

Finding Synonyms in a Regular Language Using Reinforcement Learning*

Ian Johnson[†] and Mark Plumbley
Dept. of Electronic Engineering,
King's College London,
Strand,
London,
WC2R 2LS,
UK.

Technical Report Number 11

Abstract

Reinforcement learning (RL) has been very successful when it has been used to solve control and gaming problems. No application to language processing has come to light which has the same fame as the previous problems. A method is to be presented by which a machine when trained with RL can create synonyms that exist in a regular language. The regular expressions are generated by an operator grammar. The machine produces sentences which are evaluated for numerical equivalence with an input expression. Q-learning[1, 2] is used to modify an agent's policy which is represented by a lookup table. A range of training parameters are found which give perfect performance and training times are compared between two Q-learning algorithms. A lookup table implementation is too limited in terms of using it to handle larger, and more complex grammars. The use of neural networks is discussed including a report on early results using a smaller operator grammar.

*This research was funded by Engineering and Physical Sciences Research Council, UK.

[†]e-mail: ian.johnson@kcl.ac.uk

1 Introduction

Synonyms are two or more words or expressions of the same language that have the same (or nearly the same) meaning in some or all senses. Here we only consider a synonym to be an expression that has exactly the same meaning or outcome as another expression within the same language. For example, in integer arithmetic, a synonym for “ $3 + 3$ ” is “ 3×2 ” as the two expressions result in the same outcome, i.e. 6.

A conversation can contain a number of utterances that communicate a series of intentions[3]. If our intentions are communicated successfully then an interlocutor (someone who takes part in dialogue or conversation) could paraphrase what we had just said, i.e. form a synonym. This is an ability that humans take for granted, but can a machine be trained to discover synonyms that exist in a language?

Lookup tables and neural networks (NNs) have been successfully trained with reinforcement learning (RL) to find solutions to control and gaming problems. Route finding has been a popular task: a goal finding navigation task was solved using a lookup table representation for a simple environment with a barrier[4]. While navigating an agent to a goal in environments with randomly placed obstacles such as those presented in [5, 6] where shown to be solvable. Sensors detect range to an obstacle and angle and distance to the goal. The agent’s policy was good enough to navigate through larger, unseen environments. A RL approach was used to solve a robot reaching task[7]. The neural controller uses schemas which are descriptions of “function decompositions of sensory and motor processes”. And the lowest level schemas can be implemented as pieces of computer program or NNs. Another application of RL is learning to survive within a hostile environment[8]. The environment consisted of food, enemies and obstacles. The agent has to learn to avoid its enemies and obstacles as well as obtaining food. A move of the agent will cost energy, and loss of all energy or capture will result in the end of a trial. The problem of routing packets around a communication networks has been successfully implemented using a modified Q-learning algorithm called Q-routing[9, 10]. Lookup tables at each node in a network are trained to minimise the time it takes to deliver a packet to its destination. The choice of neighbouring node to send the packet to is based upon local estimates of the time remaining to its destination. Q-routing outperforms a *shortest path* router at high network loads in an irregular network containing 36 nodes. Under low load Q-routing, after a brief period of training, performs as well the shortest path method.

Only two of the gaming applications will be discussed here: Backgammon and Go. Tesauro’s TDGammon[11] learnt to play Backgammon not only

to expert level, but also changed the way human players open the game. TD(λ)[12] was employed to find an evaluation of the board state. It is also interesting that Backgammon is a non-deterministic game as there are dice involved. In contrast Go is a deterministic, strategy game for two players. It also has a large branching factor (approximately 200) that make tree search approaches, that are for instance used in chess, infeasible[13]. Here a network with less than 500 weights learnt Go board evaluations within 3000 games, and was able to beat a commercial Go program at low playing level.

RL is capable producing complex and robust behaviour even with partially observable, non-deterministic, control and game environments. However, no application to language processing has come to light which has the same fame as the examples above (see Sutton and Barto's recent introductory book[14]).

The aim of this report is to propose and test a method whereby RL can be used to solve a language processing task: finding synonyms in a regular language. As no previous work exists for this type of task using RL a simple lookup table implementation is used to test the feasibility of the method, and to highlight any possible problems that may occur in future work.

The remainder of the report takes the following form: section 2 introduces the grammar used and the motivation for using RL to try and solve the task in hand. Section 2 also contains a brief introduction to RL. The agent type used in this report is described in section 3 and how the simulations were conducted in section 4. The results from the simulations carried out are presented and discussed in section 5 and in section 6 conclusions are drawn. Finally, early results from work carried after this implementation are reported in section 7.

2 Reinforcement Learning for Synonym Finding

The grammar used to generate regular expressions for which synonyms must be found is shown in figure 1. This is a simple operator grammar that

$$\begin{aligned} S &\rightarrow T + T \mid T - T \mid T \times T \mid T \div T \\ T &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

Figure 1: The grammar used to generate the regular language.

creates 400 strings such as “ 3×2 ” and “ $9 \div 6$ ”. As will be shown later the

expressions need to be evaluated so ten are omitted to overcome divide-by-zero errors which would occur on expressions like “ $5 \div 0$ ”. Integer arithmetic with no remainder is used to evaluate these expressions, e.g. “ $2 \div 6 = 0$ ” and “ $3 \div 2 = 1$ ”.

Given the regular language (defined in figure 1) we wish a machine to learn synonyms for expressions that the grammar generates. If we use a supervised learning method, such as backpropagation[15] or a variant as used by Allen and Seidenberg[16] to allow a connectionist network to make grammaticality judgements, then we would require both the input sentence and the synonym. Sentences can have many synonyms, e.g. in the regular language used here the sentence “ 3×2 ” could be written as “ $3 + 3$ ”, “ $12 \div 2$ ” or “ $120 \div 10 - 6$ ”. Supervised learning requires one input to be associated with one output, so which synonym should be chosen? If some criteria was to be used to choose a synonym, e.g. length of synonym must be under 5 symbols long, then this set maybe large.

Reinforcement learning (RL) relieves us of the problem of having to choose a training pair for supervised learning. RL trains an agent through trial-and-error, i.e. in this instance experimenting with candidate expressions (expressions which may or may not be synonyms). Once a candidate expression has been chosen and compared with an input expression a scalar reward can be used to adapt the agent’s behaviour. Its behaviour could then, over time, produce candidate expressions that become closer to a synonym. A short introduction to reinforcement learning follows. Readers who are familiar with reinforcement learning may proceed to section 3.

2.1 Reinforcement Learning

Reinforcement learning is a class of problems where a learning agent must act in situations to maximise a scalar reward signal[14]. The learner is not told which actions to take by an external expert, as in supervised learning. Instead the learner has to find actions which give the largest the reward by trial-and-error. The agent acts within an environment to complete some goal: this must have some relation to a state within the environment. For the agent to act within the environment it has to be able to perceive the state and execute actions which change the state.

The agent is to learn to make decisions in environmental states. Anything outside of the agent is the environment. The agent acts on the environment over a period of discrete time steps, $t = 0, 1, 2, 3, \dots$. At each time step the agent senses the environment state, $s_t \in S$, where S is the set of all environmental states. From this sensation an action, $a_t \in A(s_t)$, has to be decided upon. Where $A(s_t)$ is the set all possible actions that the agent can

perform in the current state, s_t . Irrespective of the agent's past history it makes a transition to a new state, s_{t+1} , and receives, from the environment, an immediate reward, $r_{t+1} \in \mathfrak{R}$. This interaction is shown in figure 2.1.

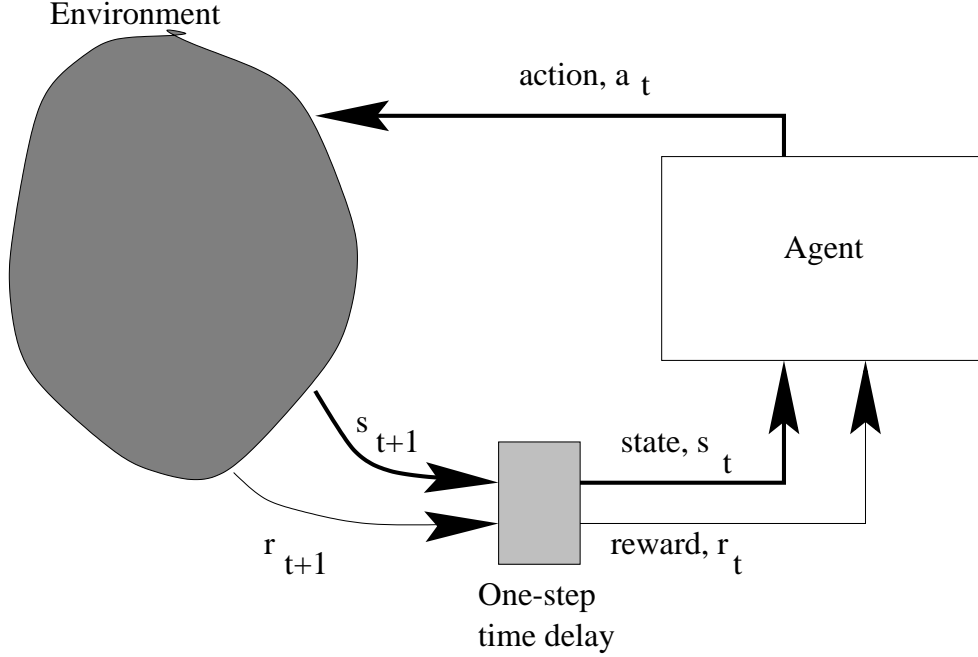


Figure 2: Reinforcement learning's agent/environment interaction.

The next state, say y , is only dependant upon the current state, say x , and current action, a . Or, the state transition probability is,

$$\begin{aligned} P_{s_t, s_{t+1}}(a_t) &= Prob\{ s_{t+1} = y \mid s_t = x, a_t = a \} \\ &= Prob\{ s_{t+1} = y \mid s_0, a_0, s_1, a_1, s_2, a_2, \dots, s_t = x, a_t = a \} \end{aligned}$$

and the expected value of the reward is,

$$R(s_t, a_t) = E\{ r_{t+1} \mid s_t, a_t \}$$

Any environment which posses these dynamics is a finite Markov decision process (MDP), as the state and action spaces are finite[14].

With each tick the agent carries out a mapping from environmental state to action. This is the agent's *policy*, π_t , i.e. a method of acting in the light of a stimulus. The reward, r_{t+1} , informs the agent how good the state transition s_t to s_{t+1} was; it also defines the goal in the problem. Example: Consider a gridworld with a barrier shown in Figure 2.1 (as presented in [4]).

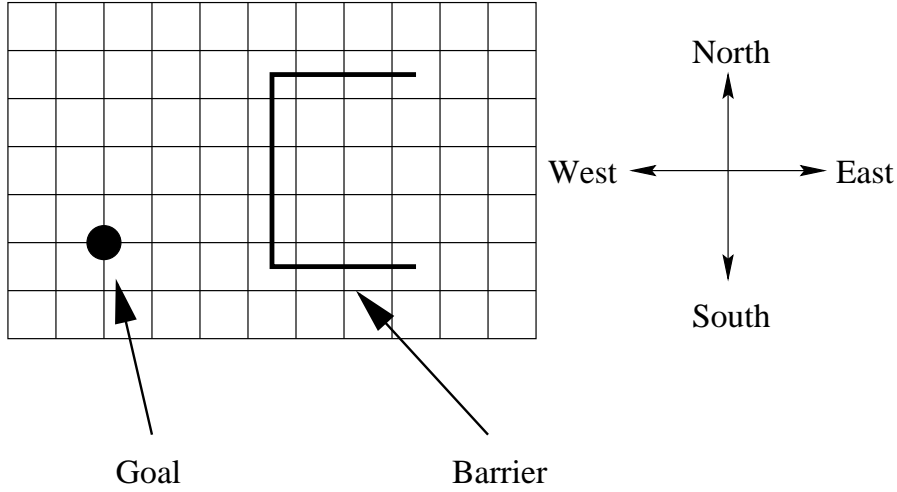


Figure 3: Gridworld: intersections are possible locations or states. A goal state and barrier are shown.

In the gridworld an environmental state is any intersection of grid lines, the goal state is shown as a black spot, and the C-shaped barrier as a thicker black line. The learning agent can only move between states if there is a connecting line. The permissible actions are north (N), east (E), south (S) and west (W), i.e. $A(s) = \{N, E, S, W\}$. The agent has to find its way to the goal state, s_{goal} . The reward for moving to the goal state is $R(s_{goal}, a) = 0$, and for moving into any other state $R(x, a) = -1$. Trying to move through the barrier will result in an immediate reward of -1 and no change of state. The goal state has more of a reward than any other state transition defining this as a goal state.

For the agent to be able to find an acceptable goal finding policy it must estimate a *value function*.

2.2 Value Functions

A value function is a prediction of how *good* it is to be in a particular state, i.e. how good it is to take an action from a state. The *goodness* of a state while following policy π is $V^\pi(s)$, and can be defined in terms of expected future discounted rewards, or

$$V^\pi(s) = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right\}, \quad (1)$$

whilst the goodness of taking an action in a state following the policy π is,

$$Q^\pi(s, a) = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right\} \quad (2)$$

where γ is the *discount rate* ($0 \leq \gamma \leq 1$). This weights rewards seen sooner in time more than ones seen further ahead. It controls the influence of the long-term consequences of actions in a problem and effects the rate of learning. If $\gamma = 0$ we have a *myopic* agent which only sees the immediate reward. Whereas, if $\gamma = 1$ then all rewards are weighted equally and in general is not finite. However, for a finite horizon problem and $\gamma = 1$ the value of $V(s)$ or $Q(s, a)$ is finite as all rewards after a certain time are zero.

In the gridworld example from a starting state x , after n time steps, and using a stationary policy π , then $V^\pi(x) = -1 - \gamma - \gamma^2 - \dots - \gamma^{n-1}$. If $\gamma = 1$ then the value of a given state becomes the negative number of time steps that the agent will take to get to s_{goal} , the goal state. And as $\gamma \rightarrow 0$ then $V^\pi(x) \rightarrow -1$ as the later terms in the series become more and more negligible. A typical value function for this problem is shown in Figure 4. Note that $V^\pi(s_{goal}) = 0$ (the lowest point of the surface in figure 4) for *any* policy as the goal state is absorbing; meaning that the state transition probability $P_{s_{goal}, s_{goal}}(a_t) = 1 \forall a_t \in A(s_t)$. This transition yields a reward of zero.

2.3 Policy Improvement

It is the job of the agent to find a policy which will maximise the value function. If we have a value function, V^π , for some policy π in which state, s , should we change the policy? A way to do this is to, in state s , choose an action a and then follow the original policy, π , from then on. This behaviour has a value of

$$Q^\pi(s, a) = E_\pi \{ r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s, a_t = a \} . \quad (3)$$

If the action a is a better action than the one perscribed by π then $Q^\pi(s, \pi'(s)) \geq V^\pi(s)$. The new policy is then better than the previous one, this is called *policy improvement*. The policy π' must be as good, or better than π , i.e.

$$V^{\pi'}(s) \geq V^\pi(s) . \quad (4)$$

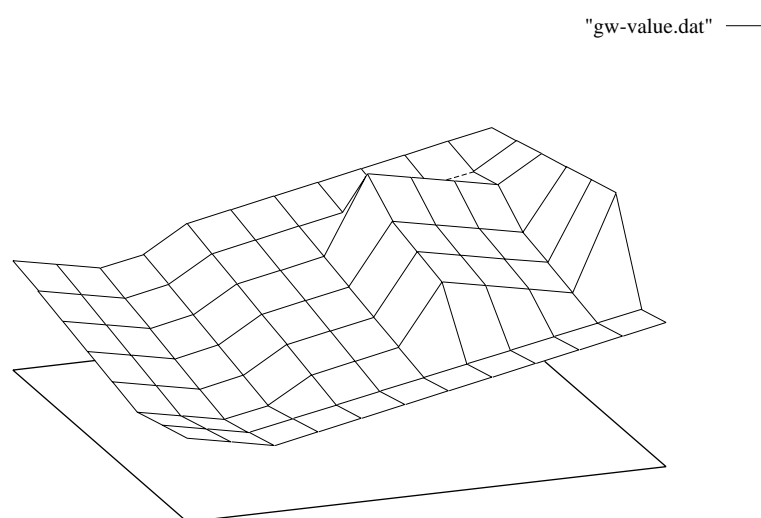


Figure 4: Value Function for Gridworld. This shows the negative of the value function for a stationary policy, which is the number of time steps needed to arrive at the goal state. An intersection on the surface is an environmental state.

2.4 Q-learning

With Q-learning[1, 2] an action value is learnt, $Q^\pi(s, a)$. If it is learned correctly then the value function is,

$$V(s) = \max_{a_t \in A} Q(s, a_t) \quad (5)$$

The predicted action value can be updated with,

$$Q(s, a_t) = Q(s, a_t) + \eta[r_t + \gamma V(s_{t+1}) - Q(s, a_t)] \quad (6)$$

So, substituting Equation 5 we arrive at,

$$Q(s, a_t) = Q(s, a_t) + \eta \left[r_t + \gamma \max_{a_t \in A} Q(s_{t+1}, a_t) - Q(s, a_t) \right] \quad (7)$$

or the *one-step* Q-learning update function[1, 2].

For a lookup table implementation Q-learning will converge with probability 1 if the number of trials, k , is infinite, learning rate $0 \leq \eta < 1$, and the stochastic approximation convergence conditions are met[2]. These are,

$$\sum_{k=1}^{\infty} \eta_k = \infty, \quad \sum_{k=1}^{\infty} \eta_k^2 < \infty \quad (8)$$

Equation (8) ensures that we actually change the Q-values so to overcome the initial values, and that the learning rate will decrease sufficiently so that the Q-values can converge respectively.

2.5 Experience Replay

Using only the one-step Q-update function, equation (7), a *delayed reward*[12, 1, 17] (see section 3 for how the delayed reward occurs in this application) may never propagate back through an action sequence which lead to that reward, and Q-function value updates may not allow this action sequence to occur again. This leads to earlier actions never “seeing” the reward given to the whole sequence. A longer convergence time can result as a consequence. To alleviate this potential problem *experience replay*[8] (ER) can be used. To implement ER one must store the experiences of the agent. An *experience* is a quadruple $\langle s_t, a_t, r_{t+1}, s_{t+1} \rangle$ of current state s_t , action a_t chosen in the current state, reward r_{t+1} , and next state s_{t+1} respectively. The experiences of the agent can be presented again and again but can be used more effectively if they are presented temporally backwards; this allows the delayed reward to propagate back through the action sequence in one presentation. This method is used here and can lead to faster convergence of a policy. The Q-function values are updated with equation (7).

3 The Agent

The agent is implemented as a lookup table whose input is a 5-tuple $\langle q_i, n_{i1}, n_{i2}, q_o, n_{o1} \rangle$. Where $q_i \in \{+, -, \times, \div\}$ is the input operator, $n_{i1} \in \{0, 1, 2, \dots, 9\}$ is the first number at the input, $n_{i2} \in \{0, 1, 2, \dots, 9\}$ is the second number in the input expression, $q_o \in \{+, -, \times, \div\}$ is the output operator chosen, and $n_{o1} \in \{0, 1, 2, \dots, 9\}$ is the first number chosen by the agent. A starting state in the environment is any input where the inputs q_o and n_{o1} are Not-an-Operator (NaO) and Not-a-Number (NaN) respectively. The state arrived at when the second number, $n_{o2} \in \{0, 1, 2, \dots, 9\}$, is chosen denotes the end of a trial and the next starting state ($\langle q_i, n_{i1}, n_{i2}, NaO, NaN \rangle$), or input expression, is presented. Therefore no need for a sixth input field, n_{o2} , second number output.

The agent adapts its policy using Q-learning which was described in section 2.4. At each location in the lookup table ten Q-function values are stored which represent different actions at different times. The first action to be decided upon is which operator to use: only four Q-function values are used to identify these operators and the remaining six are labeled as NaO. The following two time steps are interpreted as numbers. The agent must learn to output a valid operator then two numbers.

At each time step an action is chosen, on the following time step this action is feedback to its respective element at the input. Following the state trajectories ABCD and AIEFG in figure 5 the feedback can be seen; e.g. the output of state B, \times , is feedback to the q_o element of the input. This feedback is necessary to allow the agent to perceive a change of position in its environment. The suitability of the synonym can only be evaluated when the complete string has been output. Therefore, the performance measure can only be given on completion of a candidate expression: this is a delayed reward.

3.1 Exploration Strategy

A simple, and random exploration technique is used by the agent and is called ϵ -greedy[14]. This is an exploration technique that selects random actions with a probability, ϵ , otherwise an action is chosen which maximises $Q(s_t, a_t) = \max_{a \in A(s_t)} Q(s_t, a)$, where $A(s_t)$ is the set of available actions in the current state, s_t . The amount of exploration is diminished during training so the agent policy becomes increasingly deterministic.

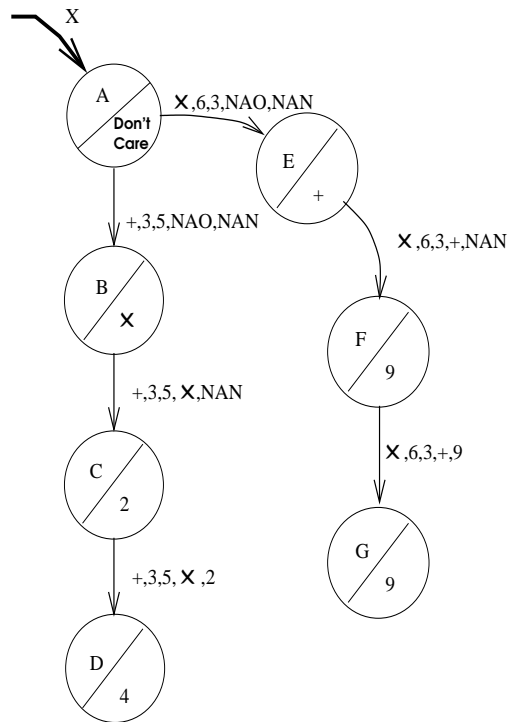


Figure 5: Agent's possible internal state transition diagram for two input strings.

4 Experimental Details

A range of training parameters which would produce a perfectly performing agent was to be found. The discount factor, γ , was set to 1.0 as we were assured that the sum of the discounted rewards would be finite. This leaves the learning rate, η , the initial amount of exploration, ϵ_0 , and the rate of decay in the exploration to be altered. Contour and surface plots were constructed which show how the final performance was effected with $\eta = 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0$ and $\epsilon_0 = 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0$. ϵ was linearly decreased to zero over 1, 50, 250, and 750 of the 2000 epochs, or presentations of all 390 training inputs, available for training.

Agents were trained using the standard one-step Q-learning update, and the standard one-step Q-learning update with experience replay. All agents' Q-function values were initially set to the same random values in the range $[-1, 1]$. This effectively enforces a little exploration to take place in the early stages of learning as some actions will have greater Q-function values than others.

When a goal state has been reached a reward signal can be formulated, any other state transition yields a reward of 0. The reward for a lookup table agent, $r_{candidate}$, is,

$$r_{candidate} = \begin{cases} -100.0 & \text{if NaO, or candidate expression has the form } x \div 0, \\ -|\Delta_{lu}| & \text{otherwise.} \end{cases}$$

Where Δ_{lu} is,

$$\Delta_{lu} = (n_{i1} q_i n_{i2}) - (n_{o1} q_o n_{o2})$$

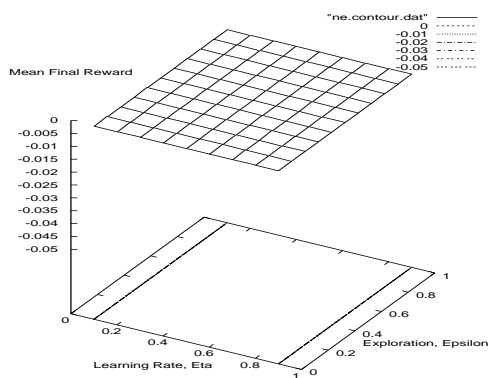
Where $q_i \in \{+, -, \times, \div\}$ and is the input operator, $n_{i1}, n_{i2} \in \{0, 1, 2, \dots, 9\}$, $q_o \in \{+, -, \times, \div\}$, and $n_{o1}, n_{o2} \in \{0, 1, 2, \dots, 9\}$. As the agent becomes better at producing numerically equal expressions $r_{goal} \rightarrow 0$.

5 Results and Discussion

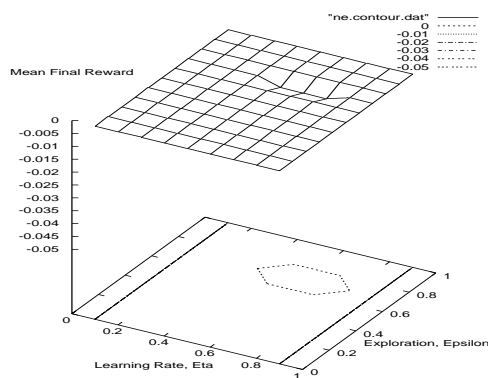
Contour and surface plots show how the three variable training parameters effect the *mean final reward* (MFR) presented to the agent. The mean reward awarded to the agent in the last epoch or,

$$\text{Mean Final Reward (MFR)} = \frac{\sum_{n=1}^N r_{candidate}^n}{N},$$

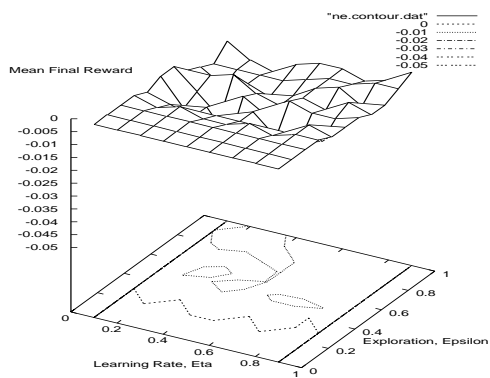
where $r_{candidate}^n$ is the delayed reward for an action sequence (candidate expression) starting from the n^{th} starting state, and N is the number of starting states or possible input expressions, in this case 390. Therefore, a perfectly performing agent will yield $MFR = 1$. Figures 6 and 7 show performance for agents trained with the standard Q-update and with experience replay respectively.



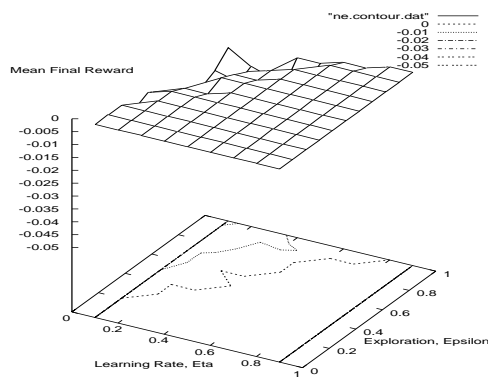
(a) ϵ decay: 1 epoch.



(b) ϵ decay: 50 epochs.

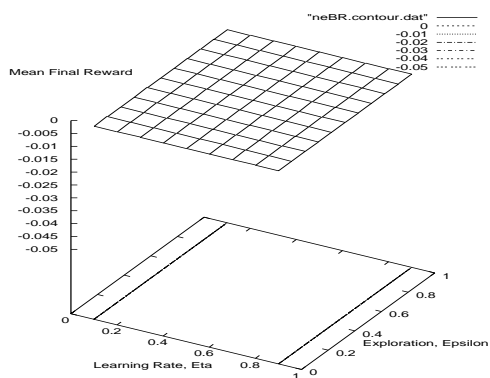


(c) ϵ decay: 250 epochs.

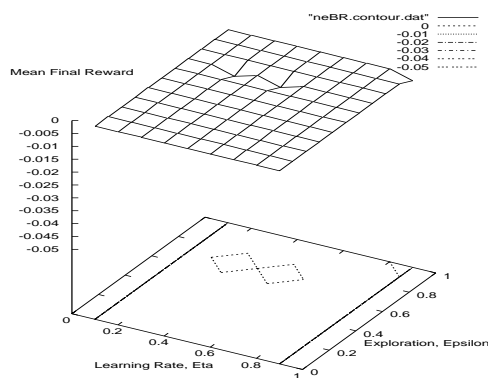


(d) ϵ decay: 750 epochs.

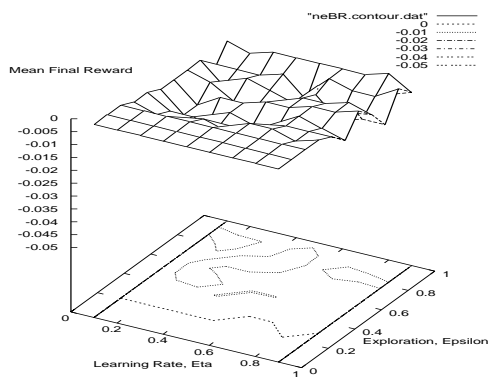
Figure 6: One-step Q-update.



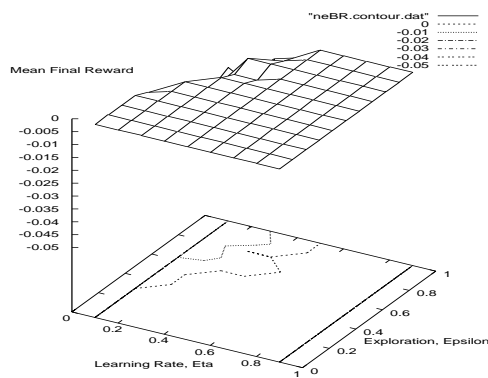
(a) ϵ decay: 1 epoch.



(b) ϵ decay: 50 epochs.



(c) ϵ decay: 250 epochs.



(d) ϵ decay: 750 epochs.

Figure 7: One-step Q-update with Experience Replay.

The plots demonstrate that it is possible for an agent trained with Q-learning to successfully produce synonyms which exist in the regular language used. With a learning rate of zero the Q-function values are never updated and therefore the agent’s performance is as if no training had taken place. A learning rate of 1 contravenes the range of η that will ensure convergence and as a result the agent’s performance is worse after training than if the Q-function values had been randomised. The most notable of the training parameters to effect the final performance is the rate of decrease in exploration. A decay within 250 epochs constricts the range of the initial exploration probability, ϵ_0 , the most (see figures 6(c) and 7(c)). Longer exploration time can increase this range (see figures 6(d) and 7(d)), but the little exploration enforced on the agent due to the randomisation of Q-function values is all that is needed.

Table 1 shows the number of epochs needed to give perfect performance while taking greedy actions during training.

Learning Rate, η	Standard Q-Update	Standard Q-Update with Experience Replay
0.2	1458	1409
0.4	823	760
0.6	617	570
0.8	506	460
0.999	448	400

Table 1: Training times for agents taking greedy actions during training.

Using a large learning rate yields quicker convergence and using experience replay gives approximately a 10% decrease overall on training time from the one-step Q-update without experience replay.

The lookup table implementation is very limited as it cannot exhibit any generalisation, and becomes large when large input spaces or long action sequences are required. The restricted action sequence length will only allow the agent to produce three actions. Therefore any deleterious actions early in the sequence cannot be remedied. For example, we may wish to find a synonym for the input expression “5 + 5”. The agent in the first three time steps outputs “7 × 2”. This action sequence could be corrected by the addition of two further actions, namely “−4”. To be able to find synonyms in more complex languages we require the very attributes which limit this model. The larger number of tokens in a language can mean a larger number

of inputs, and complex languages can produce longer sentences.

6 Conclusion

We presented a method whereby a lookup table can be trained to find synonyms that exist in a regular language using Q-learning. This method was tested on an operator language. Candidate expressions are formed by a machine alleviating the need to produce a training pair needed by supervised learning. The reinforcement signal, or reward, is a measure of how similar the outcomes of the input and the candidate expressions are.

There exists a range of training parameters which gives perfect performance for all the exploration decay rates used. Executing greedy actions during training produces the widest range of training parameters. Both training algorithms show that little or no user enforced exploration is needed to solve this synonym finding task.

Faster training can be achieved using a large learning rate, but less than 1, and experience replay but is only marginally better than the one-step Q-update without experience replay for this short action sequence.

Finally, the limitations of the model were discussed with a view to its application to more complex languages.

7 Future Work

The limitations of the lookup table are to be overcome with the use of feedforward neural networks which can approximate a Q-function. Neural networks are able to generalise and as such an exhaustive presentation of all sentences in a language is unnecessary. Using a smaller grammar than the one shown in figure 1 (regular expressions generated are $(0|1|2|3)(+|-|\times|\div)(0|1|2|3)$), work is in progress to emulate the promising results presented here using agents built from four neural networks. Each agent has its own area of expertise as they are only exposed to expressions with one operator. This is an attempt to isolate an agent from any conflicting rewards which may accrue from processing expressions with differing operators. Early results using the ϵ -greedy exploration strategy and an on-line connectionist Q-learning algorithm, modified connectionist Q-learning[6, 5] (as known as SARSA(λ)[14]), show a reluctance to solve this task frequently and reliably. These results are surprising since the lookup table agents can solve this task quickly and with little exploration for a larger grammar.

References

- [1] C.J.C.H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge University, UK, 1989.
- [2] C.J.C.H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.
- [3] W.J.M. Levelt. *Speaking: From Intention to Articulation*. MIT Press, 1995.
- [4] A.G. Barto, R.S. Sutton and C.J.C.H. Watkins. Learning and sequential decision making. Technical Report UM-CS-1989-095, University of Massachusetts, Department of Computer Science, September 1989.
- [5] G.A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University, 1994.
- [6] G.A. Rummery. *Problem solving with reinforcement learning*. PhD thesis, Cambridge University Engineering Department, 1995.
- [7] A.H. Fagg. *Reinforcement Learning for Robotic Reaching and Grasping*, chapter 14, pages 281–308. North Holland Press, 1993.
- [8] L. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.
- [9] J.A. Boyan and M.L. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. In Jack D. Cowan, Gerald Tesauro, and Joshua Alspecter, editor, *Advances in Neural Information Processing Systems 6*, pages 671–678. Morgan Kaufmann, San Francisco CA, 1994.
- [10] M.L. Littman and J.A. Boyan. A distributed reinforcement learning scheme for network routing. In Joshua Alspecter, Rodney Goodman, and Timothy X. Brown, editor, *Proceedings of the 1993 International Workshop on Applications of Neural Networks to Telecommunications*, pages 45–51. Lawrence Erlbaum Associates, Hillsdale NJ, 1993.
- [11] G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8:257–277, 1992.
- [12] R.S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

- [13] N.N. Schraudolph, P. Dayan, and T.J. Sejnowski. Temporal Difference Learning of Position Evaluation in the Game of Go. In David Touretzky, editor, *Advances in Neural Information Processing Systems 6*, July 1994.
- [14] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
- [15] D.E. Rumelhart, G.E. Hinton and R.J. Williams. Learning representations by back-propagation. *Nature*, 326(6088), 1986.
- [16] J. Allen and M.S. Seidenberg. *The Emergence of Language*, chapter “The Emergence of Grammaticality in Connectionist Networks”. Lawrence Erlbaum Associates, Inc., 1999.
- [17] S.P. Singh and R.S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1-3):123–158, 1996.