Reinforcement Learning Techniques in 'Jumper'

MITCHELL BRUNTON and SHAANAN N. COHNEY
University of Melbourne

In this paper, we focus on the development of two agents for playing 'Jumper', under a memory limit of 1024KB and a time limit of 15s per game. Our aim was twofold, to develop a winning agent, and to investigate the effectiveness of machine learning as applied to the problem. Our first AI utilized game-search, and saw us choose between Negascout, Minimax and, alpha-beta pruning variants of the aforementioned. For the development of our evaluation function we investigated 1) the number of safe spaces secured 2) The number of regular spaces secured 3) the number of jump moves possible from a given state 4) board density. For the evaluation to be effective, we tackled the problem of weight generation using machine learning. Our final system used Temporal Difference learning with a gradient descent model. It was found however that learning during play was ineffective and so after learning, we assigned static weights. This agent succeeded in defeating all other agents developed by the competing researchers. The searching player was found to function optimally with Minimax and $\alpha - \beta$ pruning with a weighting of safe spaces to regular spaces of 0.6:0.4.

The second AI developed was modelled after G. Tesauro's TD-Gammon backgammon player and used a multi-layer perception neural network in place of a standard evaluation function. It was trained against three different players; 1) itself 2) the search player 3) a random player.

Our tests show that while the Neural Network player did learn, it developed only so far as to reduce its margin of loss, with outcomes converging after around 20,000 games. However, it did display intelligence in discovering features that previously required expert knowledge to identify.

While machine learning applied to search resulted in superior outcomes, the neural network indicated much room for further improvement. Improvement in the neural network is expected to result in more insight into intelligence in general, exhibited through its zero-knowledge discovery of features.

General Terms: Machine Learning, Jumper

Additional Key Words and Phrases: temporal difference learning, neural networks, negascout, minimax, alpha-beta

1. INTRODUCTION

The development of intelligent agents for zero-sum two agent board games remains a challenging problem within the field of AI. Jumper is a two player, sequential game played on an NxN board. The branching factor in Jumper is non constant but is experimentally measured to be an average of 37.26 over 1000 runs, within the capabilities of modern search algorithms on reasonable time scales (measured in seconds per move). This report looks to address the problem of reinforcement learning as applied to devising an effective game-playing agent for the game of Jumper. AIs were evaluated in terms of efficiency and effectiveness. We divided the problem into two distinct sections; improving the performance of tree search agents with expert knowledge and, developing a neural network AI (also referred to as MLP Player) that started with zero-knowledge. The research for our agents draws largely on Tesauro's TDGammon [Tesauro 2002] and the more modern TD-Leaf algorithm [Baxter et al. 1999]. We aim to develop the two techniques side by side in order to evaluate which is most effective in developing a competitive player. Importantly we limited our AIs to 1024KB of memory and 15s per game, which allows us to focus on improvement in technique rather than in hardware advances.

Whilst game search may have limited other applications, the results from improving learning techniques have the ability to contribute to the improvement of general AI in the sense that they allow the extrapolation of general conditions from limited data. This was especially evident in the identification of important evaluation features by our zero-knowledge Neural Network AI.

In developing the search AI we compared search algorithms including: minimax with $\alpha - \beta$ pruning [Richards and Hart 1961] and Negascout [Reinefeld and Hsu 1983]. We also investigated evaluation features, along with the ability of two different machine learning schemes, gradient descent learning and TD-Leaf learning as described in [Baxter et al. 1999] to judge the relative worth of these features, and fine-tune our game-playing agent's ability to evaluate board states.

Our final results were promising in that our MLP Player definitely developed a measure of capability with the game, and was by far more time efficient than the searching AIs. However, it could not attain the level of strength displayed by them. TD learning techniques applied to search proved to be highly effective in developing a strong player, but came at the cost of time efficiency. Our final AI thus utilized feature weights hard coded from the results of training in order to minimize overhead during play.

The strongest AI we produced was able to defeat all competing researcher's variants and is thus a good indicator of the strength of reinforcement learning techniques as applied to this specific problem.

2. AGENT DESIGN

We designed three distinct agents, utilizing three separate paradigms.

Our first designed agent was a random move-finder. Each game state was encapsulated in a node, and that node's children were generated. A child was selected at random, and the move corresponding to that child was returned. As expected, this algorithm performed poorly, however it allowed us to ensure the structure of our code was sound before we moved on to more adept playing strategies.

The second agent was based around regular game search methods. Research on turn based game agent design has traditionally focused on two areas. The first of these, search algorithm design, is focussed on developing techniques for efficiently exploring game trees. Key developments include 'minimax', a basic technique, attributed to John Von Neumann, and the subsequent refinement via $\alpha - \beta$ pruning discovered in the late 1950s [Richards and Hart 1961]. However, even with pruning, most games are not searchable down to terminal nodes. This required the development of heuristics for the utility function: an evaluation function. Evaluation function design is the second core challenge in designing game playing agents and was the more important of the two factors in determining player strength within our constraints. This was because all the competing AIs were able to explore roughly the same search space so the delineating factor was our ability to analyse a given board state.

The third agent designed was based around a feed-forward, multi-layer perception neural network. In simpler terms, it utilized a network of nodes with multiple inputs and dual outputs to model a complex non-linear reward function.

The win condition for 'Jumper' is having more pieces on the board than your opponent at a terminal node, when all board spaces are occupied or dead. This was the starting point for the development of both of evaluation function and the input for back-propagation in our neural network player.

2.1 Search Algorithms

Our initial attempt at a search agent employed a standard minimax algorithm which expanded the game-tree to depth four. The agent immediately bested both its human creators and achieved a one-hundred percent win rate against the random AI. Further improvements made to minimax did not directly affect playing ability, only efficiency. However, under time constraints an improvement in efficiency allowed for the evaluation function to be more computationally expensive, and allowed searching to a greater depths. The first improvement we made was the implementation of $\alpha - \beta$ pruning. By not exploring subtrees for which there can be no better outcome than what has already been seen, we can "prune" large sections of the game-tree, saving time. The effectiveness of this approach is dependent on how well ordered the nodes of our tree are. In $\alpha - \beta$ pruning, when taking the 'max' of a node's children, it is most beneficial for child nodes to be ordered from best to worst, and when taking the 'min', optimal ordering is worst first. However, sorting nodes incurs a significant overhead. We experimentally verified that for our evaluation function the added computation it takes to sort nodes was not outweighed by the savings from pruning. This could be due to our failure to find a heuristic measure of a node's utility of both sufficient efficiency and that closed reflected it's true utility.

We tried several techniques to improve move ordering, the first of which was to use our evaluation function. This ordered maximum nodes via the difference in number of player pieces and number of enemy pieces. This tactic had the draw-back of being incredibly slow, instead of just evaluating leaf nodes, we were now evaluating every non-root node in our game tree. Our heuristic had to be something much faster to calculate. It seemed a reasonable assumption that jump moves are better than non-jump, or "place" moves. Each node, as well as encapsulating a game board, also holds the Move object which links it to its parent node. By ordering nodes in terms of the length of their Move object, we were able to process nodes corresponding to long jumps first, then shorter jumps and finally place moves. This ordering scheme had the added benefit of using the exact same code for maximising nodes and minimising nodes: the "best" child node of a maximising node is the node associated with the longest player jump move, and the "worst" child node of a minimising node was the node associated with the longest enemy jump move.

Our method which generated a node's children contained two sub-methods: one to generate a list of valid place moves and one to generate a list of valid jump moves. Initially in our code, "place" children were generated in order before "jump" children in the returned list of moves. We found that by reversing this, and ordering jump moves first, the time it took to choose a move improved by a factor of eight. Moreover, this method was also computationally more efficient that sorting nodes by move length. This final node-ordering scheme introduced no overhead, and any further attempts we trialled to order the nodes using other heuristics resulted in slower overall runtime.

Negascout is an alternative to alpha-beta pruning that aims to further minimize the size of the explored search tree [Reinefeld and Hsu 1983]. The "nega-" prefix denotes that rather than utilizing separate functions to calculate the utility of maximum and minimum nodes, a negative recursive call is used. This decreases the difficulty of implementation. The "scout" denotes that the searches are first performed using a null window, which can result in a gain in efficiency given sufficiently well-ordered nodes. This potentially allows searching to greater depths. We found that with the given move ordering, $\alpha - \beta$ pruning was more efficient than Negascout. Our attempts to further order the nodes carried a computational cost which outweighed the benefits gained by Negascout. We thus chose minimax with alpha-beta pruning as the search algorithm for our final game-playing agent.

We also trialled only considering the child nodes corresponding to place moves if there were no children corresponding to jump moves. Whilst the efficiency did improve drastically, the exponential nature of increasing search depth meant that we still couldn't feasibly search any deeper without violating our time constraints. Regardless, we tested this approach at a search depth of six, against our original player at a search depth of four, and the original player proved victorious. We believe this is probably due to Jumper not lending itself well to a greedy approach: good players will often make moves which don't immediately maximise their progress, in order to set up better moves later on. By limiting our player's move choices, we infringed on its ability to work towards good moves in the future, and this was not counterbalanced by the increased search depth.

2.2 Evaluation Function Design

The skill of our agent was directly correlated with the effectiveness of our evaluation function in representing the utility of a given board state. A typical evaluation function consists of a linear weighted sum of different feature functions, where each feature measures some property of the board, with the assumption that these properties accurately summarise what constitutes a strong board position for the player. Our evaluation function returned a +1 for a winning board configuration, 0 for a draw, and -1 for a loss. Our initial evaluation function initially returned the value of a single feature; measuring the difference between the number of player pieces and enemy pieces on the board. By observing our agent playing games against itself, we noticed that the edge cells tended to be filled in quicker than inner-cells. More generally, place moves tended to form clumps of filled-in cells. We surmised that the reason for this was that these positions were less likely to be jumped by the enemy. This led to the development of our second feature function, which calculated the number of safe player pieces, less the number of safe enemy pieces. A piece was deemed 'safe' if it could not be jumped from any child state of the current board. Our improved agent outplayed our single-feature implementation. However the likelihood of a win fluctuated as we altered the two weights w_1 and w_2 in our evaluation function.

We noted that this approach was ignoring potentially important information; by designating a piece as either "jumpable" or "non-jumpable", we were ignoring the fact that certain pieces can be more jumpable than others. Cells can be jumped over four different axes (horizontally, vertically, left/right-diagonally), and so we devised a feature which took into account the number of different directions each piece could be jumped from (ranging from 0 to 4). Another feature was added which measured the density of the board, the reasoning being that each of the values of other feature functions would have a more pronounced effect if the game was close to being over. For example, having two more pieces on the board than the enemy would be more significant if there were only four empty cells left, than if there were twenty.

In order to automate the selection of effective weight values, we implemented two different machine learning schemes: gradient descent learning, and temporal-difference leaf learning. These methods have a lot of cross-over, as both involve iteratively adjusting individual weight values by shifting them towards a local minima of some error function. In our case, this error function was the difference between the utility of a node, as calculated by evaluating it directly, and its true utility, which involves using our move-finding algorithm to expand out its full tree.

Gradient descent learning, was applied in the initialisation phase of our player. A set of random boards were generated, and each of these boards were evaluated using our linear weighted sum of feature functions (initially each weight is set to a uniform value). We then expanded each node, using our alpha-beta algorithm, and calculated its true utility. While the absolute difference between the result of our evaluation function on the node and the node's true utility was greater than some value epsilon, we updated each weight parameter so it moved in towards a local minima of an error function. This essentially allowed our evaluation function to look further down a game tree when it evaluated leaf nodes, as their projected utility starts to resemble the utility of nodes from a greater depth. In order to have a larger set of random game boards, we reduced our search depth while performing gradient descent learning to just two. We are unsure of all of the consequences of this reduction.

We also implemented temporal difference learning in the form of the TDLeaf algorithm as per [Baxter et al. 1999]. This also updates the weight vector, however it does so on-the-fly, as we are expanding our game tree. When we have found our optimum route down the expanded game tree, we update each node along this path, taking its true utility to be the utility of the leaf node of this path. This form of machine learning ended up being more effective than gradient descent, possibly due to the fact that it updated nodes which represented current or likely future board states of the game, rather than just the random nodes used in gradient descent. This allows temporal-difference learning to respond in real time to different playing strategies.

2.3 Neural Network Player

Our neural network aimed to replicate the functionality of a standard evaluation function, but to do so with inputs correlating solely to the board state at any given time and no knowledge engineering. By using reinforcement learning techniques, it was hoped that the two outputs of the neural network would correlate to the probability of a win for each player. The aim of this agent was to determine the levels of expert knowledge required in designing a strong agent. We utilized a three layer neural network, modelled after that of G. Tesauro in his paper on TDGammon [Tesauro 2002] (details in Appendix A).

Each 'perceptron' or node in the network utilized a sigmoid function to convert inputs into a single output value, so as to normalize the values and distribute them appropriately from -1 to 1, each node's inputs were assigned a weight before being fed in. The aim was that, by adjusting the weights appropriately, the function would approximate the utility value for a given board.

The agent was designed within the Encog Framework (https://github.com/encog), in order to ease development and allow testing of multiple learning algorithms.

In achieving our goal of designing a strong agent, a number of challenges presented themselves.

A key parameter in the neural network model included the number of hidden units in the second layer of our three-layered network. The number of hidden units represents a tradeoff: as the number of hidden units increases, the number of weights in the network increases exponentially, and so too does the required time for back-propagation. However, more hidden units allows the network to represent more complex functions, and if one does not provide a sufficiently complex function to regress to, the output will not accurately reflect the desired model (our reward function) [Cybenko 1989]. For the 6x6 board we settled on sixty hidden units, as increasing it beyond this limit restricted the amount of training runs we could perform within the time allotted for our research. An additional trade-off that was later discovered and discussed in Section 3, was that as the complexity of the function being modelled increased, so too did the number of local minima in the approximations, leading to sub-optimal agents.

In order to most closely replicate Tesauro's model, we utilized a simple back-propagation algorithm with a learning rate of 0.4 and a momentum of 0.9, the variation of which was outside the scope of our investigation. However these two parameters likely had an effect on our results.

The next important parameter in creating a viable agent was the number of games played during training. We expected our agent to improve significant as the number of games increased. Teasuro experimentally determined that for backgammon, somewhere between 10^5 and 10^6 games were required before his agent was of competitive standard [Tesauro 2002]. Due to time constraints, we trained up to a maximum of 300,000 games. Our need for creating large training data sets under the time constraints of our research required that we focus the majority of our training efforts on playing our agent against itself as our other agents played too slowly (Figure 2). However we did also run some preliminary training tests against other opponents, but results were not available at time of writing.

Unlike Teasuro, our training occurred at the end of games. This was initially planned so as to allow us to evaluate the agent's performance accurately at the end of each 'training run', however it later proved to be a sub-optimal design choice. While it simplified coding, it meant our agent could only learn from its decisions once a game was over. This was in contrast to the dynamic adaptation of both TDGammon and our TD-Leaf agent described above. In retrospect this was a poor decision as we wasted further opportunities for improvement.

It was discovered during our initial investigation that due to the deterministic nature of its opponents, our agent would explore very little of 'Jumper's' search space before the network weights converged. This lead to poor play against agents that it had not been trained against. In order to circumvent this issue, we added two separate modes of play to our agent; learning and non-learning. Under learning mode, we set a probability epsilon to 0.1% under which the agent would make a random move rather than the best evaluated move. This lead to increased exploration of the game tree and over the course of thousands of games improved play.

Due to time constraints, our neural network agent was optimized to play as white, with a planned investigation of it's effectiveness as a black player dependant on the success of the white agent and further time.

3. EVALUATION OF AGENTS

Our testing was largely performed against our own AIs, a random player, a trained MLP Player, a standard Minimax player (with alpha-beta pruning), a Negascout player and a modified Minimax player with feature weights adjusted by TD Learning. With our Minimax (with alpha-beta pruning) performing the strongest of the set, it served as a good benchmark.

vs	Random (B)	MLP50K (B)	Minimax (B)	Negascout (B)	Minimax with TD Learning (B)
Random (W)	56.2%	100%	0%	0%	0%
MLP50K (W)	100%	47%	100%	100%	0%
Minimax (W)	100%	100%	100%	100%	0%
Negascout (W)	100%	100%	0%	100%	0%
Minimax with TD Learning (W)	100%	100%	100%	100%	74%

Fig. 1: 6x6 Board - White Win Percentage from 1000 games

Figure 1 shows the win percentages for each player against each other player with (W) and (B) indicating their color. Results show a clear hierarchy in the strength of players. Expectedly, all beat the random player. Further than that, the search AIs consistently beat the MLP AI, and our final AI with TD determined weights emerged as the victor.

AI	Time per Move
Minimax	22090ms
Minimax alpha-beta with TD-leaf:	250ms
Minimax alpha-beta with Gradient Descent	289ms
Minimax alpha-beta with TD and Gradient Descent	246ms
Minimax alpha-beta without Learning	199ms
Minimax alpha-beta (place before jump)	1756ms
Minimax alpha-beta (jump before place)	224ms
Negascout	391ms
MLP Player	3.31ms
Random Player	0.163ms

Fig. 2: Move Timings (conducted on 6x6 random board, searching to depth 5 over 1000 trials)

For our comparison of player efficiency Figure 2 also showed a clear ordering. The neural network was the fastest of all the intelligent players, with the faster of the search AIs two orders of magnitude behind, even after $\alpha - \beta$ pruning and move ordering optimizations.

AI	Wins-Losses
Gradient Descent (W)	22-20
Gradient Descent (B)	33-13
Total (With-Without)	35-53

Fig. 3: Win Comparison of Gradient Descent vs Without (starting on random board)

AI	Wins-Losses
TD-leaf starting first	29-20
TD-leaf starting second	35-11
Total (With-Without)	55-40

Fig. 4: Win Comparison of TD-leaf vs Without (starting on random board)

Figures 4 and 3 show the results of games of Jumper between an agent with hard-coded weight parameters, and agents equipped with machine-learning. When an agent with machine learning played as white (first), it beat its opponent. However in both cases the margin was not as great as when the agent with hard-coded parameters played first. The parameters chosen for the hard-coded agent were arrived at through observation of how these parameters converged under machine learning.

Figure 5 shows the improvement over time of the MLP Player, with the average difference between player scores on the y axis calculated as the number of white pieces on the board minus the number of black pieces on the board. It shows an initial steep improvement in outcomes followed by a levelling off. This occurs against all opponents and for all board sizes we tested.

One potential cause of the limited win behaviour exhibited by the neural network player could be the result of three factors we've identified. The first could be a lack of training. There is also the issue of local minima in the vector space of the error function. Finally the number of hidden units determines an upper bound on the complexity of the function. Our weights appeared to converge at around 20,000 games, an effect consistent with what Tesauro identified as 'saturation'. This was proportional to the size of the

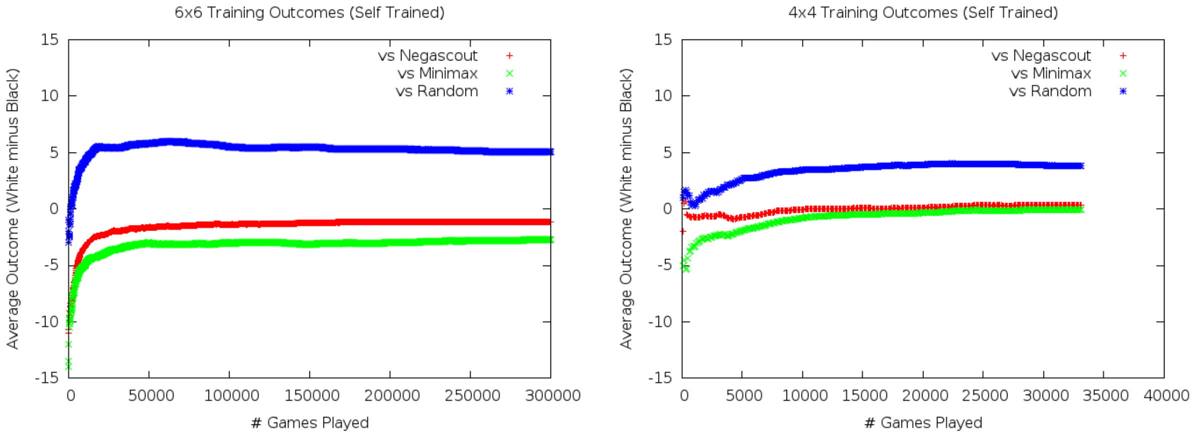

Fig. 5: MLP Player Training

network [Tesauro 2002]. Such effects can be reduced by increasing the number of hidden units, however this exponentially increases the time taken to train the network.

The results were also found to be sensitive to initial weights, which are randomly generated, and this lead to variance across runs other than as displayed above. In one instance, the MLP player weights converged to an average win case for a 4x4 board, but this was not found to be replicable.

Another issue facing further improvement by the MLP player was over training against certain opponent types. When trained against a given deterministic opponent player (Negascout or Minimax) the network converged to an average win. This is an example of 'over-training' whereby rather than becoming a strong generalist player of the game, the MLP player adjusted to beat the specific deterministic move sequence. This was demonstrable by introducing a small element of randomness into opponents under which circumstances the MLP's advantage was negated. Additionally an over-trained network generally demonstrated poor outcomes against the other test AI. This issue could possibly be reduced in future by use of a better training algorithm that further takes into account the possibility of such local minima. Additionally one would train against a series of different players in order to learn from their various play-styles, however this is worthwhile only after establishing a base-line level of learning.

One observation not noted in the results section was feature discovery by the MLP player. After around 20,000 games, the MLP self-trained player began to play with strategies commonly exhibited by human players, including 'corners first', and a preference for safe spaces over spaces that could be jumped. This is important as it shows the development of human-like intelligence within the AI, even though it did not develop to a level where it could consistently beat the searching algorithms.

Our most effective player was also one of our least complex. The only addition over standard minimax was the introduction of alpha-beta pruning, and a very modest node-ordering scheme which involved generating jump-nodes before place-nodes. While our final weight parameters were derived from observations of gradient descent and TD-leaf machine learning techniques, it served to be more effective to hard-code these constants into our final implementation.

4. CONCLUSION

Our research in building an adept game-playing agent saw us trial many different techniques, however the majority of these did not make it into our final player. Whilst our implementation of gradient descent, and temporal difference machine learning helped us to arrive at optimal parameters for our parameter weight vector, it was more effective to hard-code these values, rather than recompute them each time the agent was run. The node ordering scheme which proved most effective was also the simplest, and whilst our neural network showed promise of further improvement, it was our more traditional minimax with alpha-beta algorithm which displayed the best game-playing ability. By focussing on an effective evaluation function, and keeping our player efficient enough to search to a depth of five, we were able to devise a very successful agent. Further improvements would likely focus on improving the TD Learning data set and storing it across games.

The neural network player showed strong signs of learning, improving to the point of developing features given zero-knowledge other than a set of legal moves. However, under the conditions we trialled it did not learn to the point of defeating the AIs developed using different reinforcement techniques. In particular, a strong advantage was held by the AI trained using TD Learning. Teasaro indicated that TDGammon improved to the point of being competitive with expert players largely after the integration of features from 'Neurogammon' into the Neural Network, and adding such expert knowledge is a strong point for future research in improving outcomes. However Tesauro also notes that "this provides only short-term benefit and is dangerously likely to turn out to be a waste of time in the long run" [Tesauro 2002]. Thus efforts are better focused on keeping the input simple but developing new learning techniques. One possible option is performing a search beyond a single ply and this is the recommended avenue for future investigation.

Our results indicate currently that simplicity is the key to developing strong players for 'Jumper'. Dynamic learning algorithms fared poorly compared to pre-training our agents. Whilst there may exist stronger features and parameter settings, it would require further research and development to identify these.

APPENDIX

A. NEURAL NETWORK STRUCTURE

The neural network was based off TDGammon and utilized a similar structure. Each cell was represented by three input units, one to represented occupied by black, one for white, and one for dead. A clear space was represented by a zero on all three and each was active with a value for one if the condition was matched. There were two additional non-cell input units, a bias unit and a unit that indicated whose turn it was to move. The input cells were then completely connected to a series of sixty (for 6x6) hidden units, each with a sigmoid activation function. Finally, these were completely connected to two output units, one representing the probability of Black winning and one representing the probability of White winning, from the input game state. The reason for separating the two was to include the possibility of a draw, such states where Pr(White) and Pr(Black) both tend to zero.

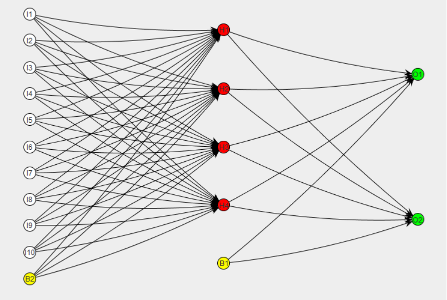

Fig. 6: An example three-layered neural network, additional units not shown

B. NEGASCOUT PSEUDOCODE

```
negascout(Board, alpha, beta, depth, color):
  if(d == maxdepth or game_over):
    return color*evaluate(Board)

  a = alpha
  b = beta

  for each Move m in Board:
    Board'=Board.apply_move(m)
    t=-negascout(Board', -beta, t, d+1, -color)

    if(t>a && t<beta && notFirstMove && d<maxdepth-1):
      a = -negascout(Board',-beta,-t,d+1,-color)

    a = Max(a,t)

    if(a>beta):
      return a

    b = a+1

  return a
```

C. TD LEARNER PSEUDOCODE

```
gradientDescent():
  trainingSet := getTrainingSet(SIZE)
  for node in trainingSet:
    updateWeights(node)

tdLeaf(nodes, leafUtility):
  for node in nodes:
    updateWeights(node, leafUtility)

updateWeights(node):
  util = evaluate(node)
  trueUtil = expand(node)
  while (abs(util - trueUtil) > EPS):
    for i = 1..n:
      w_i = w_i - N*(util - trueUtil)*f_i(node)
      util = evaluate(node)
      trueUtil = expand(node)

updateWeights(node, trueUtil)
  util = evaluate(node)
  while (abs(util - trueUtil) > EPS):
    for i = 1..n:
      w_i = w_i - N*(util - trueUtil)*f_i(node)
      util = evaluate(node)
```

ACKNOWLEDGMENTS

We are grateful to the following people for resources, discussions and suggestions: Mahsa Salehi, Prof. Chris Leckie., Prof. Michael Kirley

REFERENCES

Jonathan Baxter, Andrew Tridgell, and Lex Weaver. 1999. TDLeaf (lambda): Combining temporal difference learning with game-tree search. *arXiv preprint cs/9901001* (1999).

George Cybenko. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems* 2, 4 (1989), 303–314.

Imran Ghory. 2004. Reinforcement learning in board games. (2004).

Robert Hecht-Nielsen. 1989. Theory of the backpropagation neural network. In *Neural Networks, 1989. IJCNN., International Joint Conference on*. IEEE, 593–605.

John Lenz. 2003. Reinforcement Learning and the Temporal Difference Algorithm. (2003).

Qian Liang. 2003. *The Evolution Of Mulan: Some studies in game tree pruning and evaluation functions in the game of amazons)*. Ph.D. Dissertation. The University of New Mexico, Albuquerque, New Mexico.

James L. McClelland. 2013. *Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises*. Chap 9. pages.

Alexander Reinefeld and Tsan-sheng Hsu. 1983. An Improvement to the Scout Tree Search Algorithm. (1983).

DJ Richards and TP Hart. 1961. The alpha-beta heuristic. (1961).

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall.

Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction*. Vol. 1. Cambridge Univ Press.

Gerald Tesauro. 2002. Programming backgammon using self-teaching neural nets. *Artificial Intelligence* 134, 1 (2002), 181–199.