A Dynamic Balanced Level Generator for Video Games based on Deep Convolutional Generative Adversarial Networks

Morteza Rajabi, Mehrdad Ashtiani1,* Behrouz Minaei Bidgoli and Omid Davoodi

Computer Games Research Laboratory, School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran

E-mail: morteza_rajabi@alumni.iust.ac.ir, m_ashtiani@iust.ac.ir, b_minaei@iust.ac.ir and omiddavoudi@cmail.carleton.ca

Abstract

In the gaming industry, creating well-balanced games is one of the major challenges developers are currently facing. Balance in games has different meanings depending on the game type. But, most of the existing definitions are esteemed from the flow theory. In this research, we have used generative adversarial networks (GANs) to automatically create balanced levels. In the proposed work, a level of a 2D platformer game is considered as a picture and is fed to the network. The levels are randomly created while adhering to a set of balance requirements. Those levels that can be solved with the help of an agent using reinforcement learning in the number of tries set by designers are given as input data to the network. Finally, the network automatically generates new balanced levels and then, the levels are checked to see if they have the game’s minimum necessary requirements and also to check if they can be solved by the reinforcement learning agent. The best performing network is then selected for the level generation. In the series of performed evaluations, it is shown that after the training process, the proposed approach is capable of generating levels that are well-balanced with considerable accuracy.

Keywords: Generative adversarial networks, dynamic difficulty adjustment, reinforcement learning, video games, game balance.

1. Introduction

In the game development process and especially game design, balance is one of the most difficult and time-consuming activities [1]. As Ernest Adams in his well-known book fundamentals of game design has stated: “To be enjoyable, a game must be balanced well—it must be neither too easy nor too hard, and it must feel fair, both to players competing against each other and to the individual player on his own.”[2] This definition is inspired by the flow theory introduced by Mihaly Csikszentmihalyi a famous psychologist [3]. He hypothesized that a person’s skill and the difficulty of a task interact with a result of cognitive and emotional states. Due to the diversity of video game styles, balance in games is defined in different ways. For example in turn-based games, designers tend to balance the game in order to make fairness in the win-rate [4] or in other games, designers tend to adjust the difficulty of passing levels and balance them dynamically [5, 6, 7]. In strategy games, designers try to find the unbalanced behaviors of the players to prevent the formation of dominant winning strategies [8]. In general and as a widely used definition, the meaning of balance in a video game is that playing the game and passing the levels by the player does not go beyond the framework of what the game designer has imagined or intended to happen [9, 10].

1 Address: School of Computer Engineering, Iran University of Science and Technology, Hengam St., Resalat Sq., Tehran, Iran, Postal Code: 16846-13114, Tel/Fax: +98-21-73021480, Mobile Phone: +989126304045.
Recently, a lot of research has been performed on creating balance in computer games. Bosc et al. [8], Karavolos et al. [11], Olesen et al. [12], Morason et al. [13], Bangay et al. [14] and Uriarte et al. [15] have all tried to address this issue using different approaches. The researches that are based on manual human-based decision making have been more accurate compared to the automatic balanced level generators. However, being time-consuming and error-prone encourages the researchers to explore other methods in which an automated intelligent algorithm plays an essential role. These methods are often faster than the manual level creation process in detecting and fixing bugs in the created levels. But, still the mistakes may occur when the human expert wants to choose the proper parameters and settings for the level generation algorithm. Thus, in the end, the human-errors may manifest themselves in another form, impacting the balance of the created level. In this paper, we propose a method by leveraging the power of recently introduced deep convolutional generative adversarial networks (GAN) [16]. Currently, deep convolutional GANs are gaining significant success in the field of image processing. The general similarities and the potential mapping between these two domains motivated us to use these networks for the automatic generation of balanced game levels.

To this aim, we propose a method for balancing video games using deep convolutional GANs in which the map of the game, as well as the corresponding parameters, will be automatically determined without any human intervention. The way these networks work in the field of image processing is that they are fed a number of images as input. Then, after the training steps completed, they generate new images (i.e. not existing in the training input set) that are similar to the input images. Of course, GANs have found many other useful application domains as well. For example, GANs have been successfully applied in handwritten digits generation [17], human face creation [18], and even creating bedroom layouts [19]. GANs have also been employed in the field of video and sound analysis such as frame prediction in videos [20, 21] and sound generation [22].

To obtain the initial data, a basic level generator was implemented that randomly generated the levels without considering whether the level is going to be solvable or not. To effectively use the GAN, one needs a huge amount of input data. If a human expert was assigned to determine whether the game levels are solvable or not in a manual process, it would take a very long time to do so, making the approach almost impractical. Therefore, we have used reinforcement learning approaches [23, 24] to simulate the human play process. Then, after performing the repetitions, a large number of solvable levels are generated and given as input to the GAN. Subsequently, the network is prepared to build levels that have well-known features by specifying and changing various parameters. As a summary, the followings are the major contributions of the proposed approach:

1. Providing a metric for evaluating different procedural level generation approaches and especially GAN-based approaches. To the best of our knowledge, there is no such metric and evaluation mechanism in the literature yet and usually, the evaluation is performed by a human operator using a manual evaluation process.
2. Introducing an approach for random balanced level generation using deep convolutional GANs. The solvability of the level is determined using a reinforcement learning agent allowing the level designer to define the requirements for a balanced level and automatically generate such levels without any human involvement.
3. Leveraging the power of reinforcement learning approaches to determine the level difficulty and the adherence of the created levels to the balance requirements specified by the level designer.

In the remainder of this paper and in Section 2, the related work of this research will be
reviewed. In Section 3, the background knowledge required for the proposed approach is given. In
Section 4, the proposed model for automatic balanced level generation using GANs will be
introduced. In Section 5, the results of the evaluation of the proposed method are given. Finally,
the paper concludes in Section 6.

2. Related work

A variety of research has been performed in the past years on automatic game balancing and
dynamic difficulty adjustment. In this section, we will review the related work of our research and
analyze their strengths and weaknesses.

In one of the first attempts to use GANs in the context of game level generation, Volz et al have
used the Mario game levels [25] to investigate the possibility of using GANs for generating new
levels [26]. For doing this, the authors have created pre-designed levels and formatted them in
28*14 windows and have fed them to the GAN. Then, the generative network will output the new
levels. After the creation of the new levels, the authors have used an A* agent used in the Mario
AI Competition in 2009 to determine the solvability of the levels. In this research, latent variable
evolution [27] is used to investigate how latent GAN vectors can be evolved through a fitness-
based approach in the context of level generation. Also, CMA-ES [28] strategy is used to evolve
the latent vector.

In another work, Shaker et al. in their book have aggregated a lot of recent approaches to
procedural content generation in computer games [29]. Different approaches such as search-based
techniques [30], L-systems [31], constructive generation [32] and Fractal methods [33] are
investigated in a wide range of game genres such as card-games and 2D platformer games. Besides
the numerous advantages that each of these approaches provides, a common downside is the role
of human operators to hard-code the knowledge of content creation for every specific context. This
is indeed not the case in our approach where domain knowledge is created using a reinforcement
learning approach.

Similarly, Perez et al. researched the mechanisms to create balanced AI in video games [5]. The
challenge they are tackling is to create a compelling experience for the professional as well as
novice players of a 2D platformer game. By a compelling experience, the authors mean that the
new and novice players should not leave the game because of its high difficulty and the
professional ones because of its simplicity. The method they have used in their study is to
dynamically change the parameters of the game such as player's speed, number of obstacles and
so on. These changes will dynamically occur according to how the player plays the game. In this
study, a runner 2D platformer game was designed. Then, different game parameters such as speed,
type, and cycle of obstacles have been linked together with the help of evolutionary fuzzy cognitive
maps (FCM)[34, 35]. In this research, the parameters are manually linked together by the authors
in a pre-determined way. The simplicity and soundness of the proposed approach aside, the
extraction of the influential parameters in the level and determining their impact on the difficulty
of the game is a challenging task for the level designer and is prone to a lot of human errors. In
addition, creating the predetermined influence factors and setting the correct coefficient degree for
them is not a simple job and requires an extensive amount of try and error by the human operator.

In another interesting work, Morosan et al. used the class of evolutionary algorithms [36, 37]
in order to find the values of the parameters leading to game balance [38]. This algorithm is a
subclass of the genetic algorithms [39]. In this work, by defining the fitness function [40], (a
function of player's winning rate and the amount of difference between the new parameters and
the old ones) the parameters of the game will dynamically change to reach the balance the game
designer has desired. In this research, the game designer first chooses the expected winning rate of the level. The Pacman game is tested as a sample. The game was simulated for a large number of repetitions using the evolutionary genetic algorithms. The winning rate value of the new parameters compared with the initial parameters was recorded. According to the balance function formula, the experimental values that had a smaller difference with the initial values, as well as a winning rate that was nearly expected, are selected as the new balanced parameters. After the successful results in the Pacman game, these experiments were also examined in the Star Wars game and achieved similarly satisfying results. The use of the genetic algorithm in this study causes balanced values to be as close as possible to the values that the designer expects. However, the selection of parameters by humans may cause a lot of errors. For example, a parameter that does not have a significant effect on the balance of the game may mistakenly affect the evolution process and create a significant impact on the accuracy of the newly modified parameters.

Xia et al. have also researched the area of game balance using the procedural content generation technique [41, 42]. They have used this method on a game designed with the Unity3D engine to evaluate their research. The game is a two-dimensional shooting game in which the player is attacked by several enemy types. When the player starts the game their Mana is 0. Therefore, the Mana left to the player at the end of each turn (which can be positive or negative) is considered as an offset value. The aim of this research is to find a way in which at the end of each turn, the player ends up with zero Mana points. This is performed with the help of the PSO [43] and dynamic behavior-changing [34] techniques. Although the proposed approach is successful in creating balance in a well-structured way, the approach is time-consuming to implement and learn.

In another research line, Paulo Silva et al. investigated the dynamic difficulty adjustment based on the players’ behavior type [6]. The DOTA game has been used for this research. As the first step, a general gameplay pattern of different players is extracted. From this pattern, it is pointed out that the complexity of the game for novice players causes them to leave the game. Also, the simplicity of the game for professional players makes them get bored quickly. Therefore, the authors have tried to set up a strategy that adjusts the difficulty level of the game according to each individual player. For example, if the player is a beginner, the difficulty level of AI would decrease to some extent so that the player would be either winning or getting defeated in a balanced manner. To evaluate the performance of this research, a number of features have been extracted from the game and then a relationship for evaluation has been introduced. It is shown that the approach can flexibly change the difficulty level of the game based on the players’ behavior. Of course, creating multiple AI scenarios and triggering each of them based on a certain condition requires extensive and accurate AI design.

In the very limited researches where deep neural networks have been used, Karavolos et al. have used this method to balance the levels and parameters of a first-person shooter (FPS) game [11]. Adjustable parameters in this study are gameplay maps as well as the game's weapon parameters [44, 45]. The idea behind this research is to use deep neural networks to detect balanced levels. For doing so, the game designer determines the map of a level along with the parameters of the weapons. Then, the neural network will comment on whether it is balanced or not. A strength of this study is that, after the learning stage and with a high speed, the network recognizes the designed levels and tags them as balanced or unbalanced. But, it still requires a human operator in several stages in the process (i.e. feature extraction, network parameter design and so on).

From the very few researches that have applied basic GAN as their method of level generation, Giacomello et al. have used such networks to generate procedural content [46]. The required data for the neural network are obtained through human-made levels and stored as WAD file. WAD
format files include all the level’s information. In order to evaluate the results, the authors used the SLAM algorithm [47]. But, by reviewing the body of research performed in the field of GANs, it can easily be inferred that most of these studies are conducted with a much larger number of input data compared to this research. So it seems that the volume of data in this study, due to the size of the images and the number of features is forcibly limited to a very low amount.

In another research, Morason et al. have investigated balancing in the PacMan game [13]. Designers have a set of goals and rules in their minds which they want to observe in the game. Because of that, a useful tool for game designers has been presented to be used in order to turn the designers’ vision of a game into reality as accurately and easily as possible. Neural networks and genetic algorithms have been used in order to generate agents of various skills and this is particularly valuable for cases where no pre-existing agent exists.

Bangay et al. have researched about achieving balance in a large real-time strategy game [14]. They have proposed an attribute space representation as a common framework for reasoning about balance in the combat scenarios found in these games. The typical attributes of range, speed, health, and damage have been used in this work. For measuring balance in this type of game, authors have paid attention to the health of each unit at the end of the game and they have calculated the difference between each unit's health \( h_1, h_2 \). If \( h_1 - h_2 \) is positive it means that team 1 wins, if it is negative it means that team 2 wins and if it is 0 it means the game is balanced. The proposed model can predict the win probability and it helps to improve the game balance. This framework is a suitable tool for identifying the unbalanced situation in the RTS game, but it is created only for RTS games and the definition of balance in this work is too limited (if the difference between unit health becomes zero it means that game is balanced).

Finally, Uriarte et al. have also investigated about generating balanced maps for StarCraft [15]. PSMAGE has been presented in order to generate balanced maps for the popular real-time strategy (RTS) game StarCraft. This approach has used Voronoi diagrams to generate an initial map layout and after that, it has assigned different properties to each of the regions in the diagram. This approach generates the map procedurally. After generating maps, PSMAGE uses a collection of evaluation metrics, in order to measure how balanced a map is. Balanced Map in this work means refers to maps having two conditions. First, if all the players have the same skill level, they should have the same chances of winning the game at the end. Second, in the case of StarCraft, no race should have a significant advantage over the others. In Table 1, the approaches are summarized and compared.

3. Background knowledge
This section reviews the basic concepts of GANs and reinforcement learning algorithms. In our proposed approach, GANs have been used to create new levels, using data derived from reinforcement learning and simulation methods. In this section, we give a brief overview of these two approaches.

3.1. Generative adversarial networks
The generative adversarial networks are first introduced by Goodfellow et al. in 2014 [16]. These networks are considered to be a subclass of deep learning algorithms. In fact, they are a neural network composed of two main components, namely the generator and the discriminator:

1. **Generator**: The generator is fed real numbers as input. The output of the generator which is equal to the size of the input dataset is the data that the generator tries to generate with the
characteristics of the input dataset.

2. **Discriminator**: The task of the discriminator is to identify fake data. The input of this network is the data from the real data set (marked with a real tag) as well as the output of the generator (marked with a fake tag). The purpose of the discriminator is to recognize these labels appropriately and separate real and fake data.

These two components complement one another to build the data correctly and with the properties of the input data set. The function of this neural network is as follows: First, the generator produces a given number of random noise data as input and sends them as output to the discriminator using the forward propagation technique. The discriminator receives the generator’s data and with forward propagation, calculates the network’s output. The generator keeps the weights fixed while calculating the error and using the backward propagation technique, the weights of the discriminator are modified with the aim of better diagnosing the real and fake data. Then, the generator produces a number of new noise data and with forward propagation, the output will be calculated. As the generator is trying to trick the discriminator, this new output is labeled as genuine. In this step, the discriminator’s weights are kept fixed and by calculating the error and using the backward propagation technique, the generator’s weights are updated.

In Figure 1, the architecture of the GAN is shown. The parameter \( z \) is the random input vector given to the generator. The parameter \( X \) is the training data that is produced by the generator from the random input vector. These are fed to the discriminator so that it can distinguish between real and fake constructed data. Then, the error rate is calculated using the cost function which will affect the weights of the generator and discriminator. If we call the data produced by the generator as \( p_g \), then the generator’s purpose is that \( p_g \) should have the features of \( X \) as the input data. If we represent the random vector with \( z \), then \( G(z; \theta_g) \) is the function that converts the random vector into the input data space. The parameter \( \theta_g \) in the above function represents the parameters of the multi-layer perceptron network. We also denote the discriminator with \( G(x; \theta_d) \), where \( X \) represents the input data. The output of this function is a value indicating whether this data is real or artificially constructed by the generator. Similar to the generator function, \( \theta_d \) denotes the discriminator’s parameters of the multi-layer perceptron network. Simply, the GAN acts like a min-max dual game.

If we represent this game by \( V(G,D) \), the GAN operates using the Equation 1:

\[
\min_G \max_D V(D,G) = E_{x \sim p_{data}(x)}[\log D(x)] + E_{z \sim \rho_z(z)}[\log (1 - D(g(z)))] \tag{1}
\]

Where \( G \) is the generator network, \( D \) is the discriminator network, \( E \) is the cost function and \( G(z) \) is the output of the generator network.

If a sufficient amount of time and input data are available, the general answer to the above equation would be that the data artificially produced by the generator will be exactly equal to the input data (i.e. \( p_g = p_{data} \)). This is shown in Figure 2. As can be seen in the figure, the data produced by the generator is converged over time with the input data. In this figure, the bottom solid line denoted by \( z \) displays the random vector domain and the upper solid line denoted by \( x \) displays the input domain. Vertical lines are drawn from \( z \) to \( x \) show how to map a random vector domain to a data domain. Step (a) is the beginning of the network’s training, whereas steps (b), (c) and (d), respectively, show how the generated data is converged toward the input data.

Many studies have been performed in the domain of image processing using GANs. In Figure 3, the results obtained from using a GAN are shown in four different datasets. Part (a) shows how
a GAN manages to generate handwritten numbers. Part (b) illustrates the output of a GAN in the domain of human face generation. In parts (c) and (d), the GAN is used to produce animal images. The difference in the output of a GAN depends on the network architecture as well as the input data collection mechanism, as shown in the figure.

3.2. Reinforcement learning

Reinforcement learning is a class of machine learning algorithms where an agent learns optimal behavior by interacting with the environment and getting rewards for its actions. In each step, the agent perceives the environment and chooses the action it considers the best. By implementing the action, the state of the environment might change and the agent is given a numerical reward. The ultimate goal of the agent is to maximize the gained reward over time. Figure 4, shows the general scheme of a reinforcement learning algorithm.

Reinforcement learning is usually modeled using a Markov decision process [49]. A Markov decision process can be defined as \((S,A,t,r)\) where \(S\) is a finite set of possible environment states, \(A\) is a finite set of actions, \(t: s \times A \times S \rightarrow [0,1]\) is a function detailing the probability that the state of the environment changes if the agent implements an action and \(r: s \times A \times S \rightarrow R\) is the numerical reward the agent gains by implementing an action resulting in a particular state transition.

3.3. Q-learning

One of the most popular temporal difference reinforcement learning methods is Q-Learning [50]. In its basic form, it can be written as Equation 2:

\[
Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha [R_{t+1} + y \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]
\]

where \(S\) is the environment state, \(A\) is an action, \(t\) is the reward and index \(S\) is the current time step. Also, \(Q(S,A)\) is a function of how valuable it is to implement action \(A\) when the state of the environment is \(S\). The equation offers a way to update the \(Q\) function when the agent implements action \(A_t\) in environment state \(S_t\) which results in the reward \(R_{t+1}\) and a new state \(S_{t+1}\). As the agent interacts with the environment, the \(Q\) function is updated until the gained reward for any episode is maximized. As a result, the Q-Learning algorithm can be written as Algorithm 1 shown in Figure 5.

3.4. Deep Q-networks

Deep-Q-Networks or DQNs try to model the \(Q\) function with a deep neural network, usually a convolutional neural network, so that the output is a vector showing the \(Q\) values for each possible action of the agent. Equation 3, shows the way DQN is updated.

\[
Y^DQN_t \equiv R_{t+1} + y \max_a Q(S_{t+1}, a; \theta^-)
\]

DQN is inherently unstable and uses a mechanism called experience replay to use older experience for learning the \(Q\) value. This decreases the instability of the network and helps to improve the learning process dynamically.
3.5. Hierarchical reinforcement learning

One of the problems of reinforcement learning is the curse of dimensionality, where the increase in the dimensions of the problem exponentially increases the time needed for the agent to learn the optimal policy. One of the ways to tackle this problem is to define abstraction layers in the form of temporal or spatial abstractions to reduce the dimensionality of the problem and improve the learning speeds. This type of reinforcement learning is called hierarchical reinforcement learning [51].

A. The options framework

One of the most popular frameworks proposed for hierarchical reinforcement learning is the options framework [51]. In this framework, a number of options, which are macro actions that consist of a number of low-level actions are defined. Each option $O$ can be defined as $< I, \pi, \beta >$ where $I$ is a set of environment states where the option can be called, $\pi$ is the policy for this particular option and $\beta$ is the probability where the option is terminated in each environment state. In our example, options are defined as trying to reach each coin and the end state in the game. In other words, $I$ for the options that try to reach the coins is the states in which those coins exist. On the other hand, $I$ for the option trying to reach the end is the states which do not have any coin remaining on the map. The parameter $\beta$ for each of these options is the state where the agent reaches the exact tile the coin or the end is located at. For each of these options, a separate DQN is defined that gets awarded for reaching its respective goal.

The agent is only permitted to use these options as its actions. After each option is terminated, a new option from the remaining pool of the available ones is selected. This arrangement was chosen to increase the speed at which the agent learns to finish the game.

4. The proposed approach

The proposed model in this paper uses a trained neural network to generate levels that have balanced parameters. In other words, if we input a vector with random values to the neural network, the output will be a balanced level. In this section, we will first present the general framework and then the detailed algorithms of the proposed model.

As for the first step, we have designed a Mario-like 2D platformer game using the Python programming language. The levels of the game contain the starting point, ending point, platforms, and coins. The rule of the game is that the player starts from the starting point. Then, they must get all of the coins. In the end, after receiving every coin, the player should go to the ending point to win and pass the level. If the player fails to finish the game in 300 cycles or falls down from a platform, they lose. As we have used a reinforcement learning algorithm to identify whether a level is solvable after a certain amount of repetitions or not, earning coins or reaching the endpoint has a positive score while failure creates a negative score. For every positive score, the AI learns the pattern and will try to apply it in the next steps. In the next step, a series of simulations are performed. If the level is solved at least once in 300 attempts, this level is stored as a balanced solvable level. Otherwise, it will be stored as an un-solvable and unbalanced level. The 300 number is a selected threshold we have used in this paper. This threshold can be easily modified by different game designers based on their needs and requirements. These solvable levels are fed to the GAN network so that it can start generating similar levels with the same overall characteristics. Hence, satisfying the aim for automatically creating balanced levels.
4.1. The general framework

The GAN requires a very large number of input data points. To achieve this, we need to design a game where its levels could be used as the neural network inputs. Therefore, a two-dimensional 2D platformer game is designed for this purpose. The logic of the game is written using the Python programming language. In addition, Pygame is used to display the logic of the game if the user has enabled the graphical implementation of the game's settings. The simple 2D platformer game designed for our research is shown in Figure 6.

The general layout of the proposed approach is shown in Figure 7. As can be seen in the figure, the game's map is read from a bitmap image with a size of 15*8 pixels. Every pixel's RGB values represent the status of that cell in the game. At the beginning of the game, the level is generated based on the pixels of the map written using Algorithm 2 shown in Figure 8. The player in the game can go right, left or jump. The jump can only be performed when the player is standing on a platform. The jump speed and the player speed can be changed in the game configuration file.

The logic of the game at any time is calculated according to the user’s inputs as well as the previous state of the game map. For example, if the user pushes constantly to the right, during that time cycle, the player moves based on the player speed in the configuration file. Other aspects such as game physics, collision with coins or endpoints, gravity calculations and anything that requires game's logic is performed in this cycle and then the state of the game will be transitioned to the next time cycle. In the rest of this section, the primary stages will be explained in detail.

4.2. Level generation

In order to obtain balanced levels, it is necessary to create levels in a random manner. Then, among the generated levels, select the ones that are balanced and use them as inputs for the neural network. Therefore, algorithm 2 is used to randomly map the levels. These levels should have the necessary conditions for a playable game. This randomly generated level may be balanced, unbalanced, or even unsolvable. For this reason, reinforcement learning is applied to determine the level's solvability.

4.3. Level evaluation

Random levels should be considered in terms of solvability and balance. For simulating the way, a real player progresses through a level, reinforcement learning has been used. That is, each randomly generated level is played 300 times. If among these 300 times, the level is solved even once, it is stored at a balanced level. Of course, this parameter is only selected as an intuitive baseline and can change depending on the actual requirements of the level designer.

Reinforcement learning ultimately tries to maximize the accumulated reward for the agent. If the rewards in the game are set up in a way that reaching the coins or getting to the ending state while having no remaining coins, will result in a positive numerical reward for the agent, the reinforcement learning algorithm tries to maximize the reward and helps the agent finish the game. While the goal of this paper is not to create a player agent for the game environment, for the purpose of evaluating the created maps, such an agent can be used to see if a map can actually be solved or not within the stated number of repetitions. Therefore, a hierarchical reinforcement learning agent with options framework and DQN as the baseline algorithm is set up in the game and given the task of finishing a given map. If the agent is able to finish the map in less than a number of episodes, the map is considered solvable. As there is no need to actually learn how to play the game, the first time an agent finishes the given map, the learning process is terminated.
The DQNs are defined as neural networks of four initial convolutional layers with two layers of multi-layer perceptron after them. The last layer has as many neurons as the actions the agent can use in the game which are \textit{left, right and jump}. Options are defined as explained in Section 3.5 and every one of them has a DQN assigned to learn how to reach its specific goal.

By using this architecture, the chance of determining whether a map is solvable or not is increased as previous successes in reaching intermediate goals such as getting coins are used to improve the performance of the agent in the next episode. Algorithm 3 shown in Figure 9 shows this architecture for evaluating a generated map. The architecture of the deep neural network used in the reinforcement learning algorithm of this study is also shown in Figure 10.

4.4. Learning level generation using GAN

After the balanced levels are saved, they should be fed to a suitable extension of GAN. For this research, the implemented model of deep convolutional GAN is considered. Adam Optimizer has been used with an input of 0.003. Also, training the network is performed using the binary cross-entropy method.

The assumed batch size of the network is 32 and for each dropout layer, it is considered to be 0.2. As discussed in Section 3, the GAN consists of two parts namely the generator and the discriminator. The GAN for this research has a different activation function on each layer which will be presented in detail in the following.

A. The generator

The generator is a neural network that receives noise data as input and delivers data to the output layer. The generator's neural network architecture used in this research is shown in Figure 11.

As shown in Figure 11, a 1*100 vector of the noise data is received randomly in the input layer of the generator. Then, in the next layer, a layer with 256*15*8 neurons will be placed as the tanh activation function. In front of this layer, there exist three layers of 8*15*128. The output layer is an 8*15*5 convolution with sigmoid activation function which is the same image dimensions as that of the level in five different channels. In this research, each pixel color is considered as a channel. The result is a 5-channel neural network. In this way, when reading balanced maps to use as inputs, the input is stored as a 3D array. In this 3D array, the first dimension is the row, the second dimension is the column and the third dimension is the channel.

B. The discriminator

The discriminator is a neural network that receives the training data and determines whether the input data is real or fake. The discriminator’s neural network architecture used in this work is shown in Figure 12. As shown in the figure, in the input layer of the discriminator, an 8*15 image that contains five channels is received. This image, if provided as the training dataset, has a real data tag and if it is received from the output of the generator, it has a built-in label.

Then, in front of this layer, there are three layers with 128*15 *8 neurons with the activation function tanh. In the next layer, a layer with 256 neurons is placed as the tanh activation function. The output layer contains a neuron with a sigmoid activation function whose value is between 0 and 1. This number represents the extent to which this learning data belongs to the training data set or the output of the generator.

Algorithm 4 shown in Figure 13 represents the training algorithm for the GAN used in this research.
After the training process, as well as the allocation of the deep neural network’s weights, are finished, a new level is created using the algorithm 5 shown in Figure 14. This level is a better candidate from the viewpoint of the discriminator.

Algorithms 4 and 5, respectively perform the training process. In the end, a model that generates balanced levels is stored to be used in the level generation process. The summary of the steps taken in the proposed approach are as follows:
1. The generator constructs a random 100-digit vector between 0 and 1.
2. To perform a forward propagation, the output of each layer goes to the next layer. This will result in the final layer of the generator to create an image of the level.
3. The discriminator receives 32 images from the generator and 32 images from the training dataset as input.
4. The output of the discriminator’s final layer, which is the prediction of the network from each data label, is computed in the form of forward propagation.
5. The generator will keep its weights fixed and by calculating the error and using backward propagation, the weights of the discriminator are modified to better detect the real and fake data.
6. The generator produces a number of new noise data which is calculated by the forward propagation method.
7. Finally, the weights of the discriminator are kept constant. By calculating the error and using backward propagation, the weights of the generator are modified with the aim of producing data having the characteristics of the input data set.

After the completion of every 50 learning steps, the generator creates 25 levels. To generate any of these 25 steps, the generator first produces 20 levels and then the discriminator stores the level that is considered to be the best.

5. Evaluations and comparisons

In this section, the evaluation scenarios devised to demonstrate the applicability and accuracy of the proposed model are given. First, the simulation setup is discussed and then, the designed evaluation scenarios are given.

5.1. Simulation setup

The simulations for the 100,000 different levels were performed on a computer with a Core i7 7800 processor, an NVidia 1070 Ti graphic card, and 32GB of RAM for 40 days. From these 100,000 levels, 56,000 levels were stored as balanced and subsequently used as input data for the GAN. In another configuration, we have performed the same evaluations using the cloud computing center of Iran University of Science and Technology [52]. We have provisioned 4 Core i7 7800 processors, 4 NVidia GTX 1080 (SLI), and 80 GB of RAM. The evaluation time was reduced to 11 days.

Generally, training deep learning approaches using a huge dataset is a time-consuming process. Most of the related work in this field has been faced with a similar challenge. For example, Frank et al in their research has tried to reduce the training time of their previous work which takes more than 10 days to be completed [53]. In another related work, Berner et al have tried to use deep reinforcement learning approaches to learn the Dota 2 game and the learning time takes more than 3 months to be finished [54].
Because the training process takes place on a large volume dataset, it is also intensive in terms of power consumption. We have applied a set of techniques to potentially control and reduce the amount of required power. These techniques are as follows:

1. In the proposed approach, the agent will stop the learning process whenever it can solve the level in any of the attempts. Due to the provided definition of a balanced level in our work, the average learning time would be much faster compared to the case where more than one attempt was required to evaluate the balance characteristic for a level.

2. The agent will skip the level that is not solvable which saves time for learning other remaining levels.

Finally, it is worth mentioning that the learning process is a pre-processing activity. After the learning part is completed, the identification of balanced levels will be performed significantly faster compared to a manual process. Such automation is the primary aim of the current research and also the related researches.

5.2. Evaluation scenarios

In the proposed approach, when the balanced levels are selected, they are then fed into the DCGAN architecture as the training input data. After about 170,000 epochs, the network begins generating levels that meet the minimum requirements. Also, 25 random levels in the form of images are generated and saved every 50 full training epochs. Figure 15 shows how the change in the output of the generator network during the training process is occurring.

As shown in the figure, the pixels of each step are set up completely random. After a small amount of training, the network learns that more pixels of the image should be white. It is also evident from how the platforms are placed in the image that the network is learning through time. In the early stages of the training process, the network cannot properly guess the number of cells for starting and ending points. But, as the training progresses, the network’s advancement is significant. As the training network grows more and more, the number of starting and ending cells reduces till the point where only one starting and ending points are generated in the later stages and coins are also modified to the right size. In the figure, the yellow squares denote the coins, black squares are the platforms, the green and red squares denote the starting and ending points respectively and white squares are the empty spaces. The network does not initially generate levels that have the exact requirements given in Algorithm 2 shown in Figure 8. In other words, as the main approach of this work is to use GAN networks as an inherently unsupervised learning approach, such adherence will take place after the training process is completed both for the generator and discriminator components. Hence, initially there is a possibility that the generated maps from the GAN network have 3 or 4 coins or the number of platforms differ in count as given in Algorithm 2. But, such differences do not fail the main purpose of this paper and eventually and after the training process is completed levels with the stated conditions will be generated.

After completing the network’s training process, stored models are used to assess the condition of the level. As it was mentioned, the randomly generated levels have a series of basic rules. The minimum requirements of a map in the game are:

1. It should have exactly one starting point.
2. It should have exactly one ending point.
3. There must be at least 1 and at most 3 coins on the playing field.
4. There must be at least 4 platforms.
5. The level must be solvable between 10 to 300 episodes of RL assisted search. In other words, in order to keep the difficulty balance of the created level and preventing the level from being
too easy or too difficult, the condition for assuming a level to be balanced is considered to be the number of attempts that the RL agent tries to solve the level. Hence, if the agent solves the created level in less than 10 tries, then the level is considered to be too easy and if the number of tries to solve the level gets above 300 then the level is assumed to be too difficult to solve. Of course, these threshold numbers can be changed and modified by the level designer to get the best outcome.

For every 1000 epochs of training, 200 levels are created by the generator. Each of these levels is selected by generating 20 new levels using the generator and picking the best one using the discriminator. Each of these 200 levels is checked to see if they meet the minimum requirements. The result of this assessment is shown in Figure 16. Based on this figure, it can be seen that the minimum requirements are met quite early in the training of the GAN and it reaches a 95 percent adaptation rate to level requirements after completing about 40000 training epochs. On the other hand, as seen in Figures 17, 18 and 19, it takes more time before the average accuracy of the GAN model peaks. Also, meeting the minimum requirements is not the same as generating balanced levels. So, another evaluation step is necessary to assess this approach.

For evaluating the solvability of the produced levels, after every 10000 training epochs, the model is saved and the generated levels are checked for solvability. Therefore, these models, with the generator and the discriminator network are placed alongside the simulation file. Then, instead of getting the level from the code, the levels are obtained from the generator model. Getting the level from the trained model is performed by generating 20 levels. The discriminator selects the best level amongst those. Then, if the selected level has the necessary conditions to be used as a level, it will be used in the game itself. Otherwise, this process continues until the level that has all the requirements for being a level is selected.

To evaluate the performance of the GAN used in this paper, the model state in every 10000 training epochs is used to create potentially solvable levels. In this way, 17 models are selected from the stored models. Then, from each of these models, 2000 levels are generated and the solvability rate is calculated. In addition and for the sake of having a baseline for the comparison, the game has been executed 2000 times using a random level generated by a random level generator. Figure 20 shows the percentage of the balanced levels created using both of the level generators. As shown in Figure 20, the best rates of the solvable levels are in the model saved after the 150000s training epoch in which approximately 83.6 percent of the levels that the generator produces are solvable and balanced. It is also observed that the solvability rate of the levels generated using the best-learned model has improved by 17.5% compared to the random level generator.

6. Final Discussions

One of the points mentioned among the advantages of the proposed approach is the non-existence of the human operator involvement in the level generation process. In other words, a large set of random levels are fed to the GAN to generate the appropriate levels based on the level requirements specified by the level designer. This choice is made based on the fact that creating human-generated levels is both time-consuming and costly. It is important to note that we do not claim that the proposed approach has better accuracy compared to the manually generated levels using a human operator and can replace this manual process. The primary advantage of the approach in an applied context is that it can provide the level designer with a huge number of balanced levels in which she/he can select from based on her/his preferences or taste. Of course, by changing the
required parameters in different application contexts, the approach can be viewed as a potentially
general mechanism for creating levels in various game genres.

7. Conclusions

Creating balanced levels in video games is one of the most important challenges for game and
level designers. The lack of attention to the balance of the game may cause a major failure and a
huge profit loss for the company. Thus, in order to automatically create balance in the game
consequently reducing the human-error in the design process, in this paper, an approach is
proposed based on generative adversarial networks (GANs). The proposed approach focuses on
generating balanced levels in 2D platformer games. To do so, a level generator is created that
generates random levels. Then, using reinforcement learning algorithms the solvability of the
levels is investigated. The levels that are identified as solvable are stored as potential candidates
and are then used as inputs of the GAN’s neural network. After the completion of the training
process, the neural network has saved its best model and in the end, it has produced a series of
balanced levels. The generated levels and their appropriateness are then evaluated and compared
to a baseline random level generator. It is shown that the proposed approach using GAN can
generate a balanced level with 83.6 percent accuracy. This result implies that, if a level designer
maps the levels into an image file and apply a GAN-based approach such as the one presented in
this work, she/he can build a new balanced level with high precision.

The approach used in this paper has the following strengths compared to the other commonly-
used approaches in the respective field:

1. Because the input is fed as images, it can be used in other game genres in which the levels or
parameters can be stored in an image format. Therefore, this research is not specific to the 2D
platformer game genre.
2. The selection of the features of the neural network is not performed by a human operator. Thus,
the probability of the existence of human errors is low.
3. After training the neural network, the process of generating a new balanced level is performed
efficiently and with high speed.

This research can be improved in the future to a great extent. The suggested extensions we think
are possible for the future of this research are as follows:

1. Changing and improving the parameters and layers of the generative adversarial network may
result in better outcomes. The modifications of the parameters and the number of layers or
even using another activation function that is better suited to this problem can be considered
as a natural next step.
2. Experimenting with the applicability of the method in a game from another genre. The
suggested genres that seem to be a potential fit are dungeon crawlers, beat’em up games and
even first-person shooters.
3. Using the proposed framework to generate not only levels but also other types of content like
personalized units, balanced weapons, business model design and even texture and 3D model
generation are some of the exciting areas that can be explored.
4. In this research, we tend to generate levels using GAN networks with the balance conditions
mentioned by game designers. The aim has not been on generating levels with the conditions
given in the base random level generation algorithm (i.e. Algorithm 2). Creating a better base
random level generator and trying to force the GAN network to generate levels that have
exactly the same rules and conditions of the base level generator can be an interesting line of
research for further expansions.
5. The reinforcement learning agent in this article has been used in order to check solvability and difficulty of levels in the shortest possible time. Because of the approach of the reinforcement learning algorithms, the results seem to be logical but it does not guarantee that all of the levels have engagement or fun for players. So, the games may follow the definition of balance used in this paper but may not be necessarily engaging or even fun. Putting more focus on the engagement metrics of the levels is an interesting subject to be explored in further expansions.

6. The network used in this paper takes a long time to check and distinguish balanced and unbalanced levels. Changing this network or using an additional pre-processing stage (for example identifying impossible levels immediately and filtering them before giving the generated levels to the RL algorithm) can improve the overall speed of the approach making it more practical to be used in more complex scenarios.

Compliance with Ethical Standards

This study has received no funding from any organization.

Conflict of Interest

All of the authors declare that they have no conflict of interest.

Ethical approval

This article does not contain any studies with human participants or animals performed by any of the authors.

References


Biographies

Morteza Rajabi has completed his bachelor and masters degree in computer engineering from Iran University of Science and Technology. His research interests lies in the utilization of machine learning methods in the domain of video game development.

Mehrdad Ashtiani received his B.S. degree in software engineering from Iran University of Science and Technology (2009), Tehran, Iran, the M.S. degree in software engineering from Iran University of Science and Technology (2011), Tehran, Iran and finally his Ph.D. degree from Iran University of Science and Technology (2015), Tehran, Iran. He currently is an assistant professor at the school of computer engineering-Iran University of Science and Technology. His main research interests include trust modeling and the applications of uncertainty modeling to the domain of computer science.
Behrouz Minaei Bidgoli is currently an associate professor at the school of computer engineering-Iran University of Science and Technology. He leads the Data Mining Lab (DML) that performs research on various areas in artificial intelligence and data mining, including text mining, web information extraction, and natural language processing.

Omid Davoodi is a PhD candidate at Carleton University, Ottawa, Canada. He completed both his bachelor and masters degrees in computer engineering at Iran University of Science and Technology. His research interests include explainable artificial intelligence, reinforcement learning and procedural content generation using machine learning.

Figure 1. GAN’s architecture [48].
Figure 2. The convergence of data made by the Generator toward the input data [16].
Figure 3. Outputs of GAN compared to the original data in (a) MNIST dataset (b) TFD dataset (c) CIFAR-10 dataset (fully connected model) (d) CIFAR-10 dataset (convolutional discriminator and deconvolutional generator) [16].
Figure 4. The general schema of the reinforcement learning agent [23].
Figure 5. Algorithm 1: Q-learning: An off-policy TD control algorithm.
Figure 6. The designed 2D platformer game for the proposed approach.
Figure 7. The general layout of the introduced framework.
Figure 8. Algorithm 2: Level generation.
Figure 9. Algorithm 3: Reinforcement learning algorithm for solvability.
Figure 10. Reinforcement learning’s neural network architecture.
Figure 11. The generator’s neural network architecture.
Figure 12. The discriminator’s neural network architecture.
Figure 13. Algorithm 4: Training the Generative Adversarial Network.
Figure 14. Algorithm 5: Generating the map.
Figure 15. The progress being made in the generated level from the starting epoch to the end. The top left image shows the first epoch which is completely random. As the training progresses, better levels are generated both from a logical and balance points of view. In the final epoch shown as the bottom right image, a well-formed final level is generated.
Figure 16. The percentage of generated maps that meet the minimum requirements per each 1000 epoch.
Figure 17. Discriminator loss in DCGAN training epochs.
Figure 18. Model accuracy average in DCGAN training epochs.
Figure 19. Generator loss average in DCGAN training epochs.
Figure 20. The comparison between the balanced levels generated from the proposed model and a random level generator.

Table 1. A comparison of game balance approaches
Figure 1.

Figure 2.

Figure 3

Figure 4.
Algorithm 1: Q-learning: An off-policy TD control algorithm

Input:
(1) $K$: number of training episodes
(2) $M$: number of steps for each episode

PROCEDURE:
1: Initialize $Q(S,A)$, $\forall S \in A(s)$ arbitrarily, and $Q(terminal\ state) = 0$
2: FOR (i = 0; i < K; i++)
3:     Initialize $S$
4:     FOR (j = 0; j < M; j++)
5:         Choose $A$ from $S$ using policy derived from $Q$ (e.g. $\epsilon$-greedy).
6:         Take action $A$, observe $R$ and $S'$.
7:         $Q(S, A_j) \leftarrow Q(S, A_j) + \alpha \left( R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S, A_j) \right)$
8:         $S \leftarrow S'$
9:     IF ($S$ is terminal)
10:         break
11:     END-IF
12:     END-FOR
13: END-FOR
END-PROCEDURE

Figure 5.

Figure 6.

Figure 7.
**Algorithm 2: Level Generation**

**Input:**
1. \( c \): number of coins
2. \( x \): number of row slots
3. \( y \): number of column slots

**Output:**
1. An image in size of \( x \) and \( y \) with \( c \) coins and one starting flag and one ending flag

**PROCEDURE:**
1. Compute \( \text{randVar} \) value from a random value between 4 and 8.
2. FOR (\( i = 0 \); \( i < \text{randVar}; i++ \))
3. Select random point in the map that is free.
4. Create random horizontal platforms of between 0 and 4 starting from that point.
5. END-FOR
6. Select a random point in the map exactly above a platform that is free and set it as the starting flag.
7. Select a random point in the map exactly above a platform that is free and set it as the ending flag.
8. FOR (\( i = 0 \); \( i < c; i++ \))
9. Select a random point in the map one or two pixels above a platform that is free.
10. Place a coin on the selected point.
11. END-FOR

END-PROCEDURE

**Figure 8.**

**Algorithm 3: Reinforcement learning algorithm for solvability**

**Input:**
1. A generated map
2. \( K \): number of training attempts
3. \( M \): RL training episode

**Output:**
1. Whether the map is solvable or not

**PROCEDURE:**
1. Start the RL agent in the environment.
2. FOR (\( i = 0 \); \( i < M; i++ \))
3. IF (the episode was completely successful)
4. Finish the training and add the map to the solvable levels
5. END-IF
6. IF (number of training episodes > \( K \))
7. Stop the training and add the map to the unsolvable levels
8. END-IF
9. Perform RL training
10. END-FOR

END-PROCEDURE

**Figure 9.**

**Figure 10.**
Algorithm 4: Training the Generative Adversarial Network

Input:
(1) The level map in size 8*15
(2) Iteration: number of training iterations
(3) X: number of training steps

Output:
(1) Trained weights for creating the level maps

PROCEDURE:
1: Start the RL agent in the environment.
2: FOR (i = 0; i < Iteration; i++)
3:     FOR (j = 0; j < X; j++)
4:         Sample mini-batch of m noise samples \( z(1), \ldots, z(m) \).
5:         Sample mini-batch of m examples \( x(1), \ldots, x(m) \) from the dataset.
6:         Update the discriminator by ascending its stochastic gradient.
7:     END-FOR
8:     Sample mini-batch of m noise samples \( z(1), \ldots, z(m) \).
9:     Update the generator by descending its stochastic gradient.
10: END-FOR
END-PROCEDURE

Algorithm 5: Generating the map

Input:
(1) Random numerical vectors of size 100 with values between 0 and 1

Output:
(1) Level maps

PROCEDURE:
1: Give the input vectors to the generator and compute the output maps
2: Use the discriminator to get its opinion on the fakeness of the generated maps
3: Sort the maps using the fakeness value
4: Select the maps that the discriminator considers the least fake.
END-PROCEDURE
Figure 18.

Figure 19.

Figure 20.
Table 1.

<table>
<thead>
<tr>
<th>Proposed approach</th>
<th>Ability to change the game mechanism and parameters</th>
<th>Ability to generate levels</th>
<th>Dynamic difficulty adjustment</th>
<th>Using RL in order to check level solvability and difficulty state</th>
<th>Using GAN for generating levels</th>
<th>Not using human-generated levels as a dataset</th>
<th>Allowing designers to define balance in a customized way</th>
<th>Ability to potentially use the proposed model in other game genres</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evolving Mario Levels in the Latent Space of a Deep Convolutional Generative Adversarial Network</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dynamic game difficulty balancing in real-time using evolutionary fuzzy cognitive maps</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Automated game balancing in Ms. PacMan and StarCraft using Evolutionary Algorithms</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Game balancing with ecosystem mechanism</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Dynamic difficulty adjustment through an adaptive AI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Learning the patterns of balance in a multi-player shooter game</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>DOOM level generation using generative adversarial networks</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Balancing Players of Different Skill Levels in a First-Person Shooter Game</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>An Automated High-Level Saliency Predictor for Smart Game Balancing</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Real-time challenge balance in an RTS game using rtNEAT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Evolving a Designer-Balanced Neural Network for Ms PacMan</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>PSMAGE: Balanced Map Generation for StarCraft</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Generating an Attribute Space for analyzing Balance in Single Unit RTS Game Combat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Dynamic Difficulty Adjustment of Game AI for video game Dead-End</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Balancing Turn-Based Games with Chained Strategy Generation</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>