Reinforcement Learning with Neural Networks: Tricks of the Trade

Article *in* Studies in Computational Intelligence · January 2013

DOI: 10.1007/978-3-642-28696-4-11

CITATIONS
6

READS
328

2 authors:

Christopher J Gatti
Independent Scholar
25 PUBLICATIONS **178** CITATIONS
SEE PROFILE

M. Embrechts
Rensselaer Polytechnic Institute
186 PUBLICATIONS **3,245** CITATIONS
SEE PROFILE

Chapter 11
Reinforcement Learning with Neural Networks: Tricks of the Trade

Christopher J. Gatti and Mark J. Embrechts

Abstract. Reinforcement learning enables the learning of optimal behavior in tasks that require the selection of sequential actions. This method of learning is based on interactions between an agent and its environment. Through repeated interactions with the environment, and the receipt of rewards, the agent learns which actions are associated with the greatest cumulative reward.

This work describes the computational implementation of reinforcement learning. Specifically, we present reinforcement learning using a neural network to represent the valuation function of the agent, as well as the temporal difference algorithm, which is used to train the neural network. The purpose of this work is to present the bare essentials in terms of what is necessary for one to understand how to apply reinforcement learning using a neural network. Additionally, we describe two example implementations of reinforcement learning using the board games of Tic-Tac-Toe and Chung Toi, a challenging extension to Tic-Tac-Toe.

11.1 Introduction

Reinforcement learning is a machine learning method that enables the learning of optimal behavior by an agent through the repeated interaction with an environment. Optimal behavior in this case can be defined as the set of sequential decisions that result the best possible outcome. This learning process can essentially be regarded as a process of trial and error, which is coupled with the receipt of feedback that guides the learning of the best actions to pursue during the decision process.

Board games are prime examples of reinforcement learning. Players take turns assessing the state, or the configuration, of the game, and then select the action that

Christopher J. Gatti · Mark J. Embrechts
Rensselaer Polytechnic Institute, Troy, NY
e-mail: {gattic,embrem}@rpi.edu

P. Georgieva et al. (Eds.): Adv. in Intelligent Signal Processing and Data Mining, SCI 410, pp. 275–310.
springerlink.com

they believe will most likely result in an outcome that is in their best interest. Game play therefore consists of the making of sequential decisions in order to achieve a goal, which is most often to win the game. During the game however, the true utility and benefit of each action is not known; only at the end of the game is there certain evidence of the utility of all of a players' actions played during the game.

Consequently, board games have been a prominent domain for the development of computational implementations of reinforcement learning. This is largely due to the fact that board games have well-defined and completely understood rules and actions. Real-world domains, in comparison, may have complex underlying processes that are not fully understood, thus complicating the process of learning how to interact within such domains. Reinforcement learning has been applied to many board games, including checkers [12], chess [10], and Othello [2]. The most notable and successful implementation is the application of reinforcement learning to the game of backgammon by Tesauro [16, 17, 18]. In this work, reinforcement learning was used to train a neural network to play backgammon to such a high level that it could challenge world-class human opponents.

Board games are also useful for explaining reinforcement learning in terms of both the methodology and the practical implementation details. This is also due to the well-defined domains of board games and because the elements of reinforcement learning and board games are quite analogous to each other. The purpose of this work is to present reinforcement learning at a basic level, including only what is essential to understanding and implementing this learning method. More specifically, this work explains the application of reinforcement learning using a neural network to the game of Tic-Tac-Toe, which is then extended and adapted to the game of Chung Toi, a challenging extension to Tic-Tac-Toe. Additionally, we present various tricks and techniques that have the potential to improve the efficiency of training the neural network for reinforcement learning. This work aims to provide the essential background knowledge that is required to implement reinforcement learning to any game-like learning scenario which relies on sequential decisions.

This work begins with both a general overview and a formal definition of reinforcement learning in Section 11.2. Section 11.3 focuses on the implementation of reinforcement learning with a neural network where we describe all of the necessary components to apply reinforcement learning and discuss parameter and algorithm settings as well as modifications to the basic implementation. We then describe in detail two example applications of reinforcement learning in Section 11.4, that of Tic-Tac-Toe and Chung Toi. Finally, we conclude and summarize this chapter in Section 11.5. Throughout this chapter some material is occasionally repeated using similar wording, and this is purposely done in order to reinforce important concepts of this learning paradigm or to present the material in a slightly different manner.

11.2 Overview of Reinforcement Learning

Reinforcement learning is based on an *agent* that repeatedly interacts with an *environment* (Fig. 11.1). Interactions between the agent and the environment proceed

by the agent observing the state of the environment, selecting an action which it believes is likely to be beneficial, and then receiving feedback, or a *reward*, from the environment that provides an indication of the utility of the action (i.e., whether the action was good or bad). More specifically, the agent selects actions based on the its perception of the value of the subsequent state. The feedback provided to the agent is used to improve its estimation of the value of being in each state. In the general reinforcement learning paradigm, rewards may be provided after each and every action. After repeated interaction with the environment, the agent's estimation of the true state values slowly improves, which enables the selection of more optimal moves in future interactions.

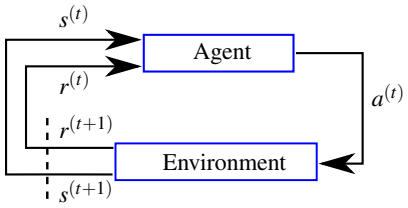

Fig. 11.1. The reinforcement learning paradigm consists of an agent interacting with an environment. The agent observes the state of the environment $s^{(t)}$ and pursues an action $a^{(t)}$ at time t in order to transition to state $s^{(t+1)}$ at time $t+1$, and the environment issues a reward $r^{(t+1)}$ to the agent. Over time, the agent learns to pursue actions that lead to the greatest cumulative reward. Figure adapted from [15].

This method was largely developed as a machine learning technique by Sutton and Barto [14, 15]. Many formulations of reinforcement learning have been developed in order to either accommodate various types of environments or to utilize slightly different learning mechanisms. One of the fundamental reinforcement learning methods is the temporal difference algorithm, which is the main focus of this work. This algorithm uses information from future states, and effectively propagates this information back through time, in order to improve the agent's valuation of each state that was visited during an episode. Consequently, the improvements to the estimations of the state values has a direct impact on the action selection processes as well. This algorithm is particularly useful in scenarios that do not necessarily follow the schema shown in Fig. 11.1 where rewards are provided following every action. In some scenarios, with board games being a prime example, feedback is often only provided at the end of the game in the form of a win, loss, or draw. The temporal difference algorithm can be used to determine the utility of actions played early in the game based on the outcome of the game, which results in a refined action selection procedure.

11.2.1 Sequential Decision Processes

Reinforcement learning is more formally described as a method to determine optimal action selection policies in sequential decision making processes. The general framework is based on sequential interactions between an *agent* and its *environment*, where the environment is characterized by a set of *states* S, and the agent can pursue *actions* a from the set of possible actions A. The agent interacts with the environment and transitions from state $s_t = s$ to state $s_{t+1} = s'$ by selecting an action $a_t = a$. Transitions between states are typically based on some defined probability $\mathcal{P}^a_{ss'}$. Note that the subscript t indicates the particular time step at which states are visited or actions are pursued, and thus the sequences of states and actions can then be thought of as a progression through time.

The sequential decision making process therefore consists of a sequence of states $s = \{s_0,\ s_1,\ ...,\ s_T\}$ and a sequence of actions $a = \{a_0,\ a_1,\ ...,\ a_T\}$ for time steps $t = 0, 1, ..., T$, where state s_0 is considered to be the initial state. Feedback may be provided to the agent at each time step t in the form of a *reward* r_t based on the particular action pursued. This feedback may be either rewarding (positive) for pursuing beneficial actions that lead to better outcomes, or they may be aversive (negative) for pursuing actions that lead to worse outcomes. Processes that provide feedback at every time step t have an associated reward sequence $r = \{r_1,\ r_2,\ ...,\ r_T\}$. A complete process of agent-environment interaction from $t = 0, 1, ..., T$ is referred to as an *episode* consisting of the set of states visited, actions pursued, and rewards received. The total reward received for a single episode is the sum of the rewards received during the entire decision making process, $\mathcal{R} = \sum_{t=1}^{T} r_t$.

For the general sequential decision making process define above, the transition probabilities between states $s_t = s$ and $s_{t+1} = s'$ may be dependent on the complete sequences of previous states, actions, and rewards:

$$Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t,\ a_t,\ r_t,\ s_{t-1},\ a_{t-1},\ r_{t-1},\ ...,\ s_1,\ a_1,\ r_1,\ s_0,\ a_0,\} \quad (11.1)$$

Processes in which the state transition probabilities depend only the most recent state information $(s_t,\ a_t)$ are said to satisfy the *Markovian assumption*. Under this assumption, the state transition probabilities can be expressed as:

$$Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t,\ a_t\} \quad (11.2)$$

which is equivalent to the expression in (11.1) because all information from $t = 0, 1, ..., t-1$ does not affect the state transition at time t.

If a process can be assumed to be Markovian and can be posed as a problem following the general formulation defined above, this process is amenable to both modeling and analysis. The Markov decision process (MDP) is the specific process which entails sequential decisions and consists of a set of states S, a set of permissible actions A, and a set of rewards R. Each state $s \in S$ has an associated true state value $V^*(s)$. The decision process consists of repeated agent-environment interactions in which the agent attempts to learn the true value of each state. The true state

values are unobservable to the agent however, and thus the agents' estimates of the state values $V(s)$ are merely approximations to the true state values.

The transition between states $s_t = s$ and $s_{t+1} = s'$ when pursuing action a may be represented by a transition probability $\mathcal{P}^a_{ss'} = Pr\{s_{t+1} = s \mid s_t = s,\ a_t = a\}$. For problems in which the transition probabilities between all states are known, the transition probabilities can be used to formulate an explicit model of the agent-environment process. A policy π is defined to be the particular sequence of actions $a = \{a_0,\ a_1,\ ...,\ a_T\}$ taken throughout the decision making process. Similarly, the expected reward for transitioning from state s to state s' when pursuing action a may be represented by $\mathcal{R}^a_{ss'} = E\{r_{t+1} \mid s_t = s,\ a_t = a,\ s_{t+1} = s'\}$. An optimal policy π^* is considered to be the set of actions pursued during the process maximizes the total cumulative reward R received.

The optimal policy can be determined if the true state values $V^*(s)$ are known for every state. The true state values can be expressed using the Bellman optimality equation, which states that the value of being in state s and pursuing action a is a function of both the expected return and the optimal policy for all subsequent states:

$$\begin{aligned} V^*(s) &= \max_{a \in A} E\left\{r_{t+1} + \gamma V^*(S_{t+1}) \mid s_t = s,\ a_t = a\right\} \\ &= \max_{a \in A} \sum_{s^+} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^*(s')\right] \end{aligned}$$

The Bellman equation provides a conceptual solution to the sequential decision making problem in that the value of being in a state is essentially a function of the value of all future states. The solution to reinforcement learning problems is often regarded as a policy π, or an action selection procedure, that leads to an optimal outcome. When implementing reinforcement learning however, the Bellman equation does not have to be explicitly solved. Rather, the identification of optimal policies in reinforcement learning problems is merely based on the notion that future information is relevant to the valuation of current states, and that accurate estimations of state values can be used to determine the optimal policy.

As previously mentioned, if the transition probabilities $\mathcal{P}_{ss'}$ between states are explicitly known, these probabilities can be used to formulate a model of the system. If however, these transition probabilities are not known, transitions between states must be made in a different manner. In this case, we can use the structure of the environment together with the allowable actions to essentially constrain and generate the transitions between states. Using the game of Tic-Tac-Toe as an example, the environment (including the board layout, players, and pieces), the rules of the game, and the possible actions are clearly defined. When state transition probabilities are not known, as in these cases, the agent-environment system is considered to be *model free*. The transition probabilities are essentially learned however, through the repeated interaction between the agent and environment.

11.2.2 Reinforcement Learning with a Neural Network

The overarching concept of reinforcement learning is that the state values are learned through repeated agent-environment interaction, and that the behavior, or the action policy, of the agent emerges from the knowledge of the state values. Initially, the agents' state value estimations are erroneous because it has no knowledge of the environment. Agent-environment interactions improve the accuracy of the agents' state value estimates, such that these estimates become closer to the true, unknown state values.

A computational implementation that models the agent-environment interaction therefore requires some method of keeping track of all state values as well as the ability to update the state values based on additional knowledge gained through interactions with the environment. A basic method to track and update state values is to explicitly store each state value in a look-up table. This method however, is only practical for problems that have a relatively small state spaces. Additionally, when using a reinforcement learning algorithm that is based on state-action pair values, as opposed to just state values, this table must track the values of all state-action pair combinations, and can thus be quite large even for relatively small problems.

Another method that can be used to track all state values and that is not limited by the size of the problem is the use a function approximator that learns to approximate the true state values. This is accomplished by associating an input state vector $\mathbf{x}$ with a corresponding state value $V(\mathbf{x})$ as an output. In this work, we focus on using a neural network as the function approximator, and thus the neural network represents the agent which learns the state values. This approach is advantageous over look-up tables because it requires a relatively small number of parameters and because neural networks can approximate nonlinear state value functions. Other types of function approximators may be used as long as there are free parameters which can be updated and adjusted in order to produce better approximations of the true state values. In the case of a neural network, the connection weights between the nodes are adjusted such that the network output becomes a better approximation of the true state values. This weight updating procedure is driven by the agent-environment interactions and the feedback received by the agent, and this process is represents the training of the neural network.

The neural network interacts with the environment in two ways: 1) pursuing actions, 2) and learning. More specifically, the action selection procedure proceeds by the neural network observing the current state of the environment and using the state-value estimates of the potential subsequent states to select which action to pursue. When the agent selects actions that always correspond to next-state values with the greatest value, the agent is said to follow a *greedy* action selection policy. The learning procedure uses the rewards received, which are a direct result of the actions pursued, to adjust the network weights, which thus improves the state value estimates. In other words, the rewards received are what guides the learning of the state values.

11.3 Implementing Reinforcement Learning

Applying reinforcement learning to a problem formulated as a sequential decision making process requires the representation of a few key elements. The main components are: 1) the *environment* and 2) the *agent*. The environment is characterized by a set of *states* S. The agent transitions between states $s \in S$ by selecting *actions* a from a set of possible actions A based on the *values* of potential subsequent states $V(s')$. The agents' state value estimates (i.e., its knowledge about the environment) is refined through interacting with the environment and by the receipt of *rewards*, such that positive rewards are received for beneficial actions and negative rewards are received for unfavorable actions. Additionally, the learned knowledge, or the expertise, of the agent can be evaluated using some form of a knowledge evaluation method to determine how well the agent is learning.

The section that follows describes various aspects of each of these elements with the aim of explaining how each element can be applied in practice. An implementation of reinforcement learning also requires the user to specify numerous settings and parameters, and many different combinations may yield successful results. This section also provides some insight appropriate parameter and algorithm settings.

The description of the implementation of reinforcement learning is facilitated with the aid of a tangible example of a decision making process. The well-known game of Tic-Tac-Toe will be used in the following section to make some concepts clear. For those unacquainted with Tic-Tac-Toe, a brief overview follows. Tic-Tac-Toe is a board game between 2 players that is played on a board consisting of 9 positions configured in a 3×3 square. One player is assigned pieces represented by $\bigcirc$ and the other is assigned pieces represented by $\times$. Players alternate taking turns and place their pieces on open board positions with the goal of trying to get 3 of their own pieces in a row either along a horizontal, a vertical, or a diagonal.

11.3.1 Environment Representation

The construction of the environment for use within a reinforcement learning framework is specific to the task at hand. As the actions (i.e., state transitions) of the agent are essentially governed by the environment, it is important to develop an in silico environment that accurately reflects that of the real environment.

State Encoding Scheme

The state of the environment is numerically represented by a state vector $\mathbf{x}$, which serves as the input to the neural network. The construction of this vector and what this vector represents can influence the ability of the agent to learn. All information that is relevant to the environment and that may influence the behavior of the agent must be fully contained within this state vector by the encoding of specific features

about the environment or domain. The particular features that are used to describe each state are not limited by any particular encoding scheme, and thus there may be multiple state encoding schemes that work well for the same environment.

In the game of Tic-Tac-Toe for example, the most apparent state encoding scheme is a *raw* encoding scheme. This scheme represents the state of the environment in terms of the configuration of the pieces on the board. This encoding scheme and state vector may also include information regarding which player is to select the next move, as this information may also affect which actions are selected. The configuration of the pieces on the board may be represented by a 9×1 vector $\mathbf{p}$, with each element of the vector corresponding to one particular position on the Tic-Tac-Toe board. The presence of a piece on the board can be indicated by a 1 for ○, a -1 for ×, and a 0 for an open board position. A binary 2×1 vector $\mathbf{t}$ may also be used to indicate the player that is to select the next move, such that $[1,\ 0]^T$ corresponds to ○ selecting the next move and $[0,\ 1]^T$ corresponds to × selecting the next move. These two vectors could then be concatenated to form a single 11×1 state vector $\mathbf{x} = \left[\mathbf{p}^T\ \mathbf{t}^T\right]^T$.

A simple raw encoding scheme however, may not be best state representation in terms enabling the neural network to learn the state values. When a raw encoding scheme is not used, the idea is to extract features from the environment that are thought to be relevant to the valuation of each state, and thus also relevant to the goal of the learning process. Examples of feature-based encoding schemes for the game of Tic-Tac-Toe are presented by [8]. These encoding schemes use additional features beyond the raw encoding, which include singlets (lines with exactly one piece), doublets (lines with exactly two of the same pieces), diversity (number of different singlet directions), and crosspoints (an empty position for which two singlets of the same piece intersect). The utility of hand-crafted features has shown to be considerably advantageous in the game of backgammon [17]. In this work, the encoding scheme used both the locations of each piece on the board (raw encoding) as well as specific features that aimed to provide significant expert insight to the state valuations. These additional features proved to be very effective, enabling the neural network to be able to challenge world-class human opponents.

Training Methodology

The methodology used to train the agent to learn a domain can also have a significant effect on the efficiency at which the agent learns the game and on the ultimate performance of the agent to interact with the environment. In the game of Tic-Tac-Toe, the opponent may also be considered a component of the environment that interacts with the agent. In this case, a simple approach to training the agent is to use an opponent which simply selects moves at random, thus introducing a stochastic element to the environment. This strategy is likely to result in an agent which has little expertise in the game however, because very little skill and knowledge is required to match the random actions selected by the opponent.

Another strategy is to use an opponent that has equivalent skill to that of the agent. This can be accomplished simply by using the agent as both the agent as well as the opponent, and this strategy is known as *self-play* training. The output of the neural network can be thought of as the confidence in which the episode will result in a successful outcome for the agent. If, in the game of Tic-Tac-Toe for example, the agent is regarded as ○, the output of the neural network is therefore the confidence in which ○ will win the game. Furthermore, if a reward of 1 is received by the agent for a win, and a reward of -1 is received for a loss, output values from the neural network that are closer to 1 indicate greater confidence that ○ will win the game and output values closer to -1 indicate greater confidence that ○ will lose the game. The confidence in which ○ will win the game can equivalently be thought of as the confidence in which × will lose the game. Conversely, the confidence in which ○ will lose the game is equivalent to the confidence in which × will win the game.

This relationship between ○ winning and × losing (and vice versa) can be used to train a single neural network that learns from all games, regardless of the outcome. As previously described, the action selection procedure for ○ may follow a greedy policy in which actions resulting in subsequent states with the greatest positive value are selected. This policy can be used because a positive reward corresponds to ○ winning the game. The same neural network may also be used to select actions for × using a similar policy, except that actions resulting in subsequent states with the smallest value (which can be negative) are selected. This policy can be used because a negative reward corresponds to × winning the game. Although the action selection policies for each player are converses of each other, the weights of the neural network are updated uses the same procedure as outlined in Sect. 11.3.2.1 regardless of which player selects the action.

The self-play training scheme is just one method that allows for the use of a training opponent that has some expertise. Another method that can provide a challenging training opponent includes having the agent play against a program that is known to have considerable expertise such as those trained using other methods (e.g., minimax, alpha-beta pruning, etc.) [21]. Two additional training procedures rely on merely presenting game scenarios to the agent or having it learn from observations. One of these methods uses database games, which consists of complete sets of state vectors from a large number of games that have been stored in a database. The learning strategy when database games are used is similar to that described above however, in this case the agent does not select moves, but rather it is merely presented with sequential state vectors. A second methods similar to that of database games is to present game scenarios to the agent which have been selected by expert human players. This method attempts to embed specific expert knowledge into the agent by using specific game scenarios. The use of methodologies that are not based on self-play may only allow the agent to gain expertise that nearly equals that of the training opponent or training method. The self-play training method on the other hand, seems to be able to continually increase its playing expertise as it continues to play training games, as is exemplified by [17, 18]. Self-play also training allows the agent to potentially learn novel strategies that have not been conceived or widely used by human players [18].

Evaluation of Agent Expertise

The success of a reinforcement learning implementation to a board games could be defined in various ways. As the ultimate goal of such implementations is to develop a machine learner that rivals, and potentially exceeds, the knowledge and expertise of humans, the playing ability of the agent versus opponents with different levels of expertise provides a measure of success. Note that the opponent that is used to evaluate the expertise of the agent is separate from that used to train the agent using the self-play training method defined above. The agent's learned expertise may first be evaluated using the simplest opponent, one that selects all moves at random. One would expect that the agent should be able to win consistently against this opponent. A next step may be to evaluate the agent using an opponent that has some expertise. This could be done by allowing the opponent to take greedy end-game moves. Such moves allow a player to select moves such that the resulting move allows the player to win the game (greedy win) or to block the win of the opposing player (greedy block), regardless of the estimated state value of the particular move. Another method to create a more challenging opponent is the allow the opponent to use the knowledge gained by the agent, but for only a small percentage of its actions. More extensive evaluations could consist of playing humans with varying levels of expertise or by having expert human players evaluate the quality of the actions selected by the agent [17]. Additionally, the expertise of the agent may also be evaluated under different conditions of the environment which may provide an advantage to one player. In Tic-Tac-Toe for example, this could be done by allowing ○ always to play the first move, thus giving ○ and advantage.

Reward Values

Rewards play an important role in temporal difference learning as these ultimately allow for the agent to learn by guiding the state value estimates toward their true values. The particular reward values used should be based on the possible outcomes for an episode of agent-environment interaction, and the reward values should reflect the desirability of the outcomes. For the game of Tic-Tac-Toe, the possible outcomes include a win, a loss, or a draw, and these correspond to outcomes that may be regarded as good, bad, and indifferent, respectively. Thus, a possible set of rewards may be 1 for a win, -1 for a loss, and 0 for a draw. Modifying the rewards from these values, such as using a negative reward for a draw, will change the state values and ultimately change the behavior of the agent.

The example reward values described above are one potential option for scenarios in which rewards are only provided at the end of an episode. For scenarios in which feedback can be provided during an episode that reflects the quality of actions with some certainty, rewards can be provided at these time points as well. Example of this include when there are clear sub-goals that must be achieved while achieving

the overall goal [1], or for continuous domains in which there are continuous measures of performance [3]. This chapter focuses only on domains where feedback is provided at the end of an episode.

11.3.2 Agent Representation

As previously mentioned, the agent can be represented by a neural network that acts as a function approximator. The neural network attempts to learn the true values of each state by interacting with the environment and refining the weights so that the outputs of the neural network become better approximations of the true state values. This section discusses exactly how the the weights of the neural network are updated using the temporal difference algorithm as well as various settings and parameters that can be used with this weight refinement algorithm.

11.3.2.1 Temporal Difference Algorithm

The temporal difference algorithm, also referred to as TD(λ), can be thought of as an extension to the back-propagation algorithm [13, 19]. The back-propagation algorithm is used to train neural networks, for example, for classification or function approximation by solving the credit-assignment problem. In other words, the network weights are individually adjusted based on how much each weight contributed to the error of the network output. TD(λ) on the other hand, solves the *temporal* credit-assignment problem, which is also referred to as back-propagation through time. This method uses information (i.e., network errors) from past time steps to update network weights at current time steps. As there is considerable similarity between the back-propagation algorithm and the temporal difference algorithm, the back-propagation algorithm will first be reviewed and will then be extended to the temporal difference algorithm. Both of these algorithms will be based on the 3-layer neural network configuration shown in Fig. 11.2. The following descriptions assume the reader has a basic understanding of neural networks and of the back-propagation algorithm.

The weight update equation of the back-propagation algorithm for a weight w_{jh} of a neural network (a weight from a node in layer h to a node in layer j) takes the general form:

$$\begin{pmatrix} weight \\ correction \\ \Delta w_{jh} \end{pmatrix} = \begin{pmatrix} learning \\ parameter \\ \alpha \end{pmatrix} \times \begin{pmatrix} prediction \\ error \\ E \end{pmatrix} \times \begin{pmatrix} local \\ gradient \\ \delta_j \end{pmatrix} \times \begin{pmatrix} input\ of \\ neuron\ j \\ y_h \end{pmatrix} \quad (11.3)$$

where the learning parameter α modulates the magnitude of the weight adjustment, E is the prediction error, δ_j is the local gradient that is based on the derivative of the transfer function evaluated at the node in layer j, and y_h is the output of hidden node h (which is also the input to output node j) and is computed as $y_h = f(v_h)$ where the induced local field is $v_h = \sum_i w_{hi} y_i$ and $f(\cdot)$ is a transfer function. The prediction

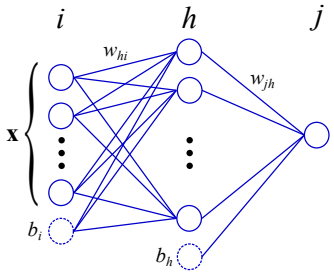

Fig. 11.2. Neural network with input layer i, hidden layer h, and output layer j. State vector $\mathbf{x}$ is input to the network and outputs a value from the single node in layer j. Weights are defined such that w_{hi} represents the weight from a node in layer i to a node in layer h, and that w_{jh} represents the weight from a node in layer h to a node in layer j. Bias nodes b_i and b_h are also included in the input and hidden layers, respectively, and can be implemented by increasing the input and hidden layers by one node each and using a constant input of 1 to both bias nodes.

error from this network is stated as $E = (y_j^* - y_j)$ where y_j is the value of output node j and y_j^* is the corresponding target output value. The expression for Δw_{jh} can be written more explicitly using the partial derivative of the network error E with respect to the network weights:

$$\begin{aligned}\Delta w_{jh} &= -\alpha \frac{\partial E}{\partial w_{jh}} \\ &= -\alpha \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial v_j} \frac{\partial v_j}{\partial w_{jh}} \\ &= \alpha \, (y_j^* - y_j) \, f'(v_j) \, y_h\end{aligned}$$

where $f'(v_j)$ is the derivative of the transfer function evaluated for the induced local field v_j. This weight adjustment expression can be extended for updating the weights Δw_{hi} connecting nodes from input layer i to nodes in hidden layer h and can be expressed as:

$$\begin{aligned}\Delta w_{hi} &= -\alpha \frac{\partial E}{\partial w_{hi}} \\ &= -\alpha \frac{\partial E}{\partial y_h} \frac{\partial y_h}{\partial v_h} \frac{\partial v_h}{\partial w_{hi}} \\ &= \alpha \, (y_j^* - y_j) \, f'(v_j) \, w_{jh} \, f'(v_h) \, x_i\end{aligned}$$

where v_h is the induced local field at hidden node h and x_i is the output of input node i (which is also the input to input node i). Note that the neural network used in this

work has a single output node (Fig. 11.2), and thus the network error E is computed from only this output node. This is a slight simplification of the general form of the neural network with multiple output nodes, in which case the back-propagation algorithm propagates all errors at the output nodes in layer j back to the hidden layer h.

The back-propagation algorithm can then be extended to the temporal difference algorithm with a simple modification of the weight update equations. Using the example of a game play scenario, the basic implementation of this algorithm adjusts network weights following every play during the game (i.e., iterative updates). This algorithm works by discounting state value prediction errors by a temporal difference factor λ. The general form of the TD(λ) algorithm can be stated as:

$$\begin{pmatrix} weight \\ correction \\ \Delta w_{jh} \end{pmatrix}^{(t)} = \begin{pmatrix} learning \\ parameter \\ \alpha \end{pmatrix} \times \begin{pmatrix} prediction \\ error \\ E \end{pmatrix}^{(t)} \times \sum_{k=0}^{t} \begin{pmatrix} temporal \\ discount \\ \lambda \end{pmatrix}^{t-k} \times \begin{pmatrix} local \\ gradient \\ \delta_j \end{pmatrix}^{(k)} \times \begin{pmatrix} input\ of \\ neuron\ j \\ y_h \end{pmatrix}^{(k)} \tag{11.4}$$

which adds a temporal discount factor λ and a summation from the initial time step $t = 0$ up to the current time step t. Note that from here on, the convention of using a superscript enclosed in parentheses (e.g., $y_j^{(t)}$) will be used to indicate the particular time step t of the corresponding variable. The weight updates occur at every time step during an episode up to and including the terminal time step $t = T$. The superscript-parentheses convention is used to distinguish the values of variables at a particular time step from exponents and subscripts that also may be associated with variables. Thus superscripts enclosed in parentheses are *not* exponents but merely refer to the value or a variable at a particular time step; however, superscripts that are not enclosed in parentheses are exponents, as in the case of λ^{t-k} for example.

In Eq. (11.4), the prediction error E is also modified from that for the back-propagation algorithm. This error, also known as the temporal difference error, is instead based on the difference between the predicted state values at $t+1$ and t, as well as the reward received:

$$E^{(t)} = \left(r^{(t+1)} + \gamma V^{(t+1)} - V^{(t)} \right) \tag{11.5}$$

where $V^{(t)}$ and $V^{(t+1)}$ are the predicted state values at the current and subsequent time steps, respectively, and $r^{(t+1)}$ is the reward provided for transitioning to state $s^{(t+1)}$. The values of $V^{(t)}$ and $V^{(t+1)}$ are determined by evaluating the respective state values ($\mathbf{x}^{(t)}$ and $\mathbf{x}^{(t+1)}$) through the neural network using forward propagation, where the next-state vector $\mathbf{x}^{(t+1)}$ is determined based on an action selection procedure, which is explained later in this chapter. This expression for the temporal difference error also discounts the subsequent state value $V^{(t+1)}$ by a factor γ, which serves to attenuate the value that the network is attempting to learn. In many

applications, the discount factor γ is set to 1 however, there is some support for the notion that learning improves when γ is set between 0.5 and 0.7 [6]. Note that this error expression is from where the temporal difference algorithm gets its name, as it is based on the difference in the predicted state values at two different time steps.

The general form of the TD(λ) algorithm can be more explicitly written for updating the network weights such that $w \leftarrow w + \Delta w$. The weight updates between nodes in the output layer j and nodes in the hidden layer h ($\Delta w_{jh}^{(t)}$) at time step t can be stated as:

$$\Delta w_{jh}^{(t)} = \alpha \left(r^{(t+1)} + \gamma V^{(t+1)} - V^{(t)} \right) \sum_{k=0}^{t} \lambda^{t-k} f'(v_j^{(k)}) y_h^{(k)} \tag{11.6}$$

where $f'(v_j^{(k)})$ is the derivative of the transfer function at node j evaluated at the induced local field $v_j^{(k)}$ at time step k. Equation (11.6) can then be extended to updating the weights between nodes in the hidden layer h and nodes in the input layer i ($\Delta w_{hi}^{(t)}$) at time step t as:

$$\Delta w_{hi}^{(t)} = \alpha \left(r^{(t+1)} + \gamma V^{(t+1)} - V^{(t)} \right) \sum_{k=0}^{t} \lambda^{t-k} f'(v_j^{(k)}) w_{jh}^{(t)} f'(v_h^{(k)}) x_i^{(k)} \tag{11.7}$$

A basic implementation of the TD(λ) algorithm requires only the use of Eqs. (11.6) and (11.7). Extending these equations using some relatively simple techniques however, can significantly reduce the computational cost in terms of both time and space, and can improve the efficiency of the learning algorithm.

The use of a momentum term with coefficient η can be added to Eqs. (11.6) and (11.7) in order to incorporate a portion of the weight update from the previous time step $t-1$ into that for the current time step t. This has the effect of smoothing out the network weight changes between time steps and is often most effective when training the network in a batch manner. Batch training is where weight updates are computed during every time step, but the updates are only applied after every n time steps where $n > 1$ and could extend across multiple episodes. After adding the momentum term, Eqs. (11.6) and (11.7) become, respectively:

$$\Delta w_{jh}^{(t)} = \eta \Delta w_{jh}^{(t-1)} + \alpha \left(r^{(t+1)} + \gamma V^{(t+1)} - V^{(t)} \right) \sum_{k=0}^{t} \lambda^{t-k} f'(v_j^{(k)}) y_h^{(k)} \tag{11.8}$$

$$\begin{aligned} \Delta w_{hi}^{(t)} = {} & \eta \Delta w_{hi}^{(t-1)} \\ & + \alpha \left(r^{(t+1)} + \gamma V^{(t+1)} - V^{(t)} \right) \sum_{k=0}^{t} \lambda^{t-k} f'(v_j^{(k)}) w_{jh}^{(t)} f'(v_h^{(k)}) x_i^{(k)} \end{aligned} \tag{11.9}$$

There are a few important notes about the above two equations. The error term, the difference between subsequent state values $V^{(t+1)}$ and $V^{(t)}$ (neglecting the reward term for now), is essentially the information (i.e., feedback) that is used to update network weights. Drawing from terminology of the back-propagation algorithm, $V^{(t+1)}$ can be considered the target output value y_j^*, and $V^{(t)}$ can be considered the predicted output value y_j. The network error is therefore based on the next state value $V^{(t+1)}$, in spite of the fact that this value is merely an estimate and may actually be quite different than the true state value.

The general form of a sequential decision making processes proceeds over $t = 0, 1, ..., T$. For all intermediate (i.e., non-terminal, $t \neq T$) time steps, the next-state values $V^{(t+1)}$ are available based on their predicted value after pursuing an action; an associated reward $r^{(t+1)}$ may also be provided at these time steps as well. The incorporation of rewards at intermediate time points is dependent on the specific problem. For example, some problems have well-defined subgoals that must be achieved en route to an ultimate goal. In such cases, non-zero reward values may be provided when these subgoals are achieved. In other problems such as board games, there are often no such subgoals, and thus reward values at all intermediate time steps $t = 0, 1, ..., T-1$ are 0. The temporal difference error is then only based on the difference between subsequent state values (i.e., $\gamma V^{(t+1)} - V^{(t)}$).

At the terminal time step $t = T$, there is no next state value $V^{(t+1)}$, and this value is set to 0. The reward at this time step $r^{(t+1)}$ is non-zero however, and the temporal difference error is then based on the previous state value and the reward value (i.e., $r^{(t+1)} - V^{(t)}$). For problems in which there are no intermediate rewards, it is important to note that the reward provided at time step T is the only information which is known with complete certainty, and thus this is the only true information from which the neural network can learn the values of all previously visited states.

The specific values used as rewards are often problem-dependent and have an influence on the ability of the neural network to learn the state values. In many board games, the use of rewards consisting of a 1 for a win and a -1 for a loss (and possibly a 0 for a draw) often work well. For problems that are based on a more continuous performance metric, reward values may take on values over a range (e.g., $r = [-1, 1]$) depending on the performance of the agent.

Equations (11.8) and (11.9) require that information (terms within the summation) from all previous states $s^{(0)}$, ..., $s^{(t-1)}$ be used to determine the appropriate weight adjustment at state $s^{(t)}$. At time step t, information from all previous states is merely discounted by λ. This can be exploited to reduce the number of computations required for each weight update, which can be significant for episodes with many time steps. Weight updates at time t can be made based only on information from the current time step t as well as a cumulative aggregation of the information from previous time steps up to $t-1$.

Let the summation in Eqs. (11.8) and (11.9) be called g. For each network weight there is a corresponding value of g, which is initially set to 0 at the beginning of each game. The value of $g^{(t)}$ at the current time step t can be computed by discounting its previous value $g^{(t-1)}$ by λ and adding it to the current state's weight adjustment.

The values of g for the hidden-output weights $(g_{jh}^{(t)})$ and the input-hidden weights $(g_{hi}^{(t)})$ become, respectively:

$$g_{jh}^{(t)} = f'(v_j^{(t)})y_h^{(t)} + \lambda g_{jh}^{(t-1)} \tag{11.10}$$

$$g_{hi}^{(t)} = f'(v_j^{(t)})w_{jh}^{(t)}f'(v_h^{(t)})x_i^{(t)} + \lambda g_{hi}^{(t-1)} \tag{11.11}$$

Replacing the summations in Eqs. (11.8) and (11.9) with g as in Eqs. (11.10) and (11.11), respectively, yields the following weight update equations:

$$\Delta w_{jh}^{(t)} = \eta \Delta w_{jh}^{(t-1)} + \alpha \left(r^{(t+1)} + \gamma V^{(t+1)} - V^{(t)} \right) g_{jh}^{(t)} \tag{11.12}$$

$$\Delta w_{hi}^{(t)} = \eta \Delta w_{hi}^{(t-1)} + \alpha \left(r^{(t+1)} + \gamma V^{(t+1)} - V^{(t)} \right) g_{hi}^{(t)} \tag{11.13}$$

11.3.2.2 Additional Insight into the Temporal Difference Equation

There are two ways to think about the reinforcement learning algorithm, as presented by Sutton and Barto [15]: the forward view and the backward view. The forward view focuses on the values and rewards of future states which the agent is attempting to accurately estimate. In other words, the agent is always looking at future information in order to improve its present state value estimates. In Eqs. (11.8) and (11.9), this forward-looking concept is evident in the error term that is comprised of the current state value, next-state value, and the next-state reward.

In the case of board games where all non-terminal rewards are 0, the agent adjusts its value predictions based on the error between the future state value $V^{(t+1)}$ and $V^{(t)}$ (for $t \neq T$), and thus $V^{(t)}$ evolves toward $V^{(t+1)}$. Note however, that $V^{(t+1)}$ may be erroneous due to the random initialization of the neural network weights, and thus the weight updates may also be erroneous, especially in the early stages of training. When $t = T$, the agent adjusts its value predictions based on the reward value $r^{(t+1)}$ and the current state value $V^{(t)}$, and thus $V^{(t)}$ evolves toward $r^{(t+1)}$. In this case, the value of $r^{(t+1)}$ is known with complete certainty, and thus the final weight updates are in the correct direction. When many episodes are performed, the state values at latter time points evolve and become better approximations of their true state values because of their temporal proximity to the reward value. With even more episodes, state values at early time points in the episode can better approximate their true values. The ability of the agent to approximate the true state values is dependent on many things, such as the parameters in the temporal difference equations above, and this will be shown later in this chapter.

The backward view focuses on the mechanics of the temporal differencing. More specifically, this view considers how past states at time steps $0, 1, ..., t-1$ influence learning at the current state t. This concept is evident the summation term in Eqs. (11.8) and (11.9) in that the gradient, or the potential for influencing weight updates, from past states is discounted. Thus, past states essentially have less influence on how the state value error affects the weight updates. To see this more clearly,

suppose that at time step $t = 2$, the summation in Eq. (11.8) is written explicitly as (while neglecting the momentum term for simplicity):

$$\Delta w_{jh}^{(2)} = \alpha \left(r^{(3)} + \gamma V^{(3)} - V^{(2)} \right) \left[\lambda^2 f'(v_j^{(0)}) y_h^{(0)} + \lambda^1 f'(v_j^{(1)}) y_h^{(1)} + \lambda^0 f'(v_j^{(2)}) y_h^{(2)} \right]$$

It can be seen that the effect of the prediction error $(r^{(3)} + \gamma V^{(3)} - V^{(2)})$ on the weight change $\Delta w_{jh}^{(2)}$ due to the gradient at time step $t = 0$, is discounted by λ^2, thus reducing the effect that the gradient at this time point has on the weight update.

Using the Temporal Difference Algorithm

The reinforcement learning framework, and the TD(λ) algorithm described above proceeds in a simulation-like fashion with the agent repeatedly interacting with the environment for a specified number of episodes N. This process is outlined in Algorithm 1. Recall that the state values of the current state $V(s)$ and the subsequent state $V(s')$, where $s = \mathbf{x}^{(t)}$ and $s' = \mathbf{x}^{(t+1)}$, are determined by evaluating the respective state vectors through the neural network using forward propagation.

```
Initialize neural network (initialize all weights w);
for N episodes do
    Initialize s;
    repeat for each step of episode:
        a ← action given by policy π for s;
        Take action a, observe reward r and next state s';
        E ← r + γV(s') − V(s);
        w ← w + Δw where Δw = f(E);
        s ← s';
    until s is terminal state;
end
```

Algorithm 1. The TD(λ) algorithm using an iterative weight updating scheme. Weight changes Δw are performed according to Eqs. (11.8) and (11.9).

11.3.2.3 Neural Network Settings

The neural network lies at the heart of the TD(λ) algorithm and thus the settings and parameters of the neural network can influence the ability of the network to learn accurate state value estimates. There has been a fair bit of work concerning the setting of parameters for neural networks. Some of the most effective methods are known as Efficient BackProp training methods, which are discussed in [9] and [7]. As these techniques have been found to be useful for efficiently training neural networks with the back-propagation algorithm in classification and function approximation

problems, these methods also have the potential to be useful for training using the TD(λ) algorithm. This section discusses some of the important points regarding neural network settings as they relate to the TD(λ) algorithm based on results from the literature and from empirical results.

Neural Network Architecture

As with all uses of neural networks, the architecture of the network, in terms of the number of layers and number of nodes within each layer, is a primary decision. In most applications of reinforcement learning with neural networks, a three-layer neural network is used, consisting of an input layer, a single hidden layer, and an output layer. While the number of hidden layers can be increased, it seems that this does not necessarily improve the learning of the neural network. This may be due to the fact that a single hidden layer is all that is necessary to approximate nonlinear functions with a neural network. Furthermore, while the state value function is often nonlinear, it may not be overly complex, and a single hidden layer may therefore be sufficient. Another reason for not using multiple hidden layers is due to the increased computational expense required for training the additional weights, which has not been justified by a significant increase in performance.

For the use of the TD(λ) algorithm described herein, the number of input nodes is determined by the number of elements in the state vector $\mathbf{x}$. As discussed above, there is not a unique state encoding scheme for each problem or environment, and thus the size of the state vector may vary depending the encoding scheme, which then dictates the number of nodes in the input layer. The number of output nodes is often just 1, the value of which represents the estimated state value of the corresponding input state vector. The number of hidden nodes however, can have a significant effect on the ability of the neural network to learn. In general, hidden layers with more nodes seem to perform better, and hidden layers with 40–80 hidden nodes have been used to with great success [17]. This could be due to the fact that the larger number of hidden nodes is required to accurately approximate the nonlinear state value function. Note again though, that using more hidden nodes increases the total number of weights in the network, which increases the computational expense of using such a network. A very large number of hidden nodes will also likely not improve the performance of the network significantly, as many of the weights may not contributed significantly to the state value estimates. Finally, bias nodes are often used in the input and hidden layers, which only connect to the immediate latter layer (as in Fig. 11.2).

Transfer Functions

There are a number of choices for the type of transfer functions used at the nodes of the neural network. The most common transfer function is the basic sigmoid transfer function of the form $f(v) = \frac{1}{1+e^{-v}}$, where v corresponds to the induced local

field at a particular node, as previously described. This transfer function was used for both hidden and output nodes in the network created by [17]. However, there are other transfer functions that can be used that have had similar success, either in reinforcement learning or in neural networks in general. For the hidden layers, these include the hyperbolic tangent function $f(v) = \tanh(v)$ and a modified modified hyperbolic tangent function of the form $f(v) = 1.7159\tanh(\frac{2}{3}v)$ [9]. The purpose behind the modified hyperbolic tangent function lies in the notion that the values of v stay away from the saturation region of the transfer function, and thus the derivative of the transfer function at these values is non-zero and more effectively contributes to adjusting network weights. For the output layer, the basic linear function $f(v) = v$ has been used as an alternative to the sigmoid transfer function.

There does not seem to be any single transfer function that has been found to be generally applicable across applications of reinforcement learning. One thing that is important is the pairing of the output transfer function and the reward values provided to the agent (i.e., the values which the output node is ultimately driven toward). The selection of the output transfer function should be guided by ensuring that the range over which this transfer function operates corresponds to the reward values. For example, if reward values consist of a 1 for a win and a 0 for a loss, the output transfer function should be the sigmoid transfer function which operates over $[0, 1]$. If the reward values consist of a 1 for a win and a -1 for a loss, an output transfer function should be selected that has the ability to operate over $[-1, 1]$, such as the linear transfer function or the hyperbolic transfer function.

Weight Initialization

The weights of the neural network can be initialized in a number of ways. The simplest method is to sample the network weights from a uniform distribution which extends over a small range, such as [-0.2, 0.2] as used in [20]. Sampling from a uniform distribution over a small range that is centered at zero allows for the transfer function to operate near its linear region. When this is the case, the gradients of the transfer function are large, enabling efficient learning. When the transfer function operates in the saturation regions however, the gradients are small, and this results in small weight changes [9].

LeCun et al. [9] suggest another method to initialize network weights, which is said to further increase the learning efficiency. In this method, the weights are initialized based on the number of incoming weights to each node. More specifically, for each node, weights are initialized by randomly sampling from a distribution with a mean of zero and a standard deviation of $\sigma_w = m^{-1/2}$ where m is the number of weights leading into node w. LeCun et al. also notes that the success of this method also relies on normalizing the data that are input to the network and using a modified version of the hyperbolic tangent transfer function (previously described).

Learning Rates

The learning rate parameter α can have a substantial impact on the learning and the performance of the neural network. When neural networks are used for problems such as function approximation or classification, high values of α may not allow the weights of the network to converge, whereas low values of α are more likely to allow for weight convergence, although training may be very slow. Thus, a moderate learning rate is optimal.

Another important point is that each layer should have its own learning rate. This is due to the fact that latter layers (layers closer to the network output) tend to have larger gradients than earlier layers (layers closer to the input layer). When a single value of α is used for all layers, the weights of the latter layers may change considerably while those of the earlier layers may change very little, leading to unbalanced weight changes between layers and inefficient training.

An effective method to overcome the differences in the size of the layer gradients is described in [4], and the method presented here follows a similar method. Setting the learning rates consists of a number of steps. Initially, the learning rates for all layers are set to 1. The learning rates of all layers are then increased by $n_l\sqrt{2}$ where n_l is the number of previous layers from the output layer. For example, in a 3 layer neural network, $\alpha = 1$ for the hidden-output layer and $\alpha = \sqrt{2}$ for the input-hidden layer. The final step is to scale all learning rates proportionally such that the largest learning rate is equal to $\frac{1}{n_p}$ where n_p is the number of training patterns.

For reinforcement learning, this method may not be directly applicable as n_p is dependent on the weight updating scheme, such as iterative weight updates (following every move) or batch (epoch) weight updates (following sets of moves or episodes). Additionally, if a batch size of 1 episode is used, n_p is dependent on the number of time steps within the episode. The most important detail about setting the learning rates however, is the that the ratio of the learning rates between layers must be set appropriately. Determining the largest learning rate of all layers may be determined by starting with the simple case of iterative weight updates and using trial and error while monitoring the weight changes over the course of training. When batch weight updates are used instead, the learning rates should be scaled by the number of episodes per batch update. Batch training has been found to work well in other applications of neural networks however, and this may also be beneficial for reinforcement learning provided parameters are adjusted accordingly.

11.3.2.4 TD(λ) Settings

In addition to the settings and parameters for the neural network, there are also a few settings that are specific to the TD(λ) algorithm that can also influence ability of the neural network to learn state values. The primary settings that relate to the TD(λ) algorithm include the temporal discount factor λ and the action selection policy of the agent.

Temporal Discount Factor λ

The main parameter of the TD(λ) algorithm, λ, determines how much influence previous time steps have on weight updates in current time steps. This parameter can be set over the range [0, 1]. When $\lambda = 0$, only the most recent time step $t-1$ influences the weight update at time step t, and thus information from time step t is only passed back 1 time step. When $\lambda \to 0$ but $\lambda \neq 0$, time steps prior to $t-1$ have some, though very little, influence on weight updates at time step t. Thus information from time step t is propagated backward in time very slowly. As $\lambda \to 1$ but $\lambda \neq 1$, time steps prior to $t-1$ have increasingly more influence on the weight updates at time step t, and information at time step t is propagated backward in time more quickly. Finally when $\lambda = 1$, all time steps have equal influence on weight updates at time step t, and this is considered to follow Monte Carlo behavior.

The effects of λ can be seen by considering the reward at the terminal time step T. Suppose that $\lambda \approx 0$, but $\lambda \neq 0$. If a single reward value (either positive or negative) is received at the terminal time step T, this information will not propagate backward in time much beyond time step $T-1$. In order for this reward to reach states at time steps close to the initial time step, many episode of a similar process must be followed, with the reward slowly being propagated backward through time steps with each episode. Consider a second case in which $\lambda \approx 1$, but $\lambda \neq 1$. In this case, again with a single reward provided at the terminal time step T, all time steps have nearly equal influence on future time steps, and thus information is propagation backward in time extending over many time steps. Note that in the case of board games, the only true information about the environment is provided at the end of the episode, and the information propagated backward through time at all other time steps is based on the agents' state value estimates. If the agents' state value estimates are erroneous for the entire episode, erroneous information is being propagated backward in time. This then conflicts with the true information provided as a reward at the end of the episode.

The two scenarios above lead to the notion that λ should be set somewhere between 0 and 1. Doing so attempts to balance the speed of information propagation and the amount of certainty in the information being propagated. Numerous implementations of TD(λ) to board games have suggested that using $\lambda \approx 0.7$ seems to work well (see, for example, [5, 17, 20]).

Exploration Versus Exploitation

In order for the agent to form estimations of all state values it must be able to visit each state. For the agent to learn *accurate* estimations of all state values, it must visit each state numerous times. The type of learning method for the TD(λ) algorithm described in this chapter is referred to as an *on-policy* learning method because the agent only learns about state values for those states which it explicitly visits. (Other temporal difference methods allow for the learning of state values for states which

are not explicitly visited by the agent, and these are referred to as *off-policy* learning methods.)

Learning is therefore dependent on the actions that are pursued by the agent. While the notion of action selection is absent from the weight update Eqs. (11.8) and (11.9), the action selection policy π is central to reinforcement learning and is a key step during agent-environment interaction (as in the action selection step in Algorithm 1).

A naïve action selection policy, also called a *greedy* policy, is as follows. All possible actions at time step t are evaluated by the agent by evaluating all possible next-state vectors $\mathbf{x}^{(t+1)}$ through the neural network. These next-state vectors are evaluated in order to obtain estimated state values for each possible subsequent state. The action that corresponds to the greatest next-state value is then selected and pursued. This approach is also called a pure exploitive action selection policy such that the agent always *exploits* its learned knowledge.

The greedy action selection policy is not often the most efficient policy for learning accurate estimates of all state values. This is because the agents' state value estimates are not necessarily equivalent to the true state values. The difference between the estimated and true state values results from the fact that the agents' state value estimates are dependent on the particular states visited and the rewards received (i.e., the state trajectories of the agent). Additionally, in early interactions between the agent and environment, the agent has no knowledge about the environment. In other words, its estimated state values are quite erroneous. Thus, if a greedy action selection policy is pursued all of the time, the agent will often be following a suboptimal policy because of the erroneous state value estimates. Following such an exploitative policy is likely to be a very inefficient learning strategy because it may take many episodes for the agent to realize relatively accurate state value estimates. Relating this to a neural network implementation, the agents' state value estimates are erroneous because of the randomly initialized weights between the nodes. As the agent begins to interact with the environment however, the agent gains knowledge about the environment, and this is reflected in the adjustments of the network weights leading to improved state value estimates.

Another action selection policy that could be followed is one in which the agent *explores* actions that are regarded as suboptimal based on its state value estimates. By following this policy, the agent would again evaluate all possible next-states, but it would select an action at random from the set of actions which does not include that with the greatest next-state value. This policy is referred to as an explorative action selection policy. Similar to the deficiency of following a purely exploitative action selection policy, following a purely explorative action selection policy is also likely an inefficient learning strategy because the agent never gets a chance to test its learned knowledge.

A relatively efficient learning strategy can be produced by combining the policies of action exploitation and action exploration. In such a policy, the agent largely relies on exploiting its knowledge, and does so by selecting actions that have the greatest next-state value $\varepsilon\%$ of the time. The other $(1-\varepsilon)\%$ of the time the agent uses an

explorative policy by selecting its action at random from the set of available actions. This action selection policy is referred to as an ε-*greedy* policy. This policy allows for the agent to both test its learned knowledge about the environment while also exploring actions that may be optimal despite the fact that they may be regarded as suboptimal by the agent's current knowledge.

11.4 Examples of Reinforcement Learning

Reinforcement learning can be applied to many different types of problems that can be formulated as a sequential decision making process. Many different benchmark problems have been developed in order to test the applicability and extensibility of various reinforcement learning methods. Some of these problems include gridworld, the pole-balancing problem, and the mountain-car problem. Gridworld consists of an environment represented by a 2-dimensional grid with a specific starting location, a goal location, and potential hazard locations. The goal of the agent is to navigate through the gridworld from the start to the goal while avoiding the hazards. The pole-balancing problem consists of a cart, which can only move horizontally, and that has a vertical pole attached by a hinge. The goal of this problem is to keep the pole oriented nearly vertically by moving the cart to the left or to the right. The mountain-car problem consists of a car that is attempting to climb out of a valley; however, the car does not have enough power to drive out of the valley [11]. In order to drive out of the valley, the car must drive up the opposite side of the valley to gain sufficient inertia in order achieve its goal. This problem is challenging because the car must first move away from the goal in order for it to ultimately achieve its goal.

This section however, will demonstrate how reinforcement learning performs on two board games, Tic-Tac-Toe and Chung Toi, which is a challenging variation of Tic-Tac-Toe. The implementations shown in this section will apply some of the environment, neural network, and TD(λ) settings that were previously discussed, and will explore some of the effects of these settings.

11.4.1 Tic-Tac-Toe

A base scenario for the implementation of Tic-Tac-Toe is described below, which includes all of the settings used in this scenario. The parameters and settings used in this base scenario are also provided more concisely in Table (11.1); note that parameters α, η, ε, and λ were kept constant through training. Following this base scenario, the parameter ε was changed to different values to determine its effect on the ability of the agent to learn the game.

Environment Settings

The game of Tic-Tac-Toe was implemented by first constructing the environment, which consisted of coding the structure of the environment including the board and

the rules of the game, defining a state encoding scheme, and developing an evaluation opponent. The environment consisted of a 3×3 matrix with each element corresponding to one location of the Tic-Tac-Toe board. The rules of the largely game consisted of defining the allowable actions that either player could take; i.e., players could only place pieces on open locations of the board. Additionally, players would alternate taking turns by placing pieces on the board. Note that this form of alternating play pairs well with the self-play training scheme, and self-play training was just what was used in this case. Following the placement of each piece, the board was evaluated to determine if either player had won by having three of the same piece in a row. If either player had won, the game (i.e., episode) was terminated and the corresponding reward was provided. The agent, represented by $\bigcirc$, received a reward of 1 if it had won the game, -1 if $\times$ had won the game, or 0 if the game resulted in a draw.

The state encoding scheme was the same as that previously described in Sect. 11.3.1. Briefly to review, $\bigcirc$ was encoded by a 1 and $\times$ was encoding by a -1. The 3×3 board was represented by a 9×1 vector, with one element in this vector corresponding to each location on the board. This vector represented the presence or absence of pieces at each location on the board such that a 1 or a -1 was placed in an element of this vector if a $\bigcirc$ or a $\times$ occupied the corresponding board position, respectively, or a 0 was placed if the board position was open. A 2×1 turn-encoding vector was used to indicate which player was to play the next move. If $\bigcirc$ was to play the next move, this vector was $[1,\ 0]^T$; if $\times$ was to play the next move, this vector was $[0,\ 1]^T$. The complete state vector $\mathbf{x}$ was created by concatenating the board and turn-encoding vectors, resulting in an 11×1 state vectors.

As previously mentioned, the agent was trained using the self-play training scheme. Greedy end-game moves, including greedy wins and greedy blocks, were allowed during the training games. During evaluation games, the agent was not allowed to take greedy end-game moves but instead had to rely only on its learned knowledge. The playing performance of the agent was evaluated by playing 500 evaluation games against an opponent upon initialization (0 training games) and after every 500 training games; 500 evaluation games were found to be sufficient for stable performance estimates at each evaluation session. The evaluation opponent consisted of a random opponent, such that all moves for this player were randomly selected. During both training and evaluation games, $\bigcirc$ always played the first move of the game.

Neural Network Settings

The agent was represented by a 3-layer neural network with 11 input nodes (corresponding to the 11×1 state vector), 40 hidden nodes, and 1 output node. Bias nodes were used on the input and hidden layers, which had a constant input of 1. Nodes in the hidden layer used the hyperbolic tangent transfer function and a linear transfer function was used at the output node. The weights of the network were initialized using the method by [9], previously described. The learning rates α for

each layer were set individually, with $\alpha = 0.01$ for the input-hidden layer weights and $\alpha = 0.007$ for the hidden-output layer weights. Network weights were updated using an iterative method such that they were updated after every play.

TD(λ) Settings

The temporal discount factor λ was set to 0.7, similar to that found to be successful in other implementations of reinforcement learning [17, 20]. During training, the agent pursued an exploitative action selection policy 90% of the time ($\varepsilon = 0.9$), and thus pursued what it perceived to be suboptimal actions 10% of the time.

Table 11.1. Settings for the base scenario of Tic-Tac-Toe.

General settings	
Number of games	2500
Starting player during training	○
Starting player during evaluation	○
Number of evaluation games	500
Frequency of evaluation games	500
Neural network settings	
Layers	3
Nodes per layer [i, h, j]	[11, 40, 1]
Transfer functions [h, j]	[tanh, linear]
Weight initialization method	as per [9]
Learning rate [α_{hi}, α_{jh}]	[0.0100, 0.0071]
Momentum coefficient (η)	0.0
Weight update method	iterative
TD(λ) settings	
Temporal discount factor (λ)	0.7
P[exploitation] (ε)	0.9
Environment settings	
Rewards [win, loss, draw]	[1, -1, 0]
Training methodology	self-play
Evaluation opponent	random player

The results of all scenarios are presented using performance curves obtained by playing the agent against the evaluation player over the course of training. Performance curves consisted of the % of wins, losses, and draws at each evaluation session. The curves that are shown are slightly smoothed such that the performance values at any point during training was the average of that particular value and the two adjacent values. This was done to more clearly clearly show the average performance of the agent, which can fluctuate to some degree due to the random nature of the training process.

Results for Tic-Tac-Toe

Using only the basic settings outlined in the base scenario above, the agent was able to learn to play the game of Tic-Tac-Toe against a random opponent to an acceptable level with relatively little training (Fig. 11.3). The game was learned sufficiently within about 500 games to win almost 95% of the evaluation games. Although this particular example used a trivial opponent, it serves as a good basis to ensure that the implementation is working correctly.

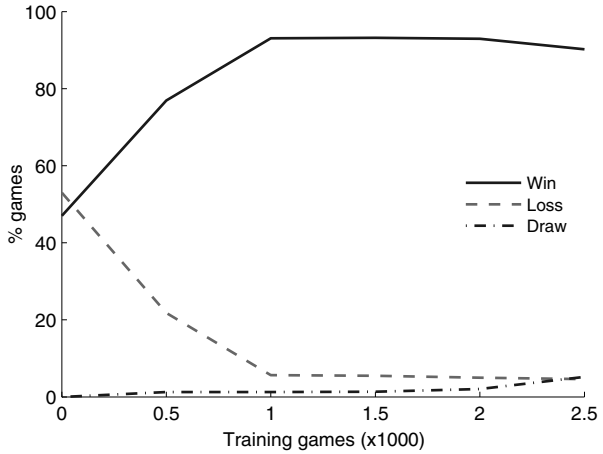

Fig. 11.3. Performance of the agent playing Tic-Tac-Toe in terms of % of wins, losses, and draws against an opponent that selects random actions.

Effect of Action Exploitation/Exploration

The effect of the agent's action selection policy was evaluated by changing the parameter ε. Recall that this parameter affects how often the agent exploits its learned knowledge versus how often it explores other actions. Four scenarios were tested with ε set to 0.1, 0.4, 0.7, and 1.0. These experiments used the same scenarios as described above such that $\bigcirc$ always selected the first move, self-play training was used, and the evaluation opponent selected random moves.

Figure (11.4) shows that the agent is able to learn the game relatively quickly and almost all evaluation games for ε = 0.1, 0.4, and 0.7. For ε = 1.0 however, the agent learns very little, which is due to it never exploring what it perceives to be suboptimal actions. An additional interesting note about these scenarios is the similarities between the cases with ε = 0.1, 0.4, and 0.7. It seems that in this type of environment with a simple game against a completely random opponent, the agent can learn very well regardless of how often it exploits its knowledge, so long as it can explore other options.

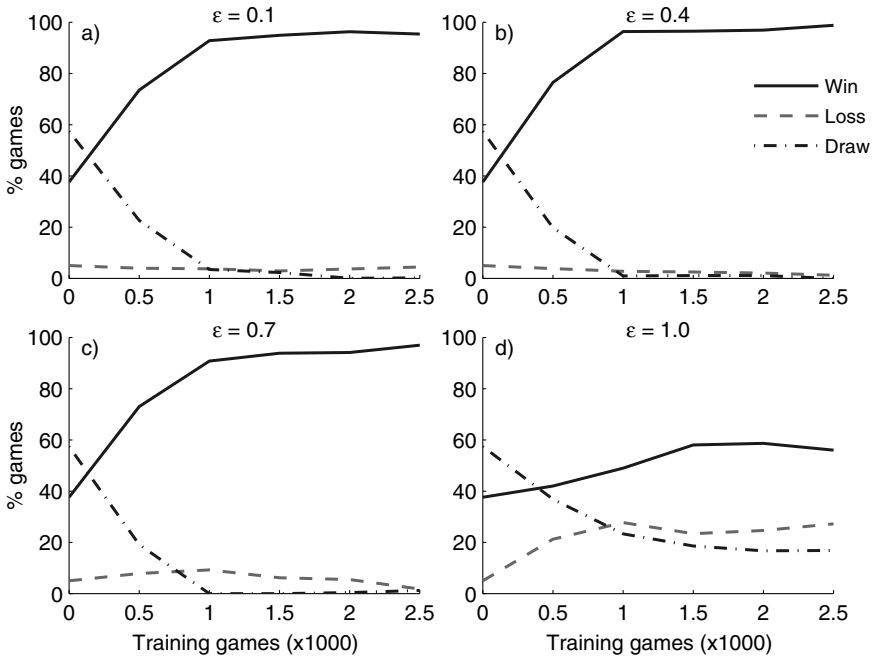

Fig. 11.4. Performance of the agent playing Tic-Tac-Toe in terms of % of wins, losses, and draws against the random player for different values of ε: **a**) $\varepsilon = 0.1$, **b**) $\varepsilon = 0.4$, **c**) $\varepsilon = 0.7$, **d**) $\varepsilon = 1.0$.

11.4.2 Chung Toi

Reinforcement learning was next applied to the game of Chung Toi, which is an extension to the game of Tic-Tac-Toe. Due to the similarities between the games, implementing Chung Toi required relatively minor modifications to the components developed for Tic-Tac-Toe. The following describes the implementation of Chung Toi, and all parameters used in the test scenarios are presented in Table (11.2). Parameters α, η, ε, and λ were also kept constant through training as with Tic-Tac-Toe.

The game of Chung Toi is played on the same 3×3 board that is used for Tic-Tac-Toe. Also similarly to Tic-Tac-Toe, the goal of the game is to get three of ones' pieces in a row along a horizontal, vertical, or diagonal of the board. One difference between Chung Toi and Tic-Tac-Toe however, is that in Chung Toi each player has only three pieces, with one player playing white pieces and the other playing red pieces. These pieces are octagonally-shaped and are labeled with arrows, and these arrows play an important role in the game: the orientation of the arrows restricts the allowable moves of the pieces.

The game of Chung Toi proceeds in two phases. The first phase consists of players placing their pieces on open board positions (alternating turns). Pieces can be placed such that their arrows are oriented either cardinally (parallel to the board axes) or diagonally. If, after each player has placed their 3 pieces on the board, neither player has three of their own pieces in a row, the second phase of play begins. In this phase, players are allowed to translate and/or rotate their pieces. Rotations can be applied to any piece on the board however, pieces may only be translated in directions which correspond to the direction in which their arrows are pointed. For example, if a piece is oriented diagonally, it may only move to an open board position that is diagonal to its current location. Additionally, if a combined move is selected that both translates and rotates a piece on move, the translation occurs before the rotation. Thus, because each player has only 3 pieces and because players can move pieces around the board, the game of Chung Toi is open-ended and there is no upper limit on the number of moves in each game (as in Tic-Tac-Toe).

Environment Settings

The environment for the game of Chung Toi was based on that of Tic-Tac-Toe and required small adaptations and extensions, the first of which was extending the state encoding scheme. As the orientation of the pieces plays an important role in Chung Toi, this information also needed to be included in the state encoding. The state vector for Chung Toi consisted of three parts: a 9×1 vector $\mathbf{p}$ indicating the presence (or absence) of the pieces; a 9×1 vector $\mathbf{r}$ indicating the orientation of the pieces; and a 2×1 vector $\mathbf{t}$ indicating which player was to play the next move. The vector $\mathbf{p}$ was similar to the position vector used in Tic-Tac-Toe where a 1 indicated the presence of a white piece (which represented the agent), a -1 indicated the presence of a red piece, and a 0 indicated an open board position. The vector $\mathbf{t}$ was the same as for as for Tic-Tac-Toe where $[1,\ 0]^T$ indicated white was to play the next move, and $[0,\ 1]^T$ indicated red was to play the next move. The vector $\mathbf{r}$ was used to encode the orientations of each piece where a 1 indicated that the corresponding piece in $\mathbf{p}$ was oriented cardinally, a -1 indicated that the corresponding piece in $\mathbf{p}$ was oriented diagonally, and a 0 indicated an open board position. The complete state vector $\mathbf{x}$ consisted of the concatenation of vectors $\mathbf{p}$, $\mathbf{r}$, and $\mathbf{t}$ into a single 20×1 state vector.

The other elements of the environment including the reward scheme and the performance evaluation method were quite similar to those of Tic-Tac-Toe. The agent (white) received a reward of 1 if it won a game and received a -1 if red won a game. In reality, the game of Chung Toi cannot end in a draw, as players keep translating and/or rotating pieces around the board until one player wins. In order to constrain the open-ended nature of the game however, the maximum number of moves allowed was 100, and games that exceed this number were considered draws, in which case the agent received a reward of 0.

Similar to Tic-Tac-Toe as well, the training scheme consisted of self-play with greedy end-game moves only allowed during training and not allowed during

evaluation. The performance of the agent was evaluated against a *'smart'* random opponent this time however. This opponent largely played randomly selected moves, but whenever possible, it could take greedy end-game moves. Thus, this type of opponent is more challenging than the pure random opponent. The agent's performance was evaluated using 500 evaluation games that were played upon initialization and following every 1000 training games. For both training and evaluation games, the starting player of each game was the agent (white).

Neural Network Settings

The neural network for the game of Chung Toi used slightly different settings than that for Tic-Tac-Toe. This was done to explore the utility of changing such settings and these changes do not necessarily indicate that these settings are superior. The input layer of the neural network used 20 nodes, corresponding to the number of elements in the state vector. The learning rates α for each layer were scaled downward compared to Tic-Tac-Toe, resulting in learning rates of $\alpha = 0.001$ for the input-hidden layer weights and $\alpha = 0.0007$ for the hidden-output layer. The magnitude of the learning rates was determined by monitoring the magnitude of the weight changes during training. The transfer function used in the hidden layer was the modified hyperbolic tangent function suggested by [9]. Neural network weights were initialized using the method in [9]. These two features were previously discussed in Sect. (11.3.2.3).

Results for Chung Toi

Figure (11.5) shows the performance of the agent against the *'smart'* random opponent. In spite of the more challenging game and environment, the agent is able to learn the game and perform very well, winning nearly 90% of the evaluation games. A *'smart'* opponent was also tested with the game of Tic-Tac-Toe although the results were much worse; the agent won approximately 55% of the games and the remainder of games largely resulted in draws. The difference in performance between the two games with the same settings may therefore be due to the differences in the environment. In the case of Tic-Tac-Toe, the environment could be considered constrained such that there is a limited number of moves per game, whereas in Chung Toi, the games are open-ended.

Effect of Temporal Discount Factor λ

The effect of the temporal discount factor λ was explored by using values of λ = 0.0, 0.4, 0.7, and 1.0, and the results under these different conditions are shown in Fig. (11.6). These results seem to indicate that, from the values evaluated, $\lambda = 0.7$ allows the agent to learn and perform best whereas very little is learned with other values of λ. This agrees with other works using the temporal difference algorithm that have also suggested that $\lambda = 0.7$ enables efficient learning [17, 20]. Thus, this particular setting for λ may be more universally applicable to different domains

Table 11.2. Settings for the base scenario of Chung Toi.

General settings	
Number of games	5000
Starting player during training	White
Starting player during evaluation	White
Number of evaluation games	500
Frequency of evaluation games	1000
Neural network settings	
Layers	3
Nodes per layer $[i, h, j]$	[20, 40, 1]
Transfer functions $[h, j]$	[modified tanh, linear]
Weight initialization method	as per [9]
Learning rate $[\alpha_{hi}, \alpha_{jh}]$	[0.0010, 0.0007]
Momentum coefficient (η)	0.5
Weight update method	iterative
TD(λ) settings	
Temporal discount factor (λ)	0.7
P[exploitation] (ε)	0.9
Environment settings	
Rewards [win, loss, draw]	[1, -1, 0]
Training methodology	self-play
Evaluation opponent	*'smart'* random player

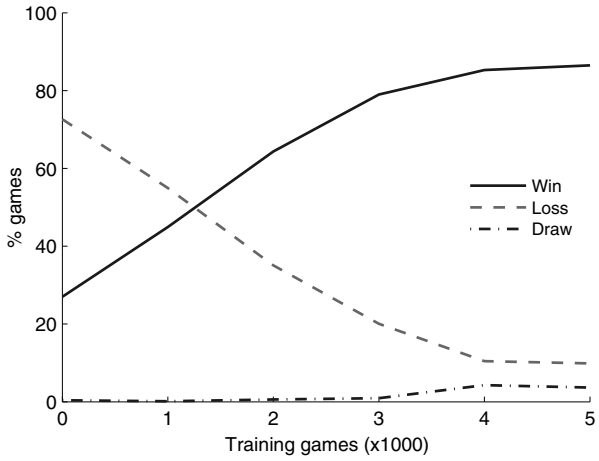

Fig. 11.5. Performance of the agent playing Chung Toi in terms of % of wins, losses, and draws against the *'smart'* random player.

and environments than other parameters. Additionally, in these scenarios evaluating λ, it was observed that the magnitude of the weight changes was proportional to the different values of λ. Intuitively, this makes sense based on the weight update equations (11.8) and (11.9). Considering this and the fact that the magnitude of weight updates are also largely based on α, it is possible that an optimal setting for α may be dependent on λ.

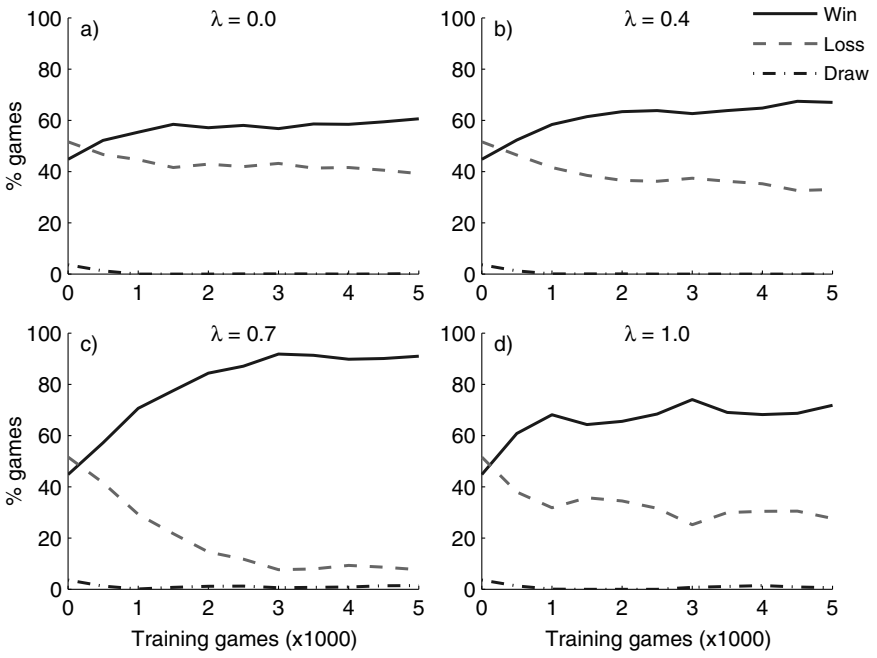

Fig. 11.6. Performance of the agent playing Chung Toi for different values of λ: **a**) $\lambda = 0.0$, **b**) $\lambda = 0.4$, **c**) $\lambda = 0.7$, **d**) $\lambda = 1.0$.

11.4.3 Applying/Extending to Other Games/Scenarios

From the above examples it should be clear that the implementation of reinforcement learning is based on a framework that entails two main entities: an agent and an environment. The agent, represented by a neural network and trained using TD(λ), is not specific to the application, but can be used in any reinforcement learning application where state value function approximation is required. Once this neural network and its learning algorithm are created, it can easily be used in many domains. The environment, on the other hand, is specific to the particular application and requires more extensive development. The major elements of the environment are listed below, along with possible adaptations that could be made when extending reinforcement learning to new domains.

State encoding: The state encoding scheme is central to any reinforcement learning implementation and it is from this that the agent (neural network) must learn. Although raw board encoding schemes were used in the example implementations in this work, the importance of a quality state encoding scheme should not be understated. Raw encoding schemes are useful for easy human interpretation however, they may not be optimal for enabling efficient learning by a neural network. Encoding schemes that include special, expert-crafted features that are relevant to the ultimate learning goal may be very useful to the neural network. A carefully constructed encoding scheme may also reduce the state space of environment, making accurate function approximation easier as well. For example, encoding Tic-Tac-Toe with features such as singlets, doublets, diversity, and crosspoints [8] results in a state vector that is invariant to the orientation of the board. In other words, symmetric board configurations (mirror images) would result in the same state vector, whereas with a raw encoding the state vectors would be unique for each board configuration and orientation. A reduced state space may then allow for the use of a smaller neural network that can be trained faster.

Action selection: The action selection procedure first determines the set of all possible actions $a_{ss'} \in A$ that can take the agent from the current state s to the next state s'. Each of these potential actions are then evaluated by passing the corresponding next-state vectors $\mathbf{x}_{s'}$ through the neural network. Finally, a particular action $a_{ss'}$ is selected based on some action selection policy.

State evaluation: The state of the domain or scenario must be continually assessed in order to determine when an episode terminates. In the games of Tic-Tac-Toe and Chung Toi, the boards were assessed after every action to determine if there were three of the same piece in a row.

Reward distribution: Rewards must be distributed based on the state of the environment. While the development of this component is not as involved as the those listed above, it is very important to the reinforcement learning framework. In the games of Tic-Tac-Toe and Chung Toi, rewards were only distributed at the terminal time step. However, there are many domains in which rewards, or some form of feedback, can be provided *during* an episode in addition to at the end of an episode. The equations for updating the weights of the neural network do not change; the reward term r in Eqs. (11.8) and (11.9) can simply be assigned a non-zero value at intermediate time steps $t = 1, 2, ..., T-1$.

Agent evaluation: In order to determine if the agent is learning, its knowledge must be tested in some manner either during or following the agent-environment interaction process. With respect to board games, and as was used in Tic-Tac-Toe and Chung Toi, opponents can be developed with different skill levels. Agent evaluation may also be performed using other methods, including other computer programs or using human experts.

Table 11.3. Variables used in text.

Variable	Description
t	Current time step
T	Terminal time step
S	Set of all states
A	Set of all actions
R	Set of all rewards
$s^{(t)}$ and s	State s at time t
s'	State s at time $t+1$
$a^{(t)}$	Action a pursued at time t
$r^{(t)}$	Reward r received at time t
$\mathcal{P}^a_{ss'}$	Probability of transitioning from state s to state s' when pursuing action a
$\mathcal{R}$	Cumulative reward received
$\mathcal{R}^a_{ss'}$	Probability of receiving a reward when transitioning from state s to state s' while pursuing action a
$V^*(s)$	True state value of state s
$V(s)$	(Perceived) state value of state s
π	Action policy
π^*	Optimal action policy
γ	Next-state discount factor
$\mathbf{x}$	State vector
$V()$	Value of state $\mathbf{x}$
i	Input layer to neural network
h	Hidden layer of neural network
j	Output layer of neural network
w_{hi}	Weight to a node in layer h, from a node in layer i
w_{jh}	Weight to a node in layer j, from a node in layer h
Δw_{hi}	Change to weight w_{hi}
Δw_{jh}	Change to weight w_{jh}
b_i	Bias node in the input layer i
b_h	Bias node in the hidden layer j
α	Learning rate parameter
E	Error term
δ_h	Local gradient at a node in layer h
y_i	Output of node in layer i; also the input to a node in layer h
y_h	Output of node in layer h; also the input to a node in layer j
y_j	Output of node in layer j
y^*_j	Target output of node in layer j
v_h	Induced local field at a node in layer h
v_j	Induced local field at a node in layer j
$f(\cdot)$	Transfer function at a node in neural network
$f'(\cdot)$	Derivative of transfer function at a node in neural network
λ	Temporal discount factor
η	Momentum parameter
g_{jh}	Summation of temporal gradient for weights connecting nodes in layer h to nodes in layer j
g_{hi}	Summation of temporal gradient for weights connecting nodes in layer i to nodes in layer h

Table 11.4. Variables used in text *(continued)*.

Variable	Description
σ_w	Standard deviation of distribution for sampling weights for initialization
m	Number of weights leading into a node
n_l	Number of layers previous to the output layer of the neural network
n_p	Number of training patterns
ε	Proportion of actions for which knowledge is exploited

The implementations presented in this work were based on the basic TD(λ) algorithm with no significant modifications. There are many settings, parameters, and extensions to this method that could be modified or added, and these changes have the potential to improve learning both in terms of efficiency and performance. It is also very likely that there are interactions between multiple settings, thus complicating the behavior of the algorithm. Parameter settings in reinforcement learning applications often seem to be domain-specific and concrete rules for specifying parameters remain to be developed.

One easily implementable extension to the basic TD(λ) algorithm is to use parameters that change dynamically over the course of training, either by simple annealing or based on the learning of the agent. One example of this could be to integrate the agent's performance into the learning process. The rationale for doing this could be that parameters of the TD(λ) algorithm, or of the neural network, may be better set based on the performance and knowledge of the agent, and thus these settings and parameters would dynamically change during the learning process. As a simple example, ε, which dictates how often the agent exploits its knowledge versus how often it explores other actions, could be set equal to the agent's performance. When the agent's performance is high, indicating that it has gained substantial knowledge, ε would similarly be high, and the agent would rely on its learned knowledge more frequently and only explore suboptimal actions occasionally. Conversely, when the agent's performance is low, indicating it has little knowledge, ε would also be low, and the agent would rely less on its knowledge and instead explore its action space more frequently with the intention of increasing its knowledge.

11.5 Summary

In summary, reinforcement learning is a powerful learning method that is ideally suited for situations which can be posed as sequential decision making problems. The use of a neural network allows for the learning of behavioral policies in environments where the state space is large, thus prohibiting the use of conventional methods such as dynamic programming. This learning method also does not rely on state transition probabilities to be known, which are often required for more explicit sequential state models such as Hidden Markov Models. The use of the temporal

difference algorithm enables the learning of associations between an agent's actions and rewards received despite the fact that rewards may be received at different points in time. The temporal difference algorithm is based on solving the temporal credit-assignment problem, which extends the credit-assignment problem solved with the back-propagation algorithm.

This work aimed to provide an introduction to the implementation of reinforcement learning using a neural network. The work is not comprehensive however, and there are many extensions and possible modifications that may enable improved learning. There are also other learning algorithms, beyond TD(λ), that can be used for reinforcement learning. Some of these algorithms include Q-learning, Sarsa, and actor-critic methods [15], all of which exploit slightly different learning strategies.

The games of Tic-Tac-Toe and Chung Toi were used to show just how reinforcement learning could be applied to a sequential decision making task. These implementations relied on the basic form of the TD(λ) algorithm with settings that have proved to be useful in other applications with neural networks. Even with this simple implementation the neural network was able to perform well in these different domains.

The use of a neural network for learning the state values for reinforcement learning may be considered a trick in itself. In addition, the Efficient BackProp techniques for training neural networks, based on the approaches of LeCun et al. [7, 9], may also be advantageous. These techniques concern the network architecture, transfer functions, weight initialization method, setting of the learning rates, as well as the use of batch (epoch) training, and these methods may allow for more efficient training in some applications. Additionally, the setting of parameters related to the temporal difference algorithm, such as the temporal discount factor λ and the action exploitation parameter ε, can also affect the training efficiency, and there are likely interactions between neural network and temporal difference algorithm parameters as well.

Future work may extend and explore the utility of the techniques discussed in this work. One potential approach that could lead to more efficient training is an epoch-based training scheme which a sliding memory stores a large number of the recently observed states (e.g., 5000), and then epoch training randomly selects a batch of states from this memory. This approach has similarities to that of database games [21] however, the use of this modified epoch training method may enable more efficient training.

Acknowledgements. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada.

References

1. Bakker, B., Schmidhuber, J.: Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. In: Groen, F., Amato, N., Bonarini, A., Yoshida, E., Kröse, B. (eds.) Proc. of the 8th Conf. on Intell., Amsterdam, The Netherlands, pp. 438–445 (2004)

2. Binkley, K.J., Seehart, K., Hagiwara, M.: A study of artificial neural network architectures for Othello evaluation functions. Trans. Jpn. Soc. Artif. Intell. 22(5), 461–471 (2007)
3. Doya, K.: Reinforcement learning in continuous time and space. Neural Comput. 12(1), 219–245 (1999)
4. Embrechts, M.J., Hargis, B.J., Linton, J.D.: An augmented efficient backpropagation training strategy for deep autoassociative neural networks. In: Proc. of the 15th European Symposium on Artificial Neural Networks (ESANN), Bruges, Belgium, April 28-30, pp. 141–146 (2010)
5. Gatti, C.J., Linton, J.D., Embrechts, M.J.: A brief tutorial on reinforcement learning: The game of Chung Toi. In: Proc. of the 19th European Symposium on Artificial Neural Networks (ESANN), Bruges, Belgium, April 27-29 (2011)
6. Ghory, I.: Reinforcement Learning in Board Games. Technical Report CSTR-04-004, Department of Computer Science. University of Bristol (2004)
7. Haykin, S.: Neural Networks and Learning Machines, 3rd edn. Prentice-Hall, New York (2008)
8. Konen, W., Bartz–Beielstein, T.: Reinforcement Learning: Insights from Interesting Failures in Parameter Selection. In: Rudolph, G., Jansen, T., Lucas, S., Poloni, C., Beume, N. (eds.) PPSN 2008. LNCS, vol. 5199, pp. 478–487. Springer, Heidelberg (2008)
9. LeCun, Y.A., Bottou, L., Orr, G.B., Müller, K.-R.: Efficient Backprop. In: Orr, G.B., Müller, K.-R. (eds.) NIPS-WS 1996. LNCS, vol. 1524, p. 9. Springer, Heidelberg (1998)
10. Mannen, H., Wiering, M.: Learning to play chess using TD(λ)–learning with database games. In: Benelearn 2004: Proc. of the 13th Belgian-Dutch Conference on Machine Learning, pp. 72–79 (2004)
11. Moore, A.: Efficient memory-based learning for robot control. PhD Thesis. University of Cambridge (1990)
12. Patist, J.P., Wiering, M.: Learning to play draughts using temporal difference learning with neural networks and databases. In: Benelearn 2004: Proc. of the 13th Belgian-Dutch Conference on Machine Learning, pp. 87–94 (2004)
13. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning internal representations by error propagation. In: Rumelhart, D.E., McClelland, J.L. (eds.) Parallel Distributed Processing, vol. 1. MIT Press, Cambridge (1986)
14. Sutton, R.S.: Learning to predict by the method of temporal difference. Mach. Learn. 3, 9–44 (1988)
15. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1988)
16. Tesauro, G.: Neurogammon: A neural network backgammon program. In: Proc. of the International Joint Conference on Neural Networks., vol. 3, pp. 33–40 (1990)
17. Tesauro, G.: Practical issues in temporal difference learning. Mach. Learn. 8, 257–277 (1992)
18. Tesauro, G.: Temporal difference learning and TD-Gammon. Communications of the ACM 8(3), 58–68 (1995)
19. Werbos, P.: Beyond regression: New tools for prediction and analysis in the behavioral sciences. Ph.D. Dissertation. Harvard University, Cambridge, MA (1974)
20. Wiering, M.A.: TD learning of game evaluation functions with hierarchical neural architectures. Master's Thesis. University of Amsterdam (1995)
21. Wiering, M.A.: Self-play and using an expert to learn to play backgammon with temporal different learning. J. Intell. Learn. Syst. & Appl. 2, 57–68 (2010)