LAB ON MACHINE LEARNING AND APPLICATIONS IN INTELLIGENT VEHICLES

Simple traffic racer

Christian Reichmann

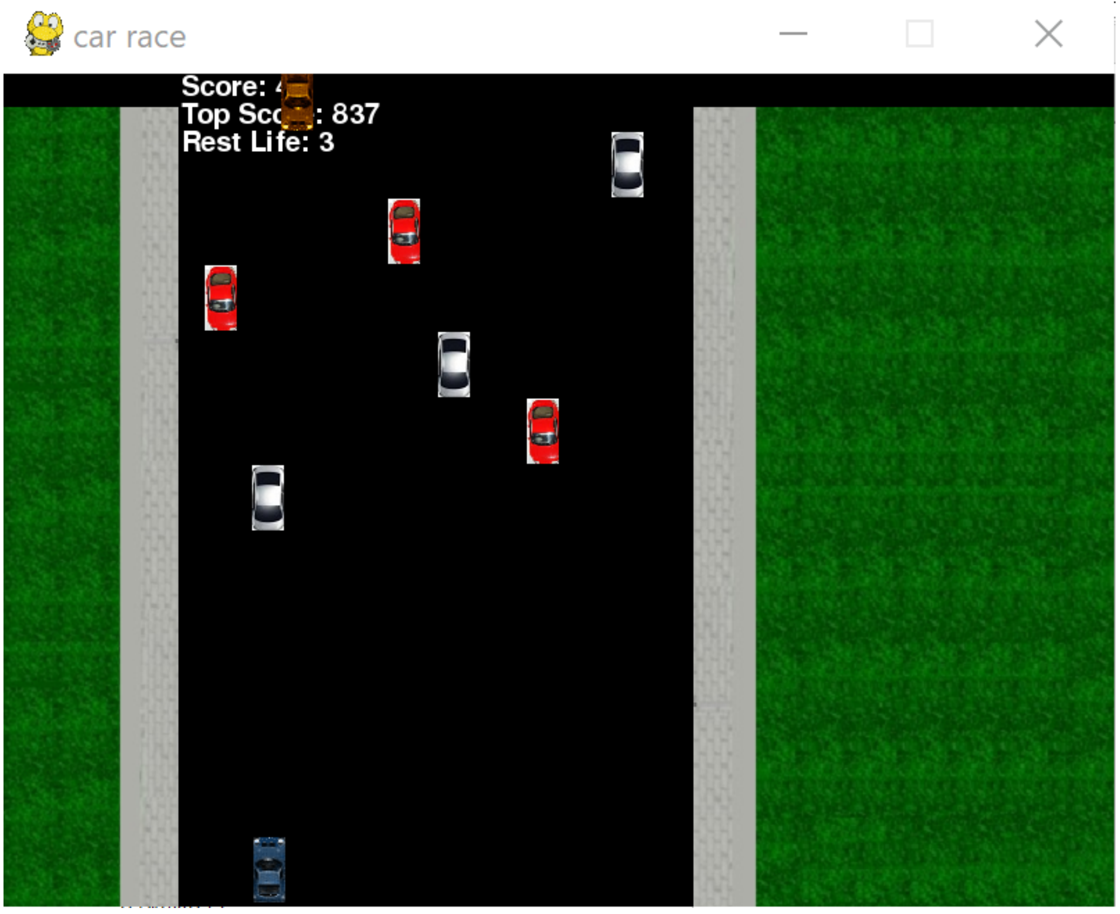
51813605

July 30, 2019

Contents

1 Introduction

The following report, covers the final project assignment in Lab on Machine Learning and Applications in Intelligent Vehicles. The goal is to create a smart AI that can drive a car without crashing into other vehicles or walls.
For this, a base project is provided, called simpletrafficracer. It is a simple game, where the user can control a car using the keyboard arrow keys. Other cars are coming from the opposite direction and the goal is to not crash into any obstacles. See the following picture for a simple screenshot of the game:



The goal should be reached using any kind of machine learning approach (imitation learning, reinforcement learning).

2 General program description

This chapter covers shortly a description of the program that was provided initially.
The game is provided from the pygame community. Initially the game is just controllable via the keyboard arrows. The version that was handed out, already includes an approach for reinforcement learning. In this chapter, only the existing reinforcement learning approach is

described, not the game itself.
The learning algorithm bases on a neural network using 2 convolutional layers, both with relu activation function, maxpooling with 2,2 pool size and 2,2 strides and a batch normalization. Furthermore, a flattening layer is added, connected to a 1000 neurons dense layer with relu activation function. Attached to that, a dropout layer is added and another batch normalization. The whole network leads to a final dense layer with the same number of neurons as possible actions (4). The whole network is created using keras.
The input to the neural network is the currrent state, which is represented as the last 4 frames of the game. The reward function is simply 0 except when the car hits an obstacle, then it is set to -1.
The algorithm starts with an observation phase of 5000 cycles, where the actions are just executed randomly to fill the replay memory.
Once the observation phase is finished, the reinforcement learning starts. This is done, by taking a minibatch out of the replay memory (10000 states) and training the network based on the current prediction of the network and the actual next state, with regards to the reward target. The actual action is then taken from the current state and the neural network prediction for this state.

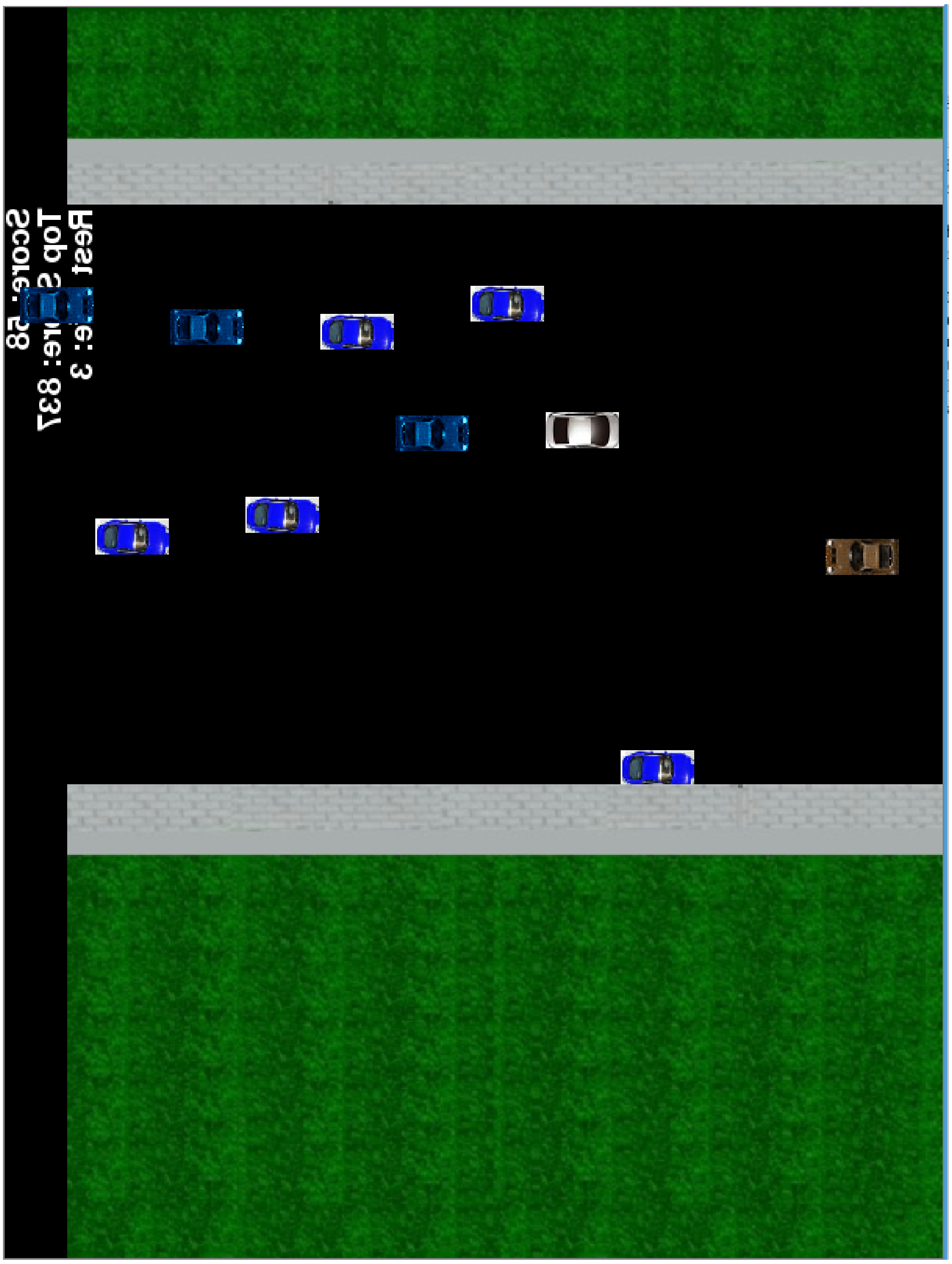
3 Performed changes in the approach

Generally a few small bugs (intentionally inserted) have been found and fixed. One example for this, is that the current state s_t was never updated. Not all bugs that were found are discussed here.

3.1 State representation

The first change in the code, is the state representation. Currently the state is represented as a 80x80 colored image. It is understandable, that this approach contains way too much information for our actual goal as we do not care about colors of the cars or similar. Therefore, the state representation was converted to a 80x80 grayscale image. The following steps are performed to do so:

1. First the input image that was used until now is shown:

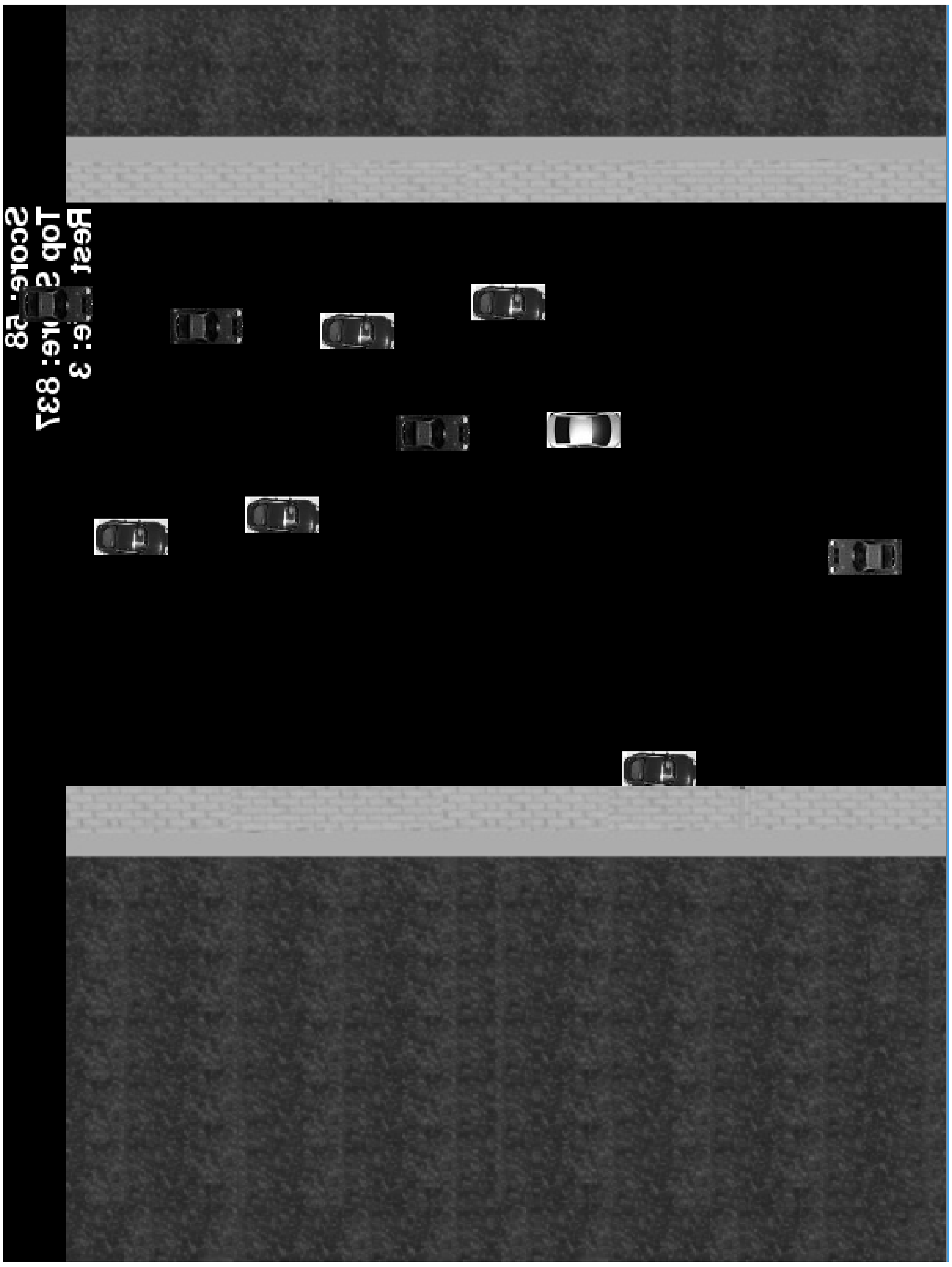


2. The next step is to convert the above image to a grayscale image. The following code is

used for that:

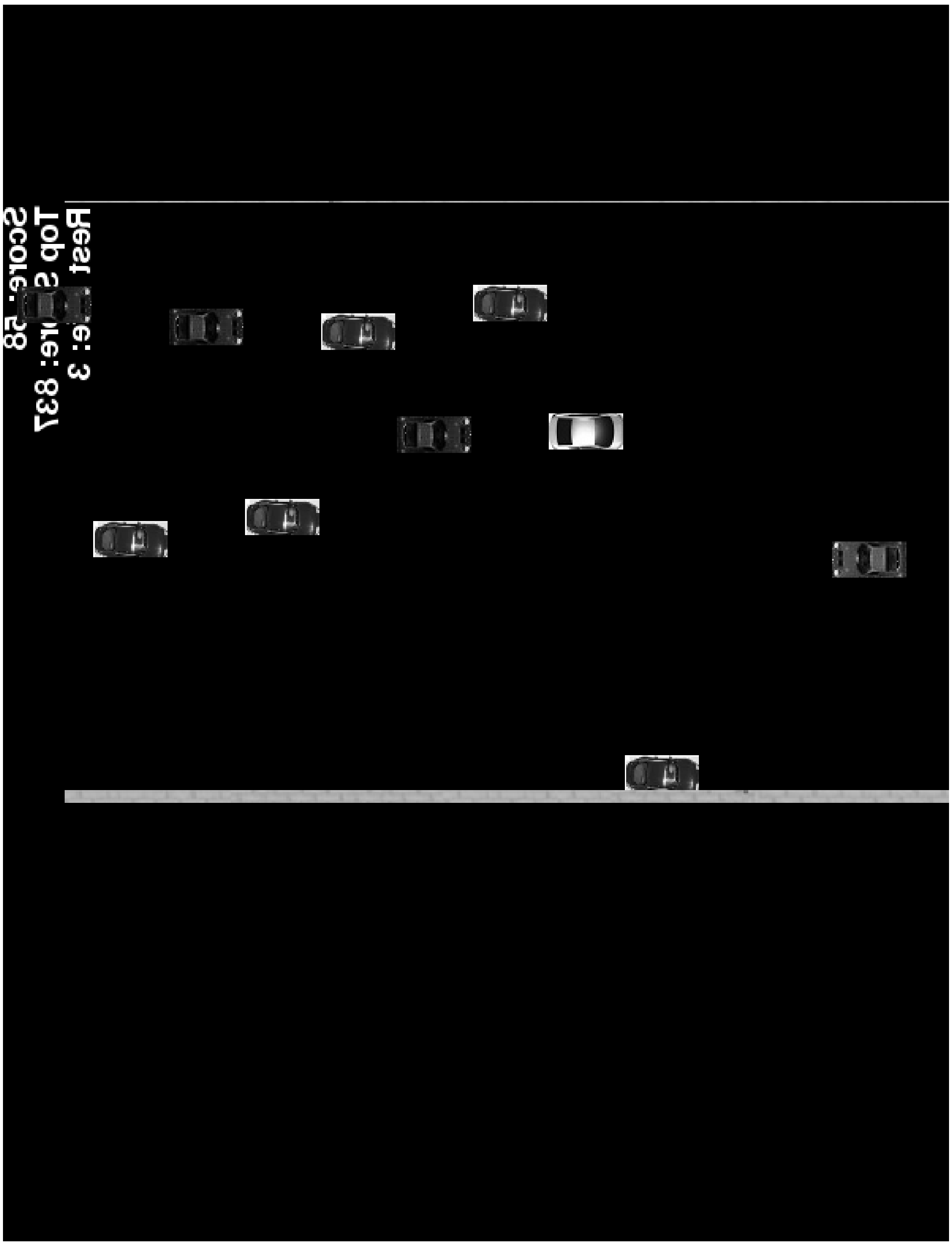
```
x_t_gray = cv2.cvtColor(x_t,cv2.COLOR_BGR2GRAY)
cv2.imshow('Show image before removing borders',x_t_gray)
```

This results in the following image:



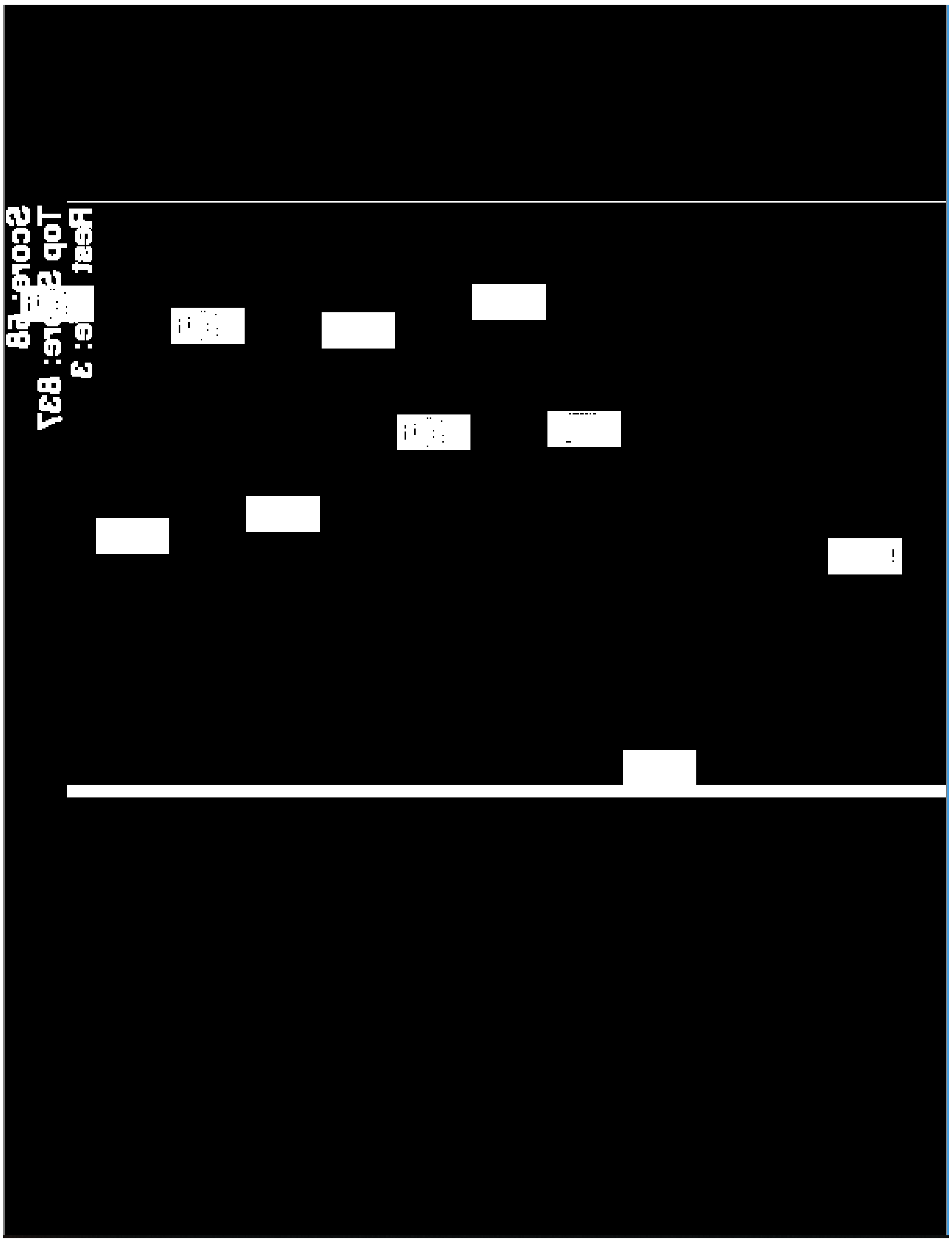
3. Now the borders of the wall and the grass beyond that are removed:

```
# Remove the borders of the game, just keeping one small line on the side
x_t_gray[:125,:]=0
x_t_gray[505:,:]=0
cv2.imshow('Show image after removing borders',x_t_gray)
```



4. Next, a binary threshold is applied to only differ between black and white:

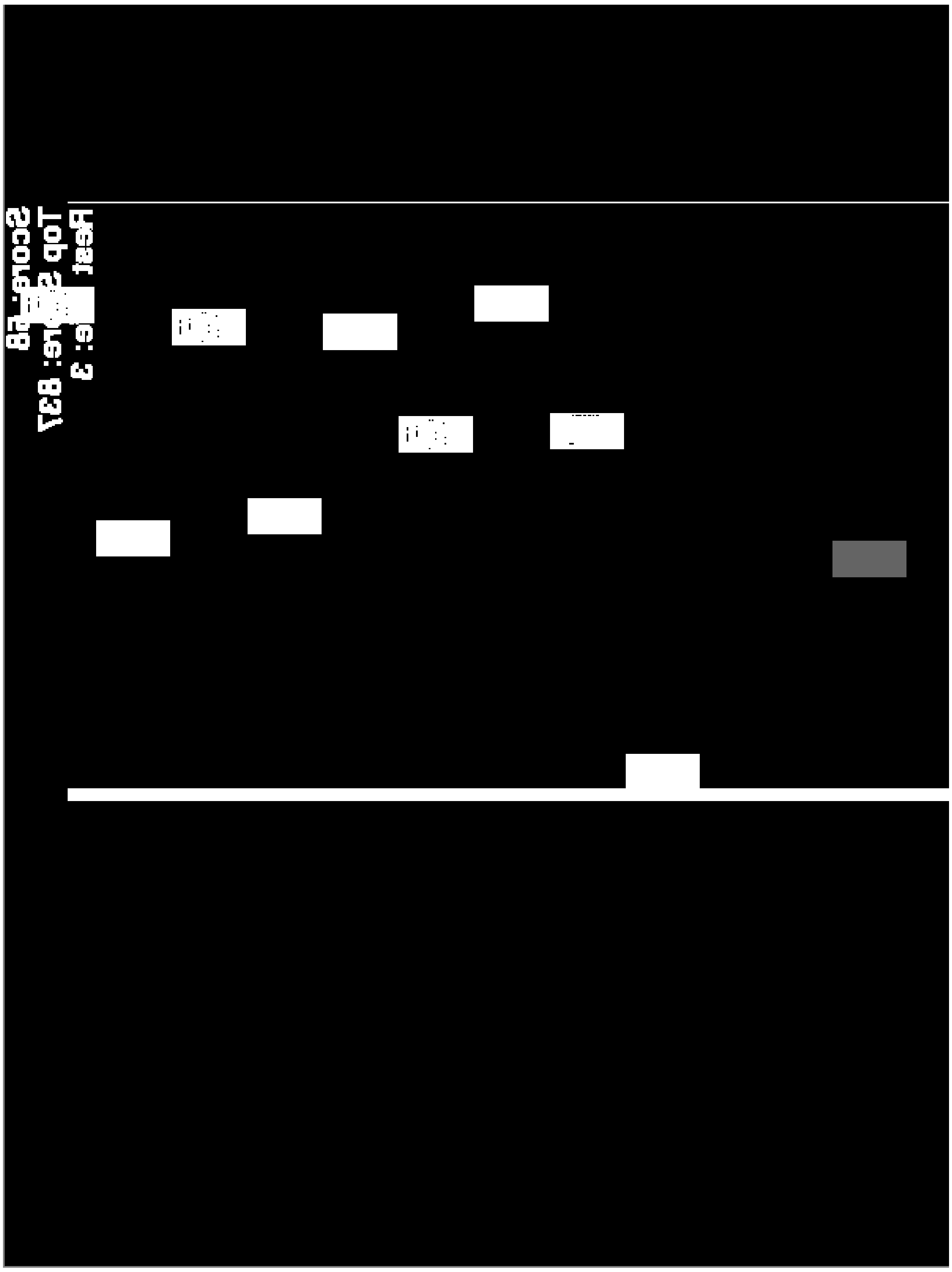
```
# Perform image thresholding converting the image to a binary image
retval,x_t_gray = cv2.threshold(x_t_gray,1,255,cv2.THRESH_BINARY)
#original reshape should rotate the picture but did not work
cv2.imshow('Show image after binary threshold',x_t_gray)
```



5. Now the color of the player itself is changed to gray, to differ between obstacles and

players:

```
#Now change color value of the player itself to differ from enemies
x_t_gray[playerRect.left:playerRect.right,playerRect.top:playerRect.bottom]=100
cv2.imshow('Show image after changing color of player',x_t_gray)
```

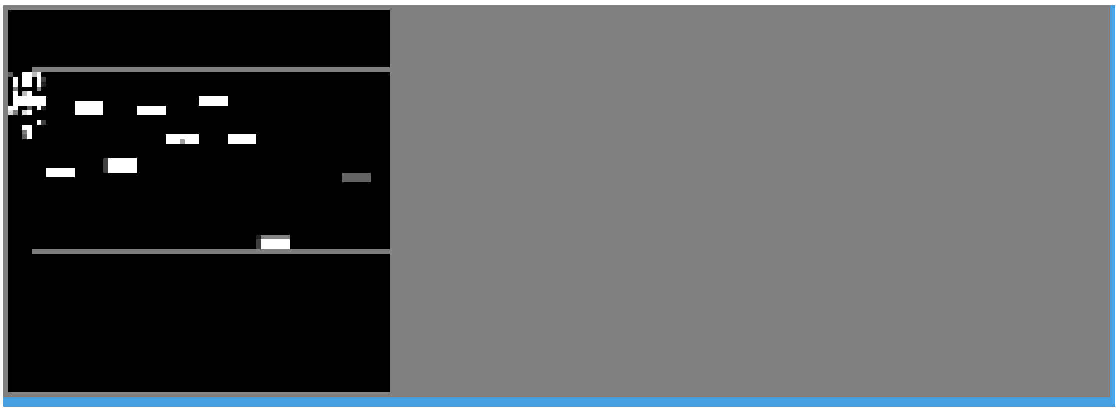

Score: 58
Top Score: 837

6. Make the image cubed so that it can be resized to 80x80:

    ```
    # Make image cubed so that it can be reshaped to 80x80
    a = np.zeros((800,200),dtype=np.uint8)
    x_t_gray=np.hstack((x_t_gray,a))
    cv2.imshow('Show image after changing color of player',x_t_gray)
    ```

7. Last, the image is resized to 80x80 pixels:

    ```
    x_t = x_t_gray[::10,::10]
    cv2.imshow('Test small',x_t)
    ```



By doing these adjustments, the neural network input is not a 80x80x3 variable but only 80x80x1. This accordingly also needs to be changed in all according code parts.

3.2 Reward function

The next change was done with respect to the reward function. In general, we differ between sparse and dense reward functions. Considering that the current reward is constantly 0 and -1 when an obstacle is hit, the reward function is very sparse. This means that it will take a long time to train the network as there is not a lot of information on when the algorithm performs correctly.
Therefore a different reward function was implemented. Still, when a player hits an obstacle, the reward function is set to -1. But also, everytime one enemy is passed, the reward function is 1. This is still a quite sparse approach but should already perform better.

A possible more fancy reward function could use the change of distance to the closest obstacle. If the change of distance is positive it is good, and if the change of distance is negative it is bad. Due to the limited amount of time and the faced issues (discussed in the next chapter) it was not possible to test this reward function.

```
r_t = 0
for b in baddies[:]:
    if b['rect'].top > WINDOWHEIGHT:
        baddies.remove(b)
        r_t = 1 # Reward when one baddie was passed
```

During the implementation it was also thought to change the reward function to the current score and to a big penalty of e.g. -100 when an obstacle is hit. That way, the network constantly receives a feedback with regards to reward. Due to computational lacks, the two approaches could not be compared (see Results section).

4 IMITATION LEARNING

Additionally to the reinforcement learning, the neural network could be initialized using imitation learning. That way, a human is actually playing the game and training the network based on the same reward function. To do this, a few changes had to be done:

- Add actions for the keyboard arrows:

```
# Brought to you by code-projects.org
if event.type == KEYDOWN:
    if event.key == K_ESCAPE:
        model.save_weights('model.h5')
        terminate()
    if event.key == K_UP:
        action_index = 2
    if event.key == K_DOWN:
        action_index = 3
    if event.key == K_LEFT:
        action_index = 0
    if event.key == K_RIGHT:
        action_index = 1
```

- Comment the part where the action index is reset and where the action index is taken from the network output:

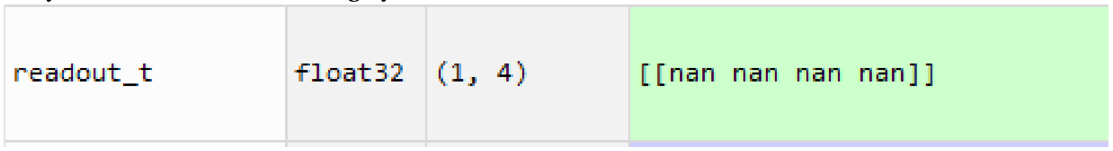
```
#         action_index = 0
#         if random.random() <= epsilon or t <= OBSERVE:
#             action_index = random.randrange(ACTIONS)
#             a_t[action_index] = 1
#         else:
#             readout_t = model.predict(np.reshape(s_t,(1,80,80,4)))
#             action_index = np.argmax(readout_t)
#             a_t[action_index] = 1
```

This imitation learning was executed for approx. 2000 cycles (after observation time). The problem is that the computational power of the used hardware is too small, therefore as soon as the network is being trained, the frames per second of the game drop to around 3 FPS, which makes the imitation learning very exhausting.

5 Faced problems

5.1 Keras framework not working on own machine

During the development of the algorithm it was seen that the model prediction returns a NaN array after the second training cycle.



```
readout_t    float32    (1, 4)    [[nan nan nan nan]]
```

Many hours have been spent to debug into the keras framework and search the internet for solutions. After porting the same software to another laptop, the code was running fine - until the end it could not be found out why the code is not working on the original laptop.

This problem represented the biggest issue and delayed any further process.

5.2 Computational power

As soon as the model prediction and fitting algorithms are executed, the game framerate drops to approx. 3 fps. This is due to the limited computational power on the laptops that could be used. To at least get the framerate up, the minibatch size for the training was reduced from 32 to 4.
It was also tried to execute the code on an Amazon EC2 node, but unfortunately the game needs to have a display connected in order to run.

6 Results

Unfortunately, it was not possible to create a satisfying smart AI agent that drives autonomously. This is due to the computational inability of fitting the model properly and mainly due to the delay of progress because of the training issue on one laptop.
Due to the computational inability, it was also not possible to compare different reward functions, as the training just takes too long.
The highest score that could be achieved with the agent was 161