Evolving Game State Features from Raw Pixels

Baozhu Jia(✉) and Marc Ebner

Institut für Mathematik und Informatik,
Ernst Moritz Arndt Universität Greifswald,
Walther-Rathenau-Strasse 47, 17487 Greifswald, Germany
{baozhuj,marc.ebner}@uni-greifswald.de

Abstract. General video game playing is the art of designing artificial intelligence programs that are capable of playing different video games with little domain knowledge. One of the great challenges is how to capture game state features from different video games in a general way. The main contribution of this paper is to apply genetic programming to evolve game state features from raw pixels. A voting method is implemented to determine the actions of the game agent. Three different video games are used to evaluate the effectiveness of the algorithm: Missile Command, Frogger, and Space Invaders. The results show that genetic programming is able to find useful game state features for all three games.

Keywords: Genetic programming · General video game playing · Voting method

1 Introduction

All types of games, including video games and board games, provide a testbed for artificial intelligence research. AI technology has developed successful game players for many games, including go and chess. In 1996, Deep Blue [12] became the first computer to win a chess game against the world champion Garry Kasparov. In 2016, AlphaGo [18] beat Lee Sedol in a five-game match. This was the first time that a computer Go program beat a 9-dan professional Go player. These examples show that a computer program can be better than a human being in certain specific areas. Some would even argue that the program has a higher intelligence for this particular area than the human player. However, usually games where computers have become good players are no longer considered to require intelligent behaviour.

The success of these AI technologies has inspired researchers to explore machines with more general-purpose intelligence. The computer program is no longer limited to play one specific game, instead knowledge can be transferred to another game. General Video Game Playing (GVGP) is the design of artificial intelligence computer programs that can play many different video games. Therefore, the computer program should be game-independent and use as little game specific knowledge as possible during the learning process.

DOI: 10.1007/978-3-319-55696-3_4

The main contribution of this paper is using genetic programming [9,10] to evolve game state features. A voting based method is used to determine the behaviour of the game agent, i.e. direction of motion and shooting behaviour. During the learning process, the only game knowledge used in this paper are game screen grabs and game scores. This knowledge is obtained from the game engine and then passed on to the learning algorithm.

In order to evaluate the efficiency of the algorithm, three different video games are used to evaluate the algorithms: Frogger, Missile Command and Space Invaders. All games run on the general video game engine [2]. We compare our results with another genetic programming algorithm [8] which uses handcrafted game state features. The results show that the algorithm which uses evolved game state features from raw pixels performs better than the algorithm which uses handcrafted features.

This paper is organised as follows. Section 2 briefly introduces previous work. Section 3 describes the game engine and games which are used to evaluate the algorithm. Section 4 presents how genetic programming is used to evolve visual features. The results are shown in Sect. 5. Section 6 gives the conclusion.

2 Related Research

The General Game Playing Competition [4] has been organised every year by AAAI since 2005. This competition focuses on turn-taking board games whose rules are not known. Monte Carlo tree search [3,13,14] has shown its powerful search ability in general game playing.

Bellemare et al. [1] created an Arcade Learning Environment which provides a platform to evaluate general, domain-independent AI technology. Naddaf [16] introduced two model free AI agents to play Atari 2600 console games in his master's thesis. One AI agent uses reinforcement learning while the other uses Monte Carlo tree search. Two handcrafted game state features are introduced. Hausknecht [6,7] presented a HyperNeat-based general video game player to play Atari 2600 video games. It uses a high-level game state representation. HyperNeat is said to be able to exploit geometric regularities.

In 2015, DeepMind [15] presented a Deep Q-Network which combines deep learning and reinforcement learning to play Atari 2600 video games, achieving human-level game play in many games. Deep Q-Network did so with minimal prior knowledge, receiving only visual images (raw pixels) and game scores as input. Guo et al. [5] trained both a neural network and a deep neural network with the action choice returned from Monte Carlo tree search as ground truth. Monte Carlo tree search is assumed to make correct decisions if it has enough searching depth. In 2016, AlphaGo [18] created by DeepMind has beaten Lee Sedol, a top-level Go player. AlphaGo relies on two different components: A tree search procedure and convolutional neural networks to guide the tree search. Two convolutional neural networks are trained: one is a policy neural network and the other is a value neural network, which are also trained using Reinforcement Learning.

GVG-AI [2] is another platform to test the general AI technology. Based on this platform, General Video Game AI Competition has been held every year since 2014. This competition explores the problem of creating controllers for general video game playing. Perez et al. [17] put forward a knowledge-based Fast Evolutionary Monte Carlo tree search method, where the reward function adapts to the knowledge and distance change. Jia et al. [8] presented a video game player based on genetic programming. Three handcrafted game state representations were used. Three trees were evolved based on Genetic Programming whose values were used to determine the game agent's movement in the horizontal/vertical direction and the shooting behaviour.

3 Materials

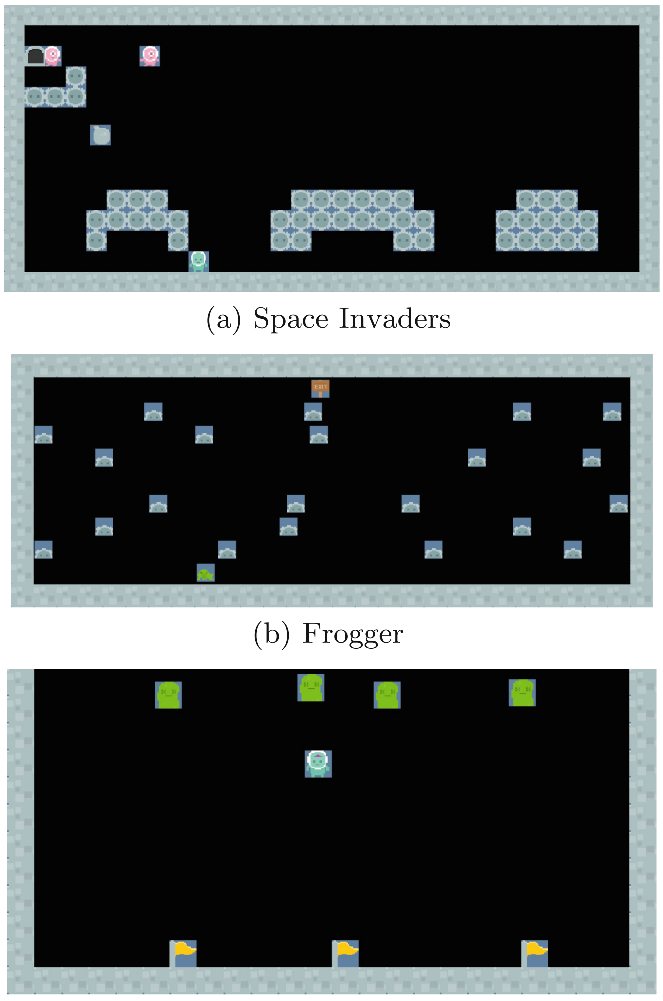
3.1 Games

The games used to evaluate the algorithm in this paper are run on the GVG-AI game engine. This game engine is able to run many games similar to old Atari 2600 games. We test our video game player on three different games: Space Invaders, Frogger and Missile Command. Screen grabs from the three games are shown in Fig. 1. Space Invaders is a classic arcade game. An alien invasion is coming in from above. The player has to control a small gun which is able to shoot vertically at the incoming space ships. Frogger is a classic game where a small frog needs to cross a road. The player essentially needs to move from the bottom of the screen to a goal position located at the top of the screen. The game agent has to watch out for cars while crossing the road. In Missile Command, the player needs to use smart bombs to destroy incoming ballistic missiles. The player needs to decide at what location the next smart bomb will explode. A smart bomb will destroy all incoming missiles within a certain radius.

3.2 Handcrafted Game State Features

In our previous work, we presented a method which combines handcrafted game state features and Genetic Programming in order to play different video games. We first identify the positions of all objects shown in the game screen. Five different object classes are distinguished. For each class, we inform the game avatar about the position of the nearest object as illustrated in Table 1. The game state is represented by four terminal symbols X_i, Y_i, D_i, and A_i with $i \in \{1, \ldots 5\}$. The position of the nearest object is available as x- and y-coordinates, via symbols X_i and Y_i. The distance to the nearest object is available through D_i and the angle is given by A_i.

Three trees T1, T2 and T3 are evolved by taking the game state features as input to determine the behaviours of the avatar. Among them, the value of T1 and T2 determine the avatar's moving direction in the horizontal and vertical direction, as illustrated in Fig. 2. The value of T3 determines whether the avatar will release a shooting action as illustrated in Table 2.



(a) Space Invaders

(b) Frogger

(c) Missile Command

Fig. 1. Three games created using the game engine GVG-AI.

Table 1. Terminal symbols [8]

Terminal symbol	Description
X_i	x coordinate of the nearest object for class i relative to avatar
Y_i	y coordinate of the nearest object for class i relative to avatar
D_i	Euclidean distance between avatar and the nearest object for class i
A_i	Angle between vector pointing from self to the nearest object and the horizontal axis

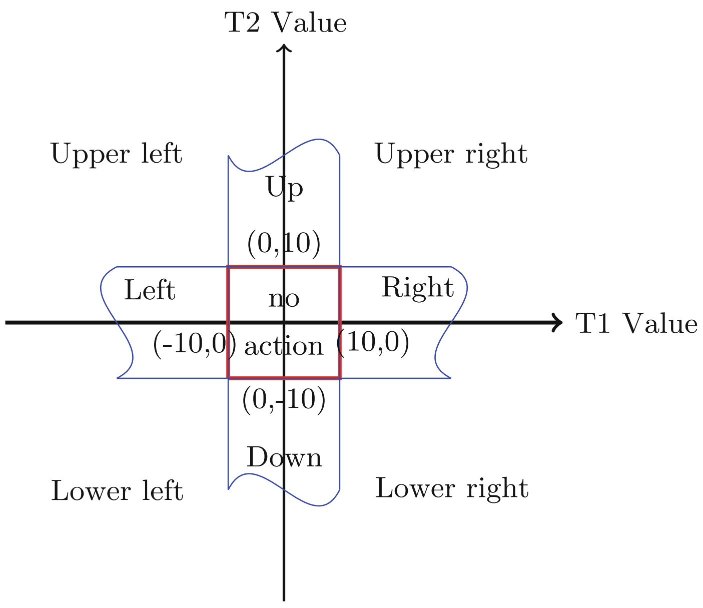


Fig. 2. The moving direction of the avatar depends on the value of tree 1 and tree 2.

Table 2. Depending on the value of tree 3, the button will be pressed or not.

Value of tree 3	Actions
$[0, \infty)$	Press_button
$(-\infty, 0)$	No_action

4 Evolving Video Game State Visual Features Using Genetic Programming

4.1 Evolving Game State Features

It is a great challenge for a game player to find the proper game state features, especially for games with complex game play. In this paper, we use genetic programming [9,10] to evolve the game state visual features.

The computer programs evolved by genetic programming are traditionally represented as tree structures. Trees can be easily evaluated in a recursive manner. We used the ECJ package [11] to evolve the playing strategies in this paper. In our work, the game engine communicates with ECJ via the TCP/IP protocol. For each step, the game engine sends the game screen grab to ECJ. The evolved program then computes and returns back to game engine the actions which will be executed by the avatar. Once the game ends, irrespective of whether the game is won or lost, the game scores are passed on to ECJ, and are used to compute the fitness of the program.

The terminal set, that we have used for our experiments, is described in Table 3. All terminals return an object of type Image. Each terminal returns one channel of the down-scaled game screen grab: red channel, green channel, blue channel, yellow channel and grey channel. The red, green and blue channels are readily available from the screen grab. The other channels are computed from this data. During the learning process, the game screen grab is passed on to ECJ from the game engine. The size of the down-scaled screen grab is one sixteenth of its original size.

The elementary functions are shown in Table 4. From this table, we see that all the arguments and return values of these functions have the type Image. We have used arithmetic functions, such as addition, subtraction, multiplication and division. It should be noted that a protected division is used here.

They are used to combine features from different channels. We also applied a Gaussian filter and a non-local-maxima suppression function. If the latter two functions are applied in sequence, objects will be reduced to points, i.e. the local maxima. The attenuation function can be used to put an emphasis on objects close to the avatar. The attenuation function is a exponential function, whose value will attenuate with the object's distance to avatar. It can be used to put an emphasis on objects close to the avatar. The return value of the tree is an image which has the same size as the input image.

For each experiment, we perform 10 runs with different initialisation of the random seed. For each run, a population with 200 individuals is evolved for up to 100 generations. Crossover is applied with the probability 0.4. Mutation is applied with the probability 0.4. Reproduction is applied with the probability 0.2. Tournament selection is of size 3 is used as strategy. The ramped half-and-half tree building method (HalfBuilder) is used to initialise the first population of individuals.

Table 3. Terminal symbols I

Terminal	Return type	Description
imageGray	Img	Gray image
imageR	Img	Red channel
imageG	Img	Green channel
imageB	Img	Blue channel
imageY	Img	Yellow channel

4.2 Voting for Actions

As described in the previous section, the return value of the genetic programming tree is an image. We search the resulting image in order to locate the position of the maximum value (V_{max}) and the minimum value (V_{min}). The point having the maximum value is regarded as the goal position of the game, i.e. a location on the screen that is of positive interest. The point having the minimum value is regarded as the position of a potential threat to the game avatar. The behaviour of the game avatar will be determined by these two points. It should always move towards the goal. However, it should also keep an eye on the potential threat. Once the location of the potential threat enters a certain area surrounding the avatar, then the avatar will move away from the threat. In this paper, we combine

Table 4. Function set

Function	Output type	Description
add(Img a, Img b)	Img	$o(x, y) = a(x, y) + b(x, y)$
subtract(Img a, Imgb)	Img	$o(x, y) = a(x, y) - b(x, y)$
multiply(Img a, Img b)	Img	$o(x, y) = a(x, y) \cdot b(x, y)$
divide(Img a, Img b)	Img	If $b(x, y) \neq 0$ then $o(x, y) = a(x, y)/b(x, y)$, otherwise $o(x, y) = 0$
gaussian(Img a)	Img	Gaussian smoothing with standard deviation 1.1
attenuation(Img a)	Img	Attenuates the image data $i(x, y)$ depending on its distance to the avatar. $o(x, y) = i(x, y) \cdot \exp^{-distance/50}$
nlms(Img a)	Img	Non local maximum suppression with neighbourhood of 5×5
gate(Img a, Img b, Img c)	Img	If $a(x, y) > 0$ then $o(x, y) = b(x, y)$, otherwise $o(x, y) = c(x, y)$
max(Img a, Img b)	Img	If $a(x, y) > b(x, y)$, then $o(x, y) = a(x, y)$, otherwise $o(x, y) = b(x, y)$
min(Img a, Img b)	Img	If $a(x, y) < b(x, y)$, then $o(x, y) = a(x, y)$, otherwise $o(x, y) = b(x, y)$
upper(Img a)	Img	Keeps only values in the uppermost 25%. Let the pixel range be $[V_{\min}, V_{\max}]$. If $i(x, y) \geq V_{\max} - 0.25(V_{\max} - V_{\min})$ then $o(x, y) = i(x, y)$, otherwise $o(x, y) = 0$
lower (Img a)	Img	Keeps only values in the lowest 25%. If $i(x, y) \leq V_{\min} + 0.25(V_{\max} - V_{\min})$ then $o(x, y) = i(x, y)$, otherwise $o(x, y) = 0$

the two behaviours by voting. If one action helps to move towards the goal, it will get a reward +1. Whereas, if the action will cause a threat to avatar, it will get a punishment score −5, if the action helps move away from the threat, it will get a reward +2 (Table 5). If neither happens, then the value will remain the same. The action with the largest value will be the one used to determine the direction of motion. The maximum value also determines the action of button, which will be pressed if and only if $V_{max} > V_{threshold}$. There are 18 actions which are same with the actions listed in Fig. 2 and Table 2.

Table 5. Avatar behaviour.

Avatar behaviour	Action scores
Avatar moves towards goal	Score += 1
Avatar moves away from threat	Score += 2
Avatar is threatened	Score −= 5

5 Results

Figure 3 illustrates a sample tree which is used to extract game state features for Frogger. The input images are all extracted from down-scaled screen grabs. The return value of the tree is also an image, which is shown in Fig. 4. We search for the maximum and minimum points in the returned images. The two points are overlaid on the original screen grab. The maximum is marked with a green rectangle and the minimum point is marked with a red rectangle. In the Figure, we can see that the point having the maximum value is the desired home location of the frog. The point having the minimum value is the most dangerous car to the avatar.

Figure 5 shows the fitness of the best individual for these three games. We conduct 10 runs for each experiment. The average fitness of the best individual for each game is shown with a bold line. This algorithm is compared with our previous results using hand crafted features. Figure 6 shows the average best fitness for the two algorithms. The comparison is shown in Table 6. Mann Whitney U Test is used to compare the average best fitness in generation 9, 19 and 99. As shown in Table 7 the final results in generation 99 are not significantly different.

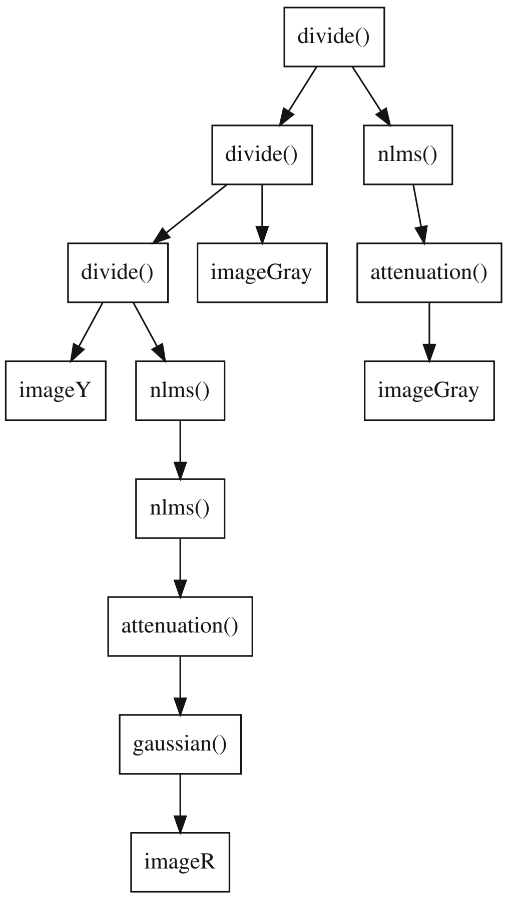


Fig. 3. A sample GP tree for extracting game state features of Frogger.

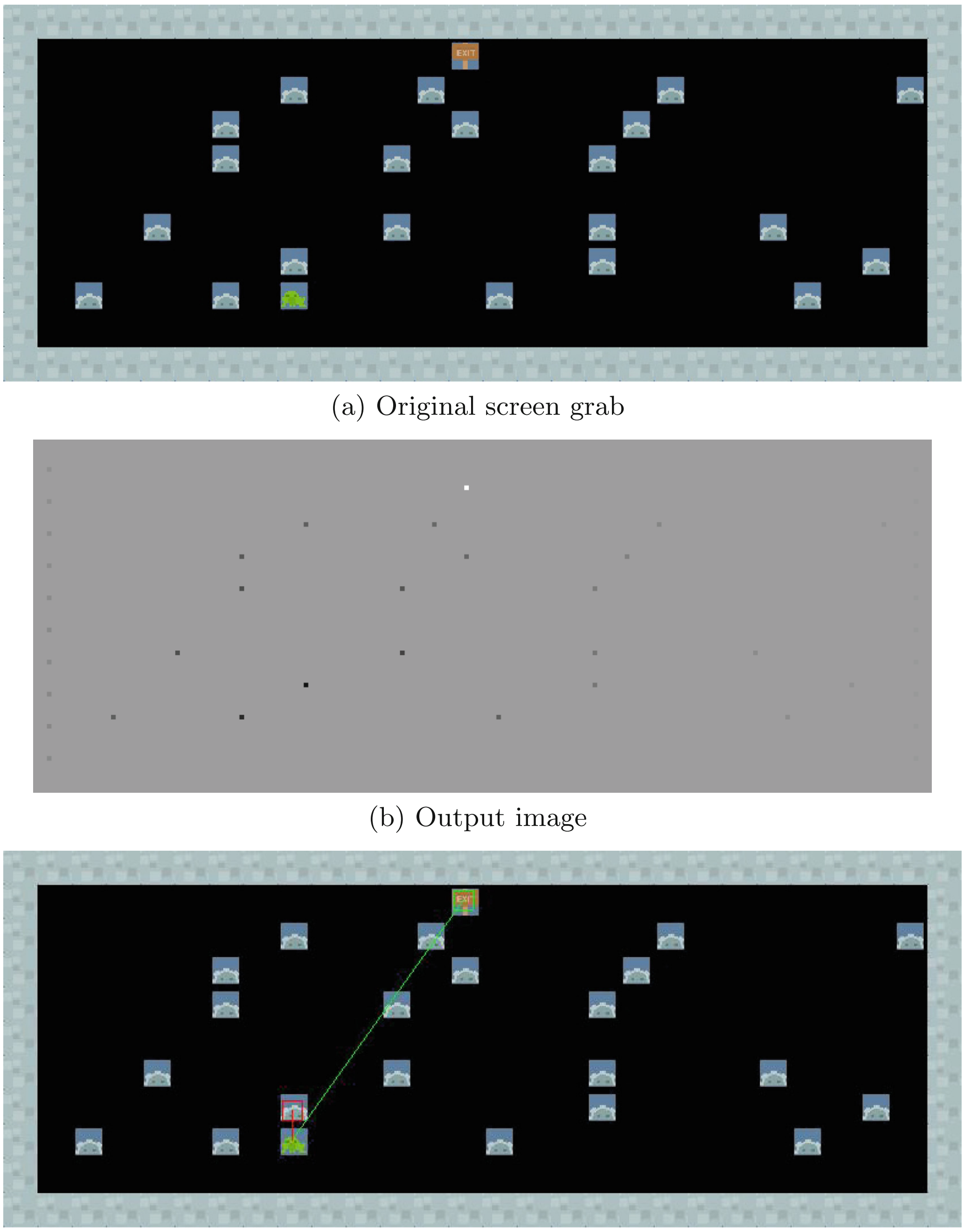
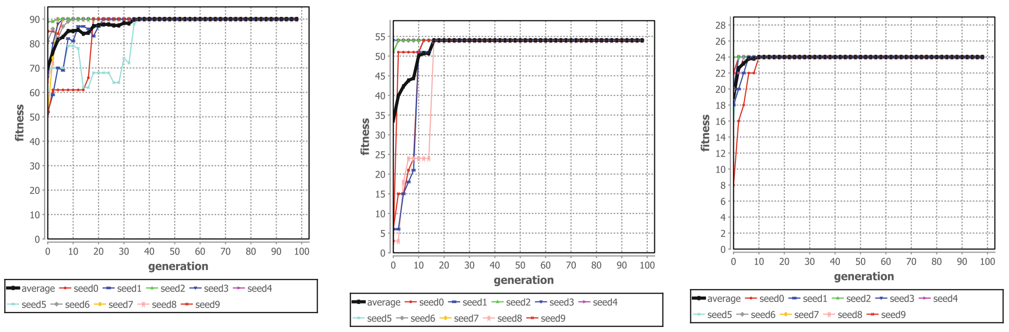


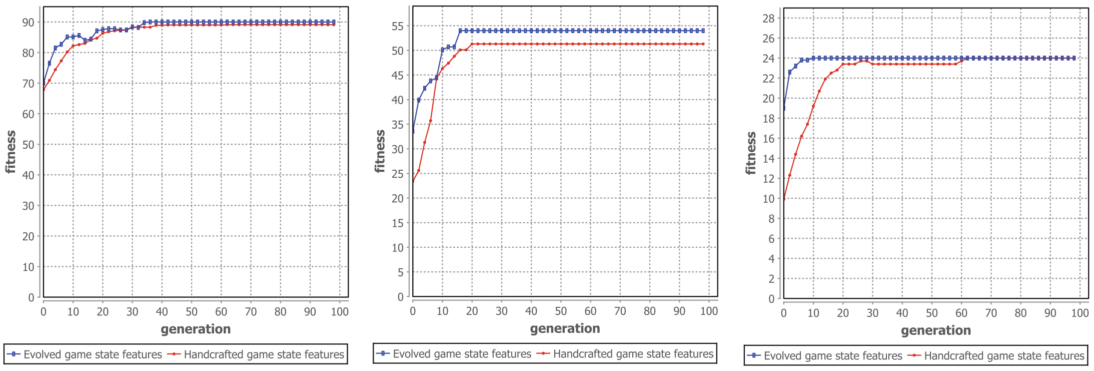
Fig. 4. Extracted game state features for Frogger. (a) the original screen grab. (b) output image from the sample GP tree. (c) maximum and minimum points overlaid on the original screen grab.

However, the algorithm using evolved game state visual features performs significantly better than the algorithm with hand crafter features for the games Frogger and Space Invaders. In other words, this algorithm is able to find good solutions faster than the original algorithm.



(a) Space Invaders (b) Frogger (c) Missile Command

Fig. 5. Best fitness for the three games. For each experiment, we perform 10 runs with different random seeds.



(a) Space Invaders (b) Frogger (c) Missile Command

Fig. 6. Comparison of the average best fitness between the two algorithms.

Table 6. Average scores over 10 runs obtained from the three games: Space invaders, Frogger, and Missile command. The maximum possible scores are: Space Invaders:90, Frogger:54 and Missile Command:24.

Game	Average score	
	GP+Handcrafted features	Evolved features+Voting
Space invaders	89 ± 2.84	90 ± 0
Frogger	51.3 ± 8.54	54 ± 0
Missile command	24 ± 0	24 ± 0

Table 7. Comparison of the average best fitness using Mann Whitney U Test in generation 9, 19 and 99. f_E is the average best fitness for the algorithm using evolved game state features. f_H is the average best fitness for the algorithm using handcrafted game state features.

Games	Hypothesis	p value		
		Gen = 9	Gen = 19	Gen = 99
Space Invaders	$H_o : f_E = f_H, H_1 : f_E > f_H$	0.0024	0.36	0.35
Frogger	$H_o : f_E = f_H, H_1 : f_E > f_H$	0.02	0.22	0.36
Missile Command	$H_o : f_E = f_H, H_1 : f_E > f_H$	0.42	0.38	0.5

6 Conclusion

In this work, we have used genetic programming to evolve the game state features from the raw pixels. The input image is provided as individual color channels. These color channels are processed to generate one output image. From the output image, two locations (maximum response and minimum response) are extracted. These locations correspond to a desired goal position and a position where a possible threat is located. A voting method is used to generate a movement action for the avatar from these two locations. Three different video games are used to evaluate our algorithm. The results show that this algorithm is able to find a good strategy faster when compared to an algorithm using hand crafted features.

References

1. Bellemare, M., Naddaf, Y., Veness, J., Bowling, M.: The arcade learning environment: an evaluation platform for general agents. J. Artif. Intell. Res. **47**, 253–279 (2012)
2. Perez, D., Samothrakis, S., Togelius, J., Schaul, T., Lucas, S.: GVG-AI Competition. http://www.gvgai.net/index.php
3. Finnsson, H., Bjornsson, Y.: Simulation-based approach to general game playing. In: Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, pp. 259–264 (2008)
4. Geneserech, M., Love, N.: General game playing: overview of the AAAI competition. AI Mag. **26**, 62–72 (2005)
5. Guo, X., Singh, S., Lee, H., Lewis, R., Wang, X.: Deep learning for real-time atari game play using offline monte-carlo tree search planning. Adv. Neural Inf. Process. Syst. **27**, 3338–3346 (2014)
6. Hausknecht, M., Khandelwal, P., Miikkulainen, R., Stone, P.: HyperNEAT-GGP: a HyperNEAT-based atari general game player. In: Genetic and Evolutionary Computation Conference(GECCO) (2012)
7. Hausknecht, M., Lehman, J., Miikkulainen, R., Stone, P.: A neuroevolution approach to general atari game playing. IEEE Trans. Comput. Intell. AI Games **6**, 355–366 (2013)
8. Jia, B., Ebner, M., Schack, C.: A GP-based video game player. In: Genetic and Evolutionary Computation Conference(GECCO) (2015)
9. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press, Cambridge (1992)
10. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. The MIT Press, Cambridge (1994)
11. Luke, S.: The ECJ Owner's Manual, 22nd edn. (2014)
12. Campbell, M., Hoane, A.J., Hsu, F.H.: Deep blue. Artif. Intell. **134**, 57–83 (2002)
13. Mehat, J., Cazenave, T.: Monte-Carlo Tree Search for General Game Playing. Technical report, LIASD, Dept. Informatique, Université Paris 8 (2008)
14. Mehat, J., Cazenave, T.: Combining UCT and nested monte-carlo search for single-player general game playing. IEEE Trans. Comput. Intell. AI Games **2**(4), 225–228 (2010)

15. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidieland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D.: Human-level control through deep reinforcement learning. Nature **518**, 529–533 (2015)
16. Naddaf, Y.: Game-independent AI agents for playing atari 2600 console games. Master's thesis, University of Alberta (2010)
17. Perez, D., Samothrakis, S., Lucas, S.: Knowledge-based fast evolutionary MCTS for general video game playing. In: Proceedings of IEEE Conference on Computational Intelligence and Games, pp. 68–75 (2014)
18. Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D.: Mastering the game of go with deep neural networks and tree search. Nature **529**, 484–489 (2016)