|  |  |
| --- | --- |
| Muzero’s systematic implementation for connect four | Abstract  The Google DeepMind’s latest AI Engine called MuZero caused a frenzy when it defeated the 18 times world champion in Go, Lee Sedol. With more than 200 possibilities per move, Go is one of the most complex games known to man. However, with the help of Artificial Neural Networks, they were able to beat human genius in that game. MuZero learns through self-play and no external data is needed except the rules of the game. After playing millions of games with itself, it is able to make intelligent choices for any position of the game. It uses a combination of CNN and RNN to construct the engine. It uses MCTS search to make decisions. In this paper, we perform a systematic implementation of MuZero for the connect-4 game.  Ashwin Ramdas  MSc in Data Analytics  At Dublin Business School  Under the guidance of  Dr.Amir Ismailly  Supervisor |

Table of Contents

i. Declaration2

ii. Acknowledgement3

1. Introduction4

2. Literature Review6

2.1 Previous Research6

2.2 Rationale of current study11

3. Methods13

4. Implementation21

5. Results31

6. Discussions32

6.1 Interpretation in terms of empirical evidence32

6.2 Discussion about Limitations, future research and conclusions 32

**7. References**34

**i.**

**Declaration**

This is to certify that the project ***“MuZero’s systematic implementation for connect four”*** is an original work done by ***Ashwin Ramdas*.** It was completed in a period from 5th June 2020 to 25th August 2020. It was submitted for fulfilling the requirement for the award of degree of Master of Science in Data Analytics at Dublin Business School, Ireland during the period 2019-2020.

I, Ashwin Ramdas, declare that this research is my original work and it has never been presented to any institution or university for the award of Degree or Diploma. In addition, I have correctly referenced all literature and sources used in this work and this work is fully compliant with DBS’s academic honesty policy.

Signed: Ashwin Ramdas

Date:25/08/2020

**ii.**

**Acknowledgement**

I would like to thank my academic supervisor Dr. Amir Ismailly, Dublin Business School for his guidance and patience throughout the project. His constant help was a major source of encouragement for me throughout this time. I also want to acknowledge with deep sense of gratitude to the developers of the jupyter notebook for providing an open source platform for implementing our code.

I would also like to thank my peers, for their constant motivation, support and extremely valuable inputs which enabled me to complete this research successfully.

***Ashwin Ramdas***

***M.Sc. in Data Analytics***

***Dublin Business School***

**1.**

**Introduction**

For hundreds of years, games like Chess[1], Go[2], Shogi, Poker[3, 4] and many such board games have been a source of mystery and excitement in equal parts and mastering them have been a human endeavour that have been unfulfilled for the most of the humans that have existed. These games are so complex as each move in any of these games involve a large number of possibilities that until recently have been thought to be infinite. However, with the advancement into computing, that factor has been sort of nullified by the establishment of databases and remote servers that bring with them an unlimited amount of memory to compute an almost infinite amount of possibilities. Therefore, that leaves only the problem of how to play these games in the best possible way and find the best possible moves, which we are trying to deal with the lookahead searches using trees.

MuZero is latest among a host of chess engines that have claimed to be the best chess engines. However, that title can safely be given to MuZero as it beat StockFish (it’s nearest competitor) in a match of 100 games, where MuZero won 72 of those games, while remaining games were drawn and MuZero lost none of them.

More than the number of wins or the absence of loss, the fascinating part was the way they won. Therefore, we aimed to apply such an engine on Connect-four to check out how it works on the slightly complex but solved game.

**Connect Four**is a very popular connection board game worldwide. It Is a specifically two player game where the player is given the choice of one of the two colors namely red and yellow. The players then, take turns in dropping that specific colored discs in a vertically suspended grid, which is essentially a matrix of seven columns and six rows. The lowest available space in the columns is occupied by the disc as the pieces fall down. The game ends when four discs of the same colour occupy the same horizontal, vertical or diagonal four spaces. It is said to be a solved game where the 1st player can always win if they play the right moves.

It is also known as

·         **Gravitrips** in the USSR

·         **Four Up**

·         **Plot Four**

·         **Find Four**

·         **Four in a Row**

·         **Four in a Line**

·         **Drop Four**

In a two-player game, Connect Four has the perfect information for both players that are playing the game. The games where one player at a time plays and the players have all the knowledge of the moves that have taken place and all moves that can take place, at a given time, is described by the term “perfect information”. Connect Four can be classified as an adversarial, zero sum game as in the game, some one’s advantage is the opponent's disadvantage.

For a Connect Four game the number of possible games board positions is used as a measure of complexity. There are 4,531,985,219,092 positions[[5]](https://en.wikipedia.org/wiki/Connect_Four#cite_note-oeis-5) for all game boards populated with 0 to 42 pieces for classic Connect Four played on 6 high, 7 wide grid.

James Dow Allen (October 1, 1988), and independently, Victor Allis (October 16, 1988) solved the game for the first time. A knowledge-based approach was taken by Mr. Allis and he worked out nine strategies as the solution to the game of Connect-four. Winning strategies was also given by Allen in his analysis of the game. Brute-force analysis was not deemed feasible because the game was too complex for the computer technology available at the time when initial solutions for Connect-Four were found. Beginning with John Tromp's work in compiling an 8-ply database, brute-force methods have been used to solve the Connect-4 game (February 4, 1995).

Connect Four’s solution is that first player wins. The first player can force a win with perfect play, however, on or before the 41st move by starting in the middle column. When the first player starts in the columns adjacent to the centre, the game is a theoretical draw. By starting with the four outer columns, the first player allows the second player to force a win.

We’ll see in subsequent chapters in great detail how the MuZero engine works and how it was implemented for the game of Connect four.



**2.**

**Literature Review**

**2.1. Previous Research (in terms of the current study)**

It is important to first acknowledge how we got here. In the 1920s, two scientists proposed a theory which said that if we can somehow mimic the functioning of a single neuron and use them to make an artificial network of neurons that are present in the brain, we can make theoretically make a fully functional artificial brain. But their idea was too ahead of their time and humans simply didn’t have enough computing power yet to create them. Their idea was seemingly lost, when in 1980s, it was revived again to theoretically create an artificial network of neurons. This idea still took some time to be accepted and when in 1990s and 2000s, as the computer became much 19stronger, Artificial Neural Network (ANN) became a scientifically viable endeavor and something that has directly led to Mu Zero.[5]

Around the time this was happening, there were a lot of people trying to make chess engines, which are arguably our first attempt to make computer play and master an ancient human game. From the 1960s till the 1990s, there were a lot of engines that made the claim of “having solved the game of Chess”. However, closer inspection revealed otherwise. This all culminated into the historic match of 1996 between IBM’s Deep Blue[1] computer and the then-world champion of chess, Gary Kasparov. Deep Blue won the first game of the six-game match.  However, Kasparov won three and drew two of the following five games, defeating Deep Blue by a score of 4–2.[1]

However, in 1997, in the rematch, Deep Blue[1] won game six, thereby winning the six-game rematch 3½–2½ and becoming the first computer system to defeat a reigning world champion in a match under standard chess tournament time controls.[1]

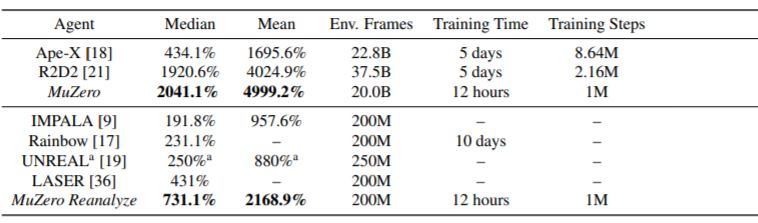
Although, chess engines had beaten human grandmasters before this, this was the first computer to beat one under standard time control, this realised an old quest of mastering chess. Since then, a lot of chess engines had come into the market. Each better than the last one. Latest one of these engines was StockFish.[6, 7]

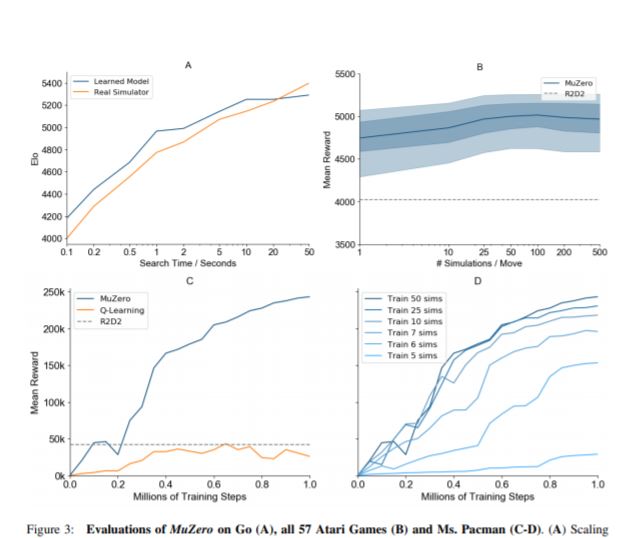
After Chess, Go[2] had been the next challenge and had been a long-standing problem in the field of Artificial Intelligence (AI). With Chess, there are 20 possibilities per move, with Go that number goes up to 200. This was the problem that the people at DeepMind wanted to solve. There effort started with the development of AlphaGo, an engine made to play Go. However, it needed to be tested. To do that test, their team went to European champion of Go[2] at the time, whose name was Fan Hui. They organised an official match between Fan Hui and AlphaGo, and the machine won five-nill in the five-game match. It was again a historic moment. However, Fan Hui was only a 2-p professional, where as the world champion was 9-p in the Go rating system. Therefore, the Go world was confused as to how to perceive this victory and most were sceptical about the abilities of the program.[2]

Therefore, the DeepMind team challenged the world champion in Go, Lee Sedol, who was not only the current world champion but had been so for 18 times before that too. Therefore, it was fair to say that DeepMind was facing the genius human at Go. [2]

The match was again a five-game match played at Seoul in South Korea. In the first game, the machine beat Lee Sedol comfortably making a move that was said to be a genius move, beyond the capabilities of a normal human being. A move that had the probability of 1-in-10000 and was dubbed a God move.  Lee Sedol went on to lose the next two games too and with that the match. In game four of the match, Lee Sedol led from the start and made a move which made the AI “crazy” as it started to make non-sensical moves as it realised that it was going to lose. On closer analysis, it was found that this move by Lee was also 1-in-10000 probability move and therefore, there is room for improvement as the AI can’t really match human intuition yet. The last match was a close one and AlphaGo just wins by a small margin. Therefore, the AI wins four - one in the five-game match.[2]

 Since then, AlphaGo[2] and its subsequent new versions namely, Alpha Zero[8] and then, Mu Zero, which both take a slightly different approach to decision making, have mastered several games namely Shogi, Star craft II, Chess, 57 different Atari games etc. and are considered to be at the forefront of AI technology. Therefore, we shall try to understand the workings of MuZero and how these different versions are different from each other.[8]





Comparison of MuZero with older engine and methods

Many experiments were run for understanding the model’s role in MuZero, which focused on the Atari game of Ms. Pacman and the board game of Go. We tested the scalability of planning was tested first. This was done in Go’s canonical planning problem. The performance of search in MuZero using a learned model, was compared to the performance of search in AlphaZero using a perfect model. Most importantly, MCTS with different thinking times was compared to evaluate the fully trained AlphaZero or MuZero. Even when a MuZero model is trained for much smaller searches (around 0.1s thinking time) than the larger searches (up to 10s thinking time) that it is end up being used on, it is seen to perform well. In the paper, the scalability of planning across all Atari games was also investigated. MCTS with different numbers of simulations was compared, which used a fully trained MuZero. The improvements are marked less than in Go much more than planning, perhaps because of greater model inaccuracy. However, with search time, performance improved slightly but plateaued at around 100 simulations. Even with a single simulation – i.e. when selecting moves solely according to the policy network – MuZero performed well, suggesting that, by the end of training, the raw policy has learned to internalise the benefits of search. Next, we tested our model-based learning algorithm against a comparable model-free learning algorithm. We replaced the training objective of MuZero (Equation 1) with a model-free Q-learning objective (as used by R2D2), and the dual value and policy heads with a single head representing the Q-function Q. Subsequently, we trained and evaluated the new model without using any search. When evaluated on Ms. Pacman, our model-free algorithm achieved identical results to R2D2, but learned significantly slower than MuZero and converged to a much lower final score. We conjecture that the search-based policy improvement step of MuZero provides a stronger learning signal than the high bias, high variance targets used by Q-learning. To better understand the nature of MuZero’s learning algorithm, we measured how MuZero’s training scales with respect to the amount of search it uses during training. Figure 3D shows the performance in Ms. Pacman, using an MCTS of different simulation counts per move throughout training. Surprisingly, and in contrast to previous work [1], even with only 6 simulations per move – fewer than the number of actions – MuZero learned an effective policy and improved rapidly. With more simulations, performance jumped significantly higher. For analysis of the policy improvement during each individual iteration

MuZero rely on ANN which makes the latest and best AI that can play and win a game using Neural Networks

**Artificial neural networks** (**ANNs**) are computing systems that are vaguely inspired by the biological animal neurons that make up the animal brain. They are usually known as Neural Networks.

An ANN is made up of a collection of connected units or nodes which are called artificial neurons, as they were modelled on the neurons in a biological brain. Each of these connections can transmit a signal to other neurons just like the synapses in a biological brain. An artificial neuron signals the neurons connected to it after receiving signals from other neurons and processing it. A real number represents the "signal" at a connection, and some non-linear function of the sum of its inputs computes the output of each neuron. The connections are also known as *edges*. As the learning proceeds, the weight of neurons and edges adjusts accordingly. The strength of the signal is increased or decreased by the weight at a connection. Some Neurons may have a threshold, if the aggregate signal crosses the threshold, a signal is sent. Typically, neurons make up the layers that perform the functions. Different layers may perform different transformations are performed by different layers on their inputs. The first layer (the input layer) receives the input and the last layer (the output layer) gives the output, possibly after traversing the layers multiple times.

Example data is used to train the neural networks and make them learn which “input” will provide what “result” as the data is made up of “inputs” and “results”. The neural net does that by making probability-weighted associations between the two, saved within the data structure of the net. The difference between the processed output and the target output is determined, which is done to train the neural net. This is the error. The network’s weighted associations are adjusted according to a learning rule where this error value. The neural network produces output similar to the target with the help of successive adjustment. The certain criteria used to terminate the training after a sufficient number of adjustments are done.

Examples are considered which helps such systems “learn” to perform tasks, usually without being programmed with task-specific rules. For eg., They can be trained to detect the images with any cat in it by training the model with the help of the images labelled by a person as “cat” or “no cat”. The model which is the result of this process can be used to identify images containing cats. They don’t know that the cat has fur, tails, whiskers and cat-like faces as they have no prior information about cats or how they look.

**Components of ANNs**

#### Any Artificial Neural Network contains three main components, namely:

#### *Neurons*

As the human brain is made of biological neurons, artificial neurons make up the Artificial Neural Network(ANN) Inputs of each neuron produce a single output that can be in turn sent to multiple neurons to produce their output. The inputs can range from sample value of an external data, which includes images, document etc., to outputs of other neurons. The output of the output neuron at the end accomplish the task such as translating a part of tex.

We take all inputs weighted sum which are weighted by the weights of the connections of the neuron to the inputs to find the output of the neurons. A bias term to this sum is added, which s usually called “activation”. To get the output, an activation function is used which is usually non-linear and the sum is passed through it. External data such as images, documents etc. are used as initial inputs while accomplishing any task such as recognizing an image are the ultimate output.

#### *Connections and weights*

#### Connections make up the network. Each connection provides the neuron’s output as an input to another neuron. The relative importance of a connection is represented by the weight which is assigned to each neuron. Multiple input and output connections are present in any neuron.

#### *Propagation and Backpropagation*

#### The input to a neuron from the outputs of its predecessor neurons and their connections as a weighted sum is computed by the propagation function. The result of the propagation is added with a bias term.

The process of adjusting the weights so that the error found during learning is compensated is called Backpropagation. among the connections, the effective division of this error amount happens. Technically, with respect to the weight, the cost function’s gradient associated with a given state is calculated by back prop. stochastic gradient descent or other methods, such as Extreme Learning Machines, "No-prop" networks, training without backtracking, "weightless" networks, and non-connectionist neural networks are done to perform weight updates.

Reinforcement learning plays a big part in MuZero and the functions of the network are made

Model-based and model-free [42] are the two principal categories that Reinforcement learning is subdivided into. A model of the environment is constructed by the model-based RL as an intermediate step.

Classically, a Markov-decision process (MDP) represents this model [31]. It has two components:

a state transition model (which predicts the next state)

and a reward model, (which predicts the expected reward during that transition).

The model is typically conditioned on the selected action, or a temporally abstract behavior such as an option are used typically to condition the model [43]. The optimal value or optimal policy for the MDP is computed using planning algorithms like value iteration [31] or Monte-Carlo tree search (MCTS) [7] after the model has been constructed. The state representation is constructed by the algorithm, that the model is made to predict, in large and/or partially observed environment. The agent’s unable to optimize its representation or model for the purpose of planning, therefore, modelling errors may increase during planning as the representation learning, planning and model learning have a separation between them, making it potentially problematic.

A common approach to model-based RL focuses on direct modelling of the observation stream on the pixel level is a common approach that the model-based RL focuses on. It has been hypothesized that the compounding error problem can be mitigated by deep, stochastic models [14, 20]. In large-scale problems, however, pixel-level granularity’s planning is not tractable computationally. Other methods build a latent state-space model that is sufficient to reconstruct the observation stream at pixel level [48, 49], or to predict its future latent states [13, 11], it is sufficient to build a latent state-space model is built by other methods. More efficient planning is facilitated by this method but still the majority of the model capacity is concentrated on irrelevant detail potentially. In visually complex domain like Atari, the results of these prior methods lag well behind model-free methods (even in data efficiency) as none of them constructed a model that facilitates effective planning [45]. A recent approach to model based Reinforcement learning was also introduced focusing on end-to-end on predicting the value function [41]. Constructing an abstract MDP model is the main idea behind these methods. It is such, that planning in the newly constructed MDP is equivalent to planning in the real environment.

The cumulative reward of a trajectory through the abstract MDP equals the cumulative reward of a trajectory in the real environment, when starting from a real state, which is known as value equivalence. for predicting value (without actions), value equivalent models were first introduced by The predictron [41]. There is no requirement for its transition model to match real states in the environment, even though the underlying model still becomes an MDP. A hidden layer of a deep neural network is how the consequent MDP model is viewed as, instead. The expected cumulative sum of rewards matches the expected value with respect to the real environment in the process of training an unrolled MDP, e.g. by temporal-difference learning.

So, let’s see how Alpha Go works – at the very basic, the AI is fed rules of the games and database of a collection of strong amateur games to help it take better decisions. This AI then, plays millions of games with itself and trains itself to play the game with new players.

After Alpha Go, came Alpha Zero[8], this was made to play chess and after training for around four hours, it beat Stock Fish in a 100-games match, where there were only 27 draws and zero loss, while it won every other game. The only difference it has from Alpha Go was that it was not fed only the rules and not any old games, making it a proper example of unsupervised learning. It then, played millions of games with itself to train itself better. This was important to remove the biases that a human payer may have.[8]

**2.2. Rationale of the current study, Aims or hypothesis**

After Alpha Zero, came MuZero[9] where the Ai was not even given the rules of the games or even the objective of the game. It is just put in a game environment and asked to play. Here, the Mu Zero learns everything about the game as it plays the games like any human child will. More importantly, like any genius child will learn the game and apply itself. This removes any bias that the rules may bring and also, allows the creators to put the AI in an unknown environment and make it learn. This allows a scope to put the AI in an unknown environment other than chess and hopefully, get more insight into the task from the “genius kid”. MuZero[9] builds upon AlphaZero’s powerful search and search-based policy iteration algorithms, but incorporates a learned model into the training procedure. MuZero also extends AlphaZero to a broader set of environments including single agent domains and non-zero rewards at intermediate time-steps.[9]

If we go into the technical aspects of this, MuZero[9] is an Artificial Neural Network made with the combination of Convoluted Neural Network (CNN) and Recurrent Neural Network (RNN) and the process through which it chooses the right decision to take is known as Monte-Carlo Tree Search Algorithm. So, it is a machine trained through Reinforcement Learning which uses Monte-Carlo Tree Search to do a look ahead search of the best actions to take while maximising the reward that can be achieved.[9]

MuZero [9] involves building agents with planning capabilities. Monte-Carlo Tree search have been very successful in domains, such as chess and Go, where a perfect simulator is already available. However, in real-world problems, the environments are often complex and unknown. The Mu Zero program combines tree-based search with a learned model while achieving superhuman performance in challenging and visually complex domains, without any knowledge of their underlying dynamics or rules. MuZero[9] has a model that, when applied iteratively, predicts the quantities most directly relevant to planning: the reward, the action-selection policy, and the value function. Mu Zero achieved a new milestone when the algorithm was evaluated on 57 different Atari games, the canonical video game environment for testing AI techniques, in which model-based planning approaches have historically struggled. When evaluated on Go, chess and shogi, without any knowledge of the game rules, MuZero matched the superhuman performance of the AlphaZero algorithm that was supplied with the game rules.[9]

MuZero uses a model-based reinforcement learning (RL) [10] which aims to address the issue of being in an unknown environment by planning with respect to the learned model of the environmental variables that it creates in the first step. Typically, these models have either focused on reconstructing the sequence of full observations [11, 12] or the true environmental state[13-15]. However, prior work wasn’t up to par in visually heavy domains, such as Atari games environment. Instead, the most successful methods are based on model-free RL[10], a model that estimate best policy and value function directly from interactions with the environment. However, model-free algorithms are in turn far from best in domains that require precise and sophisticated lookahead, like the games of chess and Go.

A state-of-the-art performance in Atari 2600 is achieved by MuZero using a newer approach to model-based RL. The Atari environment is a visually complex set of domains where it gives such a performance while maintaining superhuman performance in tasks such as chess, shogi and Go.

Predicting the aspects of the future that are immediately relevant to the planning is the main idea of the algorithm. The observation (for e.g. an image of the Go board or the Atari screen) as an input is received by the model and transformed into a hidden state by it. A recurrent process is used to iteratively update the hidden state, which receives the previous hidden state and a hypothetical next action. At every one of these steps the model predicts the policy (e.g. the move to play), value function (e.g. the predicted winner), and immediate reward (e.g. the points scored by playing a move). The model is trained end-to-end, with the sole objective of accurately estimating these three important quantities, so as to match the improved estimates of policy and value generated by search as well as the observed reward. There is no direct constraint or requirement for the hidden state to capture all information necessary to reconstruct the original observation, drastically reducing the amount of information the model has to maintain and predict; nor is there any requirement for the hidden state to match the unknown, true state of the environment; nor any other constraints on the semantics of state. Instead, the hidden states are free to represent state in whatever way is relevant to predicting current and future values and policies. Intuitively, the agent can invent, internally, the rules or dynamics that lead to most accurate planning.

**3.**

**Methods**

One of the main principles that MuZero uses to train its neural network is known as reinforcement learning.

**Reinforcement learning** (**RL**) is a machine learning related area of study which describes how software agents take actions in order to maximize the cumulative reward, in an environment it has been put in. Reinforcement learning is one of three basic branches of machine learning, where the other branches are supervised learning and unsupervised learning.

Reinforcement learning paradigm is different from any supervised learning model in not requiring input and output pair that are labelled and in not explicitly correcting the sub-optimal actions Instead exploration of uncharted territory and exploitation of current knowledge is tried to be used in balance.[5]

Markov decision process (MDP) form is needed therefore the environment is made in that form, as dynamic programming techniques are used by many reinforcement learning algorithms. The reinforcement learning algorithms do not assume information of an exact mathematical model of a MDP and they target large MDPs where exact methods become infeasible which is the main difference between classical dynamic programming methods and these algorithms.

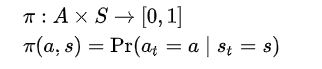
Even if the issue of exploration is disregarded and even if the state was observable (assumed hereafter), past experience is utilized to find out which actions lead to higher cumulative rewards, which is the main problem even if we ignore the issue of exploration and assume that the state can be observed.

### Criterion of optimality

### The reinforcement learning algorithms find the optimal agents using some criteria of optimality. These are: -

### *Policy*

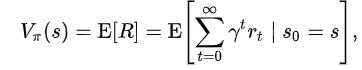
A map is modelled using the agent’s election of action. This is called *policy*.



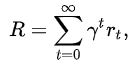
The probability of taking action (a) while in a certain state (s) is provided by the policy map. Non-probabilistic policies are also present.

***State-value function***

The *expected return* going from state(s), i.e.,   and following policy()till the end, is known as State-value function () . Hence, roughly speaking, the value function estimates "how good" it is to be in a given state.



where the random variable R denotes the **return**, and is defined as the sum of future discounted rewards (gamma is less than 1, as a particular state becomes older, its effect on the later states becomes less and less. Thus, we discount its effect).





The algorithm must find a policy with maximum expected return. From the theory of MDPs it is known that, without loss of generality, the search can be restricted to the set of so-called *stationary* policies. A policy is *stationary* if the action-distribution returned by it depends only on the last state visited (from the observation agent's history). The search can be further restricted to *deterministic* stationary policies. A *deterministic stationary* policy deterministically selects actions based on the current state. Since any such policy can be identified with a mapping from the set of states to the set of actions, these policies can be identified with such mappings with no loss of generality.

**Brute force**

The brute force method is a type of reinforcement learning that includes two steps:

·         Sample returns of each possible policy when following that policy.

·         The largest expected return from which ever policy is achieved, is chosen.

The large or even infinite number of policies, that can happen, is a problem of brute-force method. The variance of the return may be large which is another problem with it as an accurate estimate of the return of each policy required many samples.[5]

Assuming some structure and allowing the samples to be generated is the able to ameliorate the problem and it can be achieved by value function estimation and direct policy search.

Value function

Value function maintains a set of predictions of expected returns for any policy (be it “current” [on-policy] or the optimal one[off-policy]) to approach attempt to find a policy maximizing the return.

These methods are reliant upon the theory of Markov states and Markov Decision Process. In these theories, optimality is formally defined as: Any policy is known as the optimal policy if the best expected return is achieved from any starting state (which means the starting positions plays no role in this definition). Optimal policies are usually found among stationery policies.

The value of a policy(  ) is defined by



where R stands for the return associated with policy ( ) from the initial state (s) .

Defining   as the maximum possible value of  , where   is allowed to change,



Any policy that has reached to these optimal values in each state is known as *optimal*. Clearly, an optimal policy by definition increases the expected return  to the maximum, since  , where S  is a state randomly sampled from the distribution .

Although optimality can be defined by state values, defining action-values is also very useful. Given any state s,

an action a

and a policy   ,

the action-value of the pair (s,a)  under   is defined by



where R now stands for the random return associated with first taking action  a in state s  and following  , thereafter.

If   is an optimal policy, we act optimally (take the optimal action) by choosing the action from   with the highest value at each state (s). The optimal policy’s  *action-value function* is called the *optimal action-value function* and is commonly denoted by  .

In short, the optimal action-value function’s knowledge alone is enough to calculate the optimal action to be performed.

Value iteration and policy iteration are the two basic approaches to calculate the optimal action-value function. a sequence of functions ( ) that converge to   is computed by both the algorithm. expectations over the whole state-space is computed to calculate these functions, which is impractical for all MDPs except some (they are practical for the smallest (finite) MDPs. Samples are averaged over and function approximation techniques are used to represent the value functions over large action-state spaces and to primarily approximate expectations in reinforcement learning methods.

***Monte Carlo methods***

Policy iteration is mimicked by an algorithm that uses the Monte-Carlo methods. Policy iteration has of 2 steps:

*policy evaluation* and *policy improvement*.

The policy evaluation step uses the Monte-Carlo methods[6]. If a stationary, deterministic policy   is given, the computation of the function values  is the goal (or at least a good approximation) for each and every state-action pairs(s, a)  . With the assumption that the MDP[8] is finite, the problem is episodic and sufficient memory is available to store the action-values and after every episode the latest one starts from any random initial state. Then, averaging the sampled returns that originated from (s, a) over time can help calculate the predictions of the value of a given state-action pair (s, a). Given sufficient time, this procedure can thus construct a precise estimate Q of the action-value function  . This finishes the description of the policy evaluation step [8].

In the policy improvement step, the next policy is obtained by computing a *greedy* policy with respect to Q  : Given a state s, an action that maximizes   is returned by the new policy. The calculations associated with maximizing actions can be deferred to when they are needed in lazy evaluation. [2]

Disadvantages of this procedure includes:

·         A suboptimal policy may be evaluated for a long time by the procedure.

·        The estimation of the *single* state-action pair that started the trajectory is improved by a long trajectory as the procedures uses samples inefficiently.

·         The convergence is slow when the trajectories along the returns have *high variance*

·         The episodic problems are the only ones that these procedures work on;

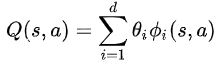
·         The small, finite MDPs are the only ones they work on.

***Temporal difference methods***

The procedure is allowed to change the policy before the value settles which in turn corrects the first problem. Convergence may be prevented by it which too may lead to problems. The class of *generalized policy iteration* algorithms are created because the most recent algorithm gives rise to them [5]. This category has many *actor critic* methods.

Trajectories are allowed to contribute to any state-action pair in them which solves the second issue that usually arises. This may help with the third problem to some extents also. Although when returns have high variance, a better solution is Sutton’s temporal difference methods developed from the Bellman equation. In Temporal Difference, computation can be incremental or batch. Incremental computation is defined as when after each transition the memory is changed and the transition is thrown away, and batch computation is defined as when the transitions are batched and the estimates are computed once based on the batch. The information in the samples can be better used by batch methods for e.g., least-squares temporal difference method, while when batch methods are infeasible due to their high complexity, the incremental methods are the only choice. Some methods try to combine the two approaches. The fourth issue is also overcome by methods based on TD[8,9].

*Function approximation methods* are used in order to address the 5th issue. *Linear function approximation* starts with a mapping   that assigns a finite-dimensional vector is assigned to each state-action pair (s, a) by the help of mapping that happens at the start of the Linear function approximation. Then, the components of   with some *weights*   are linearly combined to obtain the action values of a state-action pair:



The algorithms then adjust the weights are then adjusted by the algorithm, rather than adjusting the individual state-action pair’s values. Exploration of methods based on ideas from nonparametric statistics which can make up their own features have been done.[13]

The Q-learning algorithm and its many variants are given rise to by using value iteration as a starting point.

Highly precise predictions of the clashing action values may be required which can cause some problems with using action-values. These action-values can be hard to obtain when the returns have noise in them. Temporal difference methods mitigate these problems to some extent. Generality and efficiency are compromised by the use of the compatible function approximation methods. Another temporal difference specific problem occurs because it is recursive Bellman equation reliant. This can be solved by using the parameter  of TD methods which helps in continuously interpolate between Monte-Carlo methods (not relying on Bellman eq.) and the basic Temporal Difference (Bellman eq. reliant). This issue can, thus, be solved in this way.

**Direct policy search**

Another method is to searching directly in the policy space is another method of reinforcement learning, in which case, it is a stochastic optimization problem[5]. In stochastic optimization, gradient-based and gradient-free methods are the two approaches available.

Mapping from the parameter (finite-dimensional) space to the space of policies is performed: let

the parameter vector be   ,

the policy associated to   is denoted by 

the performance function is defined by



as a function of the parameter vector , this function will be differentiable under mild conditions. Gradient ascent can be seen if the gradient of  was known. As an analytic expression for the gradient is unavailable, only an estimate which is nosy is available. Williams' REINFORCE method (the likelihood ratio method in the simulation-based optimization) can be used to construct such an estimate among many others. Robotics research has also used such policy search methods. Many policy search methods are based on local search [12]. Therefore, they may get stuck in local optima.

Reliance on gradient information is avoided by a huge class of methods[16]. For e.g., methods of evolutionary computation, simulated annealing etc. In theory, a global optimum can be achieved by a large number of gradient free method.

Due to noisy data, the convergence of policy search methods happens slowly. For e.g., when the trajectories and variance of the returns are large as the problem is an episodic one. Temporal differences reliant value-function based method may be of help in this case. The proposal and the good performance of *actor–critic methods* have occurred just in the recent years[18].

MCTS was used in AlphaZero’s Neural Network albeit in a different way.

The general-purpose Monte-Carlo (MCTS) algorithm is used in AlphaZero in place of alpha-beta search with domain-specific enhancements which is more common. Every search contains sequence of simulated self-play games which travels through a tree from the start state sroot  until a leaf node is reached. A move a with low visit count (not frequently explored), high value (averaged over the leaf states of simulations), and high move probability according to the current neural network fθ is selected at the start of every simulation. A vector π is returned which provides a probability distribution over moves, πa = Pr(a|sroot). [2]

Reinforcement Learning using self-play is used to train the parameters θ of the Deep Neuron Net in AlphaZero which actually starts from values of parameters θ are randomly initialised. In every game, the current position is taken as the root node sroot = st at the specific turn *t* as MCTS[5] is performed from the current position to select a move at ~ πt . This can be done while following a proportional policy (which concentrates on exploration) or a greedy policy (which concentrates on the exploitation) w.r.t. the visit count at the root position. The game outcome (z: −1 for losing, 0 for drawing, and +1 for winning) is computed according to the laws of the game and the terminal position (sT) is scored according to it at the end of the game. The similarity of the policy **p**t to the probabilities πt is increased to a maximum and the error between prediction (vt) and the real outcome is reduced to minimum by the updating of the neural Network parameter θ. The loss function used is obtained by adding over cross entropy losses and MSE (mean-squared error). The gradient descent on this loss function is performed by adjusting the parameters θ. In subsequent self-play games, the updated parameters are used [4].

We now describe the MuZero algorithm in more detail.

Predictions are made at each time-step t, for each of k = 1...K steps, by a model µθ, with parameters θ, conditioned on past observations and future actions  conditions the model µθ, with parameters θ. At each time-step, which includes k=1…K steps, predictions are made by this model [3]. Three future quantities:

* the policy ,
* the value function  ,
* and the immediate reward  ,

where the true (observed) reward is u,

the real actions are selected by policy π,

and the discount function is γ.

Specifically, A combination of a representation function, a dynamics function, and a prediction function is used to represent the model at any time-step t.

An immediate reward r k and an internal state s k is computed by the dynamics function  in a recurrent process at any hypothetical step k. This model behaves like a MDP model that computes the expected reward and state transition for any given state and action[17].

But no semantics of environment state attached to this internal state , unlike traditional approaches to model-based RL [10]. Future quantities (policies, values, and rewards) should be accurately predicted and that is the sole purpose of this state. This state is the hidden state of the overall model.

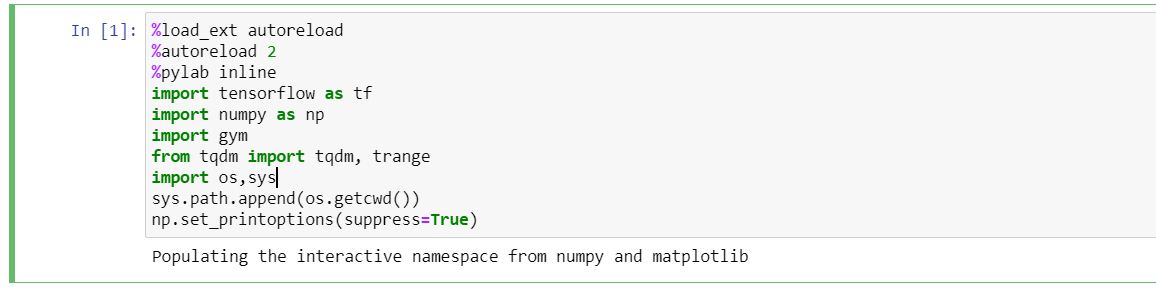
In this paper, the dynamics function has a deterministic representation. The policy and value functions are computed from the internal state s k by the prediction function,  is used to convert the internal state s k into policy and value functions. The “root” state s 0 is initialized using a representation function which stores past observations, the “root” state  is initialized. Above its support for future predictions, this has no special semantics. Hypothetical future paths [ a 1 , ..., ak] based on the past observations[o1, ..., ot] can be searched over by such a model. For e.g., the k step action sequence which maximizes the value function is selected when naïve search is used. Usually, the internal rewards and state space is operated on by any MDP planning algo. The Monte-Carlo Tree Search is similar to AlphaZero’s search, specifically speaking, which is generalized to allow for single agent domains and intermediate rewards. The current model parameter θ produces policy, value and reward estimates which is used by MCTS at each internal node. A policy πt and estimated value νt is given as output by the Monte-Carlo Tree Search Algo. Which in turn leads to the selection of an action at+1 ∼ πt. The policy, value, and reward, are accurately matched by the jointly training all the parameters of the model. This is done for every hypothetical step k, to target values which were observed after the conclusion of k actual time-steps. As with AlphaZero, an MCTS search is used to generate improved policy targets. Minimising the error between predicted policy and search policy is the first objective. Also like AlphaZero, the game is played or MDP is used to generate improved value targets. However, unlike AlphaZero, with discounting and intermediate rewards, we allow long episodes to run. This is done when we bootstrap n steps from the search value . On the final step of the episode, specific rewards are given to each final outcome {lose, draw, win}. Specifically, minimizing the error between the value target and the predicted value ()is the second objective. The observed rewards are the reward targets; the third objective is therefore to minimizing the error between the predicted reward (r k t) and the observed reward (ut+k) is the third objective. Finally, an L2 regularization term is also added:

 where loss functions for reward, value and policy respectively are . [2]

**4.**

**Implementation**

The paper that described MuZero was written by the Google DeepMind[3,5]. In the original paper, they explained the algorithm and provided a pseudo code, this was used by us to develop our very own MuZero engine which we tested on the game known as Connect-4. The code that came out of the project is explained as follows:



Here, we imported all the libraries that are needed in this project. This includes the all-important Deep Learning library of Python known as Tensorflow [9]. Other than that we have the numpy library which provides us the power to do numerical operations on n-dimensional array which was very important as the project is heavily reliant on vectors to create a working Connect4 model. The gym by OpenAI, OS, SYS and pylab libraries were also included to help create the proper training environment. The tqdm library to create a progress

In this part of code, we defined some functions that were to be used later in the code. Reformat\_batch, to\_one\_hot, and bstack function is used while training the model.



Then, the MuZero Model is defined as the class MuModel, it contains the representation function, the dynamic function and the prediction function[2].

The representation function helps the AI create and store a virtual model of the environment around it, which it creates while walking through the environment.

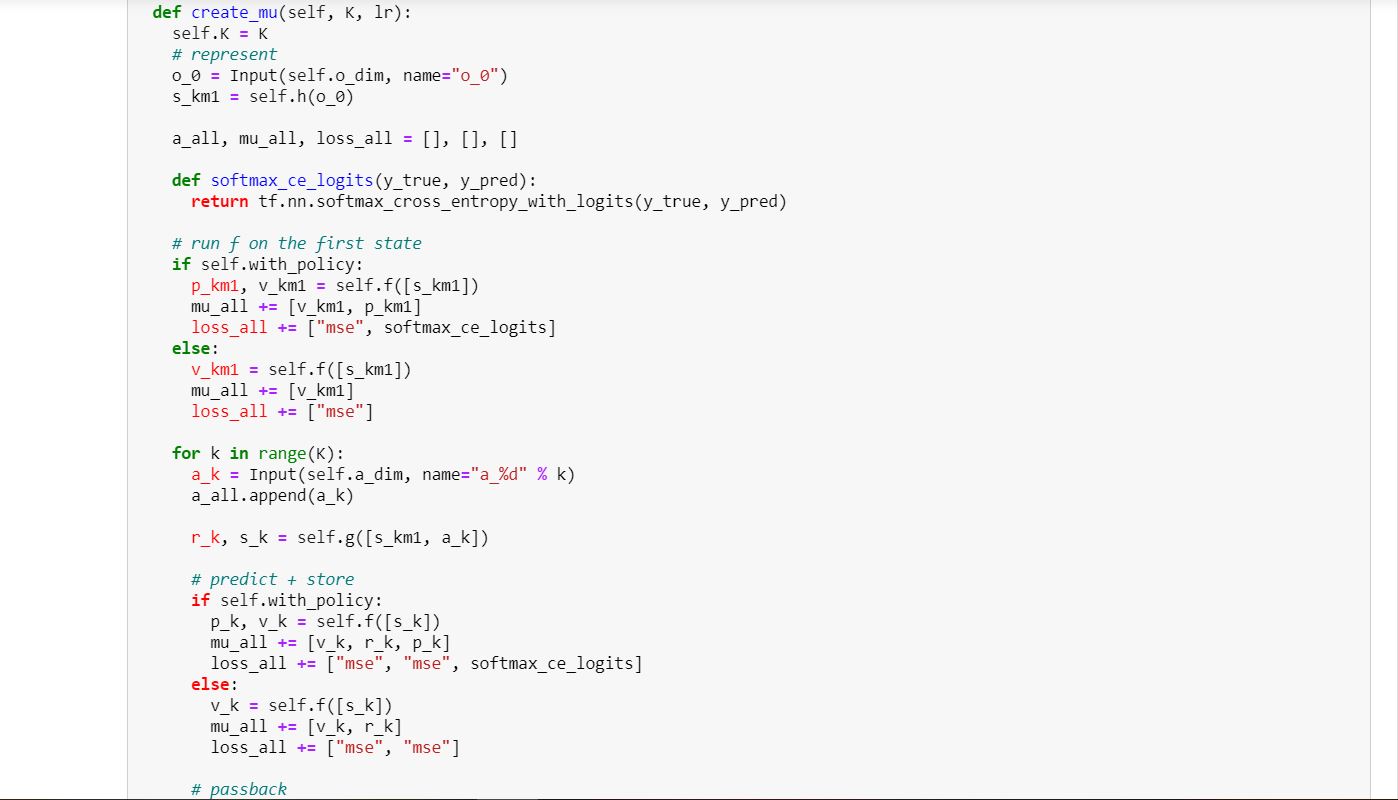
The dynamic function helps in converting the old state and action into a new state and reward.

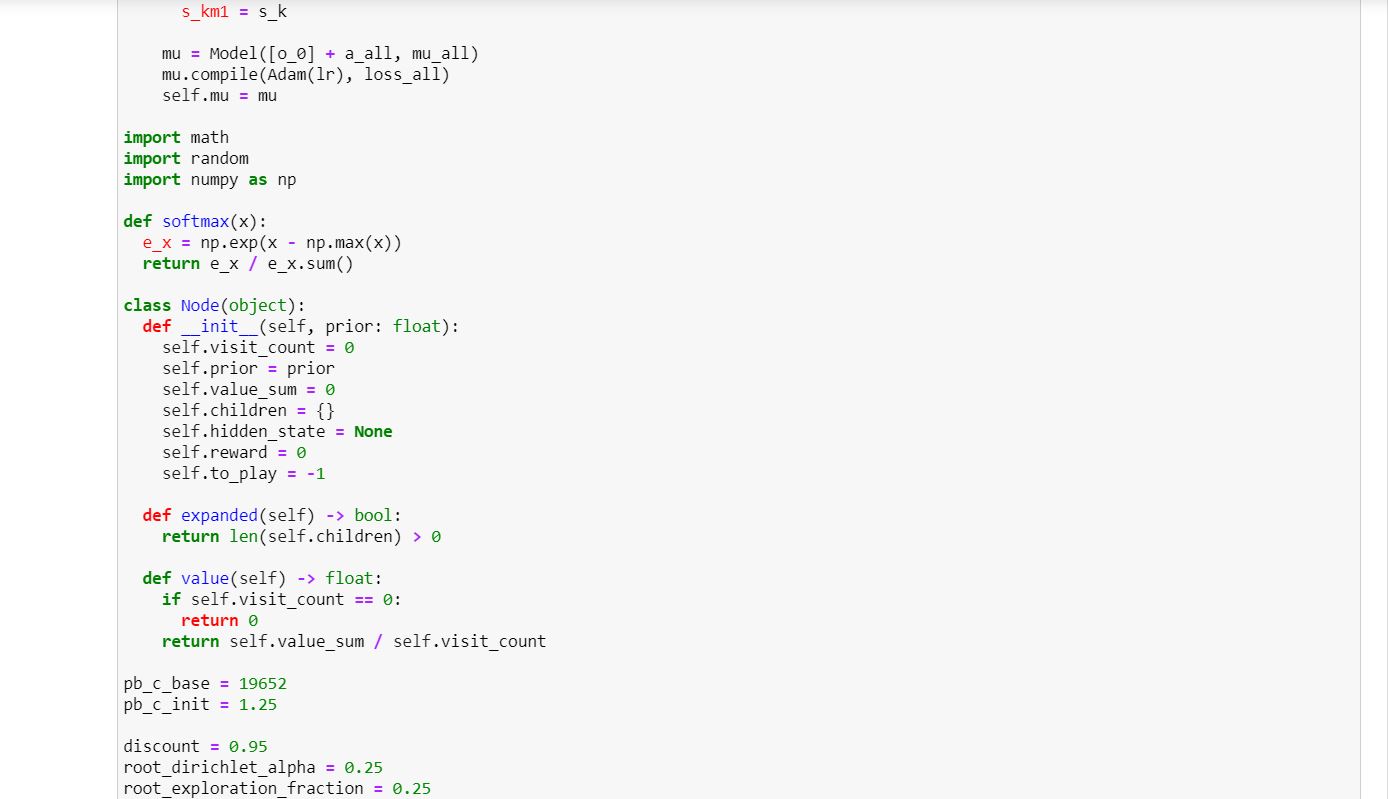
The prediction function makes predictions for each state where it outputs policy and value for each available state.

We call these functions through other functions like ht, gt and ft.

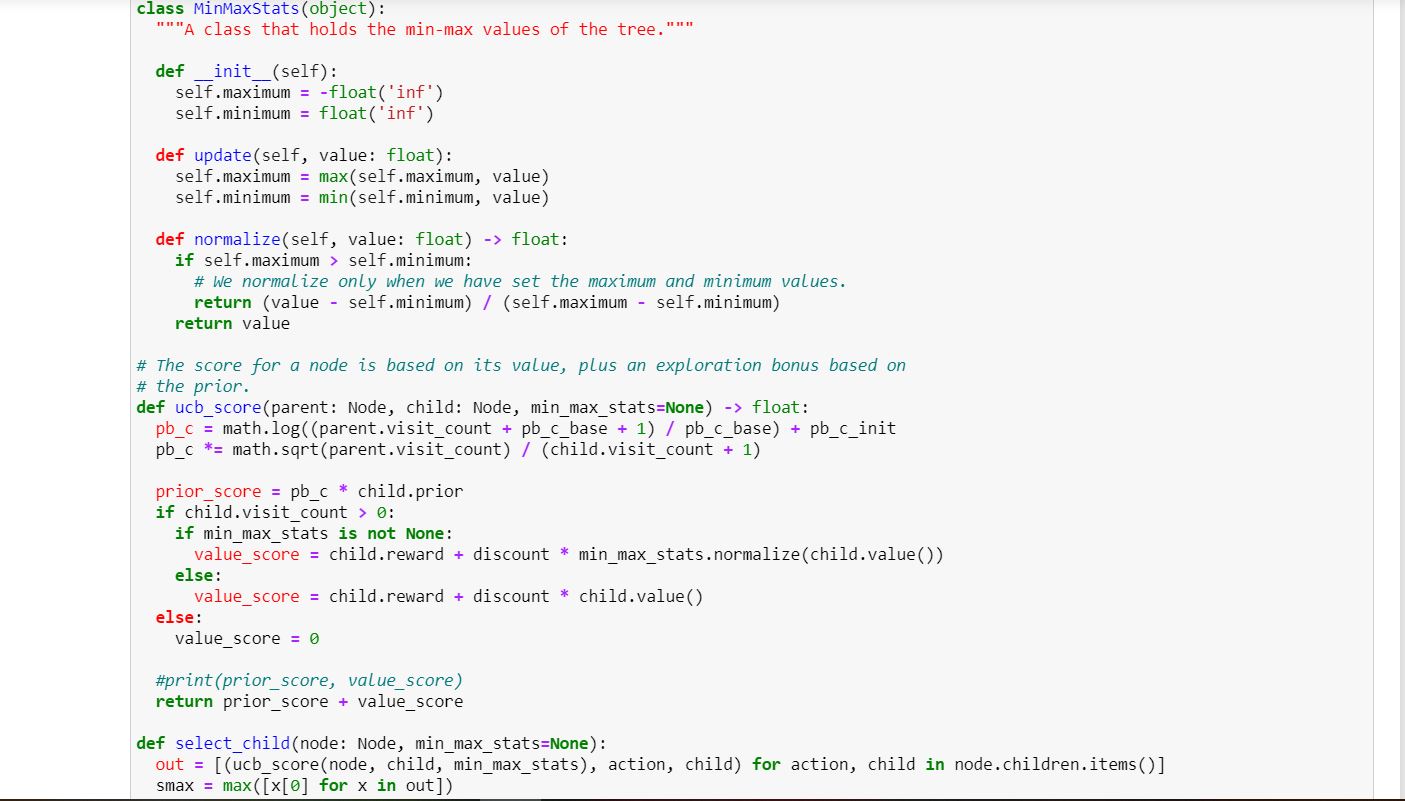
create\_ mu is to combine all those function into a model-



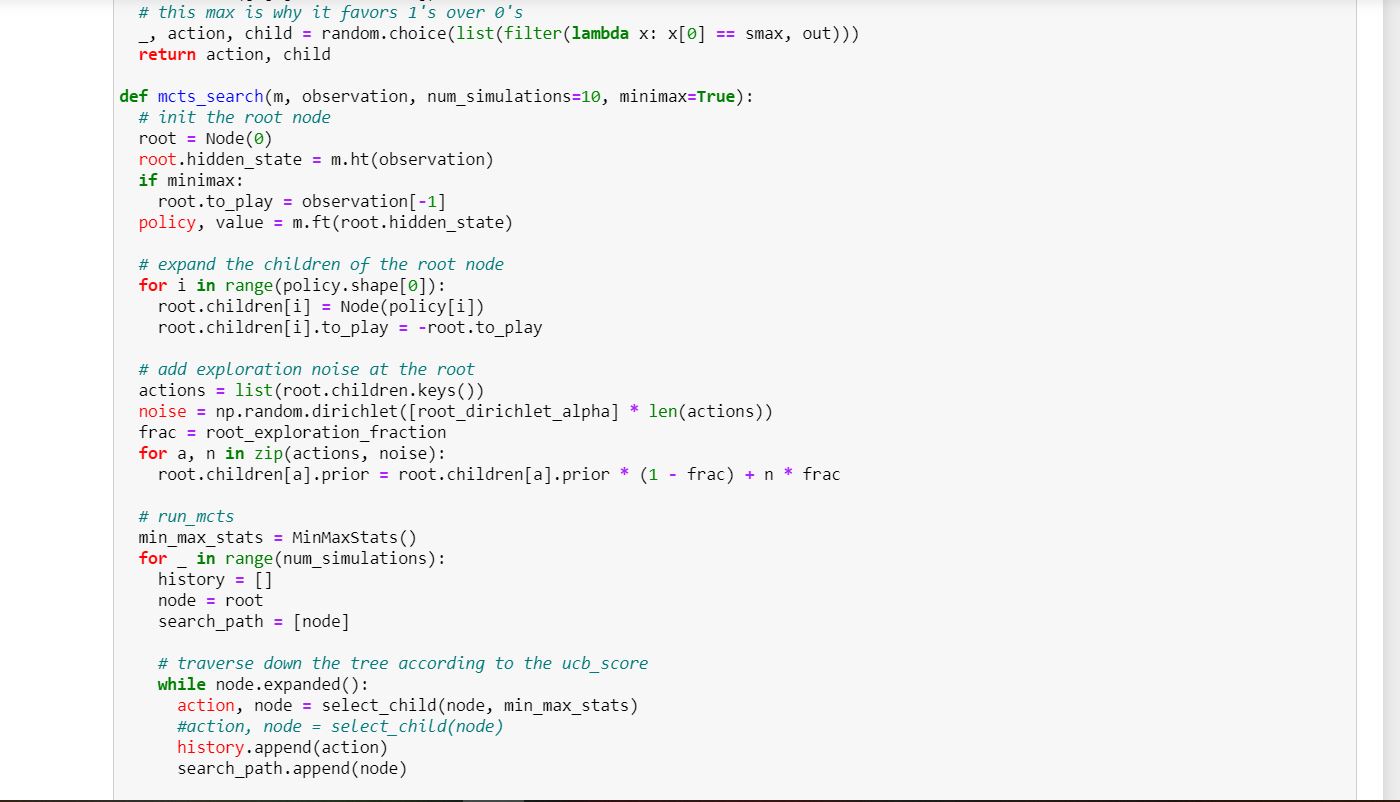


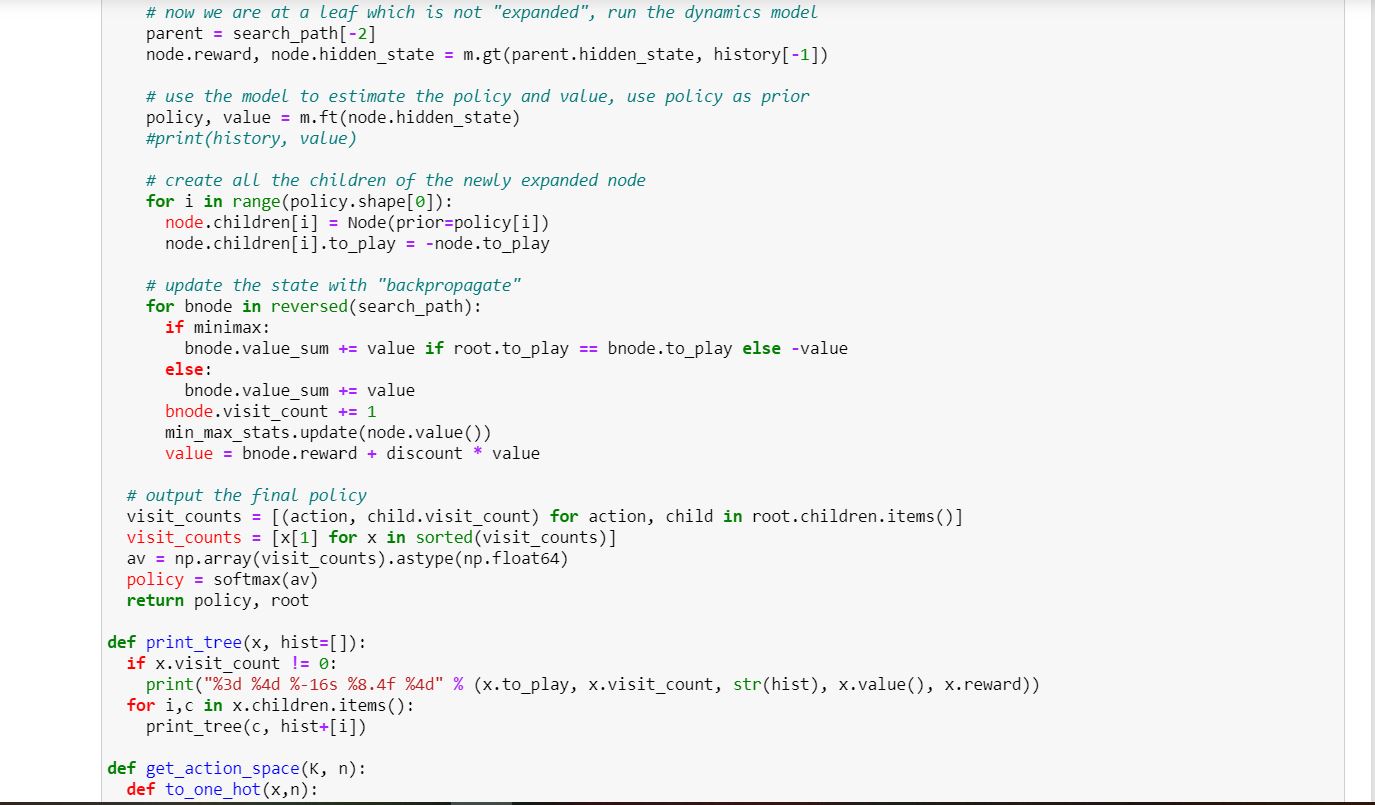


Then, the MinMaxStats() class is made to store the minimum and maximum information of the model. The model compares and selects various nodes of any resultant decision tree with the help of ucb\_score() and select\_node().

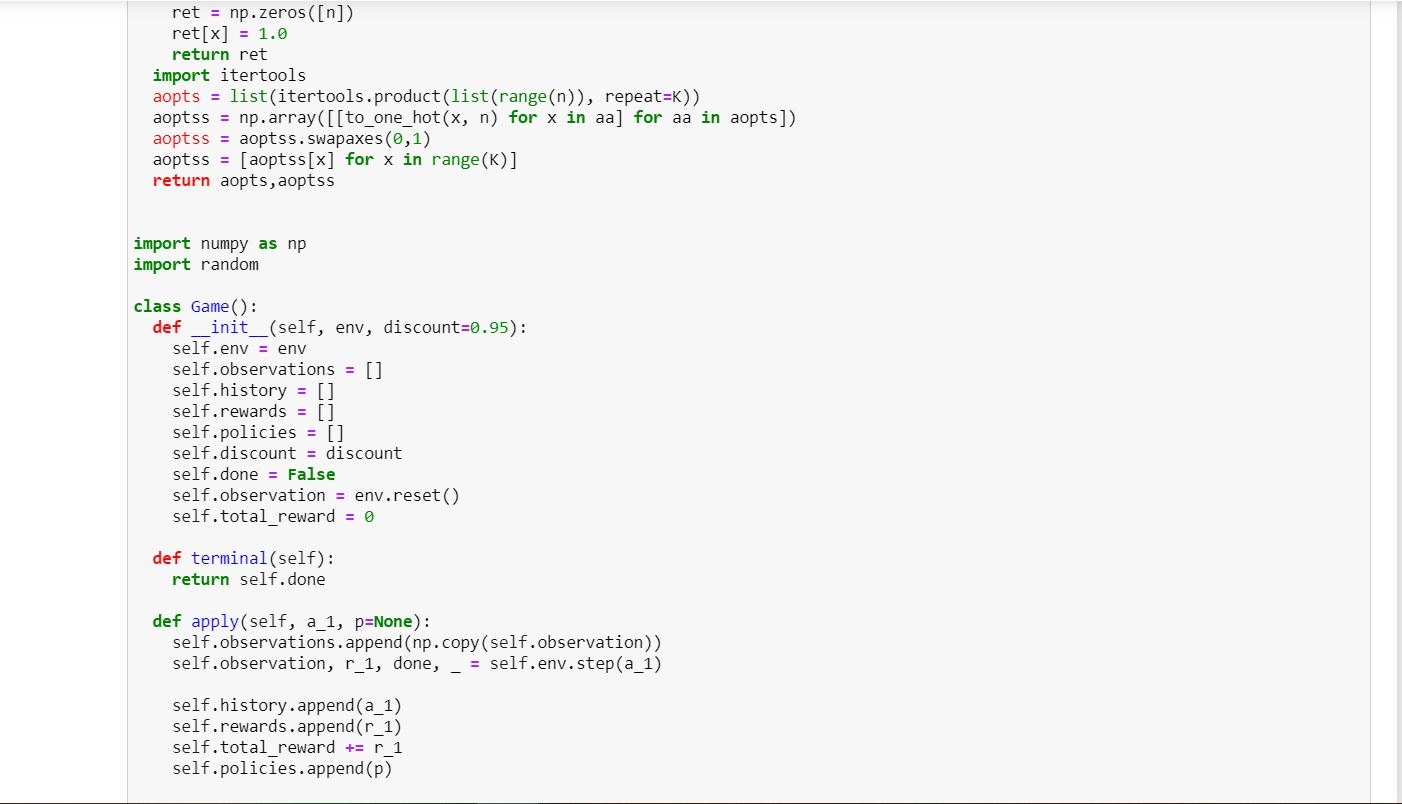


The mcts\_search() provides the actual MCTS search code in the algorithm. It is followed by the function to print the decision tree (print\_trees) and get action space of the environment (get\_action\_space).[11]





Then the class called Game is defined that will do the job of setting up the environment for the AI to move around and learn in [12].

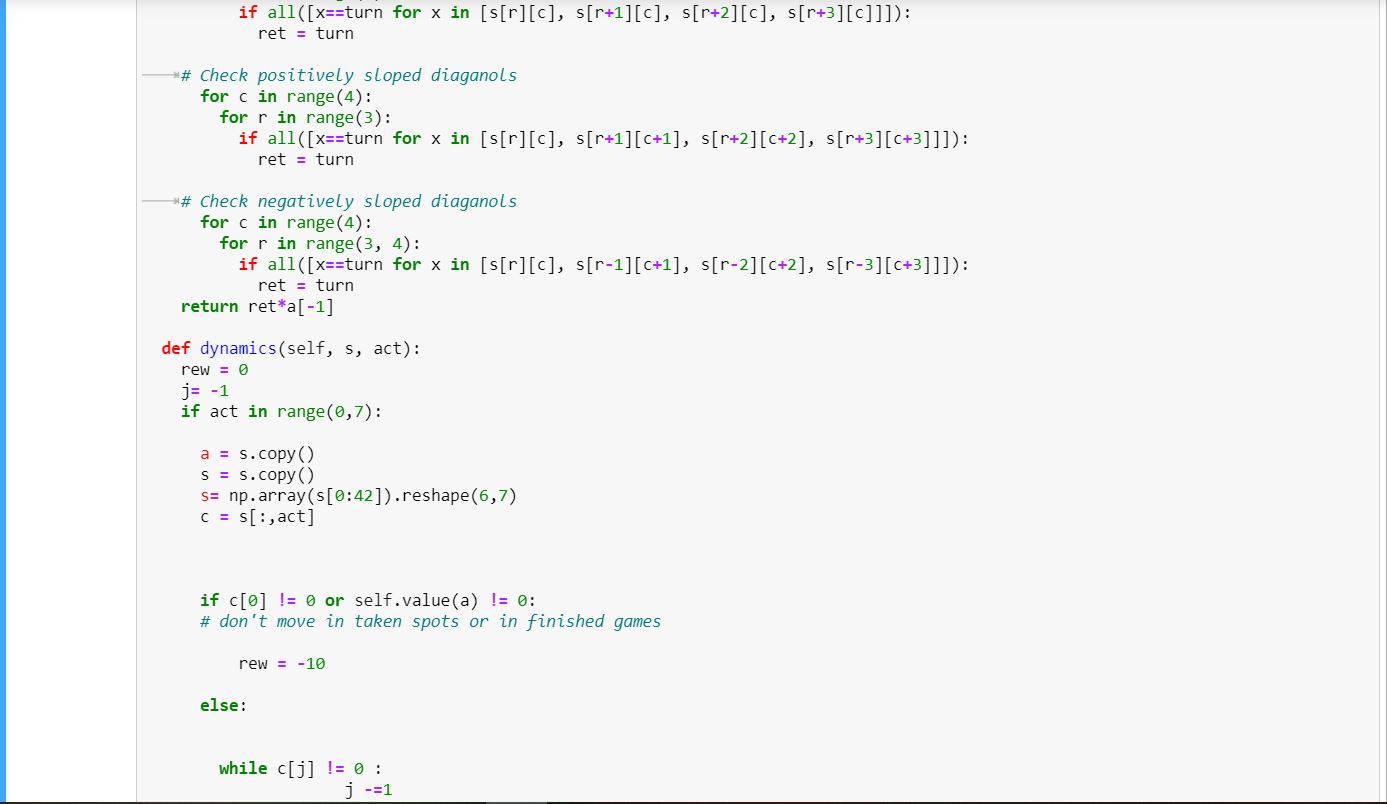




After that, the next class that is defined is ReplayBuffer(). The games that it had self-played and all the data associated with it, is stored with the help of this class [13].



The game is made by converting an array of 44 elements into a 6x7 matrix of 42 elements and then performing actions on it. At the start, the board is an identity matrix but it starts getting filled up by the red turn which is signified as 1 and yellow are -1. Eventually, it starts becoming a matrix filled up with 1s and -1s as the zeroes starts to decrease. In the original array of 44 elements, the last element signifies the turn. If it is 1, it is red’s turn and if it is -1, it is yellow’s turn. This is the code for that game contained in the class called ConnectFour().





Then, a function is defined named play\_game() to make MuZero make moves in the environment.

After which training is done for 4 epochs as the j variable signified epoch. At the end of each epoc the length of games played, the reward won, game’s history, and losses in each epoch.



**5.**

**Results**

The result of our project was that we were able to recreate the original MuZero albeit for a different game. We developed it for Connect four and saw superhuman level calculations and performance from the AI. [5]

The loss percentage was around 30% for the last few iterations. And it suggests fairly good moves when tested for the game positions, when we tested it.

This was the graph obtained after the second iteration of training of the model, here, the x axis signifies time and the y-axis signifies the depth of the decision tree the model travelled for that particular moment. [13,16]

This was the output for a random position on the board, for which the engine gives the probability of win for all the possible moves and the rewards associated with it.

**6.**

**Discussion**

**6.1 Interpretation in terms of empirical evidence**

High-performance planning [8] or model-free reinforcement learning methods [5], are the concepts on which many of the innovations in AI have been made. The benefits of both approaches have been combined in the method used in this paper. Our algorithm, MuZero, has matched the extraordinary performance of high-performance planning algorithms has been matched by MuZero in their favoured domains (logically complex board games such as Connect Four). While also outperforming state-of-the-art model-free RL algorithms in their favoured domains (visually complex Atari games) in the original paper.

Most importantly, our method [10] does not require any no knowledge of the game rules or environment dynamics is required for our method to work. This will potentially pave the way towards the application of powerful learning and planning methods to a plethora of real-world domains for which no perfect simulator actually exists [8, 13].

**6.2 Discussion about Limitations, future research and conclusions**

The Google DeepMind team is making a lot of strides in the world of AI. They are working on a lot of new and exciting stuff, involving new iterations of the MuZero engine. Muzero’s ability to make decisions in a new environment by making its own model, was inspired by the DeepMind’s need to make a model that can make decisions in a real-environment, in real time and they are taking big strides in that direction. Recently, they used the engine to commandeer a small remote-controlled helicopter and it eventually learned to fly it [15].

The main aim of the project seems to be to produce the perfect simulator for domains and tasks for which a perfect simulator doesn’t exist yet. Therefore, it is bound to make our handling of the tasks in life more efficient [16].

So, following the tradition of simulator, another product in the DeepMind repertoire is a simulator known as AlphaFold. Alpha Fold is used for protein shape prediction, which uses a similar architecture to MuZero. Therefore, they are getting close to making AIs which will hugely influence the general world in the time to come [2].

It can be eventually used to simulate the motion of planetary bodies according to the environmental constraints that are present [9].

It may be the first step of humanity towards a perfect simulator for our world. Although this vision seems a good distance away, the people of the world before telephone was invented didn’t believe that it was possible to talk cross-Atlantic in real time but it happened. Therefore, we would have the perfect simulator of the universe.

The limitations to MuZero [3,5]and such engines are that it works well for games and activities where reward and the state of winning is properly defined. However, it can’t still be developed where there are a lot of variables and chaos in the system. Therefore, it has not still been developed to, say, make art as there is no exact answer when art is involved since it is usually subjective.

In the future, it may even be used for that.

However, MuZero [6,8] may be used anywhere, there are fixed rules. Therefore, it can be used in various fields like space travel, physics, facial recognition etc.

**7.**

**References**

1.            Campbell, M., A.J. Hoane Jr, and F.-h. Hsu, *Deep blue.* Artificial intelligence, 2002. **134**(1-2): p. 57-83.

2.            Silver, D., et al., *Mastering the game of Go with deep neural networks and tree search.* nature, 2016. **529**(7587): p. 484-489.

3.            Brown, N. and T. Sandholm, *Superhuman AI for heads-up no-limit poker: Libratus beats top professionals.* Science, 2018. **359**(6374): p. 418-424.

4.            Moravčík, M., et al., *Deepstack: Expert-level artificial intelligence in heads-up no-limit poker.* Science, 2017. **356**(6337): p. 508-513.

5.            Anthony, M. and P.L. Bartlett, *Neural network learning: Theoretical foundations*. 2009: cambridge university press.

6.            Acher, M. and F. Esnault, *Large-scale analysis of chess games with chess engines: A preliminary report.* arXiv preprint arXiv:1607.04186, 2016.

7.            Romstad, T., et al., *Stockfish: A strong open source chess engine*. 2017.

8.            Silver, D., et al., *A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play.* Science, 2018. **362**(6419): p. 1140-1144.

9.            Schrittwieser, J., et al., *Mastering atari, go, chess and shogi by planning with a learned model.* arXiv preprint arXiv:1911.08265, 2019.

10.         Sutton, R.S. and A.G. Barto, *Reinforcement learning: An introduction*. 2018: MIT press.

11.         Hafner, D., et al. *Learning latent dynamics for planning from pixels*. in *International Conference on Machine Learning*. 2019.

12.         Kaiser, L., et al., *Model-based reinforcement learning for atari.* arXiv preprint arXiv:1903.00374, 2019.

13.         Deisenroth, M. and C.E. Rasmussen. *PILCO: A model-based and data-efficient approach to policy search*. in *Proceedings of the 28th International Conference on machine learning (ICML-11)*. 2011.

14.         Heess, N., et al. *Learning continuous control policies by stochastic value gradients*. in *Advances in Neural Information Processing Systems*. 2015.

15.         Levine, S. and P. Abbeel. *Learning neural network policies with guided policy search under unknown dynamics*. in *Advances in Neural Information Processing Systems*. 2014.

16.         Puterman, M.L., *Markov decision processes: discrete stochastic dynamic programming*. 2014: John Wiley & Sons.

17.         Sutton, R.S., D. Precup, and S. Singh, *Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning.* Artificial intelligence, 1999. **112**(1-2): p. 181-211.

18.         Coulom, R. *Efficient selectivity and backup operators in Monte-Carlo tree search*. in *International conference on computers and games*. 2006. Springer.

19. Hecht-Nielsen, R., 1992. Theory of the backpropagation neural network. In *Neural networks for perception* (pp. 65-93). Academic Press.