Experiments with Learning for NPCs in 2D shooter

Anonymous Author(s)
Affiliation
Address
email

Abstract

Machine learning for modeling the behavioral and cognitive activity of non-player characters (**NPCs**) in video games is a promising field. While most of the mainstream and successful games in the game market mainly rely on an 'illusion' of AI with no learning at all, they get away with better graphics and other promising game features. Furthermore, the character behavior is usually a static rule based scripting that maps states and actions. Games that are built on such static scripts are unable to hold on to the interest of the game player as one eventually finds a loop hole and exploits the same. Dynamic scripting is one way to incorporate dynamically changing features in a game. But even that would need the developer to foresee each and every aspect while script creation and would exhibit some kind of repetitiveness in NPC behaviors. In this paper, a simple 2D shooter scenario has been used as an example to model the NPCs that learn from the player's game playing techniques.

1 Introduction

World video game market is expected to exceed $ 61.9 Billion by 2012, according to a new report by Global Industry Analysts, Inc. The consumer base ranges from children to young adults as well as grown-ups. Shooter games (first person/third person/others) make up for a considerably large cut of the pie. One of the most compelling yet least exploited technologies in games these days is machine learning. Hence, there is still a vast window of opportunity to make video games even more interesting from the player point of view, if these techniques are used correctly.

As was stated before, the action performed by non-playing characters are usually determined by the underlying game AI. A point to note here, is that in this paper the term 'AI' is used in it's academic sense and not in the 'game industry' sense. In the latter sense, AI has a broader meaning, which encompasses techniques like path-finding, nearest neighbor etc. Programming AI for NPCs in shooter games is a problematic task because of two main reasons. First, the developer has to come up with all the possibilities and states of the game the character might encounter. Based on numerous combinations of those, one has to formulate rules for subsequent states and actions. The planning doesn't stop here as different actions might lead into different states and thus the decision process gets more complex. Add to these the fact that games will only get complicated with time to attract more audience. Second, since all possible elements of the character's response is frozen before the game is shipped, the gameplay will have a limited number of 'elements of surprises' for the player and soon will exhibit repetitiveness both in actions and general behavior. This is highly likely with most of the games, and is bound to happen sooner than later as the player gains experience.

Applying ML to games, by no means is a recent technique. A technique similar to *temporal difference learning* for checkers was first employed by Samuel back in 1959. Since then, ML techniques have been applied to different types of computer games ranging from board games to high-tech graphics based video games. From the player perspective, there can be two forms of learning in shooter based games, *out-game learning* (OGL), where everything is learned offline and learning stops once the game is shipped and *in-game learning* (IGL), which as the name suggests has the game characters learning adapt to the gameplay. OGL can again be broadly classified into two types based on how and when exactly the learning happens. The first type is when a game has two modes of play, a normal 'play' mode, which as the name suggests is the normal gameplay and the other is a training mode where one trains the NPCs according to the behavior desired in the gameplay. The other type is usually based on evolutionary methods where, agents/ NPCs are trained (evolved) as the game is played. The player may or may not specify the kind of evolution that is desired. In this paper, the first type of OGL is considered and few machine learning techniques are experimented with, in a simple two dimensional shooter game.

The rest of the paper is organised as follows.Section 2 deals with the details of feature selection, game rules and the machine learning tools and techniques employed in this paper. Details of the implementation of the paper is explained in 3. In section 4, the results are presented and related explanations are given. A summary of the task as well as the conclusion is done in section 5. Prospective for future work are highlighted and discussed in section 6. References are listed down in section 7 .

2 Overview of Learning Method

In the presented work, the behavior of the NPCs are modeled using Artificial Neural Networks. The main idea is to use appropriate features selected from the data recorded from real persons' gameplay to train and model the behavior of the computer BOT. The learning in the ANNs has been implemented using the backpropagation algorithm, explained in section 2.3. The next subsection, 2.1 highlights the motivation behind going for collecting data from one's gameplay instead of coming up with an optimization or cognitive model for the character. Then section 2.2, deals with the general rules and selection of features to train the network for getting meaningful outcomes.

2.1 Motivation

A very quick answer to what all factors come into play when modeling the behavior for an NPC, is readily available once it is looked at from a human point of view. Let us list down some of the pertinent 'state' based questions that often get considered when one is playing a simple shooter game of some sort. It is assumed that boundary and other complicated environmental factors are not taken into account.

- Which opponents are shooting and who are not?
- How far is the missile/firearm that has been launched at me?
- Which opponents have me in their view/ have a clear shot ?

And, given the answer to the above questions, the task is now to answer the following 'action' based questions.

- When to moving towards and when to move away from an opponent?
- which opponent to attack first?

2.2 Rules and Feature Selection

In the presented paper, a simple 2D shooter game is created for consideration. To emphasize more on the use of machine learning and to avoid other complicacies, a game from scratch was preferred over working on an already established source code. A screenshot of the simulation is shown in figure 1.The goal, to win a round of the game is to either survive for a specific duration of time or kill all the opponents, whichever done first. For simplicity, the characters (now onwards referred to as 'shooter'), can only move in 4 directions i,e, along the positive and negative; x and y axes. The shooters are equipped with missiles which they can fire one at a time per target.

The problem at hand is slightly different from conventional machine learning examples in the sense that there is complete freedom in selecting the dataset for training the model. A naive approach is to randomly select as many examples from the state space of the game. This however is wasteful as

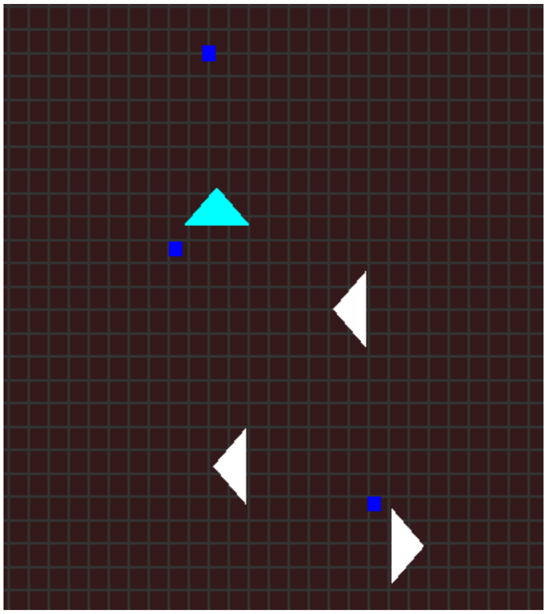

Figure 1: A screen from the game, the sky blue shooter is trained by a neural network

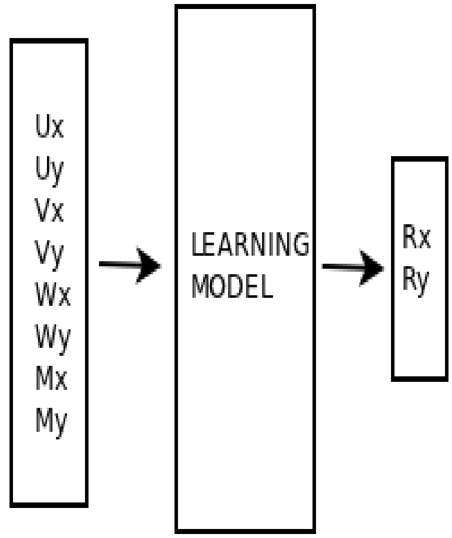

Figure 2: The overall model structure, U, V, W and M represent the nearest 3 neighbors and the missiles' location respectively. The network outputs the velocity R of the shooter.

the interest is generally on few key states e.g how a human controlled shooter evades the missiles that are shot at him. Also, it is quite likely that if random garbage data is fed to the model, it will ultimately learn very little of the main objective. Hence it becomes utmost important to record data at specific moments only. A little tuning goes into selecting the optimum amount of examples i.e. not too low for the model to learn little to nothing and neither too many for it to over-learn. Having said that, it is also vital to not push the data sampling rate too high, as then it becomes difficult to collect enough data. In present work, data is sampled at every $N = 10$ gameticks.

Selection of appropriate features is always more vital to obtaining better results than selection of the training method or the quantity of the dataset. To model the movement of the shooter, it's three nearest neighbors and the closest missile that has been targeted at him are taken into account. All the data recorded are converted to the frame of reference of the shooter. That is rather than considering the origin to be at some fixed point in the space, the world is transformed such that the origin is with respect to the shooter. The model is designed to output the velocity of the shooter given the inputs described above. In the training phase, one of the shooters is controlled by a human being and others are programmed to target it. This although is unlikely in a real game scenario (since others will not 'always' target a particular shooter) but nonetheless, the shooter is trained to evade worst case scenarios. Ideally, when part of a large scale video game, the user would have full control on the way the training phase is carried out.

2.3 ANN with Backpropagation for navigation

In this work, a single hidden layer neural network is used to model the dependency of the navigation (i.e. movement in the arena) of the shooter, given the features as inputs. Neural networks are chosen to learn the dependency of the aforementioned features on the shooter's velocity because they are both compact and computationally efficient in the run time, making realtime outcome of results possible. The training is modeled as a *supervised classification problem*, in which given the input, the task is to assign one of the four possible velocity directions to the output. The objective function is formulated as a negative log likelihood function, as shown in equation-1 and the weights and biases at various layers are updated by minimizing it over the training dataset with respect to respective variables.

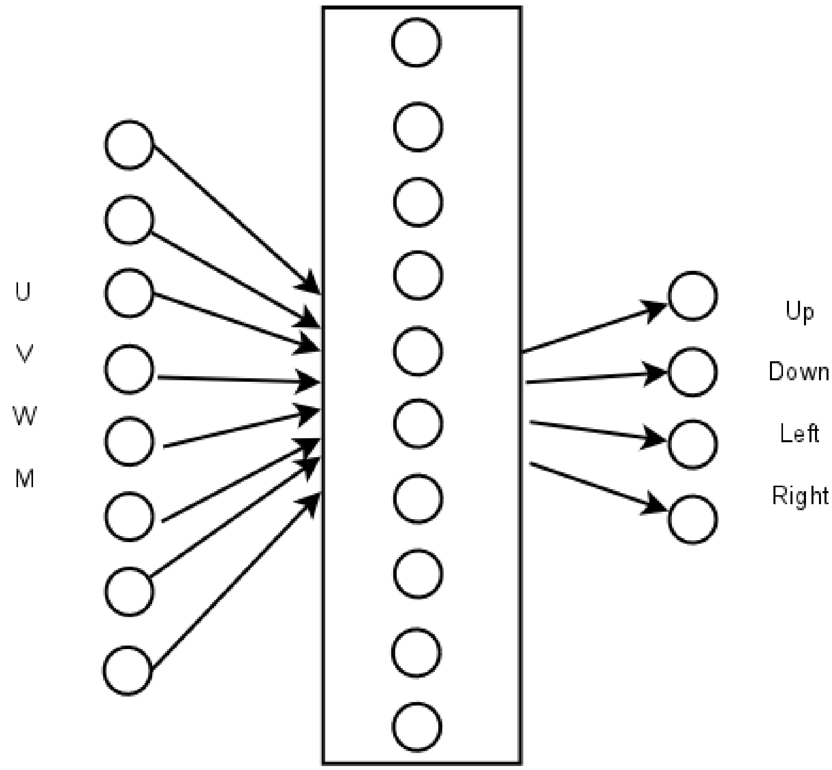

Figure 3: The detailed neural network architecture. The input is a 8 dimensional vector and the output, a 4 dimensional one. One hidden unit with 8 neurons are used.

The ANN architecture selected in the presented work is very simple and intuitive. The input and the output layer has $n_i = 4$ and $d = 8$ neurons respectively, corresponding to the input and the output vectors shown in figure 3. Note that, the input variables are 2D vectors and the outputs are essentially class labels, which in this case represent the four possible direction of movement. The hidden layer consists of $n_h = 10$ neurons. Adding more neurons to the hidden layer or adding an extra layer to the network provided little to no benefit and unnecessarily increased the computational load.

$$L = -\sum_{i=1}^{n}\sum_{j=1}^{d} t_{ij} log(p_{ij}) \tag{1}$$

In the likelihood equation 1, n represents the number of examples available to train the model, $d = 4$ is the number of neurons in the output layer, corresponding to the four possible direction of motion as mentioned before in section 2.2. Hyperbolic tangent activation function *tanh* is used to map the output of the hidden layer to the $(-1, 1)$ range. Softmax functions (equation - 2) are used to force the output between $(0, 1)$ so as they can be interpreted as probabilities. t_{ij} is an indicator which tells us which output j, input i maps into. Thus it is 1 if input vector i belongs to class j, else it is 0. An expression to yield the error, given a batch of data X, y to train the network, is shown in equation 3.

$$p_j = \frac{o_j}{\sum_{k=1}^{d} exp(o_k)} \tag{2}$$

$$e_{batch} = \frac{\sum_{i=1}^{n}(1 - t_{i,k_i})}{n} \text{ }, where\ k_i = \max_j p_{ij} \tag{3}$$

3 Implementation

The system is implemented in python. External packages numpy and pickle are used for computations and IO. The graphics aspect of the game uses the pyopenGL package and pyopenGL.GLUT as the window manager. The neural network used for training the shooter is the one that was implemented in assignment 5 of the course.

4 Results

According to the experiments carried out, about 5000 samples are enough to train a shooter to navigate accurately through a arena containing atmost 4 other shooters that are programmed only to shoot at the former (i.e. the current location of the shooter is made available to others). However, if other shooters are not explicitly given the location of their target and instead are forced to 'watchout', the learned shooter is capable to survive longer and tackle more opponents. It takes about 100 epochs of the training data, for the validation error to stabilize at a value of $E = 0.1$. A sample navigation run of the learned shooter is demonstrated in this video. The execution, as mentioned before, occurs is in realtime.

5 Conclusion

A simple 2D game implementing a neural network to model the navigation behavior of a NPC is presented. The dataset for training and validating the network was recorded during the gameplay of a human player. The network is then used to obtain the best possible move for a NPC given a particular game situation (which in this case is a vector consisting of the nearest three opponents' and the approaching firearm's relative position. A clear advantage of using neural networks for doing this task, over any other state, action and reward based decision model, is that the later is highly likely to be computationally more expensive to be evaluated in realtime. ANNs on the other hand break down an otherwise entangled decision tree into a much simpler mathematical equation containing of weighted summations and easy function operations.

6 Further Work

The current work has lots of scope to be worked upon and improved. Infact, this work creates a basic framework and opens up a gate for future possible findings in realms of adaptation in gameplay of survival and/or shooter based games. The work can be extended by considering the situation in 3D. More freedom in terms of movement may be included so as the problem could then be tackled as a higher dimensional regression problem instead of a classification one. Also, the fact that almost any other real life counterpart can be added to a video game, broadens the scope for instilling intelligence and automated learning into it. A simple example would be making the NPCs capable to experiment on its own and learn how to use it's surroundings to pose new threats to the player, that might bring in elements of surprise and fresh interest. This in turn would boost the longevity of the game and directly impact revenue associated with the production house of the game.

7 References

[1] John E. Laird & Michael van Lent (2005) Machine Learning for Computer Games, *Game Developers Conference* , GDC-2005

[2] Jonathan Dinerstein, Parris K. Egbert, Hugo de Garis and Nelson Dinerstein (2004) Fast and learnable behavioral and cognitive modeling for virtual character animation. *Computer Animation and Virtual Worlds*15: 95-108 (DOI: 10.1002/cav.8)

[3] Ken Mott (2009) Evolution of Artificial Intelligence In Video Games: A Survey *Survey paper*

[4] Samuel A,L. (1967), Some Studies in Machine Learning. Using the Game of Checkers. II - recent progress, *IBM Journal*

[5] Aaron Hertzmann (2003), Machine Learning for Computer Graphics: A Manifesto and Tutorial

[6] Kenneth O. Stanley, Bobby D. Bryant & Risto Miikkulainen (2005), real-Time Neuroevlution in the NERO Video Game, *IEEE Transactions on Evolutionary Computation* Vol. 9, NO. 6

[7] Pieter Spronck, Marc Ponsen, Ida Sprinkhuizen-Kuyper (2006) & Eric Postma, Adaptive Game AI with Dynamic Scripting, *Journal of Machine Learning*, Vol. 63, Issue 3