Object Detection and Supervised Learning: An AI Plays Downwell

Edwin Pua
Department of Computer Science
Stanford University
puaedwin@stanford.edu

Abstract

In this project, I discuss how I built an AI agent to play an indie game named Downwell. To do this, I use the outputs of a Yolov5 object detection network as inputs to a many-to-one RNN to output the correct keystroke on a given frame. This neural network architecture was able to create an AI that successfully beat the first level of Downwell. This kind of problem - responding to object detection - has many practical applications, like self-driving cars.

1 Introduction

Downwell is a vertically-oriented arcade platformer about falling down a well. Armed only with their trusty pair of Gunboots, the main character dives headfirst into the well to save their cat at the bottom.

In Downwell, there are only three buttons - left and right arrow keys to move, and space bar to shoot. The goal of this project is to create an AI agent that can adopt "human behaviors" while playing Downwell, such as bouncing on enemies, shooting enemies, walking off platforms, and making it further into the well.

To do this, my goal is to use a supervised learning approach. Given the vectorized bounding boxes of the past sixty frames, the AI agent will be able to output which combination of left, right, and space to press on the current frame, represented by a 1x3 vector ['left', 'right', 'space'].

2 What Makes This Interesting?

What makes Downwell interesting compared to similar arcade games is its more modern complexity. Most older arcade games have only a few elements to consider at a time. For example, in Snake, only the apple and the snake are objects that require detection. In Downwell, however, there exists far more elements on screen at a given time - a vast array of enemies, blocks, and platform layouts. To train a neural network to play Downwell, a lot of data and hidden units will be necessary.

In the context of deep learning as a whole, this project is interesting because many urgent computer vision problems have no intermediary software such as an API, instead relying on reading the environment at a moment's notice. As such, identifying enemies or spikes on the screen at a given frame and outputting the correct keystroke is analogous to many practical applications, like a self-driving car identifying stop signs or pedestrians on a road and outputting whether or not to rotate the wheel, press the gas, or slam the brake.

3 Related work

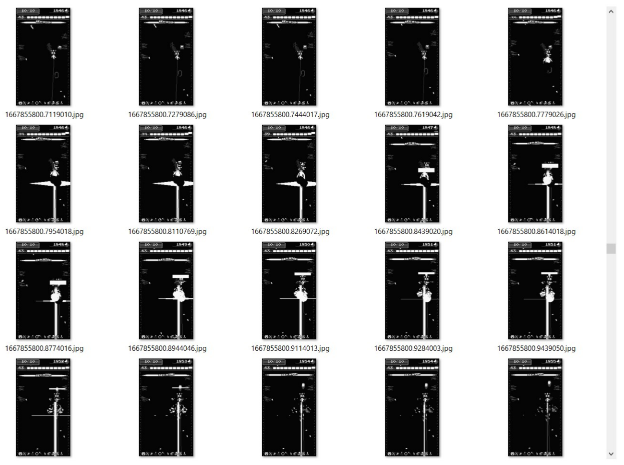
Initially, my plan was to employ reinforcement learning. For example, the previous CS230 project "SnakeAI" by Thomas Jiang and Claire Mai would've served as a good reference point.[1] This project uses reinforcement learning using distance reward (i.e. negative reward when the snake moves away from the apple, positive reward when the snake moves toward the apple). I could adopt a similar approach (e.g. negative reward when taking damage, positive reward when moving downwards, etc.). Essentially, this approach would mean maximizing reward instead of minimizing loss.

The problem with this approach is that in order to learn a complex game like Downwell effectively, the game needs have in-game API support in order to run expansive simulations and learn tractably. The Dota 2 AI, OpenAI Five, for instance, played against itself for over 10000 years, which would not have been possible without Dota's API.[2] Downwell has no API.

Therefore, the paper that served as the most valuable reference point for me is the past CS230 project "Counter-Strike Self-play AI Agent with Object Detection and Imitation Training" by Chenyang Dai. [3]

There are many parallels between my project and this one. For one, both projects have minimal pre-existing datasets - I will have to personally record gameplay and hand-label the dataset myself. Additionally, I also plan to have two networks interact with each other - an object detection network and movement network that collaborate to play the game. Finally, this approach relies primarily on supervised learning as opposed to the more popular reinforcement learning for games in machine learning.

4 Dataset



This project uses two datasets: one for the object detection network and another for the movement network. Since there is virtually no dataset online available for Downwell, most of this data was collected myself.

The object detection dataset consists of 1906 images, captured using a Python script that takes screenshots of the screen at set intervals. I then hand-annotated using Roboflow, a labeling tool for Yolo object detection. [4] Here, for each image, I created bounding boxes around the player, bounceable enemies, non-bounceable enemies, breakable blocks, platforms, and other objects of interest. Then, I applied data augmentation, resizing the 566x317 images to 640x640 and applying gray-scale, flip, and noise, with the following justifications:

- Gray-scale: there are only three colors in Downwell - black, white, and red. On gray-scale, red is turned into gray, so there is no tradeoff in information when grayscale is applied.
- Flip: Enemies, blocks, and the player can all appear in-game with a "facing right" orientation and "facing left" orientation. The same cannot be said of, say, rotation augmentation - the player will never appear rotated 90 degrees, for example.

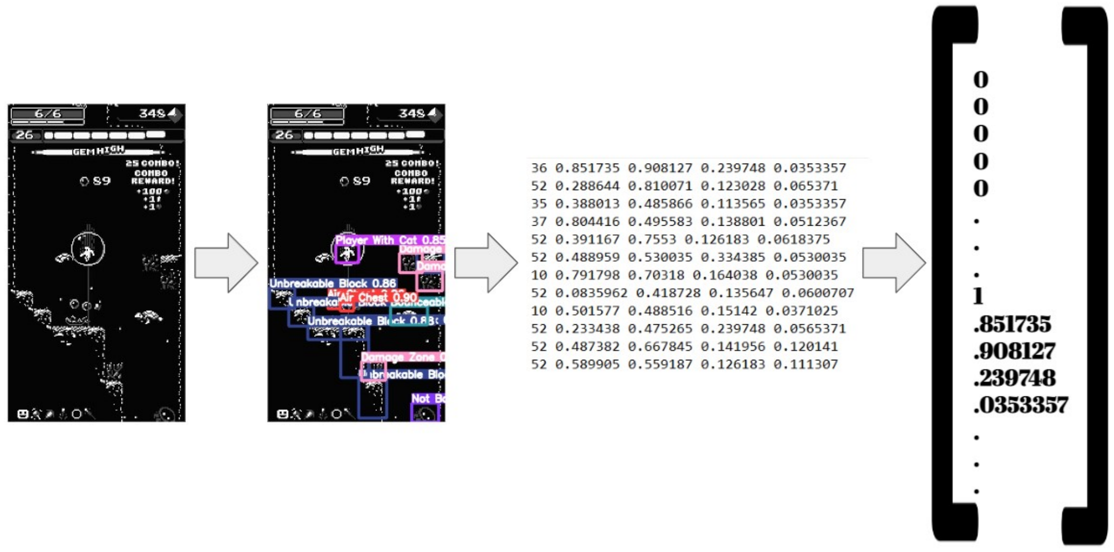
- Noise: When enemies in Downwell are killed, they release gems, which for the purposes for this project are noise. Additionally, the backgrounds of Downwell often have various particle effects. Thus, since noise will be such a natural part of the game anyway, it helps to train Yolo on noisy inputs.

Additionally, I noticed that sometimes that Yolov5 would not pick up more niche objects, such as upgrades, because they were underrepresented in the dataset. So I decided to take screenshots of more uncommon objects in order to balance the data.

The movement network dataset consists of 416656 contiguous gameplay frames (X), captured with the same screenshot Python script alluded to above, as well as the corresponding keyboard inputs pressed at each frame (y), captured using a Python script that records timestamps of input presses. These keyboard inputs are represented as a 1x3 vector, ['left', 'right', 'space']. For instance, a vector of [1, 0, 1] indicates that left and space were pressed on that frame.

5 Network Architecture

I transformed the gameplay frames (X) into an input that would be readable by my neural network. First, I trained Yolov5 on my object detection dataset, and with these new weights, I obtained the bounding boxes on all the gameplay frames. The bounding boxes of each frame are saved to a .txt file, in the format "class, xcenter, ycenter, length, width". These bounding boxes are then transformed into a 1x650 vector with the format indicated in the image below:



```
36 0.851735 0.908127 0.239748 0.0353357
52 0.288644 0.810071 0.123028 0.065371
35 0.388013 0.485866 0.113565 0.0353357
37 0.804416 0.495583 0.138801 0.0512367
52 0.391167 0.7553 0.126183 0.0618375
52 0.488959 0.530035 0.334385 0.0530035
10 0.791798 0.70318 0.164038 0.0530035
52 0.0835962 0.418728 0.135647 0.0600707
10 0.501577 0.488516 0.15142 0.0371025
52 0.233438 0.475265 0.239748 0.0565371
52 0.487382 0.667845 0.141956 0.120141
52 0.589905 0.559187 0.126183 0.111307
```

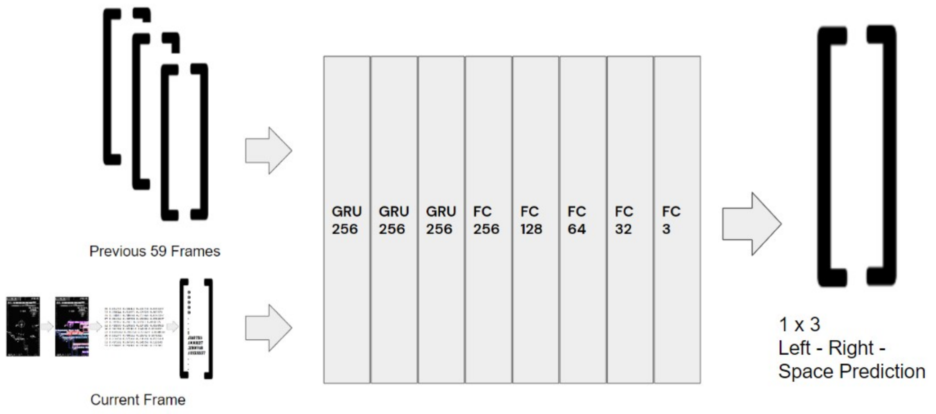
```
0
0
0
0
0
.
.
.
1
.851735
.908127
.239748
.0353357
.
.
.
```

The object of class 36 has its bounding box reserved into the vector as vector[36 x 5] = 1, vector[36 x 5 + 1] = xcenter, vector[36 x 5 + 2] = ycenter, vector[36 x 5 + 3] = length, and vector[36 x 5 + 4] = width. (vector[36 x 5] = 1 signals to the network that this object is present in the image).

No object of class 34 is present in the image, so vector[34 x 5] = 0.

Objects of class 52, Unbreakable Blocks, can appear multiple times in an image, so multiple slots of the vector are allotted for Unbreakable Blocks (e.g. 52x5 through 67x5) and similar classes. It is important that each class has its own designated spot on the vector so that the neural network can start to learn that class' properties.

These vectors are then grouped into contiguous 60-frame slices that can now be passed into the following neural network:



This neural network follows a many-to-one architecture, consisting of 3 GRU layers (256 hidden units each) and 5 FC layers that decrease in the number of hidden units on each layer. GRU layers are used here because they are great for time-series data, as we are passing in the input of shape (batch-size=32, time-steps=60, num-of-features=650). And because multiple outputs are possible at the same time (e.g. "left" and "space"), the last FC layer uses a multi-class sigmoid to predict the probability that a given key should be pressed, while the other FC layers use ReLU. Because we are using sigmoid as the output, the loss function is a standard binary crossentropy loss:

$$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

where M is the number of classes (3 in this case), p (y_{pred}) is the predicted probability observation o is of class c (e.g. [.8354, .1243, .6179] for the 3 classes), and y (y_{true}) is 0 or 1 depending on whether or not class label c is the correct classification of o (e.g. [1, 0, 1] for the 3 classes).

I chose not to employ transfer learning and built the network from scratch. This is because GRUs do not seem like the best candidate for transfer learning. For instance, weights trained on English sentences cannot be transferred for Spanish sentences, as the set of possible words completely changes. The entire RNN weights generally have to be retrained from scratch.

6 Experiments and Results

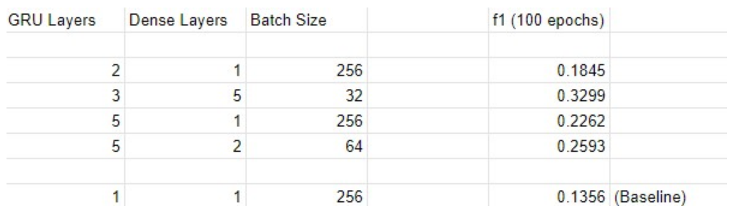
Here is a video of the AI agent beating the first level of Downwell: https://www.youtube.com/watch?v=-v3yHr3YUpI

Here is the same video, but with bounding boxes included, giving insight into how the computer sees Downwell: https://www.youtube.com/watch?v=VjBtvJXVRnM

Notice the way agent is successfully able to identify enemies and eliminate them by bouncing on them or shooting them. It is also able to recognize that it must walk off platforms in order to progress and avoid enemies from above.

There are a few problems with the object detection network, as shown by the video. Sometimes, the objects will flicker in and out of the computer's vision - other times, distinct objects get misclassified. This can be fixed by training the object detection network on more data, making it more robust.

The following are the different sets of hyperparameters I tried:



GRU Layers	Dense Layers	Batch Size		f1 (100 epochs)	
2	1	256		0.1845	
3	5	32		0.3299	
5	1	256		0.2262	
5	2	64		0.2593	
1	1	256		0.1356	(Baseline)

Quantitatively, I used an f1 metric to evaluate the hyperparameter choices and to strike a balance between precision and recall (TP, FP, FN = number of true positives, false positives, false negatives:

$$Precision = \frac{TP}{TP + FP}$$
$$Recall = \frac{TP}{TP + FN}$$
$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2 * TP + FP + FN}$$

Qualitatively, I looked at how well each model did after booting up a game of Downwell - is it identifying enemies? How far does it get before dying?

In both cases, the model with the most FC layers does the best. While all the models had the player get stuck frequently, this model produced the cleanest runs and maximized f1.

Initially, the baseline model would not even minimize training loss - training loss would stay constant between each epoch. Thus, I decided to increase the network size. Then, all the models quickly began to memorize the training dataset, hyperfocusing on outliers and unimportant tendencies as the validation loss stayed constant. Thus, for regularization, I decided to add dropout=0.5 to each GRU layer and a batch norm step after the last GRU layer and each FC layer.

Another interesting hyperparameter I had to tune was a custom hyperparameter called the noise threshold. Consider the following scenario: your character is in the middle of moving right, but a slime appears on the left side. Because of the way GRUs work, if the character is moving right, then the model will be more likely to predict that the character will keep moving right rather than spontaneously change direction. However, because the slime has appeared on the left side, the character will also want to move left to bounce on it. As a result, the following pattern in predicted key presses can occur:

[.235, .764, .122]

[.356, .719, .144]

[.443, .651, .098]

[.599, .486, .172]

Based on the way that the leftmost value is increasing, it is clear that the algorithm now has gained the urge to start pressing left due to a detected object. However, if we were to press a given key if the class probability is greater than 0.5, then in this example, the character would start moving left a little too late. To rectify this, I press down a key if a class probability increases by an amount greater than noise threshold. After sampling values between 0.001 and 0.1, I found that .02 for the noise threshold works best qualitatively.

7 Conclusion/Future Work

Personally, I am ecstatic that I managed to build an AI agent that could play Downwell to some extent. This feels like just the start - I have many ideas I would like to experiment with, such as changing the feature vectors to encode objects' distance from player as opposed to object position and size. There are many improvements I can make to my existing infrastructure as well - for example, simply training the object detection network on more data would help prevent the "bounding box flickering" in the AI agent demo and help the movement network determine the correct key presses more accurately.

8 Appendix

Screenshots were taken using the MSS library.[5]

Inputs are recorded using the python keyboard package.[6]

The AI sends keyboard inputs based on predictions using the pyautogui module.[7]

9 References

[1] Jiang, Thomas & Mai, Claire (2021) "SnakeAI"

http://cs230.stanford.edu/projects_fall_2021/reports/103085287.pdf

[2] OpenAI Five https://openai.com/five/

[3] Dai, Chenyang (2021) "Counter-Strike Self-play AI Agent with Object Detection and Imitation Training" http://cs230.stanford.edu/projects_fall_2021/reports/102988723.pdf

[4] Roboflow https://roboflow.com/

[5] MSS https://github.com/BoboTiG/python-mss

[6] Python keyboard https://github.com/boppreh/keyboard

[7] PyAutoGUI https://github.com/asweigart/pyautogui