

INTRODUCTION

1.1 MACHINE LEARNING

Machine learning is the subfield of computer science that, according to Arthur Samuel in 1959, gives "computers the ability to learn without being explicitly programmed." Evolved from the study of pattern recognition and computational learning theory in artificial intelligence machine learning explores the study and construction of algorithms that can learn from and make predictions on data – such algorithms overcome following strictly static program instructions by making data-driven predictions or decisions, through building a model from sample inputs.

Machine learning is employed in a range of computing tasks where designing and programming explicit algorithms with good performance is difficult or infeasible; example applications include email filtering, detection of network intruders or malicious insiders working towards a data breach computer vision.

Machine learning is closely related to (and often overlaps with) computational statistics, which also focuses on prediction-making through the use of computers. It has strong ties to mathematical optimization, which delivers methods, theory and application domains to the field. Machine learning is sometimes conflated with data mining, where the latter subfield focuses more on exploratory data analysis and is known as unsupervised learning. Machine learning can also be unsupervised and be used to learn and establish baseline behavioral profiles for various entities and then used to find meaningful anomalies.

1.2 PROBLEM STATEMENT

To create bots that mimic human decision making in a game by using machine learning. Reading data from a game — Dota 2, which has heroes with multiple spells. Depending on the scenario, the player chooses to move the hero away, towards the enemy hero, or decides to use spells. We parse this data using protocol buffers and send it to our machine learning algorithm as actions. After we have a large set of test cases. We create a game, read data from the game, send it to the machine learning algorithm as parameters, which then selects the an appropriate action and sends it to the bot, which executes said action in-game. This effectively teaches a machine to make real-time decisions which mimic the player actions based on the scenario.

1.3 DOTA2 (Defense of the Ancients)

Dota 2 is a free-to-play multiplayer online battle arena (MOBA) video game developed and published by Valve corporation. The game is the stand-alone sequel to Defense of the Ancients (DotA). Dota 2 is played in matches between two teams that consist of five players, with both teams occupying their own separate base on the map. Each of the ten players independently control a powerful character, known as a "hero", that each feature unique abilities and different styles of play. During a match, a player and their team collect experience points and items for their heroes in order to fight through the opposing team's heroes and other defenses. A team wins by being the first to destroy a large structure located in the opposing team's base, called the "Ancient".



Fig 1.1 Description

The two teams—known as the Radiant and Dire—occupy fortified bases in opposite corners of the map, which is divided in half by a crossable river and connected by three paths, where are referred to as "lanes". The lanes are guarded by defensive towers that attack any opposing unit who gets within its line of sight.

A small group of weak computer-controlled creatures called "creeps" travel predefined paths along the lanes and attempt to attack any opposing heroes, creeps, and buildings in their way.

Creeps periodically spawn throughout the game in groups from two buildings, called the "barracks" that exist in each lane and are located within the team's

bases. The map is also permanently covered for both teams in fog of war, which prevents a team from seeing the opposing team's heroes and creeps if they are not directly in sight of themselves or an allied unit. The map also features a day-night cycle, with some hero abilities and other game mechanics being altered depending on the time of the cycle.



Fig 1.2 Mini-Map

A mini-map is a miniature map that is often placed at a screen corner in video games to aid players in orienting themselves within the game world. They are often only a small portion of the screen and thus must be selective in what details they display. Elements usually included on Mini-maps vary by video game genre. However, commonly included features are the position of the player character, allied units or structures, enemies, objectives and surrounding terrain. Mini-maps have become very common in real-time strategy and MMORPG video games because they serve as an indication of where the current screen lies within the scope of the game world. . Most first person shooter games also have some version or variant of the mini map, often showing enemies in real time.

1.4 PROJECT PURPOSE

Golem (An Automation or Robot) is a machine learning implementation that uses multiple algorithms to read-in game entities and and interprets actions. Use machine learning to learn your play-style and decisions. The game data is saved in a Google's Protocol Buffers format, which is decoded to read the game data and the players decisions and movements in-game.

A large number of test-cases are obtained and is passed into the appropriate machine learning algorithms — which learn how the player makes decisions based on the current in-game scenario. After n leaning data are parsed by machine learning algorithms, a bot is created which constantly sends the current in-game scenario to the machine learning system as parameters, which then replies with the appropriate action taken by the player.

1.5 MODULES DESCRIPTION

1.5.1 PROTOCOL BUFFER PARSING

- Protocol Buffers is a method of serializing structured data.
- Player data which needs to read as test cases are stored in the protocol buffer format as game entity data.
- This data is parsed using a protobuf parser, which we have written in Go-lang.
- This data is then moved to SQL tables so that it can be easily accessed.

1.5.2 ORGANIZING DATA

- The data obtained from parsing the protocol buffer files needs to be stored in a structured format.
- We have chosen SQL as the structured format, as we need fast access to this data to send the commands to the bot every 33ms.
- The data is read by the machine learning algorithms from SQL, this data is the test cases it uses to decide its output.

1.5.3 IMPLEMENTING MACHINE LEARNING ALGORITHM

Some of the machine learning algorithms such are:

1. Neural Networks - These compute multiple decisions that need to be every tick and gives out the list of commands that need to be executed in that tick.

2. Markov Decision Trees - Probability to move to every step is calculated as a function of action and a Markov Decision table is calculated.

3. Markov Reward Function - Probability of every action is computed and a reward function is applied to it, to calculate how favorable the step is.

A Markov decision process (known as an MDP) is a discrete-time state transition system. It can be described formally with 4 components.

- States
- Action
- Transition Probability
- Real-Valued reward

1.5.4 READING IN-GAME ENTITIES AND CREATING BOTS

- Lua is used to read the game data in which the bot is currently playing and give the bot commands to execute.
- These commands are decided by the machine learning algorithms and sent to the bot by the Lua network platform.
- Network platform is integrated into the game using sockets and uses HTTP requests to fetch and provide data to the bot.

Lightbringer: We are using lightbringer to create bots. It creates the lua script to handle bots and queries the server for commands on what the bot should do.

Writing bots for video games is a discipline with growing popularity both in academia and among gaming/coding enthusiasts. The game offers a LUA sandbox for scripting and modding various aspects of the game, meant for people who would like to create new content such as game modes, heroes, abilities and whatnot.

The Dota2 mode is a copy of the basic game mode and the default map, but acts as a proxy between the game and a web service through JSON objects. Though this complicates things a bit, it is unfortunately the only way, as the game only allows HTTP requests to access other processes, as the io package and loadlib are not supported by built-in LUA runtime. The repository also provides an example implementation of a bot, but rather for demonstration purposes.

LITERATURE SURVEY

2.1 ALPHAGO

What is GO ?

The game of Go originated in China 3,000 years ago. The rules of the game are simple: players take turns to place black or white stones on a board, trying to capture the opponent's stones or surround empty space to make points of territory. As simple as the rules are, Go is a game of profound complexity. There are an astonishing 10 to the power of 170 possible board configurations - more than the number of atoms in the known universe - making Go a googol times more complex than Chess

AlphaGo is a narrow AI computer program developed by Alphabet Inc.'s Google DeepMind in London to play the board game Go. In October 2015, it became the first Computer Go program to beat a human professional Go player without handicaps on a full-sized 19×19 board. In March 2016, it beat Lee Sedol in a five-game match, the first time a computer Go program has beaten a 9-dan professional without handicaps.

AlphaGo's algorithm uses a Monte Carlo tree search to find its moves based on knowledge previously "learned" by machine learning, specifically by an artificial neural network (a deep learning method) by extensive training, both from human and computer play

As of 2016, AlphaGo's algorithm uses a combination of machine learning and tree search techniques, combined with extensive training, both from human and computer play. It uses Monte Carlo tree search, guided by a "value network" and a "policy network," both implemented using deep neural network technology. A limited amount of game-

specific feature detection pre-processing (for example, to highlight whether a move matches a nakade pattern) is applied to the input before it is sent to the neural networks.

The system's neural networks were initially bootstrapped from human gameplay expertise. AlphaGo was initially trained to mimic human play by attempting to match the moves of expert players from recorded historical games, using a database of around 30 million moves. Once it had reached a certain degree of proficiency, it was trained further by being set to play large numbers of games against other instances of itself, using reinforcement learning to improve its play. To avoid "disrespectfully" wasting its opponent's time, the program is specifically programmed to resign if its assessment of win probability falls beneath a certain threshold; for the March 2016 match against Lee, the resignation threshold was set to 20%.

Versions:

- AlphaGo Master
- AlphaGo Lee
- AlphaGo Fan

Similar Systems: Facebook has also been working on their own Go-playing system darkforest, also based on combining machine learning and Monte Carlo tree search.

2.2 EXISTING SYSTEM

- Currently the bots are coded manually for every scenario of the game.
- Due to the high complexity of the game it is cumbersome to code bots FOR every scenario in the game.
- As Dota is a team oriented game, it is further a challenging task to code bots that interact with each other to achieve specific goals.

2.3 PROPOSED SYSTEM

Golem uses machine learning to learn your play-style and decisions. Machine learning needs training data to create a good map of which action to take based on the scenario. After n learning data are parsed by machine learning algorithms, a bot is created which constantly sends the current in-game scenario to the machine learning system as parameters, which then replies with the appropriate action taken by the player.

2.3.1 ADVANTAGES

- Anti-cheat - Machine learning can understand how a human plays a game and can detect when a non-human entity is used to cheat.
- Bot Automation in other fields - Machine learning bot creation implementation can be used to create bots for other fields with similar requirements.
- Understanding human decision making and patterns - Can be used to further many fields in human-machine interaction and create more human-like bots.

2.4 SOFTWARE DESCRIPTION

2.4.1 Python

Python is a widely used high-level programming language for general-purpose programming, created by Guido van Rossum and first released in 1991. An interpreted language, Python has a design philosophy which emphasizes code readability (notably using whitespace indentation to delimit code blocks rather than curly brackets or keywords), and a syntax which allows programmers to express concepts in fewer lines of

code than possible in languages such as C++ or Java. The language provides constructs intended to enable writing clear programs on both a small and large scale.

Python uses dynamic typing and a mix of reference counting and a cycle-detecting garbage collector for memory management. An important feature of Python is dynamic name resolution (late binding), which binds method and variable names during program execution.

It is intended to be a highly readable language. It is designed to have an uncluttered visual layout, often using English keywords where other languages use punctuation.

It doesn't have semicolons and curly brackets "{}" which is different compared to most of the programming language. Further, Python has fewer syntactic exceptions and special cases than C or Pascal.

Python uses whitespace indentation to delimit blocks – rather than curly braces or keywords. An increase in indentation comes after certain statements; a decrease in indentation signifies the end of the current block. This feature is also sometimes termed the off-side rule.

Statement and Control flow:

- The assignment statement (token '=', the equals sign). This operates differently than in traditional imperative programming languages, and this fundamental mechanism (including the nature of Python's version of variables) illuminates many other features of the language.

- The if statement, which conditionally executes a block of code, along with else and elif (a contraction of else-if).
- The for statement, which iterates over an iterable object, capturing each element to a local variable for use by the attached block.
- The while statement, which executes a block of code as long as its condition is true.
- The try statement, which allows exceptions raised in its attached code block to be caught and handled by except clauses; it also ensures that clean-up code in a finally block will always be run regardless of how the block exits.
- The class statement, which executes a block of code and attaches its local namespace to a class, for use in object-oriented programming.
- The def statement, which defines a function or method.
- The with statement, which encloses a code block within a context manager (for example, acquiring a lock before the block of code is run and releasing the lock afterwards, or opening a file and then closing it).
- The assert statement, used during debugging to check for conditions that ought to apply.
- The yield statement, which returns a value from a generator function. From Python 2.5, yield is also an operator.
- The import statement, which is used to import modules whose functions or variables can be used in the current program.
- The print statement was changed to the print() function in Python 3.

2.4.2 Google's Go(GoLang)

Go (often referred to as golang) is a free and open source programming language created at Google in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson. It is a compiled, statically typed language in the tradition of Algol and C, with garbage collection, limited structural typing, memory safety features and CSP-style concurrent programming features added.

Language design:

Go is recognizably in the tradition of C, but makes many changes to improve brevity, simplicity, and safety. Go consists of,

- A syntax and environment adopting patterns more common in dynamic languages:
 - Optional concise variable declaration and initialization through type inference (`x := 0` not `int x = 0`; or `var x = 0`).
 - Fast compilation times.
 - Remote package management (goget) and online package documentation.
- Distinctive approaches to particular problems:
 - Built-in concurrency primitives: light-weight processes (goroutines), channels, and the select statement.
 - An interface system in place of virtual inheritance, and type embedding instead of non-virtual inheritance.
 - A toolchain that, by default, produces statically linked native binaries without external dependencies.

- A desire to keep the language specification simple enough to hold in a programmer's head, in part by omitting features common to similar languages.

Language tools:

Go includes the same sort of debugging, testing, and code-vetting tools as many language distributions. The Go distribution includes, among other tools,

- go build, which builds Go binaries using only information in the source files themselves, no separate makefiles
- go test, for unit testing and microbenchmarks
- go fmt, for formatting code
- go get, for retrieving and installing remote packages
- go vet, a static analyzer looking for potential errors in code
- go run, a shortcut for building and executing code
- godoc, for displaying documentation or serving it via HTTP
- gorename, for renaming variables, functions, and so on in a type-safe way
- go generate, a standard way to invoke code generators

Example:

```
package main

import "fmt"

func main() {

    fmt.Println("Hello, World")

}
```

2.4.3 Google's Protocol Buffer

Protocol Buffers is a method of serializing structured data. It is useful in developing programs to communicate with each other over a wire or for storing data.

The method involves an interface description language that describes the structure of some data and a program that generates source code from that description for generating or parsing a stream of bytes that represents the structured data.

Developer(s)	Google
Initial release	Early 2001 (internal) ^[1] July 7, 2008 (public)
Stable release	3.2.0 / January 28, 2017
Development status	Active
Operating system	Any
Platform	Cross-platform
Type	serialization format and library, IDL compiler
License	BSD
Website	developers.google.com/protocol-buffers/ 

Protocol Buffers is widely used at Google for storing and interchanging all kinds of structured information. The method serves as a basis for a custom remote procedure call (RPC) system that is used for nearly all inter-machine communication at Google.

Though the primary purpose of Protocol Buffers is to facilitate network communication, its simplicity and speed make Protocol Buffers an alternative to data-centric C++ classes and structs, especially where interoperability with other languages or systems might be needed in the future.

2.4.4 SQL (Structured Query Language)

Structured Query Language is a domain-specific language used in programming and designed for managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS)

Paradigm	Multi-paradigm: declarative
Family	Programming language
Designed by	Donald D. Chamberlin Raymond F. Boyce
Developer	ISO/IEC
First appeared	1974; 43 years ago
Typing discipline	Static, strong
OS	Cross-platform

QL deviates in several ways from its theoretical foundation, the relational model and its tuple calculus. In that model, a table is a set of tuples, while in SQL, tables and query results are lists of rows: the same row may occur multiple times, and the order of rows can be employed in queries (e.g. in the LIMIT clause).

Language elements:

The SQL language is subdivided into several language elements, including:

- Clauses, which are constituent components of statements and queries. (In some cases, these are optional.)[17]
- Expressions, which can produce either scalar values, or tables consisting of columns and rows of data

- Predicates, which specify conditions that can be evaluated to SQL three-valued logic (3VL) (true/false/unknown) or Boolean truth values and are used to limit the effects of statements and queries, or to change program flow.
- Queries, which retrieve the data based on specific criteria. This is an important element of SQL.
- Statements, which may have a persistent effect on schemata and data, or may control transactions, program flow, connections, sessions, or diagnostics.

SQL statements also include the semicolon (";") statement terminator.

Though not required on every platform, it is defined as a standard part of the SQL grammar.

Insignificant whitespace is generally ignored in SQL statements and queries, making it easier to format SQL code for readability.

Operators:

Operator	Description
=	Equal to
<>	Not equal to (many DBMSs accept != in addition to <>)
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Match a character pattern
IN	Equal to one of multiple possible values
IS or IS NOT	Compare to null (missing data)
IS [NOT] TRUE or IS [NOT] FALSE	Boolean truth value test
IS NOT DISTINCT FROM	Is equal to value or both are nulls (missing data)
AS	Used to change a column name when viewing results

Queries:

The most common operation in SQL, the query, makes use of the declarative SELECT statement. SELECT retrieves data from one or more tables, or expressions. Standard SELECT statements have no persistent effects on the database. Some non-standard implementations of SELECT can have persistent effects, such as the SELECT INTO syntax provided in some databases.

SELECT is the most complex statement in SQL, with optional keywords and clauses that include:

- The FROM clause, which indicates the table(s) to retrieve data from. The FROM clause can include optional JOIN subclauses to specify the rules for joining tables.
- The WHERE clause includes a comparison predicate, which restricts the rows returned by the query. The WHERE clause eliminates all rows from the result set where the comparison predicate does not evaluate to True.
- The GROUP BY clause projects rows having common values into a smaller set of rows. GROUP BY is often used in conjunction with SQL aggregation functions or to eliminate duplicate rows from a result set. The WHERE clause is applied before the GROUP BY clause.
- The HAVING clause includes a predicate used to filter rows resulting from the GROUP BY clause. Because it acts on the results of the GROUP BY clause, aggregation functions can be used in the HAVING clause predicate.

- The ORDER BY clause identifies which column[s] to use to sort the resulting data, and in which direction to sort them (ascending or descending). Without an ORDER BY clause, the order of rows returned by an SQL query is undefined.
- The DISTINCT keyword eliminates duplicate data.

2.4.5 Markov Decision Processes

A Markov decision process (known as an MDP) is a discrete-time state transition system. It can be described formally with 4 components.

- States (S)

First, it has a set of states. These states will play the role of outcomes in the decision theoretic approach we saw last time, as well as providing whatever information is necessary for choosing actions. For a robot navigating through a building, the state might be the room it's in, or the x,y coordinates. In our case the state is the position, health, mana of our character.

- Action (A)

Secondly we have Actions, these are selected from a finite set of actions our character can perform in the environment such as movement to another x,y co-ordinate, casting a spell, attacking or any other interaction with another entity in the game

- Transition Probability

$$\Pr(S_{t+1}|S_t, A_t)$$

The transition probabilities describe the dynamics of the world. They play the role of the next-state function in a problem-solving search, except that every state is thought to be a possible consequence of taking an action in a state. So, we specify, for each state S_t and A_t action, the probability that the next state will be S_{t+1} . You can think

of this as being represented as a set of matrices, one for each action. Each matrix is square, indexed in both dimensions by states. If you sum over all states \mathcal{S}_i , then the sum of the probabilities that \mathcal{S}_i is the next state, given a particular previous state and action is 1.

This equation in turn creates the Markov Property, which states the following *“that given the current state and action, the next state is independent of all the previous states and actions. The current state captures all that is relevant about the world in order to predict what the next state will be”* and is given by the equation

$$\Pr(\mathcal{S}_{t+1} | \mathcal{S}_0 \dots \mathcal{S}_t, A_0 \dots A_t)$$

- Real – Valued Reward $R(s)$

Finally, there is a real-valued reward function on states. You can think of this as a short-term utility function. This in our implementation is the gold and experience the character earns during his adventures through the game which is calculated into a function called net-worth in co-relation to the enemy net-worth to give the character a sense of how it is performing in the current run.

Finding a Policy:

In an MDP, the assumption is that you could potentially go from any state to any other state in one step. And so, to be prepared, it is typical to compute a whole policy, rather than a simple plan. A policy is a mapping from states to actions. It says, no matter what state you happen to find yourself in, here is the action that it's best to take now. Because of the Markov property, we'll find that the choice of action only needs to depend on the current state (and possibly the current time), but not on any of the previous states.

$$\Pi : S \rightarrow A$$

How do you find a good policy?

In the simplest case, we'll try to find the policy that maximizes, for each state, the expected reward of executing that policy in the state. In our example for each state we perform actions which move us to another state of environment, each state we obtain the reward for the current state and try to pick a state which will maximize these rewards. This can be given by:

$$\text{Max}(R_t | \Pi, S_t)$$

Finite Horizon Optimal Policies

Using just the approach above makes us short sighted, it does not take in par the whole environment (game) but just the maximization of rewards from our current state. We don't want our character to stop playing if it does not gain immediate rewards instead we want to plot a plan to move around a map and gain these rewards. That means, that we should find a policy that, for every initial state S_0 , results in the maximal expected sum of rewards from times 0 to k, which can be given by :

$$E(\sum_{t=0}^k R_t | \Pi, S_t)$$

Non – Stationary Policies

Our character can't start the game and expect to kill another enemy character to gain rewards it needs to farm creeps, collect runes and perform other actions in the game that may give it lesser rewards but in turn sets it up to be able to gain the higher rewards it yearns. Thus we need to select our finite horizon optimal policies depending on the time of the game and the strength (level) of our character, this changes our optimality equation to:

$$E_t(\sum_{t=0}^k R_t | \Pi, S_t)$$

Infinite Horizons

We really don't have any idea how long a game may go on for or how many states need to be selected to finish a game. Thus it's popular to consider infinite horizon models of optimality. Thus we can change our optimality equation to:

$$E_t(\sum_{t=0}^{\infty} R_t | \Pi, S_t)$$

Discount Factor

The problem now arises that we have infinite states then we add up all these states to find an optimality equation then we have infinite number of rewards which mathematically seems improbable. Thus we add a discount factor not only to make the mathematics realistic but also to make sure our character gains rewards as fast as possible early on in the game (the rewards mean lesser as the game goes on). The discount factor gamma is a number between 0 and 1, which has to be strictly less than 1. Usually it's somewhere near 0.9 or 0.99. So, we want to maximize our sum of rewards, but rewards that happen tomorrow are only worth .9 of what they would be worth today. This changes our equation to:

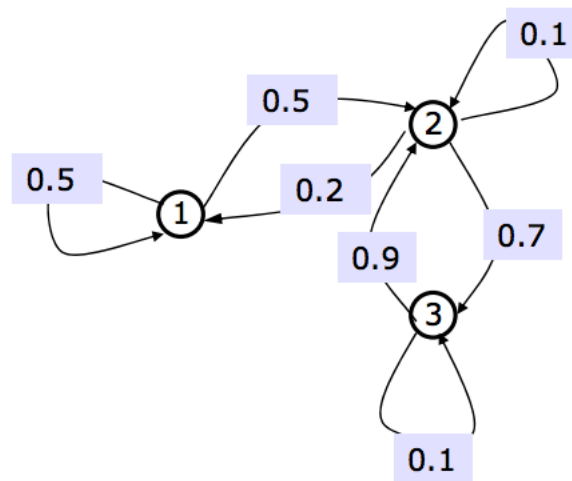
$$E_t(\sum_{t=0}^{\infty} \gamma^t R_t | \Pi, S_t)$$

where:
 $0 < \gamma < 1$ (discount Factor)

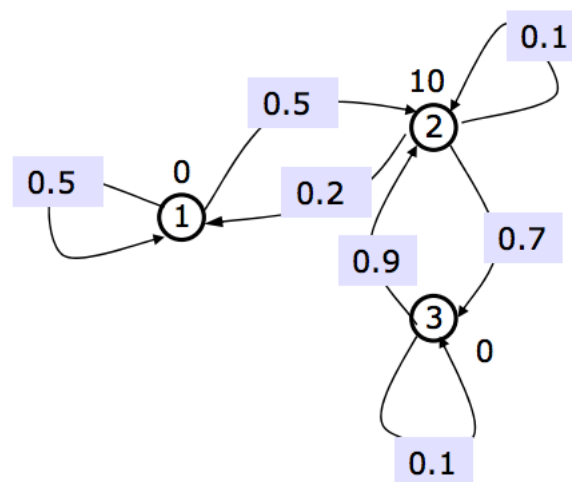
This solves a path from the time our character spawns in the game for the first time to the time it dies for the first time, on respawn our system will calculate the same optimal path it computed on the first spawn which may not be the right path to take as the

game may have progressed a long way since its death and multiple deaths and respawns are possible in this game. To solve this problem a Markov chain needs to be implemented

Markov Chains



Here we show a simple Markov chain where we have labeled our states and how we can move from one state to another. There is also a given probability of movement each state from the current state which sums up to 1.



Markov chains don't always give a reward while moving from one state to another. Here for our example we are saying that we get a reward of 0 for moving to states 1 and 3 and a reward of 10 for moving to state 2.

Calculating the future

Once we have the probability of movement from one state to another and the reward we can obtain at each state we can then calculate how rewarding each state is from the current state with the discount factor γ . This can be given by the equation:

$$V(s) = R(s) + \gamma()$$

Now, we consider what the future might be like. We'll compute the expected long-term value of the next state by summing over all possible next states, s' , the product of the probability of making a transition from s to s' and the infinite horizon expected discounted reward, or value of s' .

$$V(s) = R(s) + \gamma \sum_{s'} P(s'|s) V(s')$$

Since we know R and P (those are given in the specification of the Markov chain), we'd like to compute V . If n is the number of states in the domain, then we have a set of n equations in n unknowns (the values of each state).

Using our Markov chain example these can be calculated as follows:

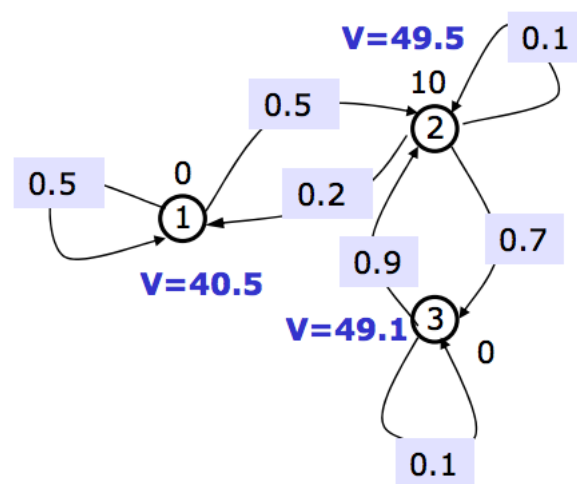
Assume $\gamma=0.9$

$$V(1) = 0 + .9(.5 V(1) + .5 V(2))$$

$$V(2) = 10 + .9(.2 V(1) + .1 V(2) + .7 V(3))$$

$$V(3) = 0 + .9(.9 V(2) + .1 V(3))$$

Now our Markov chain looks like this:



Now, if we solve for the values of the states, we get that $V(1)$ is 40.5, $V(2)$ is 49.5, and $V(3)$ is 49.1. This seems at least intuitively plausible. State 1 is worth the least, because it's kind of hard to get from there to state 2, where the reward is. State 2 is worth the most; it has a large reward and it usually goes to state 3, which usually comes right back again for another large reward. State 3 is close to state 2 in value, because it usually takes only one step to get from 3 back to 2.

2.4.5 Lua

Lua is a lightweight multi-paradigm programming language designed primarily for embedded systems and clients. Lua is cross-platform, since it is written in ANSI C, and has a relatively simple C API.

Lua was originally designed in 1993 as a language for extending software applications to meet the increasing demand for customization at the time. As Lua was intended to be a general embeddable extension language, the designers of Lua focused on improving its speed, portability, extensibility, and ease-of-use in development.



Lua programs are not interpreted directly from the textual Lua file, but are compiled into bytecode, which is then run on the Lua virtual machine. The compilation process is typically invisible to the user and is performed during run-time, but it can be done offline in order to increase loading performance or reduce the memory footprint of the host environment by leaving out the compiler. Lua bytecode can also be produced and executed from within Lua, using the `dump` function from the string library and the `load/loadstring/loadfile` functions. Lua version 5.3.3 is implemented in approximately 24,000 lines of C code.

Like most CPUs, and unlike most virtual machines (which are stack-based), the Lua VM is register-based, and therefore more closely resembles an actual hardware design. The register architecture both avoids excessive copying of values and reduces the total number of instructions per function. The virtual machine of Lua 5 is one of the first register-based pure VMs to have a wide use.

REQUIREMENT ANALYSIS

3.1 FUNCTIONAL REQUIREMENTS

In software engineering, a functional requirement defines a function of a software system or its component. A function is described as a set of inputs, the behavior, and outputs. Functional requirements may be calculations, technical details, data manipulation and processing and other specific functionality that define what a system is supposed to accomplish. Behavioral requirements describing all the cases where the system uses the functional requirements are captured in use cases.

Here, the system has to perform the following tasks:

- Read the in-game entities and send to to the server.
- The server then takes the variables and passes them into the algorithms.
- Which then calculates the best possible move to take and returns the command.
- The server then commands the bot by sending the returned command to move to the next location.

3.2 NON-FUNCTIONAL REQUIREMENTS

In systems engineering and requirements engineering, a non-functional requirement is a requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviors. This should be contrasted with functional requirements that define specific behavior or functions. The plan for implementing

functional requirements is detailed in the system design. The plan for implementing non-functional requirements is detailed in the system architecture.

Other terms for non-functional requirements are "constraints", "quality attributes", "quality goals", "quality of service requirements" and "non-behavioral requirements".

Some of the quality attributes are as follows:

3.2.1 ACCESSIBILITY:

Accessibility is a general term used to describe the degree to which a product, device, service, or environment is accessible by as many people as possible.

In our project, anyone who has the IP address of the server can connect their game to the server and play. Setup is simple and effective.

3.2.2 MAINTAINABILITY:

In software engineering, maintainability is the ease with which a software product can be modified in order to:

- Correct defects
- Meet new requirements

New functionalities can be added in the project based on the user requirements just by adding the appropriate files to existing project using Python programming languages.

Since the programming is very simple, it is easier to find and correct the defects and to make the changes in the project.

3.2.3 SCALABILITY:

System is capable of handling increase total throughput under an increased load when resources (typically hardware) are added.

3.2.4 PORTABILITY:

Portability is one of the key concepts of high-level programming. Portability is the software code base feature to be able to reuse the existing code instead of creating new code when moving software from an environment to another.

Project can be executed under different operation conditions provided it meet its minimum configurations. Only system files and dependant assemblies would have to be configured in such case.

3.3 HARDWARE REQUIREMENTS

Processor	: Any Processor above 4 GHz
RAM	: 8 GB
Hard Disk	: 80 GB
Input device	: Standard Keyboard and Mouse
Output device	: VGA and High Resolution Monitor

3.4 SOFTWARE REQUIREMENTS

- Operating system : Windows 10
- Front End : Dota 2
- IDE : Any text editor
- Data Base : MySQL
- Database Connectivity : MySQL-Python

CHAPTER 4

DESIGN & IMPLEMENTATION

4.1 DESIGN GOALS

To parse the information needed to create test cases and from these test cases run our bot in the environment the following steps were taken to design the system

4.1.1 INPUT/OUTPUT

The inputs that were taken in this case were previously played games, which were played by humans these give us an understanding or a starting point for our bot to find a path in the game which from these parsed games manifest as the output to our system

4.1.2 EFFICIENCY

To increase the efficiency, we run the server on a different system from the game. Thus the game runs on a dedicated system which only needs to concentrate loading the graphs and computing the in-game entities while as another system which acts as the server is responsible for computing out bots actions and movements and sending it to the system running the game.

4.2 SYSTEM ARCHITECTURE

The system architecture can be broken into three parts namely:

4.2.1 PARSING INPUT DATA

The first part consists of the input data which is fed into the parser in the form of a .dem file, these files store all the information that take place in the game with a time sequence. The files are then parsed by the parser which converts them from their natural format (Protobuf) to an understandable format we can then save on a SQL table for fast

access and query as our system needs to feed information every 33ms. This parsing is done by reading in-game entities' and callbacks for more information about the DOTA2 demo file. Callbacks are format it can be found on their github page here – <https://github.com/dotabuff/manta/blob/master/callbacks.go>

```
//IMPLEMENT NEEDED CALLS

//gets all the combat log data
getCombatLog(p)

//gets the where the player clicked and other movements
getPlayerClickOrders(p)

//Parse metadata file
loadMatchMetadatFile(p)

//some basic entites
getEntities(p)

p.Start()

fmt.Print("-----\nParse Complete!\n")

//LOADING INTO THE SQL TABLE

loadCombatLog()

loadPlayerOrder()

loadMatchMetadata()

fmt.Print("\n---END---\n")
```

Fig 4.1 callbacks we are checking for in the .dem files

```
func loadPlayerOrder(){
    stmt, err := db.Prepare("INSERT clickOrders SET matchId=?,entIndex=?,orderType=?,targetIndex=?,abilityIndex=?,sequenceNumber=?,queue=?")
    checkErr(err)
    for _,data := range spectatorPlayerUnitOrderArray{
        res, err := stmt.Exec(matchId,data.entIndex,data.orderType,data.targetIndex,data.abilityIndex,data.sequenceNumber,data.queue)
        checkErr(err)
        id, err := res.LastInsertId()
        loadPositions(id,data.position)
        if len(data.units) > 0 {
            loadUnits(id,data.units)
        }
    }
}

func loadPositions(id int64, value vector){
    stmt,err := db.Prepare("INSERT Positions SET logid=?,X=?,Y=?,Z=?")
    checkErr(err)
    _,err = stmt.Exec(id,value.X,value.Y,value.Z)
    checkErr(err)
}

func loadUnits(id int64,value []int32){
    stmt,err := db.Prepare("INSERT Units SET logid=?,unit=?")
    checkErr(err)
    for _,data := range value{
        _,err = stmt.Exec(id,data)
        checkErr(err)
    }
}
```

Fig 4.2 loading parsed data into the sql table

4.2.2 MACHINE LEARNING SYSTEM

The machine learning algorithm is the main brain of our system. It applies our needed algorithms to the input data and computes the input data and saves it in the form of a matrix which has a cell for every state we can be in and each cell stores the action we can take from that cell making it efficient to find the actions we are looking for instead of looping though the whole system looking for actions in the state we are currently in. The creation and storage method for these matrices were done by using numpy arrays which is an in-build python module which is commonly used for data research and computation.


```
def brain() :
    print("thinking...")
    connection = pymysql.connect(host='localhost',
                                user='golem',
                                password='whateven',
                                db='thirst')

    cur = connection.cursor(pymysql.cursors.DictCursor)

    heroId = 44 #phantom assassin
    sql = "SELECT * FROM playerPositions WHERE hero = '%s'"
    cur.execute(sql % heroId)
    result=cur.fetchall()

    for row in result:
        matchVal = row['matchId']
        time = row['time']
        x = int(row['x']) - 50
        y = int(row['y']) - 50
        for selectrow in result :
            if selectrow['time'] == time + 1:
                if selectrow['matchId'] == matchVal:
                    value = np.array([selectrow['x'],selectrow['y']])
                    if ProbabilityMatrix[x][y] == 0:
                        Data = [value]
                        ProbabilityMatrix[x][y] = Data
                    else:
                        array = ProbabilityMatrix[x][y]
                        array.append(value)
                        ProbabilityMatrix[x][y] = array
    print("done Thinking")
```

Fig 4.3 computing all positions, we can move to from our current state

4.2.3 CONTROLLING THE IN-GAME BOT

This is the final step of our system this contacts the in-game environment and gives the current game status to the machine learning algorithms as input so that state information can be computed. It also helps us control our bot in terms of movement and actions. These actions and movements are fed to the system by the machine learning algorithm as they seem fit for the current scenario. This side of our system uses a pre-implemented system called Lightbringer. Lightbringer is a DOTA2 AI framework project

which is specifically created to perform machine learning and other academic research on the game. To understand more about the architecture of this system you can find it on their github page here - <https://github.com/lightbringer/dota2ai>

The server to control to fetch information from lightbringer was created by us and written in python using a module called flask to send and receive data using post requests. This network implementation was the fastest way to send and receive information and helped in making our system more efficient by splitting the tasks across systems.

```
from flask import Flask, redirect, url_for, request, json
from levelup import levelupPlayer
import sortentites
from brain import brain

#check = True

brain()
commandQueue = []

app = Flask(__name__)

@app.route('/Dota2AIService')
def hello():
    return "Dota2 AI Servicer."

@app.route('/Dota2AIService/reset', methods = ['POST'])
def reset():
    print "reset method"
    return ""

@app.route('/Dota2AIService/select', methods = ['POST'])
def select():
    print "selecting hero"
    return json.dumps({"hero": "npc_dota_hero_phantom_assassin", "command": "SELECT"})

@app.route('/Dota2AIService/chat', methods = ['POST'])
def chat():
    print "chat text %s" % request.form['text']
    return ""

@app.route('/Dota2AIService/levelup', methods = ['POST'])
def levelup():
    value = levelupPlayer()
    print "%s" % value
    return value

@app.route('/Dota2AIService/update', methods = ['POST'])
def update():
    #print "%s" % commandQueue
    returnVal = ""
    sortentites.sortentites(request.data, commandQueue)
    if len(commandQueue) > 0:
        returnVal = commandQueue[0]
        commandQueue.remove(returnVal)
    else:
        returnVal = json.dumps({ "command" : "NOOP" })
    return returnVal

if __name__ == '__main__':
    app.run(host='0.0.0.0')
```

Fig 4.4 server that is used to interact b/w the environment and ML algorithms

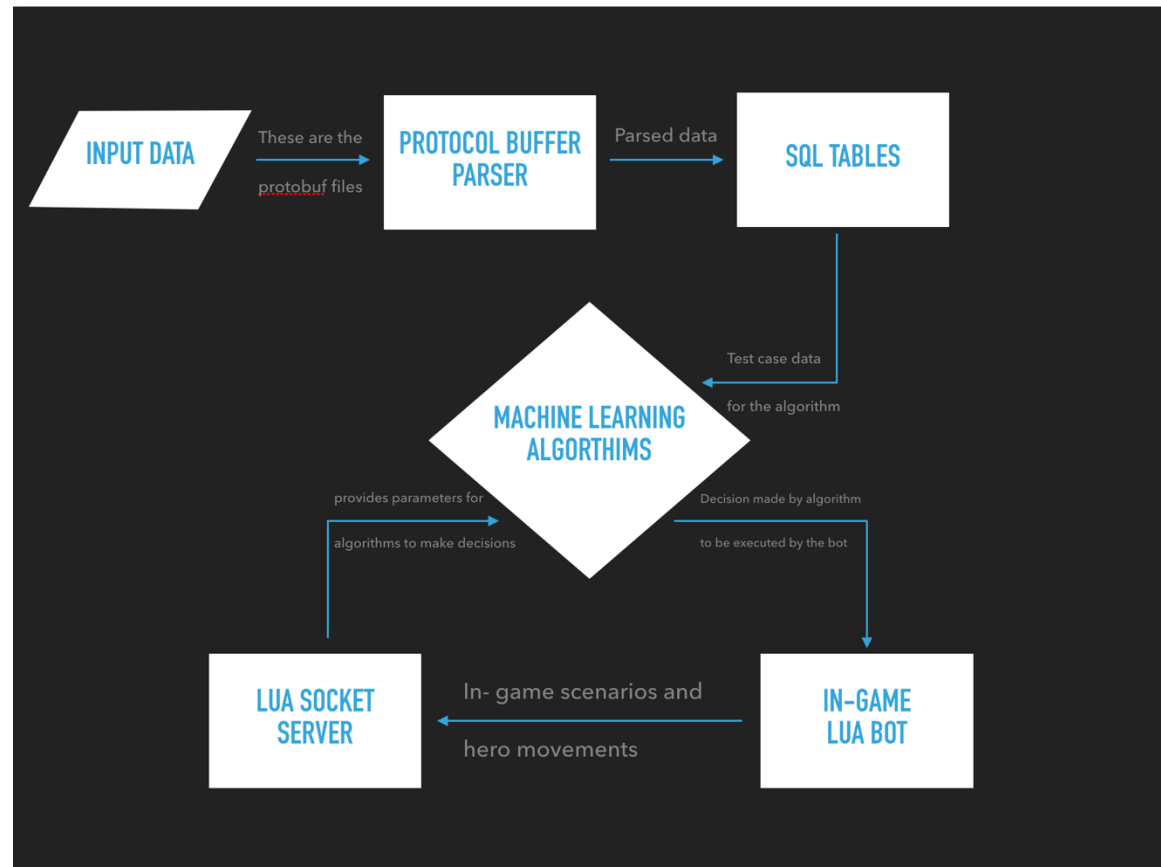


Fig 4.4 System Architecture

4.3 GAME MANUAL

The game of DOTA2 has many entities which one needs to understand about to understand what we are trying to achieve.

4.3.1 HERO (CHARECTER)

The character referred to as the “Hero” in the game. This is the main entity we can control in terms of movements, attacking and casting spells we can even buy items for this character and use these item to our advantage to have an edge over the game. The hero has attribute such as health and mana which helps us survive and gives us the ability

to cast spells. There are ten characters that play a game of DOTA five on our team and five enemy characters.



Fig 4.5 Hero (Phantom Assassin)



Fig 4.6 level, attribute, spells, health, mana and items of our hero

4.3.2 BUILDINGS

In the game there are two sides one is called the “Radiant” and other the “Dire” these are respectively the teams that are present in the game. Each team has a certain number of buildings across the map which need to be destroyed in order to win the game. The most common are towers there are eleven of these that are held by each team. The

towers are scattered across the map. Three towers are present in each lane and two guarding the ancient.

The “Ancient” is the most important building the game as the main purpose of the game is to defend these each team has one ancient and once destroyed the the opposing team wins the game. They can only be attacked if all the three towers in a lane are destroyed.



Fig 4.7 Radiant Tower



Fig 4.8 Dire Tower



Fig 4.9 Radiant and Dire ancients

4.3.3 CREEPS

Creeps and NPC (non – playable characters) in the game which spawn every 30 seconds. They do a certain amount of damage and have a fixed movement speed throughout the game. Each team has three types of creeps there are melee, range and siege. If a character is present when a creep dies it gains experience, if a character gets the last –hit which kills the creeps he also gains gold. Creeps in terms of the game progress mostly help each team gain gold and experience. They are also able to attack and bring down towers.



Fig 4.10 Different types of creeps in the game

4.3.3 ROSHAN

Roshan is a beast that is present in the game. This beast for most of the game sits in its lair. Each team has a choice to go kill Roshan which will then grant them the aegis of immortality. This item will respawn and hero who dies without the respawn timer and in the position in which they died. Yet killing Roshan is not an easy task as he does

significant damage to kill any hero and has a monumental health. Killing Roshan is a team activity that requires more than more character.



Fig 4.11 Roshan in his lair



Fig 4.12 Aegis of immortality dropped by Roshan on his death

SNAPSHOTS

TABLES	id	logid	X	Y	Z
abilityUpgrades	1	1	0	0	0
assistPlayers	2	2	0	0	0
autoStyleCriteria	3	3	0	0	0
bans	4	4	0	0	0
clickOrders	5	5	0	0	0
combatLog	6	6	0	0	0
graphGold	7	7	0	0	0
graphNW	8	8	0	0	0
graphXP	9	9	0	0	0
individualNW	10	10	0	0	0
inventorySnapshot	11	11	0	0	0
itemid	12	12	0	0	0
items	13	13	0	0	0
kills	14	14	0	0	0
levelUpTimes	15	15	0	0	0
matchData	16	16	0	0	0
metadata	17	17	0	0	0
packetEntities	18	18	0	0	0
picks	19	19	0	0	0
playerPositions	20	20	0	0	0
players	21	21	0	0	0
Positions	22	22	0	0	0
teamData	23	23	0	0	0
Units	24	24	0	0	0
	25	25	0	0	0
	26	26	0	0	0

Fig 6.1 SQL table for Positions

TABLES	id	dotaTeam	cmFirstPick	cmCaptainPlayerId	cmPenalty	logid
abilityUpgrades	1	2	0	0	0	1
assistPlayers	2	3	0	0	0	1
autoStyleCriteria	3	6	0	0	0	1
bans	4	7	0	0	0	1
clickOrders	5	8	0	0	0	1
combatLog	6	9	0	0	0	1
graphGold	7	10	0	0	0	1
graphNW	8	11	0	0	0	1
graphXP	9	12	0	0	0	1
individualNW	10	13	0	0	0	1
inventorySnapshot	11	2	0	0	0	2
itemid	12	3	0	0	0	2
items	13	6	0	0	0	2
kills	14	7	0	0	0	2
levelUpTimes	15	8	0	0	0	2
matchData	16	9	0	0	0	2
metadata	17	10	0	0	0	2
packetEntities	18	11	0	0	0	2
picks	19	12	0	0	0	2
playerPositions	20	13	0	0	0	2
players	21	2	0	0	0	3
Positions	22	3	0	0	0	3
teamData	23	6	0	0	0	3
Units	24	7	0	0	0	3
	25	8	0	0	0	3
	26	9	0	0	0	3

Fig 6.2 SQL table for TeamData

TABLES	id	accountid	playerSlot	avgKillsX16	avgDeathsX16	avgAssistsX16	avgGpmX16	avgXpmX16	bestKillsX16	bestAssistsX16	b
abilityUpgrades	1	363447150	0	9	11	11	376	366	20	21	
assistPlayers	2	101867413	1	9	7	8	478	592	31	23	
autoStyleCriteria	3	125063546	2	9	7	10	496	566	33	26	
bans	4	165573189	3	6	7	13	451	501	17	34	
clickOrders	5	183437837	4	9	6	18	402	484	24	44	
combatLog	6	426988815	128	16	16	9	566	600	16	9	
graphGold	7	146498748	129	8	5	7	606	620	29	21	
graphNW	8	238364535	130	7	9	14	375	400	22	28	
graphXP	9	327870208	131	6	16	25	304	538	6	25	
individualNW	10	205297235	132	8	11	17	520	605	8	17	
inventorySnapshot	11	283095604	0	6	5	15	448	445	18	33	
itemid	12	295205414	1	5	7	16	308	452	7	26	
items	13	140553245	2	8	6	16	375	429	25	33	
kills	14	199044936	3	14	8	11	552	610	42	29	
levelUpTimes	15	169935255	4	4	5	16	389	490	7	27	
matchData	16	334231359	128	2	8	11	255	316	3	25	
metadata	17	136530726	129	10	3	8	673	620	19	16	
packetEntities	18	141559905	130	5	7	16	445	488	11	37	
picks	19	260042956	131	12	7	12	500	548	31	31	
playerPositions	20	149009689	132	6	5	15	391	447	15	28	
players	21	141669737	0	8	6	9	508	573	30	28	
Positions	22	16951721	1	8	11	16	378	445	19	32	
teamData	23	169768621	2	11	6	11	525	584	27	28	
Units	24	52069403	3	6	6	14	422	444	23	32	
	25	240701963	4	16	7	11	631	680	28	22	
	26	30543697	128	1	9	12	226	266	3	19	
	27	68455704	129	5	8	14	384	321	24	44	
	28	124091947	130	6	7	15	342	385	13	32	
	29	58409956	131	9	6	9	526	535	21	28	
	30	890991	132	11	4	15	537	571	38	36	

Fig 6.3 SQL table for Players

TABLES	id	gameTime	kills	deaths	assists	level	logid
abilityUpgrades	1	-59	0	0	0	1	1
assistPlayers	2	-29	0	0	0	1	1
autoStyleCriteria	3	0	0	0	0	1	1
bans	4	30	0	0	0	1	1
clickOrders	5	60	0	0	0	1	1
combatLog	6	90	0	0	0	1	1
graphGold	7	120	0	1	0	2	1
graphNW	8	150	1	1	0	3	1
graphXP	9	180	1	1	0	3	1
individualNW	10	210	1	2	2	3	1
inventorySnapshot	11	240	1	2	2	3	1
itemid	12	270	1	2	4	3	1
items	13	300	1	2	4	3	1
kills	14	330	1	2	4	3	1
levelUpTimes	15	360	1	2	4	4	1
matchData	16	390	1	2	4	4	1
metadata	17	420	1	3	4	4	1
packetEntities	18	450	1	3	4	4	1
picks	19	480	1	3	5	4	1
playerPositions	20	510	1	4	5	4	1
players	21	540	1	4	5	4	1
Positions	22	570	1	5	6	4	1
teamData	23	600	1	5	6	4	1
Units	24	630	1	5	6	4	1
	25	660	1	5	6	4	1
	26	690	1	6	6	4	1
	27	720	1	6	8	5	1

Fig 6.4 SQL table for Inventory Snapshot

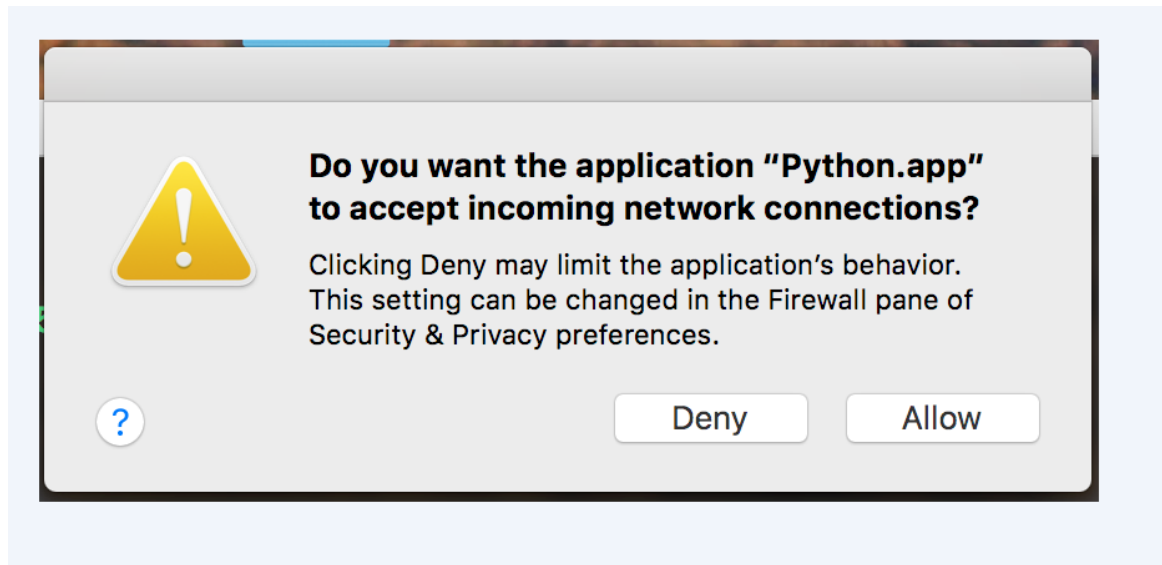


Fig 6.5 Allow program to use Local network

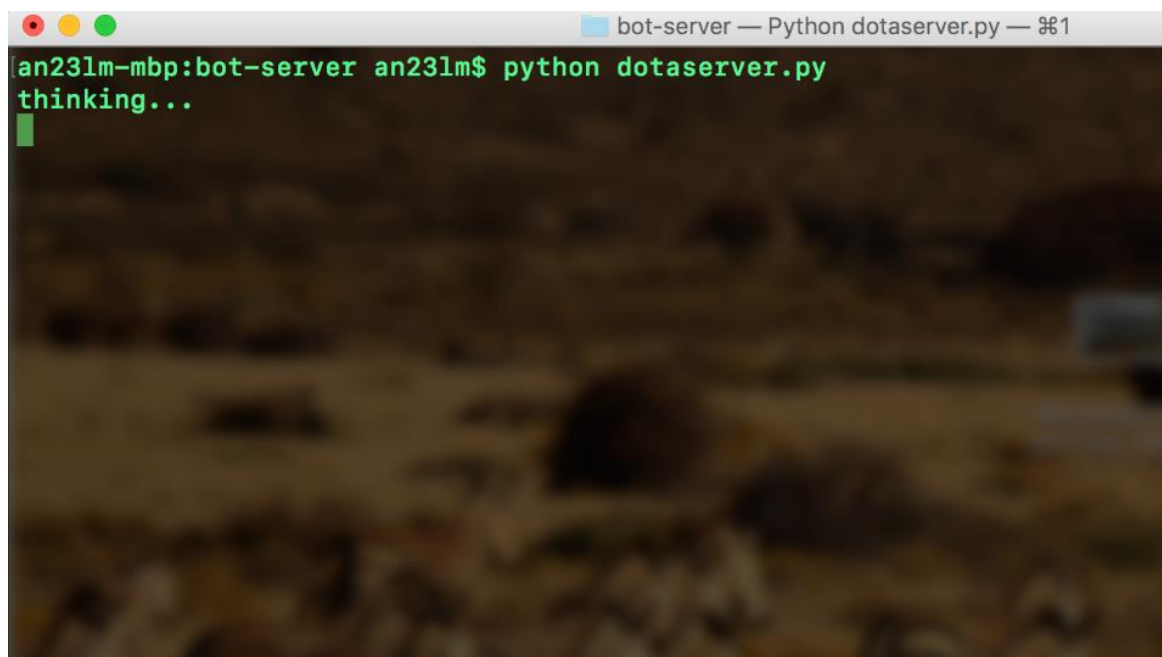
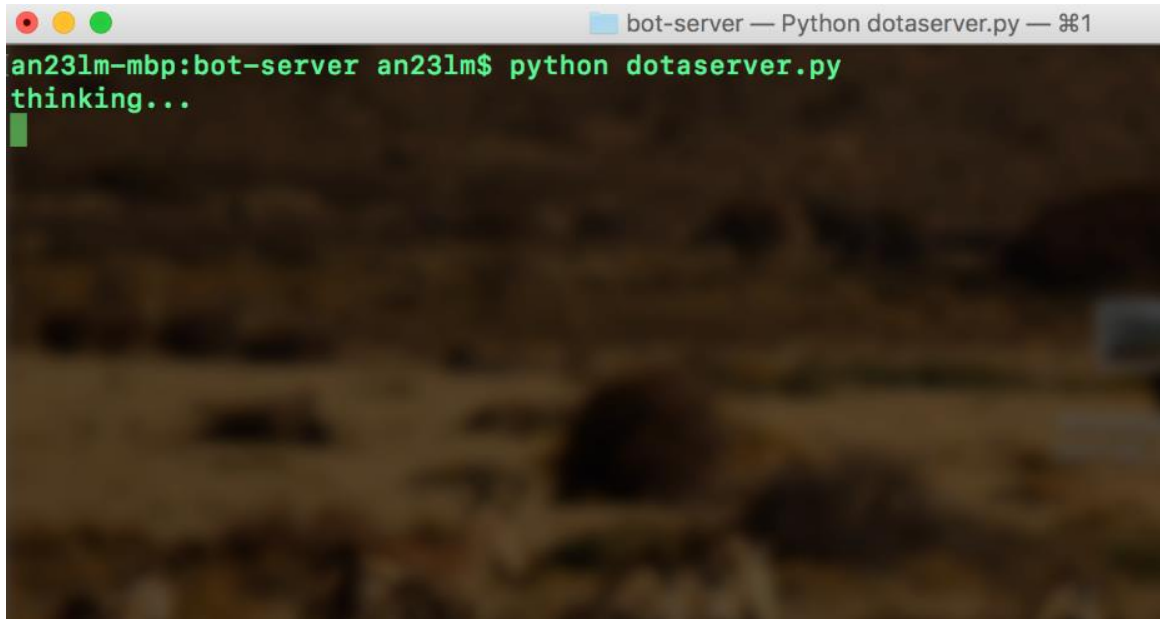
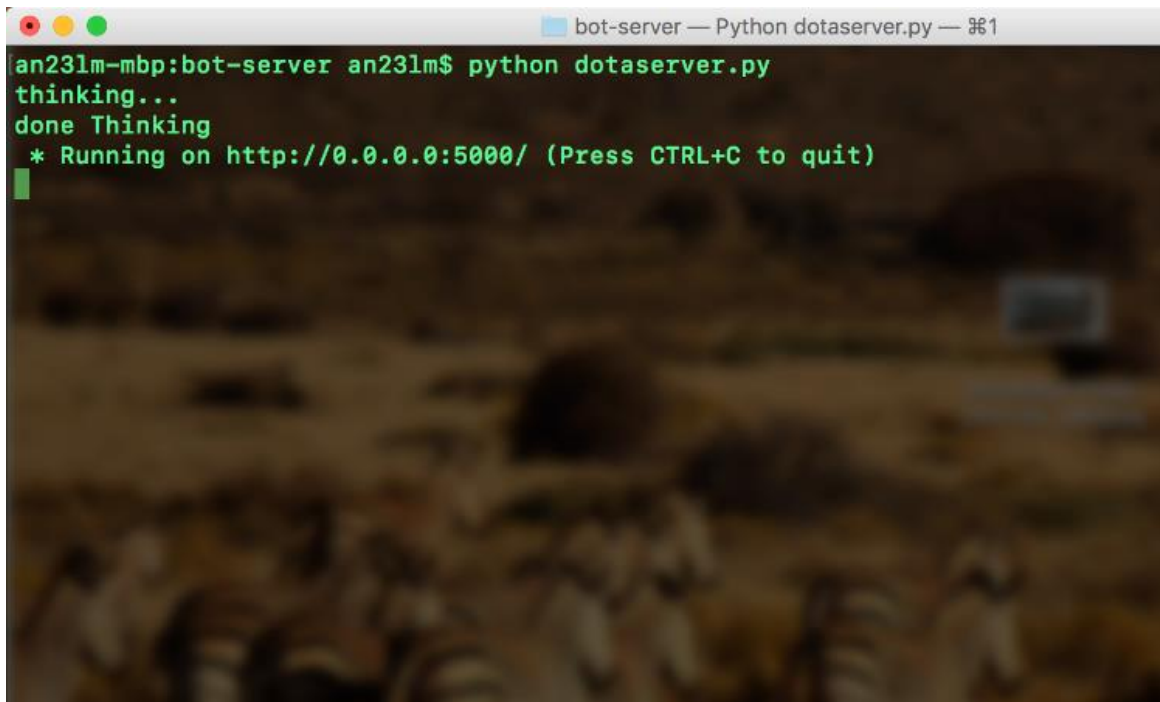


Fig 6.5 Python Bot Server

A screenshot of a macOS terminal window. The title bar at the top reads "bot-server — Python dotaserver.py — 1". The terminal text shows the command "python dotaserver.py" being executed, followed by the output "thinking...". A green cursor is visible on the line following the output.

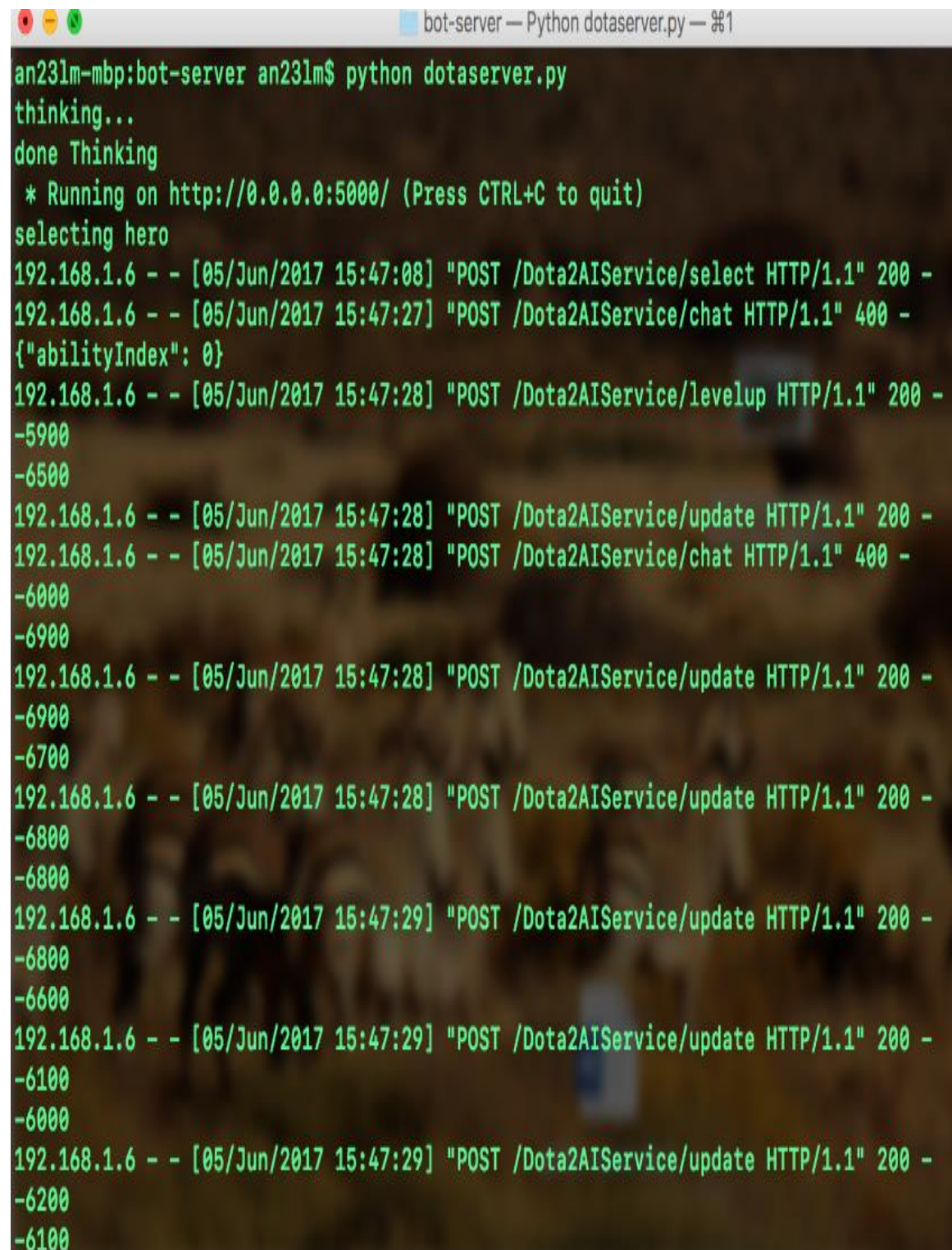
```
an231m-mbp:bot-server an231m$ python dotaserver.py
thinking...
```

Fig 6.7 Calculating Matrices

A screenshot of a macOS terminal window, similar to the previous one. The title bar reads "bot-server — Python dotaserver.py — 1". The terminal text shows the command "python dotaserver.py" being executed, followed by the output "done Thinking" and "* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)". A green cursor is visible on the line following the output.

```
an231m-mbp:bot-server an231m$ python dotaserver.py
done Thinking
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Fig 6.8 Server Running



```
an23lm-mbp:bot-server an23lm$ python dotaserver.py
thinking...
done Thinking
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
selecting hero
192.168.1.6 - - [05/Jun/2017 15:47:08] "POST /Dota2AIService/select HTTP/1.1" 200 -
192.168.1.6 - - [05/Jun/2017 15:47:27] "POST /Dota2AIService/chat HTTP/1.1" 400 -
{"abilityIndex": 0}
192.168.1.6 - - [05/Jun/2017 15:47:28] "POST /Dota2AIService/levelup HTTP/1.1" 200 -
-5900
-6500
192.168.1.6 - - [05/Jun/2017 15:47:28] "POST /Dota2AIService/update HTTP/1.1" 200 -
192.168.1.6 - - [05/Jun/2017 15:47:28] "POST /Dota2AIService/chat HTTP/1.1" 400 -
-6000
-6900
192.168.1.6 - - [05/Jun/2017 15:47:28] "POST /Dota2AIService/update HTTP/1.1" 200 -
-6900
-6700
192.168.1.6 - - [05/Jun/2017 15:47:28] "POST /Dota2AIService/update HTTP/1.1" 200 -
-6800
-6800
192.168.1.6 - - [05/Jun/2017 15:47:29] "POST /Dota2AIService/update HTTP/1.1" 200 -
-6800
-6600
192.168.1.6 - - [05/Jun/2017 15:47:29] "POST /Dota2AIService/update HTTP/1.1" 200 -
-6100
-6000
192.168.1.6 - - [05/Jun/2017 15:47:29] "POST /Dota2AIService/update HTTP/1.1" 200 -
-6200
-6100
```

Fig 6.9 Running Server

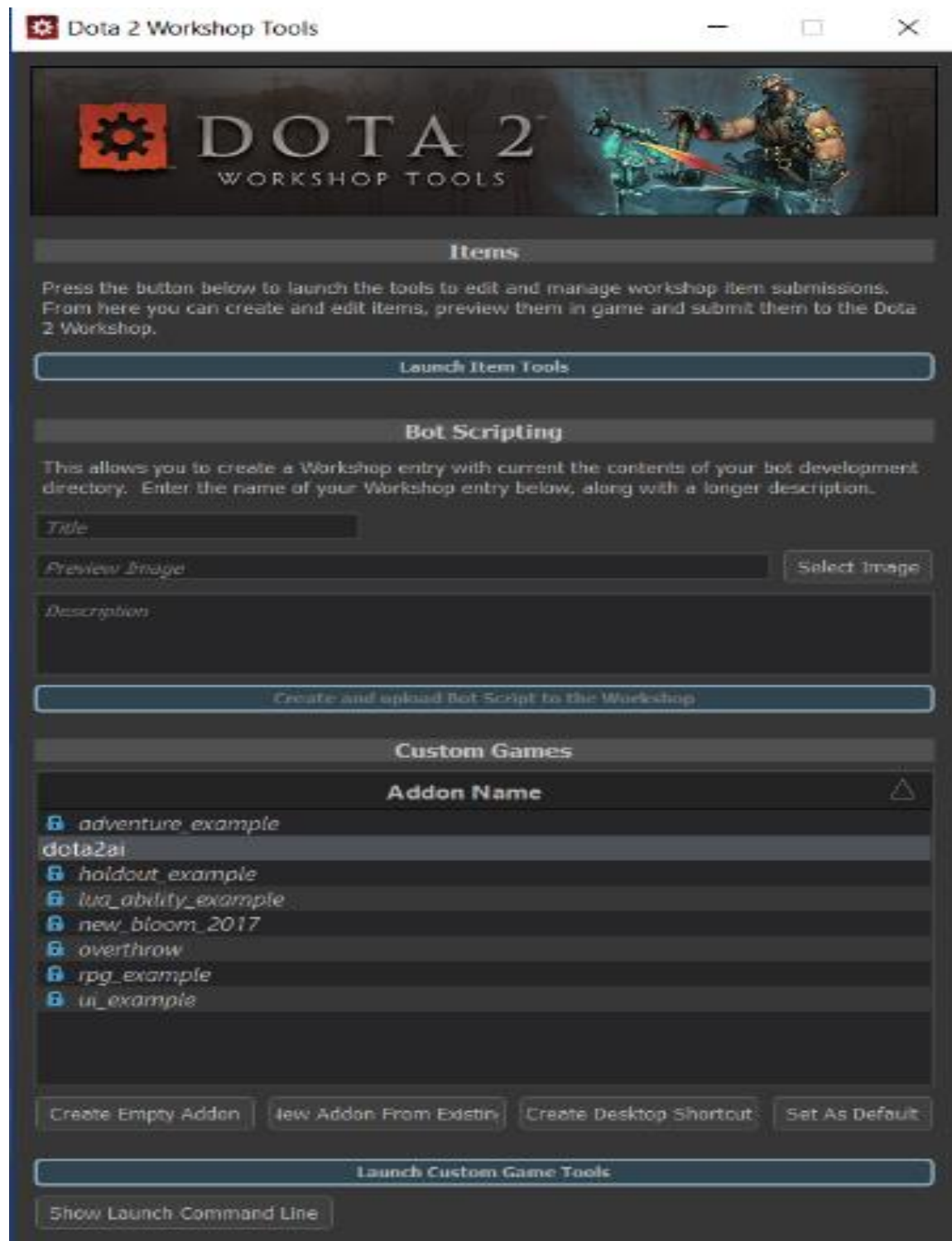


Fig 6.10 Workshop Tools

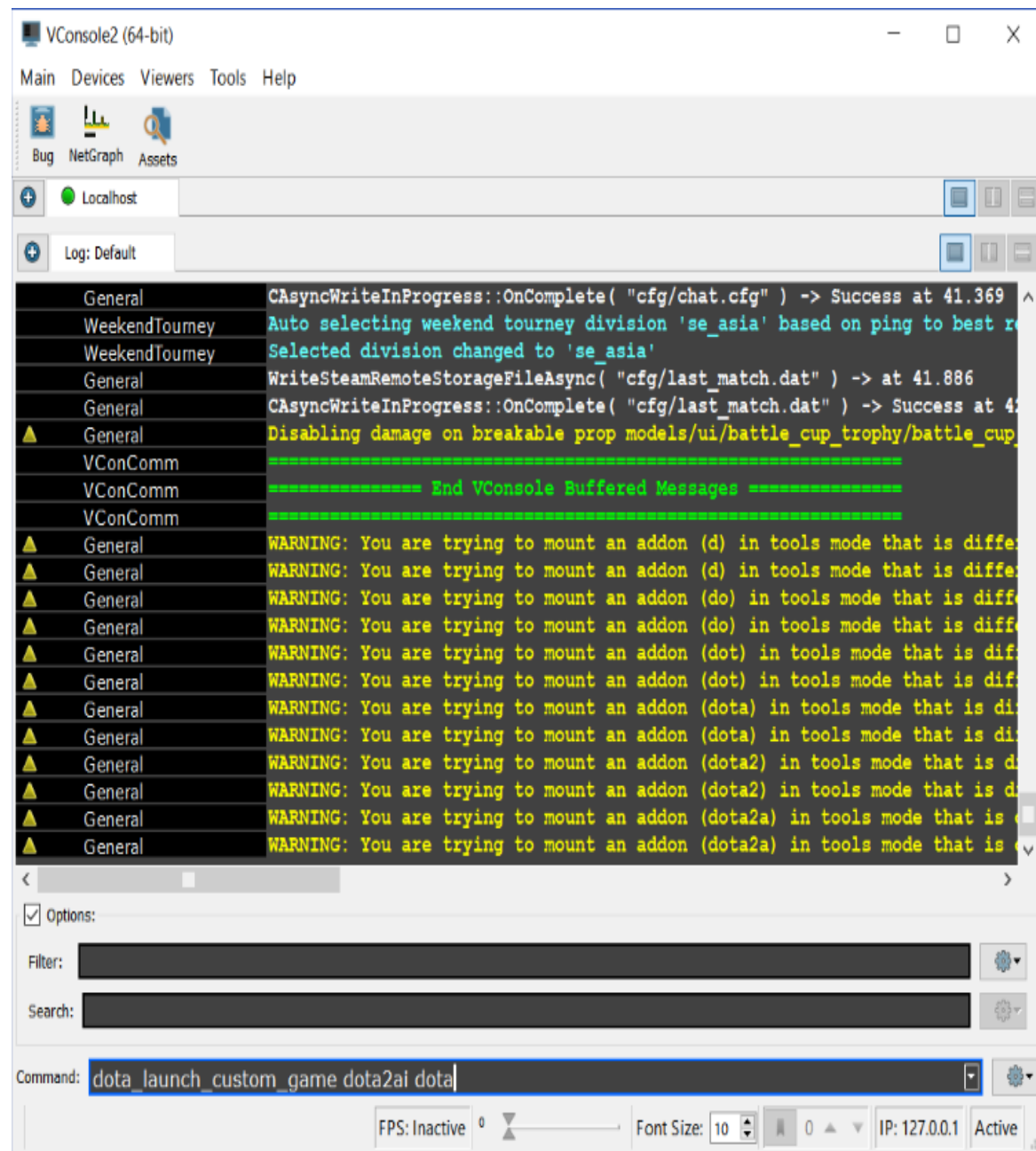


Fig 6.11 Console



CONCLUSION AND FUTURE ENHANCEMENT**8.1 CONCLUSION**

These days, machine learning techniques are being widely used to solve real-world problems by storing, manipulating, extracting and retrieving data from large sources. Supervised machine learning techniques have been widely adopted however these techniques prove to be very expensive when the systems are implemented over wide range of data. This is due to the fact that significant amount of effort and cost is involved because of obtaining large labeled data sets. Thus active learning provides a way to reduce the labeling costs by labeling only the most useful instances for learning.

Machine learning approaches are of particular interest considering steadily increasing search outputs and accessibility of the existing evidence is a particular challenge of the research field quality improvement. Increased reviewer agreement appeared to be associated with improved predictive performance.

8.2 FUTURE ENHANCEMENT

- Teaching our bot how to attack.
- Considering large amount of learning data for better efficiency.
- Bot automation in other fields.

BIBLIOGRAPHY

REFERENCE PAPERS:

- David Silver, Aja Huang, “Mastering the game of Go with deep neural networks and tree search”, 484 | NATURE | VOL 529 | 28 JANUARY 2016
- Yanzhu Du, Shisheng Cui, and Stephen Guo, “Applying Machine Learning in Game AI Design”
- Togelius, Sergey Karakovskiy and Julian, “Mario AI Competition.” In association with the IEEE Consumer Electronics Society Games Innovation Conference 2009 and with the IEEE Symposium on Computational Intelligence and Games, 9 12, 2009. [Cited: 12 9, 2009.]

TEXT BOOKS:

1. SQL The Complete Reference by *Paul Weinberg, James Groff and Andrew Oppel*
2. Pattern Recognition and Machine Learning by *Christopher M Bishop*
3. Machine Learning by *Tom M Mitchell*
4. Python The Complete Reference by *Martin C. Brown*
5. Go Programming by *John P. Baugh*
6. Data Communications and Networking, by *Behrouz A Forouzan*.

Sites Referred:

<https://www.python.org/about/gettingstarted/>

<http://ojs.pythonpapers.org/>

<http://hunch.net/>

<https://github.com/explore><https://golang.org/>

<https://sqlzoo.net/>