UNIVERSITY OF CAMBRIDGE

MPhil in Machine Learning, Speech and Language Technology
2015-2016

MODULE COURSEWORK FEEDBACK

Student Name: Mac Chamberlin

Module Title: Statistical Spoken Dialogue Systems

CRSiD: MEC68

Module Code: MLSALT9

College: CLARE

Coursework Number: 1

I confirm that this piece of work is my own unaided effort and adheres to the Department of Engineering's guidelines on plagiarism

Date Marked:

Marker's Name(s):

Marker's Comments:

This piece of work has been completed to the following standard *(Please circle as appropriate)*:

	Distinction			Pass			Fail (C+ - marginal fail)		
Overall assessment (circle grade)	**Outstandi** ☐	**A+** ☐	**A** ☐	**A-** ☐	**B+** ☐	**B** ☐	**C+** ☐	**C** ☐	**Unsatisfactor** ☐
Guideline mark (%)	90-100	80-89	75-79	70-74	65-69	60-64	**55-59**	**50-54**	**0-49**
Penalties	**10% of mark for each day late (Sunday excluded)**								

The assignment grades are given **for information only**; results are provisional and are subject to confirmation at the Final Examiners Meeting and by the Department of Engineering Degree Committee.

Introduction: The General Framework

A Dialogue State Tracker (DST) is a system that attempts to estimate what a user has requested throughout a dialogue. It accumulates evidence over a sequence of user utterances, and adjusts the dialogue state according to these observations.

The dialogue state can be thought of as representing a 'belief' about what the user currently wants. It has the the following 3 components: (1) a probability distribution over goals; (2) a probability distribution over the requested slots in the ontology; (3) and assigns a probability distribution over methods of searching for the entities, which might be by constraints, alternatives etc. In this practical, we will explore the use of such states in dialogue systems.

Q1-Belief and Focus Tracker

A belief tracker will use scores taken from a spoken language understanding engine, **SLU,** to track the dialogue state.

A SLU returns a score, $slu_{t,c}(v)$, for each turn t, each belief-state component c over (goal, method, requested) and each value v. E.g., we might see at turn 3:

- $slu_{3,method}$(by constraints)=0.3 ; $slu_{3,\text{goal area}}(east) = 0.6$; $slu_{3,\text{goal area}}(west) = 0.1$.

The way the baseline tracker tracks the dialogue state is, for a given slot, to return the value with the highest score found so far. If the above case, the baseline tracker would simply return east for the slot goal area at $t = 3$, assuming there weren't previously any goal area values with a higher score.

In contrast, the focus tracker accumulates evidence in the quantity $p_{c,t}(v)$ rather than returning the values for a given slot with highest SLU scores. The way the evidence $p_{c,t}(v)$ is updated throughout the dialogue, is as below:

$$p_{c,t}(v) = slu_{t,c}(v) + q_{c,t}p_{c,t-1}(v) \tag{1}$$

where $q_{c,t}$ is a normalising constant, updated according to:

$$q_{c,t} = 1 - \sum_{v} slu_{t,c}(v) \tag{2}$$

Implementation:

Equations (1) and (2) form the basic framework for the focus tracker. Below I present the implementations of these formulae that were requested in the practical.

Goals:

Listing 1: Updating Goals

```
q = clip(1 - sum(this_u[slot].values())
  for value in this_u[slot].keys() + hyps["goal-labels"][slot].keys():
      if value in hyps["goal-labels"][slot]:
          hyps["goal-labels"][slot][value] = this_u[slot][value] + q*hyps["goal-
            labels"][slot][value]
      else:
```

```
            hyps["goal-labels"][slot][value] = this_u[slot][value]
            # normalise the score of each value in a slot
            hyps["goal-labels"][slot] = normalise_dict(hyps["goal-labels"][slot])
```

The implementation almost exactly follows (1) and (2). We note that it is important to clip q in the range [0,1] and normalise the scores given by p after updating so that the scores can still be thought of as probabilities.

Methods:

Listing 2: Updating Methods

```
q = clip(1.0-sum([method_stats[key] for key in method_stats.keys() if key !='none']))
for method in method_stats.keys() + method_label.keys():
    if method in method_label:
        method_label[method] = method_stats[method] + q * method_label[method]
    else:
        method_label[method] = method_stats[method]
method_label["none"] = 0
method_label["none"] = 1 - sum(method_label.values())
```

The 'method' of a user's turn describes the way the user is trying to interact with the system. When updating probabilities, special care must be taken due to the 'none' turn, which essentially means there has been no change in the method requested. Because of the 'none' method is complementary to the other methods, the normalisation constant q will take the value equal to one minus the probability of the non-'none' methods.

Requested:

Listing 3: Updating Requested Slots

```
for slot in (requested_slot_stats.keys() + hyps["requested-slots"].keys()):
    if slot in informed_slots:
        p = 0
    elif slot not in hyps["requested-slots"].keys():
        p = requested_slot_stats[slot]
    else:
        p = requested_slot_stats[slot] + (1 - requested_slot_stats[slot]) * hyps["
          requested-slots"][slot]
```

In the code, requested_slot_stats stores the probabilities that each slot is requested. Once a slot has been informed by the system (by this we mean the system has provided information about the requested., for instance, address and phone number, etc.), this slot is removed from the set which the system keeps track of, and so we set its p value to 0. Otherwise, we update the probability p that the slot has been requested according to the familiar formula (2).

Results:

Below, I present the results for both the Baseline Tracker and the Focus Tracker for the task given.

```
                    Baseline Tracker
-------------------------------------------------------------------
                |   Joint Goals   |    Requested    |     Method    |
-------------------------------------------------------------------
Accuracy        |     0.5686546   |     0.9137056   |     0.6823856 |
l2              |     0.8344502   |     0.1221110   |     0.5758440 |
```

roc.v2_ca05	0.0000000	0.6055556	0.0020325

Focus Tracker

	Joint Goals	Requested	Method
Accuracy	0.7392510	0.9187817	0.6837725
l2	0.4231740	0.1202923	0.5654613
roc.v2_ca05	0.0000000	0.3204420	0.0223124

Interpretation:

In all cases, the focus tracker outperforms the baseline tracker on accuracy, which is the fraction of turns where the dialogue state hypothesis is correct. The most marked increase in performance is in 'Joint-Goals', the distribution over joint goals. The reason for this is that the baseline tracker has a much harder job of changing to a new goal state in the face of fresh new evidence. This is because the baseline tracker simply returns the goal-state with the best SLU score over all previous SLU outputs. So, if in the past the SLU value for the goal state 'football' was very high, even after much time has elapsed and the user's attention may have moved to some other object, the belief tracker will still be predisposed to return 'football' as the answer.

With the focus tracker, things are different. The focus tracker can decided whether to focus more on past SLU outputs or present SLU outputs, through the value $q_{c,t}$ in formula (1) . If $q_{c,t}$ is close to 0, this ensures that the influence of SLU recommendations from past goals is minimised. And so continuing from our example, this might enable the goal state to change from 'football' to 'rugby' even if 'rugby' has a lower SLU score at a later turn than 'football' at an earlier turn. As a result, the focus tracker is more flexible at handling goal changes and consequently accuracy improves.

L2 is the squared L2-norm of the hypothesised distribution p, which provides an estimate for the confidence of predictions output by the dialogue system. The Focus Tracker provides less confident predictions because it is effectively averaging over probabilities from previous turns, and this is reflected in the tables above.

Q2-MCC Formal Description

For this section, I implemented dialogue policy optimisation using Monte Carlo control (MCC). MCC is a reinforcement learning algorithm that updates the policy function at the end of a sequence of training episodes. When generating training episodes, MCC selects actions with respect to its policy function, Q, epsilon-greedily. After a sufficient number of episodes, MCC updates the policy function, Q. An important feature of our MCC implementation is that the size and complexity of the Q-function is indirectly controlled for by a sparsification threshold v.

Episode Generation Implementation

With probability $1 - \epsilon$, we will choose the best admissible action, otherwise we will choose a random action. We decide which action is best given a belief state, b, by:

- considering all admissible actions a
  - searching for the most similar belief-action pair (b',a') stored in our policy function for the state (b,a)

– setting EST-Q(b,a)= Q(b',a')

- Returning the action a with highest estimated value EST-Q(b,a).

Listing 4: Episode Generation

```
if admissible:
    closestDP, dummy = self.findClosest(flat_belief, action) # pick the action with
      largest Q value
    Q_best = self.dictionary[closestDP].Q
    for a in admissible:
        closestDP, dummy = self.findClosest(flat_belief, a)
        Q = self.dictionary[closestDP].Q
            if Q_best <= Q:
                Q_best, action = Q, a                          #best action so far
else:                                                          # in case of empty list
    action = Settings.random.choice(admissible)
```

The sparsification parameter v is used when updating the Q-function. After the training episode ends, the model checks if each belief-action pair (b,a) in the episode has a distance between itself and its nearest grid point (b',a') in the dictionary smaller than v. If this is so, the model will consider the (b,a) pair as identical to (b',a') and update the Q function by averaging over the returns for (b,a) and the Q value of (b',a'); otherwise (b,a) will be added to the dictionary as a new entry with the appropriate Q value.

Listing 5: Policy Update

```
if len(self.dictionary) == 0.0:
    logger.debug('starting a new dictionary')
    p = DataPoint(b, a)                          # add (b,a) with Q=Return and N=1.0
    v = DataPointValue(Q=Return, N=1.0)
    self.dictionary[p] = v
elif closestDP is None or closestValue > self.nu:
    logger.debug('adding new point')
    p = DataPoint(b, a)            # add (b,a) with Q=Return and N=1.0
    v = DataPointValue(Q=Return, N=1.0)
    self.dictionary[p] = v
else:
    logger.debug('updating Q & N of the grid point')
    v = self.dictionary[closestDP]     # update Q and N with monte carlo algorithm
    v.Q = (v.N * v.Q + Return) / (v.N + 1)
    v.N += 1
self.dictionary[closestDP] = v
```

Results

Below I plot the average accuracy during 'testing' for every 100 dialogues up to 1000 testing dialogues. The different curves represent the results for different sparsification parameters. From the results below, we can see that a setting of v=0.01 gives the best performance. The graphs were generated by adapting, where necessary, the code in the 'scriptGeneration' directory and then using Matlab for plotting.

Too high a value for the sparsifiation parameter, and what we have is over-generalisation. There are too few entires in the policy table, and too much averaging goes on over (belief,action) pairs. Too low a sparsification parameter, and the policy function becomes too complex and we overfit to the data. The 0.01 value gives the happiest medium, balancing over-fitting against over-generalisation, reaching a final accuracy just over 60%. The width of the error bars is 2 standard deviations, which

accounts for roughly 65% of the data. These seem to be relatively constant throughout all graphs, so I will omit discussion of them.

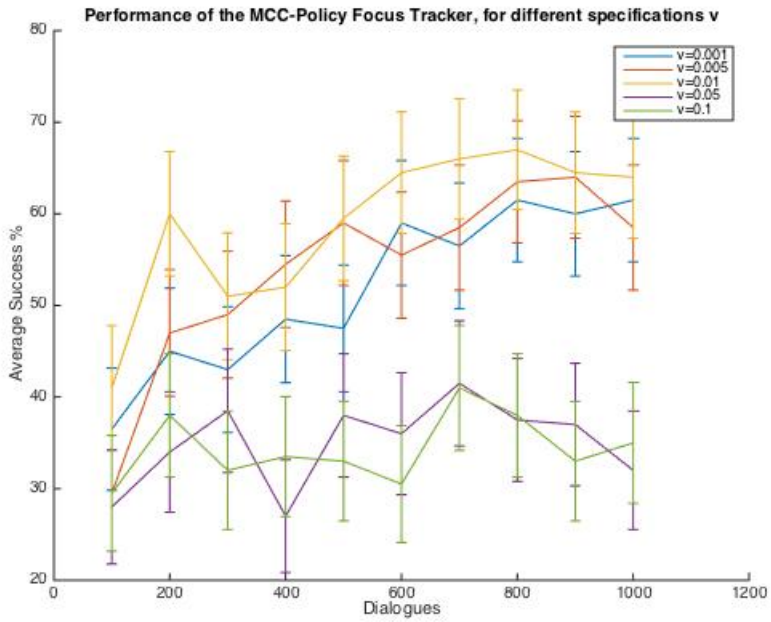

Figure 1: MCC Policy Focus Tracker

It might be instructive to also know the average accuracy throughout the 1000 dialogues, since a user would be using the tracker continuously during 'testing' period, and it isn't only the performance at the end which is of importance. As we can see from the table below, the setting of v=0.01 outperforms the other settings.

	v=0.001	v=0.005	v=0.01	0.05	0.1
Average Accuracy %	51.90	56.90	58.95	34.95	34.35

Q3-GP SARSA

In GP-Sarsa, the unknown Q-function is modeled as a Gaussian process with zero mean and a kernel defined over the belief-action values stored in the dictionary. This should result in improvements in our estimation of the Q-value because with MCC, we only used information from the nearest (belief,action) pairs to estimate the Q-values of unvisited belief-action states. For this question, we are also using a different kind of reinforcement learning algorithm (SARSA), which now involves on-policy updates, that is interleaving policy updates with actions.

Implementation

Below, I implement kernel-updates within an episode.

Listing 6: Kernel Update

```
                k_tilda_new =self.k_tilda(state,action, kernel)
        g_new = []
        if self.terminal:
            g_new = np.zeros(len(self._dictionary))
        else:
```

```
            g_new= np.dot(self._K_tilda_inv,k_tilda_new)
        # your code here... modify current_kernel and estimate_kernel
        current_kernel = kernel.Kernel(state,state)*kernel.ActionKernel(action,action)
        estimate_kernel = np.dot(k_tilda_new,g_new)
        delta_new = 0.0 if self.terminal else (current_kernel - estimate_kernel)
```

Results:

In this practical two kernels, a linear kernel and Gaussian kernel, were examined with different hyper-parameter settings.

Linear Kernel

As an extension to the practical, I tuned explored a few parameter settings for the Linear Kernel. The linear Kernel takes the form: $\mathrm{k}((\mathbf{b}, \mathrm{a}), (\mathbf{b}', \mathrm{a})) = \mathbf{b} \cdot \mathbf{b}' \delta_{\mathrm{a}}(\mathrm{a}')$, with a multiplier σ controlling for the variance of the kernel functions. From my investigations below, I found the setting $\sigma = 7.5$ yielded the best results, which outperform the other settings in accuracy terms by a few percentage points. The parameter sigma determines how much the policy function can deviate from the mean. Small values of σ characterises a Q functions that stays close to the mean value, with larger values giving the Q-Function allowing for more wiggle room.

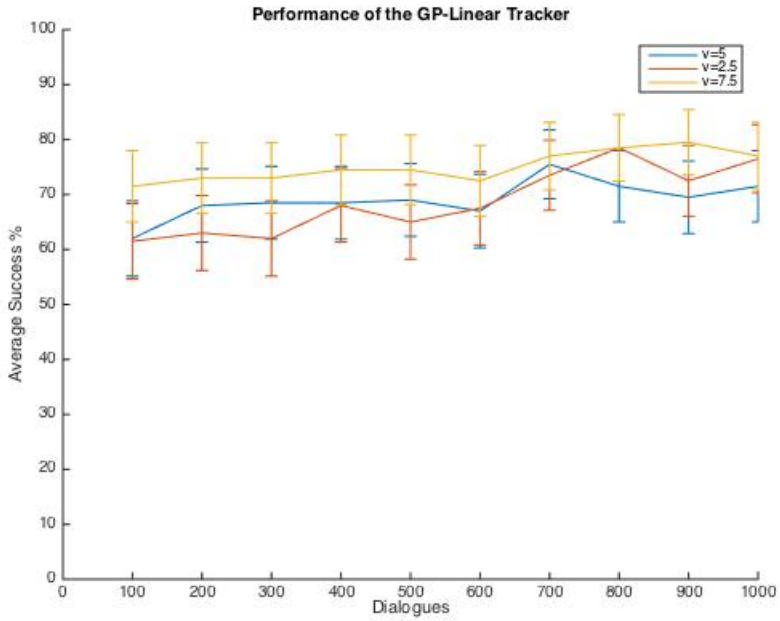

Figure 2: GP-Gauss Focus Tracker

As can be seen from the diagram below, the GP tracker with a linear kernel tends to outperform the MCC tracker by around 15%, yielding a final accuracy of 77% after 1000 dialogues. This is mainly due to the fact we have a more sophisticated way of estimating the value (Q) of belief-action pairs, by using GPs.

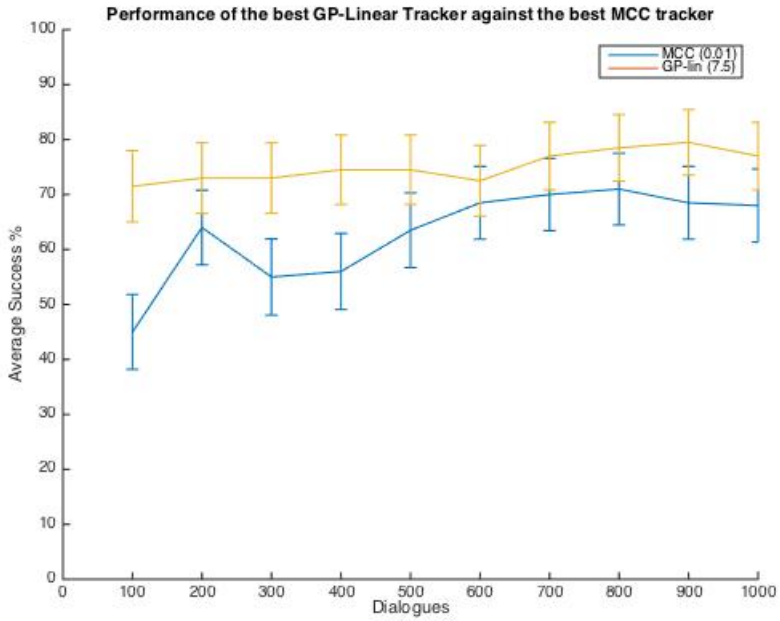

Figure 3: Comparison of Focus Trackers

It would also be instructive to know the average accuracy throughout the 1000 dialogues, since as has been stated previously, a user would be using the tracker continuously during 'testing' period. On average, we can see the GP-Lin tracker obtains accuracies 16% higher.

	MCC	GP-Lin
Average Accuracy %	58.95	75.10

Gaussian Kernel

A Gaussian Kernel has the form $\mathrm{k}((\mathrm{b},\mathrm{a}),(\mathrm{b}',\mathrm{a}')) = \mathrm{p}^2\exp(-\frac{||\mathrm{b}-\mathrm{b}'||_2^2}{2\mathrm{l}^2})\delta_{\mathrm{a}}(\mathrm{a}')$. The parameters p and l will be referred to as the variance and the length-scale. Below, I optimise for the two parameters.

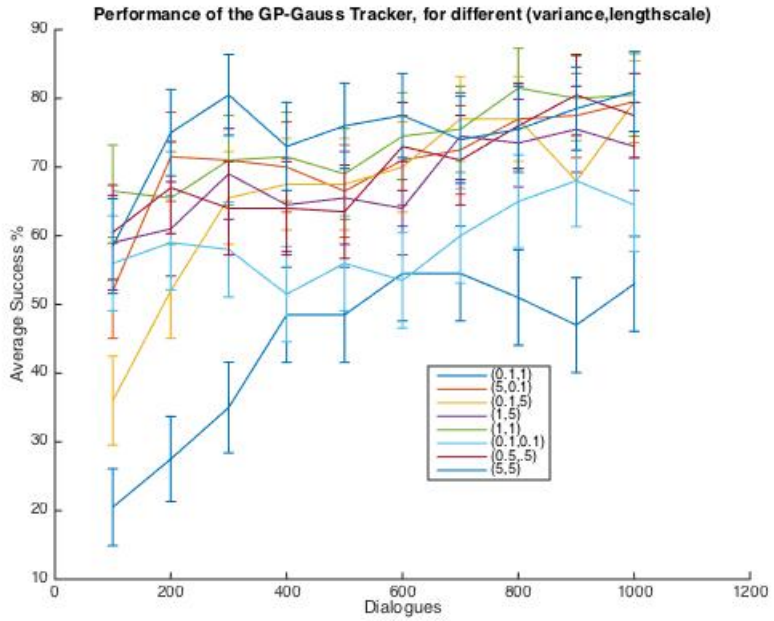

Figure 4: GP-Gauss Focus Tracker

Modifying for length-scale will control for the smoothness of the policy function Q, in terms of **b**. With too large a length-scale our policy function Q becomes spiky, and overfits to close 'b' states. Too small a length-scale and we over-generalise to distant points. The variance p determines how much the policy function can deviate from the mean. Small values of p characterises a Q functions that stays close to the mean value, with larger values giving the Q-Function more wiggle room. From my investigations, the best parameter settings seem to be (p,l)=(1,1) and (p,l)=(5,5). With p and l much larger/smaller, we can overfit/over-generalise to the data.

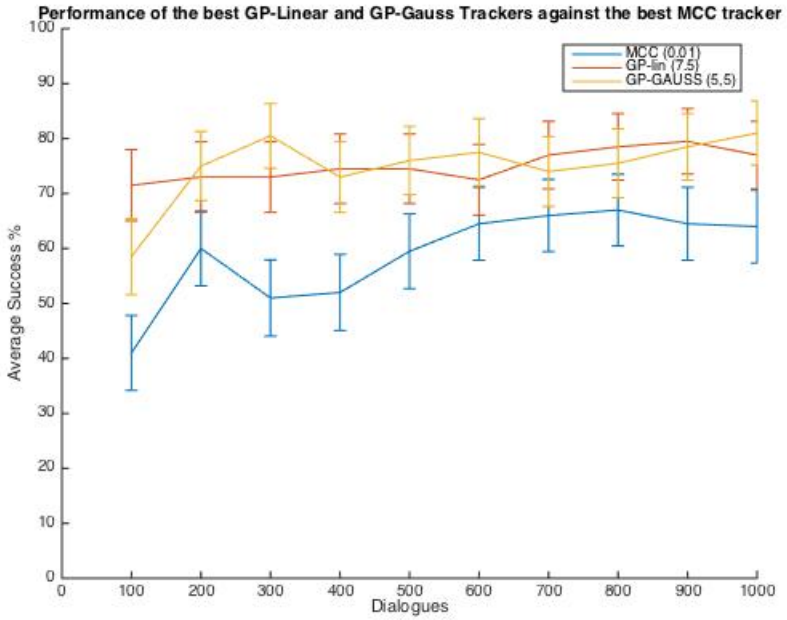

Figure 5: Comparison of Trackers

Summary

In the final figure above, I plotted the best results obtained from the MCC, linear-GP and Gaussian-GP systems. The best system was the Gauss-GP, which after 1000 dialogues obtained an average success rate of 81%. But even from the second dialogue, its success rate was above 73%. With these dialogue systems, it is not just the final accuracy that is of importance but also the accuracy of the system over the entire testing episode, since the user will be using the system throughout testing. The difference between average accuracies is about 15% for the MCC and GP-Lin systems, and 1% for the GP Gauss and GP-Lin systems. The results are presented succinctly in the table below:

	MCC	GP-Lin	GP-Gauss
Average Accuracy %	58.9	75.1	75.95
Final Accuracy %	64.0	77.0	81.0
Highest Accuracy %	67.0	78.5	81.0

In conclusion there is little by way of performance difference between the two GP systems (around 2% in favour of the GP-Gauss), when their parameter setting have been optimised, but both of these outperformed the MCC system (by about 15%). The standard deviations of the success rates were roughly the same and so were omitted from the analysis.

Extended Discussion:

Specific to the Practical:

In the practical we largely explored how Gaussian processes can be successfully applied to dialogue management systems, within the framework of a partially observable Markov decision process. According to paper [3], the principal benefits of this approach are that: GP-based policy optimisation is faster than conventional gradient based methods.

Using a GP based policy on the belief space, to an extent, also reduces the need for hand-crafting features for the belief states. This is because Gaussian processes allow more elaborate kernel functions that can be applied directly to the full action/belief space and, in a sense, make use of their own representations. However, this then gives rise to the need for optimising the kernel function parameters and finding good kernel function structures, which is not a trivial enterprise.

To develop feasible GP RL systems, we also needed to develop sparse approximations for the kernel functions. The use of kernel functions in Gaussian process involves inverting a Gram matrix which grows with the number of data-points, hence the need for the sparsification parameters that indirectly controlled the kernel size.

Other methods, such as those involving recurrent neural network based approaches are not beset with kernel related problems: tuning kernel parameters and minimising the kernel size.

RNN methods for dialogue state tracking:

In paper [4], the author describes how and RNN model for dialogue state tracking can be built. The essential features are that there is a single RNN per slot, taking the most recent dialogue turn as an input, updating its internal memory and calculating an updated belief over the values of the slot. The internal memory will be denoted as m, and the probability distribution of values is denoted as p. Features are constructed for the outputs of the SLU recommender and denoted as f, and g are the internal variables. The process of generating p and m is somewhat complicated, represented in the diagram below.

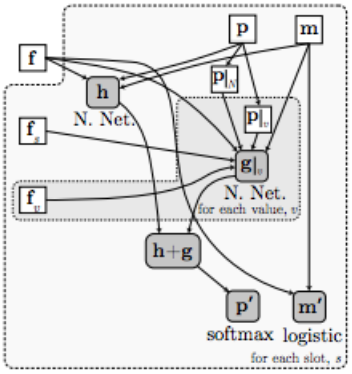

Figure 2: Calculation of $\mathbf{p}'$ and $\mathbf{m}'$ for one turn

Figure 6: RNN structure, from [4]

Instead of using the framework of a MDP, the RNNs are trained using Stochastic Gradient Descent (SGD), maximizing the log probability of the sequences of observed beliefs in the training data. It is noted that gradient clipping is used to avoid the problem of exploding gradients (Pascanu et al., 2012). A regularisation term is included, which penalises the l2 norm of all the parameters.

Paper [4] states of Neural Networks that they are good models that can deal quite naturally with sequential input and outputs from dialogue systems. They also handle high-dimensional inputs well, with little feature engineering for belief states. However, initialisation can be a problem in such models. While NN are typically better than GPs at tracking long-range dependencies in goal states (because of the internal memory), but may be worse than GPs at tracking joint-dependencies in goal-states[5]. And instead of the problem of inverting the Gramm matrix to update the kernel function while training, we are beset with the problem of vanishing gradients (Pascanu et al., 2012).

References

[1] Pei-Hao Su, Milica Gasic; Statistical Spoken Dialogue Systems MLSALT9 Practical

[2] Matthew Henderson et al.; Dialog State Tracking Challenge 2 & 3; http://camdial.org/~mh521/dstc/downloads/handbook.pdf

[3] Milica Gasic et al. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6601004

[4]http://www.matthen.com/research/papers/Word_based_Dialog_State_Tracking_with_Recurrent_Neural_Networks.pdf

[5] http://www.matthen.com/research/papers/Word_based_Dialog_State_Tracking_with_Recurrent_Neural_Networks.pdf

Appendix

```
%% Example plot in Matlab for the last plot of Q3

%%
xvals=[100,200,300,400,500,600,700,800,900,1000]

acc(1,:)=[41.0      60.0      51.0      52.0      59.5      64.5      66.0      67.0      64.5
      64.0]
```

```
acc(2,:)= [71.5    73.0    73.0    74.5    74.5    72.5    77.0    78.5    79.5
    77.0]
acc(3,:)= [58.5    75.0    80.5    73.0    76.0    77.5    74.0    75.5    78.5
    81.0]

std(1,:)=[6.82    6.79    6.93    6.92    6.8    6.63    6.57    6.52    6.63
    6.65 ]
std(2,:)=[6.49    6.41    6.41    6.32    6.32    6.44    6.15    6.04    5.96
    6.15]
std(3,:)=[6.9    6.29    5.88    6.41    6.22    6.12    6.35    6.26    6.04
    5.83]


for i=1:3
    errorbar(xvals,acc(i,:),std(i,:))
end

axis([0 1000 0 100])

xlabel('Dialogues')
ylabel('Average Success %')
legend({'MCC (0.01)','GP-lin (7.5)','GP-GAUSS (5,5)'})
title('Performance of the best GP-Linear and GP-Gauss Trackers against the best MCC
  tracker')
```

```
%% Example output from scriptTest.py used to produce one of the curves in Q2

gRun1_TEST/tra_1_1.log
gRun1_TEST/tra_1_2.log
gRun1_TEST/tra_1_3.log
gRun1_TEST/tra_1_4.log
gRun1_TEST/tra_1_5.log
gRun1_TEST/tra_1_6.log
gRun1_TEST/tra_1_7.log
gRun1_TEST/tra_1_8.log
gRun1_TEST/tra_1_9.log
gRun1_TEST/tra_1_10.log
ITER    num    1        2        3        4        5        6        7        8
  9        10
REWARD    avg    -0.83    2.15    2.15    3.6    4.66    3.62    4.42    5.09
    5.8    4.26
REWARD    std    1.32    1.43    1.42    1.34    1.38    1.52    1.42    1.44
    1.32    1.37
SUCCESS    avg    29.5    47.0    49.0    54.5    59.0    55.5    58.5    63.5
    64.0    58.5
SUCCESS    std    6.32    6.92    6.93    6.9    6.82    6.89    6.83    6.67
    6.65    6.83
TURNS    avg    6.73    7.25    7.66    7.3    7.14    7.48    7.28    7.61
    7.0    7.43
TURNS    std    0.49    0.56    0.54    0.47    0.43    0.59    0.56    0.61
    0.48    0.47
```