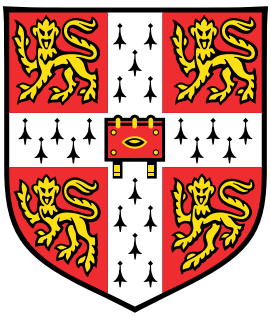
Optimising spoken dialogue systems using Gaussian process reinforcement learning for a large action set



Thomas F. W. Nicholson

Department of engineering

University of Cambridge

M.Phil in Machine Learning, Speech and Language Technology

The dissertation is submitted for the degree of Master of Philosophy

Pembroke College

11^{th} August, 2016

I, Thomas F. W. Nicholson of Pembroke 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 - 12923

Signed -  .

11th August, 2016

Abstract

In dialogue management for statistical dialogue systems we seek to learn a policy that can select the optimal action for the system to perform. This dissertation looks at how by defining kernels over the action space we allow the extension of policy optimisation to very large action sets. As well as defining a general form for performing reinforcement learning in any domain over which a kernel can be defined, we look at the specific case of using tree kernels for system actions, perform experimental justification of the approach, and report some provisional, though disappointing, dialogue performance.

Contents

Introduction

Spoken Dialogue Systems (SDS) allow users to interact with an automated system using spoken natural language as the medium of communication. Such an interaction style has been present in science fiction for decades, and has recently found commercial success, and could become one of the most pervasive interaction styles in the post-mobile era. SDS are alluring because they represent a very familiar interaction style, and can appear as natural as interacting with a truly intelligent agent.

SDS are typically multi-component systems in which each component solves a hard machine learning problem, before passing the output along to the next component in a pipeline process. In order for SDS to become the pervasive interaction styles that some envisage, each of the components will have to yield significant improvements. It is one particular component that forms the focus of this work, that of *policy optimisation* in which the system learns how to select the *action* to be performed next. This action may be informing the user of, or making a request for more information or may be external to the user, as in booking an item or phoning a certain contact.

Currently this learning of action select is performed on a much smaller hand-engineered representative subset of all possible actions which are then transformed using a hard-coded function into the actual system action that is to be performed. The presence of hand-coded features and feature mappings represents not just an aesthetically displeasing imperfection, but is also a barrier to scalability. If a system wishes to be deployed to a new domain, or have its domain expanded or use multiple ontologies, then the hand-engineered part of the system has to be either rewritten or expanded. If instead learning could take place over the original actions then a system could be instantly applied to any number of possible new domains simply by automatically optimising the policy of the SDS.

This work focuses on investigating the necessary steps to allow Gaussian process-based reinforcement to be performed on an action set, independent of size, given only a kernel function, and attempts to show the feasibility allowing a SDS to optimise the selection of a potentially very large number of original actions with an appropriately defined kernel. All practical implementation was performed within the Cambridge University Engineering Department dialogue systems tool kit, *cued-pydial*. Reinforcement learning over a large action set has been a lively area of research for decades, and the problems to be overcome are extremely challenging.

Background

Modular Composition of Dialogue Systems

A SDS is typically a modular system consisting of the sub-components shown below in fig. 1. This modular design allows individual components to be researched and experimented on in isolation, removing the possibility of confounding effects of system-level interactions.

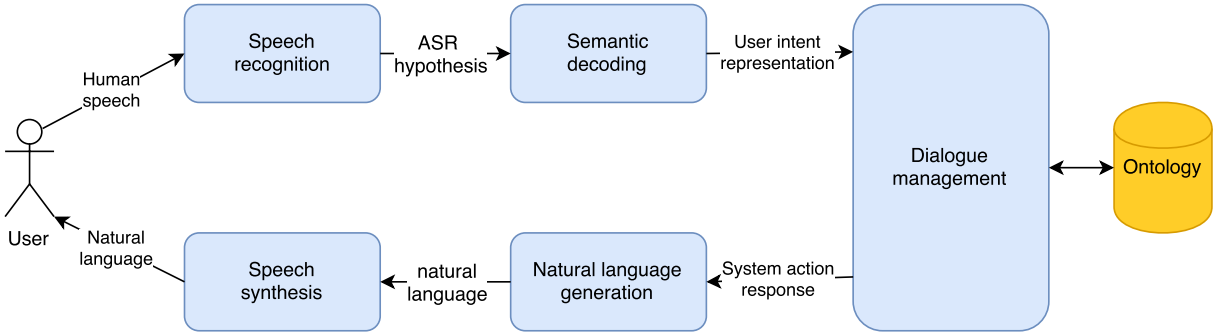


Figure 1: High level components of a SDS

Speech Recognition

Either initialising the interaction episode themselves, or being prompted by the system the user makes a spoken natural language query to the system. This waveform data is interpreted by the automatic speech recognition system, which is able to convert the user's utterance into written text. Currently ASR do not perform perfect recognition of speech, particularly in noisy environments in which SDSs are typically used, meaning that errors can still be present in the transcribed speech. To get around this problem SDS ASR modules will output an n-best list of multiple hypotheses representing a number of possible transcriptions of the user's speech.

Semantic Decoding

The natural language hypotheses of the ASR unit are passed on to the *semantic decoding module*, also termed the *natural language understanding* (NLU) module. This modules processes the current hypothesis set, possibly in the context of what has been said before, and produces a machine-readable representation of the users intention. Such a conversion is shown in fig. 2 where a free-form natural language query is converted into a structured, machine readable semantic representation of the user's intention.

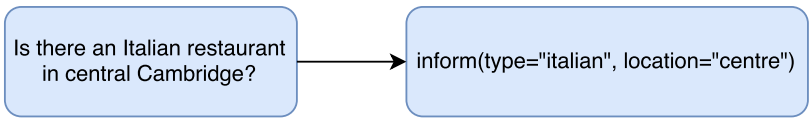


Figure 2: Conversion of ASR hypothesis to internal intention format

Ontology

The ontology defines the *domain* of the SDS, and consists of two elements. First is a record of entities about which we wish to perform queries on, which typically takes the form of a structured database. Second are the properties available to a particular domain, which will consist of a list of *requestable* and *informable* slots, which are a list of database columns which the user can request

information about, and inform the system about respectively. These two together define the set of possible master actions.

An ontology can be seen as defining the entities and related properties that the system can understand and talk about. This is important in *task-orientated* dialog systems in which it is assumed that the user has a desire to perform a particular task that the system must fulfill. The ontology defines the domain and possible tasks that can be performed in it.

An overview of permissible actions is contained in appendix A, it is recommended that the reader briefly familiarise themselves with the anatomy of system actions.

Dialogue Management

The user intent representation forms the input of the dialog management module which has the responsibility of selecting the system response. The internal structure of this module is shown in fig. 4. The user intent representation from the NLU is first passed to the *belief tracker*. The responsibility of the belief tracker is to take the system form of the user intent and convert it to a probabilistic representation of the system's belief about the user's intention throughout a dialogue episode [53]. The belief tracker can utilise information from previous turns, and so updates its ongoing hypothesis as each new user intent arrives. The belief state is represented in the *cued-pydial* system as a dictionary of slots to slot values. The names of slots, and their number of possible enclosed values is extracted from the ontology. A slot can represent a distribution over the value of a particular *goal* (what a user intends to do), the probability of a particular *method* being employed by the user (how the user intends to do it), and a distribution over *requested slots* (what info user has asked for). The correct modeling of the belief state over time is a lively area of research [19], however for this work only the baseline *focus* tracker is used. There has been work on moving away from explicit distributions over ontology-extracted slots towards implicit distributed representations as produced from a neural network [20], and to show how the proposed approach could work in such a system, we treat the focus tracker belief state as a fixed length vector representation).

The belief state acts as input to the *policy optimiser* which is responsible for learning how to best select the next system action. From the ontology the set of all possible actions are extracted, which is termed the *master action set*. The master action set includes all *inform* responses for requested entity information, all possible *select* actions requesting user-clarification on pair-wise alternatives for a particular slot, *confirm* actions for each slot-name-value combination, and an *inform* action for each entity in the ontology. For the TableTop (TT) domain this dissertation will examine, this results in a master action set size of 5025, mostly consisting of *select* actions requesting clarification on ambiguous slot values. The TT domain is relatively small, and other SDS domains will have numbers of actions orders of magnitude larger.

In the current system, in order to reduce the number of different actions that need to be enumerated, the master action set is projected down into *summary action space* by the *summary action function* to produce the *summary action set* which is of size on the order of tens. For the tabletop domain, all summary actions are:

- *request_⟨requestable-slot⟩*
- *confirm_⟨requestable-slot⟩*
- *select_⟨requestable-slot⟩*
- *inform*
- *inform_byname*
- *inform_alternatives*
- *inform_requested*

where ⟨*requestable-slot*⟩ is one of *area*, *food*, and *pricerange*.
The policy optimiser evaluates all possible summary actions against the current belief state and selects the action that it believes is the "best" action (i.e. that most swiftly leads to the user achieving their goal) to perform given the beliefs it holds about the user's intentions.

Once a summary action has been selected, it is passed to the summary action function which looks inside the belief state to decide how to project back into master actions space. An example of the construction of a master action is shown in fig. 3. Here the policy manager has selected the summary action `select_food` which prompts the user to select between which food is intended given current ambiguity over slot values. This summary action and the current belief state are passed to the summary action function. Given hand written rules it can be ascertained that a `select` method over `food` requires that the user is asked to select between the two slot values under the food slot that have greatest uncertainty, here these are `indian` and `chinese` which allow the master action to be constructed.

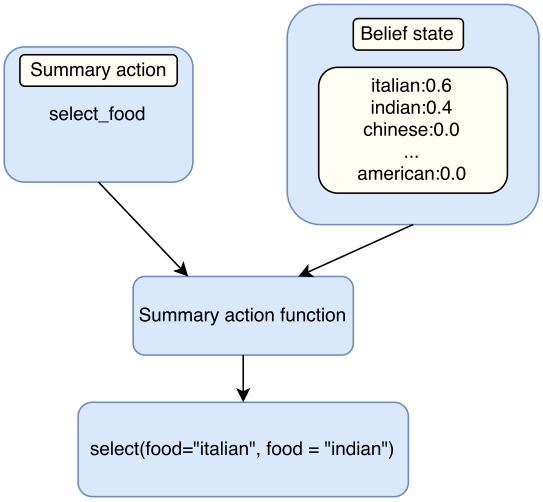


Figure 3: Transformation of summary action into master action space. We view a reduced belief state, viewing only a proportion of the *food* slot values

This flow of information within the dialogue manager is shown in fig. 4. The above data flow shows the two shortcomings of the summary action mapping approach. Firstly specific rules are required to be written for each action type. In the situation above the summary action function was hand-coded to recognise that `select` action requires the disambiguation between slots. For any new action type that is added to the system, additional rules will have to be added as to how the belief state is to be interpreted. Secondly, the summary action function relies on the belief state being explicitly interpretable which is the case for the slot-name slot-value format adopted by the currently used belief state tracker, but may not be the case for, *e.g.* neural network and other distributed representation schemes. Removing the need for the summary action function, and therefore removing the above obstacles to scalability and applicability of new methods is necessary for the advancement of SDS.

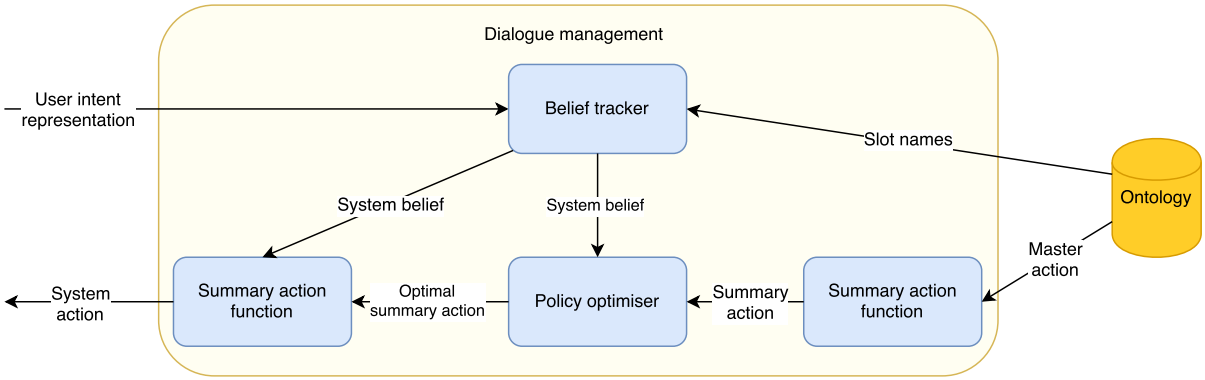


Figure 4: Dataflow within summary action-based dialogue manager. Note the complexity involved in calculating the action, and how the system in not well de-coupled

In contrast to other components in SDS which are supervised learning problems, dialog management is an exploratory problem without a fixed set of training data, a very different approach to the optimisation of other modules.

Natural Language Generation

The system action decided by the dialogue management module forms the input of the natural language generation (NLG) module. In this unit, the machine-representation of the action the system wishes to perform is converted to a human-understandable natural-language textual format.

Speech synthesis

Once a natural language format of the system action has been constructed it is passed to the speech synthesis module. This module takes the textual natural language of NLG and converts it to its audialised form, this waveform can then be played to the user.

Policy Optimisation

Of the above modules, dialogue management is of direct interest to our problem, specifically the learning of a *policy*, π that selects the best system action, a, from the set of all possible actions $\mathcal{A}$ to perform given a system belief state, s:

$$\pi(s) = \arg\max_{a \in A} \mathbb{E}_{\pi}(R(s,a))$$

Here $R(s,a) = \sum_{k}^{t=0} r(s_t, a_t)$ is a sum over all time-steps, where $r(s_t, a_t)$ is a measure of the immediate utility of taking action a_t in state s_t, and $R(s,a)$ therefore measures longterm, overall utility of choosing to perform action a when in state s. Because we know that a dialogue system proceeds in a given number of steps (i.e. it is *episodic*), k is known, and so we do not need a discounted measure of expected utility.
For dialogue systems a sensible method of assigning utility to an action is whether it leads to successful completion of the dialogue in as few steps as possible, and is therefore only measurable at the end of dialogue episode. The expectation over the policy $\mathbb{E}_{\pi(.)}(R(s,a))$ can be understood as saying: given we take action a now, what will the accumulated utility be assuming we keep taking actions decided upon by policy π.

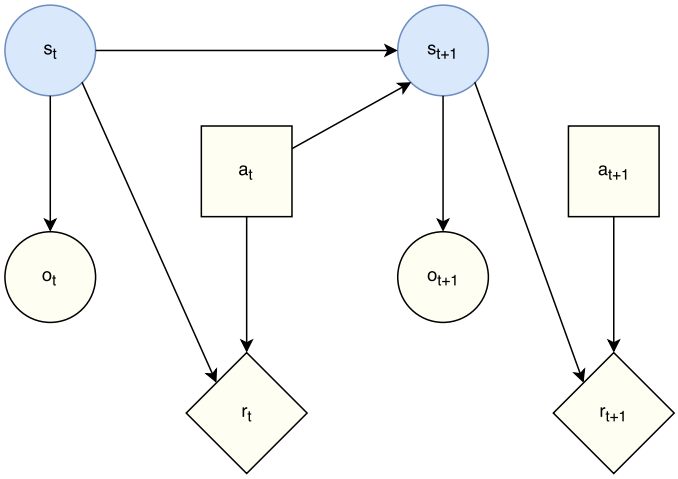


Figure 5: POMDP formulation of SDS policy optimisation

Due to speech understanding errors, and potential misunderstandings of the user, we cannot exactly track the current state of the user's intentions, and therefore we can represent graphically the above formulation as in fig. 5. The user's goals are s_t, the output of the NLU is o_t the system action selected is a_t and the immediate utility r_t. The belief state can therefore be understood to be a distribution over all possible user intentions. Arrows represent causal factors, unshaded shapes represent observed variables, and shaded shapes represent hidden variables.
This formulation is recognisable as a partially observable Markov decision process (POMDP) which is formalised as a 7-tuple [1]:

$$(\mathcal{S}, \mathcal{A}, P_T(\cdot|\cdot,\cdot), r(\cdot,\cdot), \mathcal{O}, P_O(\cdot|\cdot), h_0)$$

where:

- $\mathcal{S}$ is the set of all states ($s_t \in \mathcal{S}$), which represents all possible discrete intentions of the user. This is the user's true intention.
- $\mathcal{A}$ is the set of master actions ($a_t \in \mathbb{A}$)
- $P_T(s_{t+1}|s_t, a_{t-1})$ defines the probability of transitioning to hidden state s_{t+1} given we were in h_t and performed action a_t, this models how the user's intention responds to system actions
- $r(s,a)$ is the immediate reward for performing action a in state s, which for SDS will looks something like $r(s,a) = R_{success}$ for actions that directly cause the completion of a dialogue, and $r(s,a) = -R_{penalty}$ for all other actions.
- $\mathcal{S}$ is the set of all observable states ($s_t \in \mathcal{S}$), which is the output of the NLU.
- $P_O(o_t|s_t)$ is the defines an observation probability distribution which models how observed states are produced given a hidden state. This models how a user articulates their intention, how this is decoded by ASR and encoded by the NLU. For the general POMDP problem the probability will also be conditioned on the action taken at time $t-1$, however for our problem the belief state is not directly conditional on the system action taken at the previous step.

Policy optimisation in POMDPs is intractable [22], and while there exist approximate policy optimisation methods making assumptions specific to the SDS problem (see [40], [54]) they require the hand-factorisation of the hidden state inducing bias. However, once we notice that our belief state b_t is a distribution over the discrete hidden state s_t of the POMDP given the observation $b_t = P(s_t|o_t, b_{t-1})$, we can reformulate our problem as a Markov decision process (MDP) with continuous states, which is illustrated in fig. 6.

[1] As explained above for the SDS problem there is no need for a discounting factor γ

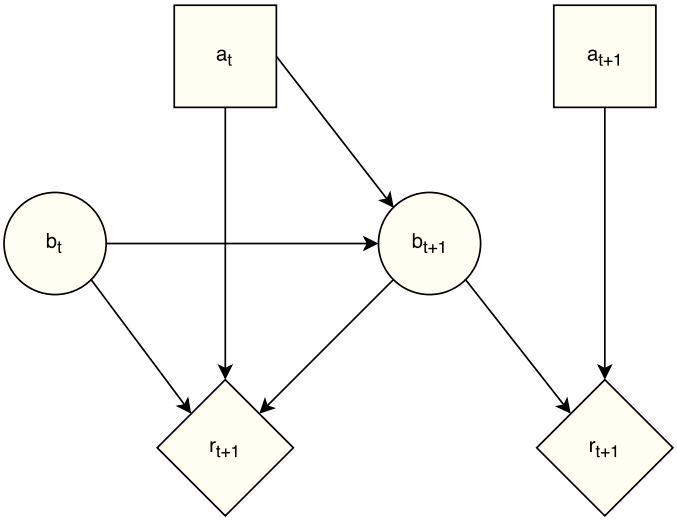


Figure 6: Graphical representation of the MDP problem. Rewards depend on the current belief state, the action and the resultant belief state.

All variables are now observed and our sequence of system belief state forms the respective b_t values; rewards are now dependent on the state and action and are R_{success} for actions that result in successful completion of the dialogue, and $-R_{\text{penalty}}$ for all non-terminus-inducing actions; a_t are the system actions.

Reinforcement learning

With the formulation of SDS policy optimisation as an MDP, we can now introduce reinforcement learning (RL) to learn a policy that selects actions so as to maximise expected reward. We define our deterministic action selection policy to be: $a = \pi(b)$, where a is a discrete action and b is the belief state.

In RL we use training information to learn how to evaluate actions rather than learning directly how to predict correct actions. By learning functions that approximate the expected long-term reward of being in a certain belief state, or taking a certain action in a state, we can define a policy so as to maximise our estimates of the long term reward, allowing both successful exploration and exploitation of our state space while implicitly learning to control the environment.

We fist define the total reward an agent receives at and after time t:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

This says that we total up all subsequent rewards, down-weighting all future rewards by using the exponential discounting factor γ^k, where $0 \leq \gamma \leq 1$ which penalises rewards futher in the future the lower γ is. Discounting assures that for infinite window MDPs we have a finite total reward, but as explained above, can be ignore for our problem.
Given this notion of long-term reward, we can define the *value* of a state, b, under a given policy π:

$$V^{\pi}(b) = \mathbb{E}_{\pi}\left\{\sum_{k=0}^{n} r_{t+k+1}\right\}$$
$$= \int_{b'} p(b'|b, \pi(b))\left[r(b, \pi(b)) + V^{\pi}(b')\right] db'$$

Which is the *Bellman equation* for value function V^{π} for a fixed horizon of length n. The Bellman equation relates the value of a state with the value of its successor states by averaging over all possible future possibilities of reward weighted by their probability. Note that we integrate over states since they are continuous, for the discrete case this would be replaced with a summation over states.

We can also look at the Q-function that maps a state action pair to the respective expected long-term reward under a given policy:

$$Q^{\pi}(b, a) = \int_{b'} p(b'|b, a)\left[r(b, a) + Q^{\pi}(b', \pi(b'))\right] ds'$$

We can then use the Q-function to define our policy:

$$\pi(b) = \arg\max_{a} Q(b, a)$$

The defined policy amounts to choosing the action that maximises long-term expected reward from a state. We can do this without requiring an explicit environment model as would be necessary when only using the value function.

There are many ways of learning the Q-function [49] with temporal difference (TD) learning making up an important subclass of approaches. In TD for learning the Q-function, our approximation is sequentially adjusted using the update:

$$Q(b_t, a_t) \leftarrow Q(b_t, a_t) + \alpha\left[r_{t+1} + \gamma Q(b_{t+1}, a_{t+1}) - Q(b_t, a_t)\right]$$

Essentially at each update the Q-function estimate for a particular state action pair (b_t, a_t) is adjusted so that it greater resembles the target value of $r_{t+1} + \gamma Q(b_{t+1}, a_{t+1})$ which is the immediate reward plus the discounted future award. The rate at which adjustment occurs is controlled by the α parameter, analogous to a learning rate. For GPSARSA $\alpha = 1$.

Now we have a rule of how to update our Q-function estimates, we can derive an online, exploratory learning algorithm:

Algorithm 1 SARSA on-policy algorithm

procedure SARSA
  $Q(b, a)$ randomly initialised $\forall b, \forall a$
  while not finished **do** ▷ Typically iterate for a fixed number of episodes
    $b_t \leftarrow$ belief state tracker output
    $a_t \leftarrow \epsilon$-greedy policy decision from Q
    for t **in** episode **do**
      perform a_t observe r_{t+1} and b_{t+1}
      $a_{t+1} \leftarrow \epsilon$-greedy policy decision from Q
      $Q(b_t, a_t) \leftarrow Q(b_t, a_t) + \alpha\left[r_{t+1} + \gamma Q(b_{t+1}, a_{t+1}) - Q(b_t, a_t)\right]$
      $b_t \leftarrow b_{t+1}$
      $a_t \leftarrow a_{t+1}$

The ϵ-greedy policy decision is defined as:

$$\pi(b) = \begin{cases} \arg\max_a Q(b,a), & \text{with probability } 1-\epsilon \\ \text{randomly sample an } a, & \text{otherwise} \end{cases}$$

Which balances the needs of exploitation (using the accrued knowledge) versus exploration (trying actions/ state pairs that have not yet been explored).

The SARSA algorithm as it is presented above stores an explicit value for each (b, a), however this is only possible for small finite state and action spaces. Instead we apply *function approximation*, and treat learning the TD-updated values of the Q-function as a regression problem so we can generalise to unseen areas of state-action space.

Gaussian process regression

In the problem of regression we seek to learn a mapping $f : \mathcal{X} \to \mathcal{Y}$ which takes a point from some input space and maps to a point in some output space. Returning to Q-function approximation we see that we can treat this as a function $Q : \mathcal{B} \times \mathcal{A} \to \mathcal{R}$ mapping from belief state-action pairs to a real which represents the expected reward.

By using Gaussian processes [41] we can define a distribution over functions which is uniquely defined by its prior mean function and covariance function:

$$f(x) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}'))$$

where

$$\begin{aligned} m(x) &= \mathbb{E}(f(\mathbf{x})) \\ k(\mathbf{x}, \mathbf{x}') &= \mathbb{E}\left[(f(\mathbf{x}) - m(\mathbf{x}))(\mathbf{x}' - f(\mathbf{x}'))\right] \end{aligned}$$

and in which we define any finite subset of the target random variables to be from joint a Gaussian distribution. For the rest of this discussion we assume that $m(\mathbf{x}) = \mathbf{0}, \forall x \in \mathcal{X}$.

Kernels

In the real world observations have noise. For the SDS's reward this noise is introduced by the system being unsure if successful completion has occurred, due to either incorrect user feedback or inaccurate automatic evaluation. We therefore assume that each observation has a component of additive noise, $y = f(x) + \epsilon$, and we assume each ϵ is independent and identically distributed Gaussian with mean 0 and variance σ^2. Let us define $\mathbf{X}_*$ to be a set of unseen datapoints for which we wish to predict the target values $\mathbf{Y}_* = f(\mathbf{X}_*)$ given that we have seen the training examples $\mathbf{Y} = f(\mathbf{X}) + \epsilon$ by the definition of GPs we have the joint distribution to be:

$$\begin{bmatrix} f(\mathbf{X}) \\ f(\mathbf{X}_*) \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(\mathbf{X}, \mathbf{X}) + \sigma^2 I & K(\mathbf{X}, \mathbf{X}_*) \\ K(\mathbf{X}_*, \mathbf{X}) & K(\mathbf{X}_*, \mathbf{X}_*) \end{bmatrix}\right)$$

where $K(\mathbf{X}, \mathbf{X}_*)$ denotes the matrix of the covariances between each repsective point in $\mathbf{X}$ and $\mathbf{X}_*$. We can now simply construct the conditional distribution of the desired test targets given the test and training input, and the training target values:

$$f(\mathbf{X}_*) | \, \mathbf{X}_*, \mathbf{X}, f(\mathbf{X}) \sim \mathcal{N}(\mu, \mathbf{\Sigma})$$

where

$$\mu = K(\mathbf{X}_*, \mathbf{X}) \left[K(\mathbf{X}, \mathbf{X}) + \sigma^2 I\right]^{-1} f(\mathbf{X}) \tag{1}$$

$$\mathbf{\Sigma} = K(\mathbf{X}_*, \mathbf{X}_*) - K(\mathbf{X}_*, \mathbf{X}) \left[K(\mathbf{X}, \mathbf{X}) + \sigma^2 I\right]^{-1} K(\mathbf{X}, \mathbf{X}_*) \tag{2}$$

The covariance function $k(\cdot, \cdot)$, also termed the *kernel* controls the modeling behaviour of the Gaussian process and many exist for different scenarios[41]. In the current dialog system a *squared exponential* (SE) kernel is used, where:

$$k(b, b') = \sigma^2 \frac{(b - b')^2}{\ell}$$

which has been shown [29] to be able to approximate any arbitrary continuous function[2]. The parameters σ and ℓ control the functions deviation from its mean and length between major changes in its direction respectively. This is the kernel that is used on the belief state in the current dialog system. The SE kernel can not be used on summary actions, since they form a discrete domain, and instead the *Kroenecker delta* function is used:

$$k(\mathbf{a}, \mathbf{a}') = \begin{cases} 1, & \text{if } \mathbf{a} = \mathbf{a}' \\ 0, & \text{otherwise} \end{cases}$$

In order to combine the two kernel values, we multiply them together:

$$k((\mathbf{b}, \mathbf{a}), (\mathbf{b}', \mathbf{a}')) = k(\mathbf{b}, \mathbf{b}')k(\mathbf{a}, \mathbf{a}')$$

Essentially this amounts to *AND*ing their values: the resulting value will only be large if both values are large. However when using the Kroenecker delta function the multiplication serves as a *switch* so that when calculating the mean of the resultant Gaussian distribution μ and $\mathbf{\Sigma}$, we construct an estimate that is a linear smoothing of all belief states that were encountered for a specific action. This is equivalent to having a separate GP over each summary action, and no "transfer of knowledge" is possible between state-action examples that have different actions, and the approach quickly becomes impractical as the size of the action set increases.

Sparse representative points

The calculation of the mean and covariance is typically dominated by the inversion *Gram matrix* $K(\mathbf{X}, \mathbf{X})$ which is an operation that is $\mathcal{O}(n^3)$ where n is the number of training points. This soon becomes infeasible, and so we enforce a sparsity inducing constraint whereby we only add an additional point to a dictionary of representative points if it is suitably distinct from other points in the dictionary. This allows us to maintain a representative set of points of size m which we use to approximate our test points. This reduces Gram matrix inversion to $\mathcal{O}(m^3)$, where $m << n$. In addition, since we add a single datapoint, $\mathbf{x}$ to our GP dictionary, $\mathbf{X}$ at the time, we can utilise the incremental matrix inversion identity:

$$\begin{bmatrix} \mathbf{K} & \mathbf{k} \\ \mathbf{k}^\intercal & k \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{I} & \mathbf{k}k^{-1} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{K} - \mathbf{k}k^{-1}\mathbf{k}^\intercal & \mathbf{0} \\ \mathbf{0} & K(\mathbf{x}, \mathbf{x}) \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ k^{-1}\mathbf{k}^\intercal & 0 \end{bmatrix}$$

where $\mathbf{K} = K(\mathbf{X}, \mathbf{X}) + \sigma^2 I$, $\mathbf{k} = K(\mathbf{X}, \mathbf{x})$ and $k = k(\mathbf{x}, \mathbf{x})$. This has the advantage that only $k(x, x)$ needs to be inverted, which amounts to division of a real, and therefore drastically reduces computation time.

[2] assuming smoothness of the target function

Given the resultant Gaussian distribution at a certain point we can choose to either select the mean or draw a sample. In this work we choose the sampling based approach, this encourages exploration, particularly when the belief state does not change between turns and the same action would be repeated multiple times otherwise.
The Bayesian treatment of modeling afforded by the GP approach has two notable benefits:

1. robustness to overfitting
2. quantification of uncertainty

Robustness to overfitting is a quality due to "Bayesian/ automatic Occam's Razor" [42]. This allows a model to be trained in limited-data environments and yet to maintain good generalisation to unseen examples. This is particularly important in dialog systems, where training data is either limited, or non-existent in which case live-interactions with real users is necessary. Performing training on real-world users is expensive and so we want to minimise the number of training examples needed to make a robust model estimate.

The quantification of uncertainty is due to GPs providing a Gaussian distribution over values rather than a point estimate. Using the variance of the resulting distribution we can quantify the models uncertainty about the predicted value. This is highly useful to reinforcement learning, since when performing an random action under ϵ-greedy exploration, we can now select the action with the highest variance to perform. This approach allows our RL algorithm to actively explore in areas in which we are least confident about, and therefore could potentially harbour the best potential improvements. Such approaches in which the model suggests potential candidates to learn from is termed *active learning*. As well as choosing the most uncertain regions, there exist other methods of selecting a certain point to evaluate called *acquisition functions* [44], which afford different exploration strategies.

Using a GP to approximate the Q-function in SARSA leads to the *GPSARSA* algorithm introduced in [14], and formalised for the SDS domain in [15]. High level pseudo-code is provided below in algorithm 2:

Algorithm 2 GPSARSA

procedure SARSA
    Define GP prior for Q-function
    for d **in** dialogues **do**
        $b_t \leftarrow$ initial belief tracker estimate
        $a_t \leftarrow \epsilon$-greedy policy decision from Q
        for t **in** turns of d **do**
            perform action a_t, observe r_{t+1} and b_{t+1}
            $a_{t+1} \leftarrow \epsilon$-greedy policy decision from Q
            if (b_t, a_t) is representative **then**
                update GP posterior estimate of Q according to $((b_t, a_t), r_{t+1})$
            $b_t \leftarrow b_{t+1}$
            $a_t \leftarrow a_{t+1}$

Related Work

The desire to move to performing policy optimisation in the full action space progresses the work in [16] in which the authors showed how it was possible to perform GPSARSA in the full belief space. However this work still used the summary action space.

Recent work in [48] examined how to user recurrent neural networks (RNNs) to perform policy optimisation in master action space. They propose a *Policy Network* which consists of an RNN with its output layer divided into partitions that predict a certain part of the system action. A depiction of the system is shown fig. 7. The belief state is fed into the RNN which outputs a distribution over intentions, a distribution over informable slots, and a probability that each particular requestable slot is present. The RNN therefore outputs a "stub" of an action consisting of an intention and the contained slot names. The values for the respective slot values are attained by examining the slot values in the system belief state and verifying them agains items in the ontology.

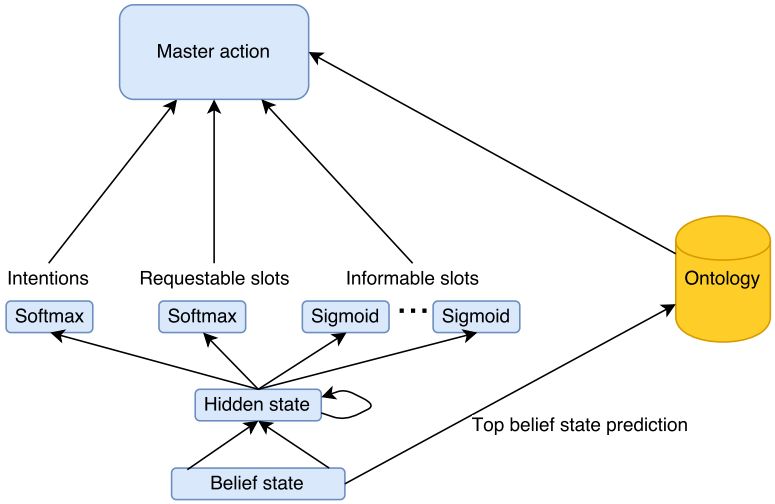


Figure 7: Neural dialogue management using RNNs

The training of this system proceeds in two phases. In the first phase the model is trained on supervised training data obtained from a Wizard of Oz experiment, and in the second phase the model parameters are optimised using the RL *natural gradient* method.

While showing promising performance, and the ability to adapt to mismatch domain it suffers from a number of drawbacks. Firstly, the need for training data restricts the ability to roll solutions out to multiple domains, additionally although only a small number (≈ 100) training examples are needed for the small problem attempted, requirements may be greater for larger domains. Secondly, there is still the need to explicitly examine the belief state in the construction of the master action. This may not be possible if the belief state is a non-interpretable structure such as a RNN hidden state. Finally, the structure of the output is fixed, so that if a new intention, requestable slot, or informable slot the model must be retrained from scratch. This inability to adapt to changes in structure of the dialogue domain is a consequence of the parametric approach taken.

Outside of the field of SDS, some work has been done on RL with large and infinite action sets.

In [39], the authors identify the selection of the optimal action as the impediment to performing RL in large action spaces. The authors generalise the notion of value functions $V(b)$ and state-action

functions $Q(b, a)$ to a set of functions to provide the generalised Bellman equation:

$$X_u^*(b) = \max_{a \in u} R(b, a) + \gamma P(b'|b, a) \max_u X_u^*(b') \tag{3}$$

where $u \in u_1, \ldots, _n, u_i \subseteq \mathcal{A}$ is a set of actions, which each action potentially occurring in more than one action set u_i, and X_u^* is the generalised value function. The above equation defines the generalised value function value to be the maximum discounted reward achievable by selecting an action a from a subset of actions u. It can be seen that by setting $u = \mathcal{A}$ we arrive at the common value function that performs a maximisation over all possible actions, and by having each action subset u_i consist of a different unique action we arrive at the Q-function. The authors show how it is possible to formulate the optimisation of the Bellman equation as a linear program (LP), however the solution requires summation over all possible state values, as well as there a constraint existing for each $\mathcal{B} \times \mathcal{U}$, which can be prohibitively large for large discrete states, and intractable for continuous action spaces such as the SDS policy optimisation problem. The authors go on to provide a specific instantiation of the generalised value function, termed the H-function, in which each action is envisioned to be at the vertex of a hypercube[3], and each action consists of the set of vertices on a face of the hypercube. Each dimension of the hypercube now partitions the data into two (possibly not disjoint) sets, and by comparison of the resultant H-function values between all pairs of sets along each dimension, we can find highest valued action by taking the union of all the highest-valued action sets. A hypercube with $|\mathcal{A}|$ vertices has $\log_2 |\mathcal{A}|$ dimensions, and so the search for the optimal action has had its temporal performance reduced from $\mathcal{O}(|\mathcal{A}|)$ to $\mathcal{O}(\log_2 |\mathcal{A}|)$. While this is a considerable improvement, the LP solution is not suitable for large/ infinite state spaces.

The decomposition of the action space as introduced above is a common theme in dealing with large action spaces. In [13], the authors also use binary splits of the action space, by assigning error-correcting output codes [11] (ECOC). ECOC is a multi-class classification approach that assigns to each output label $\{y : y \in \mathcal{Y}\}$ a binary code of length $C = \gamma \log_2(|\mathcal{Y}|)$[4]. The task is then to predict the code, rather than directly predicting the for a particular label. This reduces a classification problem with $|Y|$ classes to a set of C separate binary classification problems, the outputs of which form what the system predicts the binary representation the guessed class will be. The actual predicted class can then be outputted by finding the action that is closest to the predicted binary representation under the Hamming distance. The authors use Rollout Classification Policy Iteration[24] (RCPI), policy iteration approach that generate training examples by using Monte-Carlo (MC). These examples are then used to train a policy that directly classifies the optimal action in a particular state, thereby posing policy optimisation as a classification task. The contribution of [13] is to view the prediction of each bit in ECOC as a separate MDP and solve using RCIP. The authors apply the approach to problems with action space sizes up to $\mathcal{O}(100)$ and small discrete action spaces (250 states) and a 2d continuous state space. It is not clear how such an MC based approach would scale up to larger state spaces, and whether the ECOC-based approach would generalise well in situations where every possible action is not seen. The MC-based approach is particularly poor for systems that may wish to interact with real-world users, since multiple dialogue runs would be infeasible.

More recent work on large, discrete action sets has centered around deep reinforcement learning [31]. In [18] the authors look at the specific problem of dialog systems where the state space consists of textual strings, and actions consist of direct text responses to the current state. The action space is therefore an unbounded discrete space. They propose a deep reinforcement relevance network (DRRN). The DRRN consists of two separate DNNs, which map the state and action strings into

[3]Actions can appear more than once if the number of actions is not consistent with that needed to form a hypercube

[4]The γ integer coefficient causes the representation length to be longer than the minimum required to represent all actions uniquely, thus introducing redundancy

fixed length vector representations respectively. A dot product between these two representations then forms the system's approximation of the Q-function for that particular state-action pair. Although it is promising to extend Deep-Q learning to unbounded spaces, the work still relies on a small number of possible target actions that can be evaluated against the current system state. In addition, the use of neural networks with point-estimated parameters discounts usage in low-data environments.

By using the *actor-critic* method [3], and therefore modeling $Q(b, a)$ and π explicitly, [12] construct the *Wolperting* architecture. In this approach the discrete actions are first embedded in a multidimensional state space $\mathbf{R}^n$ using knowledge available a priori. A function is then defined $f_{\theta^\pi} : \mathcal{B} \rightarrow \mathbf{R}^n$ that maps a state to a *proto-action* in the action space, corresponding to the *actor*. A k-nearest neighbour search is then performed over all actions (an approximation of which is possible in time logarithmic in the number of actions [36]) to select the k actions $\mathcal{A}_k$ that most resemble the predicted proto-action. The *Wolperting* algorithm then scores these actions using the critic, $Q_{\theta^Q} : \mathcal{B} \times \mathcal{A} \rightarrow \mathcal{R}$, and chooses the action that maximises the Q-value at the current state: $\hat{a} = \max_{a \in \mathcal{A}_k} Q_{\theta^Q}(b, a)$. ($f_{\theta^\pi}$ and Q_{θ^Q}) are optimised using Deep Deterministic Policy Gradient [26]. This is an interesting approach that leverages the benefits Deep-Q learning in continuous actions spaces. However it is unclear how a dialog action can be embedded into a continuous space, this approach also suffers from the drawbacks of using parametric models in low-data environments.

Continuisation of discrete action problems under an actor-critic framework using neural network approximation is also used in [51], which employs continuous adaptation of ACLA[52]. A real valued action is outputted by the actor, and this is then rounded to the nearest whole number which is taken as the discrete action. This approach can only be applied to discrete action spaces that have an ordinal interpretation, which is not usable in a dialog system's action space.

Although not directly applicable to our domain, more work has been done on RL in infinite ("large") continuous actions spaces and some work is briefly reviewed here.

[38] extend ideas due to [8] of decreasing action space complexity at the cost of state space complexity for discretised-continuous-action discrete-space problems. The state space is augmented so that action selection itself becomes an MDP. For a particular state b_i to which all possible actions are associated, we choose from one of two actions, each action leads to a new state b_i^1 and b_i^2 respectively, to each of which is associated half of all the actions. This decision process is repeated until all actions have been whittled down to a single action that becomes the action to perform at b_i in the original MDP. This formulation constructs a new MDP where the states form a binary search tree at each state in the original MDP which recursively partition the original actions, and induced actions correspond to binary decisions about which half of the remaining actions to examine. This strategy decreases an $\mathcal{O}(|\mathcal{A}|)$ sweep of all actions to a $\log(|\mathcal{A}|)$ search in a binary tree of height $\log_2(|\mathcal{A}|)$. While an impressive reduction in action selection complexity, only applications to discretized continuous action-space problems are presented where the partitioning of actions follows from splitting around the median value. It is hard to see how this idea can be directly applied to discrete actions.

CACLA [50] extends ALCA[52], which uses a tabular representation for the actor and critic, to the continuous case by using a function approximator for the actor and critic. The use of the Wire Fitting algorithm due to [1] enables a set of actions and associated Q-values to be predicted. However such a tabular-based approach is inappropriate for large action space

[25] extend the actor-critic approach by using sequential Monte-Carlo (SMC) sampling to maintain a finite set of sample actions, thereby constructing their policy. Initially their method promotes exploration by drawing samples from a prior, but over time the policy actions are drawn to regions that have proven to yield high rewards. SMC sampling-based action selection processes are only

possible in continuous actions spaces where there is a notion of a continuous *neighbourhood* around points that can be explored. This is not possible in a discrete actions space where there is no notion of similarity between actions.

The actor-critic method has previously been attempted in SDS, but it was found to be prohibitively slow[23].

Given the acclaimed successes of Deep-Q learning [30] [45], promising work on Bayesian approaches to neural nets [4] is very welcome. The authors cite as specific motivation the desire to quantify uncertainty in estimates for use in active learning approaches to RL. With the development of this emerging approach the neural network approaches outlines above could be adapted to our limited-data environment.

Another relevant area of RL is *hierarchical reinforcement learning*[2]. In hierarchical-RL we seek to decompose our action set into an action hierarchy. In this hierarchy, actions higher up act as *macros* for, and get expanded to, (possibly sequences of) actions lower down the hierarchy. This has been used for decomposing the action domain to in policy decision [9]. In the MAXQ approach to hierarchical RL[10], a hierarchical structure is constructed in which an internal *Max* node represents a separate policy that is used to select from it's actions. This system of separate policy decisions is similar to [38] in which also composes action selection into a series of separate MDP, however in MAXQ we do not transition to a state that represents a subset of actions, we transition to a state that represents a sequence of actions.

Hierarchical RL is an interesting approach to action decomposition, however in all work up to now the hierarchy of action macros is typically defined by a human programmer, and this represents a barrier to rapid deployment to new domains.

Method

One possible approach to performing GPSARSA to the full master action space is simply extend the current approach of using the Kroenecker and placing a separate GP over each action. This can work for the summary action function, since each summary action will be seen multiple times during training corresponding to different master actions, therefore introducing transfer of learning. However it is unlikely that we will see particular master action more than once, and it is very likely that we will never see most of them. It is therefore necessary to introduce some notion of similarity between master actions, so that when sampling a new point in our GP, correlations from multiple previous master actions contribute.

The common "goto" kernels (*e.g.*[41] the squared exponential, polynomial, Matérn class, Ornstein-Uhlenbeck) are all unsuitable due to taking a fixed vector of real numbers of input. Instead there exists a class of *convolution kernels*[17].

Convolution Kernels

Convolution kernels construct a kernel value over discrete structured sets by summing a product of subcomponents between them, therefore forming a convolution[17].

Action as strings

Kernels can be defined over strings, trees and other discrete structures. It is immediately obvious how a master action can be viewed as a string of lexical entities, *e.g.*

inform(area="south", food="indian", name="raj", pricerange="expensive")

becomes

$$[\text{“}inform\text{”}, \text{“}area\text{”}, \text{“}south\text{”}, \text{“}food\text{”}, \text{“}indian\text{”}, \text{“}name\text{”}, \text{“}raj\text{”}, \text{“}pricerange\text{”}, \text{“}expensive\text{”}]$$

A *string kernel*[27][7] can then be used to test similarity with another action by counting the co-occurrence of (possibly skipped) n-grams. This has a number of drawbacks. Firstly we throw away information about generality of certain classes of terms. The intention is the most informative element of the string, followed by the slot names, but under the string kernel n-grams formed of slot-values will be given equal importance as more discriminative element types. In addition forming n-grams in the induced string leads to some strange n-grams that do not convey useful information about similarity between actions. An example from above is the n-gram is ["*indian*", "*name*"], which does not convey information useful in the comparison of similarity of actions. It could be possible to enforce rules on the constructed n-grams so that they weight instances that convey more information about intentions and slot names, and also obey slot boundaries, although such a system is likely to consist of hand-crafted rules based on the position of lexical entities in the string. However there is a different view actions that allows another kernel to be adopted that automatically addresses the latter drawback, and is easily extended so that it abates the former.

Actions as trees

In fig. 8 we see two examples of how actions can be construed as trees. By viewing master actions in this hierarchical manner, we automatically impart items nearer the top, i.e. intentions and slot names, with greater importance. We also see how we capture the nested nature of actions, so that slot values are only associated with the respective slot name. Using this formulation we can now introduce *tree kernels* as a method of calculating similarity between actions. In the below implementations, we assume that a tree is represented as a `TreeNode` which has an `element` field

that represents the partial action stored at this node, and a `children` field that contains a list of all the children of a node. Leaf nodes have an empty list as children.

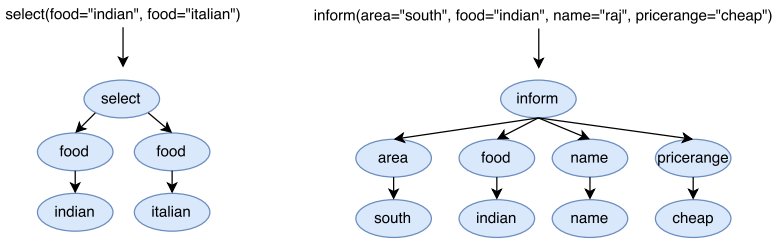


Figure 8: Example of how trees are formed from actions

Tree Kernels

Tree kernels [6][32] have been successfully applied in the natural language processing (NLP) community in areas as diverse as semantic role labeling [35], entity relation extraction [55] and question and answer classification [34].

The tree kernel operates by counting the number of common subcomponents between two trees. There are broadly two types of subcomponents that can be extracted[33]. The first are *subtrees* in which all subcomponents must contain a leaf node, an example of which is shown in fig. 9, the extraction algorithm used is shown in algorithm 3. This algorithm visits every node in the tree, adding the tree rooted at that node as a subtree, and is there of time complexity $\mathcal{O}(n)$, where n is the number of nodes in the tree.

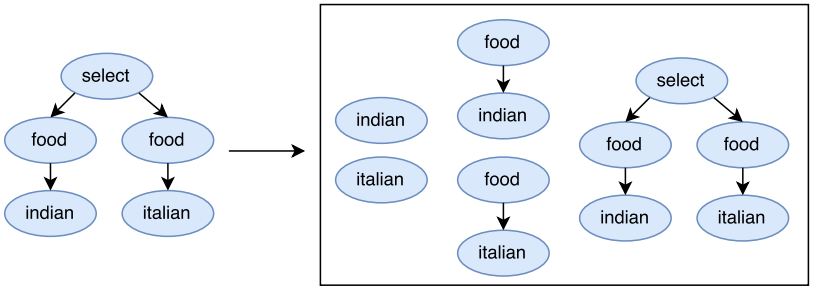


Figure 9: Example of subtrees extracted from an action

Algorithm 3 Subtree extraction algorithm

```
1: function SUBTREE(root)
2:     subtrees = [root.element]
3:     for c in root.children do
4:         subtrees += SUBSET(c)
       return subtrees
```

The second are *subset trees*, which remove the constraint that trees must contain leaf nodes, and are therefore the generalisation of subtrees, an example of extracted subset trees is shown in fig. 10, the algorithm for producing subset trees is shown in algorithm 4. This algorithm visits every node and performs work bounded by the height of the tree since our trees are balanced, and is therefore of complexity $\mathcal{O}(n \log(n))$. The spacial complexity of the subset and subtree approach is exponential

in the number of nodes in the tree. This is not a problem for the limited depth trees that we use here.

The algorithm provided here is slightly different as presented in the literature in that all paths to leaf nodes are always of a constant length (see fig. 11 for the set of unbalanced subset trees that are ignored under implementation). This greatly reduces the number of resultant subset trees, while not impacting the expressibility of the approach: since our trees of a fixed format, the omitted subset trees do not convey more information than combinations of the included subset trees.

Other works specifies both strict and non-strict trees, a strict tree being one that contains more than one node. This is typically performed to limit the feature space size. However due to the limited size of our trees and potential data sparsity issues, only non-strict subcomponents are examined in this work.

All trees have a lexical ordering enforced across children, with any ties being broken by grandchildren.

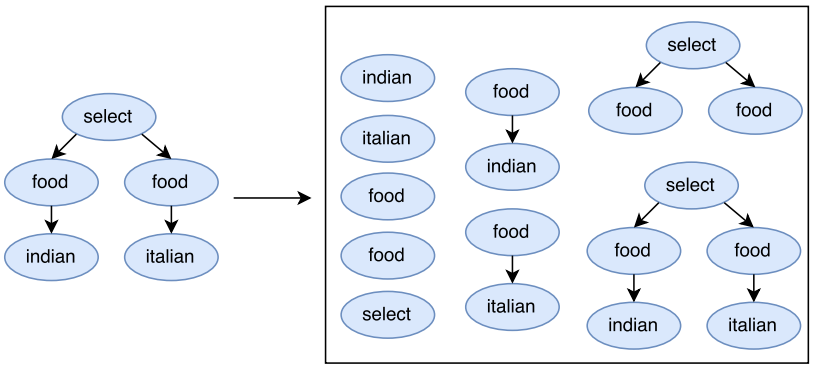


Figure 10: Example of subset trees extracted from an action

Algorithm 4 Subset tree extraction algorithm

```
function TREE_TO_DEPTH(root, depth)
    if depth = 1 then                ▷ A depth of 1 corresponds to a tree consisting of a single node
        root.children = []
    else
        for c in root.children do
            TREE_TO_DEPTH(c, depth−1)
function SUBSETTREE(root)
    subsettrees = []
    for d in {1 to DEPTH(root)} do        ▷ DEPTH() returns height of tree from provided node
        subsettrees += TREE_TO_DEPTH(COPY(root), d)
    for c in root.children do
        subsettrees += SUBSETTREES(c)
    return subsettrees
```

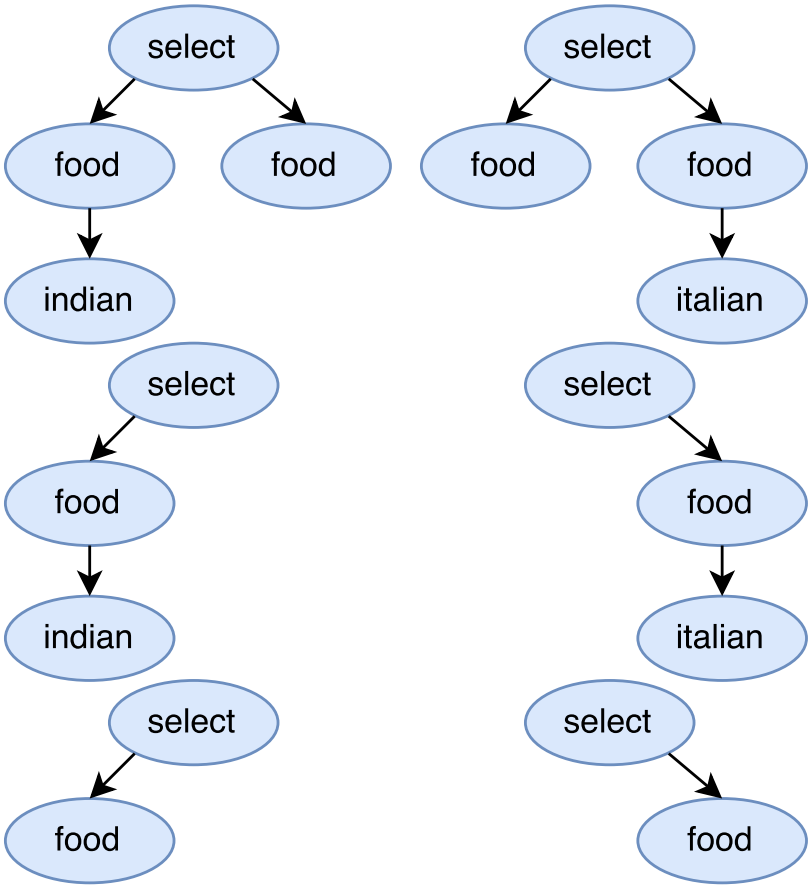


Figure 11: Subset trees omitted from this implementation

The above functions represent our feature function $\phi(x)$, and is an injective function mapping a tree into a high sparse space, $\mathbb{R}^m$ which is of dimension equal to the number of possible components with the value at each element representing the quantity of the respective component. The calculation of the kernel value between two actions x and x' can then be calculated as:

$$k(x, x') = \phi(x)^\intercal \phi(x')$$

performed on the semi-ring $(\mathbb{R}, \oplus, \otimes)$, where:

$$a \oplus b = a + b$$
$$a \otimes b = \min(a, b)$$

This has runtime complexity $\mathcal{O}(m)$, i.e. linear in the size of the feature space. Using sparse data structures [5] reduces this to $\mathcal{O}(lk)$ where l and k are the number of non-zero entries in x and x' respectively. An even more performant implementation is given in [33]. In this version the raw list of components (rather than their counts) is used and sorted at construction time, to count the number of common subcomponents the routine in algorithm 5 is used. This reduces the complexity of kernel calculation to $\mathcal{O}(l + x)$. A native C implementation of this algorithm yielded a large performance improvement.

[5] *e.g.* `csc_matrix` from scipy

Algorithm 5 Fast kernel calculation

```
1: function FAST_DOT(x, x')
2:     count = 0
3:     i = 0, i' = 0
4:     while i < length(x) and i' < length(x') do
5:         if x[i] = x'[i'] then                    ▷ Have matching subcomponents
6:             count+= 1
7:             i += 1
8:             i' += 1
9:         else if x[i] < x'[i'] then
10:            i += 1
11:        else
12:            i' += 1
       return count
```

Profiling showed that the existing implementation calculation of the dot product in the SE kernel [6] was performed using a vanilla for loop (due to the way in which the belief kernel is structured) had become a major performance bottleneck. Reformatting the belief kernel as a contiguous vector, having sorted slots corresponding to goal beliefs, and using `numpy.dot` afforded a massive speed up.

Initial experiments showed that the feature extraction as presented performed poorly. Examining the resultant kernel value pairs of actions showed that the mapping was not assigning enough importance to intentions. Similarity of intentions equates to matching a single node in the tree, and equal importance is placed on a matched slot value as for an intention, as shown in fig. 12. It can be seen that although two *select* actions should be more similar than a *select* and a *inform* the current kernel assigns a kernel value of 1 to the former, and 3 to the latter.

Weighted Tree Kernel

In order to mitigate this issue *weightings* are applied at each level in the tree, this leads to the procedure shown in algorithm 6. Instead of incrementing a counter, the weight associated with the level of the subcomponent is added to the kernel value. The example shown in fig. 12 is the worst offending case, and this can be mitigated by using the weights [3, 1, 0.5] for the intention, slot name, and slot value layer respectively.

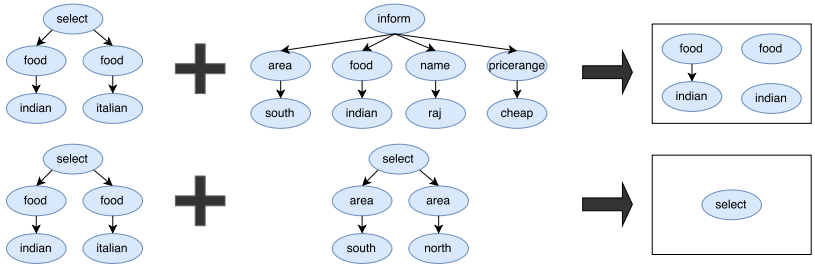


Figure 12: Non-weighted kernel emphasizes slot names and values over intentions

[6] $(x - x')^\intercal (x - x')$

Algorithm 6 Weighted ast kernel calculation

```
function FAST_DOT(x, x', w)
    count = 0
    i = 0, i' = 0
    while i < length(x) and i' < length(x') do
        if x[i] = x'[i'] then                    ▷ Have matching subcomponents
            count+= w[i]                         ▷ Weighting is applied here
            i += 1
            i' += 1
        else if x[i] < x'[i'] then
            i += 1
        else
            i' += 1
    return count
```

Reducing action selection complexity

Using the above kernels it is possible to put a single GP over the entire action space. However this introduces the problem of costly action selection. Every time the SDS system must select an action it must sequentially scan over the entire action space in order to find the the action that maximises the Q-function. Each action must be scored against each point in the dictionary, meaning that the complexity grows as $\mathcal{O}(|\mathcal{A}||D|)$ where D represents the dictionary. This computational load is prohibitively large, particularly as the dictionary grows and renders the problem intractable for all but the smallest of toy examples.

As a first step towards improving the temporal performance, a separate GP used for each intention, with each GP being defined over the belief space. When performing action selection each intention GP is evaluated at the current belief state in turn, and then only the actions consisting of the intention with the highest score are evaluated, so our policy selection criterion becomes:

$$\pi(b) = \arg\max_{a \in \mathcal{A}_j} Q(b, a)$$

in which

$$\mathcal{A}_j = \arg\max_{\mathcal{A}_i} X^*_{\mathcal{A}_i}(b)$$

where we have borrowed the notion of a generalised value function from [39](see eq. (3) in related work section above). A visualisation of this approach is provided in fig. 13. Each $\mathcal{GP}_i$ models the maximum Q-value of the contained actions. A single GP is placed over the entire action set, i.e. there is not a separate GP for each subset of actions. This allows a transfer of knowledge between actions that do not share the same intention. When performing an update for a certain belief and performed action pair, the example point is added to the dictionary for the relevant intention GP, and also to the dictionary for the GP over all actions.

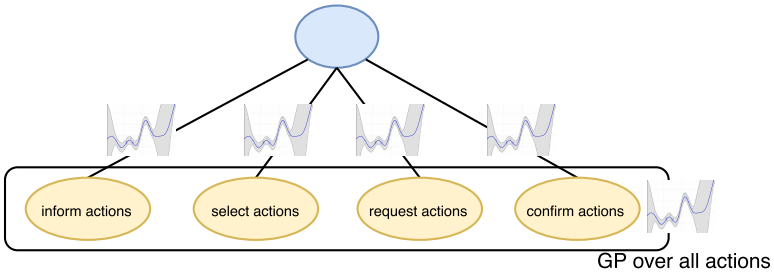


Figure 13: Placing a GP over each intention

This method follows the approach of previous works outlined above, but uses specific domain knowledge to split the action space up. Dividing the action space on intention still results extremely large action sets, with complexity now being $\mathcal{O}(max(|\mathcal{A}_i)||D|)$. The action set size per intention for the selected problem is shown in the table below:

inform	**select**	**request**	**confirm**
660	4207	4	149

Although we have a reduction in the maximum number of required action evaluations from $5025 \times 6000 = 3 \times 10^8$ to $4207 \times 6000 = 2.5 \times 10^8$, this is still prohibitively large, and so further devision of the action space is needed. One approach considered was to continue to divide actions based on domain knowledge, i.e. further split the action subsets according to slot names. The approach was discarded as it relies on a fixed action domain. Instead an approach agnostic to the specific domain was investigated.

Clustering of actions

By embedding our actions in a feature space it is now possible to perform clustering. *kernalised K-means*[21] had been previously used as a method of visualising action similarity under different kernels (see below), and to speed the development process this approach was adapted to allow hierarchical clustering of the actions.

Kernalised k-means is essentially k-means performed in feature space, $\phi(x)$. The hierarchical extension proposed here recursively clusters each successive clusters, and the high level algorithm is given in algorithm 7. The intention is to construct a hierarchy of actions in which sizes of base clusters are within a certain range. The algorithm first clusters all actions, and then recursively clusters each sub-cluster until they are below a predefined size (max_size). The algorithm then iterates over all clusters and if any are below a certain size, the constituent actions are reassigned to the closest cluster. The maximum and minimum cluster sizes are set to 50 and 10 respectively. The number of sub-clusters at each level is predetermined, and for our experiments was set to 2. Proceeding in this way, we form a hierarchical clustering of actions based on the defined kernel similarity.

Algorithm 7 Hierarchical kernalised k-means

```
function KERNALISED_KMEANS(actions, k)
    means = randomly select k actions' feature representation
    while not converged do
        assign actions to closest mean to create clusters
        update means
    return clusters
function HKKMEANS(actions, cardinality, min_size, max_size) clusters = KERNALISED_KMEANS(actions, cardinality)
    for c in clusters do
        if size(c) > max_size then
            c = KERNALISED_KMEANS(c, cardinality)
    for cluster in all clusters do
        if size(cluster) < min_size then
            for c in clusters do
                reassign c to closest cluster
```

Using the induced structure it was possible to create a binary decision tree, in which at each node there is a separate GP over each subset of actions for each child. This structure is shown in fig. 14. Again each edge has its own separate GP, and actions at the leaf nodes share a single GP over all actions. Selection of the subset of actions over which to perform action no amounts to traversing

this binary decision tree, choosing the branch which has the highest $X^*_{\mathcal{A}_j}(b)$ as modeled by the respective GP. When updating the GPs with a state-action-reward example, all GPs on the path to the base cluster containing the respective action have their dictionaries updated accordingly.

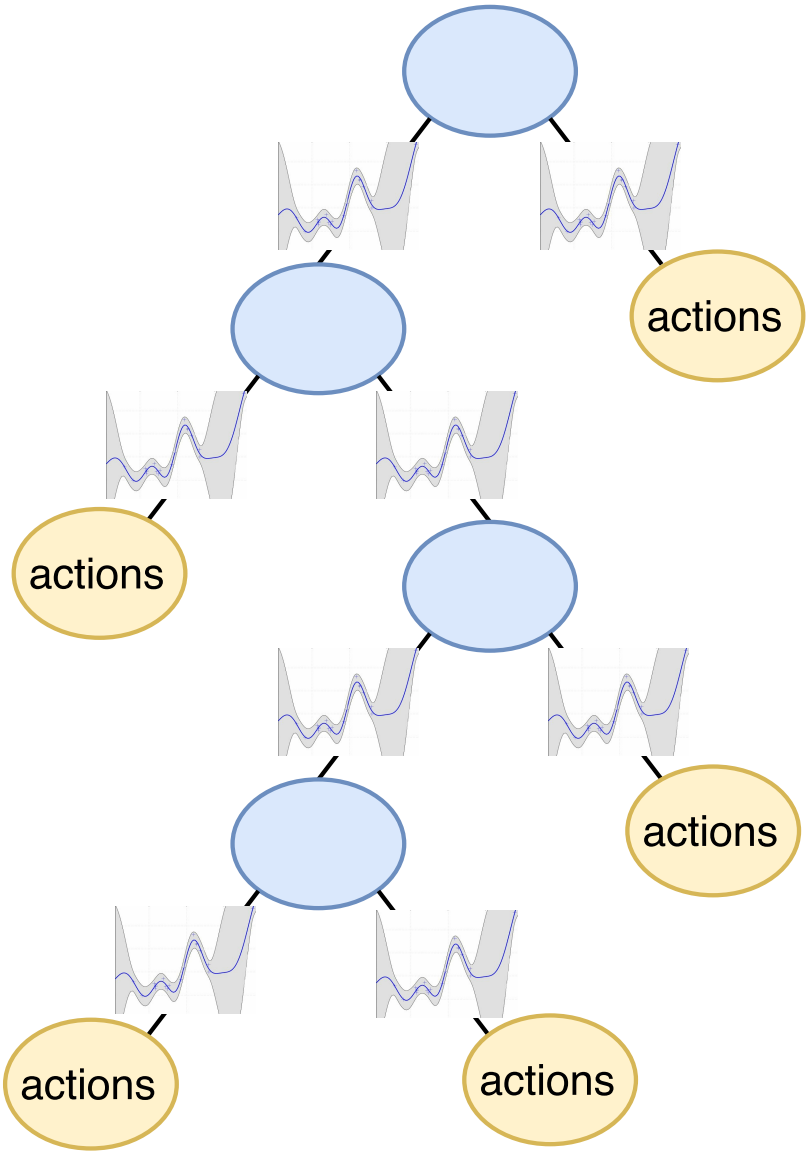


Figure 14: Topology of a hierarchical approach to action space modeling

Using this hierarchical clustering greatly reduces the computational complexity of the problem. In an ideal world the induced tree would be perfectly balanced, and each binary decision would split the action set into two equal halves, and the tree would have height $\log|\mathcal{A}|$. This would lead to computational complexity of $|\mathcal{D}|\log(|\mathcal{A}|) + \text{max\_size}$, where $max\_size = 100$ for experiments undertaken here. Under this method, the problem of action selection was rendered tractable, and full length experiments were possible.

Cold Start

Having made action selection tractable for interesting cases it was possible to perform full-scale tests. However initial results were extremely poor (see results section). Examination showed that at the beginning of a training run when the Q-value is zero over all actions and intermediate GPs in

the hierarchy, action selection devolves to random guessing resulting in a very sub-optimal decision being chosen. This causes the model to only learn what are the wrong actions to perform, and all GPs predict a negative Q-value for all states. Essentially the search space is too large at first for any progress to be made, resulting in a *cold start* problem.

Resultant dialog manager architecture

By performing policy optimisation directly in the master action space we arrive at a dialog manager architecture as shown in fig. 15.

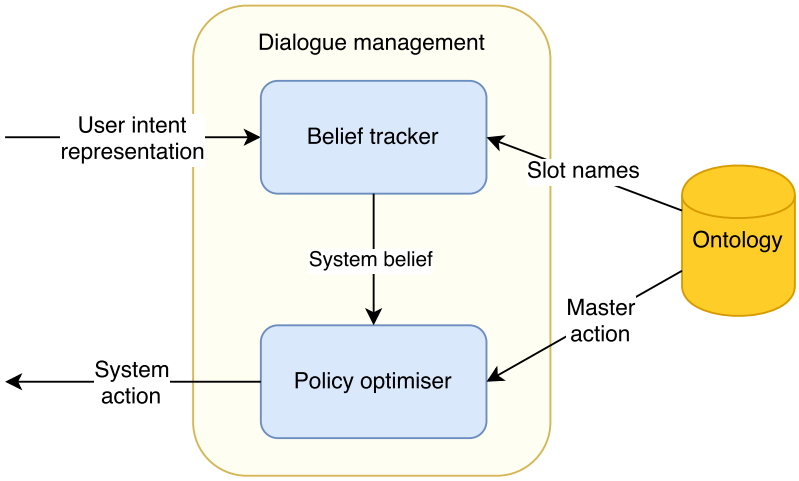


Figure 15: Dataflow within master aciton-based dialogue manager

Contrasting this to fig. 4 we see how this approach has greatly reduced the number of components and has eliminated hand crafted components.

Abstracted view

Taking a step back from the details of implementation, the approach taken in this dissertation can be seen to enable reinforcement learning to be performed on a belief space and action space given just two appropriate kernels. Our generalised, kernalised RL solution can be applied to any problem that can be specified as:

$$(\mathcal{B}, \mathcal{A}, K_{\mathcal{B}}, K_{\mathcal{A}}, R, \otimes)$$

where:

- $\mathcal{B}$ is the state space
- $\mathcal{A}$ is the actions space
- $K_{\mathcal{B}} : \mathcal{B} \times \mathcal{B} \to \mathbb{R}$ is the kernel on the state space.
- $K_{\mathcal{A}} : \mathcal{A} \times \mathcal{A} \to \mathbb{R}$ is the kernel on the action space.
- R is the reward function
- $\otimes$ is the method of combination of the two kernel values.

It can be seen that the this work allows us to perform GPSARSA-based reinforcement learning over any combination of state and action space, providing we have suitable kernels.

Experimental Setup

Experimentation was performed within the *cued-PyDial* system. The standard setup for experiments was: use of the `focus` belief tracker, the number of maximums turns to a dialogue was limited to 26, a squared-exponential kernel for the belief state, and a simulated user with a semantic error rate of 15%.

Only the [*confirm*, *inform*, *request*, *select*] intentions are predicted, initially *bye* was included, but the system learned to maximise the reward by predicting *bye* as the first action, thereby cutting the dialogue to a single step and resulting in a reward of -1.

Due to the necessity of operating directly in feature space actions that inform the user of alternatives are not employed for both the master and summary action spaces.

Results

Kernel values

	request(food)	select(food=“chinese”,food=“malaysian”)	inform(name=“curry quee”,postcode=“c.b 1, 2 b.d”)	inform(area=“centre”,food=“chinese”,name=“charlie chan”,pricerange=“cheap”)	inform(area=“north”,food=“chinese”,name=“the hotpot”,pricerange=“expensive”)	confirm(food=“chinese”)
request(food)	7.0	1.0	0.0	1.0	1.0	1.0
select(food=“chinese”,food=“malaysian”)	1.0	14.0	0.0	2.5	2.5	2.5
inform(name=“curry quee”,postcode=“c.b 1, 2 b.d”)	0.0	0.0	15.0	4.0	4.0	0.0
inform(area=“centre",food=“chinese",name=“charlie chan",pricerange=“cheap")	1.0	2.5	4.0	19.0	8.5	2.5
inform(area=“north",food=“chinese",name=“the hotpot",pricerange=“expensive")	1.0	2.5	4.0	8.5	19.0	2.5
confirm(food=“chinese")	1.0	2.5	0.0	2.5	2.5	11.5

In order to test if the kernel does define a reproducing kernel Hilbert space (RKHS), covariance matrices were constructed for random subsets of the actions for both the weighted and unweighted variants. The resultant matrices were positive semi-definite (PSD), and it is therefore assumed that the provided kernel is valid, and does define a RKHS. This is important as it means that resultant Gram matrices will be well-conditioned.

Feature space embedding

For the master action space used we have resulting feature space dimensions of 6157 and 11182 for the subtree and subset tree approaches respectively. The subset tree space is larger because the subset tree components form a superset of subtree components. Histograms showing the log the distribution of component frequencies are shown in fig. 16 and fig. 17. The subtree components are

more evenly distributed. This is because the subset tree decomposition contains a lot more singleton trees corresponding to nodes in the intention and slot name layer. Both distributions are incredibly skewed with a few components occurring a great many times, these correspond to the slot values in the subtree space, and individual slot names, slot values and intentions for the subset tree space. Unweighted tree kernels will ascribe more importance to more commonly occurring subcomponents, and for this domain unweighted kernels would place great importance on both the slot values and intentions. However weighting kernels allows us to shift importance away from components that merely have greater relative frequency.

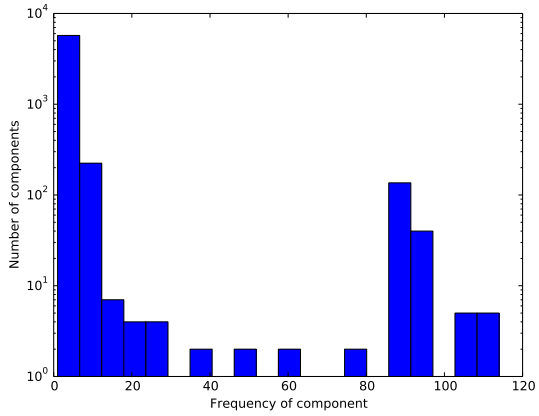


Figure 16: Log frequency distribution of subtree components

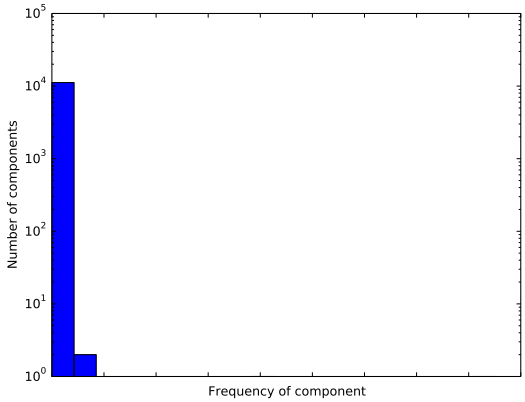


Figure 17: Log frequency distribution of subset tree components

In order to visualise the induced feature space, a representative subset of feature-space-embedded actions was chosen. SVD was used to reduce the sparse feature space to a dense representation of dimension 50. tSNE[28] was then used to reduce the dimension of the space to 2. Running tSNE directly on the full feature space caused erratic data representation, it seems in high, sparse dimensions tSNE is unable to locate useful distances between examples.

In fig. 18 we see the 2-dimensional representation of the subset-tree induced tree kernel. Actions can clearly be seen to be grouped by intention, which makes sense due to the weighing placed on sub-components that contain the intention node.

The two disparate *inform* regions consist of actions which, respectively, inform the user of an entity that meets their constraints (*e.g. inform(area="south", food="indian", name="raj", pricerange="expe*
and inform the user of a requested piece of information (e.g. *inform(name="raj", postcode="cb2 1rf")*). The proposed tree kernel can therefore be seen to be able to assign similarity based also on structure at the slot-name level. Other perceivable sub-regions were also found to be caused by similarity at the slot-name level. Slot-value appears to not have a significant effect on similarity, which is to be expected due to their low relative weighting.

Similar visualisations were performed using subset tree kernels with weights [1, 9, 1] and [1, 1, 9] putting greater relative weighting on middle and bottom layer tree structure respectively, and results are shown in fig. 19 and fig. 20. We see that even when there is relatively little weighting on the intention layer actions appear in broadly similar areas of reduced feature space. This is because although not explicitly matching on the name of the intention, similarity is implicitly assigned by matching the common sub-structures that exist between actions sharing an intention. For the mid-weighted kernel of fig. 19 we see that confirm and select actions are assigned to an almost indistinguishable region of reduced feature space, but are present in three clusters. These clusters correspond to the informable slots “cuisine”, “location” and “pricerange”. When slot names

are over-emphasised select and confirm actions become indiscernible. When the bottom layer of an action tree is epmhasised, as in fig. 18, we see that all actions of a different intention are embedded in broadly similar space. All actions share similar slot-values and the discernibility of actions on intention becomes impossible.

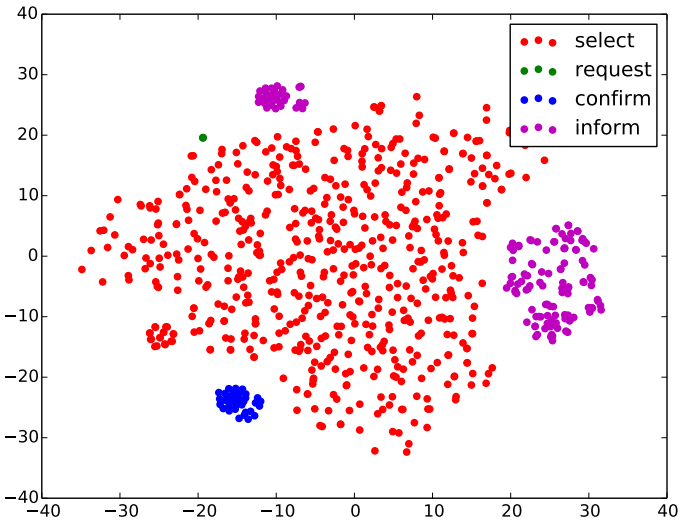


Figure 18: 2D visualisation of feature space induced by default subset tree kernel weightings. Actions are coloured by intention

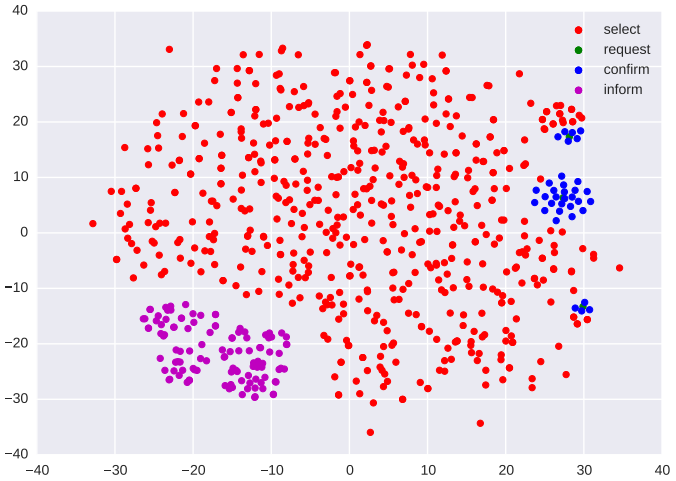


Figure 19: 2D visualisation of feature space induced by subset tree kernel with greater weight placed at slot-name level. Actions are coloured by intention

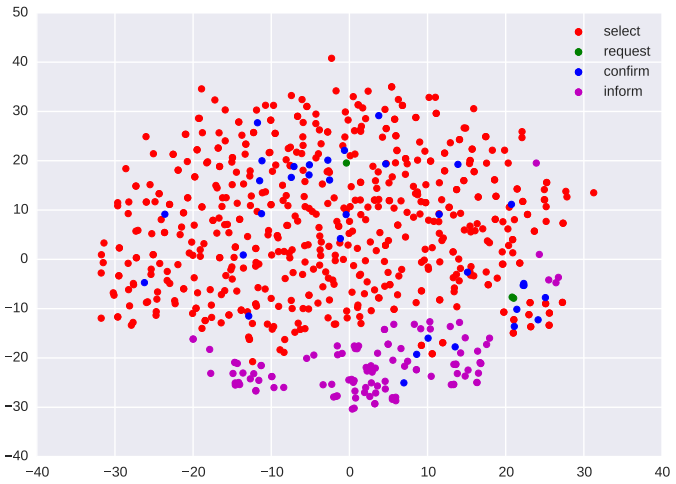


Figure 20: 2D visualisation of feature space induced by subset tree kernel with greater weight placed at slot-value level. Actions are coloured by intention

In order to visually understand the result of hierarchical clustering we visualise actions by cluster membership in fig. 21. Actions in similar reduced feature space are assigned to similar clusters, as expected. There is, however, some anomalous cluster membership, indicating that some actions are not assigned to the most representative clusters.

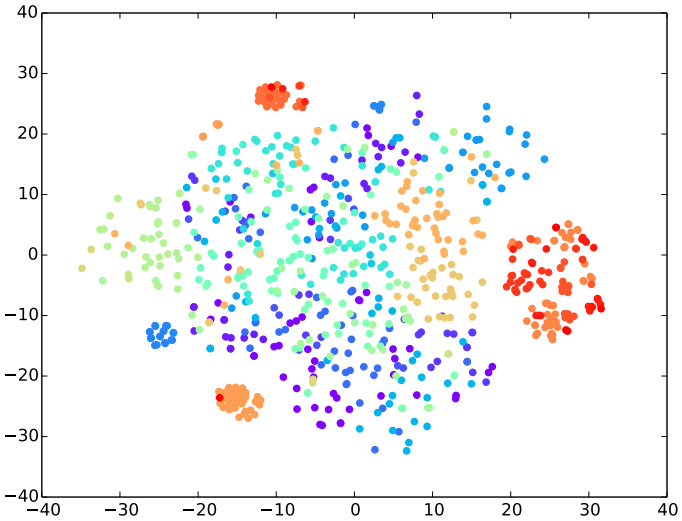


Figure 21: 2D visualisation of feature space induced by subset tree kernel with greater weight placed at slot-value level. Actions are coloured by cluster membership

Smoothness of Q-value

The assumption underlying the use of a certain kernel function is that small changes in input space will be reflected by small changes in the resultant value. In order to test verify that this assumption holds for the proposed action space kernel, the trained GP of the full subtree kernel run was used as a random belief state was held constant, and actions were *slightly changed* to see if the resultant

Q-values changed by a great amount. In this instance, "slightly changed" corresponds to changing a single node in the tree.

The results of this experiment can be seen in fig. 22. The action *confirm(pricerange="cheap")* was chosen as the base action, and corresponds to points consisting of a blue circle in fig. 22. All actions that involve changing the slot-value of the base action to each of [moderate, expensive, dontcare] were included as the "slightly changed" actions, with these points being represented as green triangles in the figure. The greatly changed actions are *confirm(area="east")*, *inform(name="kohinoor",postcode="c.b 1, 2 a.s")* (labeled in the figure as *inform_postcode* for brevity), and *inform(area="centre",food="italian",name="ask",pricerange="cheap")* (labelled as *inform_result*) in the figure, an are marked by red squares.

The results clearly show that the Q-value is smooth respect to small changes in actions. When we make a small change in the input space, as with the green triangles, we see only a very small change affected in the outputted Q-function. However when we make larger changes in action space, we see greatly different resultant Q-values. This shows that the necessary assumptions about continuity between the kernel and the output are met, and our tree-based approach therefore appears to be a suitable kernel for the target problem.

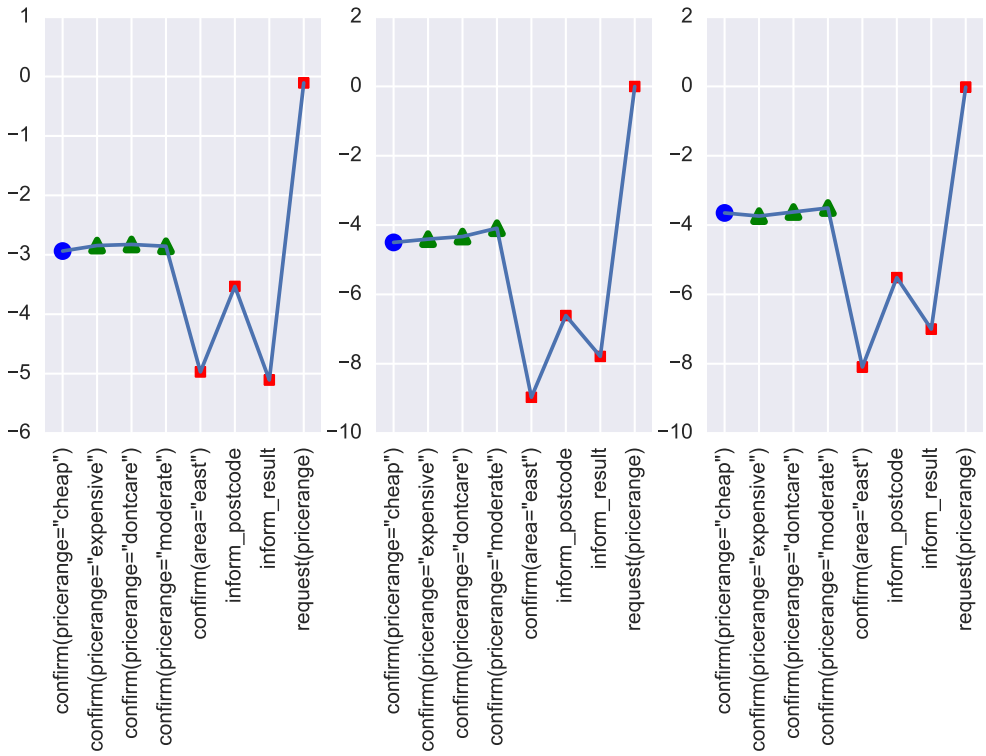


Figure 22: Evaluating continuity of Q-value with respect to action

Hierarchical representation

Clustering actions, using the method described above, results in clusters that have the following properties:

kernel type	number of clusters	mean cluster size	hierarchy depth
subtree	90	55.8	66
subset tree	95	52.9	163

We see broadly similar clustering results between the two kernel types. However the subset tree variety causes a hierarchy tree that is very tall, indicating that multiple internal decisions are needed. This is due to the finer granularity induced by the by the subset tree kernel causing greater similarities, and therefore less discriminability, between actions. Different random seed (which change the randomly selected points that are chosen as initial clusters) did not seem to affect the eventual clusters learned.

In order to check how able a model is to select the cluster that contains the best possible action, a fully trained subset tree kernel model was used to see the hit rate. In all 100 policy decisions the hierarchical model was not able to find the correct cluster, and the mean relative difference between the actual best action and the one selected under our hierarchical decomposition was -127%, showing a massive discrepancy between the actual overall best action and the action deemed to be best under our hierarchical model.

Dialogue success

A dialogue episode is considered a successful if the user is able to successfully accomplish their intended task. A typical measure of SDS performance is percentage of successful dialogues. Success results under the simulated user are shown in fig. 23 and fig. 24. The results show that the system is unable to improve over a very low baseline performance of a few percentage of successfully completed dialogues, a trend consistent between both kernel types.

The reward for the subtree kernel shows a downward trend as the number of dialogues increase. Looking at the average number of turns per dialogue we see that an increase in the length of dialogue episodes is the cause for the decrease in our reward. Looking at the actual sequence of dialogues produced it was noted that the shorter dialogues towards the beginning of training are caused by the system repeatedly selecting the same action and the episode being ended by the simulated user. However as training progresses different (albeit also incorrect) actions are selected. This is can be explained by realising that when the GP dictionary is "sparse" at the beginning of training, small changes in the belief state caused by an incorrect action will not affect a large change in the predicted Q-value, since there will be a more coarse-grained interaction between the few training points and the test point. As the size of the dictionary grows interactions become more fine-grained, and small changes in belief state can interact in greater ways with the larger set of test points in the dictionary.

The subset tree kernel, shows a slight improvement in dialogue success as the number of episodes increase. This is also reflected in the progression of average reward over time which shows a slight improvement. Not only is this due to the increase in average success, but we also see that it is caused by a decrease in average dialogue length. This shows that the system is capable of improving the correctness of system responses, and can achieve this in fewer turns.

The slight improvement in performance of the subset kernel can be understood by examining what extra components are contained in subset tree components, but not subcomponents. Examining again fig. 9 fig. 10 we see that subset tree components, also contain components corresponding towards the top. These elements are clearly important for action selection, but correlations due to them are ignored in the subtree kernel.

Looking at computational performance, we see results of how dictionary size changes over time, and how policy decision time changes as a function of dictionary size. Both subtree and subset tree kernels grow in size quicker than the summary action approach. There are a larger number of actions for which the eventual kernel value $K_{\mathcal{B}}(b, b') \times K_{\mathcal{A}}(a, a')$ can take a different value when using convolution kernels, whereas for the summary action we are limited by the size of the summary action set. Under the summary action set whole swathes of actions get bucketed together, whereas the

master action set treats them as distinct, and therefore different enough to include in the dictionary.

The subtree approach yields a slightly larger dictionary than the subset tree kernel. This is caused by the greater number of components in subset tree kernels causing more non-zero action kernel values, and therefore inducing more non-zero overall kernel values. The subtree kernel will instead have an action kernel value of zero for a greater number of actions, causing the distance between a prospective kernel entry and all entries in the dictionary to be below the sparsity threshold.

Computational performance can be seen to increase massively for both the subtree and subset tree kernel. This would be prohibitively expensive for realtime usage. Profiling showed that the main performance bottleneck for action selection are dot products involved in the calculation.

Figure 23: Dialogue successes, rewards and number of turns using subtree kernel. Error bars show one standard deviation. It can be seen that success may be slightly downward trending and the number of turns increase over time.

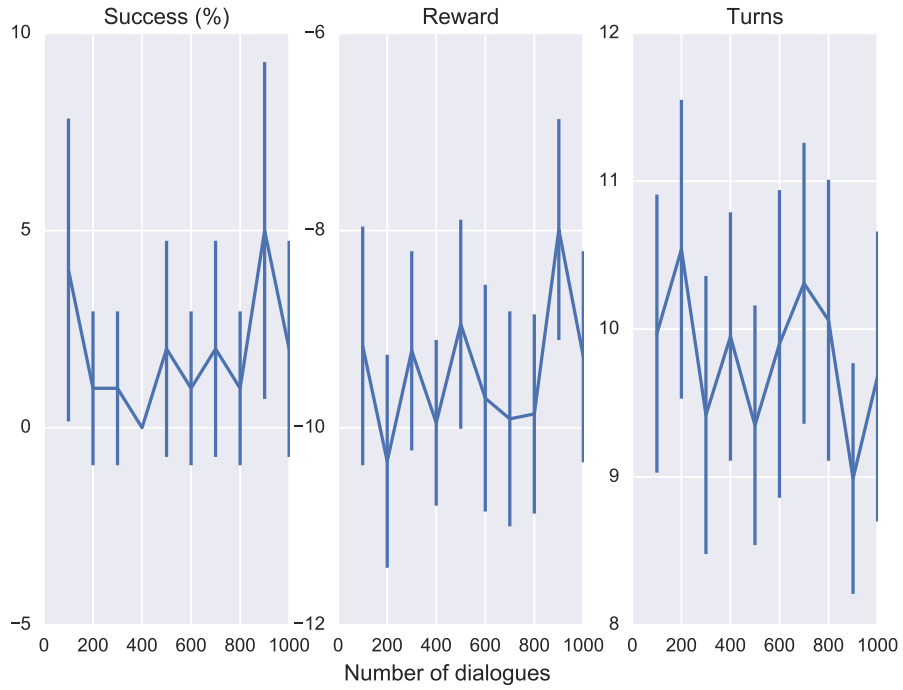


Figure 24: Dialogue successes, rewards and number of turns using subset tree kernel. Error bars show one standard deviation. The subset kernel performs slightly better, with a slight upward trend in success. This is due to the ability to model finer structure at the top of the tree

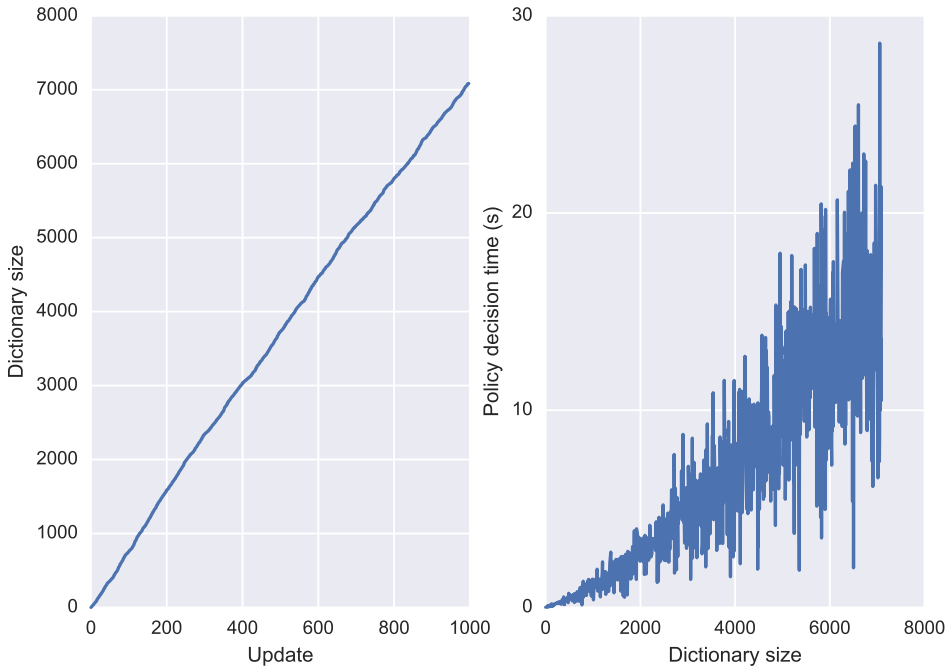


Figure 25: Subtree kernel dictionary size growth over time, and action selection time against dictionary size. Performance degrades to levels unacceptable for real-time systems.

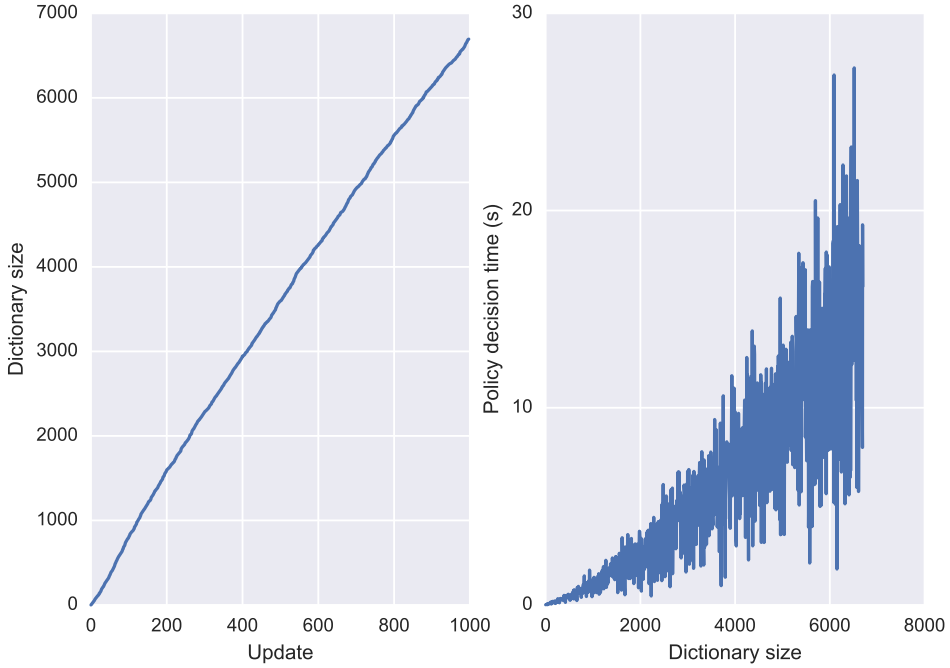


Figure 26: Subset tree kernel dictionary size growth over time, and action selection time against dictionary size. Performance degrades to levels unacceptable for real-time systems

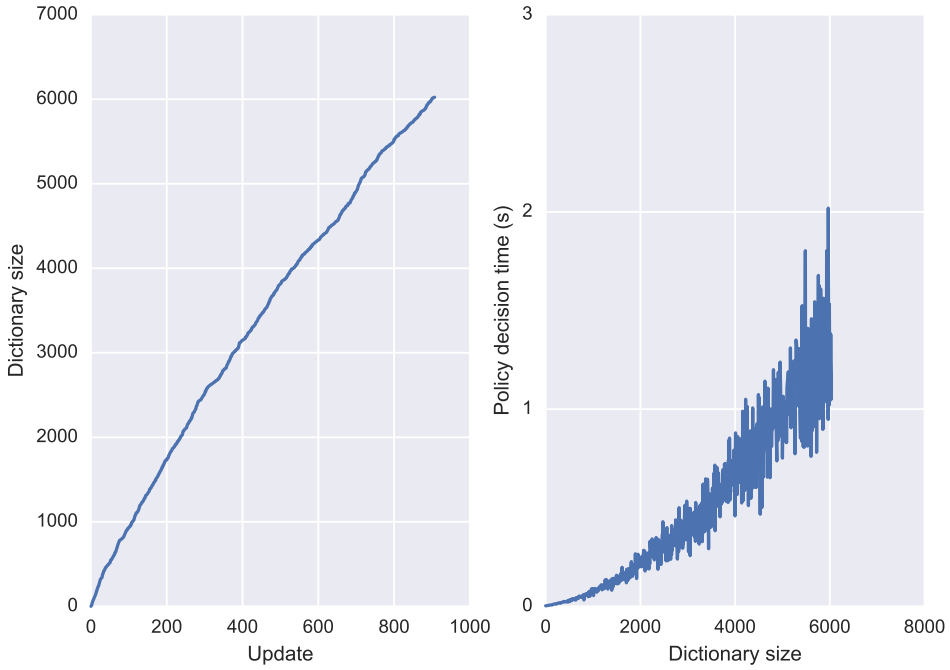


Figure 27: Summary action approach dictionary size growth over time, and action selection time against dictionary size

Condintionedness

For some predicted values in our model, the corresponding variance was negative. Investigations were unable to pinpoint the cause, and it is assumed that the interplay between kernel parameters (both for the belief and action kernels) and the sparsification factor causes our GP to be ill-conditioned. The belief kernel contains sorted sections that model only distribution shape, and not the distribution at slot values. This could cause points that have very different belief states to have identical belief kernel values, leading to ill-conditionedness. This is a very interesting lead into what is going wrong in the implementation of the proposed idea.

Future work

Active learning

The exploration of the action space only takes place due to random actions taken in the ϵ-greedy search and the variance in the Q-value estimate. A more rigorous form of estimation would be to perform active learning, where the algorithm itself decides upon the exploratory action to take dependent on some criterion. A simple possibility would be to simply choose the action whose Q-value has the highest variance, proceeding in this way would decrease general uncertainty in the model over time. Another possibility is looking at the *spread* of a Q-value estimate: if a particular action has an average mean, but a high value it is possible that the large spread of this action could result in an expected reward that is greater than an action with a higher mean but lower variance. This notion of trading off the amount information provided by an action and the expected reward is presented in [22] as the "value of information" characteristic of a particular action. Such active learning approaches, which maximise the utility of exploratory steps would be particularly advantageous in large action spaces where initial (i.e. exploratory) steps are so important to forming a useful view of the problem space. As we saw in our results, and to which seeding with the summary action function was shown not to offer an improvement, quickly building up a set of useful representative points is a problem yet to be overcome.

The current approach requires explicit feature space representation of actions when performing clustering. The decision of cluster membership requires calculation of the distance between feature space representation of our actions. This can in fact be performed purely in the induced reproducing kernel Hilbert space. By normalising our kernel values so that they lie in the range [0, 1]:

$$\bar{K}(x, x') = \frac{K(x, x')}{\sqrt{K(x, x)K(x', x')}}$$

And then defining our distance as:

$$D(x, x') = 1 - K(x, x')$$

We still have the problem of calculating a mean value for a particular cluster in order to ascertain cluster membership. This can be mitigated in a number of ways. Firstly, instead of calculating an explicit mean, the most *representative* point in a cluster can be chosen as the mean. This can be selecting the point that has the minimum average pair-wise distance with all other actions in a cluster. In order to calculate cluster membership a prospective point has its distance measured against the representative point. Another possibility is to assign points to clusters that have minimum average or maximum pair-wise distance. In this way there is no need to calculate a mean value in features space, and all interactions between actions take place implicitly via the kernel function.

The changes to clustering suggested above negates the problem of explicitly representing an action in feature space, but does nothing to remove the need to represent all actions explicitly, which may become infeasible for larger domains. Action selection can therefore not be performed by iterating over all actions (although we actually only evaluate the Q-function explicitly for a fixed-sized subset of actions, we implicitly consider all actions due to the binary decisions made at internal nodes in a decision hierarchy), and other methods must be explored.

One possibility is to explicitly predict an expected representation of the optimal action and then search for a best matching tree. The prediction of the action can be easily adapted from other approaches that employ the actor-critic method, in which we learn a model to predict an action, and a scoring model that critiques these actions. However the search for a tree that matches our representation still presents a major computational challenge. A generate-and-test schema could be possible in which generations are create by making small incremental changes to a best-guess tree,

allowing exploration of our tree-space within a certain neighbourhood. It may also be possible to learn how to generate trees from a certain representation. Work on *recursive neural networks*[47] allows conversion of a tree to a fixed-length vector representation and minimises the trees reconstruction error. By training recursive neural networks on our action trees to embed and unembed a tree in a fixed dimensional space, and then constructing a discriminative model that learns to predict an action's fixed length representation we can use the recursive neural network to unembed from the predicted tree representation.

Another possibility is to perform piece-wise construction of an action tree, where a series of decisions constructs our action tree layer-by-layer. This approach is illustrated in fig. 28. We see three consecutive decisions that build up a tree: first choose the intention, then choose the slot names, and then choose the slot values. In this way a master action is not explicitly constructed until all layer-wise decisions have been made removing all other intermediate master actions.

The current approach can be extended to be usable in this stepwise decision method. At each decision step we maintain a GP over all decisions that can be made, and choose the decision that has the highest score under the respective GP. Deconstructing an action space into a sequence of construction decisions should be possible from the ontology, however any extensions to the ontology will require the constructive rules and model to be redefined.

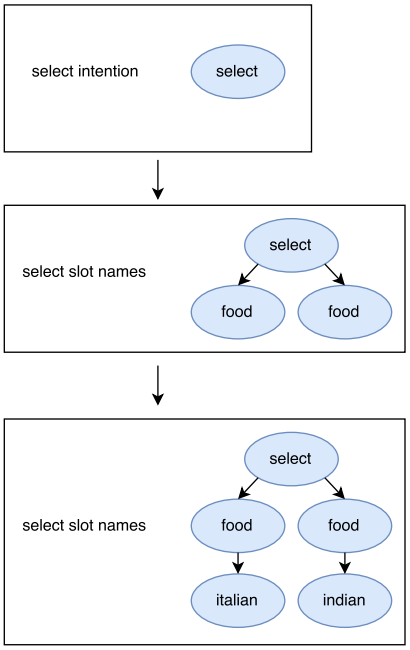


Figure 28: Layer-wise construction of a master action

Improvements to action space decomposition

The clustering approach taken in this work is a little unusual, and was shown to perform very poorly. Although it was shown to produce sensible clusters that obey distances, and a kernalised version is possible, it is not a standard clustering approach and was shown to induce clusters from which it was hard to select the optimal action. More standard hierarchical clustering approaches[37] may be preferable, and similar extensions to such approaches can be made so that they do not need to operate directly in feature space.

Improvements to correctly predicting cluster membership could come by allowing an action to be present in multiple clusters. This could be possible by both fuzzy-class membership, in which a class

can be assigned to multiple classes, with their membership-ness based on the distance to the cluster. It could also be possible to add actions to action subset clusters during prediction time as it becomes apparent that other actions are potentially more optimal. Such a system could be implemented as a quick MC search over a subset of master action space. If actions with higher Q values are found, then they can be added to the selected action subset. And if actions in a subset are found to repeatedly yield Q-values vastly dissimilar to other members of the cluster, such actions can be reassigned.

Although, as shown in the examination of continuity above, small changes in our actions don't affect large changes in the resultant Q-function, it may be more useful to measure action similarity by eventual outputted Q-value. Therefore clustering by kernel value alone a priori may be sub-optimal, and a system in which the resultant Q-value predictions are taken into account seem necessary. This results in a chicken-and-egg situation in which it is necessary to perform policy optimisation before being able to cluster, while also needing to cluster before policy optimisation.

Aside from hierarchical decomposition of the actions space, multi-classification actor-critic methods, as outlined above, could discriminatively predict the subset actions to score.

Hyperparameter tuning

In this work we have used hand-tuned/ default values for the kernel hyperparameters, the length scale and output variance of the belief kernel, and the weightings for each layer of the tree kernel. It is unlikely that these are optimal, and so GP hyperparameter training[41] would offer an improvement. This has been specifically applied to GPSARSA for dialogue management[5] and showed a decent improvement over baselines.

In order to optimise tree kernel hyperparameters for GPSARSA a training corpus of belief-action inputs to reward outputs are necessary. The negative log marginal likelihood of the training data is then minimised with respect to the hyperparameters via the conjugate gradient method[43]. The optimisation of SE kernel hyperparameters in this way is a well understood problem, and the derivative of the marginal log likelihood wit respect to the tree kernel weightings is also simple to derive. However the hierarchical structure of the proposed solution causes introduces some problems. The topology of the hierarchy and action membership are all dependent on the tree kernel weightings, and is very much not differentiable. It would therefore be necessary to optimise hyper-parameters given a certain clustering, and then update the clustering as hyper-parameter changes. This would add greatly to computation time, and does not take into account cluster membership in the optimsiation process. An approach that could optimise the resultant GP values and cluster membership would be to perform Bayesian optimisation [46] over the hyper parameters, with the resultant negative log marginal likelihood of the data (under the clustering and GPs induced by a particular hyperparameter setting) being the value to maximise.

Tuning of both the kernel hyperparamters and sparsity hyperparameters could resolve the issue of negative variance for our GP samples. Manual tuning would be a good first step in exploring whether this is the case, and more rigorous optimisation could lead to improvements in practical performance.

Changes to kernels

Experimental results have shown that subset tree components are marginally superior to subtree components. As discussed this is likely due to their ability to capture structure near the top of the tree. Subtrees could be altered so that rather than having to contain *leaf* they have to contain the *root*. This would allow the kernel to model the intention and slot names that are important to action selection.

While the tree kernels proposed here form a RKHS, it is unclear whether they are suitable for the dialogue problem. There seems to be limited scope for the *transfer of knowledge* between loosely related training points. Talking about a North Indian restaurant should impart some information about a South Indian restaurant, however the discrete nature of tree nodes means there must be an exact match.

Embedding the nodes of trees in a fixed dimensional distributed representation would induce smoother distance related to other entities. A kernel could then be constructed that took into account the distance of tree nodes, rather than just equality. The formation of such a representation is difficult. Co-occurrence statistics of slot names in the ontology and in real world training data could produce noisy estimates due to the small amounts of data involved.

The current multiplicative method of combination of kernel values for our state-action pairs could be extended. This binary operator could become a parameterised function where lesser or greater emphasis is placed on one kernel based on location in the belief and actions space. While increasing expressibility, this approach decreases training ability, and more datapoints will need to be seen to produce a model with strong generalisability qualities.

Extensions to other domains

The proposed approach need not be limited only to the chosen domain. The GPSARSA dual-kernel approach investigated here can generalise to any belief and action space over which kernels can be applied.

The tree kernels allow reinforcement learning to be performed in combinatorial domains. When applied over actions, the approach here allows actions to be performed in any combinatorial space. Such a space could consist of decision trees that define action decisions. Performing MDP in this space would then allow for predicting complex structured actions at each time step, rather than a single explicit action. A tree kernel over a belief state, on the other hand, could allow for optimisation of decisions for automated theorem proving where the belief state is a proof tree and actions are rewrite rules, with success being defined if the proof is verified. Other convolution kernels, such as string kernels, could enable a natural language action space.

Tree kernels can be extended to infinite discrete dimensions by allowing trees of unbounded depth. Due to the exponential space complexity of subcomponent calculation we can not allow subcomponents to be calculated on the full unbounded tree. Instead subcomponents can be of a fixed depth so that a *template* structure is moved around nodes of a tree. Exact action selection in an infinite space would present a large obstacle, and only sub-optimal may be calculable in a reasonable amount of time.
Examination of which domains work for the proposed solution would be an extremely interesting line of research.

Performance improvements

Profiling shows that system to be prohibitively slow. This could be improved by making cluster sizes smaller so that fewer actions need to be scored, however this would increase the problem of not being able to locate the cluster that contains the highest scoring action. Profiling showed that dot-products dominate action selection. Performing these calculations on the GP, or using vectorised CPU instructions, could lead to improvements in throughput and latency. If massive deployment to a centralised cluster, or to embedded devices was desired, it would be necessary to create a native implementation of the system that would improve portability and performance.

Conclusion

We have seen how perform reinforcement learning in any state space and action space simply by defining appropriate kernels and performing GPSARSA. The approach is automatically extensible to the large action spaces by using a kernalized clustering algorithm which can operate implicitly in the feature space through the kernel trick. This is a very general approach, and instigating its efficacy in domains previously off-limits would push the frontier of reinforcement learning.

By viewing SDS actions as trees, we are able to utilise tree kernels to have been able to extend the notion of fine-grained notions of similarities to discrete actions. This enables us to use a single GP over all actions, replacing multiple separate GPs for each discrete action. Recursively clustering the action space allows us to make sequential decisions that divide up the state space allow action selection to become tractable, but the trainability of these structures is unknown.
The proposed approach was fully integrated into the *cued-pydial* system to produce a fully working prototype. Overall testing results proved negative, although some learning seems to be taking place. Closer inspection by experimentation showed that the kernel seemed appropriate, and the clustering approaches produced sensible divisions of actions. However the hierarchically-clustered decision trees were unable to locate the optimal action, which was probably the cause of the poor performance.

Reinforcement learning in large state and action spaces is a challenging and ongoing are of research. Future work should explore data-efficient, computationally fast ways of predicting a subset of actions in policy decision. Division of state space represents only one such method, and future work could investigate the actor-critic method, and discriminative prediction of an action representation,

References

[1] Leemon C Baird and A Harry Klopf. "Reinforcement learning with high-dimensional, continuous actions". In: *Wright Laboratory, Wright-Patterson Air Force Base, Tech. Rep. WL-TR-93-1147* (1993).

[2] Andrew G Barto and Sridhar Mahadevan. "Recent advances in hierarchical reinforcement learning". In: *Discrete Event Dynamic Systems* 13.4 (2003), pp. 341–379.

[3] Andrew G Barto, Richard S Sutton, and Charles W Anderson. "Neuronlike adaptive elements that can solve difficult learning control problems". In: *IEEE transactions on systems, man, and cybernetics* 5 (1983), pp. 834–846.

[4] Charles Blundell et al. "Weight uncertainty in neural networks". In: *arXiv preprint arXiv:1505.0542* (2015).

[5] Lu Chen, Pei-Hao Su, and Milica Gašic. "Hyper-parameter Optimisation of Gaussian Process Reinforcement Learning for Statistical Dialogue Management". In: *16th Annual Meeting of the Special Interest Group on Discourse and Dialogue*. 2015, p. 407.

[6] Michael Collins and Nigel Duffy. "Convolution kernels for natural language". In: *Advances in neural information processing systems*. 2001, pp. 625–632.

[7] Corinna Cortes, Patrick Haffner, and Mehryar Mohri. "Rational kernels: Theory and algorithms". In: *Journal of Machine Learning Research* 5.Aug (2004), pp. 1035–1062.

[8] Daniela Pucci De Farias and Benjamin Van Roy. "On constraint sampling in the linear programming approach to approximate dynamic programming". In: *Mathematics of operations research* 29.3 (2004), pp. 462–478.

[9] Thomas Dean and Shieu-Hong Lin. "Decomposition techniques for planning in stochastic domains". In: *IJCAI*. Vol. 2. Citeseer. 1995, p. 3.

[10] Thomas G Dietterich. "Hierarchical reinforcement learning with the MAXQ value function decomposition". In: *J. Artif. Intell. Res.(JAIR)* 13 (2000), pp. 227–303.

[11] Thomas G. Dietterich and Ghulum Bakiri. "Solving multiclass learning problems via error-correcting output codes". In: *Journal of artificial intelligence research* 2 (1995), pp. 263–286.

[12] Gabriel Dulac-Arnold et al. "Deep Reinforcement Learning in Large Discrete Action Spaces". In: ().

[13] Gabriel Dulac-Arnold et al. "Fast reinforcement learning with large action sets using error-correcting output codes for MDP factorization". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer. 2012, pp. 180–194.

[14] Yaakov Engel, Shie Mannor, and Ron Meir. "Reinforcement learning with Gaussian processes". In: *Proceedings of the 22nd international conference on Machine learning*. ACM. 2005, pp. 201–208.

[15] Milica Gašić and Steve Young. "Gaussian processes for POMDP-based dialogue manager optimization". In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 22.1 (2014), pp. 28–40.

[16] M Gašić et al. "Policy optimisation of POMDP-based dialogue systems without state space compression". In: *Spoken Language Technology Workshop (SLT), 2012 IEEE*. IEEE. 2012, pp. 31–36.

[17] David Haussler. *Convolution kernels on discrete structures*. Tech. rep. Citeseer, 1999.

[18] Ji He et al. "Deep Reinforcement Learning with an Unbounded Action Space". In: *CoRR* abs/1511.04636 (2015). URL: http://arxiv.org/abs/1511.04636.

[19] Matthew Henderson, Blaise Thomson, and Jason D Williams. "The third dialog state tracking challenge". In: *Spoken Language Technology Workshop (SLT), 2014 IEEE*. IEEE. 2014, pp. 324–329.

[20] Matthew Henderson, Blaise Thomson, and Steve Young. "Deep neural network approach for the dialog state tracking challenge". In: *Proceedings of the SIGDIAL 2013 Conference*. 2013, pp. 467–471.

[21] Anil K Jain. "Data clustering: 50 years beyond K-means". In: *Pattern recognition letters* 31.8 (2010), pp. 651–666.

[22] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. "Planning and acting in partially observable stochastic domains". In: *Artificial intelligence* 101.1 (1998), pp. 99–134.

[23] Dongho Kim. personal communication. July 27, 2016.

[24] Michail G Lagoudakis and Ronald Parr. "Reinforcement learning as classification: Leveraging modern classifiers". In: *ICML*. Vol. 3. 2003, pp. 424–431.

[25] Alessandro Lazaric, Marcello Restelli, and Andrea Bonarini. "Reinforcement learning in continuous action spaces through sequential monte carlo methods". In: *Advances in neural information processing systems*. 2007, pp. 833–840.

[26] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[27] Huma Lodhi et al. "Text classification using string kernels". In: *Journal of Machine Learning Research* 2.Feb (2002), pp. 419–444.

[28] Laurens van der Maaten and Geoffrey Hinton. "Visualizing data using t-SNE". In: *Journal of Machine Learning Research* 9.Nov (2008), pp. 2579–2605.

[29] Charles A Micchelli, Yuesheng Xu, and Haizhang Zhang. "Universal kernels". In: *Journal of Machine Learning Research* 7.Dec (2006), pp. 2651–2667.

[30] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[31] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[32] Alessandro Moschitti. "Making Tree Kernels Practical for Natural Language Learning." In: *EACL*. Vol. 113. 120. 2006, p. 24.

[33] Alessandro Moschitti. "Making Tree Kernels Practical for Natural Language Learning." In: *EACL*. Vol. 113. 120. 2006, p. 24.

[34] Alessandro Moschitti and Roberto Basili. "A tree kernel approach to question and answer classification in question answering systems". In: *Proceedings of the 5th international conference on Language Resources and Evaluation*. Citeseer. 2006, pp. 1–510.

[35] Alessandro Moschitti, Daniele Pighin, and Roberto Basili. "Tree kernels for semantic role labeling". In: *Computational Linguistics* 34.2 (2008), pp. 193–224.

[36] Marius Muja and David G Lowe. "Scalable nearest neighbor algorithms for high dimensional data". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 36.11 (2014), pp. 2227–2240.

[37] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[38] Jason Pazis and Michail G Lagoudakis. "Reinforcement learning in multidimensional continuous action spaces". In: *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*. IEEE. 2011, pp. 97–104.

[39] Jason Pazis and Ron Parr. "Generalized value functions for large action sets". In: *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 2011, pp. 1185–1192.

[40] Joelle Pineau, Geoff Gordon, Sebastian Thrun, et al. "Point-based value iteration: An anytime algorithm for POMDPs". In: *IJCAI*. Vol. 3. 2003, pp. 1025–1032.

[41] Carl Edward Rasmussen. "Gaussian processes for machine learning". In: (2006).

[42] Carl Edward Rasmussen and Zoubin Ghahramani. "Occam's razor". In: *Advances in neural information processing systems* (2001), pp. 294–300.

[43] Carl Edward Rasmussen and Hannes Nickisch. "Gaussian processes for machine learning (GPML) toolbox". In: *Journal of Machine Learning Research* 11.Nov (2010), pp. 3011–3015.

[44] Bobak Shahriari et al. "Taking the human out of the loop: A review of bayesian optimization". In: *Proceedings of the IEEE* 104.1 (2016), pp. 148–175.

[45] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529.7587 (2016), pp. 484–489.

[46] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. "Practical bayesian optimization of machine learning algorithms". In: *Advances in neural information processing systems*. 2012, pp. 2951–2959.

[47] Richard Socher et al. "Parsing natural scenes and natural language with recursive neural networks". In: *Proceedings of the 28th international conference on machine learning (ICML-11)*. 2011, pp. 129–136.

[48] Pei-Hao Su et al. "Continuously Learning Neural Dialogue Management". In: *arXiv preprint arXiv:1606.02689* (2016).

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

[50] Hado Van Hasselt and Marco A Wiering. "Reinforcement learning in continuous action spaces". In: *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*. IEEE. 2007, pp. 272–279.

[51] Hado Van Hasselt and Marco A Wiering. "Using continuous action spaces to solve discrete problems". In: *2009 International Joint Conference on Neural Networks*. IEEE. 2009, pp. 1149–1156.

[52] Marco A Wiering and Hado Van Hasselt. "Two novel on-policy reinforcement learning algorithms based on TD (λ)-methods". In: *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*. IEEE. 2007, pp. 280–287.

[53] Jason D Williams. "Challenges and opportunities for state tracking in statistical spoken dialog systems: Results from two public deployments". In: *IEEE Journal of Selected Topics in Signal Processing* 6.8 (2012), pp. 959–970.

[54] Jason D Williams and Steve Young. "Scaling POMDPs for spoken dialog management". In: *IEEE Transactions on Audio, Speech, and Language Processing* 15.7 (2007), pp. 2116–2129.

[55] Min Zhang et al. "A composite kernel to extract relations between entities with both flat and structured features". In: *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics*. Association for Computational Linguistics. 2006, pp. 825–832.

Appendices

Appendix A - CUED dialogue act specifications In the Cambridge University Engineering Department Python Dialogue System (cued-pydial), the complete set of possible actions is uniquely defined by the ontology. All actions take the form:

<intention>(<intention-args>)

Intentions define the overall meaning of an action. Some examples of some SDS intentions, and the respective real-world actions they would be mapped to are shown below:

inform(...) → *say:* "The restaurant you're looking for is..."

select(...) → *say:* "Did you mean ... or ... ″

request(...) → *say:* "Can you tell me which ... you're looking for"

confirm(...) → *say :*"Did you mean ...?"

book(...) → *Perform an online booking*

The arguments for actions typically take the form of slot value-name pairs, of the form ⟨*slot-value*⟩ = ⟨*slot-name*⟩. Some concrete examples are: *food*="indian", *area* = "center". Slot names correspond to columns in the database of domain entities, and slot values the entry of a column for a particular entity. The permissible slot names for a particular intention are defined in the ontology. For example the ontology may define that ["sex", "height", "age"] are *requstable* slots, mean that user can request information them. This allows the system to construct actions that inform the user for information using these slot values, some examples would be:

inform(name="tom", phonenumber="555-1234")

Slots may be *informable*, meaning the user can provide information about the value of the slot, and therefore that the system can request information about them. Some examples of actions constructed using informable slots are:

request(height)

select(age="32", age="18")

Appendix B - profiling results

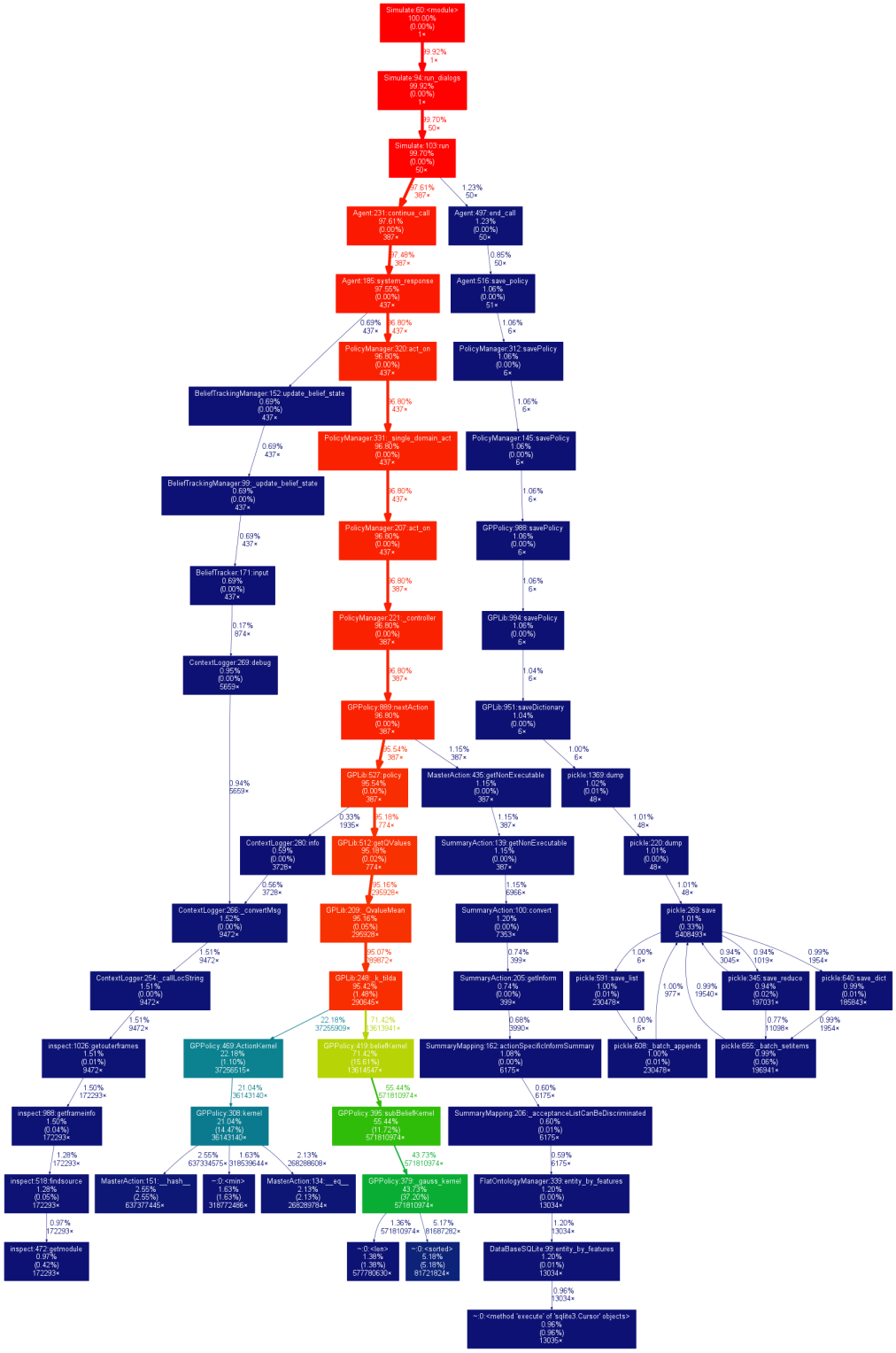


Figure 29: Profiling results of the system before massive performance work was undertaken. The red path through the tree shows the dominance of action selection in runtime

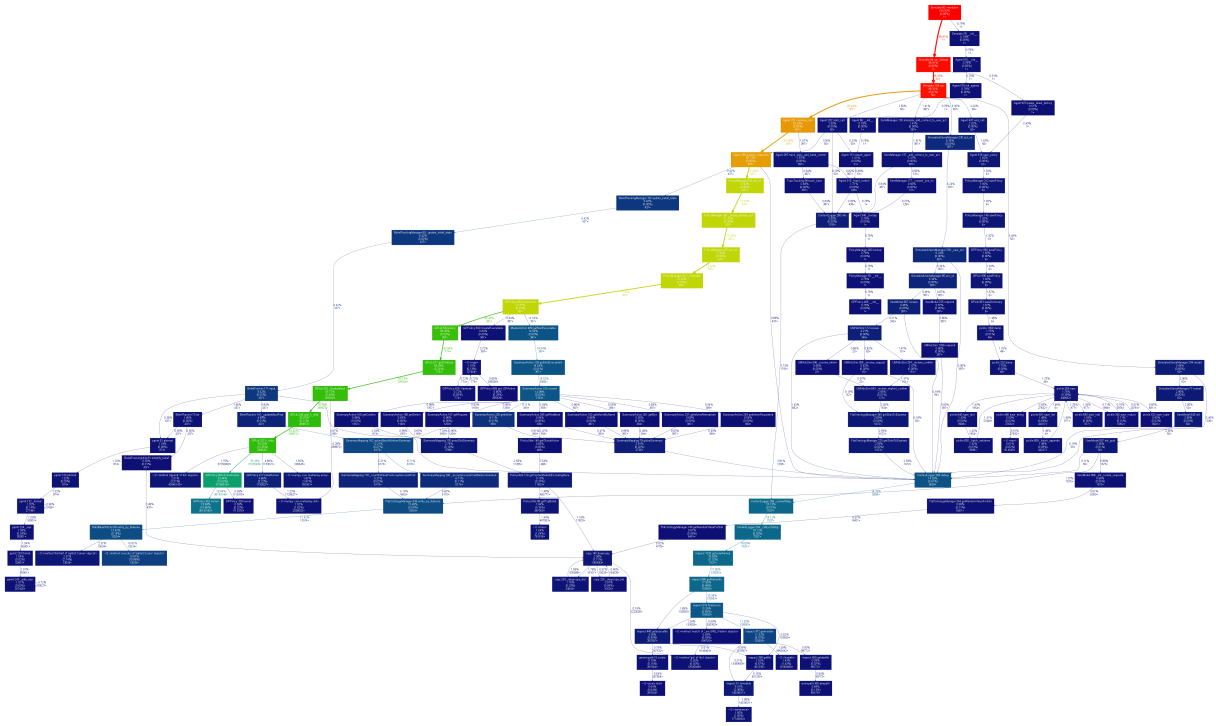


Figure 30: Profiling results of the system after performance improvements. The colours along all paths are more homogeneous showing that no single compute path is dominating, although the branch corresponding to action selection still stands out