

On-line Connectionist Q-learning Produces Unreliable Performance With A Synonym Finding Task.

Ian Johnson and Mark Plumbley

Dept. of Electronic Engineering,
King's College London,
Strand,
London,
WC2R 2LS,
UK.
e-mail: ian.johnson@kcl.ac.uk

Abstract

Neural networks (NNs) trained with reinforcement learning (RL) have the ability to produce complex, and robust behaviour which may be beneficial to language processing tasks. A method is proposed using RL to train NNs so that they might find synonyms that exist within a regular language. The learning algorithm and exploration strategy produces agents which yield consistently sub-optimal policies for expressions containing one operator, and unreliable performance over all expressions. This is surprising since previous work with lookup tables produced synonyms using a larger set of expressions for a wide range of learning rates and very little exploration[2].

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. In this paper 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[1]. 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 neural network (NN) be trained to produce synonyms that exist in a language?

The aim of the paper is to attempt to emulate the success of lookup table agents to produce synonyms in previous work[2]. These agents were successfully trained with Q-learning[3] to produce synonyms for an operator grammar. Perfect behaviour was attained within 2000 presentations of all training data with little exploration. Agents here use neural networks to attempt to find synonyms for simple regular expressions using a connectionist reinforcement learning algorithm based on Q-learning[3]. The language used is an operator language which produces a smaller number of regular expressions, “ $(0|1|2|3)(+|-|\times|\div)(0|1|2|3)$ ”, than in the previous work[2]. This new grammar produces 64 sentences such as “ $0 - 1$ ”, “ 2×3 ” and “ $3 \div 1$ ”. However, these sentences need to be evaluated so only 60 are used to prevent divide-by-zero errors with sentences like “ $2 \div 0$ ”. The expressions are evaluated using integer operations with *no* remainder, e.g. $3 \div 2 = 1$ and $2 \div 3 = 0$.

2 Reinforcement Learning for Synonym Finding

A sentence can have many synonyms, e.g. “ 3×2 ” could be written as “ $3 + 3$ ”, “ $12 \div 2$ ” or “ $120 \div 10 - 6$ ”. If we are to use a supervised learning method, such as back-propagation[4] or a variant as used by Allen and Seidenberg[5] to allow a connectionist network to make grammaticality judgements, then we would require both the input sentence and the synonym. Which synonym should be chosen? If some criteria is used, e.g. choose shortest synonym, then this set of synonyms maybe large. Reinforcement learning (RL) can be used to find the synonyms by choosing actions, or operators and numbers, to form a candidate sentence that can be compared easily with the input sentence. The comparison can only be made after the candidate sentence has been formed and hence a delayed reward is formulated. This reward for the whole action sequence, or candidate sentence, is used to adapt the policy of a learner to produce candidate sentences which could become, over time, closer to a synonym. Using RL, therefore, negates the need for picking by hand a synonym for each sentence in the language and allows a learner to discover synonyms.

The algorithm used in this paper is modified connectionist Q-learning (MCQ-L)[6, 7] (also known as Sarsa(λ)[8]). MCQ-L has been shown to outperform other connectionist Q-learning algorithms in terms of updates needed, trial lengths and successful agents in a 2D environment where a robot must find a goal while avoiding obstacles[6, 7].

A NN can be used to approximate the Q-values, $Q(s_t, a_t)$, that represent a discounted sum of all future rewards which are to be expected while taking an action, a_t , from the current state[8], s_t , or

$$Q(s_t, a_t) = E \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \right\}, \quad (1)$$

where γ is the discount factor ($0 \leq \gamma \leq 1$) and weights rewards seen sooner in time more than ones further in the future, and r_{t+k+1} is the immediate reward given to an agent at time k .

The MCQ-L update rule[6, 7] for a NN, that approximates the Q-values defined in equation (1), with a changeable weight vector, \mathbf{w}_t , is

$$\Delta \mathbf{w}_t = \eta \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] e_t, \quad (2)$$

where η is the learning rate, r_{t+1} is the immediate reward given to an agent for the state transition s_t to the next state, s_{t+1} , due to a chosen action, a_t . Each weight in the NN maintains an eligibility trace,

$$e_t = \frac{\partial Q(s_t, a_t)}{\partial \mathbf{w}_t} + \lambda \gamma e_{t-1},$$

that determines how eligible a weight is for change. The eligibility trace can be updated on-line as it is defined in terms of its previous value and the partial derivative $\frac{\partial Q(s_t, a_t)}{\partial \mathbf{w}_t}$ that can be evaluated using back-propagation[9]. It is able to distribute the delayed reward quicker through an action sequence and can speed up training. Both the discount factor, γ , and λ ($0 \leq \lambda \leq 1$) have effects upon training time.

The MCQ-L update rule, equation (2), is classed as an *on-policy* update as the Q-value at time $t + 1$, $Q(s_{t+1}, a_{t+1})$, is determined by the policy[8]. This gets rid of the problem of overestimating $\max_{a \in A} Q(s_{t+1}, a)$ in conventional Q-learning using a function approximator[10]. However, equation (2) will convert to the conventional Q-learning update[3] rule once the agent exploits its policy in the later stages of training, i.e. taking greedy actions: $Q(s_{t+1}, a_{t+1}) = \max_{a \in A} Q(s_{t+1}, a)$.

3 Agent Architecture

The synonym finding system contains four agents that are responsible for sentences containing one of the possible input operators. The input space partitioning prevents any conflicting rewards reaching an agent that can increase training times. The input to an agent is then just the numbers either side of the operator in the sentence. Once the input has been received the agent must then form a candidate sentence using the following actions: operators, $A_{ops} = \{+, -, \times, \div\}$, and integer numbers, $A_{ne} = \{0, 1, 2, 3\}$. The agents are

built from 3-layer feedforward NNs (i.e. 2-layers of weights) with $\tanh(\cdot)$ activation functions whose layer sizes are $12 \rightarrow 12 \rightarrow 1$ from input to output.

An agent contains four NNs, one network for each possible action or Q-value, whose input is a four-tuple $\langle n_{i1}, n_{i2}, q_o, n_{o1} \rangle$; where n_{i1} and n_{i2} are the first and second numbers respectively and are both $\lceil \log_2 |A_{ne}| \rceil = 2$ units wide. q_o is the operator output from the agent and is $|A_{oops}| + 1 = 5$ units wide: one extra unit is needed for a not-an-operator (NaO) code. n_{o1} is the first number output and is $\lceil \log_2 (|A_{ne}| + 1) \rceil = 3$ units wide to allow the representation of a not-a-number (NaN) code. All input layer banks take M-in-N binary codings, so $0_{10} = 00_2$ and $1_{10} = 01_2$ and so on (the NaN code in the case is $4_{10} = 100_2$), apart from q_o which takes an 1-in-N binary coding, e.g. the ‘ \times ’ operator is coded as 00100_2 , and the NaO code takes 00001_2 .

From each starting state, any input where the elements q_o and n_{o1} are NaO and NaN respectively, the agent has three time-steps in which to form a candidate sentence. The output of the agent is interpreted differently depending upon the current time-step. In the first time-step, $t = 0$, an operator is output, and the following two, $t = 1$ and $t = 2$, liberate numbers. As the outputs are formed they are fed back to the appropriate banks of input units. This enables the agent to perceive a change of state in its environment.

An agent’s exploration strategy is called ϵ -greedy[8] and chooses random actions with a given probability, p , otherwise the agent chooses an action which maximises $Q(s_t, a_t)$.

4 Experimental Details

All 60 input sentences from the regular language are used to train the agents and therefore the test set is the training set, i.e. no generalisation. It will be shown that this is a difficult task to solve and as such we are interested in whether the agents can solve the problem and not their generalisation performance.

Once all the neural network’s weights are randomly initialised so they fall in the range $[-1, 1]$ they are trained with MCQ-L for 15000 epochs¹ with a learning rate, $\eta = 0.1$, and the initial exploration measure, $p_0 = 0.4$, that is linearly decayed to 0 over 14000 epochs. The values of discount, γ , and λ are altered and take the values 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0. This creates 121 simulations for each agent within the synonym finding system that returns the performance measure used to assess an agent: *mean final reward* (MFR). The MFR is the mean reward given in the final epoch of training, or

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

where $r_{candidate}^n$ is the delayed reward for a candidate sentence from the n^{th} starting state in an operator partition, and N is the number of starting states or possible input sentences with a partition. A perfectly performing agent yields $MFR = 1$. The ‘+’, ‘-’ and ‘ \times ’ partitions contain 16 starting states, and the ‘ \div ’ partition contains 12. These simulations produce agents where perfect performance is rare: only 14, 0, 15, and 57 successful agents out of 121 agents trained within the +, -, \times and \div partitions respectively. Using less exploration, as is favoured by the lookup table agents[2], yields lower performance than the simulations which generate the results here.

Smaller ranges of discount, γ , and λ were chosen² to investigate the reliability of these results. Three different sets of initial weight configurations are used to initialise each agent. The agents are then trained noting the MFR for each simulation.

The reward for a candidate sentence is,

$$r_{candidate} = \begin{cases} 1.0 & \text{if } \Delta_{nn} = 0, \\ -1.0 & \text{if candidate sentence is of the form } x \div 0, \\ -|\Delta_{nn}| & \text{otherwise.} \end{cases}$$

Where

$$\Delta_{nn} = \frac{(n_{i1} q_i n_{i2}) - (n_{o1} q_o n_{o2})}{16},$$

$q_i \in \{+, -, \times, \div\}$ is the input operator, $n_{i1}, n_{i2} \in \{0, 1, 2, 3\}$, $q_o \in A_{oops}$, $n_{o1} \in A_{ne}$, and the second output number $n_{o2} \in A_{ne}$. Δ_{nn} falls in the range $(-1, 0]$ and as the agent produces better candidate sentences $\Delta_{nn} \rightarrow 0$. Any other rewards given to an agent are 0.

¹One epoch is one presentation of all 60 training data.

²This is due to the CPU time needed to generate results using the initial range of γ and λ .

5 Results

The graphs in figures 1, 2, 3 and 4 show the median MFR for the three neural network initial configurations trained for each operator partition. Errorbars, in these figures, show the highest and lowest MFR attained.

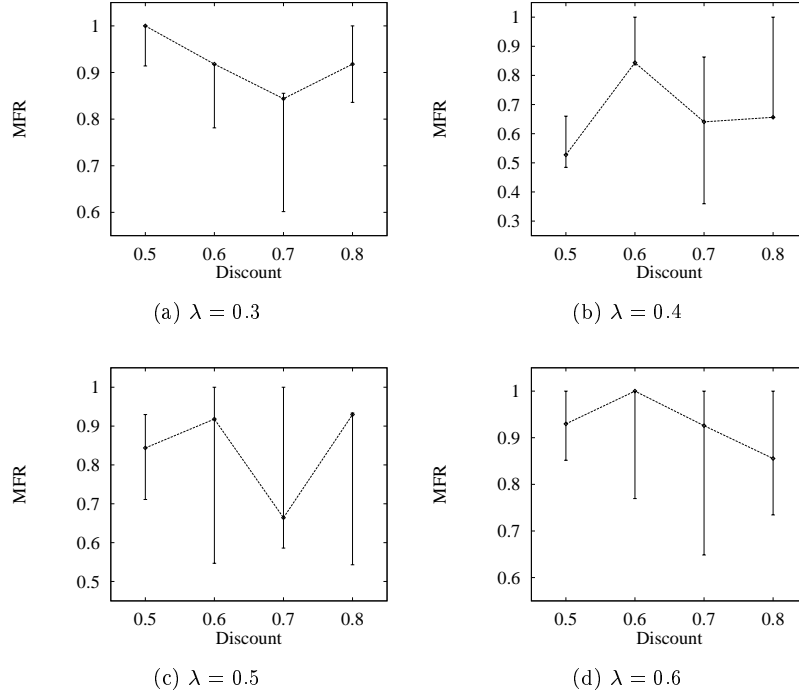


Figure 1: ‘+’ Partition.

The agents, within their partitions, produce unreliable policies for the training parameters chosen with this synonym finding task using a regular language. Their performance, however, forms a hierarchy over the partitions: ‘÷’, ‘+’ and ‘×’, and ‘−’ partition agents from best to worst performance. The ‘−’ partition agents cannot find synonyms for all their input sentences which was also the case when the larger range of γ and λ were used. This is due, in part, to the exploration strategy used: this partition contains sentences whose synonyms are in a minority of the possible output sentences that can be produced by an agent. The random choice of actions makes the finding of a synonym less likely compared to an agent within the ‘÷’ partition. In the ‘÷’ partition 50% of all input sentences equate to 0 and 28% of all possible output sentences also equate to 0. The agents in the ‘+’ and ‘×’ partitions process input sentences whose frequency of synonyms lie in between those of the ‘−’ and ‘÷’ partitions and fits very well with the hierarchy of agent performance.

6 Conclusion

In this paper we have presented a method for learning synonyms for regular expressions using modified connectionist Q-learning (MCQ-L). This was tested on a small arithmetic grammar. In some cases synonyms were found for all input sentences but performance in general is unpredictable which is possibly due to the exploration strategy.

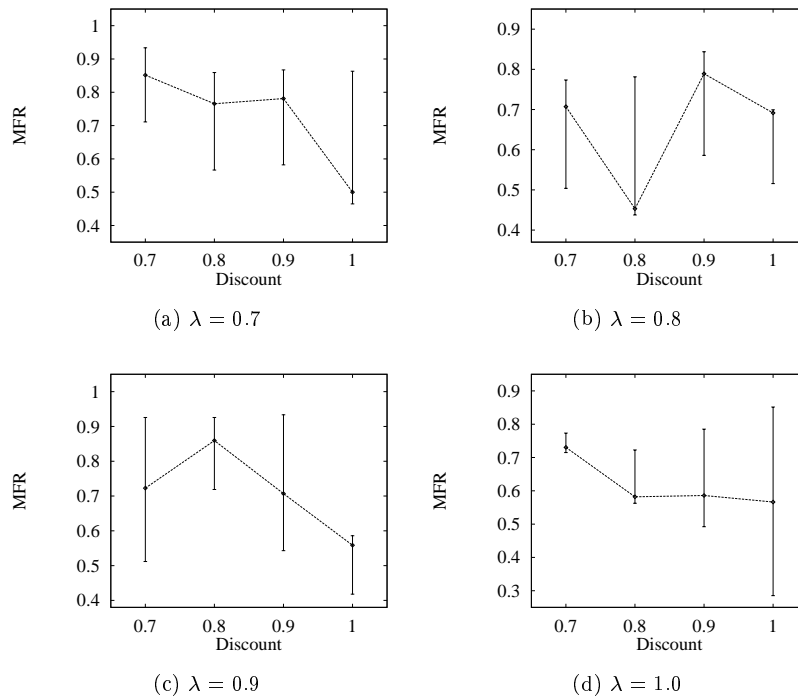


Figure 2: ‘-’ Partition.

References

- [1] W.J.M. Levelt. *Speaking: From Intention to Articulation*. MIT Press, 1995.
- [2] I. Johnson and M. Plumbley. Finding synonyms in a regular language using reinforcement learning. Technical Report 11, Dept. of Electronic Engineering, King’s College London, London, UK, 1999.
- [3] C.J.C.H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.
- [4] D.E. Rumelhart, G.E. Hinton and R.J. Williams. Learning representations by back-propagation. *Nature*, 326(6088), 1986.
- [5] J. Allen and M.S. Seidenberg. *The Emergence of Language*, chapter “The Emergence of Grammaticality in Connectionist Networks”. Lawrence Erlbaum Associates, Inc., 1999.
- [6] G.A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University, 1994.
- [7] G.A. Rummery. *Problem solving with reinforcement learning*. PhD thesis, Cambridge University Engineering Department, 1995.
- [8] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
- [9] R.S. Sutton. Implementation Details of the TD(λ) Procedure for the Case of Vector Predictions and Back-propagation. Technical Report TN87-509.1, GTE Laboratories Inc., 1989.
- [10] S. Thrun and A. Schwartz. Issues in using function approximation for reinforcement learning. *Proceedings of the Fourth Connectionist Models Summer School*, 1993.

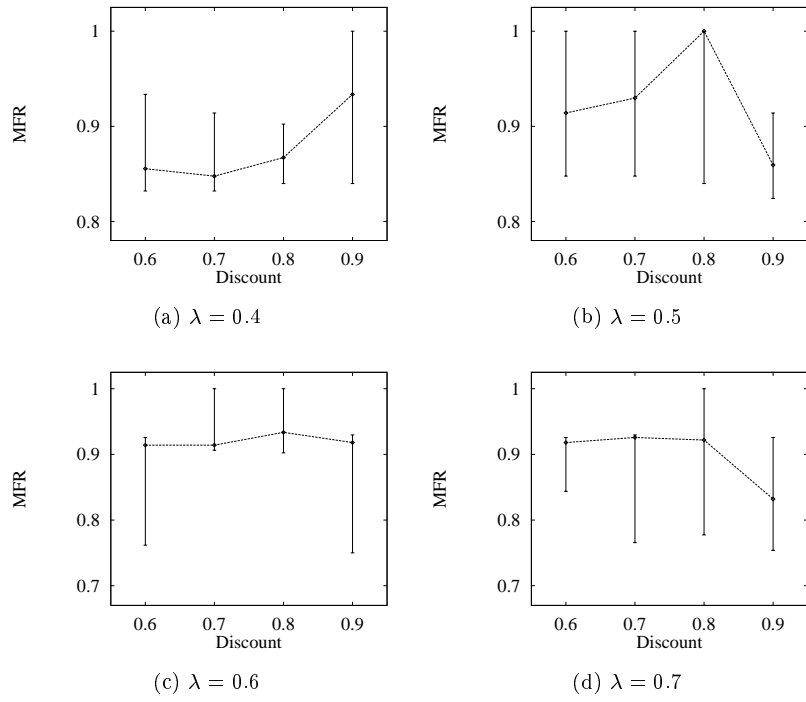


Figure 3: 'x' Partition.

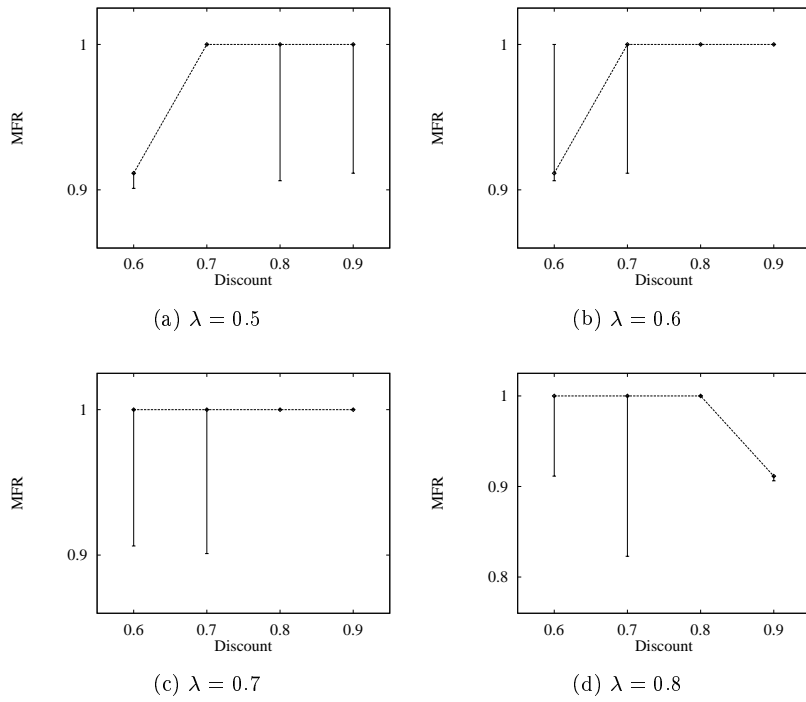


Figure 4: '÷' Partition.