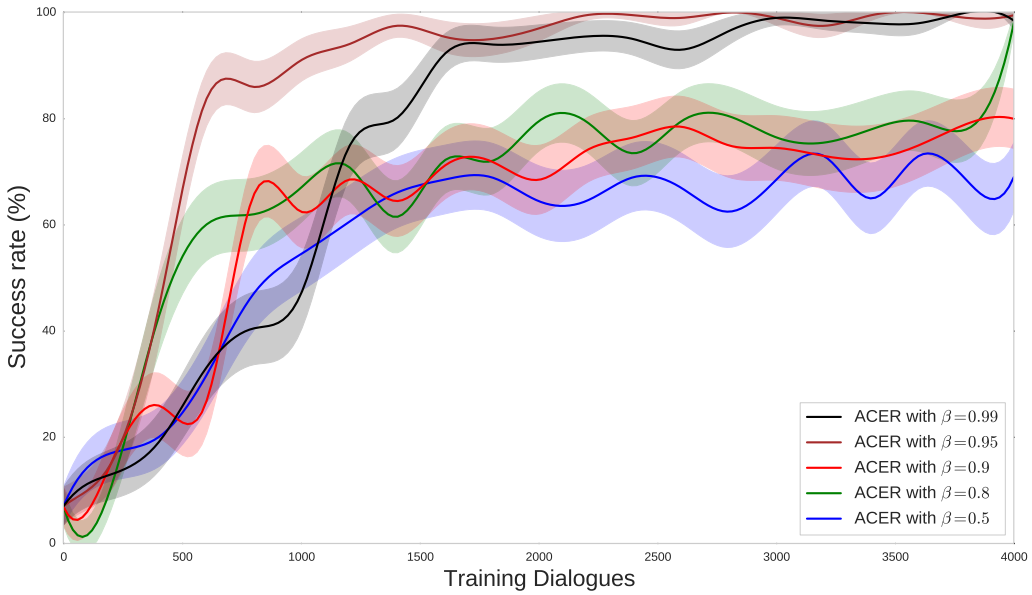


Fig. 4.9 Success rate of ACER with varying hyperparameter β.

Training iterations n For each episode gathered, we run the training step n times. Setting this number higher allows the algorithm to learn more from the gathered experience. However, if n is too high, the training might diverge due to the policy moving too much. This is illustrated in Figure 4.10: for $n = 1$, convergence is quick and performance is good. For $n = 10$, performance stays poor throughout. For $n = 30$ and $n = 50$, the algorithm diverges completely.

Fig. 4.10 Success rate of ACER with varying hyperparameter n.

4.6 Master action space

ACER compares favourably to other NN-based algorithms, but performs about equally if not slightly worse than GP in our experiments. The experiments were run on the summary action space, which only has 15 actions. In a more difficult scenario, we may have orders of magnitude more actions. In such scenarios, the computational cost of GPs can be prohibitive. If ACER still performs well under the same scenario, it might may be the overall best method to apply to larger action spaces. This is because ACER does not have the prohibitive cubic computational cost of GP, and is expected to train much more quickly.

To test our hypotheses, we deploy ACER on the master action space according to Section 3.8. Figure 4.11 compares the results to ACER on summary space. Both experiments were run with the execution mask. We can see that convergence is slower on the master action space. This is expected due to having to choose between vastly higher number of actions on the master action space (1035 as opposed to 15). However, ACER is still surprisingly effective on the master action space, converging to about the same performance as on the summary space. We note that this is without any modification to the training algorithm; as described in Section 3.8, only the underlying Neural Network is changed. ACER achieves the best results in terms of speed of convergence and final performance on master action space out of NN-based SDS policy optimiser algorithms.

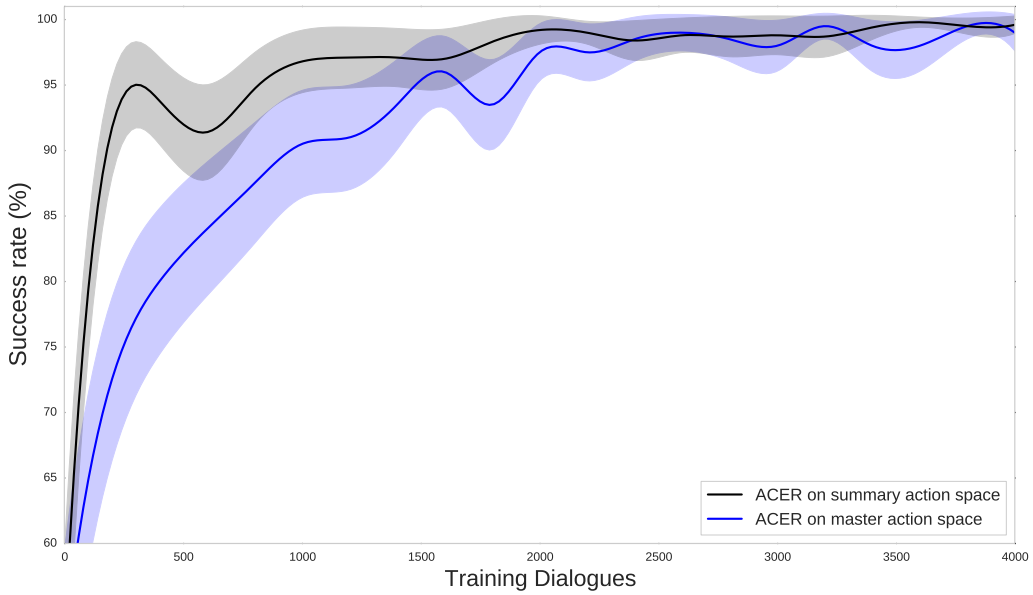


Fig. 4.11 Success of ACER on summary action space compared to ACER on master action space.

To investigate further whether ACER is the best choice of algorithm on the master action space, we modify GP to run on master action space according to Section 3.9. We compare ACER and GP both on summary and master action spaces, without the execution mask in Figure 4.12. Both GP and ACER show slower speed of convergence on master action space. This is expected, as the random initialisation of a policy on master action space will be much less sensible than an initialisation on the summary space, the latter taking advantage of the hard-coded summary to master action mapping method. However, it is surprising to see that all experiments converged to roughly the same performance of about 97% success rate, except for GP on summary, which has a final success rate of 98%-99%. This suggests that both ACER and GP can handle large action spaces quite efficiently. To our knowledge, our implementation of both ACER and GP on master action space achieves better performance than any experiment of a policy optimiser on master action space in SDS.

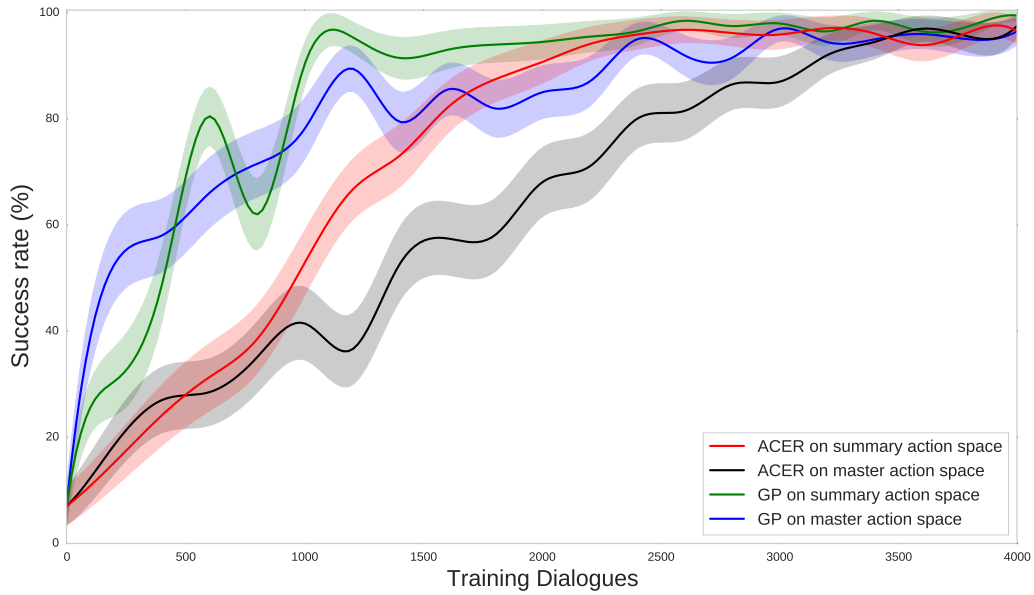


Fig. 4.12 Success of ACER and GP on summary and master action spaces, without execution mask.

GP is more sample efficient than ACER on the challenging master action space without execution mask. However, it requires vastly more computational resources to run: this experiment took 6.45 hours to run with ACER, and 8.63 days with GP[2]. Arguably, the extra computational cost overshadows the disadvantage of ACER, that it has to be run for more iterations to converge.

4.7 Resilience against errors

So far, our experiment settings were quite idealised, training and testing policies under a perfect simulator with no semantic errors. However, in real life, the pipeline surrounding the policy optimiser deals with substantial uncertainty, which tends to introduce errors. We ultimately want to measure how well a policy optimiser can learn the optimal strategy in face of noisy *semantic*-level input. In our experiments, we control this by the *semantic error rate*, the rate at which a random noisy input is introduced to the optimiser to simulate an error scenario. We focus on two desirable properties of a policy. First, ideally, the policy would learn not to *trust* the input as much, and ask questions until it is sure about the user goal, just like a real human would if the telephone line is noisy. Second, an ideal policy would not only adjust to the error rate of the training conditions, but would dynamically adjust to the

[2]The running times were measured on an Azure cloud machine with a 16-core CPU and 64GB of RAM.

conditions of the dialogue it is in. If the policy adjusts too much to the training conditions, it is said to *overfit*. This could severely limit the policy's deployability.

We test key algorithms for these two desirable properties. eNAC, the previous best NN-based policy optimiser is compared to ACER and GP. ACER and GP are also compared to their respective variants in master action space. We run the test as follows: first, we train the algorithms under 15% semantic error rate until convergence, with the execution mask. Then we take the fully trained policy and test it under a range of semantic error rates, ranging from 0% to 50%. Figure 4.13 and Figure 4.14 shows the results of the experiment.

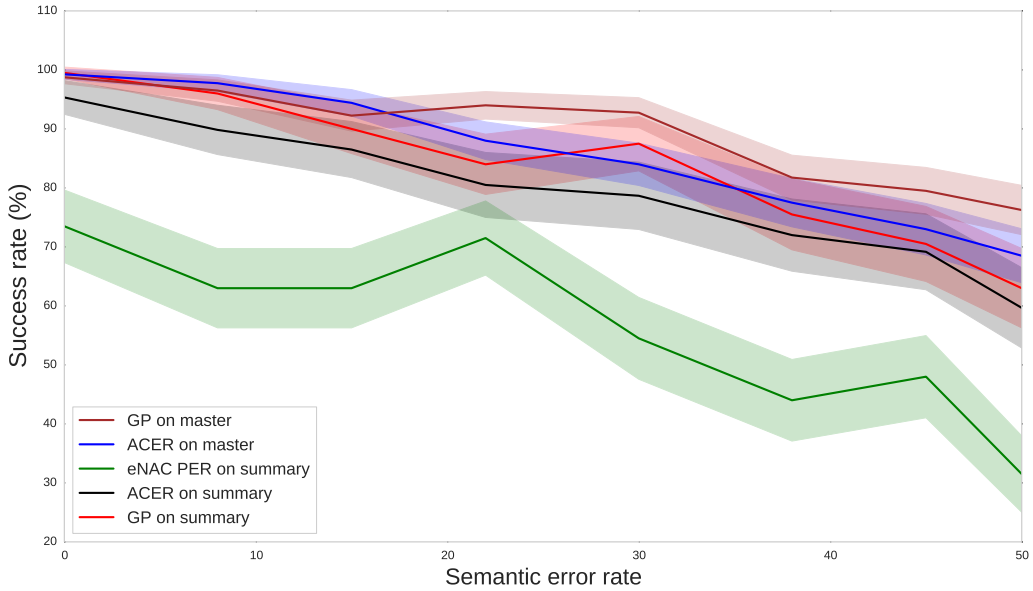


Fig. 4.13 Success rate of key algorithms when training them on 15% and testing them on varying error rates.

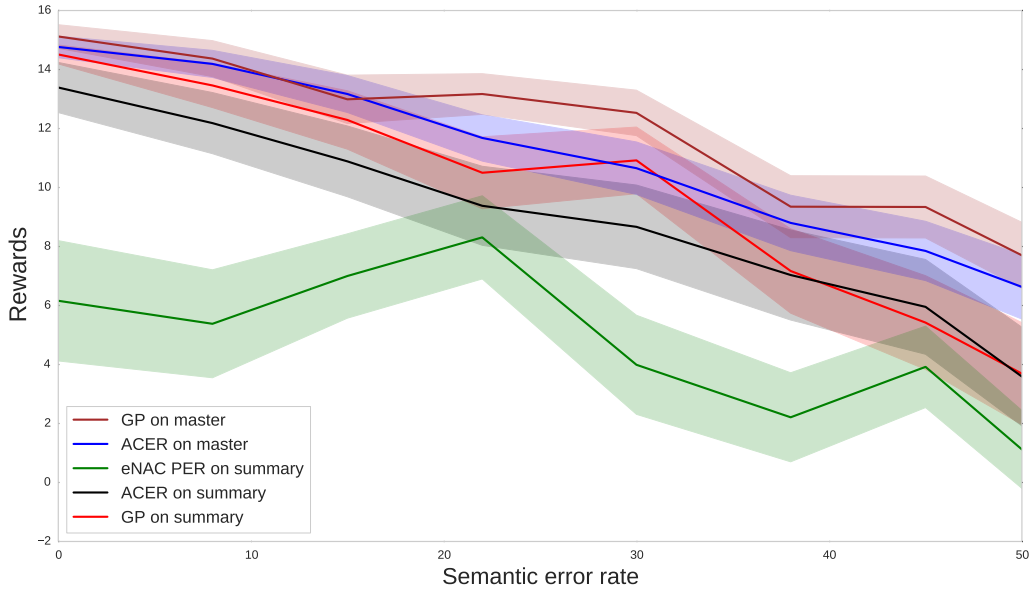


Fig. 4.14 Rewards of key algorithms when training them on 15% and testing them on varying error rates.

Success rate and reward follow the same trends. As expected, we see a general downwards trend for each algorithm as the semantic error rate increases. There is however no apparent spike in performance at the 15% semantic error rate of the training process, indicating that none of the algorithms overfit to this setting. We can see that the performance of eNAC is far behind all the other algorithms. ACER and GP are closer in performance, but GP on summary space consistently beats ACER on summary space.

It might be surprising that both ACER and GP perform better when trained on the master action space as opposed to the summary space, given that they performed worse in previous experiments. However, those experiments had no semantic errors, and a hand-crafted rigid mapping from summary actions to master actions, that relied on the belief state to find the best payload for an inform action. Under a higher semantic error rate, the belief state will be noisy and this code may not perform optimally. This highlights the benefits of expanding the scope of Artificial Intelligence in SDS: AI can be more versatile than hand-coded mappings, especially when the mapping performs *decision making under uncertainty*.

Chapter 5

Summary and Conclusions

This project has been a major success. The policy optimiser algorithms implemented for the project beat the state of the art in Spoken Dialogue Systems in three ways:

- A version of ACER [33] designed for Spoken Dialogue Systems shows better results than the current state of the art for Neural Network-based policy optimisers [27].
- This implementation of ACER is also able to train in the master action space, showing the best performance among Neural Network-based policy optimisers, as reported by Fatemi et al. [4] and Su et al. [27].
- Our implementation of GP with a redesigned kernel function achieves the best performance on master action space, something which previously was not possible.

GP suffers from an inherently high computational cost, making the algorithm unsuitable in higher volume action spaces. In such cases, the fact that ACER can be trained well on the master action space indicates that it may be the best currently known method to train policies with large action spaces.

As agents powered by Machine Learning gain more intelligence, they can be applied to more and more challenging domains. With respect to SDS, this could mean that more and more of the system will be controlled by Artificial Intelligence, while less and less of it will be hard-coded. Using the master action space is a good example of this: a hard-coded mapping between summary and action spaces *can* be used to simplify the task of the AI agent. However, as the project shows, it is no longer *required* to train in this action space. There is an algorithm (ACER) that can finally bridge the semantic gap between summary and master action spaces without the help of domain-specific code written explicitly for this

mapping [1]. This has three benefits: first, as has been demonstrated, training on master action space outperforms the mapping based on fixed code, when uncertainty (semantic errors) is involved. Second, it allows us to build a more generally applicable system, with less work required to deploy it in differing domains. Third, it allows us to consider domains that have vastly higher action spaces, even if there is no clear way to convert those action spaces into small summary action spaces (such as a general purpose dialogue system).

ACER fits well into other SDS research directions too. Successful policy optimisers need to be sample efficient and be able to be trained quickly, to avoid subjecting human users to poor dialogue performance for long. ACER uses Experience Replay for sample efficiency, together with many methods aimed at reducing bias and variance of the estimator, to achieve quick training. Recently, Su et al. [27] combined Supervised Learning with Deep Reinforcement Learning to investigate the performance of an agent bootstrapped with SL and trained further with DRL. The Neural Networks of ACER are compatible with that approach.

5.1 Future work

This project showed how the framework of Neural Network-based Actor Critic can be extended with Experience Replay, TRPO, and several other methods to design a sample efficient policy optimiser with good performance. Here, we introduce some of the many directions in which this work could be continued.

5.1.1 Supervised pre-training

As explored by Su et al. [27], Supervised Learning is inherently unsuitable to learn a policy efficiently for many reasons. First, the supervision for the learning comes from an agent which may act imperfectly or suboptimally at times. Second, the number of supervised samples may be limited, while the RL framework gathers experience for itself. Third, the key dialogue acts which lead to success or failure are not identified by the SL framework, and thus the *credit assignment problem* is unsolved. The RL framework deals extensively with this problem by devising suitable update targets.

Despite all of these shortcomings, Supervised Learning could be used to pre-train a policy before Reinforcement Learning takes over. In this way, RL would start from better performance, which may decrease the overall interactions required for convergence, as well as increase sample efficiency. This pre-training method however has to be designed in such a

[1] The design of Neural Networks in ACER was optimised for the domain, as described in Section 3.8. However, the training algorithm itself remains general.

way as to avoid leaving the policy in a local optimum, from which Reinforcement Learning may not be able to recover. It would be interesting to see how much Supervised Pretraining could improve the current performance of ACER.

5.1.2 Expanding the action space

Both of our settings, training on summary and on master action space, considered static action spaces only. Under this framework, the entire policy would have to be retrained if a new action or payload were to be introduced. This could hurt the maintainability of a real-life Dialogue System, as it would be expensive to extend the database schema or the list of actions. Ideally, the training algorithm could adapt to such changes made, being able to retain its pre-existing knowledge of the old actions.

Additionally, based on the learned knowledge of the old actions, the learning framework may be able to quickly place the desirability of the new action based on the belief state quickly. Intuitively, this works by categorising the new action based on the insight provided by the old one. A Bayesian approach of One-shot Learning has been attempted by Fei-Fei et al. [5] for object recognition. A similar method could be devised for actor-critic Neural Networks. This would not only improve sample efficiency for the new action, but also provide the benefit of not having to retrain the entire model. It would also be interesting to investigate whether in a large action space such as the master action space, the sample efficiency of ACER could be further improved by adding actions iteratively, thus starting out with a simple training task and gradually increasing the difficulty.

5.1.3 Off-policy eNAC

As discussed in Section 3.6, the motivation behind Trust Region Policy Optimisation is to restrict the gradient according to a distance metric on the policy, rather than an Eucledian metric on the underlying parameters. Small changes in the latter can result in radical changes in the policy, while restricting the step-size directly in the policy space stabilises learning. To avoid expensive Fisher Matrix computations, TRPO is based on an estimate of the natural gradient:

$$d\omega^T G_\omega d\omega \approx \mathrm{KL}(\pi(\cdot|b,\omega)||\pi(\cdot|b,\omega+d\omega)).$$

However, Episodic Natural Actor Critic can directly use the natural gradient, without an approximation, while avoiding expensive Fisher matrix calculations (Section 2.3.3). It would be interesting to see how this algorithm would perform in an off-policy setting. Some work has been carried out in this topic as part of a possible extension to the project, but initial

results were disappointing. The challenge comes down to whether the dialogue could be separated into individual dialogue acts: on the one hand, reducing the variance of IS weights necessitates separating dialogue acts apart, as otherwise, a product of IS weights is introduced for an entire dialogue. On the other hand, eNAC relies on dialogues being considered together to be able to limit introducing further unknown constants in the equation; currently, only the value of the initial state is a constant, and eNAC relies on that value being the same for every dialogue.

However, if IS variance could be reduced in a way that is compatible to natural gradient based learning, we could see another policy optimiser that is sample efficient and converges to a good performance.

References

[1] Shun-Ichi Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276, 1998.

[2] Senthilkumar Chandramohan, Matthieu Geist, and Olivier Pietquin. Optimizing spoken dialogue management with fitted value iteration. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[3] Thomas Degris, Martha White, and Richard S Sutton. Off-policy actor-critic. *arXiv preprint arXiv:1205.4839*, 2012.

[4] Mehdi Fatemi, Layla El Asri, Hannes Schulz, Jing He, and Kaheer Suleman. Policy networks with two-stage training for dialogue systems. *arXiv preprint arXiv:1606.03152*, 2016.

[5] Li Fei-Fei, Rob Fergus, and Pietro Perona. One-shot learning of object categories. *IEEE transactions on pattern analysis and machine intelligence*, 28(4):594–611, 2006.

[6] M Gašić, Simon Keizer, Francois Mairesse, Jost Schatzmann, Blaise Thomson, Kai Yu, and Steve Young. Training and evaluation of the his pomdp dialogue system in noise. In *Proceedings of the 9th SIGDIAL Workshop on Discourse and Dialogue*, pages 112–119. Association for Computational Linguistics, 2008.

[7] Milica Gasic 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.

[8] Milica Gašić, Filip Jurčíček, Simon Keizer, François Mairesse, Blaise Thomson, Kai Yu, and Steve Young. Gaussian processes for fast policy optimisation of pomdp-based dialogue managers. In *Proceedings of the 11th Annual Meeting of the Special Interest Group on Discourse and Dialogue*, pages 201–204. Association for Computational Linguistics, 2010.

[9] Matthieu Geist and Bruno Scherrer. Off-policy learning with eligibility traces: A survey. *The Journal of Machine Learning Research*, 15(1):289–333, 2014.

[10] Anna Harutyunyan, Marc G Bellemare, Tom Stepleton, and Rémi Munos. Q (\lambda) with off-policy corrections. In *International Conference on Algorithmic Learning Theory*, pages 305–320. Springer, 2016.

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

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

[13] F Jurcıcek, M Gašic, S Young, R Laroche, G Putois, M Geist, and O Pietquin. D1. 5: Online adaptation of dialogue systems. 2011.

[14] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1):99–134, 1998.

[15] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[16] Jens Kober and Jan R Peters. Policy search for motor primitives in robotics. In *Advances in neural information processing systems*, pages 849–856, 2009.

[17] Esther Levin, Roberto Pieraccini, and Wieland Eckert. A stochastic model of human-machine interaction for learning dialog strategies. *IEEE Transactions on speech and audio processing*, 8(1):11–23, 2000.

[18] Jiwei Li, Will Monroe, Alan Ritter, Michel Galley, Jianfeng Gao, and Dan Jurafsky. Deep reinforcement learning for dialogue generation. *arXiv preprint arXiv:1606.01541*, 2016.

[19] Peter Marbach, Oliver Mihatsch, Miriam Schulte, and John N Tsitsiklis. Reinforcement learning for call admission control and routing in integrated service networks. In *Advances in Neural Information Processing Systems*, pages 922–928, 1998.

[20] Rémi Munos, Tom Stepleton, Anna Harutyunyan, and Marc Bellemare. Safe and efficient off-policy reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1046–1054, 2016.

[21] Razvan Pascanu and Yoshua Bengio. Revisiting natural gradient for deep networks. *arXiv preprint arXiv:1301.3584*, 2013.

[22] Jan Peters and Stefan Schaal. Natural actor-critic. *Neurocomputing*, 71(7):1180–1190, 2008.

[23] Jan Peters, Sethu Vijayakumar, and Stefan Schaal. Natural actor-critic. In *European Conference on Machine Learning*, pages 280–291. Springer, 2005.

[24] Doina Precup. Eligibility traces for off-policy policy evaluation. *Computer Science Department Faculty Publication Series*, page 80, 2000.

[25] Christian Raymond and Giuseppe Riccardi. Generative and discriminative algorithms for spoken language understanding. In *Eighth Annual Conference of the International Speech Communication Association*, 2007.

[26] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1889–1897, 2015.

[27] Pei-Hao Su, Pawel Budzianowski, Stefan Ultes, Milica Gasic, and Steve Young. Sample-efficient actor-critic reinforcement learning with supervised data for dialogue management. *arXiv preprint arXiv:1707.00130*, 2017.

[28] Richard S Sutton. Reinforcement learning architectures for animats. In *From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*, pages 288–296, 1991.

[29] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[30] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[31] Blaise Thomson. *Statistical methods for spoken dialogue management*. Springer Science & Business Media, 2013.

[32] Blaise Thomson and Steve Young. Bayesian update of dialogue state: A pomdp framework for spoken dialogue systems. *Computer Speech & Language*, 24(4):562–588, 2010.

[33] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, and Nando de Freitas. Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224*, 2016.

[34] Jason D Williams, Kavosh Asadi, and Geoffrey Zweig. Hybrid code networks: practical and efficient end-to-end dialog control with supervised and reinforcement learning. *arXiv preprint arXiv:1702.03274*, 2017.

[35] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[36] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Mike Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. Achieving human parity in conversational speech recognition. *arXiv preprint arXiv:1610.05256*, 2016.

Acronyms

A2C Advantage Actor Critic. 24, 25, 27, 29

ACER Actor Critic with Experience Replay. v, 35

AI Artificial Intelligence. 1, 3

DQN Deep Q-network. 23, 24, 27

DRL Deep Reinforcement Learning. 22, 23, 27

eNAC Episodic Natural Actor Critic. 22, 23

ER Experience Replay. v, 23, 24, 27, 29

FIM Fisher Information Matrix. 21

GP Gaussian Process. v, 17, 35

IS Importance Sampling. v, 14, 15, 28–32, 34, 35

MDP Markov Decision Process. 8, 9

ML Machine Learning. 11

MSVE Mean Squared Value Error. 18, 23

NAC Natural Actor Critic. 21

NLG Natural Language Generation. 10

NN Neural Network. v, 2, 3, 7, 11, 18, 19, 22–24, 31, 33, 34

POMDP Partially Observable Markov Decision Process. 8

ReLU Rectified Linear Unit. 34

RL Reinforcement Learning. 1, 12, 22

SDS Spoken Dialogue System. 1–5, 7, 8, 10, 12, 27

SGD Stochastic Gradient Descent. 18, 23, 24

TD Temporal Difference. 15, 16

TRPO Trust Region Policy Optimisation. v, 35

Appendix A

Example dialogue

Below is an example dialogue between a user looking for a restaurant with a medium price range, and a system that internally translates between summary and master actions. We thus illustrate the nature of a dialogue and the distinction between summary and master actions. System responses are written as Sys: summary_action → master_action:

Sys : hello() → hello()

User : inform(type=restaurant)

Sys : request_pricerange() → request_pricerange()

User : inform(pricerange=moderate)

Sys : inform() → inform(food="turkish",pricerange="moderate",name="efes restaurant")

User : reqalts()

Sys : inform_alternatives() → inform(food="turkish",pricerange="moderate",name="anatolia")

User : request(phone)

Sys : inform_requested → inform(name="anatolia",phone="01223 362372")

User : thankyou()

Sys : bye()