Playing Super Hexagon with Convolutional Neural Networks (Milestone)

Jason Lewis
Stanford University
JLewis8@stanford.edu

Abstract

In this paper, I describe my approach of using convolutional neural networks (ConvNets) to play the game Super Hexagon by Terry Cavanagh. This game is visually minimalistic, only requires two inputs (left and right), and only contains six levels that each takes only 60 seconds to complete. However, it is known for being very difficult to play and even completing level 1 can be considered a challenge. My approach involves using a polar coordinate representation of a screenshot of the game as an input to a ConvNet that predicts the presence of players and walls for a 40 by 40 grid of subimages. This output is further processed by a human-coded decision making process that decides the current action of the player (left, right, or neutral). My final system has completed levels 1 and 2 and has made significant progress on levels 3 and 4.

1. Introduction

1.1. Super Hexagon

Super Hexagon is a video game released by independent game developer Terry Cavanagh in 2012. The gameplay involves rotating a small triangle clockwise (right) or counterclockwise (left) around the center of the screen to dodge a series of incoming walls. A wall hitting the player triangle triggers a game over, which resets the timer. The player completes a level by surviving for 60 seconds. The game contains only six levels of differing speeds, color schemes, and wall patterns (except that levels 4, 5, and 6 contain similar wall patterns to levels 1, 2, and 3 respectively).

To put into perspective how difficult the game is, the difficulty names of the six levels are, in order: Hard, Harder, Hardest, Hardester, Hardestest, and Hardestestest. One particular aspect of Super Hexagon that can make it difficult for humans is that the game screen constantly rotates independently of the player's movement and the speed and direction of this rotation changes frequently. However, this aspect of the game would not have an effect on a computer player that decides the game input based only on the current frame.

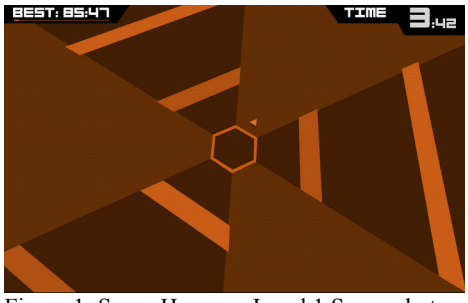

Figure 1: Super Hexagon Level 1 Screenshot

1.2. System Overview

I am attempting to use a ConvNet as the central component of an automated system (HexaBot) that can play Super Hexagon. HexaBot works by taking a screenshot image of the game and constructing a polar coordinate representation of the image. The ConvNet (HexaNet) takes this input image and returns as output the scores of three classes for a series of small subimages within the input image. These three classes are: player, wall, and background. From these scores, the player uses some decision making process to determine the best game input, which can be left, right, or neutral (no input).

2. Previous Works

2.1. Previous Super Hexagon AI

There haven't been many well documented attempts to create programs to play Super Hexagon. The best documented attempt I found to play Super Hexagon is from the website Cracked Open Mind [8]. Their approach involves binarizing a screenshot to separate the player/walls from the background and using OpenCV methods to detect and categorize shapes. Without any supervised learning techniques (as far as I can tell), their program is capable of beating all six levels of Super Hexagon.

2.2. Playing Video Games with Neural Networks

While I have not been able to find any prior work of applying neural networks to Super Hexagon, I have found examples of applying neural networks to playing other video games.

Mnih et al. trained a ConvNet with a variant of Q-learning to come up with control policies for playing various Atari games [2]. Their ConvNet (which they call a Deep Q-Network, or DQN) takes in the raw pixel data from their selected game and extracts high-level features which are used to determine from the output layer the action input for the game. Irwan Bello and Yegor Tkachenko utilizes this Q-learning process to play the Atari game Breakout [1].

Youtube user SethBling demonstrates his program MarI/O, which uses neural networks and was trained using genetic algorithms, to play Super Mario World [5]. MarI/O is based on the method NeuroEvolution of Augmenting Topologies (NEAT) developed by Kenneth Stanley and Risto Miikkulainen [6]. The output of the network determines which game inputs to activate for the given frame.

2.3. Object Localization with ConvNets

Both of the previous examples use neural networks to give as output the action input for the game. My approach to playing Super Hexagon with ConvNets is more closely related to object localization

The OverFeat model [4] is based on a framework of using ConvNets to predict classes at various locations and scales, and recombining these classifications to predict multiple bounding boxes and classes within an image. In their ConvNet, the images is passed through a series of convolutional and max pool layers to generate features. The final fully connected layers are applied in a sliding window over the output of the last convolutional layer to predict classes for different locations and scales.

The ConvNet that I use for playing Super Hexagon can be thought of as a very simplified version of this kind of network, which only contains no fully connected layers and outputs class predictions at multiple locations and just one scale.

3. Image Dataset

A significant issue with applying any kind of supervised machine learning algorithm to playing Super Hexagon is that there is no readily available image dataset. It will be easier to explain the overall design of HexaBot if I first explain how I can obtain an image dataset for Super Hexagon.

An image dataset could potentially be generated by taking screenshots of the game screen while playing the game manually, but that alone would not be enough. For supervised learning methods, each image in the dataset needs to be labeled appropriately for the algorithm being used. Since my approach involves detecting the locations of players and walls, the labels for a single image are the bounding boxes for each wall and the player on the screen. If the dataset is comprised of in-game screenshots, the labeling process would have to be done manually for each image. This process would be time-consuming and error-prone.

3.1. Screenshot Generator

A much better alternative for generating the dataset is to use a separate program that can randomly generate an image that looks like a Super Hexagon screenshot. Since such a program would have access to knowledge of every polygon on screen, it would be able to output a list of player and wall bounding boxes along with each image. As it turns out, the visual style of the game is so simple and minimalistic that writing this program would potentially be easier and more consistent than manually labeling real screenshots. A nice benefit of this approach is that the program can generate as many labeled images as desired to create an image dataset of any size.

Figure 2: Real Screenshot (left) and Generated Screenshot (right)

I have written a program to generate imitation Super Hexagon screenshots in the Processing language [3]. Figure 2 contains a real in-game screenshot and an image generated by my program using parameters chosen to imitate the real screenshot as closely as possible. While this is not a perfect replication, it is close enough to show that images generated by my program are a close enough imitation to form a useful dataset. The parameters that determine the rotation and shear of the screen and placement of objects can be randomized to generate new imitation screenshots.

3.2. Polar Coordinate Representation

For the purpose of making screenshots easier for HexaBot to process, I decided to apply a transformation

that converts screenshots into a polar coordinate representation. Figure 3 contains two annotated representations of the same image. The right image is the screenshot converted into its polar coordinate representation, using the center as the origin and half the image height as the max radius.

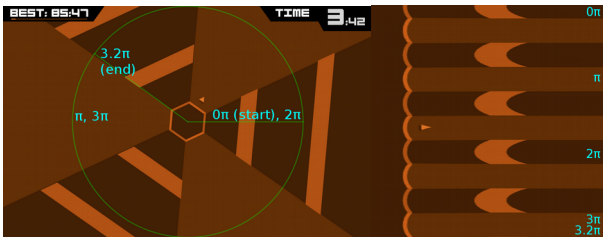

Figure 3: Annotated Representations of Screenshots in Cartesian (left) and Polar (right) Coordinates

I use a theta range of [0, 3.2π] radians instead of [0, 2π] radians, resulting in a portion of the bottom of the image being a copy of the top of the image. This change is to deal with the case of the player being cut in half at the top or bottom of the image. With this change, it is guaranteed that at least one whole player is in the image. This is important, since in any given screenshot, the system needs to form an accurate prediction for the player location in order to dodge incoming walls.

The main reason for using this polar representation is that in Cartesian coordinates, the sides of a wall are usually not axis-aligned. Since bounding boxes are axis-aligned, they will sometimes be an inaccurate representation of the set of pixels the wall occupies.

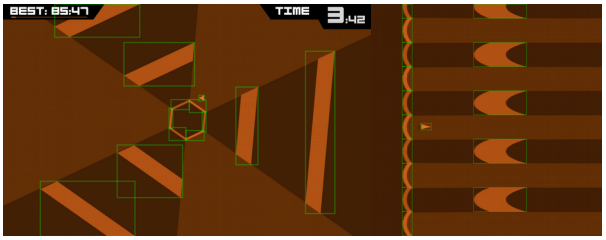

Figure 4: Axis-Aligned Boxes for Player and Walls

In Figure 4, some bounding boxes in the left image contain more background pixels than wall pixels. In the polar coordinate representation, the bounding boxes for walls become more consistent and reliable representations of the set of pixels the wall occupies. Specifically, the left edge of the box always represents the incoming side of the wall that the player needs to dodge.

Another reason for the polar coordinate representation is that decision making process at the end becomes simpler. Let's say we are given perfect bounding boxes for the player and every wall. In polar space, the walls move right to left instead of outward to inward, so dodging walls is just a matter of maximizing the distance to the closest wall directly to the right of the player.

3.3. Augmenting the player class

As I mentioned before, it is vital for the system to be able to predict the location of a player. In this context, there is an interesting issue with the images generated by the program. A single player triangle only takes up about 0.1% of the area within an image, whereas all walls take up around 15% of an image and the rest is the background. My ConvNet approach involves dividing the image into a series of small subimages, and it is incredibly rare for a player to appear in a subimage compared to a wall or background. This makes it difficult to train a ConvNet to learn how to recognize a player.

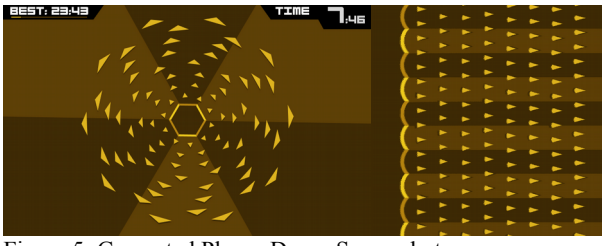

Figure 5: Generated Player-Dense Screenshot

To deal with this issue, I decided to generate some images for the dataset that do not look like they are from Super Hexagon, but contain a lot of player triangles. In Figure 5 we see an example of such an image in both Cartesian and polar representations. When my program generates an image dataset, half of the images will take this form. The end result is that over the entire image dataset, the probability of a subimage containing a player is about the same as a subimage containing a wall (about 15%).

4. HexaBot

4.1. Component Steps

The final system used to play Super Hexagon, called HexaBot, is made of a series of steps that takes a screenshot of the game window and processes it into a final game decision that will be continually executed until the next decision is made.

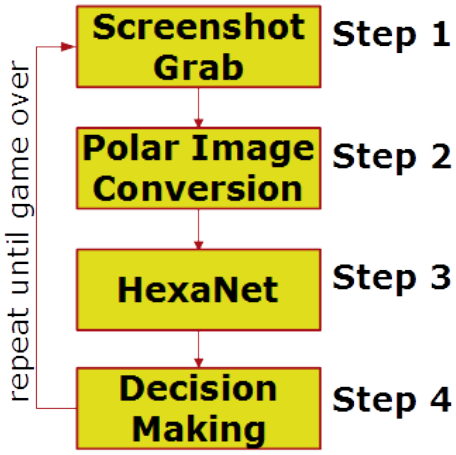

Figure 6: HexaBot Steps

Figure 7: HexaNet Architecture

The first step takes a screenshot of the game window. The default resolution of Super Hexagon in windowed mode is 768 by 480. The second step converts this screenshot into a 256 by 256 image that is the same polar coordinate representation used in the training image set. The third step is a convolutional neural network called HexaNet that processes the image. The fourth and final step processes the HexaNet output using a human-coded process that determines whether the player should move left, right, or neutral. This decision is executed in-game, overriding the decision made in the previous iteration.

4.2. HexaNet Architecture

The convolutional neural network HexaNet is the central and most important component of HexaBot. HexaNet takes as input a 256 by 256 image of the polar coordinate representation of a Super Hexagon screenshot (real or generated) and gives as outputs the probabilities of a subimage being of a player, wall, or background for a 40 by 40 grid of subimages. HexaNet was written in Python using a combination of Theano [7] and Lasagne [9].

Due to the specific architecture of HexaNet as seen in Figure 7, position (i, j) in the output (for $0 \leq i < 40$, $0 \leq j < 40$) contains class probabilities for a square subimage with opposite corners at pixels (6i, 6j) and (6i+21, 6j+21) in the input image.

To form a training set, the bounding box labels for the images in the dataset have to be converted to be compatible with the HexaNet output. For each of the 1600 subimages, if a subimage overlaps a bounding box of a player or wall, that subimage is assigned to the corresponding class.

In the architecture, the notation "CONV[X]-[Y]" denotes a convolutional layer with Y convolutional filters of size X by X by C, where C is from A x B x C, the dimension of the previous layer. The "Pad" is the amount of 0-padding used around the first two dimensions of the previous layer (padding is not used in this network). The "Stride" is how many spaces a filter moves in the A or B dimensions when applying the convolution. "MAX POOL2" denotes a max pooling layer with 2 by 2 filters of stride 2.

$$P(Y=k|x_{ij}) = \frac{\exp(x_{ij}^k)}{\sum_l \exp(x_{ij}^l)}$$

Figure 8: Softmax Function

Most layers apply additional nonlinearities, such as Relu (rectified linear unit), BatchNorm (batch normalization), Dropout, and Softmax. The softmax nonlinearity at the last layer converts all the scores at each subimage for each of the three classes into a probability distribution (where the sum of the class scores sum to 1).

5. Experiment

I was able to run the following experiment multiple times, where each experiment trains HexaNet to play a specific level or subset of levels of Super Hexagon. Using my Super Hexagon screenshot generator program, I generated a dataset of 500 labeled images in the polar coordinate representation. The color palette used to generate these images matches the color palette of the level or levels chosen for the experiment. Each image is labeled with the bounding boxes of each player and wall object. Using these bounding box labels, I generate for each image a 40 by 40 grid of class labels for each subimage as described in section 3.1.

The training set used is 480 images from the dataset, along with their respective class label grids. This leaves 20 images as the validation set. Under a specific set of hyperparameters, I trained a randomized instance of HexaNet using the backpropagation algorithm until convergence.

After training, the final network weights are used in HexaBot to play Super Hexagon.

5.1. Hyperparameters

My main goals during training was to get a network that achieves high classification accuracy (especially for the player class) on the validation set, and to get a set of layer 1 filters that visually look more like smooth patterns than random noise. For the latter goal, I created a visualizer for the layer 1 filters that normalizes the values of each filter to be viewable in RGB space. During hyperparameter selection, I took into account the layer 1 filters, the validation error, and training loss over time.

I used the Adam update rule for training network weights. For the learning rate, I settled on 5*10-4 and occasionally ran experiments as low as 1*10-5. At learning rates higher than these, the layer 1 filters had difficulty converging, even after the training loss is minimized.

In layers 1 and 2 where I was using dropout, I set the dropout parameter p to 0.25. If I set this value too high, I would often observe multiple filters converging to similar patterns visually. If I didn't use dropout at all, I would sometimes observe filters that don't converge to a useful pattern and are ignored by the rest of the network.

L2 regularization was the main tool for encouraging nice layer 1 filters. While higher regularization tended to produce nicer filters, too much regularization had a negative impact on validation error, which would peak quickly and then start decreasing. My solution was to use a different regularization for layer 1 than for the rest of the network. The default regularization amount was 0.01 for the network and a regularization amount of 1.0 for layer 1. This combination resulted in nice filters and low validation error.

5.2. Training on Level 1

Here, I report the results of running the experiment on training HexaNet on images that use a similar color palette to level 1 from Super Hexagon.

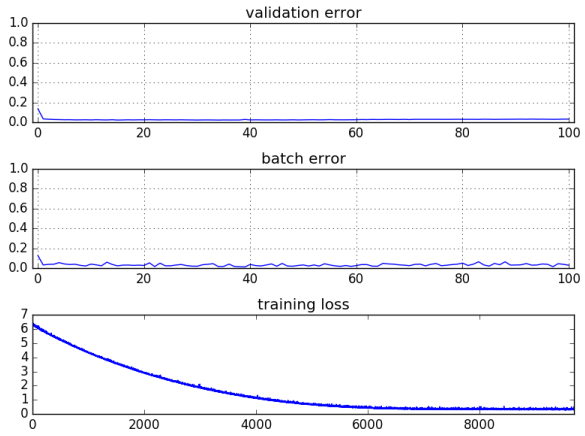

Figure 9: Error and Loss on Level 1 Dataset

The network achieves a very high validation accuracy very quickly. This is likely due to the fact that due to the minimalistic art style of the game, all the images in the dataset are fairly similar with regard to the shapes, textures, and colors present in any image. After just two epochs, the network achieves an overall 96% validation accuracy. The training loss (which includes softmax error and regularization loss) decreases smoothly over time and converges to a loss of around 0.35 by epoch 70.

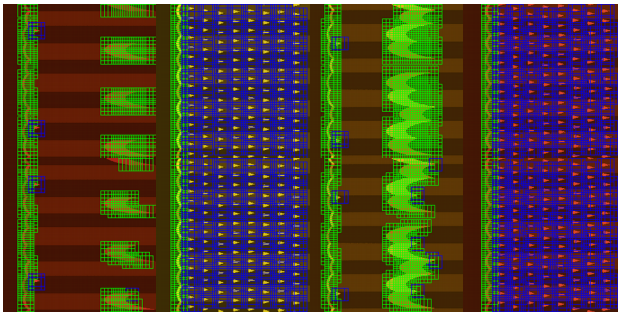
Figure 10: Four Images from Validation Set with True Labels (top) and Predicted Labels (bottom) for Bounding Boxes

In Figure 10, we see four labeled images from the validation set, with the true labels on the top row and the HexaNet predictions on the bottom row. In this visualization, a green box denotes a subimage that contains a wall and a blue box denotes a subimage that contains a player. An area with no box is part of the background.

Visually, the predictions appear to be mostly accurate except for the issue of parts of a wall occasionally being classified as part of a player. Due to the way the polar coordinate conversion warps the shape of the wall, the right side of the wall appears to have a “tail” that looks vaguely triangular. A subimage that contains a tail, when passed through the network with no context of its

surroundings, can understandably be mistaken for a player triangle.

		Predicted			
		Background	Wall	Player	Recall
Actual	Background	23716	57	154	0.991
	Wall	674	3165	57	0.812
	Player	135	0	4042	0.968
	Precision	0.967	0.982	0.950	

Table 1: Confusion Matrix for HexaNet on Level 1

In the confusion matrix in Table 1, we see how the final HexaNet performs on classifying subimages of images from the validation set. For the most part, the network does a very good job of detecting every class. We might have expected to see a large amount of error for mistaking a wall for a player as we saw before, but this is not the case. The biggest source of error is actually mistaking a wall for the background.

If we look back at the first and third validation images in Figure 10, we can see this happening at subimages located on the perimeter of a wall. The true labels were defined by the bounding boxes of the walls, which can result in some subimages on a perimeter being labeled as walls when they actually only contain background. These errors in the ground truth contribute to error in the accuracy of the network, but there aren't too many of these errors to prevent the network from learning how to accurately recognize walls and players.

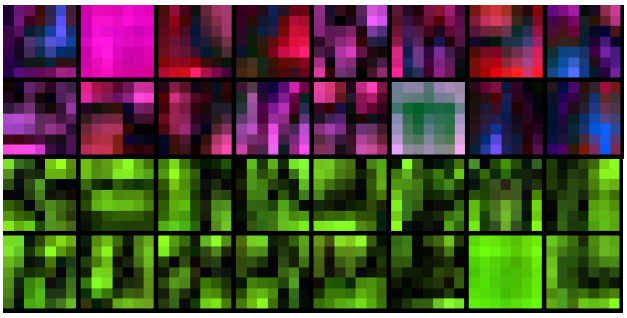

Figure 11: Layer 1 Filters Trained on Level 1 (top) and Level 2 (bottom)

The colors in the converged layer 1 filters change to match the color palette of the images in the training set. Level 1, which contains red and yellow colors, produced filters that are mostly red and magenta. Level 2, which contains a lot of green, produces filters that are dominantly green. In both cases, the filters contain smooth patterns.

5.3. Playing Level 1

Using a HexaNet trained on level 1 images, HexaBot was able to successfully complete level 1.

Trial	Time	Won
1	35.77	
2	61.15	X
3	61.67	X
4	60.78	X
5	61.03	X
6	21.98	
7	42.63	
8	62.28	X

Trial	Time	Won
9	64.15	X
10	26.95	
11	46.62	
12	63.58	X
13	35.57	
14	16.23	
15	62.20	X
16	60.72	X

Trial	Time	Won
17	63.17	X
18	60.85	X
19	62.83	X
20	61.20	X
21	49.70	
22	59.95	
23	44.18	
24	62.45	X
25	61.23	X

Table 2: Results of 25 Trials of Level 1

If debugging visualizations are turned off, HexaBot was able to operate at about 20 frames per second. With debugging visualizations turned on, this drops to about 15 frames per second. With debugging visualizations turned off, HexaBot played 25 trials of playing Super Hexagon at level 1, 15 of which it was able to complete the level by reaching a time of 60 seconds or more. This indicates that it is able to complete level 1 in about 60% of attempts. In Table 2, we see all the times achieved by HexaBot during its 25 trials.

When someone completes a level, the game keeps going until the player hits a wall. After 60 seconds of level 1, the game starts speeding up and the color palette changes to that of level 4. Since this particular experiment only trained HexaNet using images from level 1, this change of color palette confuses the network and makes it unable to extract information from the screenshot. This is why the player almost immediately fails after achieving 60 seconds.

On trials where HexaBot didn't complete the level, it achieved an average time of 37.96 seconds. Causes of a game over before 60 seconds can be attributed to shortcomings in the human-coded rules and errors in the HexaNet output, such as confusing the tail of a wall as the player.

5.4. Progress on Levels 1-5

I reran the same experiment for every level except level 6. The game actually doesn't let the player play levels 4, 5, and 6 until the player completes levels 1, 2, and 3 respectively. HexaBot wasn't able to complete level 3 yet, so level 6 is still locked. Table 3 contains the best times achieved by HexaBot in the first five levels. It was able to beat levels 1 and 2 and got at least half way through levels 3 and 4. The time of 69.86 in level 1 was achieved by a network that was trained on images from levels 1 and 4, so

that it would have some ability to keep playing after reacing 60 seconds.

Level	Best Time	Won
1	69.86	X
2	61.13	X
3	41.63	
4	54.71	
5	17.97	
6	N/A	

Table 3: HexaBot Best Times

6. Future Work

I plan to improve HexaBot to the point that it is able to complete all six levels. It has already shown promising progress in levels 3 and 4.

6.1. Reinforcement Learning

In my opinion, the most significant shortcoming of HexaBot is its reliance on human-coded rules to process the output from HexaNet. Once these rules are fixed, there is no room for improvement. If this component were to be replaced a decision making process based on reinforcement learning, the overall system would have the ability to improve itself. Potentially, this could result in a system that can play better than it could using the human-coded process.

Super Hexagon could be modeled as a Markov decision process (MDP) with an unknown transition model that could be trained to make decisions that maximize the reward from playing the game. In this MDP, the set of states can be all of the possible 40 by 40 grids that HexaNet can output. Alternatively, this state space can be simplified as the x distance from the player to the nearest wall at each of the 40 y locations in the HexaNet output, along with the most likely player y location. The set of actions in the MDP would still be left, right, and neutral. The reward in the MDP for a given frame is the distance between the player and the closest wall in the HexaNet output. A game over would have an inherent reward of 0 (or less if desired). This would encourage the reinforcement learning process to keep the game going as long as possible.

6.2. Input Classification Approach

When I started this project, my original plan was for the ConvNet to take a normal screenshot of Super Hexagon as an input and give as the output a single classification for the image as being left, right, or neutral. This approach would not include a separate decision making process since the ConvNet output would serve as the current decision for the game input.

One of the main reasons I did not try this approach in full scale is that it would have made the image dataset trickier to generate. If I were to use live screenshots for the dataset, I would have to label the actions manually. If I were to generate the screenshots with a program, I would have to write rules to determine the best action.

Another reason for avoid this approach is that I was unsure of whether a ConvNet would be able to extract enough information from the screenshot to reliably determine the best action. The network would have be able to notice the importance of the player triangle, detect its location, and determine its proximity to walls. However, it would be interesting to see how effective or ineffective this approach would be compared to the approach I actually used.

7. Conclusion

In this paper, I presented a system called HexaBot that utilizes a convolutional neural network called HexaNet to play the video game Super Hexagon. To train HexaNet, I used a labeled image dataset that was created using a program I wrote that generates images that look like in-game screenshots. These images were transformed into a polar coordinate representation that was easier to process. HexaNet predicts the presence of players and walls for a 40 by 40 grid of subimages in the original input image. This output is processed by a human-coded process that decides the action of the player. This system was eventually able to complete levels 1 and 2 of Super Hexagon. It is possible that by replacing the human-coded process with a process that uses reinforcement learning, the system can improve beyond its current capabilities and eventually complete all six levels.

This is a video of HexaBot completing level 1: https://vimeo.com/158433051.

This is a video of HexaBot completing level 2: https://vimeo.com/158855274.

If I make the source code for this project available at some point in the future, I will make it available at the GitHub page: https://github.com/Mathsvlog/HexaBot.

8. References

[1] Bello, Irwan, and Yegor Tkachenko. "Learning Control Policies from High-Dimensional Visual Inputs." Stanford CS231N (2015).

[2] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).

[3] "Processing." *Processing*. Web. 13 Mar. 2016. <https://processing.org/>.

[4] Sermanet, Pierre, et al. "Overfeat: Integrated recognition, localization and detection using convolutional networks." *arXiv preprint arXiv:1312.6229* (2013).

[5] SethBling. "MarI/O - Machine Learning for Video Games." Online video clip. YouTube. <https://www.youtube.com/watch?v=qv6UVOQ0F44>.

[6] Stanley, Kenneth O., and Risto Miikkulainen. "Evolving neural networks through augmenting topologies." *Evolutionary computation* 10.2 (2002): 99-127.

[7] "Theano." *Welcome — Theano 0.7 Documentation*. Web. 13 Mar. 2016. <http://deeplearning.net/software/theano/>.

[8] Trimaille, Valentin. "Super Hexagon Bot." *Cracked Open Mind*. 2015. Web. 13 Mar. 2016. <http://crackedopenmind.com/portfolio/super-hexagon-bot/>.

[9] "Welcome to Lasagne." *Welcome to Lasagne — Lasagne 0.2.dev1 Documentation*. Web. 13 Mar. 2016. <http://lasagne.readthedocs.org>.