REINFORCEMENT LEARNING OF NEURAL NETWORKS FOR CREATING HUMAN-LIKE POOL PLAYERS

Daniel Angelov

MEng Robotics, d.angelov@student.rdg.ac.uk

ABSTRACT

The objective of this research was to investigate the possibility of using Neural Networks to create artificial pool players that solve the game in the same way that humans would. In the base of the learning algorithm is positive and negative reinforcement on a back-propagation learning neural network. The study showed what the most appropriate format for representing the table is and how to create and interpret the output of the network. It highlights the main problems of the system such as foul-turns and oscillations into local minima and gives solutions to counteract them.

1. INTRODUCTION

The game of pool is geometrically solvable problem, thus placing it in the super-human to optimal scale of AI classification in terms of computing sequence of shots. As such it is difficult to create an appropriate system to match the level of ranging gaming ability of a group of people. By using an adaptive method for the computer players, they can rank and learn the game in the same time scale as humans, leading to better interaction. Previous research has shown that board games, such as Backgammon [9], Checkers [5], Go [6] and even chess [7] can be solved using a variety of combinations between machine learning, databases and searching algorithms. The games were solved with different success due to their nature and the size of the search space.

The problem of creating an artificial pool player was researched in a computer simulated environment. The learning algorithm can be described as starting from a blank network, using the outputs as a hint, where the player should look for a solution. With time, the neural network becomes more confident of its abilities and starts to follow the generated hints.

Due to the large space of possible ball positions and the great difference for a small change, the simulation was run extensively on several machines.

2. APPLICATION STRUCTURE

The application was created with special care to scalability. The three main areas of development are as follows – the Artificial Neural Network (ANN); the Simulator that hosts the physics engine; and the communication infrastructure that enabled to transport information between the ANN and the simulator.

2.1. Artificial Neural Network

The aim was to accomplish the task with minimum amount of preprocessing on the data. The used Multilayer Perceptron (MLP) is a 3-layered fully connected feed-forward network [1][12]. The internal structure was changed to accommodate different possibilities for viewing the problem and assessing what action parameters should the network be outputting. These variations will be discussed in the following subsections.

2.1.1. Back-propagation

Every neuron is governed by a sigmoid activation function:

$$\phi(\sigma_i) = (1 + e^{-\sigma_i})^{-1} \ where\ \sigma_i = \sum_{i=0}^{n} (\omega_i * r_i) + b_i$$

where $\phi(\sigma_i)$ is the output of the ith neuron and σ_i is the weighted sum of the n-dimensional input vector. The w_i term is the weight (strength of the connection) with the ith input and b_i is the bias weight of that neuron. The result is a non-linear S-shaped curve defined in the range from 0 to 1 [11].

The basic method to adjust the output of a neuron is to calculate the error e_i between the target t_i and the actual y_i and to feed it backwards in the network. The used back-propagation algorithm tries to minimize the sum E of the squared error e_i by changing the weights of the neuron in the direction shown by the derivative $\partial E/\partial \omega_{ij}$. This way for the output layer:

$$\frac{\partial E}{\partial \omega_{ij}} = e_i\, \phi'(\sigma_i) o_j \ \ where\ E = \frac{1}{2}\sum_{i=0}^{n}(t_i - y_i)^2$$

where $\phi'(\sigma_i)$ is the first derivative of the sigmoid function. Using gradient decent, the change of weight Δw_{ij} can be found as

$$\Delta w_{ij}(t) = \eta \frac{\partial E}{\partial \omega_{ij}} + \alpha \Delta w_{ij}(t-1)$$

where η is the learning rate parameter in the range [0..1]. In the current application it was varied between 0.2 and 0.8. The $\alpha \Delta w_{ij}(t-1)$ expression defines momentum – a way to increase convergence speed by adding a weighted term of the last weight change. The momentum parameter $0 \leq \alpha \leq 1$ controls how much the gradients are averaged over time. The momentum value α was varied between 0.6 and 1.

For a hidden layer $\partial E/\partial \omega_{ik}$ is defined as a weighted sum of the delta for the next layer:

$$\frac{\partial E}{\partial \omega_{ik}} = \left[\sum_{i=0}^{n} \frac{\partial E}{\partial \omega_{ij}} \omega_{ij}\right] \phi'(\sigma_i)$$

As a result, the error from the output layer propagates backwords, adjusting the weights of the connections, thus giving the name of the algorithm.

2.1.2. Reinforcement Learning

Markov decision process [10][12] are used to model the game of pool - in each state s, there is an action a, that leads to a state s' using the state-transition property $P_a(s, s')$ and gives a reward of $R_a(s, s')$. This way, each next state depends only on the previous state and the action taken. Even though in the game of pool humans predict where the ball will stop, as to increase the probability of an easy shot in the next state, action feedback is not applied to the neural network to prevent it from being recurrent and decrease the searched state-space - each turn is played on its own.

Q-learning is a reinforcement type technique for learning, based on Markov decision process that does not require an internal model of the system [11][13]. It can be viewed as the agent, the possible states S, and sets of actions A for each state s, $s \in S$. It learns by extracting an action-value function, that gives an expected utility function $U(s, a)$ of a given action a, moving the agent into a state s'. After each action, it updates its policy P, by which actions from A are selected. The goal of the agent is to maximize its total reward. It does this by learning which action is optimal for each state.

One way of implementing the above, which was used in the experiment, is an Artificial Neural Network to approximate the policy function [2][3][10][12]. A variant was first shown to work on a Backgammon game [9], playing with temporal difference (TD) learning [4]. In games, such as pool, the result of an action, is only acquired after it has been generated e.g. simulation-time, which yields another change in the standard structure.

The Q-learning implementation used, differs in the following ways. The reward function does not associate different states to different rewards, but generates a reward based on the movement of the agent from state s to s'. In other words, a state of the pool table does not give a score, but the action of potting a ball (moving to another state) - gives.

Another variation is how the desired action is chosen so that the agent goes to the aimed state. Based on this action, the policy should be updated, meaning the neural network trained - with the desired action-values. The two experiments involved agents that are trained through positive and negative reinforcement.

Positive reinforcement - Initially a randomly generated policy is created. Using random walks [14] occurring proportionally to the confidence of the policy, such actions are sometimes taken, so that the change of state $s \rightarrow s'$ gives a positive score. These actions are then feed backed to update the policy and such the neural network.

Negative reinforcement - The core of this method is the same as the positive reinforcement. The difference occurs when after any action a, the change of state $s \rightarrow s'$, results in a neutral or negative score. In such situations, an expert system can introduce an action a' that would have had a higher probability of moving the agent to a state s'' with a positive reward. It was experimented with the expert system giving knowledge only when the score was negative (e.g. potting the white ball) and when the result could be neutral (e.g. the cue ball did not hit any red balls, but stayed on the table). The expert system used was simplistic and generated an action a' that would have made the white ball hit a red ball currently positioned on the table.

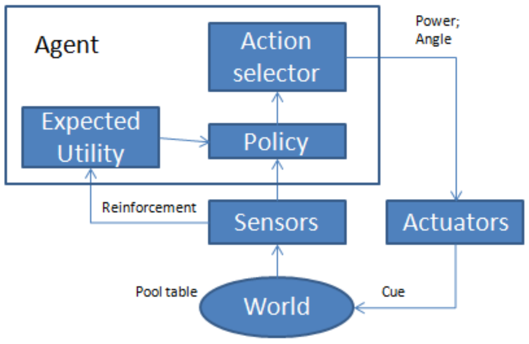

Figure 1. Agent structure.

2.1.3. Input vector

The input vector to the network was varied significantly to accommodate different levels of abstraction of the input data. The simulation streamed a top view of the table as in Fig. 2 with resolution of 256x128 pixels.

The first experiment stage was to input a sequence of normalized picture's BGR values. The image has 2^{15} pixels, each with 3 channels, resulting in the input vector being with the size of 98.3k values. The huge length of the input vector limited the size of the neural network that can be constructed to contain only 3000 neurons, a constrain imposed by the 32-bit memory address of the available hardware. This undoubtedly resulted into under-fitting the data.

The image then was converted to grayscale using the following formula:

$$Gray_{xy} = 0.3 * Blue_{xy} + 0 * Green_{xy} + 0.7 * Red_{xy}$$

It is using the fact that the green background doesn't give any additional information to the position of the red and white balls. This increased the number of neurons to

9000. However, this was still not enough for such a small network to extract key features.

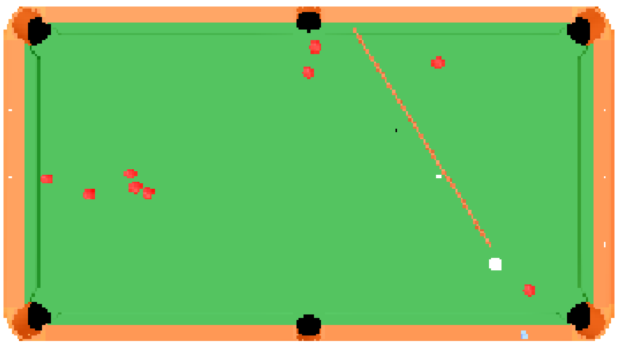
Figure 2. Top view of the pool table.

Therefore, instead of using a raw image, the simulation extracted the 3 dimensional position of each of the 15 red and one white ball. In this experiment the input vector was constructed by the normalized x_r, y_r position of the balls and a third value {0, 1}, stating whether it was potted or not (z-coordinate below or above table top). This created a much better representation of the problem. However, the influence of the white ball (x_w, y_w) was too small, and resulted in very low change in the output. This problem was viewed from a mathematical standpoint and resulted into generating the input vector from tuples containing the following information for each red ball:

$$t_i = (x_{ri}, y_{ri}, \{0,1\}, x_w, y_w)$$

This way, the white ball weight in the output vector is increased proportionally to the red ones, at the cost of an increase input vector size of 15*5 = 75.

2.1.4. Output vector

The output of the neural network is the power of the shot in range [0..1] and the angle α of the cue at which the white ball should be hit normalized to the range 0 to 2π. This resulted in a network that preferred playing in either the left or right parts of the pool table.

The core of the problem was that the angle output was not cyclic – the value of 0 was different for the network from 2π. To resolve it, an additional output was added. The last two values (o_2, o_3) would create a vector, the angle of which is used:

$$\alpha = \cos^{-1} {o_2}\Big/\sqrt{o_2^2 + o_3^2} = \sin^{-1} {o_3}\Big/\sqrt{o_2^2 + o_3^2}$$

The actual power and angle, sent to the simulation are values, picked from a Gaussian distribution with a mean value equal to the output of the network and variance proportional to the rank/confidence of the AI player.

2.1.5. Final Network Structure

The Neural Network that generated the best results had the following structure:

75 value input vector – described in section 2.1.3;
3 hidden layers – 150, 100, 20 neurons each;
3 value output vector – described in section 2.1.4;

2.2. Simulation

The simulations were created in V-REP – Virtual Robot Experimentation Platform. It encapsulated the ODE physics engine. The scene in which the game was simulated consisted of a static real size pool table, a cue, 15 red balls and one white.

A custom plug-in was written for the simulator. It managed arranging the appropriate balls on their places, listened for player turns information (power and angle), executed them and calculated the score of the turn – hitting a red ball without potting it resulted in 0.5 points, potting a red – 10 points, potting the white ball – penalty of -8 points.

2.3. Communication Infrastructure

The communication between the Neural Network and the simulator was done using ROS (Robot Operating System). It uses nodes as communication instances. Nodes transfer information using services (request, answer) and topics (streaming).

The V-REP simulator loads the custom plug-in that creates a node which advertises the position of the red and white balls and a top view of the table; and the score of the last turn. The Neural Network player subscribes to the topics and publishes the calculated turn. A graph of the structure can be seen on Fig. 3.

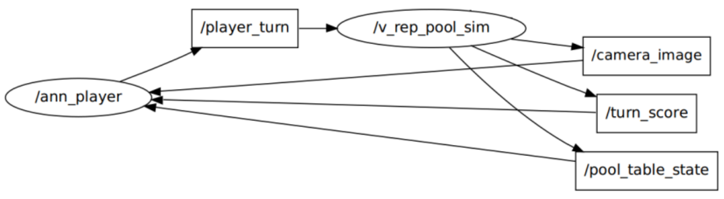

Figure 3. Communication graph between the ANN player and the V-REP simulator.

3. RESULTS

The training duration of the Neural Network had a dramatic influence on its abilities to play. It decreased the time needed to complete a game by a half in just 3 games as shown in Fig. 4. Special care was taken to ensure that at the first stage, the rank of the network will allow the Gaussian output to vary over the whole range of possibilities insuring all kind of strategies could be learnt. Networks that included negative reinforcement learning completed the game faster. However, their strategy resulted into using lots of low power shots that leaded to not hitting any other ball.

The main associated problem with the current strategy was the amount of fouls a network performed – more precisely potting the white ball. From table 1 it can be seen that introducing an expert system decreases the number of shots resulting to the cue ball in a pocket to 17-23%. This percentage is considered high even for human players. Additionally, that increases the amount of fouls described by failing to hit any balls.

Network	Accumulative white ball potted	White ball potted in last game
Positive	43%	26%
Negative	23%	17%
Negative	25%	20%

Table 1. Showing relationship between type of training and percentage of potted cue balls.

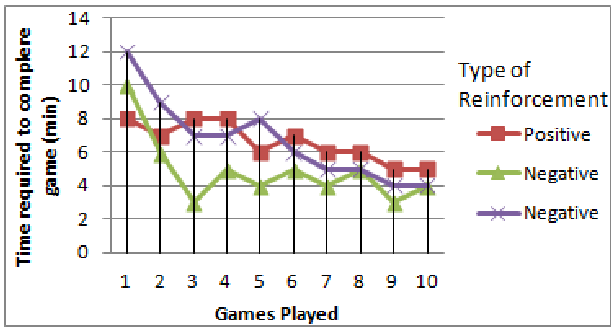

Figure 4. Graph of the number of games played vs. the time it takes to finish a game.

Another observed pitfall occurred when the neural network falls in local minima – executes in oscillatory fashion shots that previously resulted in a positive score. Positive reinforcement learning networks are prone to such effect. One of these examples is potting the cue ball from its starting position. This would result in a negative score and would move the ball again to the initial place. If the confidence of the network is low, the variance of the output would lead to a different shot and pivot the search space in a different direction.

It is likely that an only positive reinforcement network will outperform a negative reinforcement system because of its inherited ability to learn on its own.

4. DISCUSSION

The accumulated results and the dimensionality of the problems suggests that the algorithm needs to play at least 5000 well spaced shots to start approximating correctly the utility function.

Adding Negative Reinforcement to the learning algorithm decreases the number of potted white balls. It also decreases the power of the shots, thus forcing more accurate ones. However, final results on the time spent to finish a game are within range of the only positive learning.

Advantage of the Positive Reinforcement is that the search space is traversed in a stochastic manner, meaning the network can express a more "creative" way to pot a ball – e.g. using a border of the table, compared to the other option, where the expert forces a direct shot. As a result, this configuration can outperform the negative reinforcement in the long term, unless a better expert system is put as feedback.

5. CONCLUSION

In the search of creating a human-like pool player, the option of using positive reinforcement Q-learning would suggest a valid solution. The Neural Network player can continue to learn while playing against its human competitors.

6. REFERENCES

[1] T. Kwok and D. Yeung, Constructive algorithms for structure learning in feedforward neural networks for regression problems problems: A survey. IEEE Transactions on Neural Networks, 8(3):630–645.

[2] C. Touzet, *"Neural Reinforcement Learning for Behaviour Synthesis, "*, *Robotics and Autonomous Systems*, Special issue on Learning Robot: the New Wave, N. Sharkey Guest Editor, 1997.

[3] S. Sehad & C. Touzet, "Reinforcement Learning and Neural Reinforcement Learning," *ESANN 94*, D-Facto publication, Brussels, April 1994.

[4] Richard S. Sutton (1988) Learning to predict by the methods of temporal difference. Machine Learning 3, 9-44.

[5] Arthur L. Samuel (1959) Some studies in machine learning using the game of checkers. IBM Journal of Research and Development 3, 210-229.

[6] Peter Dayan, Nicol N. Schraudolph, and Terrence J. Sejnowski (2001) Learning to evaluate Go positions via temporal difference methods. Computational Intelligence in Games, Springer Verlag, 74-96.

[7] Henk Mannen and Marco Wiering (2004) *Learning to play chess using TD-learning with database games*. Benelearn conference on Machine Learning.

[8] Imran Ghory (2004), Reinforcement Learning in board games.

[9] Gerald Tesauro (1995) *Temporal Difference Learning and TD-Gammon*. Communications of the ACM 38, 58-68.

[10] Martin Riedmiller, Neural Reinforcement Learning to Swing-up and Balance a Real Pole

[11] L-J. Lin, "Reinforcement Learning for Robots Using Neural Networks," PhD thesis, Carnegie Mellon University, Pittsburgh, January 1993.

[12] Gavin A. Rummery, "Problem solving with reinforcement learning", PhD thesis, Cambridge University Engineering Department, Cambridge, July 1995, pp. 13-49.

[13] C. Watkins and P. Dayan, Q-learning, Machine Learning, 8 (1992), pp. 279–292.

[14] K.P. Lam (2005), "Learning Optimal Values from Random Walk"