Proceedings of the 5th MecE 668 Student Project Seminar
Edmonton, Alberta, Canada, April 10th, 2017

PARAMETER SCREENING FOR CURIOUS REINFORCEMENT LEARNER MOTIVATED BY UNEXPECTED ERROR

Nadia M. Ady
Department of Computing Science, University of Alberta
Patrick M. Pilarski
nmady@ualberta.ca

ABSTRACT

Curiosity is a critical component of intelligence. One method of motivating curious behaviour in computational systems is to use reinforcement learning to learn which decisions maximize the amount of unexpected error observed by a predictive component. However, reinforcement learning algorithms for prediction and control require the system designer to set multiple parameters, and it is unknown how such a curious system's behaviour might vary depending on parameter settings. Eight parameters ($\alpha_p, \gamma_p, \lambda_p$ for prediction, $\alpha_c, \gamma_c, \lambda_c, \epsilon$ for ϵ-greedy control, and β_0 for computation of White's (2015) unexpected error) were tested in an inscribed central composite experimental design. The response variable was the return. We found that the linear effects on return for $\epsilon, \beta_0, \alpha_c$, and γ_p were significant, along with the quadratic interactions between ϵ and β_0, ϵ and γ_p, β_0 and γ_p, and α_p and γ_p.

Keywords: computational curiosity, unexpected error, factor screening, reinforcement learning, intrinsic motivation.

INTRODUCTION

From the multitude of 'why' questions asked by children to the multitude of projects undertaken by researchers, human behaviour is marked by curiosity. Many other animals also engage in behaviour that is well-explained as motivated by curiosity.[2] Curiosity's pervasiveness has lead us to believe it to be a crucial component of intelligent behaviour, yet it is still poorly understood.[3] One way we hope to develop our understanding of curiosity is to model and utilize curiosity in computational systems.

Humans seem to maintain knowledge of their environment and utilize this knowledge to make decisions. One perspective on curiosity is that it should motivate behaviour that helps an agent improve its environment knowledge. In computational systems, one method of maintaining environment knowledge is through estimates or *predictions* of what the system expects to observe in the near future.[6]

In 2014, White suggested a measure of 'unexpected error' for the purpose of generating curious behaviour in a robot.[4] Intuitively, exploring situations that the agent can already predict well (leading to low error) will not improve its environment knowledge. Neither will exploring situations that have already been explored, but have such high variance that we cannot expect predictions to improve.

Motivating a system to maximize cumulative unexpected error observed over time should ideally benefit the system by leading to improved knowledge of its environment.

Reinforcement learning (RL) is a well-studied way for biological systems and machines to learn about the value of situations and choices through trial and error and then utilize those learned values to make decisions.[7] Given a *reward signal* provided to the system, there are standard RL algorithms to learn to predict and/or maximize cumulative reward over time. Using RL algorithms to predict observations, White's 'unexpected error' as the curiosity reward signal, and another RL algorithm to choose actions, we can create a 'curious system.'

However, RL algorithms use multiple parameters, and it is unknown how varying those parameters changes the behaviour of the curious system. While there is no clear measure of the 'curiousness' of observed behaviour, we may simply hope to recognize when it is different. One partial measure of the behaviour in a finite-length run of the system is the cumulative sum of the observation signal, called *return*, G, which may call for maximization or minimization.

$$G = \sum_{t=1}^{\infty} r_t^{(p)} \tag{1}$$

where $r_t^{(p)}$ is the observation or *reward* received from the environment.

The agent in our design is modifying its behaviour to maximize its cumulative unexpected error, but at this stage of the study we have little interest in the total accrued; varying its parameters may impact the magnitude of error observed, and therefore it is unreasonable to compare accrued unexpected error for different parameters.

The objective of this study is to determine which parameters, and which interactions of parameters, impact the return.

MATERIALS AND METHODS

2.1 HARDWARE AND SOFTWARE

All experiments were implemented in Python without parallelism and performed on a Lenovo Flex 3 laptop with four Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz processors running Ubuntu 16.04.1 LTS.

2.2 EXPERIMENTAL SETUP

The experimental setup can be considered in three parts: domain (environment), curious system, and test. We describe each in detail in the following sections, and the Python code is included in Appendix C. We typically refer to an RL system as an 'agent', referencing the *agency* exhibited by making choices. The curious system will hereafter be called the agent.

2.2.1 Test Design

Each test (also called a run) was composed of an initialization and 20,000 iterative discrete timesteps. To initialize the test, the domain's initial state signal is recorded, and the agent takes its initial action. In each timestep, the domain reward and new state are observed. The agent takes another action based on the observed state, then updates. The interaction between the agent and the domain in each timestep is depicted in Figure 1. Our implementation of a test can be found in Appendix C as `Test.run()`.

Within each test, we also maintain a count of how many times each action was taken and a running sum of the domain reward, and use both to compute our **response variables**.

2.2.2 Domain Design

We devised the curiosity bandit, depicted in Figure 1(a), to showcase the behaviour elicited by variations in *domain-delivered reward*. The curiosity bandit has a single state ($\mathcal{S} = \{s_0\}$), but provides the learning agent with three actions ($\mathcal{A} = \{a_1, a_2, a_3\}$), each of which results in a different *domain reward* signal.

If the agent takes a_1, its reward is drawn uniformly randomly from $[-1, 1]$. If the agent takes a_2, it always receives a reward of 0. If the agent takes a_3, then it receives a reward of $\sin(c_1 \cdot t)$, where c is a small constant, **held-constant** at $c_1 = 0.001$ in our experiment, and t is the current timestep (starting at $t = 0$).

The algorithm followed to determine the output state signal and domain reward is **controlled** throughout the experiment. The pseudo-random generation of the reward received after taking action a_1 is **allowed to vary**, because it theoretically simulates noise in the reward signal.

2.2.3 Agent Design

The agent has two components: a prediction learner and a control learner. In this section, we describe the algorithm followed and introduce the parameters varied as design factors.

For the set of all available actions $\mathcal{A}$ and the set of all observable state signals $\mathcal{S}$, the prediction learner uses the TD(λ) algorithm[7] [p. 174] to estimate the *value* function $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, where, for each state $s \in \mathcal{S}$ and each action $a \in \mathcal{A}$, the value of a, given it is taken from s, is defined by

$$Q^\pi(s, a) = \mathbb{E}\left\{ \sum_{k=0}^{\infty} \gamma_p^k r_{t+k+1}^{(p)} \middle| s_t = s, a_t = a \right\} \quad (2)$$

where $\mathbb{E}$ denotes the expected value, s_t, a_t, and $r_{t+1}^{(p)}$ are, respectively, the state observed and the action taken at timestep t, and the resulting domain reward, and $0 \leq \gamma_p \leq 1$ is a constant parameter often called the *continuation probability*.

For the TD(λ) algorithm, the agent starts with some initial estimation Q of Q^π. The initial Q is chosen by the agent designer and while it does typically affect behaviour, the effects are short-term and tend to be small in domains where $\mathcal{S} \times \mathcal{A}$ is small. Therefore, in our experiment it is a **held-constant** factor initialized with $Q(s, a) \leftarrow 0$ for all $s, a \in \mathcal{S} \times \mathcal{A}$.

The algorithm also makes use of an *eligibility trace*, $e : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. It is always initialized with $e(s, a) \leftarrow 0$ for all $s, a \in \mathcal{S} \times \mathcal{A}$.

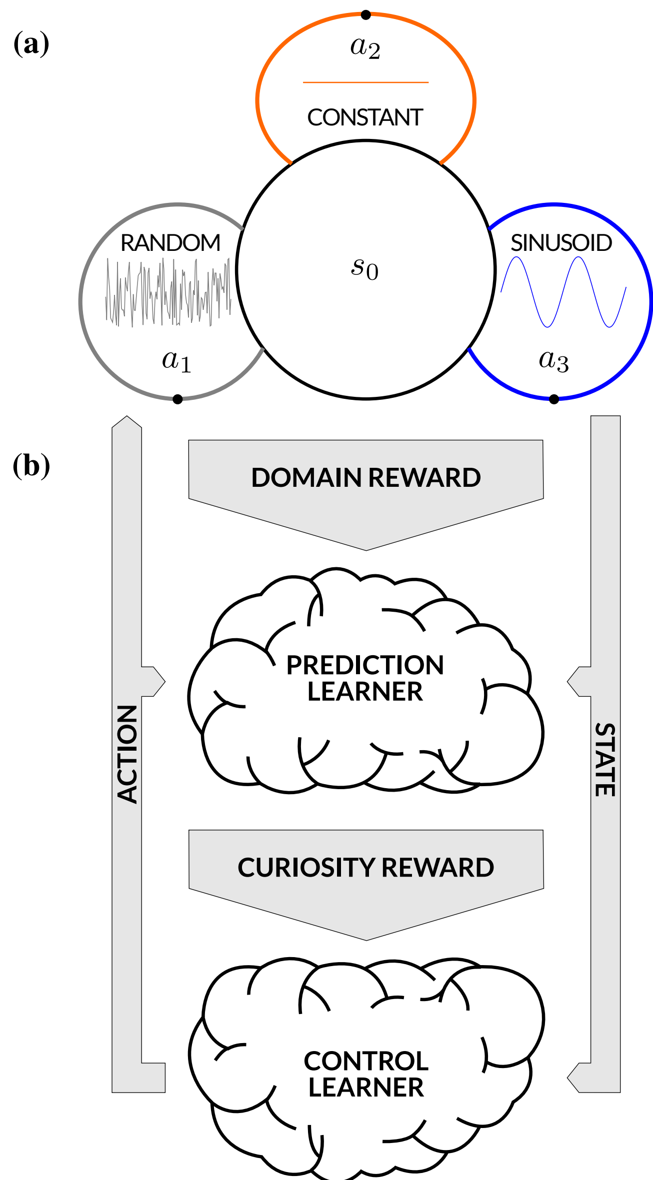

Figure 1: (a) The domain, the *Curiosity Bandit*, is depicted in relation to (b) the agent. At each timestep, the agent observes the domain's output state signal and domain reward signal. The control learner component of the agent chooses an action (a_1, a_2, or a_3), which is observed by the prediction learner and will impact the domain's next state and domain reward. The prediction learner updates its predictions about the domain reward and provides a curiosity reward signal to the control learner, which updates its predictions about the curiosity reward, and uses its predictions to select the next action. Note that by design, the domain always outputs the same state signal s_0, but the reward signal depends on the action taken (and, for action a_3, the timestep).

At each timestep, the agent observes some state s and takes some action a. At the beginning of the next timestep, the agent observes a domain reward $r^{(p)}$ and a new state s' and takes a next action a'. The agent can then use the sum of the observed domain reward and $Q^{\pi}(s', a')$ to update $Q(s, a)$.

To update, the prediction learner uses two components: the eligibility trace, e, and the *temporal difference error* (TD-error), δ. Eligibility is assigned to the action taken, via

$$e(s, a) \leftarrow \min \{e(s, a) + 1, 1\} \tag{3}$$

and the TD-error is computed as

$$\delta \leftarrow r^{(p)} + \gamma_p \, Q(s', a') - Q(s, a) \tag{4}$$

Because there may be an element of randomness to the domain's rule for determining the next state and reward given the current state and action, we do not necessarily want to change our new estimated value to the sample value—we only move it towards that value, so the estimated value for action a from state s is then updated as follows:

$$Q(s, a) \leftarrow Q(s, a) + \alpha_p \delta e(s, a) \tag{5}$$

where α_p is a learning rate parameter. To complete the predictor's update, the eligibility trace then decays as follows:

$$e(s, a) \leftarrow \lambda_p e(s, a) \qquad \forall s, a \in \mathcal{S} \times \mathcal{A} \tag{6}$$

The curiosity reward is computed using the predictor's TD-error. Essentially, we maintain a smoothly averaged estimate, ξ, of recent TD-error and divide by the root sample variance to obtain the curiosity reward (unexpected error). This method was provided by White[5] [p. 121].

The algorithm to maintain ξ utilizes a single parameter, β_0, and a holding variable τ. Holding variable τ and estimate ξ are initialized to $\tau \leftarrow 0$ and $\xi \leftarrow 0$. To update ξ during each timestep, we perform the following two steps:

$$\tau \leftarrow (1 - \beta_0)\tau + \beta_0 \tag{7}$$

$$\xi \leftarrow \left(1 - \frac{\beta_0}{\tau}\right) \xi + \frac{\beta_0}{\tau} \delta \tag{8}$$

The sample variance, $\text{var}(\delta)$, is maintained using a standard incremental algorithm (see Appendix C for the implementation) and the final curiosity reward for the timestep is

$$r^{(c)} = \frac{\xi}{\sqrt{\text{var}(\delta)} + c_2} \tag{9}$$

where c_2 is **held-constant** at $c_2 = 0.001$ in our experiment.

For control, we used ϵ-greedy Watkin's Q(λ)[7] [p. 184]. Q(λ)-learning maintains estimates of the *optimal* curiosity value Q^*, assuming the agent will choose the action with the highest curiosity value in the next step.

Watkin's Q(λ) uses updates analogous to those shown in equations (3)-(5), so our control component also uses analogous parameters α_c, γ_c, and λ_c.

To select an action, a random number between 0 and 1 is drawn. If the random number is less than ϵ, the agent will choose an action randomly (so all actions have equal probability), but otherwise, it chooses the action with the greatest curiosity value (hence the name, ϵ-greedy).

Like the domain, the described algorithms used to make predictions and select actions are **controlled** throughout the experiment, while the pseudo-random generation is **allowed to vary** because it represents the metaphorical 'coin-flip' used to decide in a slightly-random policy. However, the parameters α_p, γ_p, $\lambda_p, \beta_0, \alpha_c, \gamma_c, \lambda_c$, and ϵ are our **manipulated** variables.

2.3 PARAMETER FACTOR RANGES

Learning rates within the interval $[0, 2)$ are generally stable. However, the purpose of the learning rate is to minimize error. For this reason, the learning rates, α_p, α_c, are defined on $(0, 1]$ which are theoretically sound with regards to this purpose.[7] Also, the learning rate is known to typically have a non-linear effect on prediction error and return[8] [pp. 155, 43].

The continuation probabilities chosen for continuing tasks[8] like our domain fall in the interval $[0, 1)$ [p. 53]. Setting $\gamma_p, \gamma_c < 1$ ensures the the

Responding Variable	
Return, G	
Manipulated Variable	**Coding Function**
Learning rate	$\alpha_p = 0.51 + 0.49x_{\alpha_p}$
Continuation probability	$\gamma_p = 0.49 + 0.49x_{\gamma_p}$
Trace decay parameter	$\lambda_p = 0.49 + 0.49x_{\lambda_p}$
Unexpected error parameter	$\beta_0 = 0.5 + 0.49x_{\beta_0}$
Learning rate	$\alpha_c = 0.51 + 0.49x_{\alpha_c}$
Continuation probability	$\gamma_c = 0.49 + 0.49x_{\gamma_c}$
Trace decay parameter	$\lambda_c = 0.49 + 0.49x_{\lambda_c}$
Probability of random action	$\epsilon = 0.5 + 0.49x_{\epsilon}$

Table 1: Summary of full-factorial design and coding for factor levels.

value functions Q^{π} and Q^{*} are bounded. Prior work has shown that γ_p has an important effect on behaviour.[1]

The trace decay parameters λ_p, λ_c are used to assign most credit for an observation to the most recent choice, and decreasing amounts of credit to historical choices. The amount of credit decays by a factor of λ in each step. Like γ_p, γ_c, the parameter is set within the interval $[0, 1)$ for continuing tasks. The trace decay parameter has

The parameter ϵ represents a probability, so is bounded to $[0, 1]$. However, if $\epsilon = 0$, the agent will get stuck taking the action whose curiosity value estimate first exceeds the estimates for the other actions. Similarly, if $\epsilon = 1$, the agent will never utilize its learning; it will act randomly at every timestep. Therefore, we bound ϵ to the range $(0, 1)$.

2.4 EXPERIMENTAL DESIGN

For a design summary, see Table 1.

Since many of the parameters of interest are already expected to affect the response variables, and in many cases to have non-linear effects, we were interested in a response surface method (RSM) design to help capture this information in a simplified model.

A full-factorial design capturing non-linear effects of even five of the eight factors would take nearly 2000 tests for a single replication. Further, we hoped to run twenty replications to account for the randomness impacting each run. While this is a feasible number of tests, given that a single test takes less than a second to complete, the analysis of the resulting data requires a great deal of computation, and a quadratic model can be realized much more efficiently using an RSM.

We chose to utilize an inscribed central composite (CCI) design. CCI designs are suitable RSM designs when the extrema of the included factors represent hard limits. As described in section 2.3, the ranges of interest represent the reasonable limits for each parameter, so a CCI design was a reasonable choice. The test order was fully randomized.

RESULTS

The raw data can be found in Appendix A. The linear models were created using STATISTICA 13.

3.5 RESULTS FOR RETURN

An ANOVA was performed to determine the significant factors and interactions. Our initial model including all linear effects and quadratic interactions was validated using a run sequence plot of the residuals (5), a normal probability plot of the residuals (6), and a scatter plot of the predicted values against the residuals (7), shown in Appendix B. We defer discussion of the validation plots to section 3.6. From the Effect Estimates (Table B1), ANOVA (Table B2), and Pareto chart of standardized effects (Figure 2), we determined that the linear effects for $\epsilon, \beta_0, \alpha_c$, and γ_p were significant, along with the quadratic interactions between ϵ and β_0, ϵ and γ_p, β_0 and γ_p, and α_p and γ_p.

We reduced our model by removing all insignificant effects, up to fulfilling the hierarchy principle (we keep the linear effect of α_p, despite its relatively high p-value). We then performed the same statistical analysis for our new model. Again, the model was visually validated (Figures 6 and 8, and 10 in Appendix B) and the Effect Estimates (Table B3) and ANOVA (Table B4) computed. As is shown by

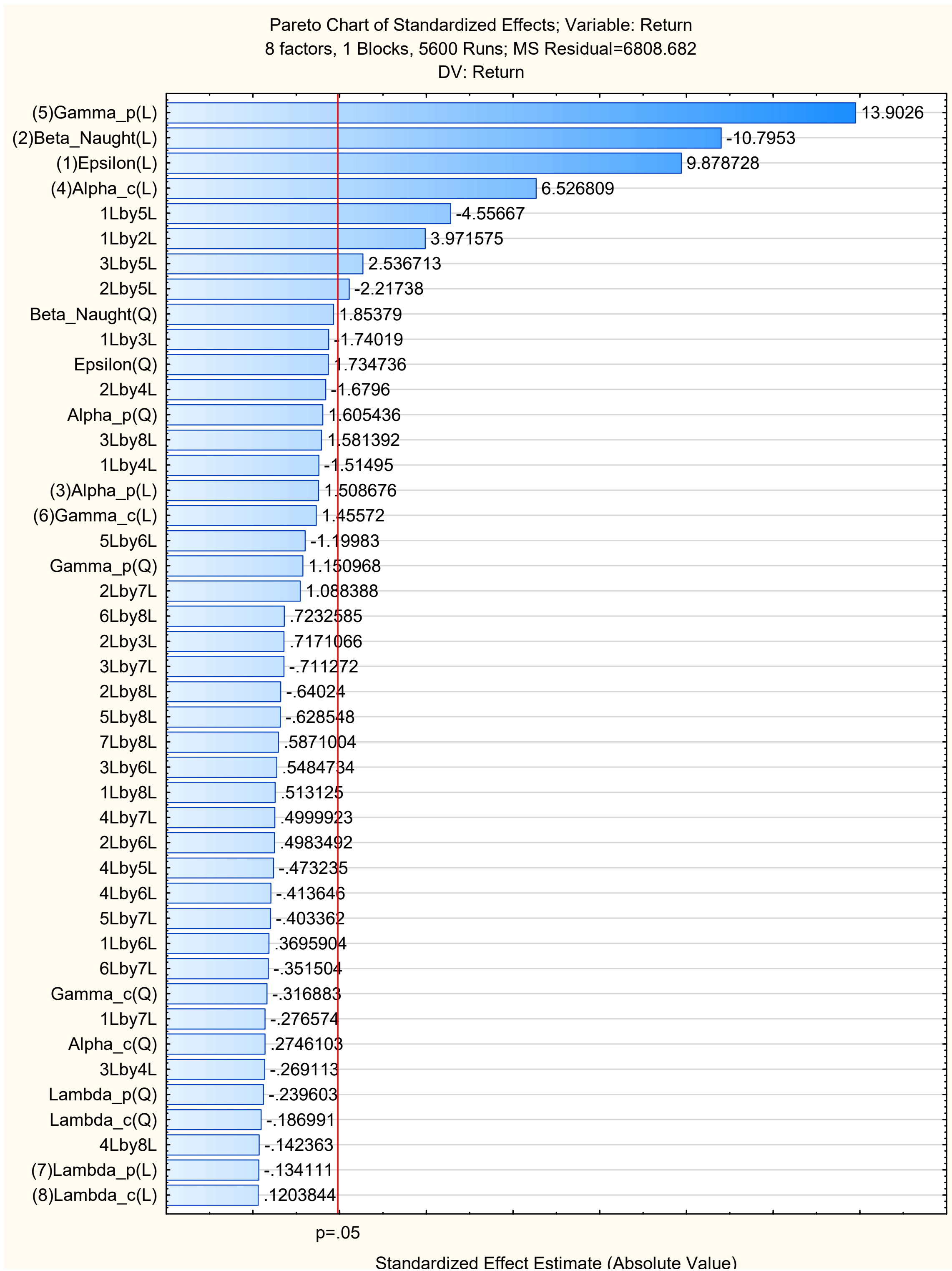

Figure 2: A Pareto chart showing the standardized effects on return for the initial model with all linear and quadratic effects.

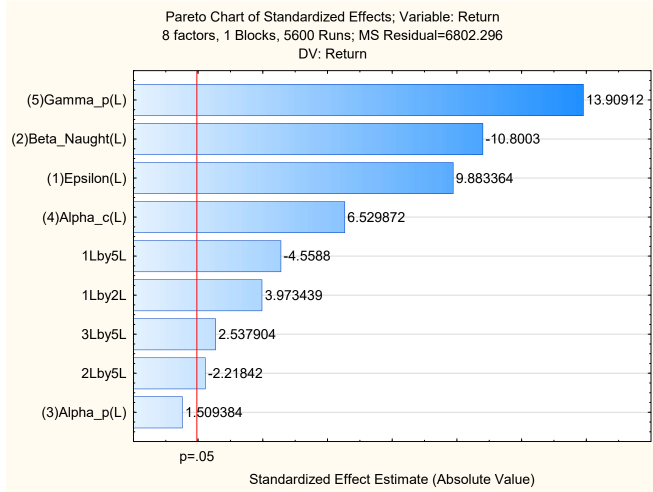

Figure 3: In this Pareto chart, the effects with bars surpassing the red line (that is, those effects which have a component to the right of the red line) are significant in the reduced model.

the Pareto chart in Figure 3, the effects listed in the previous paragraph remained significant.

With our reduced model, the return G can be written as a function of coded parameters as follows:

$$\begin{aligned} G = 139.5106 + 10.9775x_{\epsilon} - 11.9960x_{\beta_0} \\ + 1.6765x_{\alpha_p} + 7.2528x_{\alpha_c} + 15.4490x_{\gamma_p} \\ + 4.5799x_{\epsilon}x_{\beta_0} - 5.2546x_{\epsilon}x_{\gamma_p} \\ - 2.5570x_{\beta_0}x_{\gamma_p} + 2.9253x_{\alpha_p}x_{\gamma_p} \end{aligned} \quad (10)$$

3.6 MODEL VALIDATION

The assumptions of the ANOVA procedure require that the residuals of the linear model are normal.

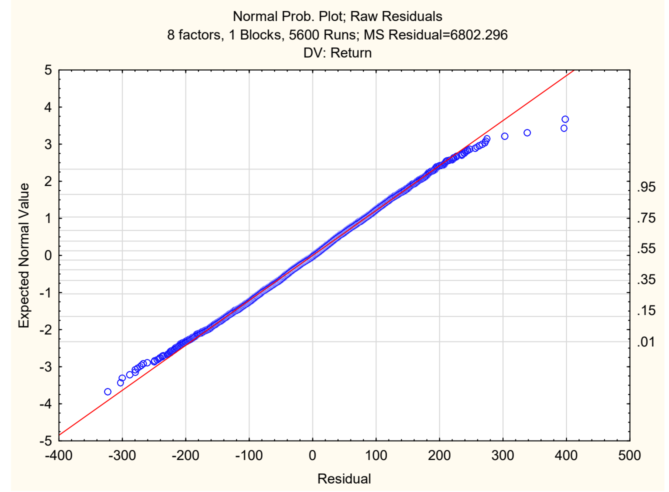

Figure 4: This plot shows the normal probability plot for the residuals, given our reduced model.

Unfortunately, we can see in Figure 4, that there are outliers. Interestingly, those outliers are runs where $x_{\epsilon} = -1$, and all other coded variables are set to 0. This suggests that our model may fail in nearby cases.

On the other hand, neither the run-sequence plot of residuals in Figure 8 nor the plot of residuals as a function of value in Figure 10 appear to show any trend suggesting further invalidity of the model.

3.7 OPTIMIZATION

A system designer may be interested in maximizing return while still using this curiosity method. We used `sqp` in Octave, and found that the following coded values maximized return according to our model, with a predicted expected return of 182.51.

$$x_{\epsilon} = 1 \quad (11)$$

$$x_{\beta_0} = -1 \quad (12)$$

$$x_{\alpha_p} = 1 \quad (13)$$

$$x_{\alpha_c} = 1 \quad (14)$$

$$x_{\gamma_p} = 1 \quad (15)$$

DISCUSSION

The initial objectives of this work were to find parameters that significantly affect the behaviour of

a reinforcement learning agent controlled using White's unexpected error as a curiosity reward.

To the best of the author's knowledge, there have been no prior attempts to determine which parameters in a curious agent impact its behaviour.

Using return as the response variable provided limited insight into this issue. Parameters which result in significantly different final return must have resulted in significantly different behaviour to do so. However, this experiment does not exhaust the possibility that some parameters which cause significantly different behaviour could still result in similar return.

Despite its limitations, return is an interesting response variable, as there could be situations where the system designer would like to maximize return while still requiring the learning system to utilize this kind of curiosity reward.

In future work it will be crucial to utilize more descriptive measures of behaviour than the return. Such measures could include the average probability of each action or other measures of the agent's predictive error.

CONCLUSIONS

We found that the significant factors were the linear effects for $\epsilon, \beta_0, \alpha_c$, and γ_p, along with the quadratic interactions between ϵ and β_0, ϵ and γ_p, β_0 and γ_p, and α_p and γ_p.

To maximize return, we found that the best values in our utilized ranges for the significant parameters were as follows:

$$\epsilon = 0.99 \tag{16}$$

$$\beta_0 = 0.01 \tag{17}$$

$$\alpha_p = 1 \tag{18}$$

$$\alpha_c = 1 \tag{19}$$

$$\gamma_p = 0.98 \tag{20}$$

ACKNOWLEDGMENTS

Thank you to Dr. Patrick M. Pilarski and Dr. Kajsa Duke for their guidance on this project.

REFERENCES

1. Ady, N. M. & Pilarski, P. M. (2016, December). Domains for Investigating Curious Behaviour in Reinforcement Learning Agents. Poster session presented at the Women in Machine Learning Workshop, Barcelona, Spain.

2. Glickman, S. E., & Sroges, R. W. (1966). Curiosity in zoo animals. *Behaviour*, 26(1), 151-187.

3. Gottlieb, J., Oudeyer, P. Y., Lopes, M., & Baranes, A. (2013). Information-seeking, curiosity, and attention: computational and neural mechanisms. *Trends in cognitive sciences*, 17(11), 585-593.

4. White, A., Modayil, J., & Sutton, R. S. (2014, July). Surprise and curiosity for big data robotics. In *AAAI-14 Workshop on Sequential Decision-Making with Big Data, Quebec City, Quebec, Canada.*

5. White, A. (2015). *Developing a predictive approach to knowledge* (Doctoral dissertation, University of Alberta).

6. Modayil, J., White, A., & Sutton, R. S. (2014). Multi-timescale nexting in a reinforcement learning robot. *Adaptive Behavior, 22*(2), 146-160.

7. Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction* (1st Ed.). Cambridge: MIT press.

8. Sutton, R. S., & Barto, A. G. (2016). *Reinforcement learning: An introduction* (2nd Ed.). Manuscript in preparation. Retrieved from: http://incompleteideas.net/sutton/book/bookdraft2016sep.pdf on April 2, 2017.

Appendix A Raw Data

The raw data is available at
https://drive.google.com/a/ualberta.ca/file/d/0B5rsyN1Hdb1qd1B4OGQ3dGxVc0U/view?usp=sharing

Appendix B Additional Tables and Plots (Return)

Effect Estimates; Var.:Return; R-sqr=.08709; Adj:.07986 (stats_cci)8 factors, 1 Blocks, 5600 Runs; MS Residual=6808.682DV: Return

Factor	Effect	Std.Err.	t(5555)	p	-95.% (Cnf.Limt)	+95.% (Cnf.Limt)	Coeff.	Std.Err. (Coeff.)	-95.% (Cnf.Limt)	+95.% (Cnf.Limt)
Mean/Interc.	131.0435	6.227438	21.0429	0.000000	118.8352	143.2517	131.0435	6.227438	118.8352	143.2517
(1)Epsilon (L)	21.9551	2.222461	9.8787	0.000000	17.5982	26.3120	10.9775	1.111231	8.7991	13.1560
Epsilon (Q)	5.0775	2.926942	1.7347	0.082843	-0.6605	10.8154	2.5387	1.463471	-0.3302	5.4077
(2)Beta_Naught(L)	-23.9921	2.222461	-10.7953	0.000000	-28.3490	-19.6352	-11.9960	1.111231	-14.1745	-9.8176
Beta_Naught(Q)	5.4259	2.926942	1.8538	0.063822	-0.3120	11.1639	2.7130	1.463471	-0.1560	5.5819
(3)Alpha_p(L)	3.3530	2.222461	1.5087	0.131438	-1.0039	7.7099	1.6765	1.111231	-0.5020	3.8549
Alpha_p(Q)	4.6990	2.926942	1.6054	0.108455	-1.0389	10.4370	2.3495	1.463471	-0.5195	5.2185
(4)Alpha_c(L)	14.5056	2.222461	6.5268	0.000000	10.1487	18.8625	7.2528	1.111231	5.0743	9.4312
Alpha_c(Q)	0.8038	2.926942	0.2746	0.783626	-4.9342	6.5417	0.4019	1.463471	-2.4671	3.2709
(5)Gamma_p(L)	30.8980	2.222461	13.9026	0.000000	26.5411	35.2549	15.4490	1.111231	13.2705	17.6274
Gamma_p(Q)	3.3688	2.926942	1.1510	0.249795	-2.3691	9.1068	1.6844	1.463471	-1.1846	4.5534
(6)Gamma_c(L)	3.2353	2.222461	1.4557	0.145527	-1.1216	7.5922	1.6176	1.111231	-0.5608	3.7961
Gamma_c(Q)	-0.9275	2.926942	-0.3169	0.751344	-6.6655	4.8105	-0.4637	1.463471	-3.3327	2.4052
(7)Lambda_p(L)	-0.2981	2.222461	-0.1341	0.893320	-4.6549	4.0588	-0.1490	1.111231	-2.3275	2.0294
Lambda_p(Q)	-0.7013	2.926942	-0.2396	0.810647	-6.4393	5.0366	-0.3507	1.463471	-3.2196	2.5183
(8)Lambda_c(L)	0.2675	2.222461	0.1204	0.904183	-4.0893	4.6244	0.1338	1.111231	-2.0447	2.3122
Lambda_c(Q)	-0.5473	2.926942	-0.1870	0.851675	-6.2853	5.1906	-0.2737	1.463471	-3.1426	2.5953
1L by 2L	9.1599	2.306357	3.9716	0.000072	4.6385	13.6812	4.5799	1.153179	2.3193	6.8406
1L by 3L	-4.0135	2.306357	-1.7402	0.081882	-8.5349	0.5079	-2.0067	1.153179	-4.2674	0.2539
1L by 4L	-3.4940	2.306357	-1.5150	0.129842	-8.0154	1.0273	-1.7470	1.153179	-4.0077	0.5137
1L by 5L	-10.5093	2.306357	-4.5567	0.000005	-15.0307	-5.9879	-5.2546	1.153179	-7.5153	-2.9940
1L by 6L	0.8524	2.306357	0.3696	0.711702	-3.6690	5.3738	0.4262	1.153179	-1.8345	2.6869
1L by 7L	-0.6379	2.306357	-0.2766	0.782117	-5.1592	3.8835	-0.3189	1.153179	-2.5796	1.9417
1L by 8L	1.1834	2.306357	0.5131	0.607884	-3.3379	5.7048	0.5917	1.153179	-1.6690	2.8524
2L by 3L	1.6539	2.306357	0.7171	0.473338	-2.8675	6.1753	0.8270	1.153179	-1.4337	3.0876
2L by 4L	-3.8737	2.306357	-1.6796	0.093092	-8.3951	0.6476	-1.9369	1.153179	-4.1976	0.3238
2L by 5L	-5.1141	2.306357	-2.2174	0.026638	-9.6354	-0.5927	-2.5570	1.153179	-4.8177	-0.2964
2L by 6L	1.1494	2.306357	0.4983	0.618258	-3.3720	5.6707	0.5747	1.153179	-1.6860	2.8354
2L by 7L	2.5102	2.306357	1.0884	0.276471	-2.0111	7.0316	1.2551	1.153179	-1.0056	3.5158
2L by 8L	-1.4766	2.306357	-0.6402	0.522043	-5.9980	3.0447	-0.7383	1.153179	-2.9990	1.5224
3L by 4L	-0.6207	2.306357	-0.2691	0.787852	-5.1420	3.9007	-0.3103	1.153179	-2.5710	1.9503
3L by 5L	5.8506	2.306357	2.5367	0.011217	1.3292	10.3719	2.9253	1.153179	0.6646	5.1860
3L by 6L	1.2650	2.306357	0.5485	0.583389	-3.2564	5.7863	0.6325	1.153179	-1.6282	2.8932
3L by 7L	-1.6404	2.306357	-0.7113	0.476946	-6.1618	2.8809	-0.8202	1.153179	-3.0809	1.4405
3L by 8L	3.6473	2.306357	1.5814	0.113845	-0.8741	8.1686	1.8236	1.153179	-0.4371	4.0843
4L by 5L	-1.0914	2.306357	-0.4732	0.636064	-5.6128	3.4299	-0.5457	1.153179	-2.8064	1.7150
4L by 6L	-0.9540	2.306357	-0.4136	0.679150	-5.4754	3.5673	-0.4770	1.153179	-2.7377	1.7837
4L by 7L	1.1532	2.306357	0.5000	0.617100	-3.3682	5.6745	0.5766	1.153179	-1.6841	2.8373
4L by 8L	-0.3283	2.306357	-0.1424	0.886798	-4.8497	4.1930	-0.1642	1.153179	-2.4249	2.0965
5L by 6L	-2.7672	2.306357	-1.1998	0.230257	-7.2886	1.7541	-1.3836	1.153179	-3.6443	0.8771
5L by 7L	-0.9303	2.306357	-0.4034	0.686698	-5.4517	3.5911	-0.4651	1.153179	-2.7258	1.7955
5L by 8L	-1.4497	2.306357	-0.6285	0.529671	-5.9710	3.0717	-0.7248	1.153179	-2.9855	1.5359
6L by 7L	-0.8107	2.306357	-0.3515	0.725224	-5.3321	3.7107	-0.4053	1.153179	-2.6660	1.8553
6L by 8L	1.6681	2.306357	0.7233	0.469552	-2.8533	6.1895	0.8340	1.153179	-1.4266	3.0947
7L by 8L	1.3541	2.306357	0.5871	0.557160	-3.1673	5.8754	0.6770	1.153179	-1.5836	2.9377

Table B1: Given our initial model, this table shows all linear effects and quadratic interactions.

ANOVA; Var.:Return; R-sqr=.08709; Adj:.07986 (stats_cci)
8 factors, 1 Blocks, 5600 Runs; MS Residual=6808.682
DV: Return

Factor	SS	df	MS	F	p
(1)Epsilon (L)	664454	1	664454	97.5893	0.000000
Epsilon (Q)	20489	1	20489	3.0093	0.082843
(2)Beta_Naught(L)	793468	1	793468	116.5378	0.000000
Beta_Naught(Q)	23398	1	23398	3.4365	0.063822
(3)Alpha_p(L)	15497	1	15497	2.2761	0.131438
Alpha_p(Q)	17549	1	17549	2.5774	0.108455
(4)Alpha_c(L)	290045	1	290045	42.5992	0.000000
Alpha_c(Q)	513	1	513	0.0754	0.783626
(5)Gamma_p(L)	1315998	1	1315998	193.2823	0.000000
Gamma_p(Q)	9020	1	9020	1.3247	0.249795
(6)Gamma_c(L)	14428	1	14428	2.1191	0.145527
Gamma_c(Q)	684	1	684	0.1004	0.751344
(7)Lambda_p(L)	122	1	122	0.0180	0.893320
Lambda_p(Q)	391	1	391	0.0574	0.810647
(8)Lambda_c(L)	99	1	99	0.0145	0.904183
Lambda_c(Q)	238	1	238	0.0350	0.851675
1L by 2L	107396	1	107396	15.7734	0.000072
1L by 3L	20618	1	20618	3.0282	0.081882
1L by 4L	15626	1	15626	2.2951	0.129842
1L by 5L	141370	1	141370	20.7632	0.000005
1L by 6L	930	1	930	0.1366	0.711702
1L by 7L	521	1	521	0.0765	0.782117
1L by 8L	1793	1	1793	0.2633	0.607884
2L by 3L	3501	1	3501	0.5142	0.473338
2L by 4L	19208	1	19208	2.8210	0.093092
2L by 5L	33477	1	33477	4.9168	0.026638
2L by 6L	1691	1	1691	0.2484	0.618258
2L by 7L	8065	1	8065	1.1846	0.276471
2L by 8L	2791	1	2791	0.4099	0.522043
3L by 4L	493	1	493	0.0724	0.787852
3L by 5L	43813	1	43813	6.4349	0.011217
3L by 6L	2048	1	2048	0.3008	0.583389
3L by 7L	3445	1	3445	0.5059	0.476946
3L by 8L	17027	1	17027	2.5008	0.113845
4L by 5L	1525	1	1525	0.2240	0.636064
4L by 6L	1165	1	1165	0.1711	0.679150
4L by 7L	1702	1	1702	0.2500	0.617100
4L by 8L	138	1	138	0.0203	0.886798
5L by 6L	9802	1	9802	1.4396	0.230257
5L by 7L	1108	1	1108	0.1627	0.686698
5L by 8L	2690	1	2690	0.3951	0.529671
6L by 7L	841	1	841	0.1236	0.725224
6L by 8L	3562	1	3562	0.5231	0.469552
7L by 8L	2347	1	2347	0.3447	0.557160
Error	37822229	5555	6809		
Total SS	41430353	5599			

Table B2: Given our initial model for all linear effects and quadratic interactions, this table provides ANOVA data.

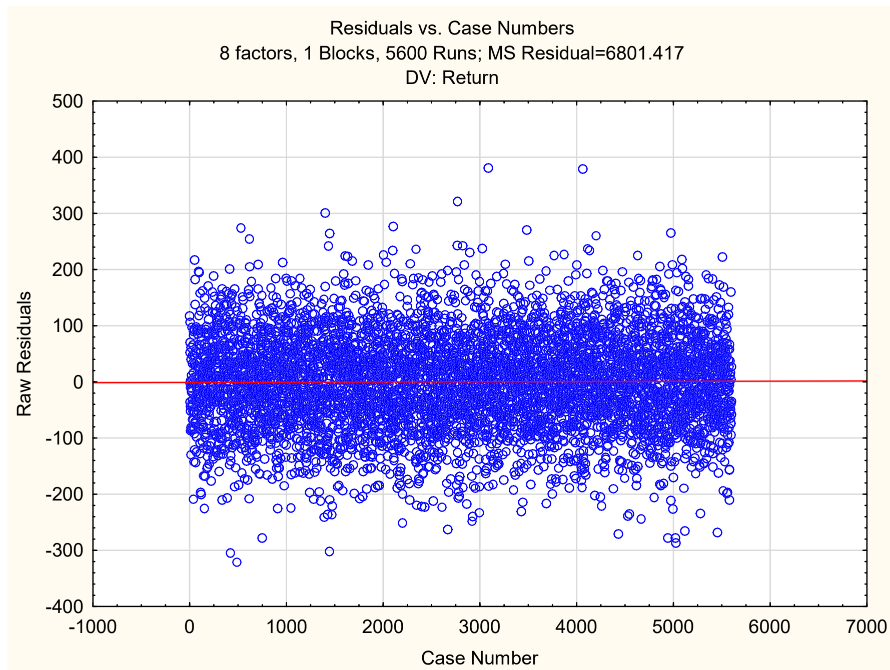

Figure 5: This plot shows the raw residuals for the initial model including all linear and quadratic effects as a function of case number (equivalently, the raw residuals are shown in the run sequence order).

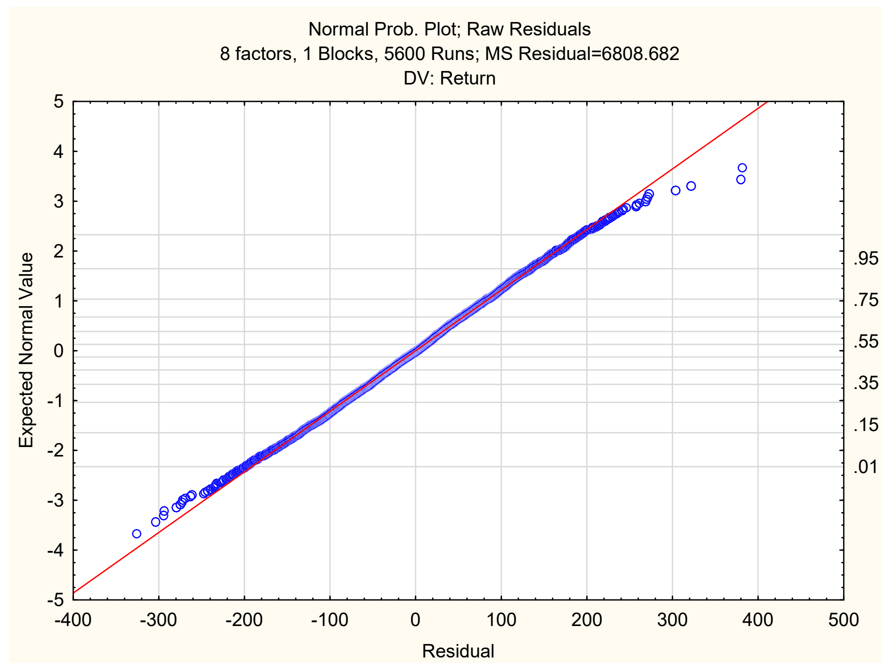

Figure 6: This plot shows the normal probability plot for the residuals, given our initial model including all linear and quadratic effects.

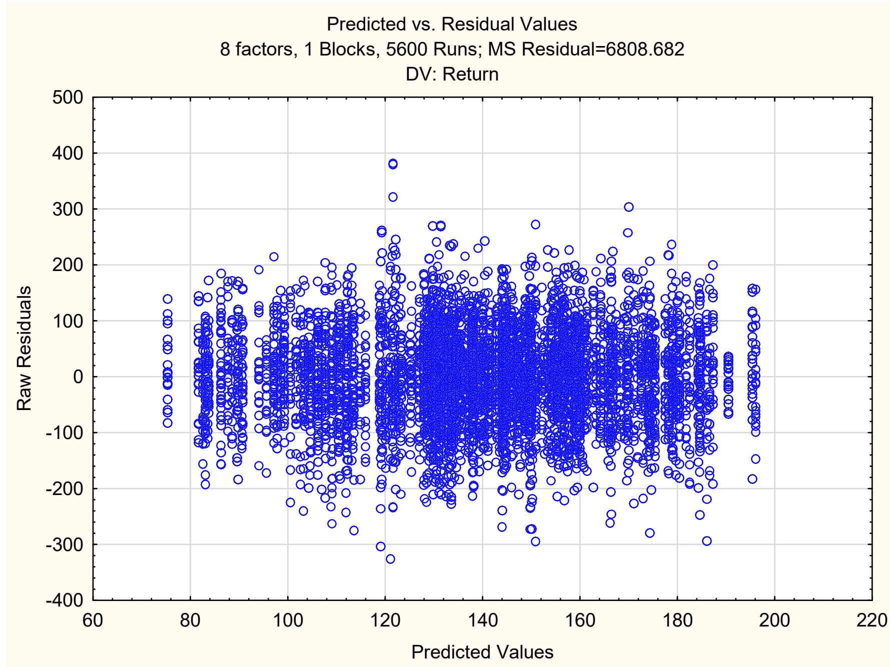

Figure 7: This plot shows the raw residuals as a function of the predicted values for the initial model including all linear and quadratic effects.

Effect Estimates; Var.:Return; R-sqr=.0822; Adj:.08072 (stats_cci)8 factors, 1 Blocks, 5600 Runs; MS Residual=6802.296DV: Return

Factor	Effect	Std.Err.	t(5590)	p	-95.% (Cnf.Limt)	+95.% (Cnf.Limt)	Coeff.	Std.Err. (Coeff.)	-95.% (Cnf.Limt)	+95.% (Cnf.Limt)
Mean/Interc.	139.5106	1.102132	126.5824	0.000000	137.3500	141.6712	139.5106	1.102132	137.3500	141.6712
(1)Epsilon (L)	21.9551	2.221419	9.8834	0.000000	17.6002	26.3099	10.9775	1.110709	8.8001	13.1550
(2)Beta_Naught(L)	-23.9921	2.221419	-10.8003	0.000000	-28.3469	-19.6372	-11.9960	1.110709	-14.1735	-9.8186
(3)Alpha_p(L)	3.3530	2.221419	1.5094	0.131257	-1.0019	7.7078	1.6765	1.110709	-0.5009	3.8539
(4)Alpha_c(L)	14.5056	2.221419	6.5299	0.000000	10.1507	18.8604	7.2528	1.110709	5.0754	9.4302
(5)Gamma_p(L)	30.8980	2.221419	13.9091	0.000000	26.5431	35.2528	15.4490	1.110709	13.2716	17.6264
1L by 2L	9.1599	2.305275	3.9734	0.000072	4.6406	13.6791	4.5799	1.152638	2.3203	6.8396
1L by 5L	-10.5093	2.305275	-4.5588	0.000005	-15.0285	-5.9901	-5.2546	1.152638	-7.5143	-2.9950
2L by 5L	-5.1141	2.305275	-2.2184	0.026566	-9.6333	-0.5948	-2.5570	1.152638	-4.8167	-0.2974
3L bv 5L	5.8506	2.305275	2.5379	0.011179	1.3313	10.3698	2.9253	1.152638	0.6657	5.1849

Table B3: This table provides the main effects and model coefficients for our reduced model.

ANOVA; Var.:Return; R-sqr=.0822; Adj:.08072 (stats_cci)
8 factors, 1 Blocks, 5600 Runs; MS Residual=6802.296
DV: Return

Factor	SS	df	MS	F	p
(1)Epsilon (L)	664454	1	664454	97.6809	0.000000
(2)Beta_Naught(L)	793468	1	793468	116.6472	0.000000
(3)Alpha_p(L)	15497	1	15497	2.2782	0.131257
(4)Alpha_c(L)	290045	1	290045	42.6392	0.000000
(5)Gamma_p(L)	1315998	1	1315998	193.4637	0.000000
1L by 2L	107396	1	107396	15.7882	0.000072
1L by 5L	141370	1	141370	20.7827	0.000005
2L by 5L	33477	1	33477	4.9214	0.026566
3L by 5L	43813	1	43813	6.4410	0.011179
Error	38024835	5590	6802		
Total SS	41430353	5599			

Table B4: This table provides ANOVA data for our reduced model.

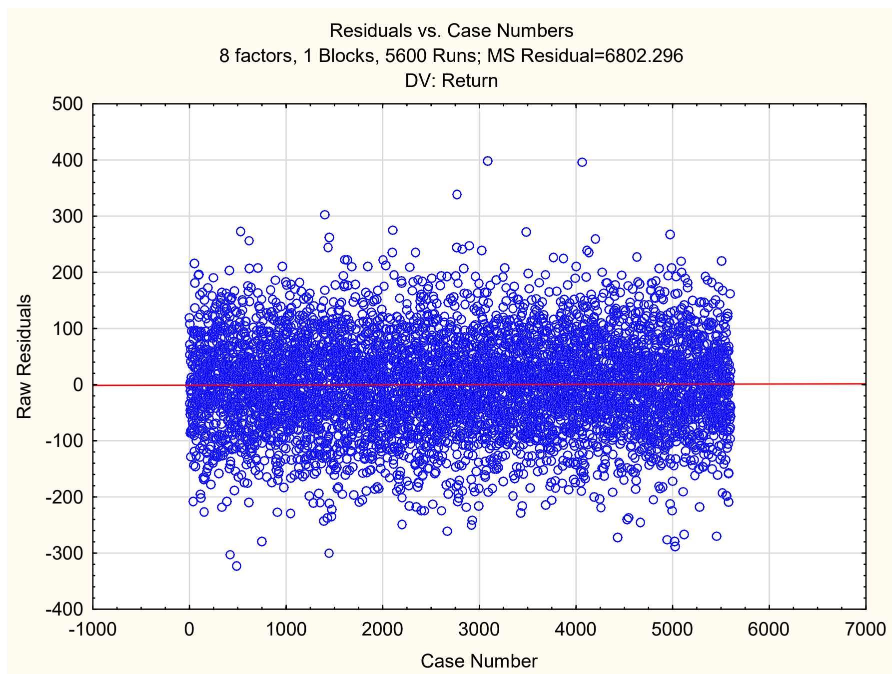

Figure 8: This plot shows the raw residuals for the reduced model as a function of case number (equivalently, the raw residuals are shown in the run sequence order).

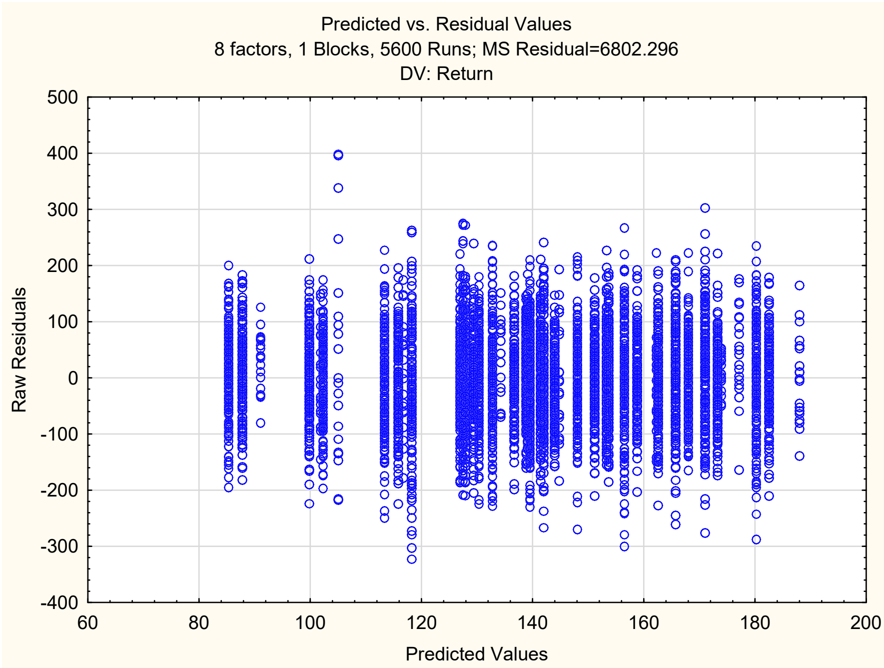

Figure 9: This plot shows the raw residuals as a function of the predicted values for the reduced model.

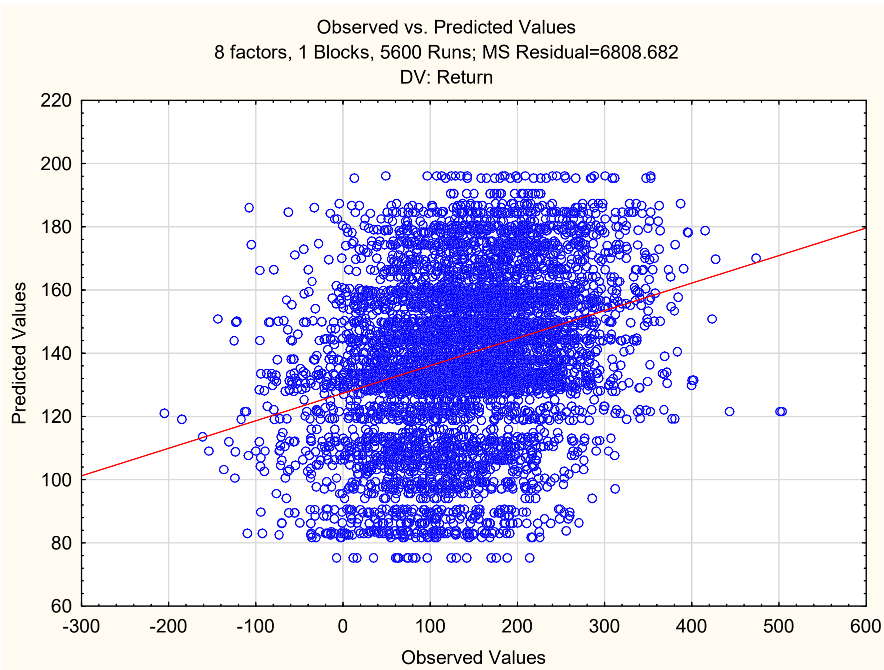

Figure 10: This plot shows values predicted by the reduced model vs observed values.

Appendix C

The following code runs all of the experiments and produces a csv (comma-separated-values) file containing the raw experimental data.

```
# parameter_screening.py
# Copyright (C) 2017 Nadia Ady
#
# This module is part of the curiosity project.
# To run, type: python parameter_screening.py

import numpy    # used for random functions


class Test(object):
    def __init__(self, domain, num_steps):
        self.domain = domain
        self.num_steps = num_steps
        self.return_sum = 0.0

        # tracks counts for each state-action pair
        self.count = {}
        for state in domain.get_state_set():
            self.count[state] = {}
            for action in domain.get_action_set():
                self.count[state][action] = 0

    def run(self, agent, initial_state):
        self.return_sum = 0.0     # reset before starting a new test
        state = initial_state
        action = agent.get_action(state)
        for step_num in range(self.num_steps):

            self.count[state][action] += 1

            domain_reward, new_state = self.domain.sample(state,
                                                          action)
            new_action = agent.get_action(new_state)

            agent.update(state, action, domain_reward,
                         new_state, new_action)

            state = new_state
            action = new_action

            self.return_sum += domain_reward
```

```python
class CuriosityBandit(object):
    def __init__(self, offset=0.001, start_time=0,
                 update_sin_every_step=True, random_seed=None):
        # constant to affect the frequency of the sinusoidal action
        self.offset = offset
        # sin_argument holds self.offset * timestep
        self.sin_argument = start_time
        # if False, sinusoid action only shifts when action is taken
        self.update_sin_every_step = update_sin_every_step

        if random_seed is None:
            numpy.random.seed()
        else:
            numpy.random.seed(random_seed)

    def sample(self, state, action):
        if self.update_sin_every_step or action == 1:
            self.sin_argument += self.offset
        reward = self.get_reward(action)
        return reward, state

    def get_reward(self, action):
        if action == 0:       # random
            return 2 * numpy.random.random_sample() - 1
        elif action == 1:     # sinusoidal
            return numpy.sin(self.sin_argument)
        elif action == 2:     # constant
            return 0

    def get_state_set(self):
        """The returned list of states is a singleton."""
        return [0]

    def get_action_set(self):
        return [1, 0, 2]

    def get_initial_state(self):
        return 0


class ActionValuedAgent(object):
    def __init__(self, domain, params, random_seed=None):
        self.params = params
        self.domain = domain

        if random_seed is not None:
            numpy.random.seed(random_seed)
```

```
        self.gamma = self.params['gamma'] if 'gamma' in \
                                               self.params else 0.9
        self.alpha = self.params['alpha'] if 'alpha' in \
                                               self.params else 0.1
        self.epsilon = self.params['epsilon'] if \
            'epsilon' in self.params else 0.1
        self.initial_value = self.params['initial_value'] if \
            'initialValue' in self.params else 0
        self.decay = self.params['lambda'] if 'lambda' in \
                                                self.params else 0
        self.UDE_keeper = UDE(self.params['beta_naught']) if \
            'beta_naught' in self.params else UDE(0.1)
        self.Q = {s: {a: self.initial_value for a in
                      domain.get_action_set()} for s in
                  domain.get_state_set()}
        self.trace = {state: {action: 0 for action in
                              domain.get_action_set()} for state in
                      domain.get_state_set()}
        self.delta = 0

    def get_action(self, state):
        if state not in self.Q:
            self.Q[state] = {a: self.initial_value for a in
                             self.domain.get_action_set()}
        if numpy.random.random() < self.epsilon:
            action_set = self.domain.get_action_set()
            action = numpy.random.choice(action_set)
        else:
            action = numpy.random.choice(
                [k for k, v in self.Q[state].iteritems() if
                 v == max(self.Q[state].values())])
        self.last_action = action
        return action


class WatkinsQLearningAgent(ActionValuedAgent):
    def __init__(self, domain, params):
        super(WatkinsQLearningAgent, self).__init__(domain, params)

    def update(self, state, action, reward, state_new, action_new):

        astar = max(self.Q[state_new], key=self.Q[state_new].get)
        if self.Q[state_new][action_new] == self.Q[state_new][astar]:
            astar = action_new

        self.delta = reward + \
                     self.gamma * self.Q[state_new][astar] - \
                     self.Q[state][action]
```

```
        self.trace[state][action] += 1
        self.trace[state][action] = 1 if \
            self.trace[state][action] >= 1 else \
            self.trace[state][action]

        for s in self.trace:
            for a in self.trace[s]:
                self.Q[s][a] += self.alpha * self.delta * \
                                self.trace[s][a]
                if action_new == astar:
                    self.trace[s][a] *= self.gamma * self.decay
                else:
                    self.trace[s][a] = 0

        # update used to compute the curiosity reward.
        self.UDE_keeper.update(self.delta)


class TDLambdaAgent(ActionValuedAgent):
    def __init__(self, domain, params):
        super(TDLambdaAgent, self).__init__(domain, params)

    def update(self, state, action, reward, state_new, action_new):

        self.delta = reward + \
                     self.gamma * self.Q[state_new][action_new] - \
                     self.Q[state][action]
        for s in self.trace:
            for a in self.trace[s]:
                self.trace[s][a] *= self.gamma * self.decay
        self.trace[state][action] += 1
        self.trace[state][action] = 1 if \
            self.trace[state][action] >= 1 else \
            self.trace[state][action]
        for s in self.Q:
            for a in self.Q[s]:
                self.Q[s][a] += self.alpha * self.delta * \
                                self.trace[s][a]

        self.UDE_keeper.update(self.delta)


class MultiBrainedAgent(object):
    def __init__(self, domain, params):
        self.domain = domain
        self.params = params
```

```
        assert 'control' in self.params
        self.control_agent = \
            self.params['control']['agent_type'](domain,
                        self.params['control']['params'])

        assert 'predictor' in self.params
        self.predictor_agent = \
            self.params['predictor']['agent_type'](domain,
                        self.params['predictor']['params'])

        assert 'control_reward' in self.params
        self.control_reward = self.params['control_reward']

    def get_action(self, state):
        return self.control_agent.get_action(state)

    def update(self, state, action, reward, state_new, action_new):
        self.predictor_agent.update(state, action, reward,
                                    state_new, action_new)

        curiosity_reward = self.control_reward(self.predictor_agent)

        self.control_agent.update(state, action, curiosity_reward,
                                  state_new, action_new)


class UDE(object):
    def __init__(self, beta_naught, small_constant=0.0001):
        self.beta_naught = beta_naught
        self.small_constant = small_constant
        self.knower_of_variance = SampleHolder()
        self.tau = 0
        self.learned_avg_delta = 0
        self.beta = None

    def update(self, delta):
        # tau_{t+1}
        self.tau = (1-self.beta_naught)*self.tau + self.beta_naught
        self.beta = self.beta_naught/self.tau

        # learn variance of delta
        self.knower_of_variance.add_variable(delta)
        v = self.knower_of_variance.get_variance()

        # learn exponentially weighted moving average of delta
        self.learned_avg_delta = (1 - self.beta) * \
                                 self.learned_avg_delta + \
                                 self.beta * delta
```

```
    def get_output(self):
        return abs( float(self.learned_avg_delta) /
                    (self.knower_of_variance.get_variance() +
                     self.small_constant))


# https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance
class SampleHolder(object):
    def __init__(self):
        self.K = 0
        self.n = 0
        self.ex = 0
        self.ex2 = 0

    def add_variable(self,x):
        if self.n == 0:
            self.K = x
        self.n += 1
        self.ex += x - self.K
        self.ex2 += (x - self.K) * (x - self.K)

    def remove_variable(self,x):
        self.n -= 1
        self.ex -= (x - self.K)
        self.ex2 -= (x - self.K) * (x - self.K)

    def get_mean(self):
        return self.K + self.ex / self.n

    def get_variance(self):
        if self.n == 0:
            return 0
        if self.n == 1:
            return (self.ex2 - (self.ex*self.ex) / self.n) / self.n
        return (self.ex2-(self.ex * self.ex)/self.n)/(self.n-1)


if __name__ == "__main__":

    init_random_seed = 2017
    num_steps = 20000
    num_replicates = 20

    filename = 'stats.csv'

    epsilons = {-1: 0.01, 0: 0.5, 1: 0.99}
    beta_naughts = {-1: 0.01, 1: 0.99}
```

```
alpha_ps = {-1: 0.01, 0: 0.5, 1: 0.99}
alpha_cs = {-1: 0.01, 0: 0.5, 1: 0.99}
gamma_ps = {-1: 0, 0: 0.49, 1: 0.98}
gamma_cs = {-1: 0, 0: 0.49, 1: 0.98}
lambda_ps = {-1: 0, 0: 0.49, 1: 0.98}
lambda_cs = {-1: 0, 0: 0.49, 1: 0.98}

with open(filename, 'w') as f:
    f.write('Epsilon,Beta_Naught,Alpha_p,Alpha_c,Gamma_p,' +
            'Gamma_c,Lambda_p,Lambda_c,Percent␣Periodic,' +
            'Percent␣Random,Percent␣Constant,Return\n')

code_combos = [(epsiloncode, b0code, apcode, accode,
                gpcode, gccode, lpcode, lccode)
               for epsiloncode in epsilons
               for b0code in beta_naughts
               for apcode in alpha_ps
               for accode in alpha_cs
               for gpcode in gamma_ps
               for gccode in gamma_cs
               for lpcode in lambda_ps
               for lccode in lambda_cs]*num_replicates
numpy.random.seed(init_random_seed)
numpy.random.shuffle(code_combos)

for code in code_combos:
    epsilon = epsilons[code[0]]
    beta_naught = beta_naughts[code[1]]
    alpha_p = alpha_ps[code[2]]
    alpha_c = alpha_cs[code[3]]
    gamma_p = gamma_ps[code[4]]
    gamma_c = gamma_cs[code[5]]
    lambda_p = lambda_ps[code[6]]
    lambda_c = lambda_cs[code[7]]

    # make spreadsheet
    with open(filename, 'a') as f:
        for c in code:
            f.write(str(c) + ',')

    test_domain = CuriosityBandit(random_seed=init_random_seed)
    test_system = \
        MultiBrainedAgent(test_domain,
                          {'predictor':
                               {'agent_type': TDLambdaAgent,
                                'params': {'gamma': gamma_p,
                                           'alpha': alpha_p,
                                           'lambda': lambda_p,
```

```
                                        'beta_naught':
                                            beta_naught,
                                        'initial_value': 0}},
                        'control':
                            {'agent_type':
                                 WatkinsQLearningAgent,
                             'params': {'gamma': gamma_c,
                                        'alpha': alpha_c,
                                        'epsilon': epsilon,
                                        'lambda': lambda_c,
                                        'beta_naught':
                                            beta_naught,
                                        'initial_value': 0}},
                             'control_reward': lambda agent:
                             agent.UDE_keeper.get_output()})
test = Test(test_domain, num_steps)
test.run(test_system, test_domain.get_initial_state())

with open(filename, 'a') as f:
    for action in test_domain.get_action_set():
        f.write(str(float(test.count[0][action])/num_steps)
                + ',')
    f.write(str(test.return_sum) + '\n')
```