Computer Vision - Rummikub

Realisation & Reflection

Bachelor Aplied Computer Science

Sander Backx

Academic year 2020-2021

Campus Geel, Kleinhoefstraat 4, BE-2440 Geel

LID VAN ASSOCIATIE KU LEUVEN

THOMAS MORE

Inhoud

1 INTRODUCTION

I was tasked to create an AI solution to solve a game of Rummikub. I split the project up into two big parts, namely computer vision and a game algorithm. I first focused on the computer vision part where I will be predicting the digit and color. After this I had enough data to build the game algorithm.

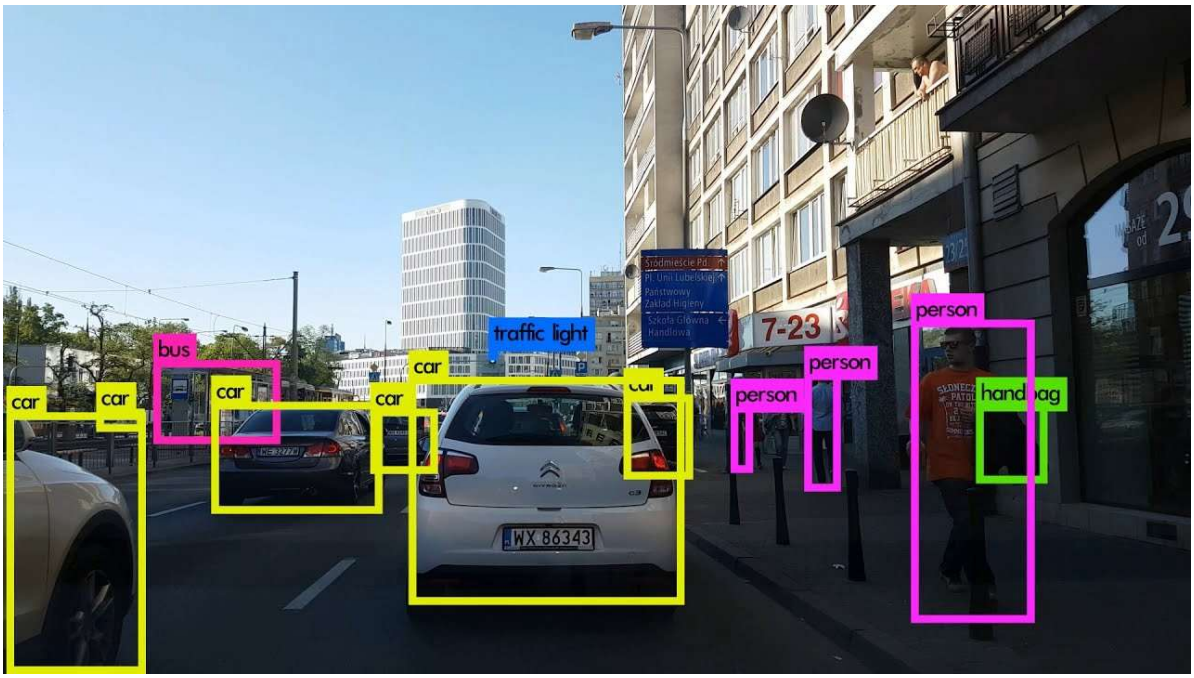
In this document I will go over my research and findings. I will also explain the current workings of the solution.

2 COMPUTER VISION

In this chapter I will explain the different solutions I tried and which solution I ended up using to predict the digit and color of a block. I will show you how I detect blocks, detect sections, detect digits, and detect the color of a block.

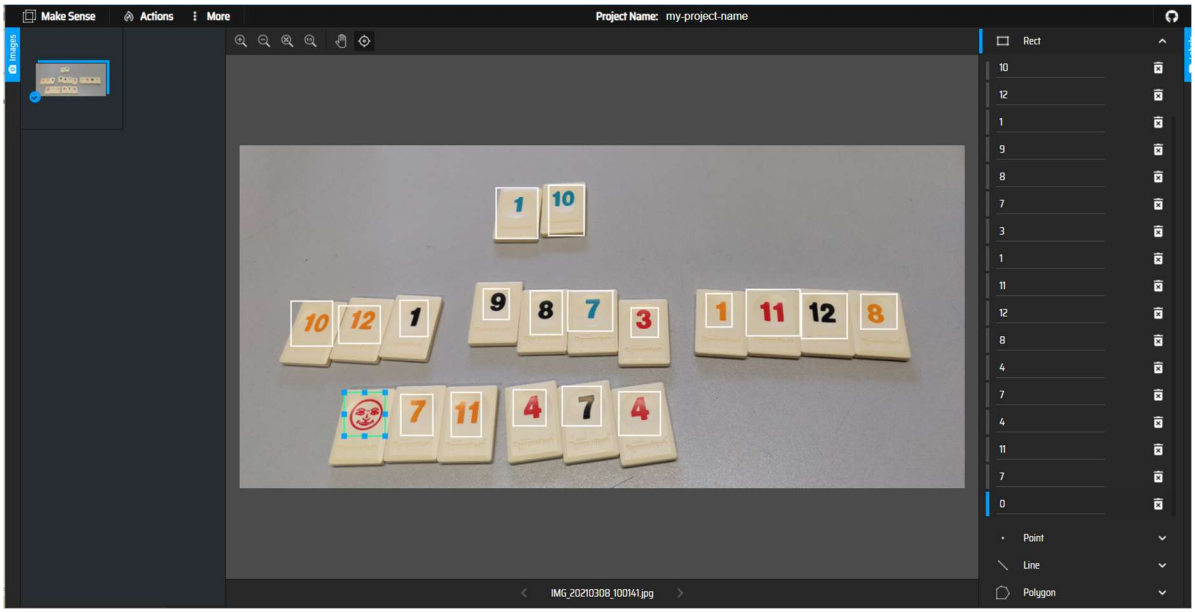
2.1 YOLO

The first thing that came to mind when thinking about a possible solution was building on YOLO. YOLO is a real time object detection system which can be fine-tuned to predict labels on objects you would like to detect.



I first tried using darknet (YOLO v3), but this installation caused some problems. I started looking at another YOLO version and found YOLO v5 which is built on PyTorch.

Once this version of YOLO was working, I started taking pictures of physical blocks and created training data with the use of makesense.ai. I first tried to detect the block digits without the colors. Tagging all the blocks on the photos and training YOLO took a long time and the result was not really that good.



After this I took another approach and tried to detect certain sections of blocks, but the results again were not that promising. For YOLO to work out I would need way more data and way more labelling and I didn't want to sink such a great amount of time of my internship into labelling images and hoping YOLO would work. So, I started to look for other solutions.

2.2 OpenCV

During the course AI Project I was tasked with creating a self-driving car. During this project we made use of OpenCV and Forza Horizon. We took frames from the game and fed this to a neural network which in return gave us controller input to feed to the game. The approach I wanted to use for Rummikub is similar. I already knew how to capture a game and work with these frames, so this gave me a great head start.

2.2.1 Data collection

I am using the official Rummikub app with the LD Player emulator to play the game. Playing the game via an app is way easier to test taking pictures physical blocks. It is also easy to set up a game. You do not have to worry about handing out blocks or playing for the opposing player.

For this project I mainly play two player Rummikub against an AI of the app itself. When I create a game, everything is set in place and the starting blocks are handed out.

I then use Pillow to perform an ImageGrab and grab a portion of my screen. Afterwards I process this image to perform my predictions.

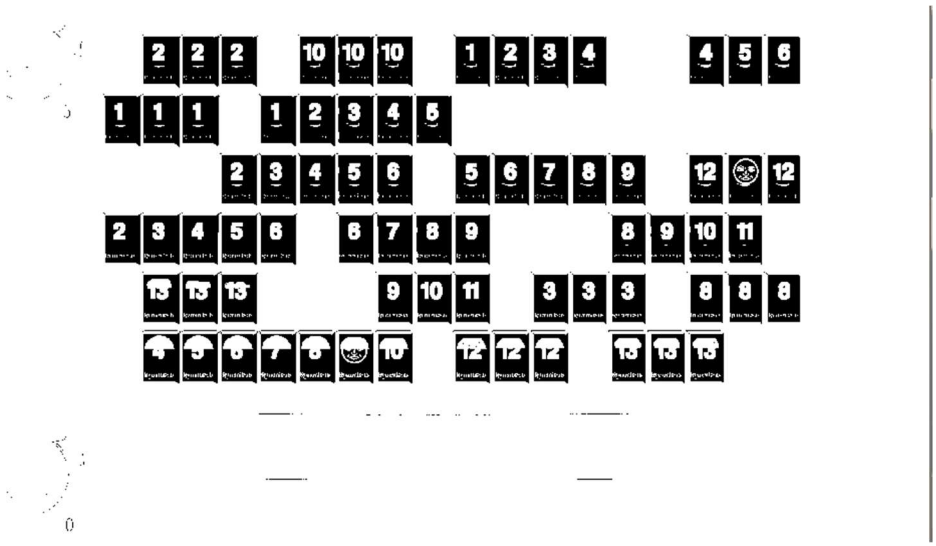
2.2.2 Finding sections and blocks

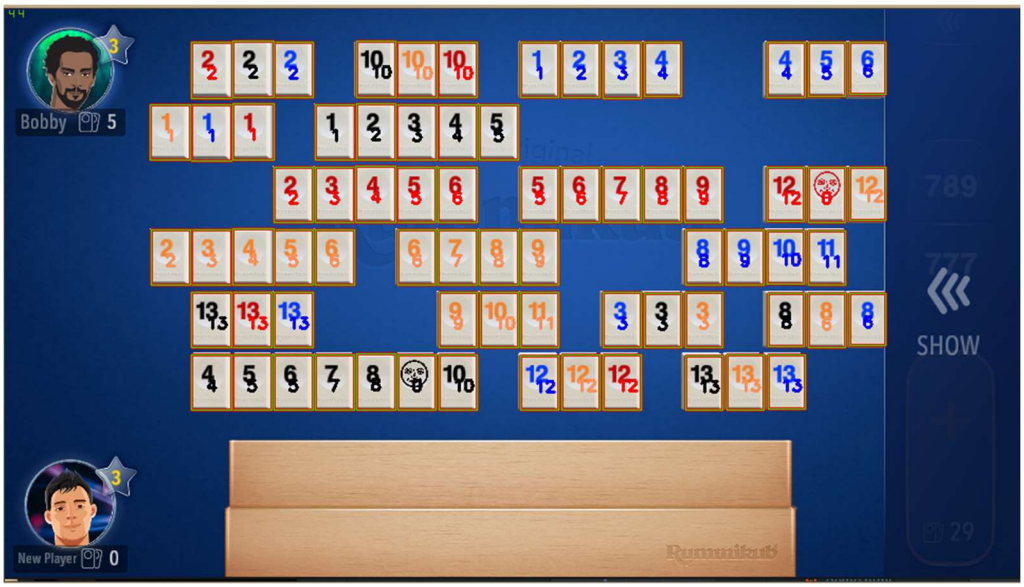
Since I've got an image of the game, my next task was to get all the blocks on the screen together with their corresponding section the reside in.

With using color filters and contours I was able to filter out sections and grabbing those bounding boxes which gave following results.

I managed to get the sections of blocks from the board. After this I was using math to figure out where the actual blocks are. I was counting how many pixels had passed and creating bounding boxes around the actual blocks. This however resulted in many bugs since sometimes these bounding boxes are not really pixel perfect neither are the blocks always the same distance away from each other. Another problem with this solution is that the number of blocks on the screen determine the size of the blocks. So, after a certain threshold of blocks, the blocks get smaller.

While I was messing with the filter to also be able to see what blocks are in my hand, I stumbled across a filter setting where I was not able to see the sections but instead I was able to get bounding boxes around the blocks instead.





This solved all the problems with my previous solution. When the blocks get smaller the filter still picks up the blocks. The distance doesn't matter anymore, and it is easier to process these blocks now that they're almost always the same size. The only problem with this solution was that now instead of finding blocks in a certain section, I must find the section with all the blocks. This didn't seem really easy, but my internship mentor had the idea of combining the two solutions and using the found sections together with the found blocks to determine which blocks are in which section.

I split the screen in two sections, one section where the actual board resides and one where my hand is shown. This makes it easier to perform predictions around the next move to be played.

2.2.3 Predicting

There are many options to predict what digit and color a certain block has. A few crossed my mind such as a Multi Label Classifier from for example Fast.ai, using a classifier to predict the digit and afterwards another classifier to predict the color, using YOLO, … But when diving deeper into this topic I found a simpler solution.

2.2.3.1 Digit

To predict the digit, I grabbed the blocks and created a ROI (Region Of Interest) around the actual digit. There is a lot of space below the digit with 'Rummikub'. This is not needed to predict the digit.

Then when I have only the digit, I used a threshold to convert this image to a binary image with only black and white pixels. This binary image will be converted to a single dimension array to predict the digit.

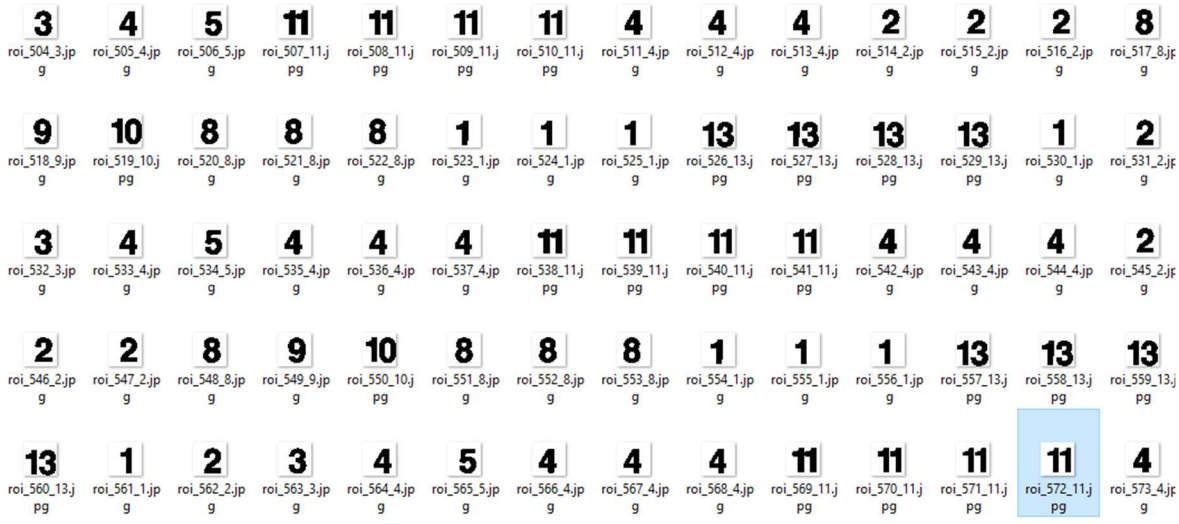
And when I saw this, I was reminded of the MNIST dataset we used in class. This is a dataset with handwritten numbers between 0 and 9.

So, I was thinking about using the same solution for this problem as the solution used for MNIST which is a logistic regression neural network. When comparing data between the dataset and the images my code created there was however a big difference. In the MNIST dataset a 4 is drawn in multiple ways, one is a bolder 4, one is a pointier 4, … but my dataset always has the same 4. I did not think I would need a neural network to solve the issue of predicting the digit since they all look the same. So, I started to do a little research and found out the easiest solution is to use a K-Nearest Neighbors algorithm.

I believe it is better to find the simplest solution possible and if this solution is not sufficient, try a more complicated solution. The simpler solutions often need way less computing power and are faster.

The K-Nearest Neighbors algorithm puts the data in certain categories (1 – 13 in this case) and when given a new image to predict, it looks at which category is the closest to this new image. When testing this solution, I got promising results.

After finding a solution I had to create data. This took some time but not nearly as much as it would have taken if I took the YOLO approach.
I played some games and while playing I often processed and saved the current blocks on the screen. This resulted in a big folder of images that I labelled by putting them in folders with the corresponding digit as name.



I then dragged these images in folder corresponding to the digit (digit 1 to a folder named '1', …) and when I thought I had enough of each digit I used a jupyter notebook to train the K-Nearest Neighbors algorithm.

```
for i in range(14):
    directory = f'resources/{i}/'
    count = 0
    for filename in os.listdir(directory):
        if filename.endswith('.jpg'):
            im = cv2.imread(f'{directory}{filename}')
            im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
            npim = im.reshape(-1).astype(np.float32())
            if count > 350:
                test_X.append(npim)
                test_y.append(i)
            else:
                X.append(npim)
                y.append(int(i))
            count+=1
            continue

np_X = np.array(X)
np_y= np.array(y)
```

```
neigh.fit(np_X, np_y)
```

```
KNeighborsClassifier(n_neighbors=4)
```

```
np_test_y = np.array(test_y)
np_test_X = np.array(test_X)
```

```
correct = 0
for i in range(len(np_test_X)):
    predict = neigh.predict([np_test_X[i]])
    actual = np_test_y[i]
    if actual == predict:
        correct +=1
    else:
        print(i)


print(correct/len(np_test_X))
```

I then exported the pickle file from the notebook and imported this file in the main code where I then predicted the block digits that are on the board. To make it easier to debug I also draw those prediction on these blocks. This makes it easier to spot mistakes. Accuracy is around 98% with mistakes mostly being between 3 and 8 and between 10 and 13.

2.2.3.2 Color

When I found a solution for the digits, I then tackled the problem of prediction the color. I used a similar approach as with the digits. I grabbed the block and used a ROI around the digit itself.

When I did some research around good ways to find out which colors are in an image I stumbled across the K-Means algorithm. K-Means splits the data up into a predefined number of clusters. I used two clusters which resulted in me getting two colors back. One of the colors is the greyish background you see on every block, and the other color is the color of the digit (for this 10 it's black).

This image is then converted into a single dimension array to be used by the K-Means algorithm.

The greyish background does not matter but since it is the most present color this will always be one of the clusters. The other cluster is the one we need to determine what color the block is. We use the RGB value from this cluster to find the correct label.

The actual RGB value is different for every digit since there is more black in a 10 than in a 1. So, when I got these results, I had to set boundaries where I say which RBG values fall in certain color regions. For example, if an RGB value is between 240, 60, 5 and 260,85,40 it is considered a blue digit. These regions I found with testing these predictions and fine tuning the thresholds.

I also drew these colors on the block so it is easier to debug.

I now can predict the digit and the color of a block pretty accurately. If there is a wrong prediction it is easy to readjust these thresholds to fix these errors.

Detecting blocks	99%
Detecting sections	99%
Detecting numbers	98%
Detecting colors	99%

99% because I rarely found mistakes.

3 SOLVING THE GAME

Now that I can find the color and digit of a block, I am ready to try to solve the game. I want the algorithm to provide me instructions on moves I need to make, and I want to try to win a game using only these instructions.

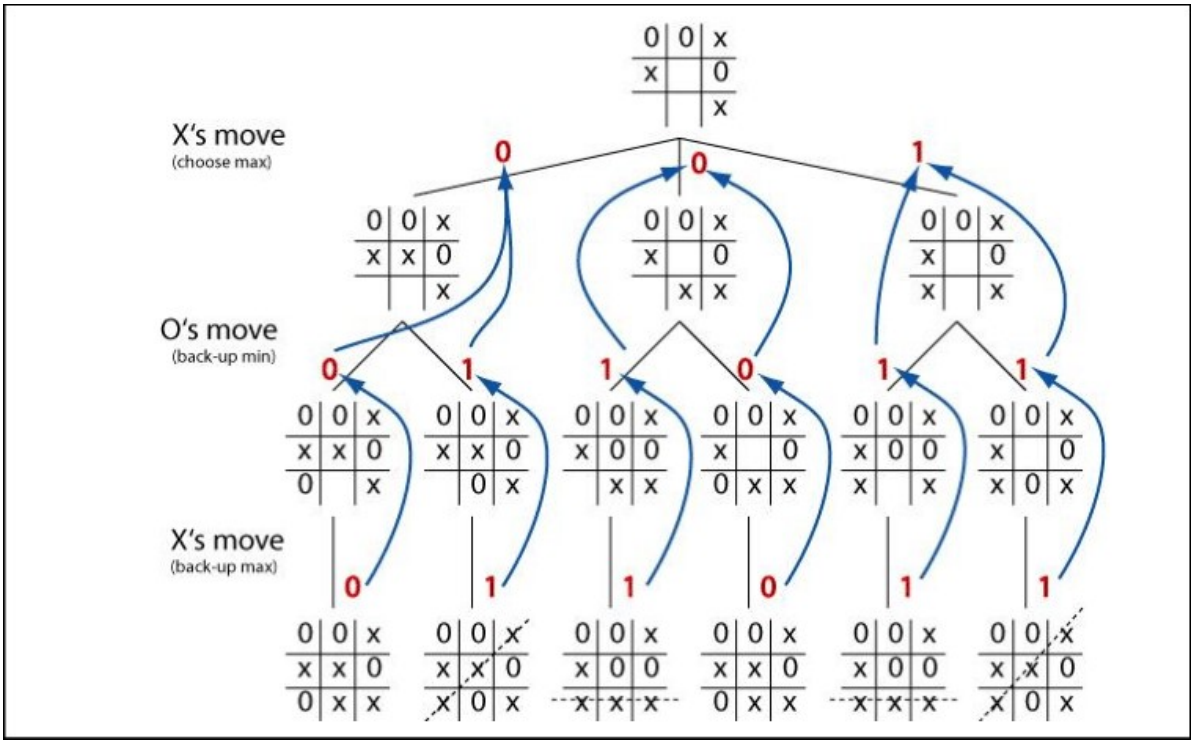
3.1 Research

In this part I will show you different solutions to solve games. I will touch on why I did not use these solutions for Rummikub and finally I will explain the solution I will use for Rummikub

3.1.1 Minimax

Minimax is an algorithm we learned about during our course AI Project. This algorithm focusses on finding the best possible next move. It creates all possible states the game can have and tries to find a state where the player wins.

The problem with this solution however is that Minimax needs all possible states and in Rummikub there are way too many possible states. You could move all the blocks on the board to whatever valid position you want and place your own blocks in whatever valid position you want which results in a lot of possible states.



3.1.2 NEAT

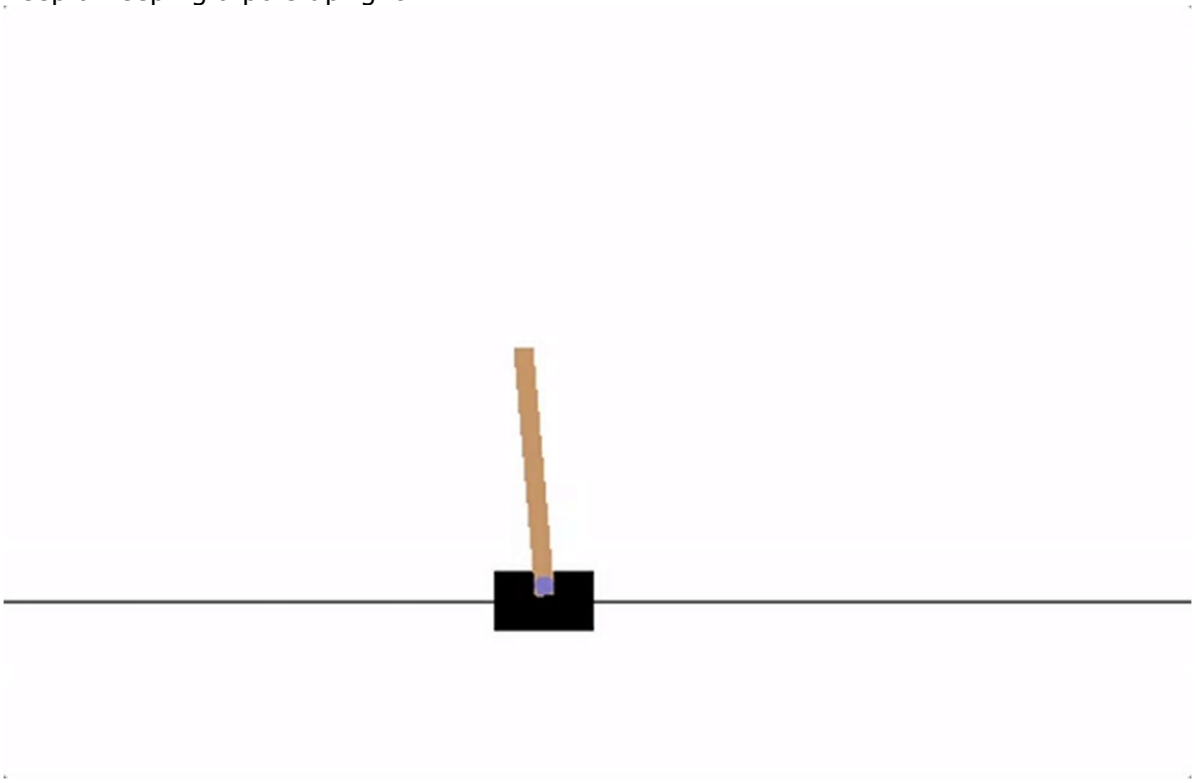
Neat is an evolutionary algorithm that creates a neural network. A certain agent has node genes and connection genes. The node genes tell the agent when to fire a certain neuron and the connection genes represent a connection between neurons. When the agent plays the game and dies its fitness is calculated. The agents with the highest fitness 'reproduce' and new agents with the same settings are created. When creating these new agents, they have a chance to mutate the weights of the neurons. This process results in agents that have mutations which result in better fitness reproducing more. In turn mutating again which can result in better agents. And so on.

This solution might not work for Rummikub since there are many actions to take. Placing certain blocks, moving certain blocks, … It is not as black and white as the example in the image.

3.1.3 DQN

DQN is Reinforcement learning with the usage of neural networks. I did not really dive deep into what DQN actually is and how it works since this is also hard to use for Rummikub where there are many actions to be made and where the environment isn't easy to translate to a DQN environment.

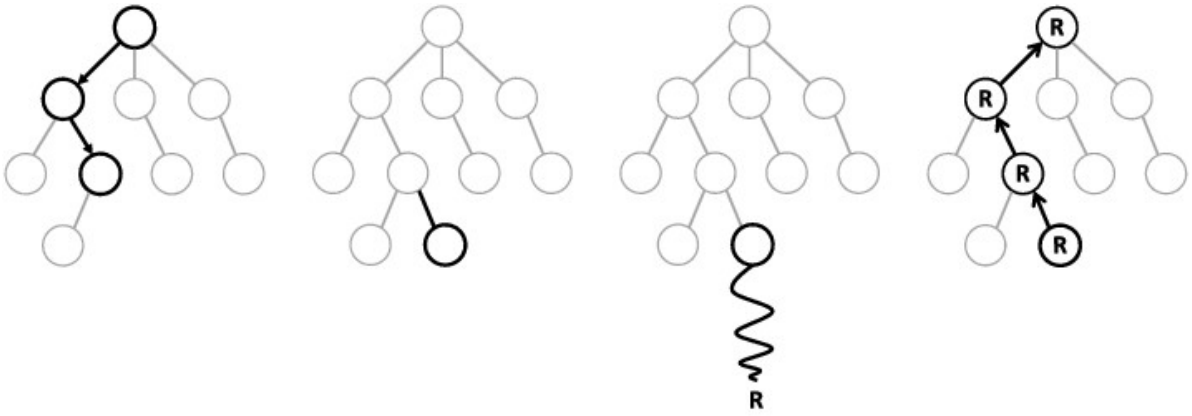
DQN is where you train an agent using images to do certain actions for example how to keep a keeping a pole upright.



This solution is also used to solve old Atari games.

3.1.4 Monte Carlo Tree Search

A friend of mine used MCTS (Monte Carlo Tree Search) to solve the game of Risk. Therefore, I also looked at MCTS as a solution I could use for Rummikub. MCTS is a Reinforcement Algorithm that looks at the possible states, picks a state and then simulates this state. If the state results in a win, a loss or no more possible moves it gives this state a certain score. After visiting and simulating enough states it picks the state with the highest score.



(a) Selection (b) Expansion (c) Simulation (d) Backpropagation

This algorithm solves the 'too many states' problem the minimax algorithm faces. The only issue with this algorithm is that this algorithm only works for fully observable games which Rummikub is not. To be able to bypass this I made the game fully observable and filled the other player's hand with random blocks that are still in the game. You could also try and scope this to only one turn. This way there is no need for the other player his hand to be visible to the algorithm. However, using random blocks is not a big problem since it is only using blocks still usable in the game. It is excluding blocks already on the board or in the observable hand. There are also many simulations which can result in many different unobservable hands.

3.2 Reconstructing the game

To be able to solve the game I first needed to reconstruct the game. I initialized a game with all the possible blocks in the game and after using the predictions to find out which blocks are on the board and which blocks are in the player's hand, I could figure out which blocks are still in the game. I used all this information to create rules around valid moves you can make. Valid sections exist of two possible variations. One is blocks with the same color but digits counting upwards like 4 black, 5 black and 6 black. One is blocks with different color but the same digit like 2 blue, 2 yellow, 2 red. All valid sections must have three blocks or more. There is a joker block which can be any digit and any color.

3.3 Using Monte Carlo Tree Search

When I had all the information needed, I started on the algorithm. I wanted to first do only simple actions and build up from there. The first action I wanted the algorithm to make was adding one block found in the player's hand to a valid section on the board.

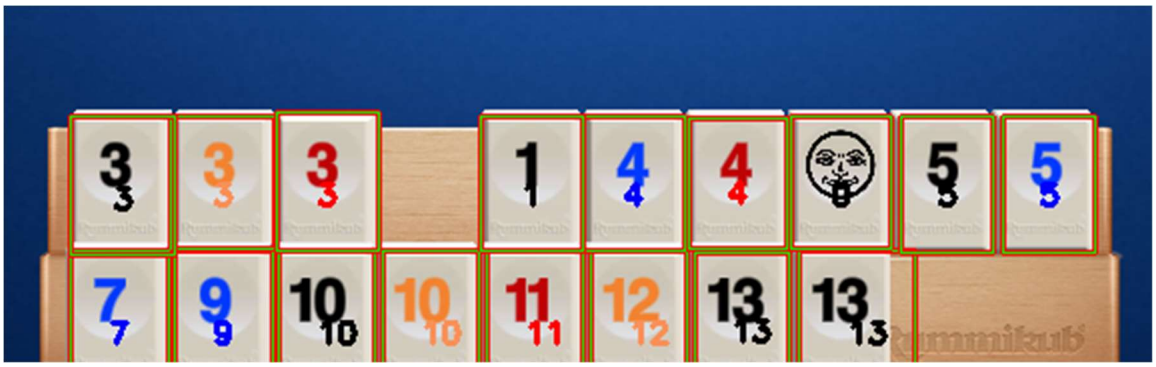
Instead of making the algorithm move the blocks, the algorithm gives suggestions to the player.


```
------------

Move block 10 red to section ['11 red', '12 red', '13 red']
------------
```

The algorithm sees that you can move the red 10 block to the section with 11 red, 12 red and 13 red and instructs you to do so.

Everyone move is a node in the algorithm. When you follow the algorithm's instructions you do another round of predicting until the algorithm finds no moves to be made.

After moving one block from the hand to a valid section worked, I started to add more moves that the algorithm can make. The next move I added, was looking at your own hand and finding new sections to be made with those blocks. For example, if you have a 1 red, 1 blue and 1 yellow in your hand the algorithm will now recommend you move these on the board as a new section.

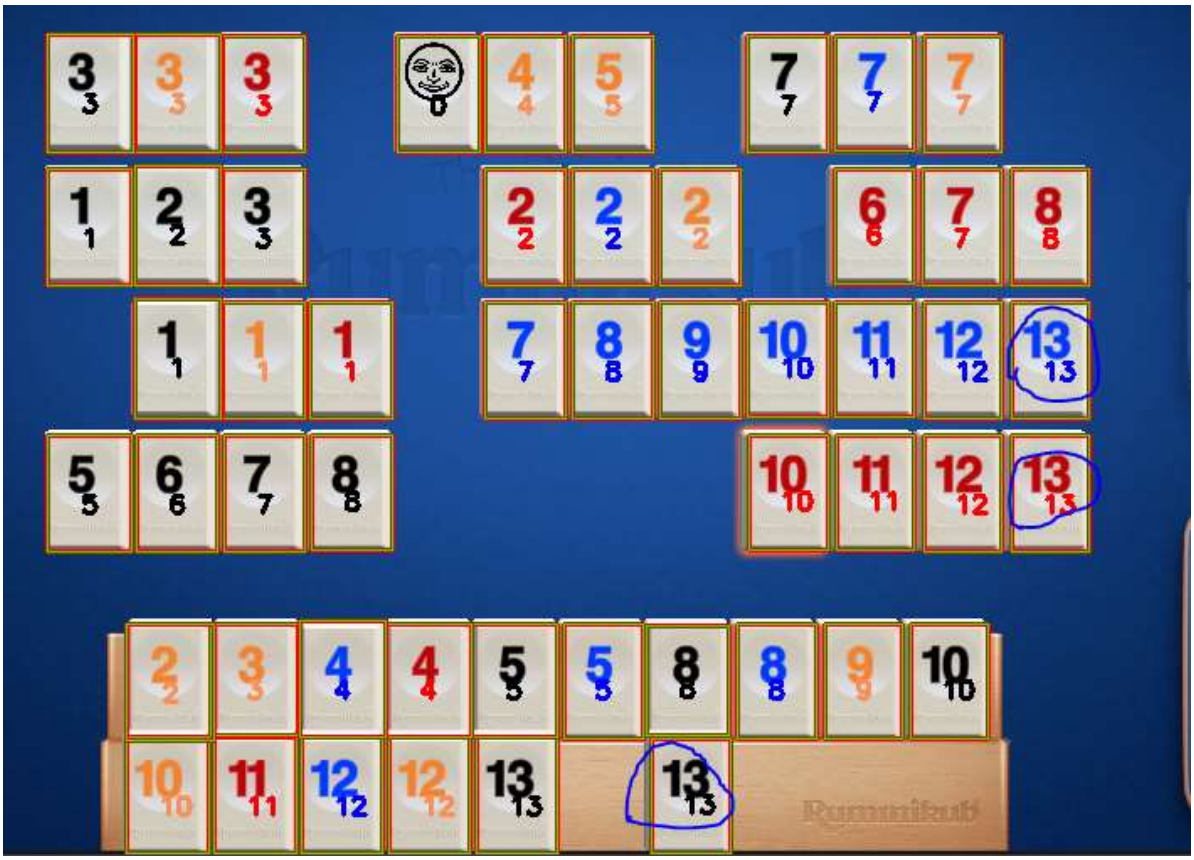


```
------------

Move blocks [ ['3 yellow', '3 black', '3 red'] ]  to the board (new section)
------------
```

The next thing to tackle was not only adding blocks to the boundaries of sections but also adding blocks in the middle to split up sections if it results in two valid sections. For example, if you have a section with 1 blue, 2 blue, 3 blue, 4 blue and 5 blue. You could add another 3 blue to split this section up into two new sections. 1 blue, 2 blue, 3 blue and a section 3 blue, 4 blue and 5 blue.


```
------------

Move block 4 blue to section ['2 blue', '3 blue', '4 blue', '5 blue', '6 blue']
------------
```

The final and more difficult thing I added was the ability to use blocks already on the board to be moved. I find all the sections with more than three blocks and look at the blocks on the outside. Figure out which one I do not have in my hand already and temporarily place these blocks in my hand. This way there are new valid moves to be found with these blocks. This addition opens a lot of options the algorithm can make. The only problem this solution still has is that the instructions when using blocks already on the board are not clear what blocks are talked about. This is still something to be fixed.



```
------------

Move blocks [ ['13 black', '13 red', '13 blue'] ]  to the board (new section)
------------
```

Next would have been also being able to use blocks on the board splitting sections in half and still resulting in valid sections. For examples 1 blue, 2 blue, 3 blue, 4 blue, 5 blue and 6 blue taking the 4 blue and using it in another move.

The algorithm looks at which blocks are on the board and which blocks are in the player's hand. After this it will try to find all possible moves to be made. When these are found the algorithm makes one of these moves and simulates again with the newfound board. After the algorithm can no longer make moves or if a winner is found the algorithm uses a value function to see if this set of actions was a good set of actions. The value function first looks if there is a winner. If the player wins the reward function will return 100. If there is no winner, the reward function will return the number of moves made + 0.5 and if the other player wins the reward function will return 0. In the end the algorithm will return the most profitable instruction to the player.

There also are bugs to be fixed with this algorithm for example having a joker in your hand will result in the algorithm instantly recommending the joker to be used instead of holding on to the joker. The algorithm not finding certain moves that should be made but instead telling the player there are no valid moves, …

These bugs are still to be fixed but there is already a general idea of how this program would work.

4 CONCLUSION

In this chapter I will conclude what I have realized. I will give a recap on the computer vision and algorithm.

4.1 Computer Vision

I have realized a solution that can predict the digit and color of blocks in the official Rummikub app. This is done by using K-Nearest Neighbors for digit prediction and K-Means for color prediction. Mistakes can still happen, mostly between 8 and 3 and between 10 and 13. When moving these blocks around and predicting again, these mistakes are often resolved.

4.2 Game algorithm

The algorithm can recommend moves to the player. These moves exist of moving one block to the board, move one complete section to the board and this with using blocks from the player their hand or blocks of sections with more than 3 blocks.

There are still things missing such as using blocks from sections with more than 7 blocks and using the middle block. Also being able to leave unfinished sections and afterwards completing this section before the turn is over.

5 WHAT'S NEXT

Since this project is not completed yet here are some ideas of what I could be improved on next.

5.1 Computer vision

The computer vision aspect of this project is already solid but there are still things that could be improved. Sometimes there are mistakes between 5 and 8 and mistaking a 13 for a 10. These are mainly when the blocks are on the edge of the board.

It might also be useful if the algorithm has knowledge of where the blocks are on the board. This way it does not need to recommend moves but might be able to move the blocks.

5.2 Game algorithm

The algorithm can use improvement. There are things that work but there are still things to be implemented. For example, the algorithm only recommends one move at the time. It would be good if the algorithm could recommend a set of moves.

It would also be good if the algorithm could move the blocks more freely. Currently the algorithm can only do valid moves. Which means that the algorithm cannot leave invalid sections which afterwards can be valid sections. You often see this at the end of the game where there are many blocks on the board, and you only have one or two in your hand.

Finally, it would be nice if the player does not need to move the blocks himself, but it is instead all automated. This is more of a quality-of-life feature and would best be implemented after everything else is implemented.

6 PERSONAL REFLECTION

In this chapter I will reflect on my personal competencies and what I learned during this internship.

6.1 General

This internship is the first time I have worked on a project for this long. Taking times to plan everything out has helped to keep me on track to deliver a result for this project. With great feedback from my internship mentors, I created a good schedule that I could follow. I had my doubts on how everything will work out with working from home but in the end, everything went very smooth, and I might even prefer working from home.

The stand-up meetings we had during the week were very helpful. These were times where I would update my mentor and could ask questions about my current struggles. This has showed me how important it is to be able to explain what you are doing and what you are struggling with. It also helps to explain problems in a simple way, this makes it easier to find the root of the problem. It also helps when you have another view on the problem that can give suggestions on how to fix it.

I am not very good at documenting work, but my mentor always gave really good feedback and gave me good tips. This feedback and these tips will certainly help me in the future.

I have also grown in making a good schedule and following this schedule. This time there was more freedom in creating a schedule than I was used to from school, and this helped in creating a schedule that works for me. I made the schedule broader (for example 'detecting blocks' over two weeks instead of smaller segments with more detailed titles) and this in turn made it easier to follow this schedule. I was able to start the day with a certain idea of what I would like to accomplish that day, work towards that goal, and do this until I finished the broader schedule item.

6.2 Competencies

In this part I will dive deeper into the competencies and what I learned in the scope of these competencies.

6.2.1 Analyzing

I have improved my analyzing skills. Getting to know about the task at hand and afterwards trying to figure out the different steps I would have to take. Then figuring out which solutions work for which task and doing research on these solutions. Trying out different things and figuring out what works. For example, starting with YOLO and ending up with K-Nearest Neighbors and K-Means.

6.2.2 Realizing

I have learned how to build a solution in small steps. Doing small steps is easier than doing everything in one big step. I have also learned more about how important it is to document what you are doing.

6.2.3 Communicating

During this internship I had many stand-up meetings with my assigned mentor. During these meetings we would talk about what I did the last few days, where I struggled and

what I will be doing next. This showed me how important it is to be good at explaining what you are doing.

I also used GitHub to share my code with my mentor. This way my mentor could keep track of what I was doing.

6.2.4 Project-based work

This internship showed me how much easier project-based work is when you have a schedule you can follow. Taking some time for planning the project and setting up internal deadlines proved to be very helpful. This way there is something you can fall back on when looking at the progress of the project.