Recurrent Neural Networks for Reinforcement Learning: Architecture, Learning Algorithms and Internal Representation

Ahmet Onat, Kyoto University, Kyoto, Japan
Hajime Kita, Tokyo Institute of Technology, Yokohama, Japan
Yoshikazu Nishikawa, Osaka Institute of Technology, Hirakata, Japan

Abstract—Reinforcement learning is a learning scheme for an autonomous agent that allows the agent to find the optimal policy of taking actions which maximize a scalar reinforcement signal in unknown environments. If the agent has access to the whole state of the environment, a reactive policy which maps the sensory input to the action is sufficient. However, if the state of the environment is partially observable, special methods for creating a dynamic policy that utilizes the past observations are necessary. To overcome this problem, the authors have proposed a method using recurrent neural networks with Q-learning, as a learning agent. This paper compares several types of network architecture and learning algorithms for this method through computer simulation. Further, the internal representation in the trained networks is examined using a clustering technique. It shows that the representation of the environmental state is developed well in the networks.

Keywords—reinforcement learning, recurrent neural networks, perceptual aliasing, internal representation

1. Introduction

Reinforcement learning is a learning method for agents to acquire the optimal policy autonomously from the evaluation of their behavior. The indication of the performance is called the reinforcement signal.

In this learning scheme, the learning agent receives sensory input from the environment. Then it decides and carries out an action based on the input. In return, it receives a reinforcement signal, and new sensory input. The agent uses these episodes to maximize the reinforcement signals aggregated in time. The following discounted cumulative reward \( V_t \) is often used:

\[
V_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}
\]

where \( r_t \) is the immediate reinforcement at time \( t \), and \( 0 < \gamma < 1 \) is a constant called the discount rate. To achieve this goal without explicit teachers, the learning agent is required to search the optimal policy, i.e., a mapping from the sensory input to the action, by trial-and-error. For this learning scheme, Barto[1] and Werbos[17] detail a number of reinforcement learning methods. Kaelbling et al.[4] have also prepared a good survey of topics relating to this learning scheme.

Q-learning proposed by Watkins[15] is a widely used implementation of reinforcement learning based on the dynamic programming technique and temporal difference methods[13]. In Q-learning, the learning agent estimates the expected discounted cumulative reward of taking action \( a_k \) at state \( x_t \). These estimates are called the \( Q(x_t, a_k) \) values. Using the results of each action, it updates the Q values according to the following equation:

\[
Q(x_t, a_t) \leftarrow Q(x_t, a_t) + \alpha (r_t + \gamma \max_a Q(x_{t+1}, a) - Q(x_t, a_t))
\]

where \( \alpha \) is the learning rate. Until the Q values converge, it is necessary to make sure that the agent experiences each state-action pair frequently enough. This can be done by selecting the actions randomly during the learning process. Once the Q values are established, the optimal action selection policy \( \pi^*(x_t) = a_k \) is simply to choose the action with the maximal Q value at each time step. Lin[6] has proposed a modification of Q-learning which enables it to be used with multi-layer neural networks combined with the error backpropagation algorithm.

Although Q-learning is based on the assumption that the whole state of the environment is available at each time step, in practice, the environment might have some hidden states. In such a case, the same sensory information from the environment can be produced by different states, and may cause reactive agents to face difficulty in yielding the optimal actions. Whitehead et al. termed this as ‘perceptual aliasing’[18]. As a class problems including this difficulty, ‘Partially observable Markovian decision processes (POMDP),’ are considered frequently.

Several approaches to solve the problem of perceptual aliasing have been proposed so far. They are categorized into three groups as follows:

The first category is to enhance the sensory input to avoid perceptual aliasing. Tan has proposed to utilize sensors better, and directly resolve the state ambiguity problem[14]. Whitehead[18] proposed the lion algorithm, and a reconfigurable sensorimotor system in this context.

The second category is to use memory, or a dynamic behavior to represent the state of the environment using past observations. Chrisman[2] and McCallum[9] propose methods which split the modeled state space of the environment, based on gathered statistics. Lin et al.[7] have proposed some methods categorized into this group. They have proposed to utilize neural networks having a tapped delay line as the input (Window-Q), recurrent neural networks to yield the Q-values (Recurrent-Q), and recurrent neural networks to model the state transition (Recurrent-Model). The authors have also proposed to utilize the recurrent neural networks of the Elman type[11] and of the X-model type[12].

The third category is to produce a reactive policy that
makes the best use of the immediate sensory information. In this category, the problem of perceptual aliasing is solved probabilistically by allowing the agent to take a stochastic policy. Littman[8], and Kimura et.al.[5] have proposed such methods.

Our previous works[11], [12] have shown the feasibility of using recurrent neural networks for reinforcement learning. In this paper, we try to extend these results, and discuss architecture and learning algorithms for recurrent neural networks suitable for reinforcement learning. Further, we examine the internal representation of the neural network, acquired by learning, using a clustering technique.

2. Q-learning for recurrent neural networks

2.1. Structure of the learning agent

The structure of the learning agent is shown In Fig. 1. It consists of a recurrent neural network (RNN) and a stochastic action selector (SAS).

For the RNN, we use an architecture which can yield the output from the immediate input with no time delay, so as to include neural networks for reactive agents. The inputs of the network consist of the sensory inputs from the environment $y_t$, the previous action $a_{t-1}$ and the internal feedback. We refer to these three input vectors by a single one $z$; concatenating them for simplicity. The network has as many outputs as the number of possible actions. Each output represents the Q-value of the associated action. For the output units, a linear activation function is used to cope with the required range of Q values. For the hidden units, the sigmoid function is used as the activation function.

The network outputs are fed to the stochastic action selector which chooses one action randomly, with a bias toward those with higher Q values. Probability of action selection follows the Boltzmann distribution[6]:

$$\text{Prob}(a_k) = \frac{1}{f} \exp(Q(z_t, a_k)/T_t)$$

$$f = \sum_a \exp(Q(z_t, a)/T_t)$$

where, Prob($a_k$) is the probability of action $a_k$ being selected to be carried out. The parameter $T_t$ is called temperature, which controls the randomness of the action selection. It is set high in the beginning of training, and lowered in each iteration to reduce randomness using the equation

$$T_{t+1} = T_{\text{min}} + \beta(T_t - T_{\text{min}})$$

where $\beta$ and $T_{\text{min}}$ are constants.

2.2. The learning procedure

The learning procedure is a combination of the Q-learning algorithm and a weight update rule of the steepest descent type. It is summarized as follows:

1. Read the current sensory inputs $y_t$ at time $t$. Form a vector $z_t$ by concatenating the previous action $a_{t-1}$ and the context (the internal feedback) to $y_t$. Feed $z_t$ into the network, and obtain $Q(z_t, a_k)$ values for each action $a_k \in A$, where $A$ is the set of possible actions.
2. Using the stochastic action selector, choose one action $a_k \in A$ and apply it to the environment.
3. Get the new inputs $y_{t+1}$, and the reinforcement $r_{t+1}$ from the environment.
4. Obtain the network output; $Q(z_{t+1}, a_k)$ for each action $a_k \in A$.
5. Calculate the error term

$$\Delta Q = r_{t+1} + \gamma \max_{a_k} Q(z_{t+1}, a_k) - Q(z_t, a_k)$$

where $\gamma$ is the discount rate.
6. Construct the error vector $\tilde{e}(t)$ by using $\Delta Q$ for the output unit corresponding to the taken action, and 0 for the other output units.
7. Update the RNN weights, using a supervised learning algorithm, with the error vector $\tilde{e}(t)$. Go to Step 1.

2.3. Architecture of recurrent neural networks

For the aforesaid structure of the learning agent, we have used two types of architecture[12], namely the Elman network[3], and the Information Loss Minimization method (ILMM)[10].

The Elman network is similar to a three layer feedforward neural network as shown in Fig. 2. However, it has feedback connections from the hidden layer through a one-step time delay register called 'the context register.' The ILMM method is proposed by Noda so as to enhance the memory capability of the Elman network. This method uses the X-model architecture shown in Fig. 3. Compared to the Elman network, it has two extra output unit groups called 'context reconstruction,' and 'input reconstruction.' The network is trained to yield the desired output, and at the same time, it is required to reconstruct the current input from the activation pattern in the hidden layer. This requirement is implemented as the minimization of the error function:

$$E = ((y_0 - d_0)^2 + (y_{RI} - y_I)^2 + (y_{RC} - y_C)^2)$$
Figure 2. Architecture for the Elman network.

where \(y_0, d_0, y_RI, y_H, y_RC\) and \(y_C\) are the activation pattern vectors of the output, the desired output, the reconstruct-input, the input, the reconstruct-context, and the context layers respectively. Addition of the two output groups makes sure that enough information about the current input and the previous state (context) is retained in the hidden layer. This results in relevant information being retained for a long time.

Figure 3. X-model architecture for the information loss minimization method.

2.4. Learning algorithms for recurrent neural networks

The following three algorithms are widely used for training recurrent neural networks:

- Error Backpropagation Algorithm (BP)[16]: The network dynamics caused by the feedback in the network is ignored, and the error backpropagation algorithm for feedforward networks is applied. Feedback connections are considered as ordinary inputs. While this method does not provide accurate gradient, it is easy to implement, and empirically it works well to some extent.

- Backpropagation Through Time (BPTT): In this method, the error is propagated back in time by unfolding the network in time, so as to consider the dynamics of the network. Thus, accurate gradient is obtained. Lin[7] has used this method with decay in the propagation of error back in time to limit it. Since this method requires a reverse-in-time calculation, it is not suitable for real-time learning algorithms.

- Real Time Recurrent Learning (RTRL)[19]: This method also considers the dynamics of the network. It makes possible the calculation of the accurate gradient with only forward-in-time calculations. Hence, it is suitable for real-time learning applications. However, during learning it requires greater computation time and memory than the above two methods.

We have chosen the BP[16] and RTRL[19] algorithms, because they are more suitable for reinforcement learning, which has real-time nature, than BPTT.

Since the algorithm of BP is quite popular, we skip its description. The training algorithm for RTRL is summarized as follows:

\[
\Delta w_{ij}^o(t) = \eta e_i f'(s_i^o(t)) h_j(t) \quad (6)
\]

\[
\Delta w_{ij}^h(t) = \eta \sum_{k=1}^n c_k(t) f'(s_k^h(t)) \left[ \sum_{l=1}^h w_{ij}^h \frac{\partial h_l(t)}{\partial w_{ij}^h} \right] \quad (7)
\]

\[
\frac{\partial h_l(t)}{\partial w_{ij}^h} = f'(s_i^h(t)) \left[ \sum_{n} w_{kn}^h \frac{\partial h_n(t-1)}{\partial w_{ij}^h} + \delta_{ln} z_j(t) \right] \quad (8)
\]

\[
\frac{\partial h(0)}{\partial w_{ij}^h} = 0 \quad (9)
\]

where, \(w_{ij}^o\) and \(w_{ij}^h\) are the weights from \(j\)th unit to the \(i\)th unit of the output and the hidden units respectively, and \(\Delta w_{ij}^o\) and \(\Delta w_{ij}^h\) are their updates. \(\eta\) is the learning rate, \(e_i\) is the error term of output unit \(i\), \(f\) and \(f'\) are the activation function of a unit and its derivative respectively, \(s_i^o\) and \(s_k^h\) are the weighted summation output of the \(k\)th output and hidden units respectively, \(h_j\) is the output of the hidden unit \(j\), and \(\delta\) is Kronecker's delta.

The dynamics of the network is calculated and propagated using the recursive term, defined in equations (8) and (9). It should be noted that in Eq. (8), the partial derivative \(\frac{\partial h_l(t)}{\partial w_{ij}^h}\) has to be calculated even among units and weights that are not directly connected, because there exists an indirect influence between them, via other feedback connections. This is what makes this learning scheme require large computation time and memory.

3. Simulation environment

For computer simulation, the following partially observable Markovian decision process is used[11], [12]. As shown in Fig. 4, there is a chain of rooms, which are connected with doors having locks. The doors of certain rooms have been locked in advance. An agent roams in this environment. In each room, the agent can take one of the following three actions:

1. Turn the door's lock. Turning the lock of a closed door inverses the state of lock, i.e. 'locked' to 'unlocked' and vice versa. Turning the lock of an open door does not have any consequences.
2. Try to open the door to the next room. A door will open only if it is not locked.
3. Attempt to go ahead into the next room. The agent can only proceed to the next room if the door is already open.

2012
The sensory inputs of the agent are the number of the room he is in, and the state of the door of that room, i.e., 'open' or 'closed'. But, the state of the lock is not directly observable. Further, in some rooms, the room number is not observable. These cause perceptual aliasing.

The agent receives reinforcement signal of -1 for bumping into closed doors; of +1 for going into the next room, and of +5 for exiting the building. For other situations, no reinforcement signal is provided. The agent is, in the beginning, put into one of the non aliased rooms randomly. The aim of the agent, i.e., the optimal behavior, is to start from the room and to go through the rooms to the exit as quickly as possible. This task requires that the agent associate each room number with the state of its lock, and in the rooms without a room number, it infers this information based on the previous rooms. Starting from the first room, the agent can finish the course in fourteen actions with the optimal policy. We define a 'run' as the process of getting to the exit from the starting room, which is selected randomly among rooms 1, 3 and 6, and 'Redundant actions' are defined as those that are not necessary if the optimal policy is used.

As seen from Table 1, the Elman/BP combination was by far the slowest. It converged after 250,000 iterations, and it also required the largest number of hidden units. The X-model/BP combination is the quickest of all, which converged in less than 20,000 steps. It is followed by X-model/RTRL, converging at around 32,000 steps. These two agents also converged with the least hidden units (8 units) although the small number of hidden units required slightly longer training time. For agents that used either RTRL or X-model, or both, learning was quicker, so the temperature could be dropped faster with smaller $\beta$. It was also observed that RTRL requires a smaller learning rate compared to BP. High learning rates caused instability in the RTRL algorithm, as predicted by Williams et al.[19]

The CPU time taken for the simulation is also an important issue. Even though 'the number of iterations until convergence' for RTRL and BP were comparable, the RTRL algorithm took longer to run on the computer. This is because the dynamics of the partial derivatives, Eq. (8), need to be computed, as well as the weight update values. The number of hidden units also affects the run time. This effect is more pronounced for the X-model network, because the number of inputs ($z_i$), and the number of outputs are dependent on the number of hidden units.

4. Results

4.1. Comparison of the network architecture and learning algorithms

Each of two sorts of architecture and the two learning algorithms were combined to give four types of agents. The initial values of the connection weights were chosen randomly from the interval [-0.1,0.1]. The output of the context registers were set to 0.5 at the beginning of runs. For all combinations, the number of hidden units and the parameters for learning were chosen so as to obtain fastest convergence. The cumulative discount rate $\gamma$ is set to 0.8 in all the simulations. Table 1 shows the performance of the tested agents. Convergence of each agent is judged by the number of redundant actions going to zero for 500 consecutive iterations. Figs. 5-8 show the convergence processes of the tested agents. In Table 1, the minimal numbers of hidden units that succeeded in learning are also shown.
the two clusters closest to each other are merged into a single one. The new cluster is represented by the average of all data points contained in it. The process is repeated until only one cluster remains. In the resulting graph, the activation vectors of hidden units that have a small distance between them, appear close together at the leaves of the branching tree.

After the clustering procedure, each leaf of the tree was matched with the actual state of the environment that produced it. By examining the tree, it is possible to see which state elements are not yet classified well by the network; if similar environment states map to different hidden unit activations, and vice versa, then it is likely that they are not yet sorted out. On the other hand, if each environmental state is mapped to its own distinctive activation pattern, then it can be said that the agent has learned to distinguish the environment states.

The results of analysis shows that all of the four types of agents developed their internal representations in a similar way. It was as follows: In the first 10% of learning, i.e., up to about 10% of the time to convergence, the internal representation of all the agents can be grouped based on readily visible inputs, i.e., state of the doors and the previous action. Some non-aliased rooms were also identified. The states of the locks and aliased rooms were not classified in these representations at all.

Between 10% and 50% of learning, the agents learned to distinguish the states of locks. For non-aliased rooms, the states of the locks were learned, but mostly all of the aliased rooms were still classified as the same.

Around 50% of learning, the agents had mostly distinguished all of the states of the environment. However, some environment states were represented by more than one cluster. A few misclassified instances also occurred at this stage.

After convergence, all of the environment states were correctly classified, and cleanly spread out. Each environmental state forms a single cluster in the internal representation. It is also observed that the clusters representing...
states were further classified into larger groups according to the state of doors, and the last action, and classified into smaller groups according to the lock states. Large groups of common room numbers were not observed. Aliased states had moved away from one another also. The input representation for the X-model/BP agent is given in Fig. 5, together with the corresponding environment states, i.e., the room, the lock, the door and the previous action. This tree was simplified by removing the branches of several deep levels.

5. Conclusion

In this paper, we have discussed methods for utilizing recurrent neural networks as a dynamic agent that can discover the optimal policy by reinforcement learning in environments with hidden states. The proposed model is a combination of Q-learning and recurrent neural networks. Two sorts of network architecture, i.e., the Elman network and the X-model, and two learning algorithms, i.e., the conventional error backpropagation (BP) and the real-time recurrent learning (RTRL), have been studied.

All of the combinations of the network architecture and learning algorithms succeeded in finding the optimal policy in the test environment. Among these, the combination of the Elman net with the BP was slowest in learning speed. The combination of X-model/BP was the fastest while the other two combinations was also much faster than the combination of Elman/BP. However, the results of an ongoing study shows that the combination of the X-model and RTRL shows more favorable performance in more difficult environments than the others. Hence it is an important subject of further study.

The internal representation in the networks is also examined using a clustering technique. The analysis shows that the recurrent neural networks have succeeded in reconstructing the hidden environmental state. The internal representation has been developed gradually with the progress of learning. A simple classification of the states based on the sensor inputs appeared first. This is followed by development of the classification of the hidden states. Finally, clear clusters corresponding to the environmental states in one-to-one sense were obtained after learning was completed.

Acknowledgements: The authors would like to thank Masami Kuramitsu of Graduate School of Engineering, Kyoto University, Kyoto Japan, for his valuable comments during the course of this research.

References