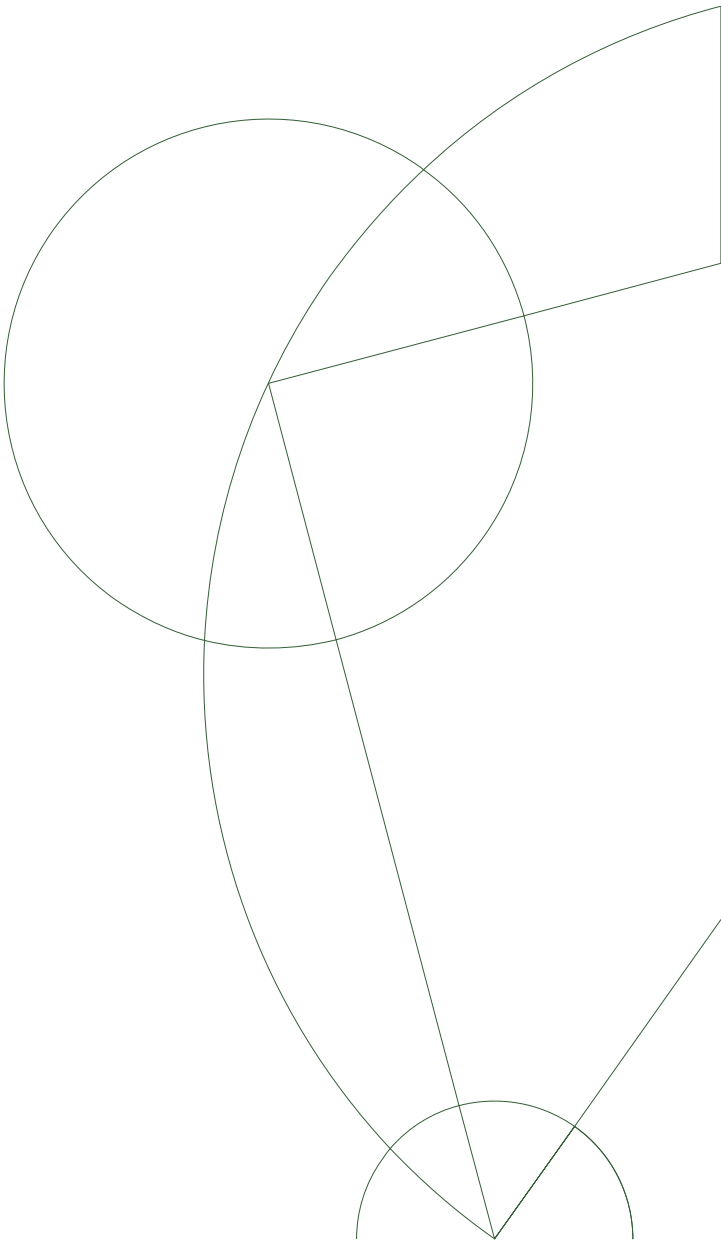
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

MSc thesis

Thor Hannibal Valsgaard - xjz106

Using Behavioural Cloning to Slay the Spire



Academic advisor: Silas Nyboe Ørting
Submitted: May 31, 2023

Using Behavioural Cloning to Slay the Spire

Thor Hannibal Valsgaard - xjz106
DIKU, Department of Computer Science,
University of Copenhagen, Denmark

May 31, 2023

MSc thesis

Author: Thor Hannibal Valsgaard - xjz106

Affiliation: DIKU, Department of Computer Science,

University of Copenhagen, Denmark

Title: Using Behavioural Cloning to Slay the Spire /

Academic advisor: Silas Nyboe Ørting

Submitted: May 31, 2023

Abstract

Slay the Spire is a turn-based video game. There are a lot of videos on YouTube of competent Slay the Spire players. In this work I first present the game and the existing programs that can play the game without human input. I then present my program, which can analyse videos of people playing Slay the Spire and find out which actions were taken in which game states. I then use that program on a set of videos from one of the best players and train a model on the data derived from those videos. That model is then used to play the game independently. The model was able to beat the game, but it was not better than any of the other similar programs. I then tried to use reinforcement learning to train a similar model. That model clearly performed better than random guessing, but it was still worse than the first one.

Dansk Resumé

Slay the Spire er et turbaseret videospil. Der er mange videoer på YouTube af kompetente Slay the Spire-spillere. I denne report præsenterer jeg først spillet og de eksisterende programmer der kan spille spillet uden menneskeligt input. Jeg præsenterer derefter mit program, som kan analysere videoer af folk der spiller Slay the Spire og finde ud af hvilke handlinger der blev udført i hvilke spiltilstande. Jeg bruger så det program på et sæt af videoer fra en af de bedste spillere og træner en model på dataet trækket ud fra de videoer. Den model er så brugt på at spille spillet af sig selv. Modellen kunne godt vinde spillet, men den var ikke bedre end nogen af de andre lignende programmer. Jeg prøvede derefter at bruge reinforcement learning til at træne en lignende model. Den model klarede sig tydeligt bedre end tilfældigt gætteri, men den var stadigvæk værre end den første.

Contents

Introduction

1

The internet has an absurd amount of publicly available video data, and videos of people playing video games are no exception. Some of those videos are even made by some of the games' best players. Therefore, it would be interesting to try to use that massive data source to teach an AI how to play a specific game. This is done with a technique called imitation learning. There are different variants of imitation learning, but the general idea is to train a model by making it imitate one or more experts at the task in question. We will be focusing on Behavioural Cloning (BC), which trains a model on expert demonstrations in a supervised fashion. BC has been used together with video games before. There are, for example, numerous cases of researchers using BC to make agents that can play Atari games [13].

The game I have picked out for this project is called Slay the Spire. It is turn-based, so it is very simple to represent actions. It is complicated enough that it is not simple to program a good policy by hand. It is also not viable to reimplement the entire game yourself, which would facilitate using reinforcement learning (RL) to derive a good policy. Reimplementing the game would be extremely tedious, and it would probably still result in an inaccurate product. And doing RL with the game in its original form would be very slow because you would have to wait on the game's animations. Therefore, it would make sense to use this massive collection of expert data to train a model using BC instead. And if we wanted to do RL, we can at least use BC to pre-train the model to a point that would be much slower to reach with RL. Though, there are a couple of mods that massively speed up the game, possibly making RL viable after all. The first one is Spirecomm [9], which allows a program to communicate with the game directly. Our program would be able to get the game state without having to wait for the animations to finish, take a screenshot, and then analyse it. The second one is the SuperFastMode mod [11], which drastically speeds up the animations in the game.

There are already a couple of existing programs that can play Slay the Spire [9] [1] [12]. They mostly use simple policies programmed manually, but some of them do simulate the battles so they can do tree-searches of the possible action sequences. I do not know of any similar programs that rely on machine learning.

There will be four parts to this project. The first part is to make a program that can take a video of someone playing Slay the Spire and find the frames where actions are taken. The program is then supposed to identify the action as well as what state the game was in when the action was taken. The second part takes all these state-action pairs and uses them to train a neural network to predict the action when given a game state. The idea is that if the network can predict the move a good player would make, it would be able to make good moves itself. The third part is to put the network into Slay the Spire and have it play the game on its own. The fourth part is to try to train a similar network, using RL, and compare it with the BC model.

Background

2

2.1 Slay the Spire

Slay the Spire is a video game released in 2019. In the game you play as one of four characters and make your way through several floors to defeat the final boss. The floors are split into three acts which each contain a boss at the end. There is also a hidden fourth act, which is outside the scope of the project. Each floor will either contain a battle, a resting site, a treasure chest, a merchant, or a multiple choice event. You choose between multiple paths going through the floors, which will lead to different types of floors.

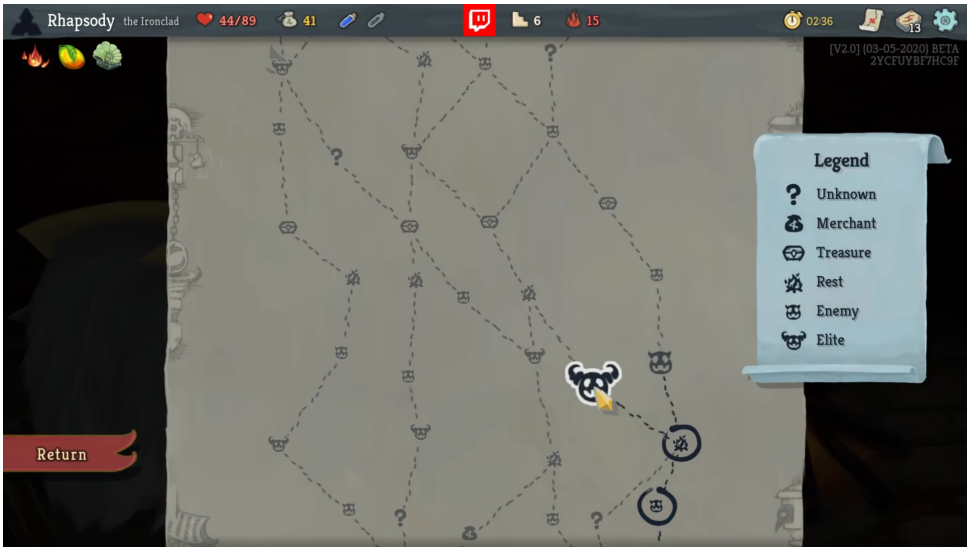


Figure 2.1: The screen for choosing which path to follow and which floors it will lead to.

When in combat, you are dealt a hand of cards which will hurt enemies, defend you, or have other effects. Each card has an associated energy cost when played. You start each turn with a limited amount of energy. When you have run out of energy, you end your turn, the enemies make their moves, and you get a new hand and replenished energy. You can also obtain potions, which can be used once for certain effects based on the types of potion.

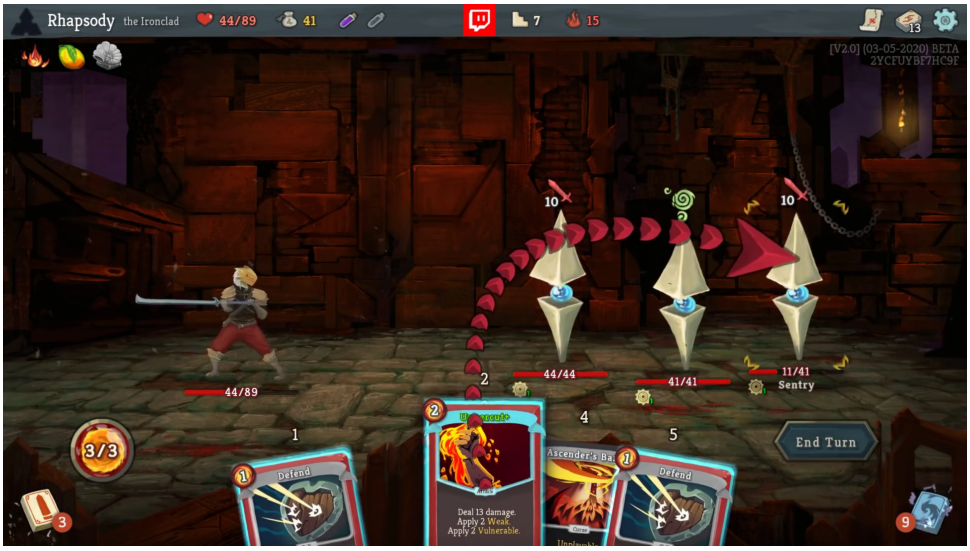


Figure 2.2: A battle screen in Slay the Spire. The player character is on the left, while the enemies are on the right. The player's energy can be seen as being "3/3" in the bottom left corner. The second number is how much energy the player starts with each turn. The card costs can be seen in the top left corner of each card.

Throughout the game you will be able to modify the deck of cards that you use in combat: You can optionally add a card after an enemy encounter, you can buy or remove them at merchants, you can sometimes get them from events, and you can upgrade cards at resting sites. Each card only has one upgraded form. You will also get relics from various sources. Relics provide many different passive effects. If you die, the run ends, and you have to start over. There is a lot of randomness in a run of the game, which is decided by a random seed. When starting a run, you can optionally provide the seed yourself. If you win the game by defeating the final boss, you unlock the next level of ascension. Each level of ascension will add a new layer of difficulty, e.g. bosses deal more damage. There are 20 levels of ascension as well as ascension 0, which is what you start with.

2.2 Existing AIs

In this work the term 'AI' will only be used to refer to a program that can play Slay the Spire, regardless of whether it uses any sort of machine learning or not. There are already a few complete Slay the Spire AIs that can play through a run from start to finish. The earliest Slay the Spire AI that I have been able to find is the Spirecomm AI [9]. The Spirecomm mod is made to facilitate communication between Slay the Spire and an external process. This means that, for example, a Python script can get information about the game state and perform actions directly through the code instead of using the mouse and keyboard. The developer of the mod also included a rudimentary AI made to be pretty much as simple as possible. Most of its decisions are just based on a priority list that tries to aim for roughly the same deck. When in a battle, it will try to block most of the incoming damage and then just play cards according to another priority list. The AI works for all the characters, but we are only interested in how well it does with the Ironclad, where it wins 7-8% of runs on ascension 0 [2].

The next AI is a modification of Spirecomm AI that improves on how the AI handles battles. It does this using sts_lightspeed [10], which is a codebase that can simulate Slay the Spire battles and do a tree search to find the best sequence of moves. The creator of the AI compared it against the standard Spirecomm AI on the same 52 seeds, where the original Spirecomm AI got 5 wins, and the tree search variant got 11 wins [12].

The final AI is named Bottled AI. It currently only works for the Ironclad. This AI also uses tree searching for making moves in battles. Other than that it also has more attention put into various other aspects of the game: path selection accounts for how much health it expects to lose/gain, events have individual logic for which option to choose, and the AI has some extra logic for specific tricky enemies. On a test with 30 seeds it got 5 wins [1].

2.3 Machine Learning Methods

2.3.1 Imitation Learning

There are different variants of imitation learning, but what they have in common is that they require a person who is good at the task we are trying to learn. This person is referred to as the expert. The expert does not have to be directly involved in the project because some methods of imitation learning just require previously recorded demonstrations. One such method is called behavioural cloning. In BC we do supervised learning with a set of expert demonstrations to make a policy that can take an arbitrary state and output an action that estimates what the expert would have done [15]. Sometimes though, the expert's demonstrations may not be in the form of easily usable data. It could be in the form of a video, where we have to somehow derive information about the states and actions ourselves. Behavioural cloning from observation (BCO) was made for this purpose. BCO tries to make an inverse dynamics model, which can take two states and find out the action necessary to get from the one state to the other. It does this by initially spending time interacting with the environment that the task is set in. During this time, it will explore the possible actions and their corresponding effects so it can learn its inverse dynamics model. Using this model, it can then figure out the actions from the video and do BC with these state-action pairs [14]. Another type of imitation learning is called inverse reinforcement learning (IRL). In IRL we try to derive a reward function from the expert's demonstrations [15]. This reward function can then be used for regular RL, or if you have a dynamics model, the reward function can be a part of the policy.

2.3.2 Reinforcement Learning

With reinforcement learning we train an agent by letting it interact with the environment and learn from its experiences. When the agent performs a sequence of actions in the environment from start to finish, it is called a trajectory. Each action in a trajectory is often stored in a replay buffer, which can then be sampled to get training data for the agent afterwards. We need to also give the agent a reward function, which tells it how good a state or action is. The agent will then try to learn which moves give the highest reward. To avoid the agent becoming entirely greedy, we can use a discounted cumulative reward. We get the cumulative reward by summing the reward of an action together with the rewards of all the following actions in the trajectory. The *discounted* cumulative reward is when we multiply each subsequent reward with a constant factor $\gamma < 1$ when calculating the cumulative reward. This way a reward is worth less, the further it is into the future. The intuition behind this is that the future usually holds some degree of uncertainty, so we should prioritise more immediate rewards. It can also incentivise the agent to reach a reward faster. When the agent has found a good policy, it is possible that an even better policy exists. If the agent only acted according to its seemingly best policy, which is called exploitation, it would never find any better policy. Therefore, we want the agent to sometimes make moves it otherwise would not have made, which is called exploration. [4]

Arguably the simplest form of RL is Q-Learning, where the agent has a table of all possible state-action pairs and their corresponding Q-values, which are the highest possible cumulative rewards for the resulting states. Initially all the entries in the table are 0, but as the agent explores the environment, it updates the values in the table. This is not viable for problems with big state spaces, where it would be better to use a Deep Q-Network. A DQN is a neural network that takes a state and outputs estimated Q-values for each possible action. As the agent then interacts with the environment, we would just train the DQN like any other neural network. These methods are value-based because they try to estimate the value of a state or action and act based on that value, but there are also policy-based methods, which output which actions to take without estimating the values of the actions. An example of a policy-based method is the Reinforce algorithm, which gets a trajectory, calculates the actions' cumulative rewards, and does gradient ascent to make the network more or less likely to perform the trajectory's actions based on the reward. There are also actor-critic methods, which combine value-based and policy-based methods. An example of this is the Advantage Actor Critic method. A2C uses two networks: a critic and an actor. The critic takes a state-action pair and estimates its value. It is trained by giving it state-action pairs and their observed values. The actor outputs a probability distribution for which action should be taken. It is trained by seeing whether the observed reward of an action is better or worse than the average value of that state. The average value is calculated by multiplying the actor's probability distribution with the critic's estimates for each action. Proximal Policy Optimisation builds upon A2C by checking if a change in the actor has a too drastic effect on its output and limiting the change if it does. [4]

Video Analysis

3

3.1 Data Retrieval

As of writing this, Slay the Spire is at version 2.3.4. The game has had many updates since its initial full release in the beginning of 2019, and it would be tedious to account for all of them. Fortunately, the only update after V2.0 that has had any effect on gameplay was V2.2 (the values of a few cards were slightly adjusted) [7]. Since we are only interested in the differences in gameplay, V2.0 will also refer to V2.1 in this work, and V2.2 will also refer to V2.3. V2.0 was released in the beginning of 2020, so by just considering V2.0 and V2.2, we will have access to over three years of Slay the Spire videos.

The videos used for this project have been taken from the channel of a YouTuber called Jorbs. These videos seemed ideal because he is very good at Slay the Spire, he has made many videos of the game, those videos are almost entirely from after V2.0, and he reliably labels his videos in the title, so you can automatically find all the videos with a specific character. To automate the process of downloading the videos, I used a command line tool named `youtube-dl`. The videos were retrieved with the following command:

```
youtube-dl --playlist-start 200 --match-title "Ironclad"
--get-id "https://www.youtube.com/playlist?list=PLesIE_v8rF22GHn_EnTJBJbRvDihqje0W"
| xargs -I '{}' -P 50 youtube-dl 'https://youtube.com/watch?v={}'
```

It starts off by using `youtube-dl` to go through the videos in a playlist with all of Jorbs' Slay the Spire runs. `--playlist-start 200` specifies that we do not want any videos before video number 200, which is the first video using V2.0. `--match-title "Ironclad"` specifies that we only want videos with the word "Ironclad" in the title. The Ironclad is one of the four playable characters, and to limit the scope of the project, we will only be focusing on this character. `--get-id` instructs `youtube-dl` to get the YouTube video IDs of all the videos that match the query. These IDs are then piped to `xargs`, which starts up 50 instances of `youtube-dl`. Each instance will handle downloading one of the videos, and `xargs` will start up another instance with a new video once a video has completed downloading. `youtube-dl` is very slow at downloading an individual video, so it is very helpful to be able to parallelise the downloads with `xargs`. Using this method I got 328 videos that totalled 436 hours and 575 GiB.

3.2 Finding Actions

The first step in analysing a given video of Slay the Spire gameplay is to find all the frames where a card is played. We need to process hundres of hours of video, so we need to prioritise speed to some extent. When a card is played, it will move into the middle of the screen and disappear. All the playable cards have a light blue border around them that persists while the card is being played.

Figure 3.1: An image sequence of what happens to a card when it is played.

The way the program detects whether a card has been played or not is by looking at a vertical line segment that the cards will intersect after being played. In that segment the program will look for a colour that resembles that of the blue border around the cards. If it finds a pixel with that colour, it will assume that a card has just been played. After an action has been found, it will skip forward a fixed number of frames to avoid counting the same action twice.

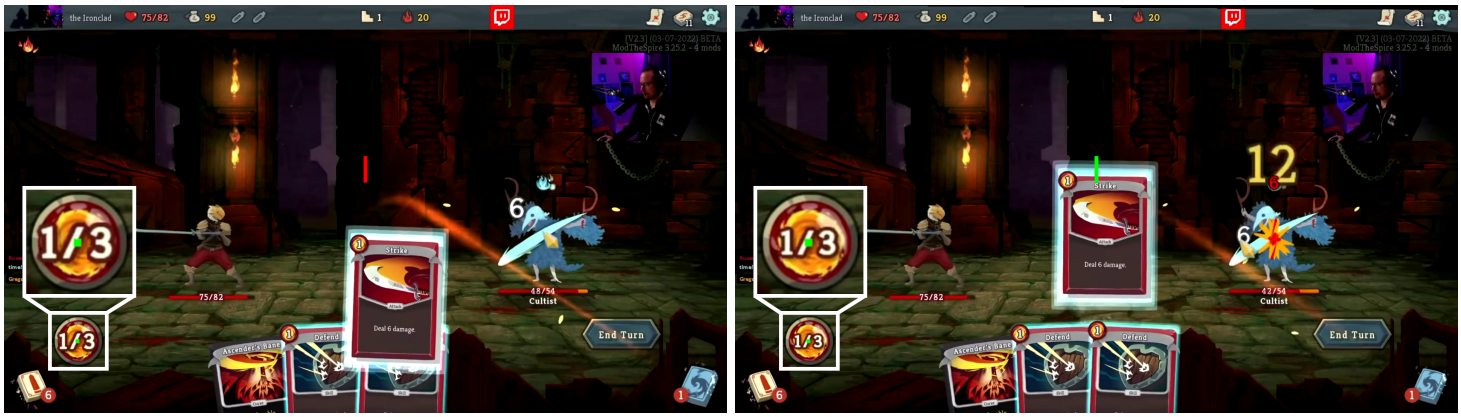


Figure 3.2: The program detecting that a card has been played. The vertical line segment in the middle of the screen is where it searches for the light blue colour. The segment switching to green indicates that it found the blue colour within the line. The energy display has been magnified for easier viewing. The green square shows the horizontal line segment used to check for the presence of the energy display.

False positives are not a problem in regards to the final output since it is very unlikely that the program will be able to find a card and read its name in the frame of a false positive. It would though make the program run slower since it would have to do additional processing on

the false positives before giving up on them, so we would still like to reduce the number of false positives. To do this, I had the program look at where the energy display is supposed to be. If it cannot find white where the slash is supposed to be, it is most likely not in a battle. It looks for white in a horizontal line segment instead of a single point because the exact position of the slash will vary depending on the amount of energy, i.e. a 1 is more narrow than a 3, so the center of the text will be to the right of the slash. When the last enemy is killed, the energy display will immediately move off screen, so the filter will accidentally label the last action a false positive. I had the program look at the energy display a few frames back to avoid this problem. The program maintains a queue of the most recent frames to avoid random access of the video, which can be quite time-costly.

To get an idea of how good this method is at detecting actions, a video of just below an hour was manually labelled. The timestamps found by the program were then compared with the timestamps from the manual labelling. Since the labels were just written as minutes and seconds instead of specific frames, the results likely have a few inaccuracies but should still provide a good approximation. Four experiments were run: one that looked at every frame in the video, one that looked at every other frame, one that also checked if the energy display was present, and one that looked for the energy display a few frames back instead. The results of the experiments can be seen in figure 3.3.

description	true positives	false positives	false negatives	runtime (s)
all frames	404	181	13	385
half frames	396	169	21	268
half frames + filter	390	45	26	263
half frames + filter + past	396	50	21	257

Figure 3.3: Evaluation of how well the program finds actions in a manually labelled video using different methods. The video had 417 actions in total. An action is considered a true positive if a unique label is less than 90 frames (1.5 seconds) away from the timestamp where the program found the action. The runtimes are only based on the program finding the actions without trying to identify them.

As we can see, looking through only every other frame instead of every frame saves around 30% of the time, and it only costs us very few extra actions. Though, this method still has a lot of false positives. Checking for the presence of the energy display greatly reduces the number of false positives at the cost of very few true positives. Checking instead a few frames back removes roughly as many false positives without losing any true positives. This last method, which was able to find 95% of the actions, was chosen for the program.

3.3 Identifying Actions

Now that we have a frame where a card is being played, we need to figure out which card it is. Since the OCR tool Python-tesseract [5] works pretty well when given a binary image, I decided to take the approach of trying to crop the name of the card, process it, and give the resulting image to Tesseract. Since we know that the top of the card is passing through our vertical line segment from the previous section, we can crop the frame to the general area of where the card

has to be. We then go through each pixel in this crop, set it to white if it is close enough to the colour of the blue border, and set it to black otherwise. Calculating whether or not a pixel is close enough to the colour of the blue border is done by calculating the squared Euclidean distance in the RGB colourspace between the colour of the pixel and a blue colour picked ahead of time and seeing if it falls within a specified tolerance. By calculating the squared Euclidean distance, we avoid a time-costly square root operation, which will be irrelevant if we just square the tolerance as well.

(a) (b)

Figure 3.4: a) the crop of the general location of the card being played. b) the same crop after each pixel has been set to black or white depending on whether or not they are close enough to the colour of the blue border.

After we have gotten a binary image with the rough shape of the blue border around the card, we can use a Hough line transformation to find the horizontal line and the two vertical lines that make up the edges of the card.

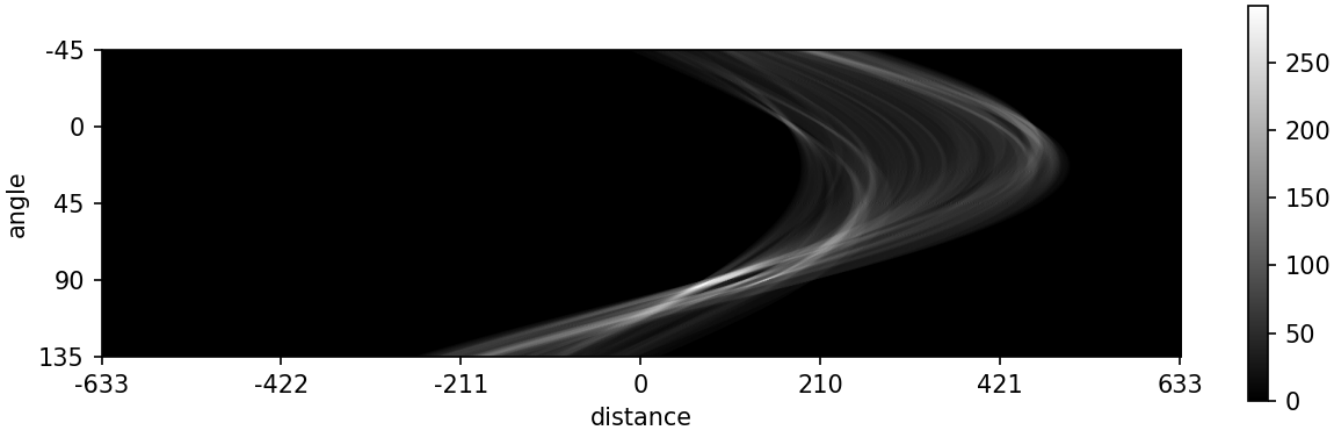
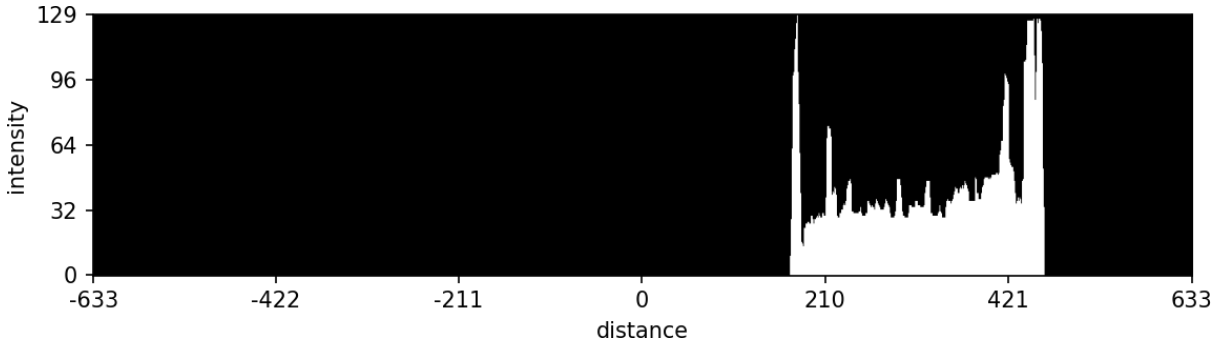
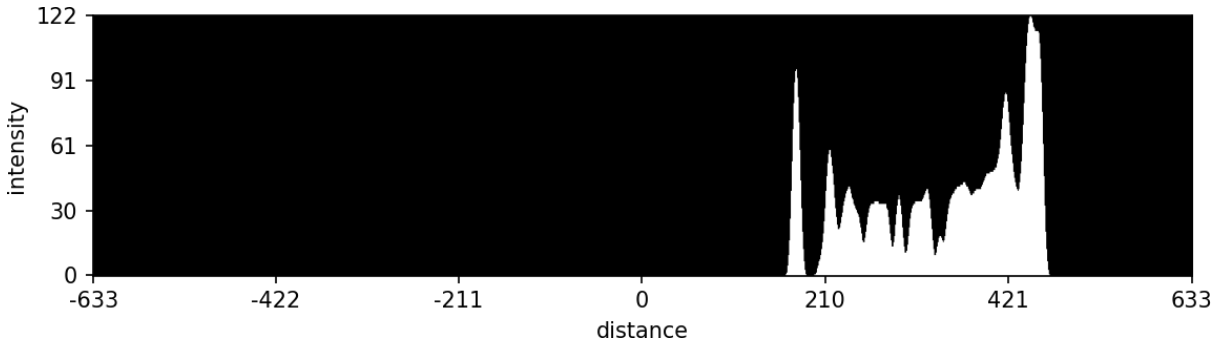


Figure 3.5: The Hough line transformation of figure 3.4b. Each pixel represents a line passing through the original image. Each line is specified with the perpendicular line segment that goes from the line to the origin. Because of the inverted Y-axis of image coordinates, our origin is effectively the top left corner, and our angles move clockwise. The Y-axis specifies the angle of the segment with the X-axis of the original image, and the X-axis specifies its length. The brighter the pixel, the more white pixels the line intersected.

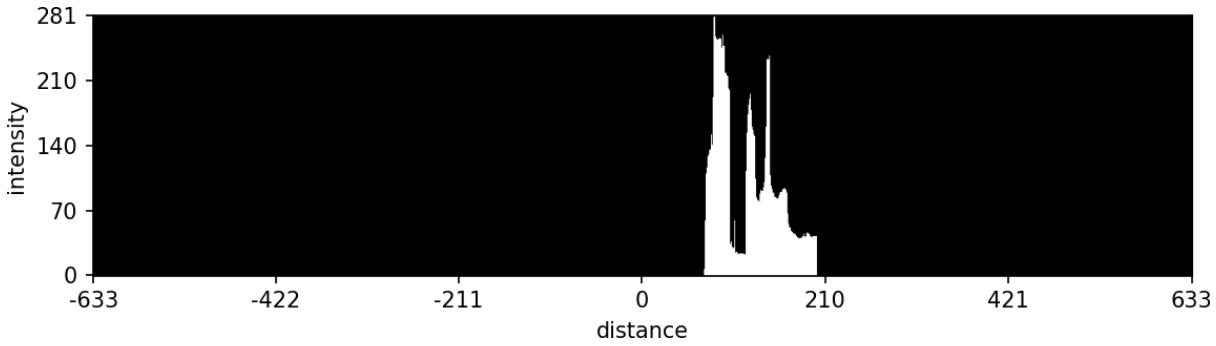
The card will never be rotated at this point, so we are only interested in perfectly horizontal and vertical lines. We therefore extract only the rows in the Hough transformation pertaining to those two angles. There might be pixels outside the card which happen to be the same colour as the blue border, so to ensure that these erroneous responses are not interpreted as edges of the cards, a highpass filter is applied to the Hough transformation: Values below a certain threshold will be set to 0 while values above the threshold will remain unchanged. A Gaussian blur is applied afterwards. This ensures that the maximum will be somewhat in the middle of the line.



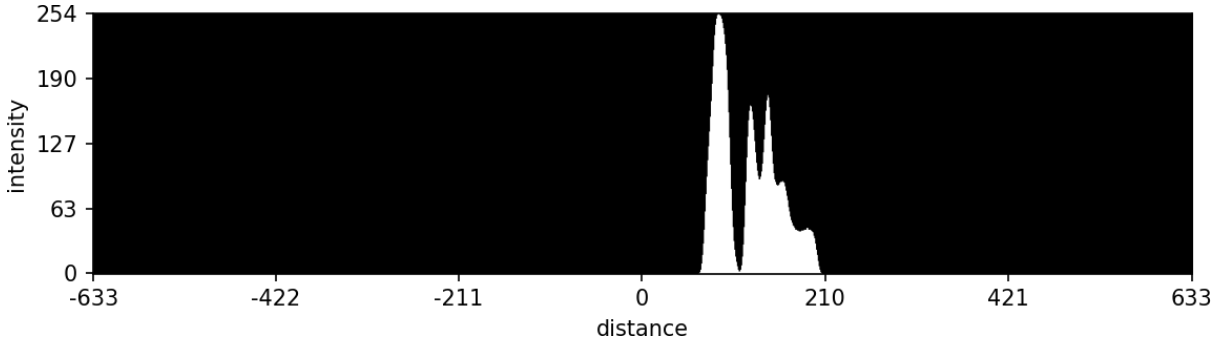
(a)



(b)



(c)



(d)

Figure 3.6: Horizontal slices of figure 3.5 at angles 0 (a and b) and 90 (c and d). (a) and (c) are the raw slices. (b) and (d) are the slices after a highpass filter and a Gaussian blur.

`skimage`'s `peak_local_max` [6] is used to find the local maxima, and they are sorted according to their position along their axes. The topmost maximum of the horizontal lines is picked as the top of the card, and the leftmost and rightmost maxima of the vertical lines are picked as the left and right side of the card. We now have a decent segmentation of the card, so we can make a pretty accurate crop of the card header.

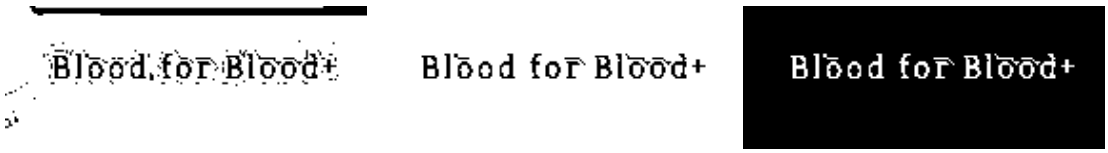
(a) (b)

Figure 3.7: a) the lines found from the Hough transformation overlayed on the original crop from figure 3.4a. b) a tighter crop of the card header based on fixed distances from the lines in (a).

3.3.1 Reading Card Headers

The text in the header will be bright green if the card is upgraded and white otherwise. We threshold each colour channel at a pretty high value and use those thresholded images to generate a binary image where a pixel is black if all three channels are above the threshold (white) or if only the green channel is above the threshold. The resulting image can be seen in figure 3.8a. The green channel would be above the threshold in both cases, but we cannot ignore the other channels because some cards will have a yellow banner behind the text, which would also be above the threshold on the green channel. The binary image is quite messy, and the blue border turns white when it gets closer to the card, so it will often also get included. To clean up the image we look at the connected components and filter away the ones that are too small or too large to be letters. We now have a much cleaner binary image (figure 3.8b) which is ready to be sent to Tesseract. The string Tesseract returns will be filtered for characters that do not appear in any card names and be compared with a list of all the names of the playable cards using the approximate string comparison library `fuzzywuzzy` [3]. The card name that best matches the string from Tesseract will be assumed to be the card that was played unless none of the card names are a decent match.

If the card was upgraded and a '+' was read, the '+' would be removed before the string comparisons and added back afterwards. The '+' is often overlooked by Tesseract or lost somewhere else in the pipeline, so a second method was included to make the program more reliable at seeing when a card is upgraded. A binary image is generated where a pixel is white if it meets two conditions: Only the green channel was above the threshold in the original image (figure 3.7b) and it was black in the image sent to Tesseract (figure 3.8b). The pixels in the resulting image (figure 3.8c) can then be summed, and if the sum is above a certain threshold, we can assume that the text is green and the card is therefore upgraded.



(a) (b) (c)

Figure 3.8: a) a binary image where a pixel is black if the corresponding pixel in 3.7b is either white or green. b) (a) where connected components that are too small or too large to be letters have been removed. c) a binary image showing where there is green text in 3.7b.

The different levels of text processing was tested with the manually labelled video. The results can be seen in figure 3.9. With all the processing the program was able to correctly identify 352 of the 396 cards (89%) found in section 3.2. If we consider that the video had 417 labels, that means that the program was able to correctly find and identify 84.4% of the actions.

description	true positives	correct card
no processing	281	325
connected components filter	308	357
additional upgrade check	352	357

Figure 3.9: Evaluation of how well the program identifies the actions found in section 3.2. The 'correct card' column shows how many cards it identified correctly if we ignore whether the card was upgraded or not. The comparisons are only made for the cases where an action was found at roughly the same time as a label (figure 3.3).

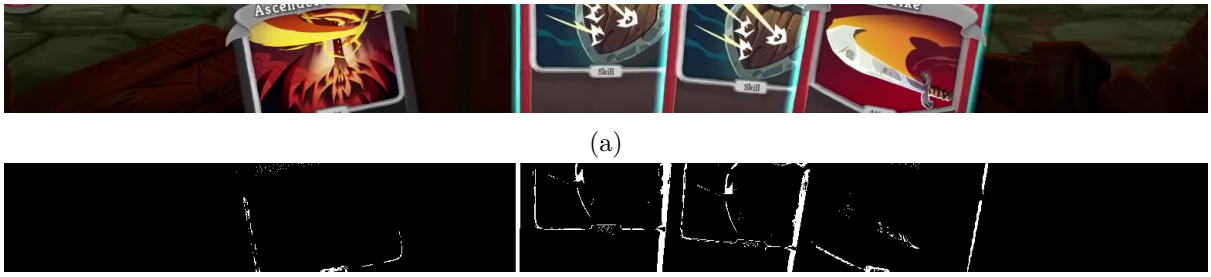
3.4 Reading the Hand

Figure 3.10: A screenshot of a hand right after a card has been played.

After we have identified which card has been played, we need to figure out the hand from which it was played. We assume that the card's position in the hand does not matter, so if we find the hand that remains after the card in question has been played, we can just shuffle the card back into the hand. We can use some of the same ideas we used for identifying which card was played: All the playable cards have blue borders around them, which can be used to get pretty good segmentations of the cards in a hand. I will refer to this as the blue method. Unfortunately, this would completely ignore unplayable cards and cards that cost more energy than available. At first one might think that that is fine since those cards are not options anyways, but they can still be relevant for decision-making, i.e. the unplayable card 'Burn' deals 2 damage to the player at the end of the turn, so the player should not just let themselves die if they could block the damage. Another example is if the player has cards that cost too much energy, but they can play 'Fiend Fire', which deals 7 damage for each card in their hand. I figured out a method for reading hands that, instead of using the blue border, relies on comparing the screenshot to an image of the background and using the colour difference to see what is foreground and what is background. I will refer to this as the general method. This method can read both playable and unplayable cards, but it is not as reliable as the blue method, so the final method used in the program is a hybrid of both the blue method and the general method.

3.4.1 The Blue Method

The first step of the blue method is to identify all the vertical blue lines in the screenshot of the hand, so we start off by turning the screenshot into a binary image where a pixel is white if it is close to the colour of the blue borders.

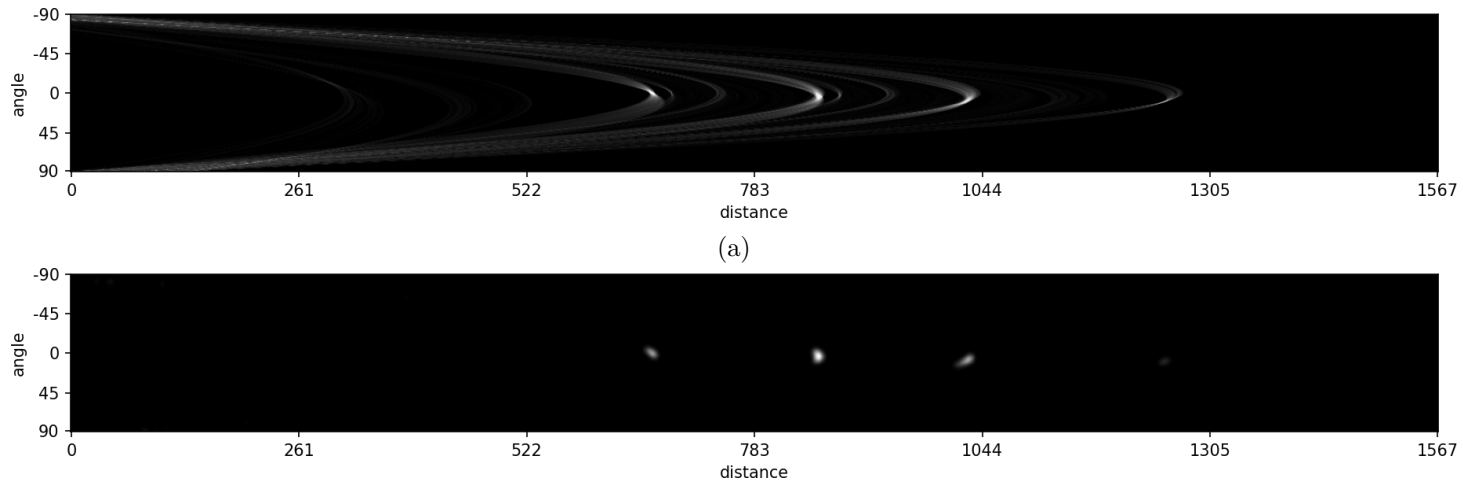


(a)

(b)

Figure 3.11: a) the crop used for identifying the hand using the blue method. b) a binary image showing the light blue parts of (a).

We then want to use a Hough transform to identify the vertical lines. The Hough transform for figure 3.11b can be seen at figure 3.12a. We can see three pretty clear responses and one less clear response to the right of them. The clear responses stem from the left sides of the three playable cards, and the less clear response stems from the right side of the card furthest to the right. Note that since the card furthest to the left is unplayable, it does not have a blue border and therefore does not show up in the Hough transform. We remove pixels that are too dark, so only the strong responses from the blue borders remain, and we blur the image to help ensure that the maxima are in the centres of their corresponding lines. The result can be seen in figure 3.12b.



(a)

(b)

Figure 3.12: a) the Hough transform of figure 3.11b. b) (a) after highpass filtering and Gaussian blurring.

Each of these points indicate a straight line that goes along a side of a playable card. Not all the cards will have their right sides visible, so we will treat each line as the left side of the card. This does mean that the last line that actually belongs to a right side will be a false positive, but as we will see that does not become a problem. For each line we know both the position and angle, so we can rotate the image so the card text is horizontal and make a crop that corresponds to roughly the size of a card. The card crops that come out of the lines found in figure 3.12 can be seen in the first row of figure 3.13. We want to get the text from each card,

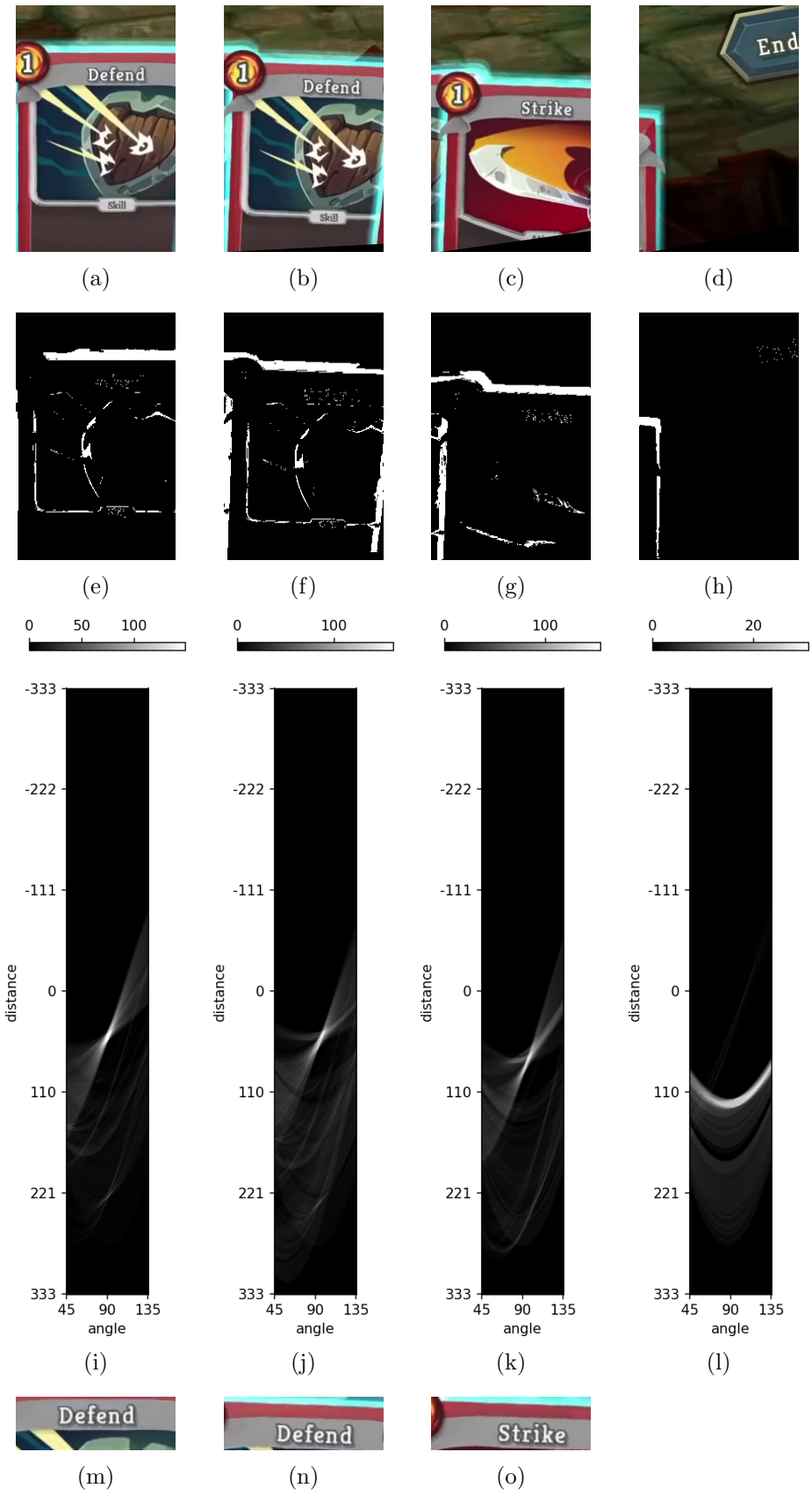
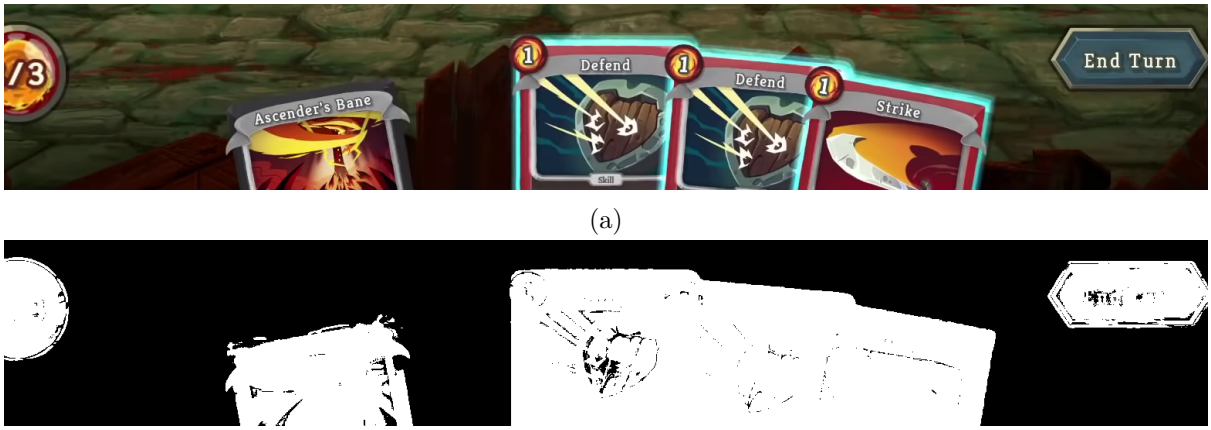


Figure 3.13: The process of cropping the card headers with the blue method. a-d) the rough card segmentations found from figure 3.12. e-h) binary images showing the light blue parts of (a-d). i-l) Hough transforms for mostly horizontal lines of (e-h). The images are normalised, so (l) appears much brighter than it actually is. m-o) header crops from using (i-l) to find the tops of the cards in (a-d).

but it is placed at different heights, so we want to find the top of the card as a point of reference to get a good crop of the text. To do this, we find the top of the blue border with the same method we used to find the left and right sides: We make a binary image showing where the blue colour is and find the maximum in the Hough transform. These steps can be seen in the second and third row of figure 3.13. Finally, in the last row of figure 3.13 we can see the text crops that we got from the process. It does not find any horizontal blue line in the last column because the response from the Hough transform is too weak, so it correctly identifies it as a false positive. The text crops are then processed according to section 3.3.1.

3.4.2 The General Method

The general method tries to get crops of the card headers by identifying which parts of a screenshot are background and using the shape of the hand's outline to segment the cards. Fortunately, the backgrounds at the bottom of the screen are mostly the same between battles, so we can take screenshots of the backgrounds from one of the videos to get images that the program can use for reference. Each act has a different general background, but we can figure out which one to use by looking at which floor we are at. There are also a couple of exceptions to account for: In act 1 there is an event that can start a battle where the background has a bunch of mushrooms, and in act 3 there are two puddles that the background will only sometimes have. To distinguish these cases, the program will look at specified small areas of the screenshot and see which one of the possible backgrounds is the most similar. Now that we have an image of the background, we can get a good segmentation of the foreground by simply checking if the colour difference between the screenshot and the background image is outside a specified threshold. This segmenting can be seen in figure 3.14.

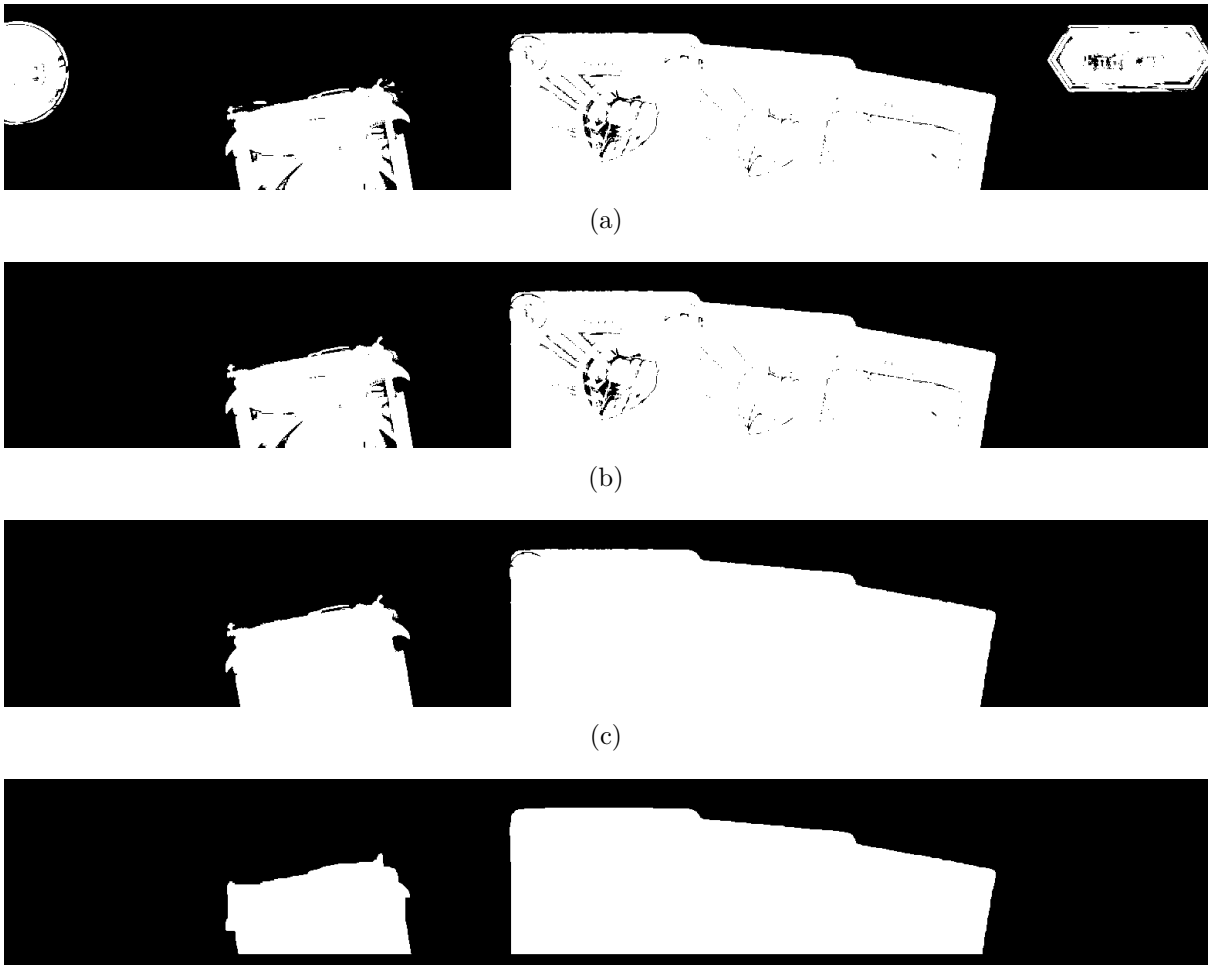


(a)

(b)

Figure 3.14: Segmenting the foreground by comparing the screenshot in (a) to an image of the background to produce (b).

We can see that the foreground also ends up including the energy as well as the 'End Turn' button. We also see that some parts of the cards end up being labelled as background because their colours are coincidentally similar to the background. Since the cards are significantly larger than the two other foreground elements, we can get rid of those foreground elements by removing connected components smaller than a specified area. The result of this can be seen in figure 3.15b. We can get rid of all the holes in the cards by temporarily inverting the image, so the background will be one massive connected component. We then filter away any smaller connected components before inverting the image again. The result of this can be seen in figure 3.15c. The silhouette of the leftmost card is still very uneven, so we try to smoothen this out by using binary closing. The result of this can be seen in figure 3.15d.



(a)

(b)

(c)

(d)

Figure 3.15: Cleaning up the segmentation from figure 3.14 by removing small connected components, removing small black connected components, and using binary closing.

We crop the image to get rid of the gap that the binary closing caused, and we use Canny edge detection to get the outline of the hand.

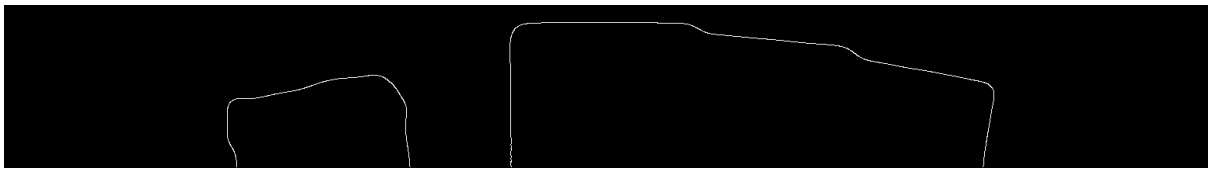
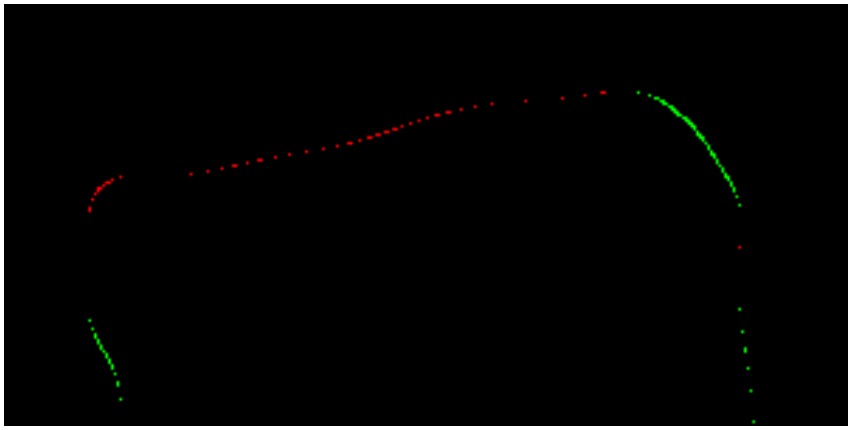


Figure 3.16: The outline of the hand, gotten by using Canny edge detection on figure 3.15d.

We now want to find all the corners of the outline. To do this we use the hit-or-miss transform. To find left corners we set the hit structuring element to $\begin{array}{|c|c|}\hline 0 & 1 \\ \hline 1 & 0 \\ \hline \end{array}$ and leave the miss structuring element empty. For the right corners we change the hit structuring element to $\begin{array}{|c|c|}\hline 1 & 0 \\ \hline 0 & 1 \\ \hline \end{array}$ instead. We can see what this does to the leftmost card in figure 3.17b. This does though end up finding a lot of "corners" on the lines that are mostly horizontal or vertical.

Thankfully, our hit-or-miss transform has a lot of responses at the actual corners, so if we filter away small connected components, we end up with only actual corners, which can be seen in figure 3.17c. Unfortunately, the uneven shape of the outline ended up having a diagonal line in its left side. We can get rid of these false positives by iterating through all the corners and removing those that have a corner above them at roughly the same x-coordinate. This also avoids the problem of the same corner being counted twice because it ended up with two large connected components instead of one.

(a)



(b)

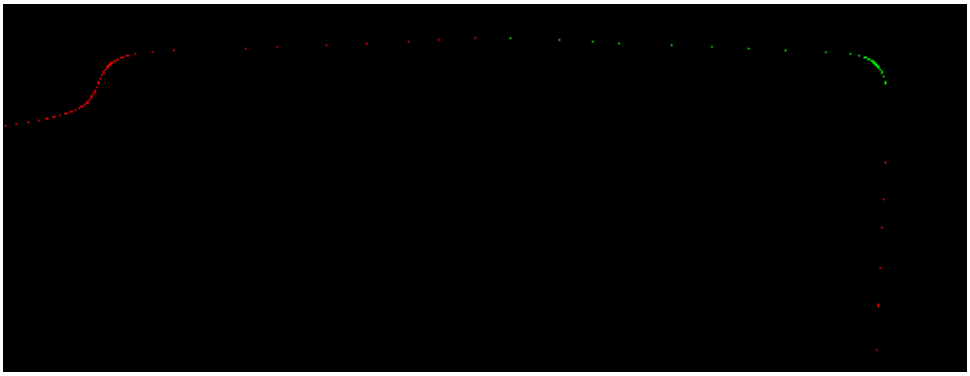
(c)

Figure 3.17: Finding corners of (a) by using two hit-or-miss transforms to get (b). Red is left corners, and green is right corners. Removing small connected components afterwards result in (c).

Figure 3.18: Figure 3.14a with the corners found by the program drawn unto it. A red dot means that it is a left corner, while a green dot means that it is a right corner.

After we have filtered away the false positives and the duplicate corners, we want to figure out the general placements of each card. We sort the corners according to their x-coordinate and go through each neighbouring pair of corners. Each pair will have four possible cases: If we have two left corners, the right one belongs to a different card, so we should use the left one as our point of reference; if we have a left and a right corner, we are looking at a single card, and both corners can be used as our point of reference, so we will arbitrarily choose the left one; if we have a right corner and a left corner, we are looking at a gap between the cards, so we should just move on to the next pair; if we have two right corners, the left one belongs to a different card, so we should use the right one as our point of reference. There is also the special case where we have a left and a right corner, but they are too far away from each other to be from the same card. An example of this can be seen in figure 3.19. In this case we will use both corners as points of reference, and they will be counted as separate cards. These rules are summed up in figure 3.20.

(a)



(b)

Figure 3.19: A special case when finding cards according their corners. (a) is the original screenshot. (b) is the result of the hit-or-miss transform on (a). Because the cards are at the same height, the left corner of the right card is not visible with the hit-or-miss transform.

first corner	second corner	distance	point of reference
left	left		first
left	right	$<= 300$	first
		> 300	both
right	left		neither
right	right		second

Figure 3.20: The rules for how a card is counted based on a neighbouring pair of corners. The first and second corner refer to the position of the corners when going from left to right. The point of reference refers to which corner will be used for the remaining steps of the method.

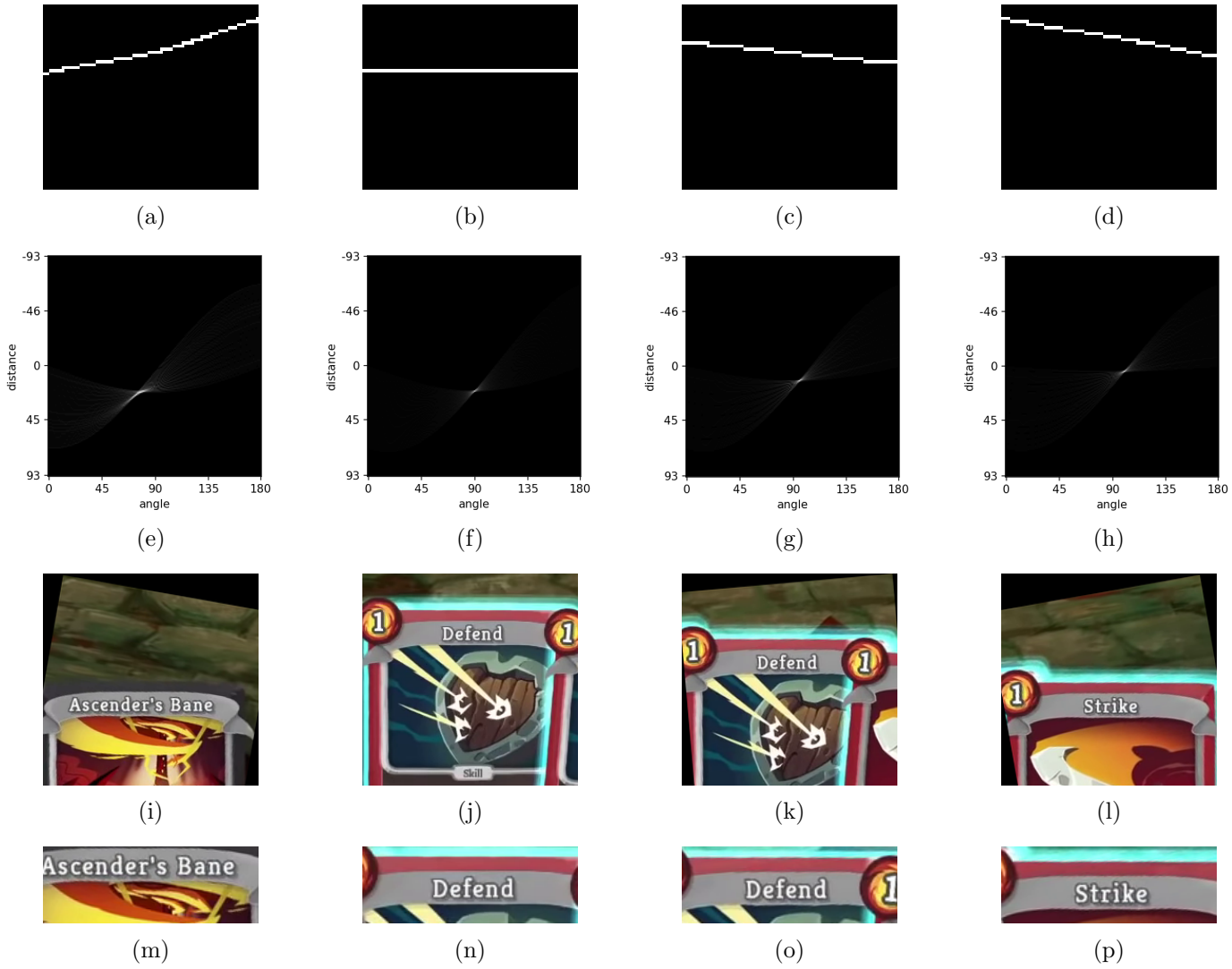


Figure 3.21: The process of cropping the card headers with the general method. a-d) crops from the tops of the cards taken from figure 3.16. e-h) the corresponding Hough transforms. i-l) the cards rotated according to the angles from the maxima of the Hough transforms. m-p) the final crops of the headers.

Now that we have one corner of each card, we want to rotate the cards around those points so the text is horizontal. We can figure out the rotation of each card by looking at the top of the card in the outline of the hand from figure 3.16. We know roughly where each card is, so we can just take a small crop in the general area. These crops can be seen in the first row of figure 3.21. Afterwards, we can use a Hough transform and find the pixel with the highest value to get the angle of the line. The Hough transforms can be seen in the second row of figure 3.21. We now take a crop of the relevant area around the corner and rotate it with the angle we just found. We cropped the image first to avoid computing a bunch of pixels in the rotation that we would not need. The rotated cards can be seen in the third row of figure 3.21. We used the corner as the pivot of our rotation, so its position has not changed. We therefore know where the tops of the cards are and can make rough crops of the text in their headers. These crops can be seen in the fourth row of figure 3.21. The text in these crops is then read according to section 3.3.1.

3.4.3 The Hybrid Method

Since the blue method and the general method have different strengths and weaknesses, I thought it would make sense to combine them. This is done by first using the blue method and keeping track of the positions of the found cards. When we afterwards use the general method and have found the general positions of those cards, we can skip any card whose position is close to one of the cards found by the blue method. This ensures that we do not count the same card twice. As we can see in figure 3.22, the blue method successfully read 58% of the cards, the general method read 71% of the cards, and the hybrid method read 81% of the cards. The hybrid method correctly reads significantly more cards than either of the other methods on their own, so it is worth the extra runtime.

description	true positives	false positives	false negatives	runtime (s)
Blue method	824	13	600	849
General method	1015	36	409	905
Hybrid method	1153	24	271	1429

Figure 3.22: Evaluation of how well the program finds the cards in the hands in a manually labelled video using different methods. The comparisons are only made for the cases where an action was found at roughly the same time as a label (figure 3.3). Summing up all the cards in the hands of the labels for these cases gives 1424 cards.

3.5 Miscellaneous Data

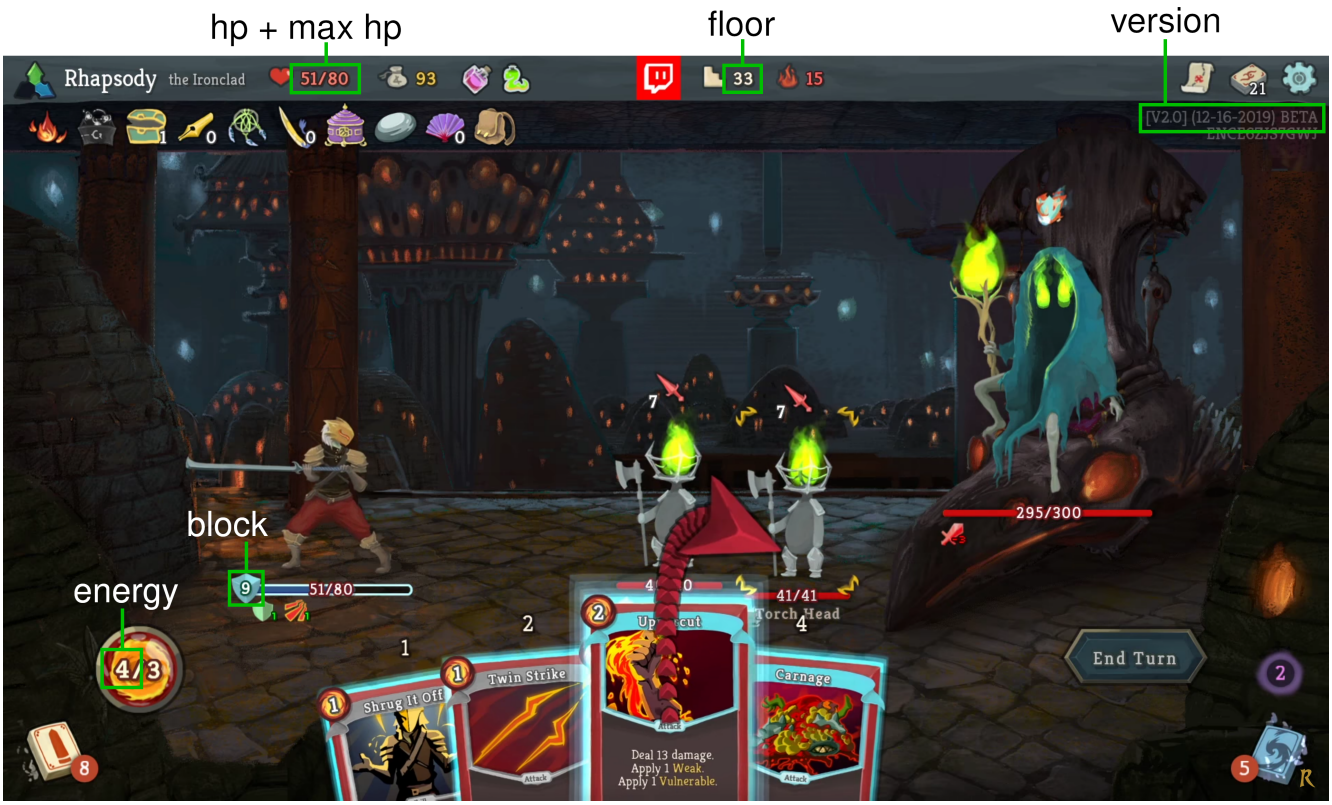


Figure 3.23: Overview of the remaining miscellaneous data that the program reads for each action.

To get the hp and max hp, we take a crop of the text in the bar at the top of the screen. The exact position does though vary based on the length of the username of the player. Hp and max hp are also displayed just below our character, but that text will move around when the character attacks, and it also risks being occluded by visual effects from combat. The red channel of the crop is thresholded, and the resulting binary image is passed to Tesseract.

(a) (b)

Figure 3.24: The screenshot of hp and max hp as well as its thresholded image.

The floor is read the same way as the hp. Thresholding only the red channel still makes sense since it is the colour the background has the least of.

(a) (b)

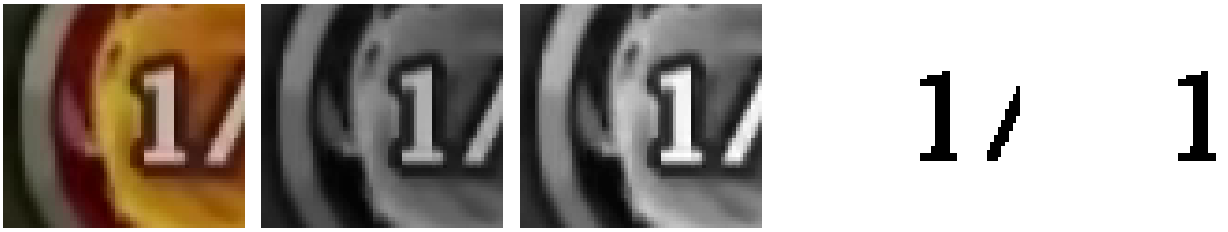
Figure 3.25: The screenshot of the floor as well as its thresholded image.

Block is read mostly the same way except the crop is converted to greyscale before thresholding. We also remove small connected components before OCR.

(a) (b) (c) (d)

Figure 3.26: The pipeline of reading the amount of block.

When reading the energy, we only care about the left number because the right number is not an actual capacity, it is just how much energy you start with each turn. Sometimes the image will be slightly darkened because the player has just looked at the draw pile, which darkens the rest of the screen. The energy crop is therefore normalised before thresholding. This could not be done while reading block because there would not always be text, so normalising the image could just result in irrelevant things becoming bright enough to be read as text. After the crop has been thresholded, we remove what remains of the slash by removing any connected components that touch the right edge of the crop. This is a surprisingly important detail because if we have 4 energy, and the slash is still in the crop, Tesseract ends up often reading the 4 as a 'Y'.



(a) (b) (c) (d) (e)

Figure 3.27: The pipeline of reading the amount of energy.

The game version can be seen in the top right corner. This text also includes the date of the build and whether or not it is a beta build, so the exact placement of the part we are interested in will move slightly, and we have to take a slightly wider crop. The crop is thresholded and passed to Tesseract. If the crop has the text "[V2._]", where _ can be any character, the program will consider the OCR reliable and make a decision on the game version. If the missing character is a 0 or a 1, the program will say that the game is V2.0, otherwise it will say that the game is V2.2. After we have found the game version, we can just use that for the rest of the video since we know that it will not change.

(a) (b)

Figure 3.28: The screenshot of the version as well as its thresholded image.

Behavioural Cloning

4

4.1 Network Architecture

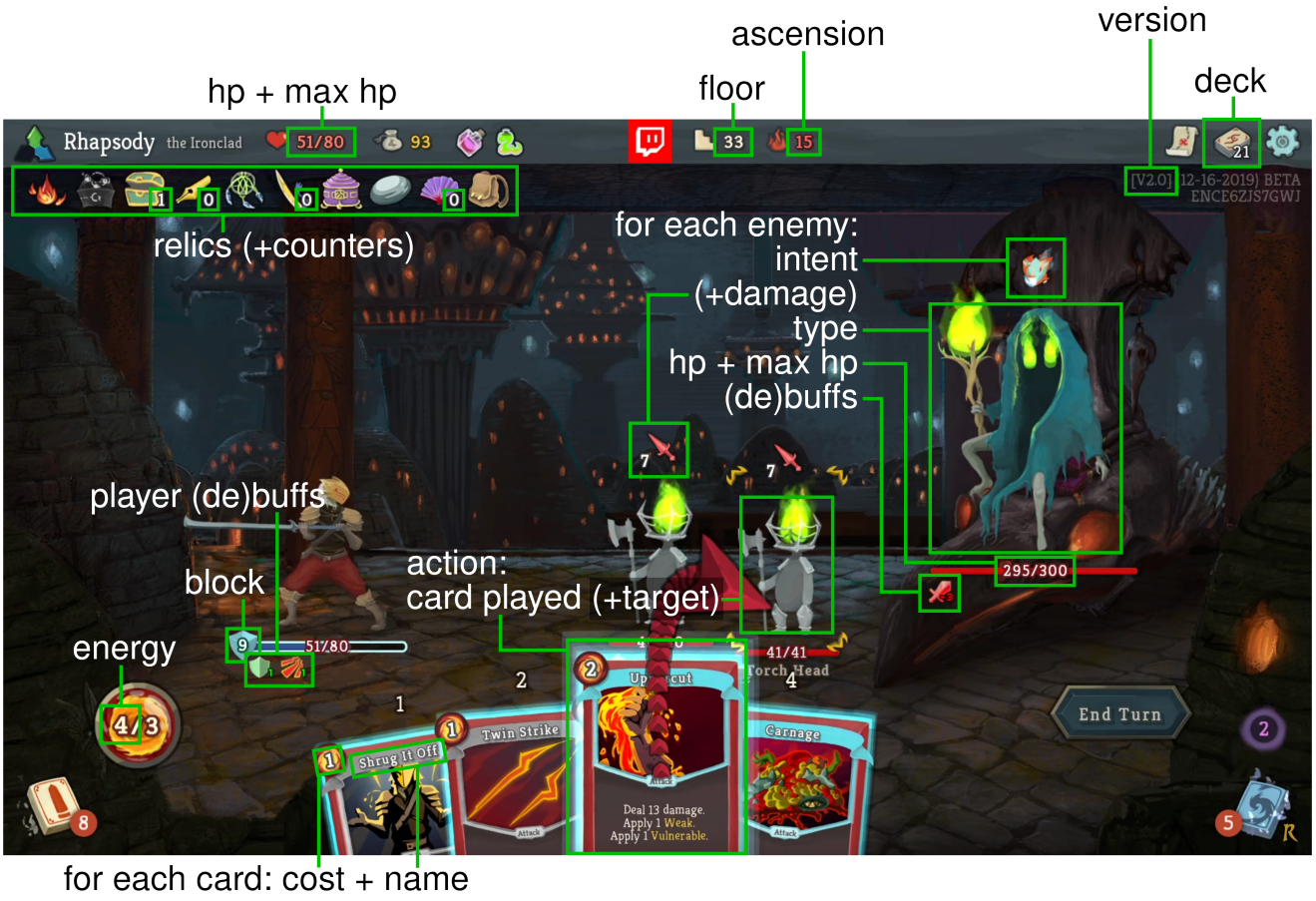


Figure 4.1: Overview of information that would be relevant for a combat AI.

4.1.1 Ideal Network

If the video analysis part was completed to the point that it would read all the information highlighted in figure 4.1, then figure 4.2 is how I would imagine the inputs and outputs of the neural network to be structured.

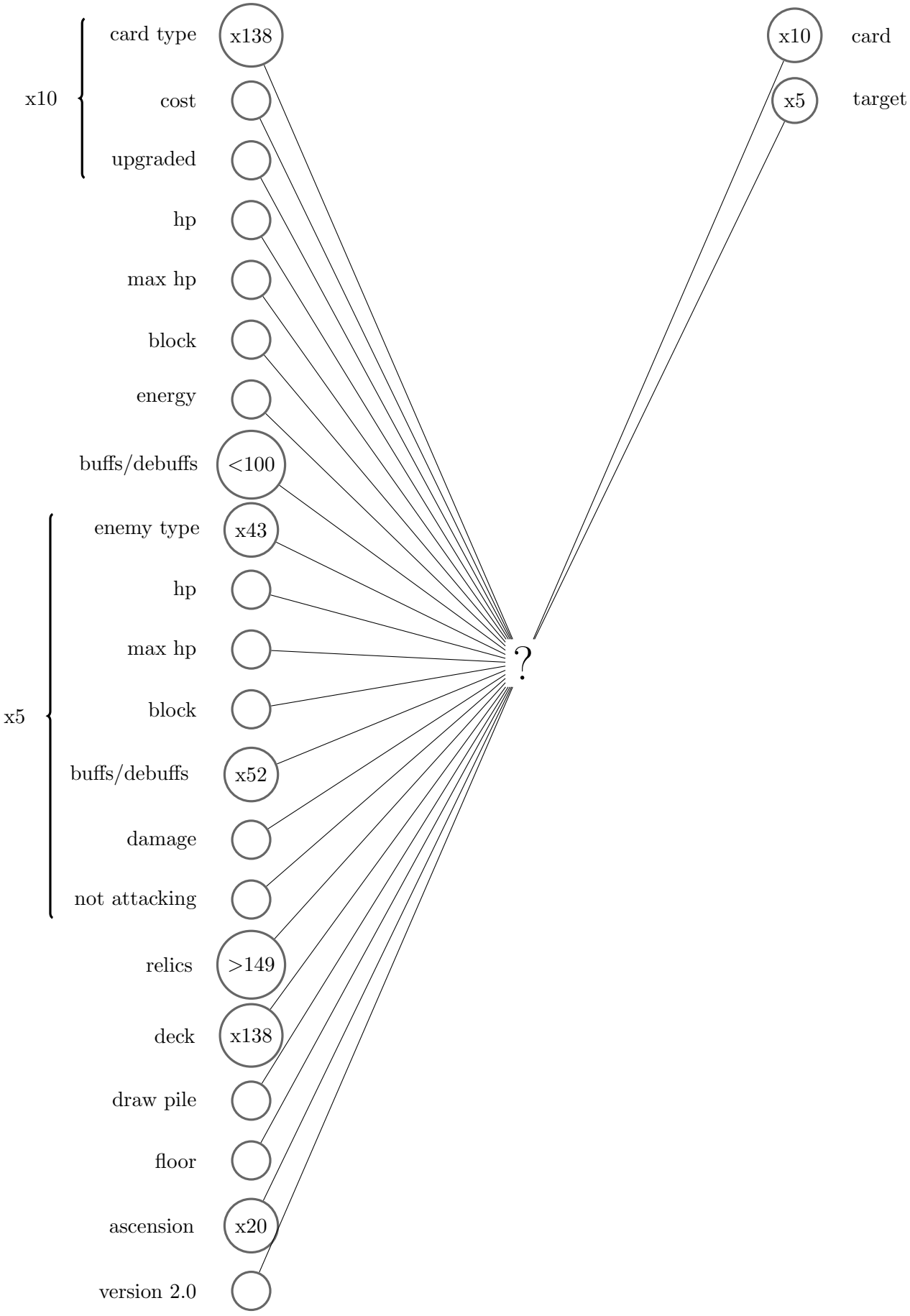


Figure 4.2: Visualisation of the interface for the ideal network architecture.

4.1.1.1 Inputs

Simple integer values like hp, max hp, block, energy, and floor will each have their own input neuron where the value is multiplied by 0.01. The Ironclad has access to 138 different cards, and the player can at most hold 10 cards in their hand, so each of the 10 cards will be a ones hot with 138 inputs. Each card in the hand will also have a cost, which will not necessarily be the same for the same type of card. These costs will be represented by 1 input for each card in the hand, where the costs will be multiplied by 0.1. There are also X-cost cards, which cost your remaining energy, but instead of adding an extra input for this special case, we can just set the cost to the same as the energy. 1 additional binary input will be added for each card to say whether the card is upgraded or not. The player can also have various buffs and debuffs, which are positive and negative passive effects. There are almost 100 that can be applied to the player, but some of them are only possible with the other characters. There will be an input neuron for each of these buffs/debuffs. If the buff/debuff can be applied more than once, the value will be multiplied by 0.01, otherwise it will just be a binary input. [8]

There can be up to five enemies at the same time, so all the inputs necessary for one enemy will have to be multiplied by five. There are 43 different enemies in the first three acts of the game (excluding the variants some enemy types have), so the type of enemy will be represented with a ones hot with 43 inputs. An enemy will have hp, max hp, block, and sometimes buffs/debuffs, which will be represented the same way as with the player. It is though a different set of buffs and debuffs that can be applied to enemies, so it will only require 52 inputs this time. The enemy will also have an intent, which says what the enemy will do on their next turn. It was not possible to get the specific intent from Spirecomm, so the intent will just be represented with two neurons: one where the damage dealt is multiplied by 0.01 and one which will be set to 1 if the enemy is not dealing any damage. [8]

The relics will be represented with a ones hot, but relics with counters will be considered different relics for each state of their counters. There are 149 relics, so this will require at least 149 input neurons. The general deck will be represented with 138 neurons (one for each card) and each value will be the quantity of the given card multiplied by 0.01. Since the card 'Mind Blast' depends on the number in your draw pile, this will also be included as an input where the number is multiplied by 0.01. There are 20 levels of ascension, and the effect of each level will stack on top of the previous ones, so there will be 20 inputs, and each will be set to 1 if the ascension level is greater than or equal to the level it corresponds to. Finally, there will be one input neuron that will be set to 1 if the game is version 2.0. [8]

4.1.1.2 Outputs

The player can have up to 10 cards in their hand at any given time, so we will need 10 outputs for each of the cards that can be played. Furthermore, some cards will require choosing a target from one of the enemies. There will at most be 5 enemies at any given time. Since the targeting is not always necessary, it seems best to have five outputs for targetting and only read their intensities when it is applicable. An alternative would be to have 50 outputs in total, so there would be one output for each card targeting each enemy, but this would make the network more complex and would not make much sense in the cases where the card does not require a target.

4.1.2 Actual Network

In this project the video analysis part only got to the point where it reads a subset of the information highlighted in figure 4.1, so the network used in the later parts of this work uses a different set of inputs and outputs to reflect this.

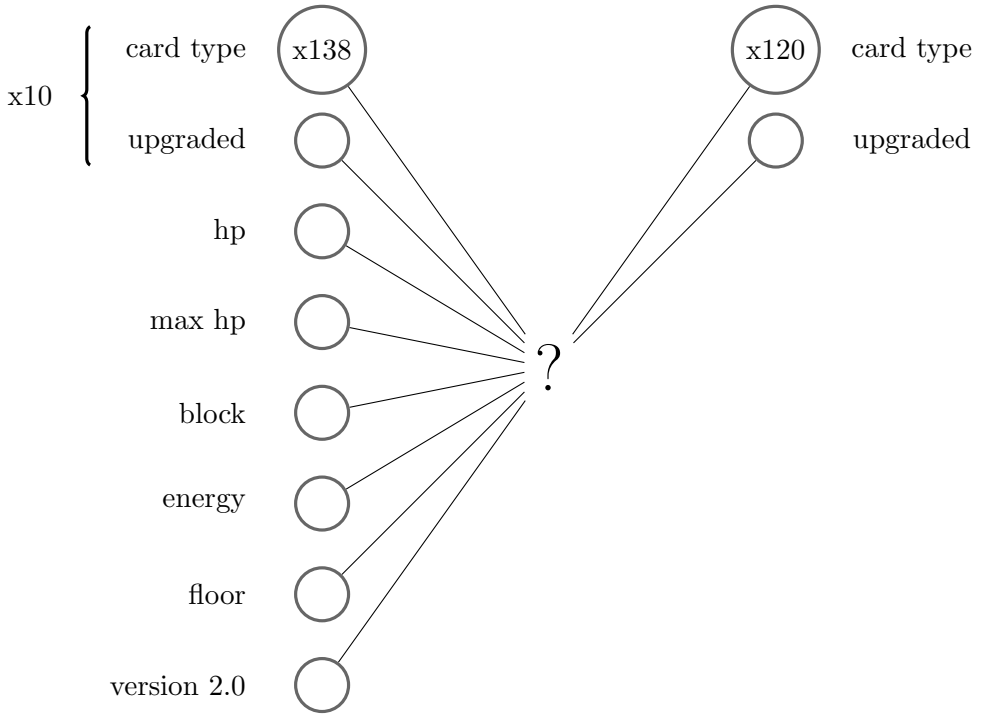


Figure 4.3: Visualisation of the interface for the actual network architecture.

4.1.2.1 Actual Inputs

We still consider all the cards and whether or not they are upgraded, so there will be 10×139 inputs to represent the hand. We also include hp, max hp, block, energy, and floor as described in section 4.1.1.1. Finally, we have the one binary input to say if the game is version 2.0. This will not be relevant if we are playing on the most recent version of the game, but we need it when training the network with BC. It would usually make sense to account for ascension level, but since our dataset consists of data exclusively from ascension 20, our network would not be able to learn anything based on ascension level.

4.1.2.2 Actual Outputs

Since our inputs do not distinguish between the costs of cards, we do not need to worry about the specific position of the card to play, only the type of the card. The Ironclad has access to 120 playable cards, so there will be 121 outputs to also account for whether the card should be upgraded or not. The last output will only be relevant if there is both an upgraded and non-upgraded version of the card in the hand. Since we do not even know which enemies there are, we omit the 5 outputs that were meant for choosing the target.

4.2 Training

After extracting the state-action pairs from all the videos, I got a data set consisting of 165,497 points. When dividing this data into training, evaluation, and test set, it was split according to runs and not just individual data points. The reasoning behind this is that each run will be a significantly different situation because of differences in deck and relics, so being able to predict data points in a run that was also partially in the training set would not be an accurate portrayal of an ability to predict new data. Of the 328 runs, 100 are set aside for the test set, 30 are used for the evaluation set, and the remaining ones are used for the training set. Since the data points only contain the actions and the resulting hands, we will have to recreate the hands from before the actions were performed. This is done by simply inserting the card from the action into a random position in the resulting hand. During evaluation the randomness seed is static to ensure that the evaluation set is the exact same each time, but during training the randomness seed is set to the epoch number so the training set will be slightly different each epoch.

After experimenting with some different setups, I got the best results from using an Adam optimiser and a network with three fully connected linear layers with widths 2000, 2000, and 1000 with ReLU activator functions. I put a Sigmoid on the output neurons, so I could use binary cross entropy loss instead of regular cross entropy loss because this gave a much more stable training loss, so it was easier to see how the training progressed.

After each epoch the network would be tested on the evaluation set. If it scored a higher accuracy than the previous best, its weights would be saved, and the model that ended up with the highest accuracy would be picked. The highest accuracy I managed to get on the evaluation set was 57.6%. The test set consisted of 51,008 data points. The expected accuracy of making random guesses on the test set was 33.4%. To get an idea of how close our model is to making the correct prediction even if it guesses wrong, we introduce the T2 score. We define the T2 score as the probability that the model makes the correct prediction with its first or second guess. We do not distinguish between whether a card is upgraded or not with the T2 score. The expected T2 score of making random guesses on the test set was 58.3%. Using the model on the test set gave an accuracy of 55.89% and a T2 score of 79.16%.

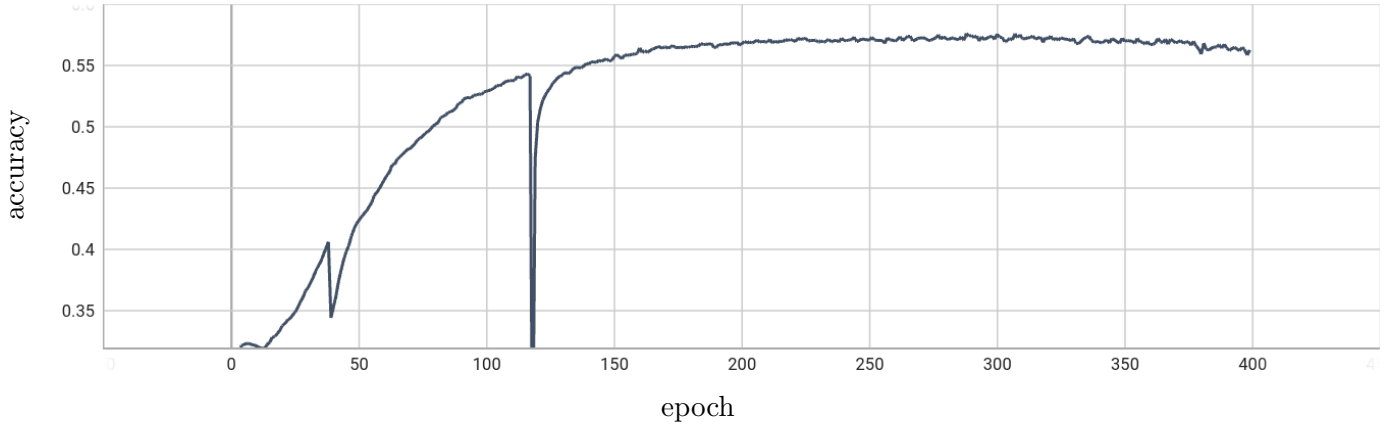


Figure 4.4: The progression of the network's accuracy on the evaluation set as the network is trained.

4.3 Data Augmentation

I wanted to try out a couple of ideas I had for augmenting the training data. The first one was to shuffle the order of the entire hand. Barring any niche cases that I do not know about, the order of the hand has no significance when playing as the Ironclad. In addition to shuffling the hand, there are also some unplayable cards that have little to no effect, which I tried to add to the hand before shuffling it. Both types of augmentation gave models that had a worse accuracy on the evaluation set. Oddly enough, they did though have a smaller loss. I decided to prioritise the accuracy over the loss because the accuracy is based on how we would choose actions with our model in the actual game. Since adding randomness to the data seemed to give worse models, I also tried to move in the opposite direction and set the randomness seed to the same number each epoch. This way the card that was played would be put back into the hand at the same spot each epoch, and it would be the exact same training set each time. Though, this also performed worse than the original training set. The progression of the T2 scores of the various types of data augmentation can be seen in appendix A.1.

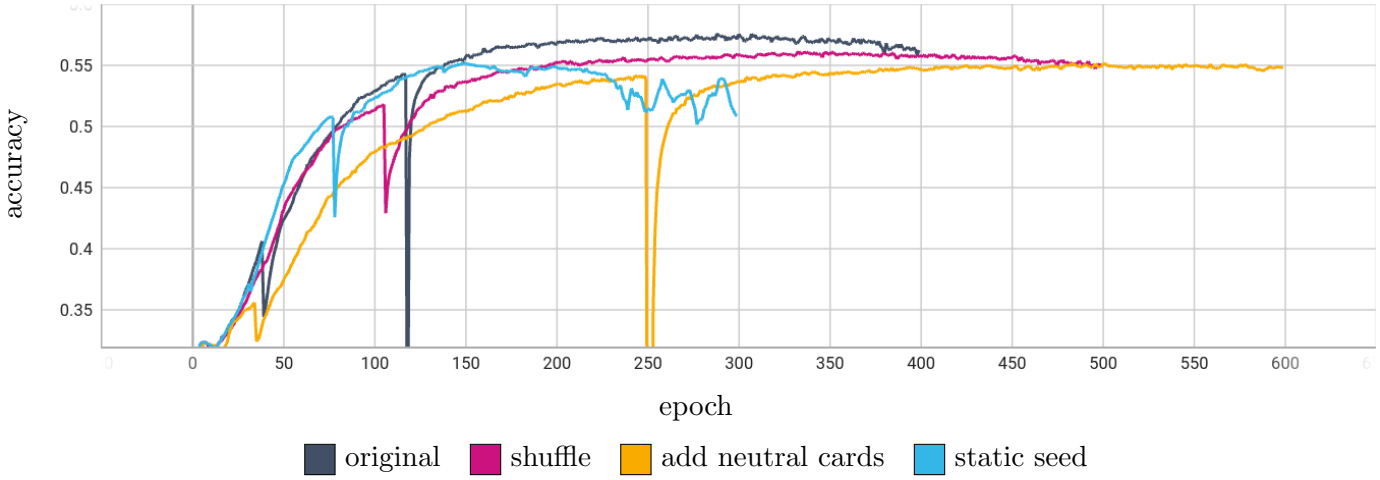


Figure 4.5: The accuracy of the network on the evaluation set when trained with augmented data.

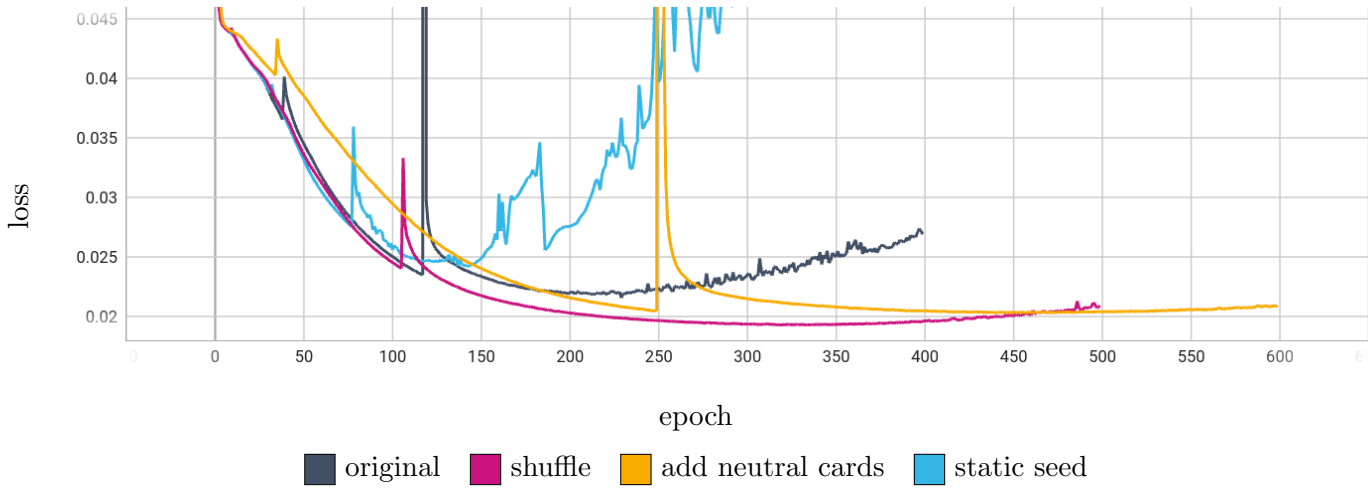


Figure 4.6: The loss of the network on the evaluation set when trained with augmented data.

Testing Against the Environment 5

To get an idea of how good the pre-trained model was at actually playing the game, I modified the Spirecomm AI to consult the neural network instead of its usual combat AI. This AI will be referred to as Shadorbs (portmanteau of shadow and Jorbs). All other decisions in the game would be made according to Spirecomm AI, including which enemy to target with the selected card. I then had this new AI play 100 games of Slay the Spire with pre-specified seeds, where I would log whether the AI won or not, which floor it reached, and what score it got from the game. I also ran the same experiment with the original Spirecomm AI as well as an AI that would play random cards. The performance of the AIs on each individual seed can be seen in appendix A.2.

AI	wins	avg. floor reached	avg. score
random	0	15.2	103.5
Shadorbs	2	20.2	161.2
Spirecomm	8	28.0	264.0

Figure 5.1: Comparison over 100 runs between three AIs for choosing cards in combat: random selection, our neural network, and Spirecomm AI.

As we can see, our network clearly performs better than random guessing, and it did manage to win a couple of times, but it also did significantly worse than Spirecomm AI.

Reinforcement Learning

6

Because Spirecomm can communicate with the game directly instead of having to take a screenshot, analyse it, and perform an action with the keyboard or mouse, it can play the game much faster than was originally expected to be possible. If you also add the SuperFastMode mod [11], you can play through a run in just a few minutes. This means that reinforcement learning might be much more viable than originally thought.

We would like to be able to continue training the network we got from behavioural cloning, so DQN does not make sense to use since the network does not attempt to predict Q-values. We are going to use A2C for this. It would probably be ideal to use PPO, but due to time constraints, this was not implemented. Our actor will use a network architecture similar to the one described in section 4.2, but we will simplify it slightly by always setting the upgrade output to 1, and we have to replace the sigmoid with a softmax so it outputs a probability distribution. Our critic will have the same architecture, except it will also take an action as input, and it will only output a scalar.

I started off with testing the RL setup by having it run through the same game seed so it would have a much smaller task to learn. I also initialised the network with random weights so its progress would be more clear. After a bit of training, the network would end up outputting a 1 for 'Perfected Strike' and a 0 for every other card. It did this even if it did not have a 'Perfected Strike' in the hand. This meant that if it could not play a 'Perfected Strike', it would just play random cards. To try to avoid this problem, I changed the formula for calculating the mean value of a state: All irrelevant cards that were not in the hand would have their values set to 0. This ran into pretty much the same problem, except now the network would only want to play 'Strike'. This is assumedly because 'Strike' is the most common card, so by focusing the whole distribution on 'Strike', it would most reliably avoid the penalty of trying to play irrelevant cards. Because of this, I changed the network architecture further so it will set the outputs of all irrelevant cards to zero *before* applying the softmax function.

The only important end goal in a battle is to kill the enemies while retaining as much health as possible. The only exceptions to this would be if we considered whether or not we used any potions or whether or not we killed an enemy with 'Ritual Dagger', which gets a permanent increase in damage if it kills an enemy. Potions are outside the scope of this project and 'Ritual Dagger' is a niche case, so we will just ignore them for the sake of the reward function. The reward function I used was

$$2 \cdot (hp_{player} - \min(0, damage_{incoming} - block)) - hp_{enemies}.$$

The function looks at the difference between the player's hp at the end of the turn and the enemies' total hp. Since the player will have to fight many enemies and the bosses have much more hp than the player, it seemed reasonable to weigh the player's hp higher than that of the enemies. I initially considered each turn its own trajectory in regards to how the cumulative reward was calculated. I did not discount the cumulative reward since there would only be

uncertainty about future states in certain cases, e.g. cards that let you draw more cards or remove random cards in your hand. I had a replay buffer with a size of 2000 (a run that makes it to the first boss will have roughly 100 actions), and after each run the critic and actor would be trained on roughly as many random samples as there were in the replay buffer. To help ensure that the actor explores all actions, every other run it would have a 10% chance of playing a random card. The actor would still sometimes output non-zero probabilities for irrelevant cards, so I wanted to test if it was best to ignore that or set their values to 0 *after* the softmax as well. Each experiment lasted for 67 runs, and the results can be seen in figure 6.1. Ignoring the irrelevant values reached an average floor of 25.6 with 4 wins, while setting the irrelevant values to 0 reached an average floor of 17.7 with 1 win. It seem clear that ignoring the irrelevant values gives better results. I then ran the first experiment, where I changed the cumulative reward function so it counted an entire battle as a trajectory, and I discounted it with $\gamma = 0.95$. This performed even better with an average floor of 31.1 and 11 wins, so this setup is used for the rest of this section.

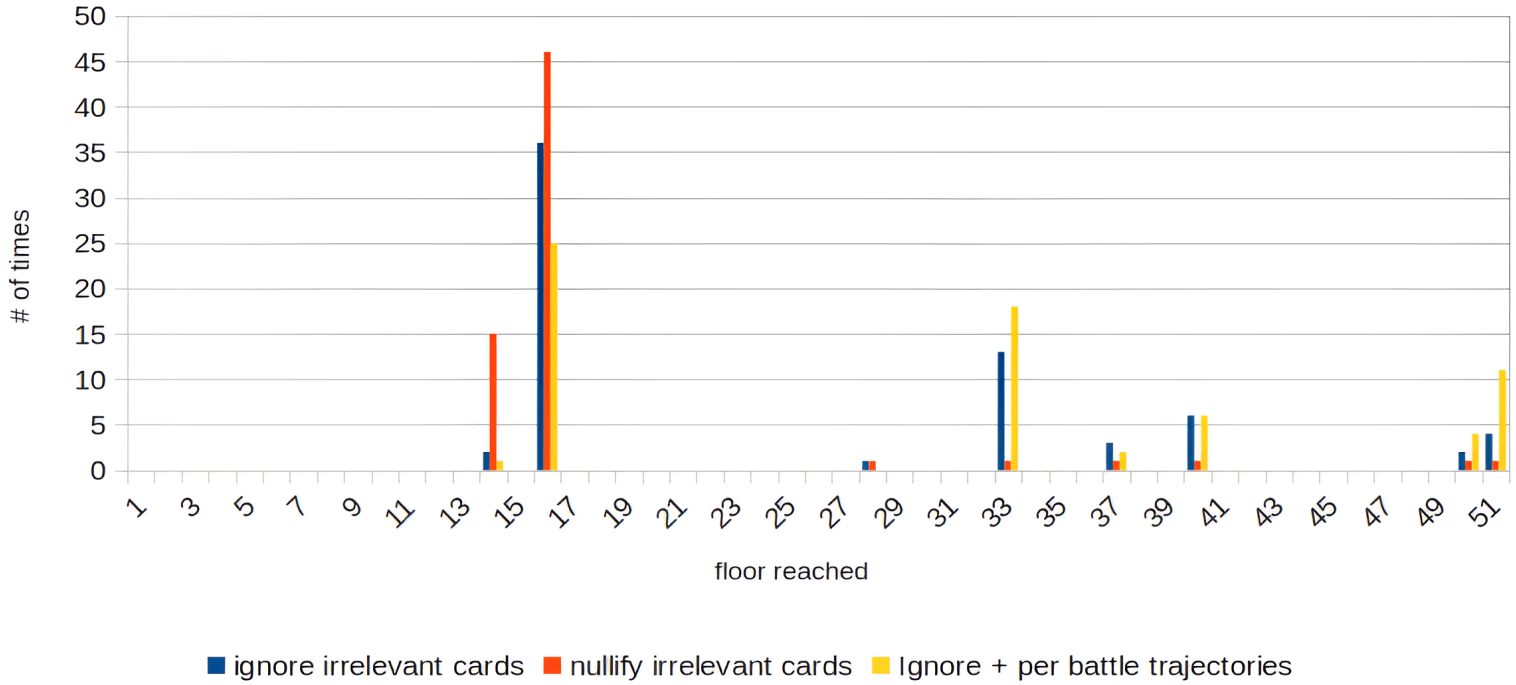


Figure 6.1: Results of 67 runs with the same seed with 3 different setups.

We want to see the training setup's ability to learn to play seeds that it has not seen before. To do this, I trained the model on random seeds instead, and on all runs it would occasionally play random cards. Since this task was more complex, I also increased the size of the replay buffer to 20,000 (it would be filled after 150-170 runs). To test the resulting models, I had them play through the same 100 seeds used in section 5. In these tests the AI would only play the card with the highest probability in the models' outputs. The results can be seen in figure 6.2. 300 training runs took roughly 10 hours. The first model only performed very slightly better than the baseline. Training the model for longer did not clearly improve it, so it would seem that the training had already plateaued. I decreased the learning rate and the number of training samples used between runs and ran the experiment again. This time the model did significantly better than the baseline. Training the model for longer again did not clearly improve it, implying that the training had plateaued. To demonstrate that BC can also be used to pre-train a model for RL, I used BC to train a model using the same architecture as the other experiments. It actually ended up performing slightly better than the model from section 4.2, but this could just be the result of some lucky training. We can see that it performs slightly better than the

other experiments, so we know that the RL setup can be improved, and it is not just limited by the network architecture. Since the pre-training does not train the critic, I just used the critic from the previous experiment. Otherwise, you could train the critic for a set amount of runs before you started also training the actor. Using the pre-trained model for RL clearly made it worse. This strongly implies that the length of the training is not the reason why the RL models perform worse than the pre-trained model.

#	pre-trained	training rate	training runs	wins	avg. floor reached	avg. score
random				0	15.2	103.5
1	no	high	265	0	17.0	118.9
2	no	high	535	1	15.6	109.6
3	no	low	292	0	20.6	159.0
4	no	low	570	1	19.7	150.4
5	yes		0	2	21.9	178.9
6	yes	low	284	1	17.4	128.5

Figure 6.2: Comparison over 100 runs between various RL experiments using the same network architecture.

To get some insight into how the training of the critic went, I logged the average L1 loss for each training batch during experiment #3 in figure 6.2. I then plotted the running average of the 300 most recent points, which can be seen in figure 6.3. It seems like it is very early in the training that the critic pretty much plateaus. This could back up the idea that the training rate was too high. We can also see that the critic has a significant amount of innacuracy when predicting actions' values.

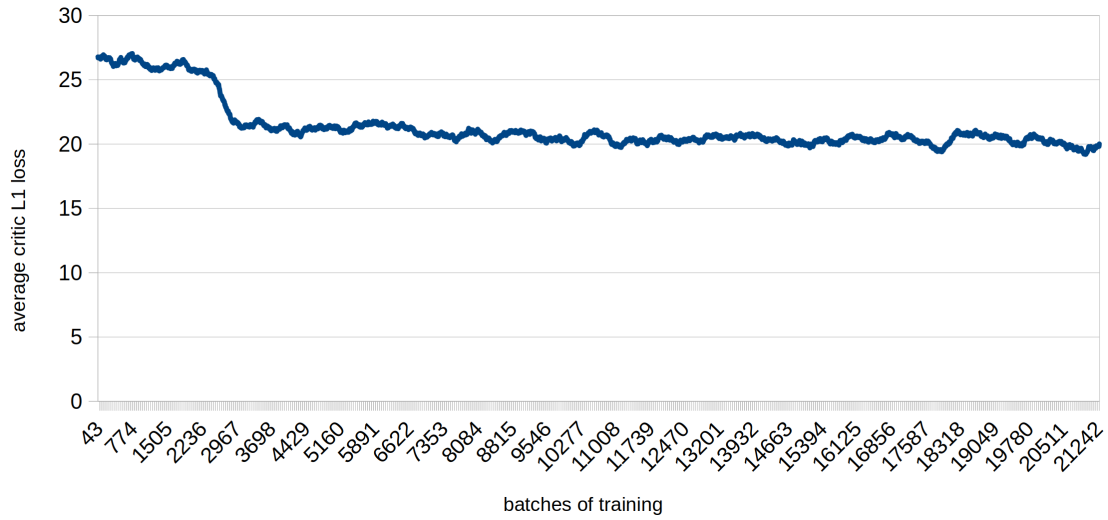


Figure 6.3: The running average of the critic's L1 loss from experiment #3 in figure 6.2. The average consists of the 300 most recent batches.

Discussion

7

The image analysis part was able to take a video and pretty reliably identify which actions were taken and which other possible actions there were, but it covered little more than that. It was able to find 95% of the actions, of which it was able to correctly identify 88%. Of those identified actions it was able to correctly identify 81% of the cards in the hand. It was able to produce a useful dataset, but it did still make plenty of errors. To make the program more reliable, it would be worth exploring some machine learning solutions. Considering that each card has a unique image on it, a convolutional network could probably perform well, and you could use the existing solution to generate training data for it. But the first priority would almost certainly be to extend the video analysis part so it would extract at least most of the information from figure 4.1. A huge limitation of the project in its current form is that the model does not even know how much damage the enemies are dealing, so it will inevitably end up defending against non-existant damage or allowing easily preventable deaths. Keeping track of the contents of the deck would probably be very difficult, so it would be reasonable to tentatively skip that part.

Using the dataset derived from the image analysis part, I was able to train a neural network to the point where it was able to predict the test set with an accuracy of 55.89% and a T2 score of 79.16%. This is clearly higher than the random guessing baseline, which had an accuracy of 33.4% and a T2 score of 58.3%, but it still made a lot of errors. There is plenty more expert data to be retrieved from YouTube, which could improve the training of the model. You could also almost certainly get a better performance by experimenting with different network architectures, but it is also very limited how well you can predict the moves when you are given as little information as the network is. Furthermore, it would not be possible to achieve 100% accuracy because there are cases where there are multiple equally viable actions, i.e. there is one enemy left with 1 hp, and the hand is filled with attacks. I tried out a couple of ideas for augmenting the dataset, but none of them gave a better performance. Though, it would be worthwhile to do a bit more exploration in regards to data augmentation to see if smaller perturbations in the data would have a positive effect: swapping two cards instead of shuffling the whole hand, inserting one neutral card instead of a random number, or only occasionally using the augmented data.

Using the model to play the game, it was indeed able to win, but it was only 2 out of 100 runs, and it performed significantly worse than the simplest one of the existing AIs. It did though clearly perform better than the baseline, so we know that it learned something useful from the training. Again, the AI would probably be able to massively improve if it got more of the information highlighted in figure 4.1. It should also be kept in mind that a significant part of the game is building your deck and choosing your relics, and it is limited how well you can do with a bad deck and relics, so at some point it would make sense to extend the various parts of the project to also handle other decisions in the game outside of battle.

We were able to also train a model with RL within less than a day. This model clearly performed better than the baseline, but it was still slightly worse than the BC model. Decreasing the rate of training seemed to significantly improve the performance of the resulting model, so

it would be worthwhile to see if decreasing the training rate further would increase the model's quality further. We were also able to pre-train a model with BC and train it further with RL. The model became worse after RL training, which further implies an issue with the RL setup. While it might not be the core issue of the RL setup, it would still be ideal to switch from A2C to PPO to ensure more stable training. One big advantage RL has over BC in this situation is that it would be much easier to expand the RL setup to consider more of the game state.

Conclusion

8

In this work, I have presented my pipeline for downloading relevant video data, extracting data points out of those videos, training a neural network on those data points, and using that network to play Slay the Spire without human input. The video analysis part was able to pretty reliably extract a subset of the relevant information from the videos it was given, including the possible actions and which actions were taken. The neural network was able to predict moves from the expert significantly better than random guessing, but it still made a lot of errors. The network was indeed able to beat the game, but it did not outperform any existing solutions. I also made a modification of Spirecomm that allowed a network to be trained, using RL. This method did not outperform BC, but it was still able to make clear improvements in less than a day. We can see that BC and RL can both be used to make AIs that can beat Slay the Spire, but more work would be required to see if they can outperform the other existing solutions.

Bibliography

[1] Bottled ai — slay the spire bot. we made a bot! — slay the spire. https://www.youtube.com/watch?v=GdhHDRVigCk.

[2] Forgotten arbiter. ai defeats slay the spire. https://youtu.be/obtg0tredJg.

[3] Fuzzywuzzy on pypi. https://pypi.org/project/fuzzywuzzy/.

[4] Hugging face deep reinforcement learning course. https://huggingface.co/learn/deep-rl-course/unit0/introduction.

[5] Python-tesseract on pypi. https://pypi.org/project/pytesseract/.

[6] Skimage documentation: feature.peak_local_max. https://scikit-image.org/docs/stable/api/skimage.feature.html#skimage.feature.peak_local_max.

[7] Slay the spire official announcements on steam. https://steamcommunity.com/app/646570/announcements.

[8] Slay the spire wiki. https://slay-the-spire.fandom.com/wiki/Slay_the_Spire_Wiki.

[9] Spirecomm on github. https://github.com/ForgottenArbiter/spirecomm.

[10] sts_lightspeed on github. https://github.com/gamerpuppy/sts_lightspeed.

[11] Superfastmode on github. https://github.com/Skrelpoid/SuperFastMode.

[12] 19/01/2023. User test447 on the data-analysis thread of the official Slay the Spire Discord server.

[13] Chen, B., Tandon, S., Gorsich, D., Gorodetsky, A., and Veerapaneni, S. Behavioral cloning in atari games using a combined variational autoencoder and predictor model, 2021.

[14] Torabi, F., Warnell, G., and Stone, P. Behavioral cloning from observation, July 2018.

[15] Zheng, B., Verma, S., Zhou, J., Tsang, I., and Chen, F. Imitation learning: Progress, taxonomies and challenges, October 2022.

Figure Sources

The following figures are based on screenshots from Rhapsody's YouTube Channel (youtube.com/@RhapsodyPlays):

- Figure 2.1
- Figure 2.2
- Figure 3.23
- Figure 4.1

The following figures are based on screenshots from Jorbs' YouTube Channel (youtube.com/@Jorbs):

- Figure 3.1
- Figure 3.2

Appendix

A

A.1 T2 Score for Types of Data Augmentation

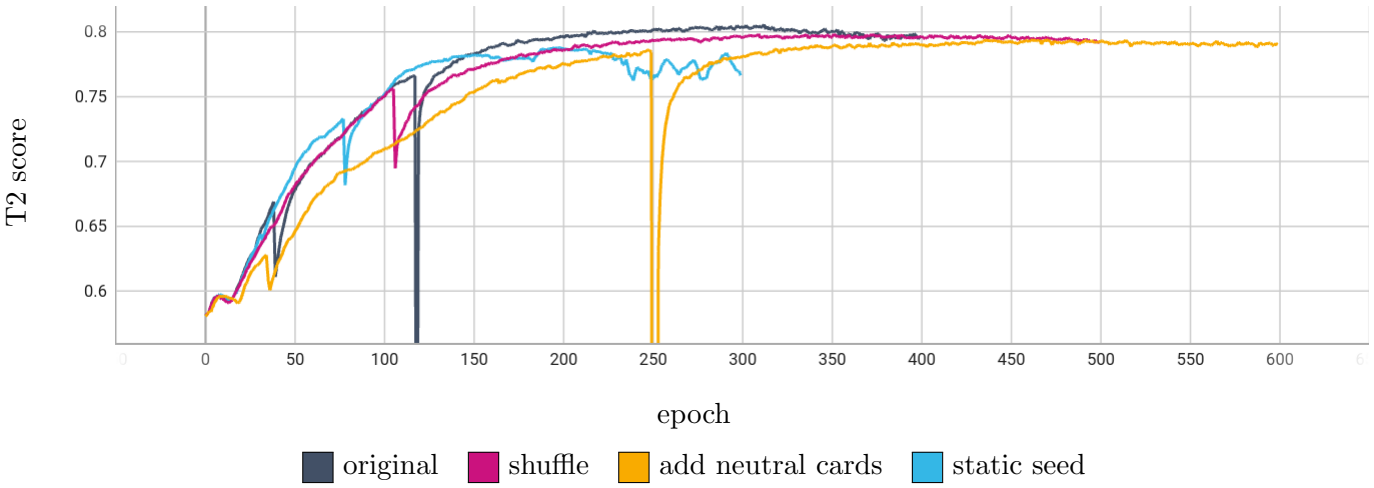
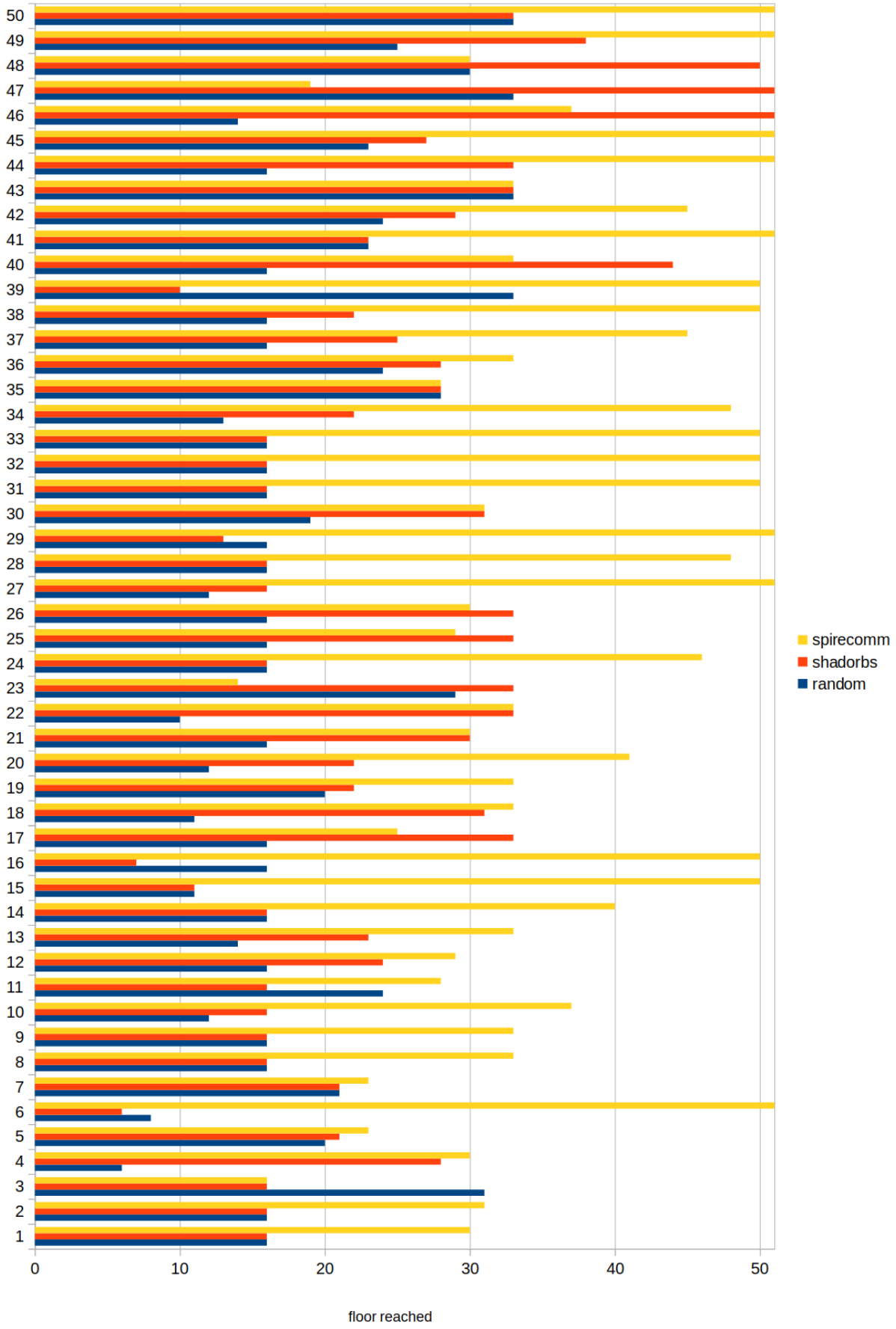


Figure A.1: The T2 score on the evaluation set for different types of data augmentation.

A.2 Environment Tests for Each Seed



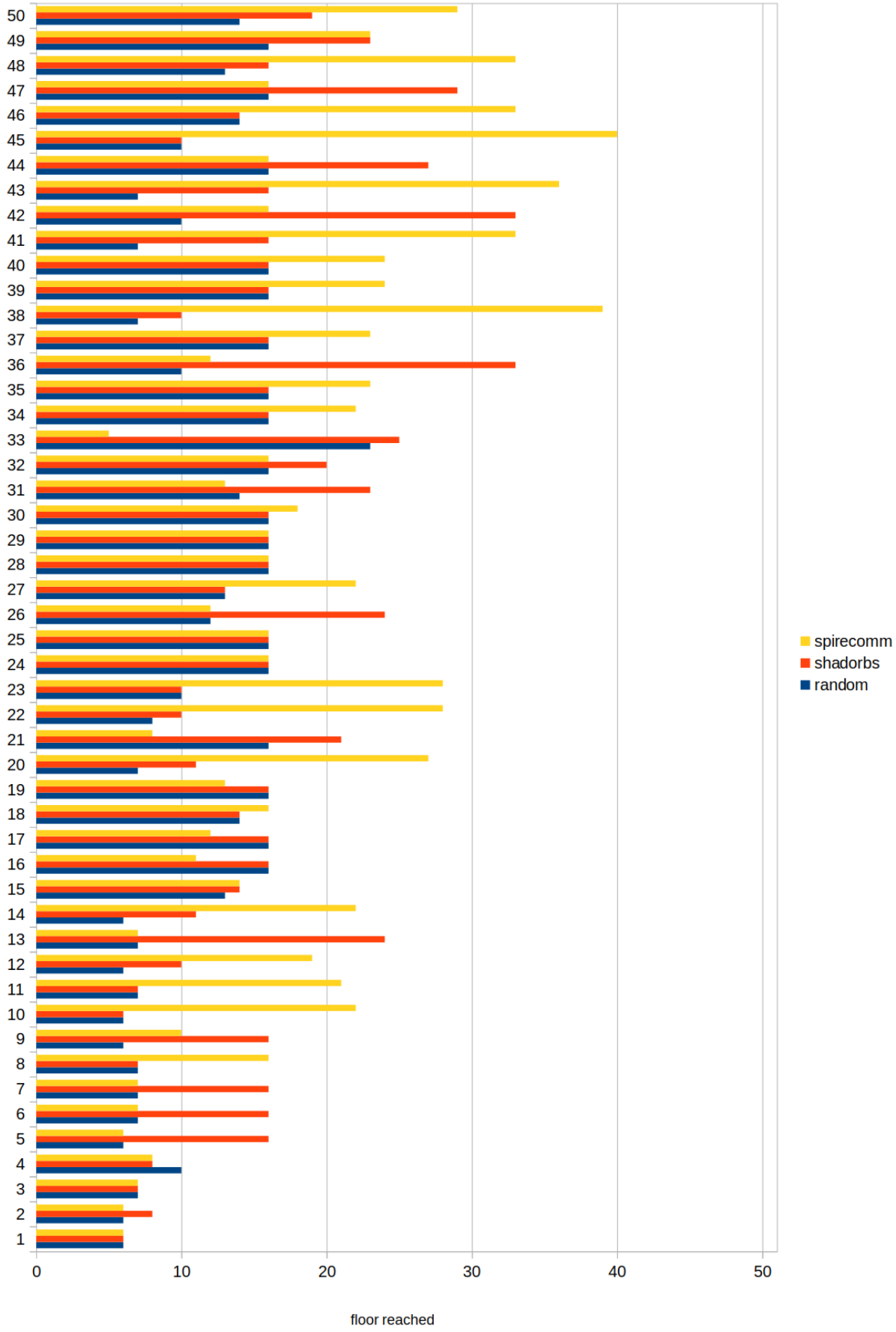


Figure A.2: The above two figures show how far each AI got with each seed during the tests in section 5. The seeds are sorted according to the average floor reached.