Department of Computer Science
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Optimization of behavioral parameters of Artificial Intelligence agents created by UtilityAI tool

Theme:
Scientific Theme

Project Period:
Fall Semester 2018

Project Group:

Participant(s):
Vladimir Caniga - 20163278

Supervisor(s):
Supervised by Assoc. Prof. Manfred Jaeger

Copies: 1

Page Numbers: 35

Date of Completion:
February 4, 2019

Abstract:

The goal of this project is to improve the development workflow of an Artificial Intelligence agent created using *UtilityAI* tool published by *Apex Game Tools*. Such agent contains multiple numeric parameters that need to be carefully optimized by the designer of the agent in order for the agent to behave in the intended way. Optimization of these parameters can be aided by definition of a reward function and employing optimizers that are then able to converge the parameters to values maximizing defined reward. Bayesian optimization was chosen as the optimization algorithm for this project and was evaluated on the Survival Shooter game and two known constructed functions.

Optimization of behavioral parameters of Artificial Intelligence agents created by UtilityAI tool

- Project Report -

Vladimir Caniga - 20163278

Supervised by Assoc. Prof. Manfred Jaeger

Aalborg University
Department of Computer Science

Keywords: Unity, Apex Game Tools, UtilityAI, Artificial Intelligence, Parameter Optimization, Bayesian optimization, Gaussian Process

Contents

Chapter 1

Introduction

1.1 Survival Shooter game

Survival Shooter is a simple isometric 3D game developed by Unity Technologies [8] mainly for technology showcase and tutorial purposes for their game engine named Unity. In the game, player controls a character that can move freely and perform several actions: shoot in a certain direction, reload his rifle, throw a bomb and use a band-aid. Moreover, he also can collect several power-ups randomly spawning across the map that include: +1 bomb and +1 band-aid that add one unit of throwable bomb and usable band-aid respectively to the player's inventory.

There are also enemies spawning in predefined locations on the map with a fixed frequency in a form of zombies that are trying to kill the player by running towards him and hitting him. The goal of this game is to survive as long as possible. Figure 1.1 shows a screenshot from the game.

1.2 UtilityAI

Utility AI is an advanced hierarchical scoring-based artificial intelligence framework for computer games. It can be plugged into the Unity game engine and provides user interface for designing and debugging AIs ranging from simple to complex ones. In contrast to common AI design approaches like behavior trees and finite state machines, Utility AI chooses approach of making several decisions available to the agent, at any point where decision should be made, and then scoring each potential decision. The Utility AI then selects the highest scoring decision and has the AI perform the action associated with this decision. This in theory allows for easier extension of the AI as its behavior can be changed by modifying scoring functions for different actions. This approach to designing AIs seems to be better in dealing

Figure 1.1: Survival Shooter in-game screenshot.

with unexpected and unforeseen situations which might have been undefined in other system using if-then conditions and can also appear more human-like [7].

UtilityAI is developed by Apex Game Tools.

1.2.1 Building Blocks of UtilityAI

When a developer decides to use UtilityAI to create an in-game artificial intelligence to either control the main character, an enemy or any other aspect of the game, he or she creates one or more so called agents. An agent can control the whole decision process of an in-game entity, for example the movement and actions of a soldier or only a portion of it. Example for that could be a real time strategy game, where one agent would be in charge of building a base and overall strategy and another one could be controlling individual units in the battle.

There are 4 main building blocks a developer uses when designing a new UtilityAI agent. Three of them form a hierarchy and from the topmost block consist of: Selectors, Qualifiers and Actions. Scorers are used in several places in the hierarchy 1.2.

- **Selector** sits at the topmost point of the building blocks hierarchy and encloses one or more qualifiers. There can be multiple selectors in one AI agent, however

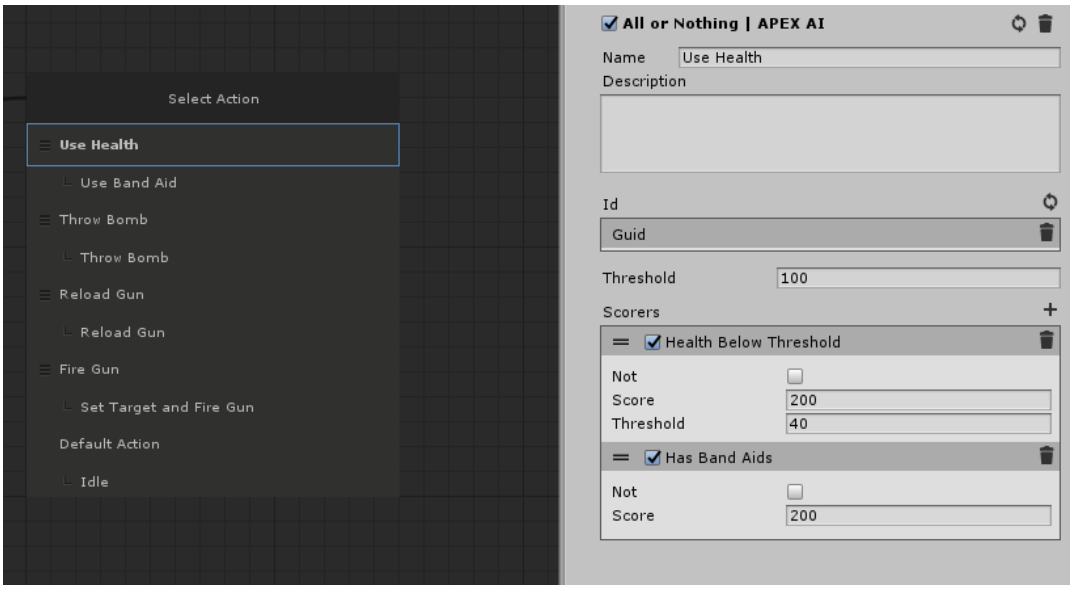

Figure 1.2: Screenshot from the UtilityAI editor showing one selector that contains multiple qualifiers and each of the qualifiers has one action associated. On the right side you can see configuration, including two scorers, for the highlighted qualifier named "Use health".

there always needs to be one selector marked as the entry point for the agent present. The responsibility of each selector is to choose one qualifier from the list of qualifiers it encloses. There are different rules on how this choice is performed, but the two commonly used rules are selecting the highest scoring qualifier and selecting the first qualifier from the list that scores above set threshold. The selectors can be interpreted as states in which the agent can find itself in. Selecting a qualifier can result in a "state transition" to another selector.

- **Qualifier** represents a logical decision/intention of the agent. Each selector might contain several qualifiers that in fact depict intentions that are applicable to the current state. When a qualifier gets selected by a selector, it can either invoke the Action associated with the intention represented by the qualifier or can lead to a "state transition" to another selector. The other selector then can be seen as a more specialized state that contains qualifiers appropriate for that situation. In order for a selector to be able to choose from different qualifiers, each qualifier needs to return a score for the current game state. For calculating such score the qualifiers use scorers that are described below. As with the selectors, there exist several types of qualifiers. Commonly used ones include a qualifier that returns the sum of score values of all scorers associated with it, a qualifier that only returns a score if all of it's associated scorers score above their respective thresholds and others.

- **Actions** are building blocks that execute code written specifically for the target game/application for each action the agent can perform. The main reason for adding this abstraction layers as opposed to simply executing the application specific function is that Actions can also make use of Scorers and set certain parameters of the executed code modifying its final behavior. As an example let's take an Action that moves a character in the game world. The Action object can have several Scorers associated with it that calculate the most favorable position for the character given the current game state. Action objects are always associated to qualifiers.

- **Scorers** are objects designed specifically for each game/application and contain code that evaluates a portion of current game state and returns a score. It is up to the designer of the agent to implement these scoring functions. The scoring functions can be as simple as counting the enemies in the vicinity of the player character to sampling the game world for positions where the controlled character can move and scoring them using complex functions counting in for instance the offensive and defensive properties of such positions.

1.3 Using UtilityAI in Survival Shooter

Apex Game Tools provides an implementation of three UtilityAI agents working together to control the player character in the Survival Shooter game. The three agents are the following:

- **PlayerScanner** agent periodically samples the game world and keeps track of all seen enemies, power ups and locations the player can move to.

- **PlayerMove** agent controls the movement of the player character. It is designed to continuously move the player character to the most favorable position based on the current game state. Conditions for determining the best position include: player distance from enemies, player distance from power-ups, effective firing range of player's rifle and line of sight to an enemy.

- **PlayerAction** agent continuously chooses and executes actions that are available to player's character. These actions include: using a band-aid, throwing a bomb, reloading the rifle and firing the rifle. Each of these actions has a set of specific rules (implemented by Scorers) that score them at any point of time in the game and so the agent is able to choose the most ideal action for any situation. Scores are computed based on the current health of the player's character, amount of band-aids and bombs in the inventory, rounds of ammunition left in the rifle and presence of enemies in the effective range of the rifle.

1.4 Project goal

When a developer finishes the design of an AI agent using UtilityAI for any application, he ends up with a hierarchy of selectors, qualifiers, actions and scorers. Assuming that all the programming work is done, there is still one important and arguably time consuming task left and that to manually optimize considerable amount of numeric variables that will inevitably be present in the finished hierarchy of UtilityAI building blocks. These parameters can have high impact on the decisions agents make and thus altering their behavior. In order to achieve the desirable agent behavior, extensive calibration runs must be performed during which these parameters are fine tuned until the agents make intended decisions in all situations.

The goal of this project is to explore the possibilities of automating this process and as a proof of concept, implement a solution for the agents provided by Apex Game Tools for Survival Shooter.

Chapter 2

Background

This chapter describes the theory behind different approaches that are commonly used for solving similar problems and with it also some arguments on why the chosen algorithm was the most suitable for this project's task.

2.1 Hyperparameter optimization

After exploring the implementation of UtilityAI tool and the workflow of designing new AI agents, one can identify strong similarity between the problem defined in this project and the problem of optimizing parameters in various machine learning models.

In the context of machine learning models, hyperparameter optimization is a problem of finding the optimal hyperparameters for the chosen learning algorithm. These hyperparameters usually cover different constraints, weights or learning rates specific to the different machine learning models. Finding and using the optimal hyperparameters yields the best performing model which minimizes the predefined loss function on given independent data.

There exist several different approaches that are commonly used to find such optimal hyperparameter values:

- **Grid search** is a simple exhausting search through all the different combinations in manually defined hyperparameter space of each hyperparameter of a learning algorithm. Since most of the hyperparameters of commonly used learning algorithms are real-valued and unbounded value spaces, manually set bounds and discretization are usually necessary before attempting grid search.
- **Random search** covers a family of algorithms that are initialized by selecting

a random value for each of the hyperparameter within its parameter space and then, at each step, try a new hyperparameter combination that is again randomly chosen in the predefined vicinity of the previous values. The algorithm then moves to the parameter setting that provided better results and repeats this step until a termination condition is met.

- **Gradient-based optimization** as the name suggests uses gradient descent to calculate the vector towards function minimum/maximum and move towards it. Gradient-based algorithms however have certain conditions, such as of course the ability to calculate the gradient.
- **Bayesian optimization** uses Gaussian processes to build a probabilistic model of the function mapping from hyperparameter values to the objective evaluated on a validation set. Bayesian optimization as well as Gaussian processes are described below in greater detail.

Each approach has its advantages and disadvantages, but since this project assumes the need to optimize a larger volume of real-valued unbounded parameters with minimal prior knowledge on the parameter space and a high cost loss function (as it requires to run the game for each evaluation), Bayesian optimization was chosen as the most appropriate one.

2.1.1 Gaussian processes

Let's start by looking at an example, where we try to find a function that best fits the observed data and let's say that we first opted for linear regression as an algorithm 2.1.

By choosing linear regression, we are limited to finding the distribution over the 2 parameters in its equation: $y = \theta_0 + \theta_1 x + \epsilon$. By examining the result one can see that we would most likely achieve lower error rate by choosing a quadratic function, so in the regression we would be looking for distribution over 3 parameters: $y = \theta_0 + \theta_1 x + \theta_2 x^2$. Gaussian process considers all possible functions in a predefined domain, effectively removing the problem of choosing the right function from the start.

A Gaussian Process is an extension of the multivariate Gaussian distribution to an infinite dimension stochastic process for which any finite combination of dimensions will be a Gaussian distribution. Just as a Gaussian distribution is a distribution over a random variable, completely specified by its mean and covariance, a Gaussian Process is a distribution over functions, completely specified by its mean function,

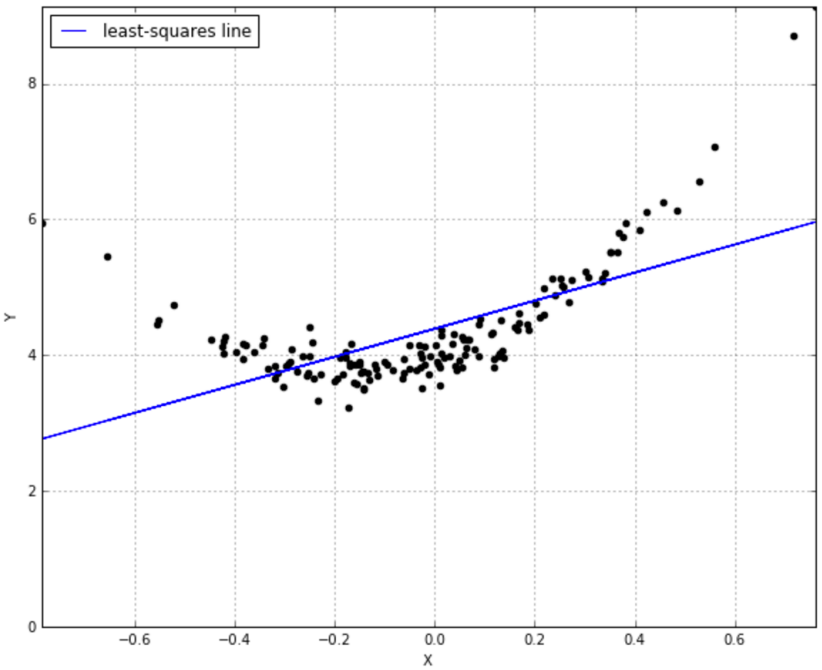

Figure 2.1: Finding a linear function to best fit the data points. Source: [1]

m and covariance function (or kernel) [2], k:

$$f(x) \sim GP(m(x), k(x, x'))$$

Mean function is analogous to a mean value for normal distribution but for the whole range of function values and covariance function is a function that describes relations between points that are close together. The idea is that it is very common to see two points that are close together in the input space to produce output values that are also close. It is often useful to intuitively think of a Gaussian Process as analogous to a function, but instead of returning a scalar $f(x)$ for an arbitrary x, it returns the mean and variance of a normal distribution over the possible values of f at x 2.2.

Various covariance functions assign different probability to the shape of the resulting objective function. It is therefore important to choose an appropriate kernel if any prior knowledge exists. Here is an example of very smooth functions generated by the Squared Exponential kernel 2.3.

And here are "spiky" functions generated by the Exponential kernel 2.4.

Let's examine the Squared Exponential kernel. The function approaches 1 when the values are close together and 0 as they get further apart. Two points that are

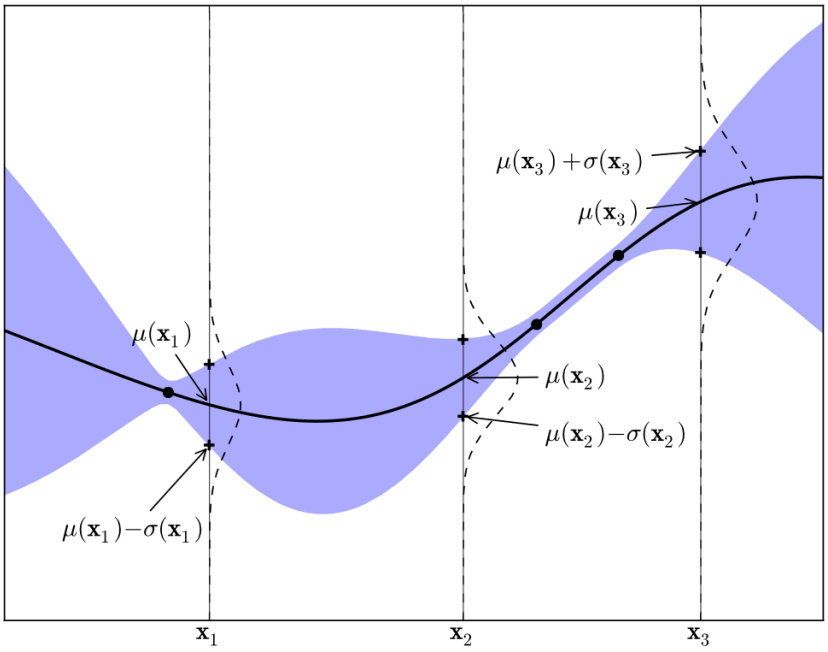

Figure 2.2: An example of a Gaussian process with 3 observations. The black line is the mean prediction of the objective function given the data and the colored area represent the mean plus and minus the variance.

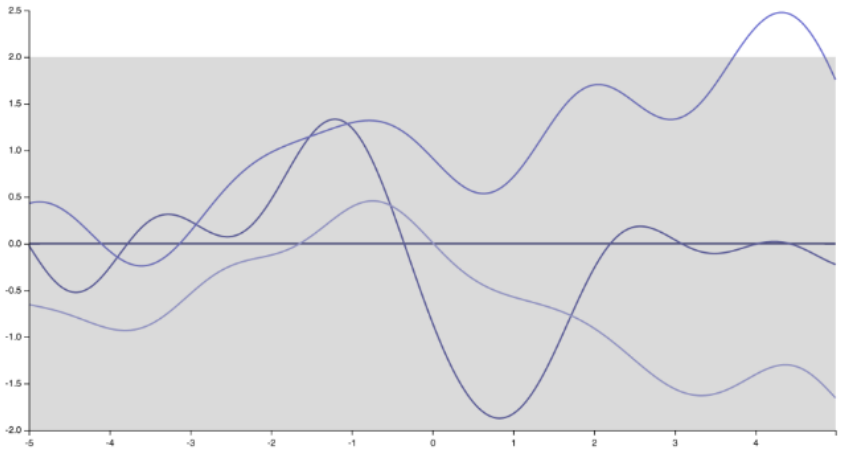

Figure 2.3: "Smooth" functions generated by the Squared Exponential kernel. Source: `https://stats.stackexchange.com/questions/222238/why-is-the-mean-function-in-gaussian-process-uninteresting/222304`

close together can be expected to have a very large influence on each other, whereas distant points have almost none. If we were to sample from the prior, we would choose $\{x_{1:t}\}$ and sample the values of the function at these indices to produce the

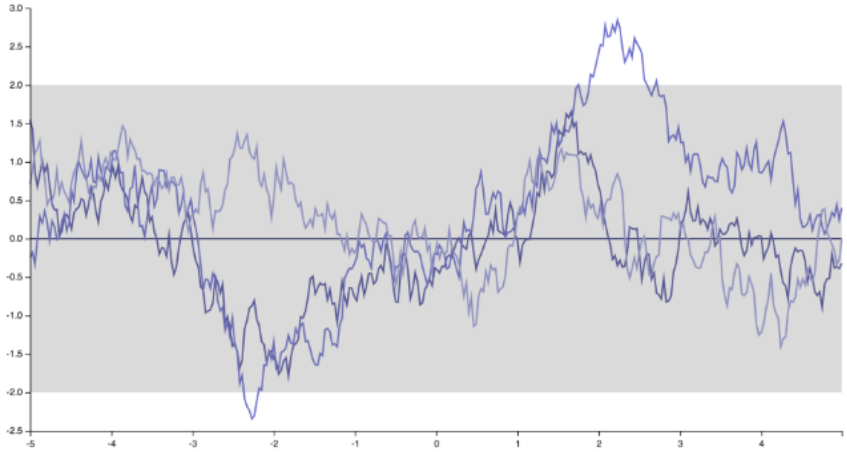

Figure 2.4: "Spiky" functions generated by the Exponential kernel. Source: `https://stats.stackexchange.com/questions/222238/why-is-the-mean-function-in-gaussian-process-uninteresting/222304`

pairs $\{x_{1:t}, f_{1:t}\}$, where $f_{1:t} = f(x_{1:t})$. The function values are drawn according to a multivariate normal distribution $\mathcal{N}(0, K)$, where the kernel matrix, which describes the correlation between all the different pairs of sampled indices, is given by:

$$\mathbf{K} = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \dots & k(\mathbf{x}_1, \mathbf{x}_t) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_t, \mathbf{x}_1) & \dots & k(\mathbf{x}_t, \mathbf{x}_t) \end{bmatrix}$$

In a noise-free environment the diagonal values of the matrix are all 1 (each point correlates perfectly with itself). Assuming that we already have the observations $\mathcal{D}_{1:t} = \{x_{1:t}, f_{1:t}\}$ and we will use Bayesian optimization to find the next point x_{t+1} which should be considered next. Let's denote the value of the function at this arbitrary point as $f_{t+1} = f(x_{t+1})$ and we get [2]:

$$\begin{bmatrix} \mathrm{f}_{1:t} \\ f_{t+1} \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} \mathbf{K} & \mathbf{k} \\ \mathbf{k}^T & k(\mathbf{x}_{t+1}, \mathbf{x}_{t+1}) \end{bmatrix}\right)$$

where

$$\mathbf{k} = \begin{bmatrix} k(\mathbf{x}_{t+1}, \mathbf{x}_1) & k(\mathbf{x}_{t+1}, \mathbf{x}_2) & \cdots & k(\mathbf{x}_{t+1}, \mathbf{x}_t) \end{bmatrix}$$

The expression for the predictive distribution using Sherman-Morrison-Woodbury formula is [2]:

$$P\left(f_{t+1}|\mathcal{D}_{1:t}, \mathbf{x}_{t+1}\right) = \mathcal{N}\left(\mu_t\left(\mathbf{x}_{t+1}\right), \sigma_t^2\left(\mathbf{x}_{t+1}\right)\right)$$

where

$$\mu_t\left(\mathbf{x}_{t+1}\right) = \mathbf{k}^T\mathbf{K}^{-1}\mathbf{f}_{1:t}$$
$$\sigma_t^2\left(\mathbf{x}_{t+1}\right) = k\left(\mathbf{x}_{t+1}, \mathbf{x}_{t+1}\right) - \mathbf{k}^T\mathbf{K}^{-1}\mathbf{k}$$

2.1.2 Bayesian optimization

Bayesian optimization is a strategy that can be used to find the extrema of objective functions that are expensive to evaluate. It is applicable in situations where the expression of the objective function is unknown, but it is possible to obtain observations of this function at sampled values. It is especially useful when it is expensive to evaluate the observations, the derivatives of the function are not available, or the function is non-convex (it has multiple local maxima/minima) [2] [3]. The goal of Bayesian optimization can be defined as: $\max_{x \in A} f(x)$, where x is in $\mathbb{R}^d$ and d is no too large, typically $d \leq 20$.

Bayesian optimization is, as the name suggests, based on Bayes' theorem which applied to the current problem says that the posterior probability of a model M given evidence E is proportional to the likelihood of E given M multiplied by the prior probability of M:

$$P\left(M|E\right) \propto P(E|M)P(M)$$

Bayesian optimization by design requires 2 components. The first one being a source of prior distribution for the model and the second one, an acquisition function [3] [4]. The prior distribution provides information on the probability of extrema values being located in the given point based on previous observations and for this project, Gaussian process is used and is described in greater detail in the previous section. The second component, Acquisition function, then guides the search for optimum and at each step of the Bayesian optimization selects the next point in which to sample and evaluate the objective function based on the prior distribution.

Listing 1 shows the pseudo algorithm for Bayesian optimization. As a first step on line 2 we find the next index x_t at which to evaluate the objective function by optimizing the acquisition function given the previous observations in the Gaussian process. Next we evaluate the objective function at index x_t and lastly we augment the data with the new index and its function value and update the Gaussian process

to reflect the new observation.

for *t = 1, 2, ...* **do**
  Find x_t by optimizing the acquisition function over the GP: $x_t = \max_x a(x|\mathcal{D}_{1:t-1})$;
  Sample the objective function: $y_t = f(x_t)$;
  Augment the data $\mathcal{D}_{1:t} = \{\mathcal{D}_{1:t-1}, (x_t, y_t)\}$ and update the GP;
end

Algorithm 1: Bayesian optimization algorithm

Acquisition functions

There are several acquisition functions that are commonly used for Bayesian optimization, but all of them serve the same purpose, which is to select the next point of the objective function to evaluate. This point is chosen by maximizing the acquisition function itself [9].

Authors of paper [6] provide nice and concise overview of three commonly used acquisition functions. Let's define function $f(x)$ as a function drawn from Gaussian process prior and known observations of the objective function as $\{x_n, y_n\}_{n=1}^{N}$. Observations together with prior induce posterior over functions. Acquisition function, defined as $a : \chi \to \mathbb{R}^+$, decides what point in χ should be evaluated next via a proxy optimization $x_{\text{next}} = \max_x a(x)$. Formula $a(x; \{x_n, y_n\}, \theta)$ depicts the dependence of the acquisition function on previous observations and on parameters of the Gaussian process. Predictive mean function and predictive variance function of the Gaussian process are $\mu(x; \{x_n, y_n\}, \theta)$ and $\sigma^2(x; \{x_n, y_n\}, \theta)$ respectively. The best current value is $x_{\text{best}} = \min_{x_n} f(x_n)$ and the cumulative distribution function of the standard normal as $\Phi(\cdot)$

- **Probability of improvement** is a strategy to maximize the probability of improving over the best current value, which under Gaussian process can be computed as:

$$\gamma(x) = \frac{f(x_{\text{best}}) - \mu(x; \{x_n, y_n\}, \theta)}{\sigma(x; \{x_n, y_n\}, \theta)}$$

$$a_{\text{PI}}(x; \{x_n, y_n\}, \theta) = \Phi(\gamma(x))$$

- **Expected Improvement** can be maximized over current best under Gaussian process as:

$$a_{\text{EI}}(x; \{x_n, y_n\}, \theta) = \sigma(x; \{x_n, y_n\}, \theta)(\gamma(x)\Phi(\gamma(x))) + N(\gamma(x); 0, 1)$$

- **Upper/Lower confidence bound** is a strategy that constructs acquisition functions that minimize regret over the course of their optimization and can be written as:

$$a_{\mathrm{LCB}}(x; \{x_n, y_n\}, \theta) = \mu(x; \{x_n, y_n\}, \theta) - \kappa\sigma(x; \{x_n, y_n\}, \theta)$$

  where κ is parameter that balances exploitation and exploration.

Exploitation-exploration trade-off

Most acquisition functions have parameters that modify their balance of exploration and exploitation. Modifying such parameter can obviously have a big impact on the performance of the Bayesian optimization algorithm. When the acquisition function is tuned for exploration, it should pick points in which variance is high and when it is tuned for exploitation, it should pick points where mean is high [2]. Figure 2.5 shows the the effect different settings of exploration/exploitation parameter can have on acquisition functions. At the top is Gaussian Process posterior and below that are three acquisition functions with three different exploration/exploitation settings – Probability of Improvement, Expected Improvement and Upper Confidence Bound.

Figure 2.6 shows three instances of first three steps of Bayesian optimization on one dimensional problem. From the top to bottom, the acquisition functions are *probability of improvement*, *expected improvement* and *upper confidence bound*. The red dotted line is the objective function, solid blue line is posterior mean of the Gaussian process and the blue area its variance. The green graph shows the acquisition function utility and the red triangle the next point that will be chosen for evaluation. Even though the Bayesian optimization was initialized with the same points, there is a considerable difference between exploration and exploitation characteristics of the different acquisition functions.

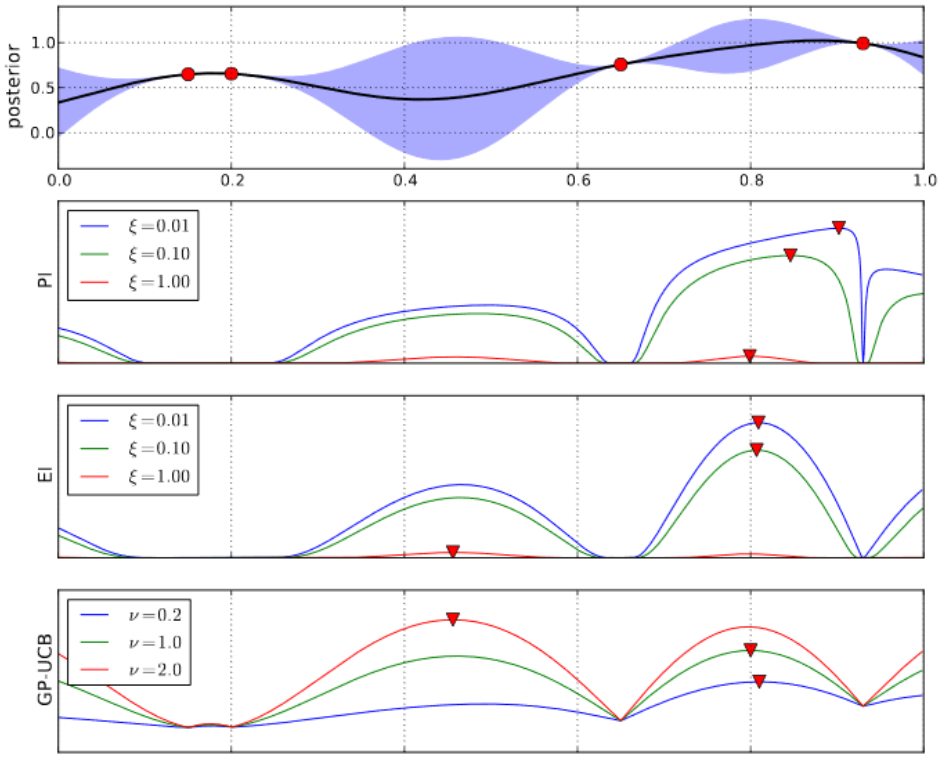

Figure 2.5: Comparison of the effect the exploration/exploitation setting has on the three acquisition functions – Probability of Improvement, Expected Improvement and Upper Confidence Bound in regards Gaussian Process posterior at the top. Source: [2]

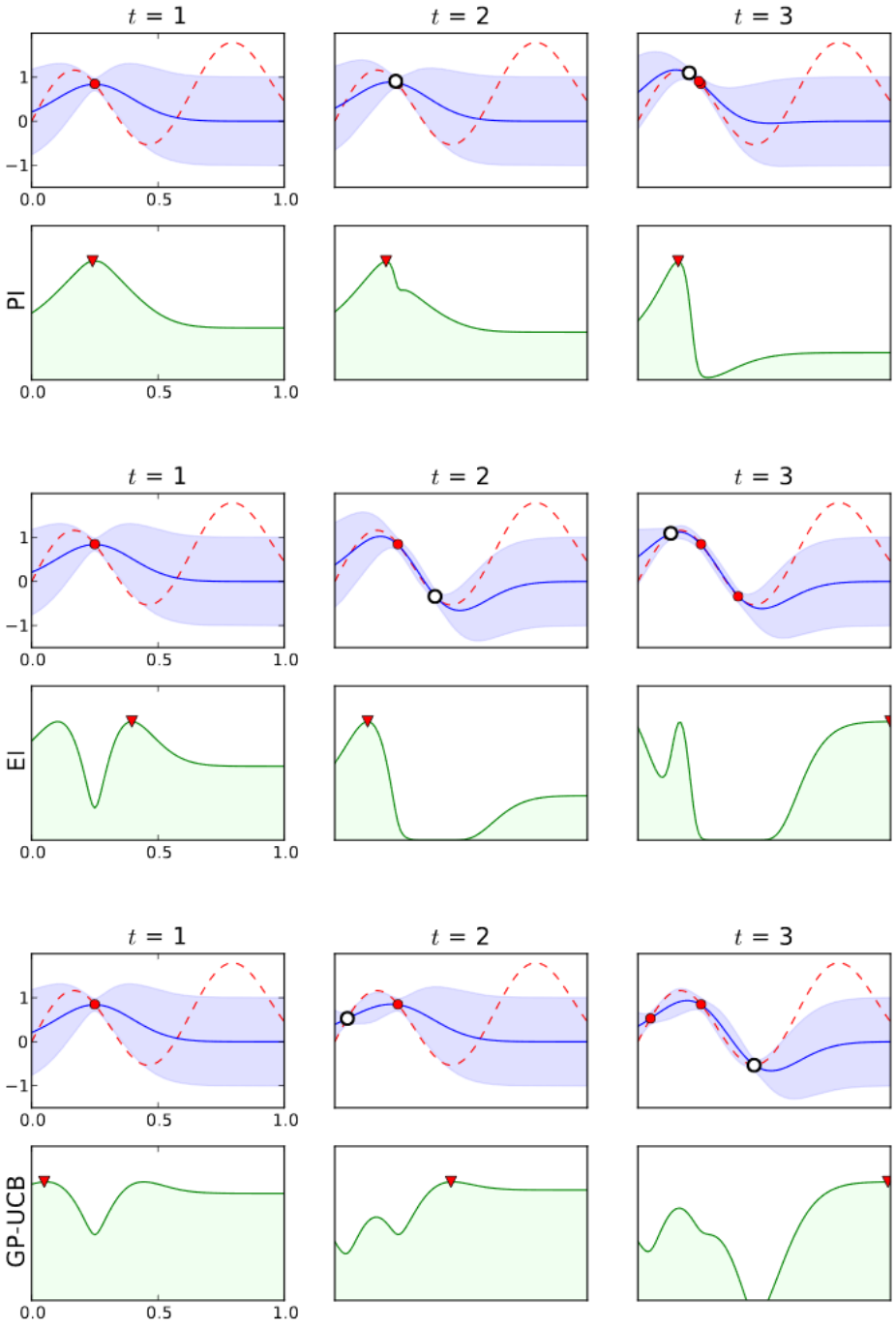

Figure 2.6: First three steps of Bayesian optimization on one dimensional problem. Source: [2]

Chapter 3

Tools and Implementation

3.1 Tools used in the project

BayesianOptimization python library

BayesianOptimization is a library developed by Fernando Nogueira [5] that provides implementation of global Bayesian global optimization and is built upon Bayesian inference and Gaussian processes that are implemented as a part of popular python library Scikit-learn.

The implementation of the algorithm supports optimization single as well as multi-dimensional inputs, meaning that it is suitable for tasks with only one parameter and also tasks with multiple parameters. It also offers several different acquisition functions: upper confidence bound, expected improvement and probability of improvement. For each of those functions it is possible to adjust the balance between exploration and exploitation changing the behavior of the optimization algorithm.

3.2 Implementation

Considerable time working on this project was spent on creating the communication interface between the game, the agents controlling the player character and a separate program for the optimization algorithm itself.

Unfortunately, there was no API included in the Utility AI that would make the process easier. It was necessary to modify several parts of the code base to enable extraction of the parameters out of the agents configuration and then, using reflection, create a way of injecting new values back into them. To make the matter worse, by mistake I received an incompatible development version of Utility AI software and it took some time until I received the correct one.

Survival Shooter game and UtilityAI tool are both written in C# programming language. There were also several modifications needed to be performed on the game itself. Since it does not support disabling graphical rendering to drastically speed up the simulation, I attempted to optimize rendering to use the least amount of resources possible to reduce the cost of evaluation of loss function used by the algorithm. In the final version, the game could run about 20 times faster than real time.

Later on it was unfortunately discovered that running the game any faster than real-time negatively affected performance of the bot and all the tests had to be executed in real-time. The exact reason for this performance degradation is unknown, but the evidence suggests that since the game simulation is tightly connected to the frame-rate of the graphics output, any small frame-rate fluctuation or frame delay introduces some randomness factor in the evaluation runs. The fact that results from evaluations executed at 2x speed yielded approximately 2/3 score value compared to results from evaluations at real-time speed even though the computer executing these evaluations was capable of maintaining consistent frame-rate for both of these scenarios further suggests that there are some simplifications and inconsistencies implemented in the Unity game engine which I was not able to manipulate. Despite running all the evaluations in real-time, there was still considerable variation between the results of individual game runs. This is again most likely caused by slightly different time delays between frames and thus the AI making decisions on slightly different game sates as in other evaluation runs. Even after all my efforts to improve the stability of the results of evaluations, it was very common to see them spread across the whole range of possible outcomes (from 0 seconds to 600 seconds). As an example, here are the results of five consecutive runs of the game with the exact same setup (the following numbers represent the amount of seconds the bot survived): **219.02**, **188.66**, **240.91**, **54.48**, **152.20**.

Throughout this thesis I ran a large number of different evaluations, I tried several sets of parameters that were responsible for different parts of the bot decision process in order to find parameters which would visibly change the bot's behavior as well as the game's outcome. Finding such parameters proved to be a very difficult task, mainly because of all the issues with inconsistencies that were impossible for me to address. Another factor was the design of agents that I was working with. The design could be described as “defensive" and was not the most suitable for common parameter optimization techniques. I will try to explain why on a set of parameters that were responsible for prioritizing different actions the bot could execute.

Figure 3.1 shows selector named **Select Action** that is responsible to choose the next action the bot is about to perform. This selector does not select a qualifier that achieves the highest score. Instead, it selects the first qualifier that scores above

threshold that is set for each qualifier. That means there already exists some prioritization of qualifiers/actions based on their order in the selector. If we examine this structure even deeper, the score for qualifier **Reload Gun** only depends on one scorer as seen in the figure 3.2. This scorer is very simple and returns a set value when the magazine of bot's gun is empty and zero otherwise. The value this scorer returns is one of the parameters the Bayesian optimization tries to fine-tune. However, in this agent design, there are only two different outcomes one can achieve. If the value is set to zero, the bot will never be able to reload his gun and when the value is set to anything higher than zero, the bot will always reload exactly when his magazine becomes empty. Using such a design for example does not allow the bot to reload his gun when he still has a few rounds in the magazine, even though it could be a good decision as there might be no enemies around him at that moment. It also makes the shooting process "fool-proof" as when the bot is allowed to reload his gun, the action to try to shoot the gun can never be chosen when his gun is empty.

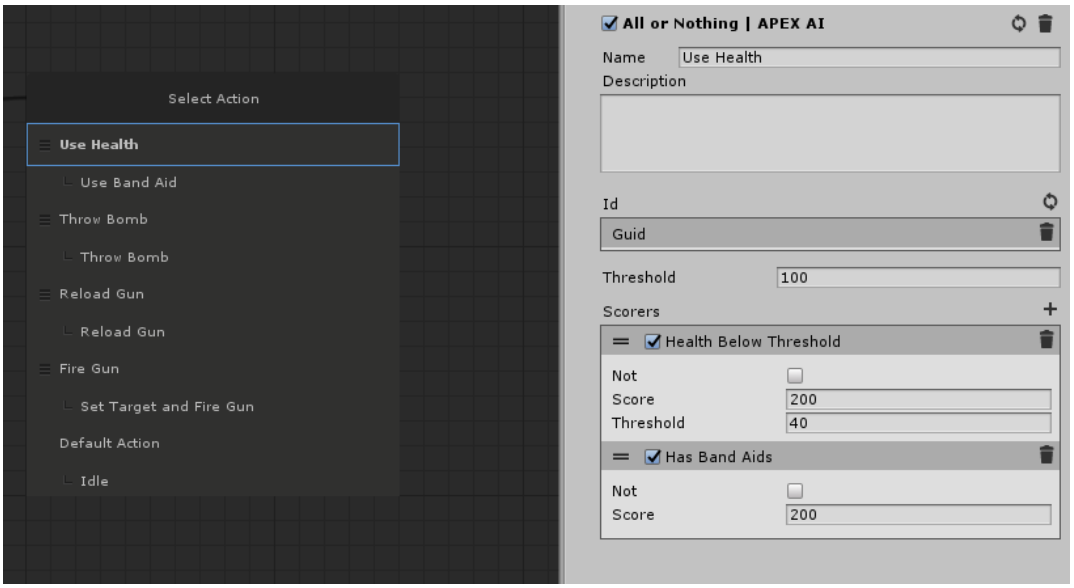

Figure 3.1: Screenshot from the UtilityAI editor showing one selector that contains multiple qualifiers and each of the qualifiers has one action associated. On the right side you can see configuration, including two scorers, for the highlighted qualifier named "Use health".

There are many occurrences of this "defensive"" design throughout the agents' implementation which cause individual parameters to have almost binary character and the different values that are above the critical threshold cause very small change in the final behavior of the bot.

Learning algorithm was implemented as a separate program in Python programming language. The loss function for Bayesian optimization algorithm was the average

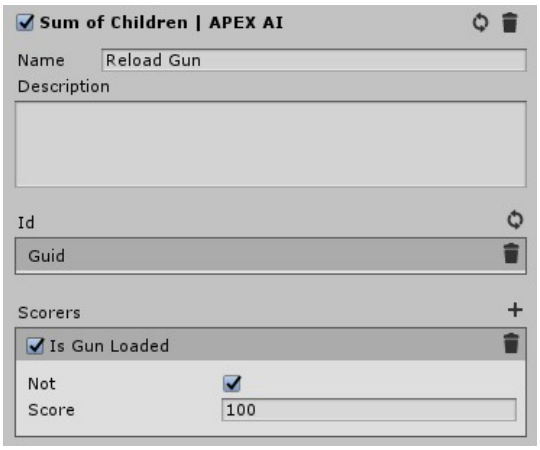

Figure 3.2: Screenshot from the UtilityAI editor showing properties and scorers of “Reload Gun"" qualifier.

survival time of the player character with parameters for his AI agents set accordingly. The need for averaging survival time across several runs of the game comes from the above described non-deterministic properties of the simulations. As a result, evaluation of the loss function is very costly and thus requiring the algorithm to converge relatively quickly.

Chapter 4

Experiments and results

All of the experiments were executed using the modified version of Utility AI software and the modified version of Survival Shooter game. Since there exist non-deterministic elements in the game design, every performance measurement was an average of 5 separate gameplays using the same settings for Utility AI agents. The scoring method was simple – the longer the agent keeps the player character alive the better.

The baseline was established as the performance of Utility AI agents manually tuned for Survival Shooter by Apex Game Tools team. These agents turned out to have an excellent performance in this particular game and they could survive in the game practically indefinitely. This also applied to most of the random parameter settings and for that reason, each game run was time limited to 10 minutes as well as some modifications were performed on the game itself to increase its difficulty – spawning frequency of the different enemies was increased. The maximum score an agent could achieve is then 600 seconds.

The process of experiments with automatically optimized parameters using Bayesian optimization was the following. At first only a small number of parameters (one to two) of the agents were set to a random value and the optimization algorithm was set to optimize only those values. Other parameters remained the same as on the manually tuned agents. The number of parameters that were randomized and optimized by the algorithm was then gradually increasing.

Multiple sets of parameters were chosen and efficiency of Bayesian optimization was evaluated on each of them. These parameter sets controlled different aspects of the bot. For example parameters in scorers for qualifiers **“Use Health"**, **“Reload Gun"** and **“Fire Gun"** can to a certain degree set the priority of equally named player actions, parameters in scorers for Action object **“Set Best Attack Target"** influence

the choice of a target to be shot at next and parameters in Action object **“Move To Best Position"** are used to choose the ideal position the player should move to next.

Because of the way the AI agents are implemented for Survival Shooter, modifications to these parameters don't seem to have a great effect on the final performance of the bot. The difference in performance is even harder to determine as individual evaluations are inconsistent. Examining the design of these agents, one can find parameter settings for certain parameter groups that will hamper the performance of the bot considerably. An example is setting all parameters to value zero in scorers for qualifiers **“Use Health"**, **“Reload Gun"** and **“Fire Gun"**, which results in the inability of the bot to execute any action and thus he only tries to run away from his enemies that will always catch him after a short while. Different non-zero values for these parameters achieve better score in evaluations, however there is no consistent trend in the score between these settings. In figure 4.1 you can see progress with intermediate results of optimization algorithm on a set of parameters that affect the choice of player action. Column “score" shows the survival time in seconds and columns right of that show the settings for different parameters. The first iteration shows the special case of a bad parameter setting and the score is significantly lower for that case.

4.1 Verifying validity of the implementation on constructed deterministic function

In order to verify functionality and viability of the proposed solution together with the tools used in the project, simulating an environment in which hardware and third party software variables can be controlled, I ran experiments, which differed from previously described experiments only by changing the evaluation function from running the Survival Shooter game to evaluating a constructed deterministic function.

4.1.1 Function with two unknown variables

The first chosen evaluation function was:

$$-x^4 + 5x^2 - 10y^2$$

Both unknown variables have limited interval $[-200, 200]$ and the function reaches it's maximum in: $x = \pm 1.5811$; $y = 0$.

Having only two unknown variables enables us to visualize the function easily (figure 4.2 plots the evaluation function around it's maximums) and thus monitoring

iter	score	fire_g...	has_ba...	health...	reload...
1	34.35	0.0	0.0	0.0	0.0
2	203.2	13.68	320.9	208.1	1.848
3	192.7	420.3	117.0	48.6	101.2
4	182.3	47.9	119.6	213.2	163.3
5	227.9	5.625	315.9	219.9	3.05
6	171.1	376.1	291.9	439.9	466.9
7	190.2	14.76	305.8	219.4	1.434
8	221.0	461.7	142.3	336.9	391.5
9	175.0	7.455	251.2	468.0	355.7
10	177.2	464.6	152.1	344.4	395.7
11	218.5	345.5	245.1	205.5	422.1
12	160.6	443.5	54.62	11.98	32.3
13	165.9	156.0	221.6	215.3	101.8
14	210.6	459.5	145.4	340.7	395.3
15	178.8	348.9	253.3	197.7	426.1
16	173.9	351.2	239.2	210.5	424.0
17	161.0	462.2	147.4	329.3	392.6
18	168.9	79.88	176.5	421.4	229.1
19	139.7	430.7	36.23	195.2	13.36
20	95.15	368.7	33.06	111.0	316.0
21	187.6	497.6	136.1	406.8	61.34
22	189.9	447.4	418.7	469.9	224.1
23	179.3	200.1	262.7	366.1	85.45
24	197.3	134.6	219.6	342.2	200.0
25	217.3	215.3	267.5	216.4	399.3
26	119.4	161.3	33.22	358.7	245.3
27	201.9	184.5	293.2	420.0	116.0
28	153.1	77.14	224.2	295.9	467.3
29	169.3	410.6	246.4	405.0	147.1
30	191.4	271.5	230.7	262.2	201.5
31	140.2	76.11	133.9	332.8	116.9
32	236.6	10.96	320.6	215.8	1.641
33	215.4	412.8	472.5	438.5	188.2
34	189.4	479.8	310.3	340.3	222.3
35	192.2	371.8	467.3	406.7	450.8
36	203.0	179.2	229.9	115.5	190.4
37	212.9	88.6	114.7	474.0	494.9
38	219.1	29.54	198.5	341.0	252.7
39	226.4	316.7	144.9	91.51	118.1
40	200.5	134.5	250.9	291.9	33.09
41	177.9	418.8	63.5	486.5	16.43
42	144.6	1.425	204.1	17.42	171.7
43	133.0	466.3	22.81	47.29	237.8
44	143.8	193.5	28.31	415.3	21.76
45	187.1	403.9	196.1	328.0	360.7
46	134.0	328.6	16.32	380.9	390.5

Figure 4.1: Results of optimization of parameters responsible for player actions

the optimizing algorithm is more straight forward. After tweaking a parameter that balances exploration and exploitation in the Expected Improvement acquisition function in the Bayesian optimization algorithm, it was possible to find values for the unknown variables that were within 5% of the real function's maximums in less than 100 iterations.

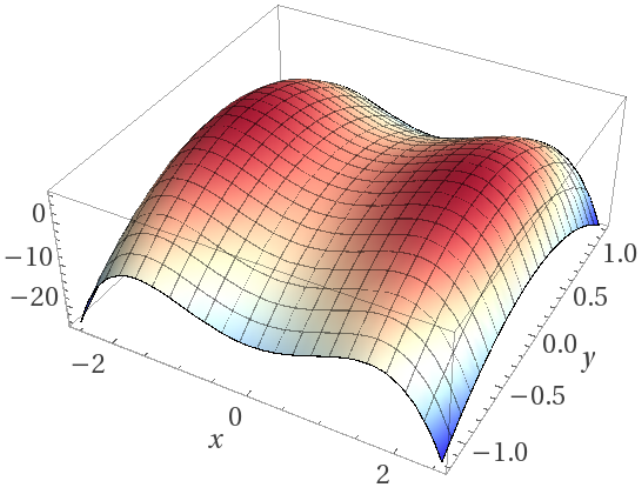

Figure 4.2: Plot of the constructed function: $-x^4 + 5x^2 - 10y^2$

The settings of the Bayesian optimization algorithm were as follows: acquisition function was chosen to be Expected Improvement with exploration/exploitation parameter xi set to 0.001 and 5 evaluations with random values for the variables were performed in the initialization stage. Figure 4.3 shows progress of the optimization algorithm.

4.1.2 Function with three unknown variables

To show the extensibility potential of the algorithm, the constructed function was extended by addition of hyperbolic tangent function of a third unknown variable making the final evaluation function:

$$-x^4 + 5x^2 - 10y^2 - tanh(z)$$

All three unknown variables have limited interval $[-200, 200]$ and the function reaches it's maximum in: $x = \pm 1.5811$; $y = 0$; $z = -200$.

Hyperbolic tangent function is plotted in figure 4.4 and with it's shape that resembles the sign function, it simulates the binary character of some parameters that were present in the Survival Shooter AI agents. Since there is very little difference in the function value in intervals $(-\infty, 0)$ and $(0, \infty)$ we only consider whether value of the

iter	score	x	y
1	-17571169.53	64.7	81.0
2	-406032797.01	142.0	-93.27
3	-373845338.41	-139.0	108.2
4	-19258588.80	66.05	156.2
5	-437825077.23	-144.6	-134.7
6	-20067010.81	66.84	116.5
7	-17711288.90	64.82	91.38
8	-19936159.00	66.67	143.1
9	-18915256.18	65.9	84.95
10	-7980223.06	53.05	87.48
11	-6255702.81	49.88	89.12
12	-4098626.06	44.78	94.17
13	-3991516.19	44.45	99.05
14	-3513596.18	43.04	96.33
15	-2415681.75	39.07	96.34
16	-1602118.51	35.09	96.44
17	-1143477.70	32.04	97.13
18	-909422.21	30.03	100.3
19	-362469126.08	138.0	-149.9
20	-830210.91	29.25	101.1
.			
.			
.			
88	-2.399	-0.2538	0.5212
89	-1604.61	-2.44	12.64
80	-834.0	-0.2814	9.134
81	-50.88	1.743	2.384
82	-1969.02	-3.203	13.84
83	-12.07	-0.0334	-1.099
84	-5.696	1.909	1.031
85	-8.445	-1.543	-1.212
86	-13.68	-1.249	-1.38
87	-1.531	1.581	0.8821
88	-16.5	-1.676	-1.505
89	-37.31	2.204	1.949
90	-36.17	-1.452	-2.056
91	-20.2	2.185	1.458
92	0.05745	0.1076	-0.005268
93	1.629	-0.7367	-0.2811

Figure 4.3: Progress of optimization process of two unknown variables.

optimized variable falls into the correct interval from the two.

The results in figure 4.5 show that the optimization algorithm was able to find values for parameter x and y that were within 10% of their real maximums again in about 100 iterations. The settings of the Bayesian optimization algorithm were the same as in the evaluation function with two unknown variables: acquisition function was chosen to be Expected Improvement with exploration/exploitation parameter xi set to 0.001 and 5 evaluations with random values for the variables were performed in the initialization stage.

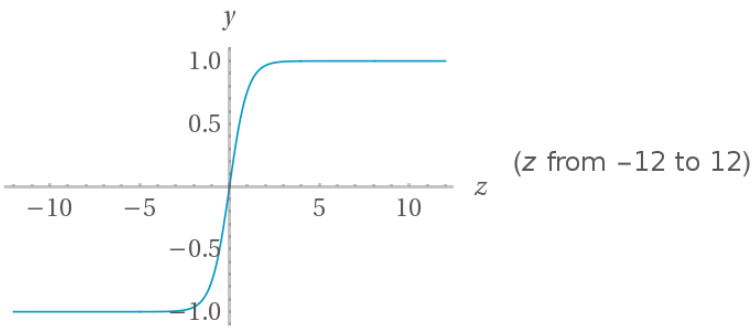

Figure 4.4: Plot of hyperbolic tangent function $tanh(z)$

iter	score	x	y	z
1	-414698501.75	142.7	-187.4	19.56
2	-59479391.22	-87.79	-109.9	-83.05
3	-121391946.97	105.0	88.23	127.3
4	-32386.64	-8.112	53.28	46.62
5	-385440100.09	140.1	-30.36	80.83
6	-141503751.23	109.0	148.9	-166.2
7	-8690937.27	-53.85	172.0	-163.9
8	-187554.47	-19.93	56.41	46.39
9	-67777.64	-13.35	60.75	43.48
10	-41716.09	-10.16	56.2	43.82
11	-94113.37	-15.5	61.35	55.25
12	-40438.29	-10.04	55.5	53.26
13	-62320.14	-13.73	52.66	48.7
14	-130164.13	-17.65	58.8	49.03
15	-37563.27	-6.228	60.21	48.67
.				
.				
.				
40	-23465.89	-0.16	-48.44	-95.05
41	-22406.32	7.492	-44.2	-94.94
42	-19297.85	2.974	-43.89	-92.41
43	-28137.60	1.055	-53.05	-92.78
44	-25599.49	1.621	-50.6	-92.25
45	-26654.25	-1.135	-51.63	-88.64
46	-4864559.67	46.48	-144.6	-160.4
47	-229203.75	8.735	-149.6	-86.16
48	-1374997370.48	192.6	-32.0	-74.48
49	-740687.41	26.83	-150.3	-141.7
50	-694074.93	24.96	-175.9	-51.25
.				
.				
.				
95	-70654.35	-11.24	74.36	55.16
96	-414696915.06	-142.7	-159.8	-43.26
97	-29334.35	-1.018	-54.17	-91.65
98	-20571.67	5.504	-44.5	-94.66
99	-3365.41	-0.6579	-18.35	-134.2
100	-8264.79	3.959	-28.46	-126.9
101	-3243.44	-1.849	-18.03	-130.9

Figure 4.5: Progress of optimization process of three unknown variables.

Chapter 5

Conclusion

The goal of this project was to explore the possibilities of using hyperparameter optimization techniques, namely Bayesian optimization, in technical fields where it is not common to see such tools implemented. For the testing environment, a simple game was chosen together with an AI development tool that showed promising properties for successful integration of an optimization algorithm.

As part of the project, an interface was implemented that enabled exporting of parameters of the UtilityAI agent's configuration and also updating these parameters when they were ran through optimization algorithm. The whole workflow is fully working and is mostly automated.

After extensive testing, modifications to the game and AI agents themselves, it was not possible to obtain definite results of the effectiveness of utilizing optimization techniques in applications such as artificial intelligence development for computer games. During this project I faced multiple hardware and third-party software problems that introduced variables which greatly influenced evaluation results and which I was unable to resolve. Additional tests were performed in controlled conditions that proved feasibility of the proposed approach if the application meets some conditions. Mainly, the application must exhibit consistent behavior that is directly dependent on the parameters to be optimized. It is also greatly beneficial if the application supports a mode of execution where only the application/game state would be simulated and could be sped up without compromising the correctness of execution (for computer games mainly disabling graphical output).

Future work could focus on exchanging the test environment to a different application or to closely collaborate with the developers of Unity game engine and Utility AI framework to mitigate problems that stand in the way of current workflow.

Bibliography

[1] Katherine Bailey. *Gaussian Processes for Dummies*. http://katbailey.github.io/post/gaussian-processes-for-dummies/. 2016.

[2] Eric Brochu, Vlad M. Cora, and Nando de Freitas. "A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning". In: (2010).

[3] Peter I. Frazier. "A Tutorial on Bayesian Optimization". In: (2018).

[4] Martin Krasser. *Bayesian optimization*. http://krasserm.github.io/2018/03/21/bayesian-optimization/. 2018.

[5] Fernando Nogueira. *Bayesian Optimization*. https://github.com/fmfn/BayesianOptimization. 2017.

[6] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. "Practical Bayesian Optimization of Machine Learning Algorithms". In: ().

[7] Apex Game Tools team. *Utility AI*. http://apexgametools.com/products/apex-utility-ai-2/. 2018.

[8] Unity Technologies team. *Unity*. https://unity3d.com/. 2018.

[9] James T. Wilson, Frank Hutter, and Marc Peter Deisenroth. "Maximizing acquisition functions for Bayesian optimization". In: (2018).