Duel of Wits a Self Learning Artificial Intelligence using Reinforcement Learning

S. Komal Kaur[1], T.Adilakshmi[2]
[1,2]*Department of Computer Science and Engineering*
Vasavi College of Engineering, Hyderabad, Telangana, India

Abstract— This paper describes methods to build a self-learning Artificial Intelligence strategy board game called "Duel of Wits". The game's objective is to attack an opponent with their given projectile that can be maneuvered using moveable mirrors placed on the board. The paper consists of 3 major parts. The first part is the game itself which is to be built using the Unity game engine. The second part of the paper deals with building traditional AI machine using Alpha-Beta Pruning algorithm. The third part and main objective of the paper is a self-learning Artificial Intelligence using proximal policy optimization with a neural network as a function approximator.
Keywords – Self-learning AI, Unity engine, proximal policy optimization, neural networks.

I. INTRODUCTION

Duel of Wits is a two player board game of 4x4 size, in which the goal is to shoot your opponent with the help of mirrors that can reflect your projectile. Each player gets alternate chance to defeat the opponent with a laser projectile. The players are placed at the opposite corners of the board. The four mirrors are placed at the centre of the board. Each piece on the board can be moved in different directions, viz., upward, downward, left and right. In addition to movements, the pieces, both players and mirrors, can be rotated. The player piece can have orientations such up, down, left, and right. The mirror piece has two orientations, viz., 45 degrees and 135 degrees with respect to positive x-axis. Unity is a cross-platform real-time engine developed by Unity Technologies, first announced and released in June 2005 at Apple Inc.'s Worldwide Developers Conference as an OS X-exclusive game engine. As of 2018, the engine has been extended to support 27 platforms. The engine can be used to create both three-dimensional and two-dimensional games as well as simulations for its many platforms. Several major versions of Unity have been released since its launch, and the project uses Unity version 2018.3.0f2.

Computational intelligence techniques were combined with games for the first time in 1959. Samuel applied a simple reinforcement learning algorithm to the board game Checkers. Without human commands, through only playing itself and observing the sequences which won games and which sequences lost games, and using the very limited hardware available at the point in time, the algorithm was able to learn strategies good enough to beat its creator. AI and Machine Learning (ML) reached their peak in the last couple of years and are stealing the show. The significant advancements in the field of AI over the last decade is remarkable. The recent improvements in the AI landscape have created an enormous interest in the business world. Game Artificial Intelligence (Game AI) refers to the techniques used in computer and video games to produce the illusion of intelligence in the behavior of non-player characters (NPCs). Artificial intelligence in games is typically used for creating player's opponents. If the computer opponent always does the same thing or is too difficult or too easy, the game will suffer. The main goal of Game AI is to not beat the player, but to merely entertain and be challenging. Advanced AIs are not always fun as they end up making the game much harder. Real AI addresses fields of machine learning and decision making based on arbitrary data input as opposed to game AI which only attempts to produce the illusion of intelligence in the behavior of NPCs.

II .Proposed Method

Traditional AI is made to handle unseen inputs, large state space, too many options possible to compute an exact optimal solution, Engineering criteria: best possible performance but in Game AI the game world is known, though it can still be large, In a known world, optimal solutions can be pre-computed, Entertainment criteria: smart enough to pose a challenge, but not smart enough to be undefeatable.
The main objective of this paper is to build an application that can learn to play 'Duel of Wits' game using knowledge gathering and game playing experience. The AI game is divided into three phases to achieve the project objective:
Build Player vs Player game mode with the help of game rules

Building Player vs Traditional AI game mode
Implement self-learning AI using Reinforcement Learning

Unity gives users the ability to create games and interactive experiences in both 2D and 3D, and the engine offers a primary scripting API in C#, for both the Unity editor in the form of plug ins, and games themselves, as well as drag and drop functionality. Prior to C# being the primary programming language used for the engine, it previously supported Boo, which was removed in the Unity 5 release, and a version of JavaScript called Unity Script, which was deprecated in August 2017 after the release of Unity 2017.1 in favor of C#.

The engine has support for the following graphics APIs: Direct3D on Windows and Xbox One, OpenGL on Linux, macOS, and Windows, OpenGL ES on Android and iOS, WebGL on the web, and proprietary APIs on the video game consoles. Additionally, Unity supports the low-level APIs Metal on iOS, macOS and Vulkan on Android, Linux, and Windows, as well as Direct3D 12 on Windows and Xbox One.

Within 2D games, Unity allows importation of sprites and an advanced 2D world renderer. For 3D games and simulations, Unity allows specification of texture compression, mipmaps, and resolution settings for each platform that the game engine supports, and provides support for bump mapping, reflection mapping, parallax mapping, screen space ambient occlusion (SSAO), dynamic shadows using shadow maps, render-to-texture and full-screen post-processing effects. Since about 2016 Unity also offers cloud-based services to developers, these are presently: Unity Ads, Unity Analytics, Unity Certification, Unity Cloud Build, Unity Everyplay, Unity IAP ("In app purchase" - for the Apple and Google app stores), Unity Multiplayer, Unity Performance Reporting, Unity Collaborate and Unity Hub.Unity supports the creation of custom vertex, fragment (or pixel), tessellation and compute shaders. The shaders can be written using Cg, or Microsoft's HLSL.

III. Implementation

The ML-Agents toolkit is a Unity plugin that contains three high-level components:
Learning Environment - which contains the Unity scene and all the game characters.
Python API - which contains all the machine learning algorithms that are used for training (learning a behavior or policy). Note that, unlike the Learning Environment, the Python API is not part of Unity, but lives outside and communicates with Unity through the External Communicator.
External Communicator - which connects the Learning Environment with the Python API. It lives within the Learning Environment.

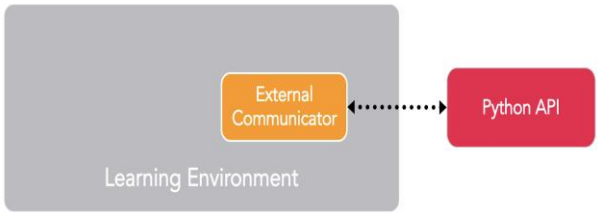


Fig 1: Communication of Unity with Tensorflow

A zero-sum game is a mathematical representation of a situation in which each participant's gain or loss of utility is exactly balanced by the losses or gains of the utility of the other participants. If the total gains of the participants are added up and the total losses are subtracted, they will sum to zero. Thus, cutting a cake, where taking a larger piece reduces the amount of cake available for others, is a zero-sum game if all participants value each unit of cake equally [5].In contrast, non-zero-sum describes a situation in which the interacting parties aggregate gains and losses can be less than or more than zero.

A zero-sum game is also called a strictly competitive game while non-zero-sum games can be either competitive or non-competitive. Zero-sum games are most often solved with the minimax theorem which is closely related to linear programming duality, or with Nash equilibrium. Alpha Beta pruning and Negamax search is a variant form of minimax search that relies on the zero-sum property of a two-player game.

3.1 Alpha-Beta Pruning

Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, Go, etc.). It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

The algorithm maintains two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of respectively. Initially, alpha is negative infinity and beta is positive infinity, i.e. both players start with their worst possible score. Whenever the maximum score that the minimizing player (i.e. the "beta"
player) is assured of becomes less than the minimum score that the maximizing player (i.e., the "alpha" player) is assured of (i.e. beta ≤ alpha), the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play.
Alpha-Beta pruning Algorithm

```
function alphabeta(node, depth, α, β, maximizing Player) is
   if depth = 0 or node is a terminal node then
      return the heuristic value of node
   if maximizingPlayer then
      value := −∞
      for each child of node do
         value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
         α := max(α, value)
         if α ≥ β then
            break (* β cut-off *)
      return value
   else
      value := +∞
      for each child of node do
         value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
         β := min(β, value)
         if α ≥ β then
            break (* α cut-off *)
      return value
```

The difficulty levels of the AI are selected by fixing the depth of the Minimax tree. For example, the easy level is limited to a single depth tree whereas the hard mode is set to a depth of 4. When the Minimax tree is expanded for a new depth level, the time required to process each node increases exponentially with each level. Hence, while the 'hard' difficulty of the AI performs extremely well in terms of winrate, it suffers from a larger delay to process the correct choice to make by searching all the nodes until the leaf nodes nodes of each subtree. With the same logic, it follows that the 'easy' difficulty reacts instantly but often lose due its lack of foresight.

3.2 NegaMax Algorithm

Negamax search is a variant form of minimax search that relies on the zero-sum property of a two-player game. This algorithm relies on the fact that max(a,b)=-min(-a,-b) to simplify the implementation of the minimax algorithm. More precisely, the value of a position to player A in such a game is the negation of the value to player B. Thus, the player on move looks for a move that maximizes the negation of the value resulting from the move: this successor position must by definition have been valued by the opponent. The reasoning of the previous sentence works regardless of whether A or B is on move. This means that a single procedure can be used to value both positions. This is a coding simplification over minimax, which requires that A selects the move with the maximum-valued successor while B selects the move with the minimum-valued successor.
Negamax

```
const int sign[2]={1,-1}
int NegaMax(Board b, int depth, int color)
{
if (GameOver(b) or depth>MaxDepth)
        return sign[color]*Analysis(b)
int max = -infinity
for each legal move m in board b {
        copy b to c
        make move m in board c
           int x= - NegaMax(c, depth+1, 1-color)
        if (x>max) max = x
```

```
    }
    return max
  }
```

3.3 MARKOV DECISION PROCESS

A Markov decision process (MDP) is a discrete time stochastic control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker. At each time step, the process is in some state 's', and the decision maker may choose any action 'a' that is available in state 's'. The process responds at the next time step by randomly moving into a new state s', and giving the decision maker a corresponding reward R(s,s').The probability that the process moves into its new state s' is influenced by the chosen action. Specifically, it is given by the state transition function P(s,s'). Thus, the next state s' depends on the current state s and the decision maker's action a. But given 's' and 'a', it is conditionally independent of all previous states and actions.

The solution for any Markov Decision Process can be found by deriving the policy of the described environment. The most common method to find the policy is to find the value of the rewards at each state and the actions corresponding to the maximum rewards in future states. The Bellman equation gives the value of each state on maximizing its rewards of future and is as follows:

$$U(s) = R(s) + \gamma \, max_a \sum_{s'} T(s,a,s')U(s')$$

Fig 2 : Bellman equation for Value function

3.4 REINFORCED LEARNING IN POLICY GRADIENT METHODS

Reinforcement Learning is the most general description of the learning problem where the aim is to maximize a long-term objective. The system description consists of an agent which interacts with the environment via its actions at discrete time steps and receives a reward. This transitions the agent into a new state. A canonical agent-environment feedback loop is depicted by the figure below.

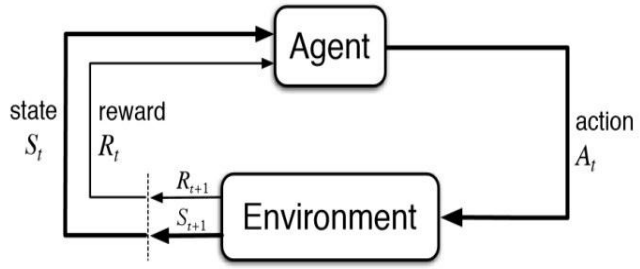


Fig 3 : Agent-Environment interaction

The Reinforcement Learning flavor of the learning problem is strikingly similar to how humans effectively behave — experience the world, accumulate knowledge and use the learning's to handle novel situations.

3.5 PROXIMAL POLICY OPTIMIZATION

Policy Gradient methods have convergence problem which is addressed by the natural policy gradient. However, in practice, natural policy gradient involves a second-order derivative matrix which makes it not scalable for large scale problems. The computational complexity is too high for real tasks. Intensive research is done to reduce the complexity by approximate the second-order method. Proximal Policy Optimization (PPO) uses a slightly different approach. Instead of imposing a hard constraint, it formalizes the constraint as a penalty in the objective function. By not avoiding the constraint at all cost, we can use a first-order optimizer like the Gradient Descent method to optimize the objective. Even we may violate the constraint once a while, the damage is far less and the computation is much simple. With the Minorize-Maximization MM algorithm, this is achieved iteratively by maximizing a lower bound function M (the blue line below) approximating the expected reward η locally.

First, we start with an initial policy guess and find a lower bound M for η at this policy. We optimize M and use the optimal policy for M as the next guess. We approximate a new lower bound again at the new guess and repeat the iterations until the policy converges. To make it work, we do need to find a lower bound M that is easier to optimize. In PPO, we limit how far we can change our policy in each iteration through the KL-divergence. KL-divergence

measures the difference between two data distributions p and q. We repurpose it to measure the difference between the two policies.

$$D_{KL}(P||Q) = \mathbb{E}_x \log \frac{P(x)}{Q(x)}$$

To limit the policy change to avoid bad decisions, we can find a lower bound function M as :

$$M=L(\theta)-C\cdot\bar{KL}$$

and with L(θ) equals :

$$\hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}\hat{A}_t\right]$$

L is the expected advantage function (the expected rewards minus a baseline like V(s)) for the new policy. It is estimated by an old (or current) policy and then recalibrate using the probability ratio between the new and the old policy. We use the advantage function instead of the expected reward because it reduces the variance of the estimation. As long as the baseline does not dependent on our policy parameters, the optimal policy will be the same. The second term in M is the maximum of KL-divergence but it is too hard to find and therefore we relax the requirement a little bit by using the mean of the KL-divergence instead.

IV. Results

The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source Unity plug-in that enables games and simulations to serve as environments for training intelligent agents. Agents can be trained using reinforcement learning, imitation learning, neuro- evolution, or other machine learning methods through a simple-to-use Python API.

Testing the code base was done using the Unity debugging tool coupled with Visual Studio debugger. Further testing of the development build was done with the help of various participants volunteering to give feedback on the based on their experience with the application. Due to Unity's support for both Android and Windows, the development build was readily available for a larger user base. Furthermore, the Unity Test Runner Tool allows for testing during development while in both, Play mode and Edit mode with test scripts which make sure that all the unit tests are passed successfully.

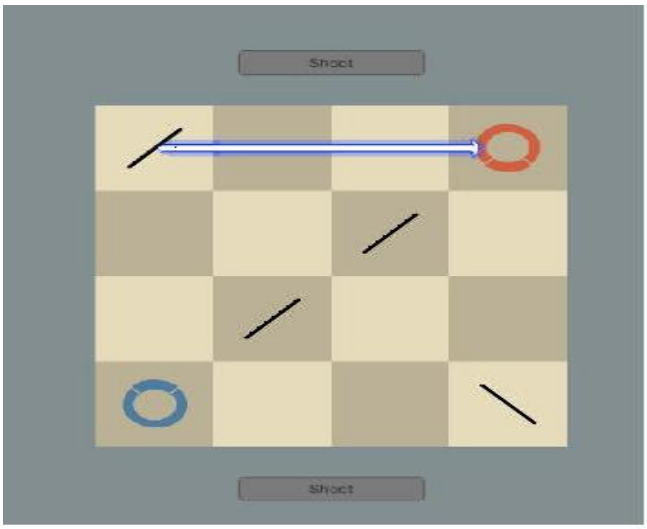
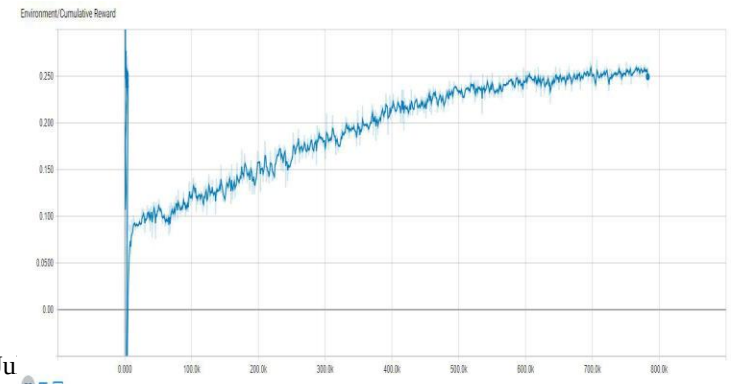


Fig 4 : Board Scene



 ISSN: 2319-6319

Fig 5: Cumulative reward/Step

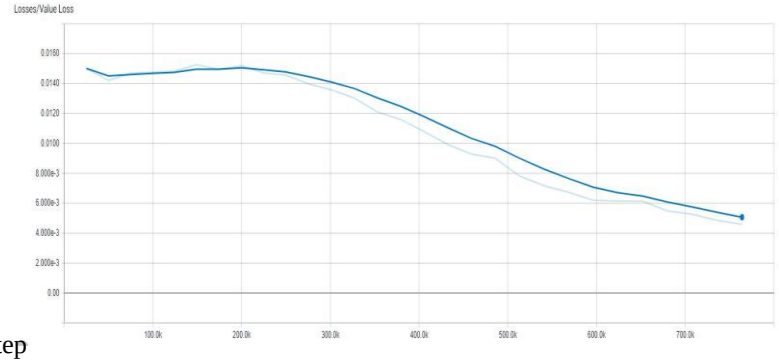


Fig 6 : Value Loss/Step

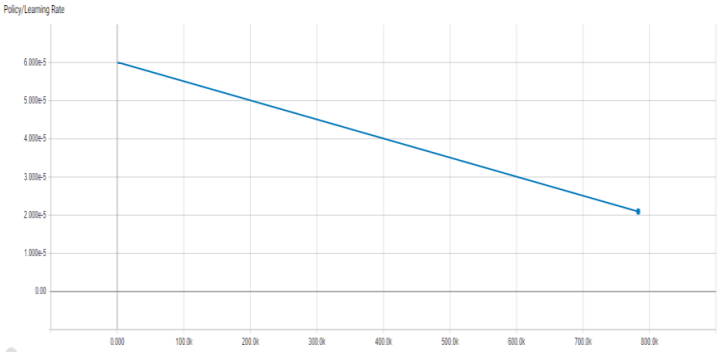


Fig 7: Learning rate/step

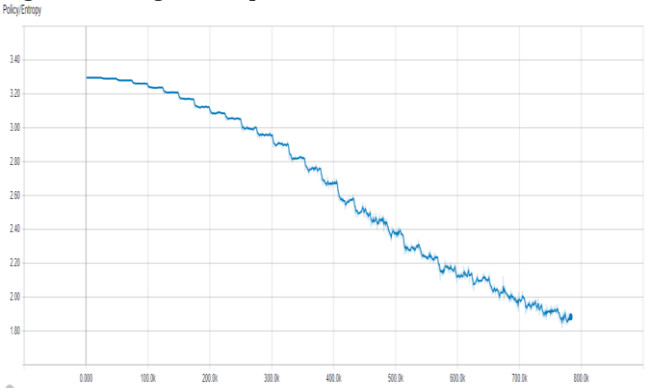


Fig 8: Entropy/Step

V.CONCLUSION

The implementation of the traditional AI as well as the self-learning AI produced satisfiable results, with the traditional AI having a win rate of 99.98% against random human opponents and the self-learning AI having over a 70% win rate against the Traditional AI. For future work, we would like to increase the performance on our AI and deploy our application as a mobile app on various application stores such as Play Store. We would also like to further research on various approaches to the solution of self-learning board games and comparisons with the previously implemented models.

REFERENCES

[1]. Prathik Patil and Ronald Alvares, Cross Platrofrm Application development Using Utility Game engine, International journal of advance research in computer science and Management Studies, 4, 2321-7782 (2015).
[2]. Unity Technologies, https://docs.unity3d.com/Manual/index.html
[3]. https://deepmind.com/blog/alphazero-shedding-new-light-grand-gameschess-shogi-and-go/
[4].https://docs.microsoft.com/enus/dotnet/csharp/programming-guide
[5]. Alpha Beta Pruning https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning
[6].https://en.wikipedia.org/wiki/Minimax
[7].https://en.wikipedia.org/wiki/Zero-sum_game
[8]. https://en.wikipedia.org/wiki/Reinforcement_learning
[9].https://github.com/Unity-Technologies/ml-agents
[10].https://medium.com/@jonathan_hui/rl-proximal-policy-optimization-ppoexplained-77f014ec3f12
[11]. https://medium.com/@jonathan_hui/rl-policy-gradients-explained9b13b688b146
[12].https://medium.com/@awjuliani/super-simple reinforcement-learningtutorial-part-1-fd544fab149
[13]..https://openai.com/blog/openai-baselines-ppo/
[14].https://www.researchgate.net/publication/281576532_Giraffe_Using_Deep_Reinforcement_Learning_to_Play_Chess