Optimising Reinforcement learning for neural networks

Evan Hurwitz
School of Electrical and Information Engineering
University of the Witwatersrand
Johannesburg, Gauteng, South Africa
e.hurwitz@ee.wits.ac.za

Tshilidzi Marwala
School of Electrical and Information Engineering
University of the Witwatersrand
Johannesburg, Gauteng, South Africa
t.marwala@ee.wits.ac.za

Abstract – *Reinforcement learning traditionally utilises binary encoders and/or linear function approximators to accomplish its Artificial Intelligence goals. The use of nonlinear function approximators such as neural networks is often shunned, due to excessive difficulties in implementation, usually resulting from stability issues. In this paper the implementation of reinforcement learning for training a neural network is examined, being applied to the problem of learning to play Tic Tac Toe. Methods of ensuring stability are examined, and differing training methodologies are compared in order to optimise the reinforcement learning of the system. TD(λ) methods are compared with database methods, as well as a hybridised system that combines the two, which outperforms all of the homogenous systems.*

Keywords: reinforcement, learning, temporal, difference, neural, network, tic-tac-toe.

1 Introduction

Artificial intelligence (A.I.) can only truly be considered worthy of the name when the system in question is capable of learning on its own [1], without having an expert *teacher* available to point out correct behaviour. This leads directly into the paradigm of *reinforcement learning* [1]. The advantage of using such techniques for gaming A.I. is that it would allow a gaming agent to actually learn while in-game, and adapt its own play to that of the player. Most *reinforcement learning* techniques explored utilise binary system representations, or linear function approximators, which severely hinder the scope of learning available to the artificial intelligence system. One notable exception is the work by G. Tessauro on *TD-Gammon*, in which he successfully applied the TD(λ) reinforcement learning algorithm to train a neural network, with staggeringly successful results. Following attempts to emulate his work have, however, been met with failure due to the extreme difficulties of combining backpropagation with TD(λ). These difficulties, and some solutions to them, are explored in this paper, with the A.I. system being applied to learn to play the game of tic-tac-toe by playing against itself. The reason for Tic-Tac-Toe as a choice of games is deliberately because of its simplicity, as the commonly occurring problems become easier to identify within the simpler system, and solutions are then also easier to develop.

2 Background

It is necessary to understand the workings and advantages of neural networks to appreciate the task of applying them in the reinforcement learning paradigm. It is likewise important to fully grasp the implications of reinforcement learning, and the break they represent from the more traditional supervised learning paradigm.

2.1 Neural network architecture

The fundamental building-blocks of neural networks are *neurons* [2]. These neurons are simply a multiple-input, single-output mathematical function [2]. Each neuron has a number of *weights* connecting inputs from another layer to itself, which are then added together, possibly with a *bias*, the result of which is then passed into the neuron's *activation function* [2]. The activation function is a function that represents the way in which the neural network "thinks". Different activation functions lend themselves to different problem types, ranging from yes-or-no decisions to linear and nonlinear mathematical relationships. Each *layer* of a neural network is comprised of a finite number of neurons. A network may consist of any number of layers, and each layer may contain any number of neurons [2]. When a neural network is run, each neuron in each consecutive layer sums its inputs and multiplies each input by its respective weight, and then treats the weighted sum as an input to its activation function. The output will then be passed on as an input to the next layer, and so on until the final output layer is reached. Hence the input data is passed through a network of neurons in order to arrive at an output. Figure 1 illustrates an interconnected network, with 2 input neurons, three hidden layer neurons, and two output neurons. The hidden layer and output layer neurons can all have any of the possible activation functions. This type of neural network is referred to as a *multi-layer perceptron* [2], and while not the only configuration of neural network, it is the most widely used configuration for regression-type problems [3].

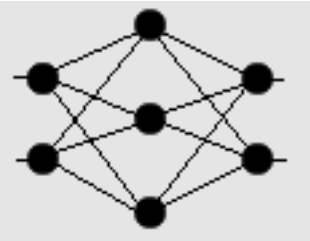

Figure 1. Sample Connectionist network

2.2 Neural network properties

Neural networks have various properties that can be utilised and exploited to aid in the solving of numerous problems. Some of the properties that are relevant to this particular problem are detailed below.

2.2.1 Universal approximators

Multi-layer feedforward neural networks have been proven to be universal approximators [2]. By this one refers to the fact that a feedforward neural network with nonlinear activation functions of appropriate size can approximate any function of interest to an arbitrary degree of accuracy [2]. This is contingent upon sufficient training data and training being supplied.

2.2.2 Neural networks can generalise

By approximating a nonlinear function from its inputs, the neural network can learn to approximate a function [2]. In doing so, it can also infer the correct output from inputs that it has never seen, by inferring the answer from similar inputs that It has seen. This property is known as *generalisation* [2]. As long as the inputs received are within the ranges of the training inputs, this property will hold [2].

2.2.3 Neural networks recognise patterns

Neural networks are often required to match large input/output sets to each other, and these sets are often noisy or even incomplete [2]. In order to achieve this matching the network learns to recognise patterns in the data sets rather than fixate on the answers themselves [2]. This enables a network to 'see' through the data points and respond to the underlying pattern instead. This is an extended benefit of the generalisation property.

2.3 Reinforcement learning

Reinforcement learning involves the training of an artificial intelligence system by trial-and-error, reflecting the same manner of learning that living beings exhibit [4]. Reinforcement learning is very well suited to episodic tasks [4], and as such is highly appropriate in the field of game-playing, where many episodes are encountered before a final result is reached, and an A.I. system is required to evaluate the value of each possible move long before a final result is achieved. This methodology allows for online learning, and also eliminates the need for expert knowledge [4].

2.3.1 Rewards and Returns

Any artificial intelligence system requires some sort of goal or target to strive towards [2], [4]. In the case of reinforcement learning, there are two such quantities that need to be defined, namely *rewards* and *returns* [4]. A reward is defined to be the numerical value attributed to an individual state, while a return is the final cumulative rewards that are returned at the end of the sequence [4]. The return need not necessarily be simply summed, although this is the most common method [4]. An example of this process can be seen below in Figure 2, where an arbitrary Markov process is illustrated with rewards given at each step, and a final return at the end. This specific example is that of a *Random walk* problem.

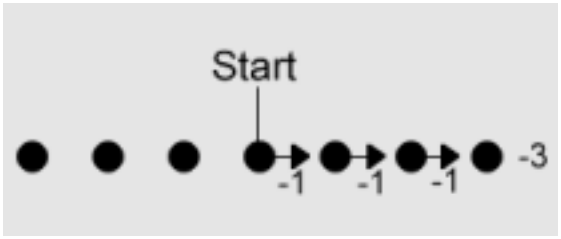

Figure 2. Random walk rewards and returns.

An A.I. system utilising reinforcement learning must learn to predict its expected return at each stage, hence enabling it to make a decision that has a lower initial reward than other options, but maximising its future return. This can be likened to making a sacrifice in chess, where the initial loss of material is accepted for the future gains it brings.

2.3.2 Exploitation vs Exploration

An A.I. system learning by reinforcement learning learns only through its own experiences [4]. In order to maximise its rewards, and hence its final return, the system needs to experiment with decisions not yet tried, even though it may perceive them to be inferior to tried-and-tested decisions [4]. This attempting of new approaches is termed *exploration*, while the utilising of gained knowledge to maximise returns is termed *exploitation* [4]. A constant dilemma that must be traded off in reinforcement learning is that of the choice between exploration and exploitation. One simple approach is the *ε-greedy* approach, where the system is *greedy*, ie attempts to exploit, in every situation with probability ε, and hence will explore with probability 1- ε [4].

2.3.3 TD (λ)

One common method of training a reinforcement learning system is to use the TD (λ) (Temporal Difference) algorithm to update one's value estimates [4] [5]. This algorithm is specifically designed for use with episodic tasks, being an adaptation of the common Widdrow-Hoff learning rule [5]. In this algorithm, the parameters or *weights w* to be altered are updated by equation (1) [5].

$$\Delta w = \alpha\left(P_{t+1} - P_t\right)\sum_{k=1}^{t} \lambda^{t-k} \nabla w P_k \qquad (1)$$

The prediction P_{t+1} is used as a target for finding the error in prediction P_t, allowing the update rule to be computed incrementally, rather than waiting for the end of the sequence before any learning can take place [5]. α and λ are the learning rate and weight-decay parameters, respectively.

2.4 Neural network advantages

Neural networks have a number of advantages that can be exploited in order to optimise a reinforcement learning A.I. system. Some of these advantages are:

- Faster learning. Due to the generalisation property of neural networks, learning from one position can be generalised to learn for all similar positions.
- Positional understanding. As a result of the pattern recognition property of neural networks, the system can learn to judge positions, rather than simply remember individual state configurations.

For these reasons it is desirable to utilise a neural network as a function approximator for a reinforcement learning system.

3 Implementation

In this section we detail the implementation of the problem domain and the various methods of optimising the neural network reinforcement learning, as well as the performance measures used to evaluate the usefulness of each presented method.

3.1 Tic Tac Toe

The game of *Tic Tac Toe*, or *noughts and crosses*, is played on a 3x3 grid, with players taking alternate turns to fill an empty spot [6]. The first player places a 'O', and the second player places a 'X' whenever it is that respective player's turn [6]. If a player manages to get three of his mark in a row, he wins the game [6]. If all 9 squares are filled without a winner, the game is a draw [6]. The game was simulated in Matlab, with a simple matrix representation of the board.

3.2 Player evaluation

The A.I. system, or *player*, needs to be evaluated in order to compare different players, who have each learned using a different method of learning. In order to evaluate a player's performance, ten different positions are set up, each with well-defined correct moves. Using this test-bed, each player can be scored out of ten, giving a measure of the level of play each player has achieved. Also important is the speed of convergence – ie how fast does each respective player reach its own maximum level of play.

3.3 TD(λ) for backpropogation

In order to train a neural network, equation (1) needs to be adapted for use with the backpropogation algorithm [7]. The adaptation, without derivation, is as follows [7]:

$$w_{ij}^{t+1} = w_{ij}^{t} + \alpha \sum\nolimits_{K \in O} \left(P_K^{t+1} - P_K^{t}\right) e_{ijk}^{t} \qquad (2)$$

where the eligabilities are:

$$e_{ijk}^{t+1} = \lambda e_{ijk}^{t} + \delta_{kj}^{t+1} y_i^{t+1} \qquad (3)$$

and δ is calculated by recursive backpropogation

$$\delta_{ki}^{t} = \frac{\partial P_k^{t}}{\partial S_i^{t}} \qquad (4)$$

as such, the TD(λ) algorithm can be implemented to update the weights of a neural network [7].

3.4 Stability issues

The TD(λ) algorithm has proven stability for linear functions [6]. A multi-layer neural network, however, is non-linear [2], and the TD(λ) algorithm can become unstable in some instances [4] [8]. The instability can arise in both the actual weights and in the predictions [4] [8]. In order to prevent instability, a number of steps can be taken, the end result of which is in most cases to limit the degree of variation in the outputs, so as to keep the error signal small to avoid instability.

3.4.1 Input/Output representation

The inputs to, and outputs from, a typical A.I. system are usually represented as real or integer values. This is not optimal for TD(λ) learning, as the values have too much variation. Far safer is to keep the representations in binary form, accepting the dimensionality trade-off as a fair price to ensure a far higher degree of stability. Specifically in the case of the outputs, this ensures that the output error of the system can never be more than 1 for any single output, thus keeping the mean error to within marginally stable bounds.

For the problem of the Tic Tac Toe game, the input to the network is an 18-bit binary string, with the first 9 bits representing a possible placed 'o' in each square, and the second 9 bits representing a '1' in each square. The output of the network is a 3-bit string, representing a 'o' win, a draw and an 'x' win respectively.

3.4.2 Activation Functions

As shown in section 2, there are many possible activation function that can be used for the neural network. While it is tempting to utilise activation functions that have a large scope in order to maximise the versatility of the network, it proves far safer to use an activation function that is limited to an upper bound of 1, and a lower bound of zero. A commonly used activation function of this sort is the sigmoidal activation function, having the form of:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{5}$$

This function is nonlinear, allowing for the freedom of approximation required of a neural network, and limiting the upper and lower bounds as recommended above. While this activation function is commonly used as a middle-layer activation function, it is unusual as an output layer activation function. In this manner, instability is further discouraged.

3.4.3 Learning rate

As the size of the error has a direct effect on the stability of the learning system, parameters that directly effect the error signal also have an effect on the said stability. For this reason, the size of the learning rate α needs to be kept low, with experimental results showing that values between 0.1 and 0.3 prove safe, while higher values tend to become unstable, and lower values simply impart too little real learning to be of any value.

3.4.4 Hybrid stability measures

In order to compare relative stability, the percentage chance of becoming unstable has been empirically noted, based upon experimental results. Regardless of each individual technique presented, it is the combination of these techniques that allows for better stability guarantees. While no individual method presented gives greater than a 60% stability guarantee (that is, 60% chance to be stable given a 100-game training run), the combination of all of the above measures results in a much better 98% probability of being stable, with minor tweaking of the learning rate parameter solving the event of instability occurring at unusual instances.

4 The players

All of the players are trained using an ε-greedy policy, with the value of ε = 0.1. i.e. for each possible position the player has a 10% chance of selecting a random move, while having a 90% chance of selecting whichever move it deems to be the best move. This selection is done by determining all of the legal moves available, and then finding the positions that would result from each possible move. These positions are sequentially presented to the player, who then rates each resultant position, in order to find the best resultant position. It obviously follows that whichever move leads to the most favoured position is the apparently best move, and the choice of the player for a greedy policy. The training of each player is accomplished via *self-play*, wherein the player evaluates and chooses moves for both sides, learning from its own experiences as it discovers errors on its own. This learning is continued until no discernable improvement occurs.

As a benchmark, randomly initialised networks were able to correctly solve between 1 and 2 of the posed problems, beyond which one can say genuine learning has indeed taken place, and is not simply random chance.

4.1 Player #1 – Simple TD(λ)

Player #1 learned to play the game using a simple TD(λ) backpropogation learning algorithm. This proved to be very fast, allowing for many thousands of games to be played out in a very short period of time. The level of play achieved using this method was however not particularly inspiring, achieving play capable of solving no more than 5 of the 10 problems posed in the rating system. The problem that is encountered by player #1 is that the learning done after each final input, the input with the game result, gets undone by the learning of the intermediate steps of the next game. While in concept the learning should be swifter due to utilising the knowledge gained, the system ends up working at cross-purposes against itself, since it struggles to build its initial knowledge base, due to the generalisation of the neural network which is not present in more traditional reinforcement learning arrangements.

4.2 Player #2 – Historical database learning

In this instance, the player learns by recording each position and its corresponding target, and storing the pair in a database. Duplicate input data sets and their corresponding targets are removed, based on the principle that more recent data is more accurate, since more learning has been done when making the more recent predictions. This database is then used to train the network in the traditional supervised learning manner. A problem encountered early on with this method is that early predictions have zero knowledge base, and are therefore usually incorrect. The retaining of this information in the database therefore taints the training data, and is thus

undesirable. A solution to this problem is to limit the size of the database, replacing old data with new data once the size limit is reached, thus keeping the database recent. This methodology trains slower than that employed to train player #1, making long training runs less feasible than for TD(λ) learning. The play level of this method is the lowest of those examined, able to solve only four of the ten proposed problems. Nonetheless, the approach does show promise for generating an initial knowledge base from which to work with more advanced methods.

4.3 Player #3 – Fact/Opinion DB learning

Building on the promise of Player #2, a more sophisticated database approach can be taken. If one takes into account the manner of the training set generation, one notes that most of the targets in the database are no more than *opinions* – targets generated by estimates of the next step, as seen in equation (1) – while relatively few data points are in fact *facts* – targets generated by viewing the end result of the game. In order to avoid this problem, the database can be split into two sub-databases, with one holding facts, and the other holding opinions. Varying the sizes, and the relative sizes, of these two sub-databases can then allow the engineer to decide how much credence the system should give to fact versus opinion. This method proved far more successful than Player #2, successfully completing 6 of the 10 problems posed by the rating system. It's speed of convergence is comparable to that of Player #2.

4.4 Player #4 – Widdrow-Hoff based DB learning

In this instance, a very similar approach to that of Player #2 is taken, with one important distinction: Instead of estimating a target at each move, the game is played out to completion with a static player. After each game finishes, the player then adds all of the positions encountered into its database, with the final result being the target of each position. This means that no *opinions* can ever enter into the training, which trades off speed of convergence for supposedly higher accuracy. This method is not optimal, as it loses one of the primary advantages of reinforcement learning, namely that of being able to incorporate current learning into its own learning, hence speeding up the learning process. Unsurprisingly, this method trains with the same speed as the other database methods, but takes far longer to converge. It achieves a similar level of play as does Player #1, being able to solve 5 of the posed problems.

4.5 Player #5 - Hybrid Fact/Opinion DB TD(λ) learning

The logical extension to the previous players is to hybridise the most successful players in order to compensate for the failings of each. Player #5 thus utilises the Fact/Opinion database learning in order to build an initial knowledge base from which to learn, and then proceeds to learn from thence using the TD(λ) approach of Player #1. This proves more successful, since the intrinsic flaw in player #1's methodology lies in its inability to efficiently create a knowledge base, and the database method of player #3 creates that knowledge base from which to learn. Player #5 begins its learning with the expected sluggishness of database methods, but then learns much faster once it begins to learn using the TD(λ) approach. Player #5 managed to successfully solve seven of the ten problems once trained to convergence. The problem of unlearning learned information is still apparent in Player #5, but is largely mitigated by the generation of the initial knowledge base.

4.6 Player comparisons

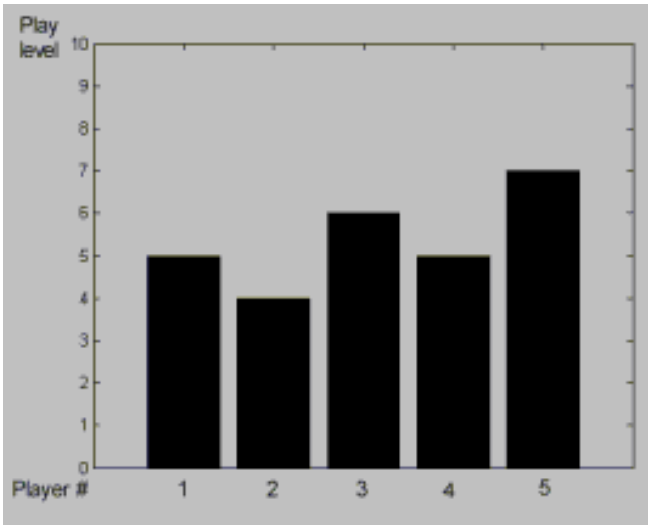

Figure 3. Relative player strengths

As is illustrated in figure #3, the hybrid method learns to play at the strongest level of all of the methods presented. Due to the drastic differences in speed and computational power requirements, it is preferable to stay away from database-based methods, and it is thus worth noting that only the fact/opinion database method arrives at a stronger level of play than the simple TD(λ)-trained player #1, and that this methodology can easily be incorporated into a TD(λ) learning system, which produces the far more promising player #5. The fact that after a short knowledge-base generation sequence the hybrid system uses the highly efficient TD(λ) approach makes it a faster and more reliable learning system than the other methods presented. As can be seen in Figure 3, however, there is still a greater level of play strength that should be achievable in this simple game, and that has been limited by the unlearning error seen in Players #1 and #5.

For future work, it is recommended that research goes into a more efficient method of calculating eligability traces, since a better calculation of the eligibility traces would solve the problem of unlearning encountered in the TD(λ) methods, removing the play-level limit that has been encountered.

5 Conclusion

While the stability of TD(λ) methods for neural networks is in question, many precautions have been presented that greatly improve the stability of the neural network learning process. Far more important is the generalisation property of neural networks. The generalisation that is the great boon of neural networks also is its greatest weakness when applied to reinforcement learning. The ability to generalise comes at the cost of updating all position prediction with each individual prediction, which limits the development of a knowledge base. Database building techniques are useful, but also lose some of the benefits of reinforcement learning, and are hence undesirable. Lastly, the usage of database techniques to develop an initial database, followed by further learning utilising the TD(λ) approach leads to the best player in terms of player strength, illustrating the need for a teacher to 'show the ropes' to an A.I. system, before it is capable of learning on its own.

For future work, the issue on *unlearning* needs to be further explored and solved before neural network reinforcement learning is in fact a viable tool for use in gaming A.I., although once this is accomplished, the scope of the method holds great promise to allow truly adaptable A.I. agents in games, able to give players a true challenge.

References

[1] Sutton, R. S. *What's wrong with Artificial intelligence*, Last viewed 24/08/2005 http://www.cs.ualberta.ca/~sutton/IncIdeas/WrongWithAI.html 11/12/2001.

[2] Wesley Hines J. *Matlab supplement to Fuzzy and Neural Approaches in Engineering.* John Wiley & Sons Inc. New York. 1997.

[3] Miller, Sutton and Werbos. *Neural Networks for Control.* MIT Press. Cambridge, Massecheussettes. 1990

[4] Sutton, R. S, Barto A. G. *Reinforcement learning: An Introduction*, MIT press, 1998.

[5] Sutton, R. S. *Learning to predict by the methods of temporal differences.* Mach. Learning 3, (1988), 9-44.

[6] Morehead A. H, Mott-Smith G, Morehead P. D. *Hoyle's Rules of Games, Third revised and updated edition*. Signet book. 2001.

[7] Sutton, R. S. *Implementation details of the TD(λ) procedure for the case of vector predictions and Backpropogation*. GTE laboratories technical note TN87-509.1, 1989.

[8] Sutton, R S. *Frequenlty asked questions about reinforcement learning*, last viewed 24/08/2005, http://www.cs.ualberta.ca/~sutton/RL-FAQ.html