Teaching a CBR System to Play Super Mario

Barcelona, Spain

Martin Isaksson & Anton Kollmats
2017-01-23

Abstract

We have implemented a Case Based Reasoning system with the intent of teaching it to play a simple computer game where the objective is to avoid approaching enemies. The CBR system takes for input a list of enemy properties for a given game state and produce a jump intensity as output. However, the CBR system is written separately from the game and could be applied to any domain where the case description is a list of objects and the required output a single number. We use a tree structure for storing cases and we also use Gaussian Processes to find new solutions based on old ones. Experiments has been performed, and we found that this particular domain is not very suitable for our CBR system.

Table of Contents

Abstract 1
Introduction 3
Basic Principles of the CBR System 3
Chosen Application Domain 3
Requirement Analysis 3
The CBR Engine 4
Case Structure 4
Case Base Structure 5
Communication Between the Game and the CBR System 6
Retrieval 6
Adaption 7
Evaluation and Learning 8
Results 8
Discussion and Summary 9
References 12

Introduction

Artificial intelligence is a very popular subject since it makes machines learn by themselves, which can contribute to the technological development and also save the companies in the industry a lot of money. One type of an artificial intelligent system is a Case Based Reasoning (CBR) system. It learns how to achieve a goal by considering how the outcome was of a solution of earlier cases.

Basic Principles of the CBR System

A CBR system usually contains a case base (a library of cases structured in an efficient way), a clever way of retrieving the most similar cases from the case base, adapting the solutions of old cases to a better solution (after evaluating the usage of an old case to solve a new case) and to make the system to learn from this by itself.

There are several different ways of building a CBR system and its functionalities, since it is highly domain dependent.

In artificial intelligence and machine learning, the domain of computer games is an excellent domain to work with. This is because games can simulate the reality very well in many situations, but more important the user has access to all the inputs, outputs and all the other parameters. This makes it easy to artificially create different scenarios and test algorithms and systems.

Chosen Application Domain

A simple replica of the old and very popular 2D game Super Mario game has been built in this project and used as a domain of applying a CBR system on it. The goal to be achieved is for Mario to finish the game. This he will do with the help of a CBR system that considers the outcome of a handled situation in the game and reuse the good solutions to new similar cases.

The game and the CBR system has been built using Python. The story of the game is that Mario is supposed to avoid the enemies that he encounters by walking/running forward and jump with different height of the jumps to survive. There can be several enemies at a time, and all the enemies will have different size and speed, which makes it hard for Mario to know in advance how the next scenario will look like.

The more enemies that Mario avoids the longer he will stay alive and therefore it is desirable to pass as many enemies as possible.

Requirement Analysis

The CBR system needs to learn from earlier situations of the game to prevent bad solutions from occurring again in similar situations in the future. It has to be able to analyze the situation, e.g. noticing how many enemies that are in the frame, the

distance between Mario and the enemies and the velocity of the enemies. From this information Mario needs to make a jump that make him survive for at least 2 seconds after he has jumped. The jump height will be set by a jump intensity that the CBR system choses. The greater the jump intensity, the higher Mario will jump.

One primary objective for this is to make a flexible CBR system that does not know much about the domain, apart from some externally applied hyperparameters. The hyperparameters should not be domain specific, however.

The CBR system has to work in real time and the time response will at least have to be as fast as a human would react. The CBR system here will be faster than a human, so there isn't any really restrictions there.

To save memory it can be good to throw away cases that are useless or should not be used again. Another solution rather than throwing away a solution is to adapt the solution and make it better so that the case solution can be used again further on. To save memory, this CBR system will not add any cases if Mario is in the air or if there are no enemies in the frame. Else it will add a case every 0.5 second or when Mario Jumps.

Any ordinary computer should be able to use the CBR system, it is rather a question of a nice game-experience.

The CBR Engine

The CBR system takes a case containing five features per enemy in a frame as input. The features are *the enemy height*, *the enemy width*, *the enemy-distance in x-direction*, *the enemy-distance in y-direction* and *the velocity of the enemy*, all received from one frame. The CBR system store the cases in a tree with the solution of each case.

When the CBR system will use a solution of a similar case as the actual/present case, it will search the tree for those similar cases and retrieve a list of candidates. The similar cases are then further processed and some number of these (stored MAX_RELEVANT_CASES) are selected using k-Nearest Neighbors and the Euclidean distance[2]. Then regression using Gaussian Processes was used to calculate the new jump intensities[3].

The system evaluates the solution, i.e. if the jump was good or bad, by reporting back several times during an interval of five seconds and says if Mario has died or not. The further away from the implementation of the solution in time, the less impact the cause of death has on if the solution was bad or not. I.e. if Mario dies right after the solution has been implemented, the jump intensity (solution) is changed drastically and if Mario dies after e.g. five seconds the jump intensity (solution) is only tuned/changed a little.

Case Structure

For every game frame that is captured, the following information about each

enemy is fed to the CBR system:

- height,
- width,
- relative distance in horizontal and vertical directions and
- velocity in the horizontal direction.

For each enemy, this information is stored in a class called EnemyInfo, which itself is stored in a list. This list is wrapped into another class, called CaseEnvironment, which represents the game environment that describes a case. The CaseEnvironment class is stored in another class, called Case, which in addition contains its associated solution in form of a floating point number specifying the jump intensity fed back to the game.

Throughout the CBR, the information describing the cases is accessed in two ways, either by reading class variables by name, or by treating them as a list of raw numbers. The purpose of the CaseEnvironment is to provide the functionality of the latter, i.e enabling typecasting the CaseEnvironment into an array of floating points.

Case Base Structure

The case library consists of a tree structure containing instances of the Case class. The tree is structured in a layer-wise manner, where each layer refers to an attribute in a case. The tree can grow very big (see in Fig. 1 where each layer is connected to each branch in the layer above), depending on how many new kinds of cases the CBR system will encounter. Except the layer-wise structure the tree is, more importantly, structured after how many enemies in a frame that has attributes in a certain interval (see Fig. 1). The attributes used to create the tree are enemy height, width, horizontal and vertical distance and the intervals are stored as HEIGHT_SPECS, WIDTH_SPECS, XDIST_SPECS, YDIST_SPECS. These hyperparameters could be replaced arbitrarily, but in our case we defined them manually to specific variables in order to mimic human perception of distance. Normally, a human categorizes distance more fuzzy, instead of using the exact coordinates which were actually supplied to the CBR system.

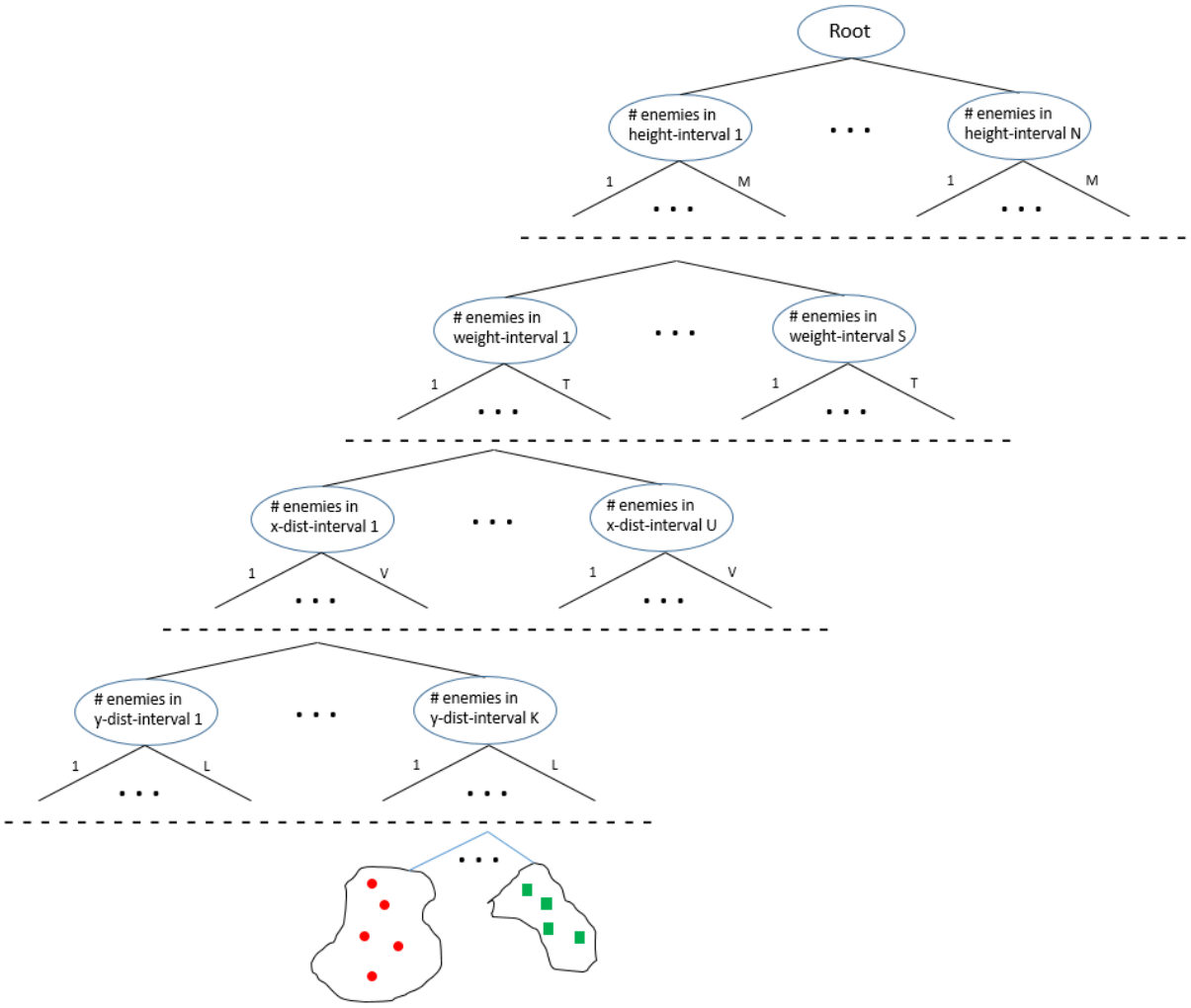

Figure 1 The structure of the case base. The Case-classes are sorted by certain attributes and stored in different leaves.

Communication Between the Game and the CBR System

Since the task of the CBR is to play a game in real-time, a communicator interface was constructed that pass information back and forth between the CBR and the game. Its mode of operation can be broken down into 4 steps:

1. Read the state of the game into an instance of CaseEnvironment
2. Pass the CaseEnvironment to the CBR system and wait for a response
3. Receive an instance of Case from the CBR system and execute its solution
4. Perform a delayed survival check (i.e if the game was lost or not) and pass the result back to the CBR system

Between steps 2 and 3, the CBR system goes through the retrieval and adaptation steps and after step 4, the evaluation and learning phases are gone through.

Retrieval

The retrieval phase is fed a CaseEnvironment class containing information about the current state of the game. Using this, it searches the Case Base for similar and

identical cases. If a perfect match is found, that case is returned. If, instead, no perfect match is found a list of similar cases are returned. If this list happens to be empty, a new solution with a zero jump intensity is returned. Otherwise, k-nearest neighbors is applied to all the variables in the case environment and the most similar ones are returned. The retrieval function is described in more detail by the following pseudo code.

```
function retrieve(input_case_environment):
# Find relevant cases and create/retrieve case with this input_case_environment
case_base, relevant_cases = search_tree(input_case_environment, case_base)
if new_case in relevant_cases:
        # Exact match found. Use that solution
        return matching_case
if len(relevant_cases) == 0:
        # No match found. Do nothing
        new_case.jump_intensity = 0
        return new_case
# Find the closest cases and their distances
relevant_cases, distances = k_nearest_neighbors(relevant_cases)
return new_case, relevant_cases, distances
```

Adaption

When the retrieval has returned a list of relevant cases (i.e similar, but not identical, cases were found) that solution is adapted in one of two ways. If only one similar case was found, the new jump intensity is selected by sampling a normal distribution specified by the single retrieved case. The mean value of the distribution is the jump intensity of the retrieved case. The standard deviation is calculated using

$$1 - e^{-\frac{d}{d_{scale}}^2},$$

where d is the distance between the found CaseEnvironment and the current CaseEnvironment in terms of the feature space. d_{scale} is a characteristic length specified before running the CBR (as SINGLE_CASE_DISTANCE_SCALE). In effect, this means that high distances (i.e low similarity) allow for more variation around the mean. Thus, for very low distances, the more similar the jump intensities.

Otherwise, if the number of relevant cases are greater than one, regression is used to fit a curve in feature space. The curve is fitted using an implementation of Gaussian Processes (GPs), using the features of the relevant cases as training data. GPs use the distance among all points in the training set to estimate a curve. For that reason, a characteristic length is supplied as a hyperparameter (as GP_LENGTH_SCALE).

The GPs return a distribution of functions that pass through the training points, specified by a mean value and a standard deviation. The greater the distance between training points, the greater the standard deviation. Effectively, this returns a normal distribution around the mean for every point in feature space, which is sampled to generate the new solution. Should the sampled value exceed the allowed intensities (a number between 0 and 1), clipping is done. Also, it is possible that great oscillations in the GPs yield infinities for the mean and standard deviation and in that case, we simply assign them the values of 0.5.

Evaluation and Learning

Once the CBR system has recommended a case to be used, the CBR receives feedback. The feedback is simply whether the game was lost or not, i.e if the player is still alive.

If the case is new, i.e it has never been tested before, it is simply removed from the case base. If the case in question is an old case, i.e its solution has been used successfully before, resulting in death, the solution is modified slightly. Once again, a normal distribution is created using the old jump intensity as a mean value and using some specified standard deviation (MODIFICATION_STD).

Results

One requirement for the CBR was to make the playing experience look smooth while using the CBR. The tests were performed, in this project, on a 1.6 GHz Intel Core i5 processor, memory 8 GB, 1600 MHz DDR3, Intel HD Graphics 6000, 1536 MB, and a 1.8 GHz Intel Core i5-3337U processor, memory 8 GB, Intel HD Graphics 4000. The system ran smoothly on the first system, while on the second system the experience was barely above acceptable.

As a measure of success we have counted the number of enemies successfully avoided and the number of games played. The specifications used are presented in Table 1.

Parameter	Values
MAX_RELEVANT_CASES	20
SINGLE_CASE_DISTANCE_SCALE	10
GP_LENGTH_SCALE	100
MODIFICATION_STD	0.1
HEIGHT_SPECS	(0,70),(70,1000)
WIDTH_SPECS	(0,70),(70,1000)
XDIST_SPECS	(0,100),(100,200),(200,400)
YDIST_SPECS	(0,100),(100,200),(200,400)

Table 1 Parameters used for the CBR.

Using them, we obtained the results shown in Fig. 2.

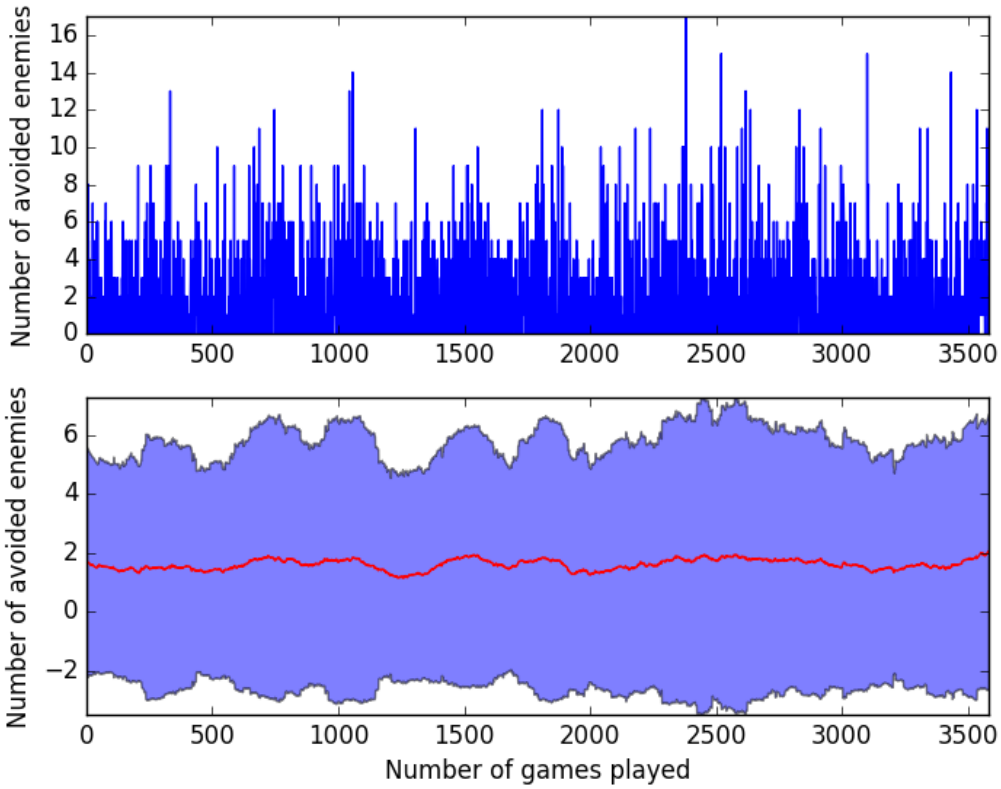

Figure 2 Number of avoided enemies as a function of the number of games played. The top panel shows the raw numbers, while the bottom panel shows the moving mean value (red curve) and two standard deviations (shaded area).

Discussion and Summary

As can be seen in Fig. 2, the results are not very satisfactory. We see virtually no increase in performance regardless of the number of games played. We do not believe that this is entirely due to implementation flaws of the CBR itself, but rather a flaw in which information is obtained from this particular domain, especially in the case of timing. To elaborate, we extract a snapshot of the game and we use only the information of that instant to make decisions. As such, the CBR does not care what has happened before, or more importantly what will happen in the next instant; in order to survive, it is forced to make a decision to jump in order to survive, even if it could be more beneficial to wait and jump later.

This is related to the fact that past decisions affect future case environment. That is, a decision to jump in one instance could land us very close to another enemy which had not yet appeared on the screen. For this reason, it is difficult to evaluate solutions, since it could be a combination of them that lead to a collision.

With that said, the implementation itself probably contribute to the results. Since we use hard boundaries in the tree, similar cases could be split up (imagine presenting one case with one attribute value 199, another with 201 and the boundary is set to 200. These two cases would not be considered similar). This is especially devastating to our domain since, in an ideal case, the difference between an environment to jump or not is very fine (jumping when the enemy is 10 pixels away might be beneficial, while jumping when the enemy is 15 pixels away might be a bad decision).

Apart from using the tree structure, several approaches were attempted before we ended up with the particular implementation presented. For instance, in early approaches we simply used the k-NN algorithm to find the closest cases and took a mean value. However, using only the distance gave us misleading results (e.g, the case of height = 10, width = 15 and the case height = 15, width = 10 would give the same distance). For this reason, we tried using the DBScan algorithm to cluster cases and then take the mean value. This too proved to be an unsuccessful approach, which we noticed by very poor results.

Our final approach relies on using regression, which seemed to improve the results slightly. However, the results are far from satisfactory and seem to be very random. In general, as pointed out before, this seem to be due to the definition of our cases. It would probably be more beneficial to have much fewer cases, e.g as suggested before using sequences of frames and treating them as time series and possibly predicting future behavior. On the other hand, this would require more domain specific knowledge and tougher constraints on the CBR system, which we designed to be very general.

In this regard, we obtained our results. Apart from the tree structure, which requires some specific tuning of intervals, our CBR system is very general. The case environment is very flexible, the EnemyInfo class can simply be replaced with any class containing some information. The return value of the CBR, of course, is constrained to be a number between 0 and 1, and the evaluation is restricted to be a boolean value designating a successful solution or not. As for the tree structure, it requires that the cases contain attributes as floating point values for which an index is assigned. For every attribute to be used by the tree, at least two intervals must be manually supplied.

Apart from that, the CBR system make no assumptions about the domain. One interesting outlook would be to evaluate this CBR system for other domains where one solution affects the outcome of other cases, which is probably the main issue with applying our CBR system to our chosen domain.

Another way to encounter the problem of teaching a system to play a computer game, is to use CNNs (Convolutional Neural Networks) with reinforcement learning. This is also a way for the system to learn by experience, but in a way that is much more similar to how the human brain works. The company DeepMind is known for their way of using just CNNs with reinforcement learning to learn a system to play a

game. These type of systems are much more complex than the CBR system we have implemented and it takes a lot of computational power to train these networks (compared to what it takes for our CBR system). Though, this kind of system is necessary to play a computer game that is not too simple[4].

References

1. Sànchez-Marrè, Miquel. *PRINCIPLES OF CASE-BASED REASONING.* Barcelona, Universitat Politècnica de Catalunya.
2. Béjar, Javier. *Unsupervised Machine Learning and Data Mining.* Barcelona, Universitat Politècnica de Catalunya.
3. Rasmussen, Carl Edward., and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. Cambridge, MA: MIT, 2006. Print.
4. ”Wikipedia,” 6 January 2017. [Online]. Available: https://en.wikipedia.org/wiki/DeepMind. [Used January 2017]