Part 2

For this part of the project, we are using dlib's face landmark detector and OpenCV's implementation of Harris Corner Detector.

To summarize, our algorithm works as follows:

1. Probe unvisited neighboring squares by going near them.
2. If face changes when we go near a square, delete that square from possible moves list.
3. After probing each square, go to one of the available squares.
4. If we are in a winning square, print "Win" and exit the game and the program.
5. After going into the square, detect corners and check if corners are aligned correctly for our movements. If corners are too much to the left, right, up or down, fix your position.
6. Go to the step 1.

Code and Explanation

0. Warnings

> ❗ Our screen resolution is 1366x768 and program only works in this resolution.

> ❗ We have used Wine to run the game in the Linux environment, timings could be different on different platforms.

> ❗ Our program uses a pretrained face landmark detector to detect worried and calm faces. Therefore, the file we have used in assignment 2, namely "*shape_predictor_68_face_landmarks.dat*" needs to be in the same folder as the "*part2.py*" but we could not upload the pretrained model due to file size limit.

1. Movement

In the beginning of the program, to write cleaner code, we have enumerated `R, L, U, D` symbols.

```
R, L, U, D = range(4) # Enum
```

Then, we have defined a `move()` function to use throughout the program. `move_let` is `R`, `L`,`U` or `D`, and `sleep_time` is the duration of our movement in the given direction.

```
def move(move_let, sleep_time=0.54):
    keys = ["D", "A", "W", "S"]
    pg.keyDown(keys[move_let])
    time.sleep(sleep_time)
    pg.keyUp(keys[move_let])
    time.sleep(sleep_time)
```

2. Worried Face Detection

To determine if the face is worried, we have used the screenshots of calm and worried states. Then, we have checked the absolute difference between the worried face's and the calm face's facial landmarks. In our experiments, we have seen that difference of the landmarks of two worried faces tend to be less than 1000, and difference between the landmarks of worried and calm faces tend to be higher than 4000. For stability, we have decided to claim differences more than 3500 are calm and others are worried.

In the following code, `w_points_x` and `w_points_y` are previously calculated landmarks of the worried face.

```
def is_worried(img):
    img = img[600:, 1200:, :]
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    rectangles = detector(gray)
    points = predictor(gray, rectangles[0])
    points_x = [points.part(k).x for k in range(68)]
    points_y = [points.part(k).y for k in range(68)]
    a = np.sum(np.abs(np.array(w_points_x) - np.array(points_x)))
    b = np.sum(np.abs(np.array(w_points_y) - np.array(points_y)))
    if (a+b) > 3500:
        return False
    return True
```

3. Position Fixing

We have determined the positions of the corners when agent is in a good place (around the center of the square). To do that, we first take a screen shot and extract the area around our agent.

```
time.sleep(3)
no_move = pg.screenshot()
no_move.save("no_move.png")
no_move = np.array(no_move)
```

Then, we use the Harris Corner Detector.

```
img = no_move[90:350, 500:850]
gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
gray = np.float32(gray)
gray = cv.GaussianBlur(gray,(5,5),0)
dst = cv2.cornerHarris(gray,2,3,0.04)
dst = cv2.dilate(dst,None)
ret, dst = cv2.threshold(dst,0.01*dst.max(),255,0)
dst = np.uint8(dst)
```

After determining the corners, to get the indices of the corners, we use numpy's `where` function and then, get the minimum and maximum values of the detected points since these are supposed to be the corners of the square.

This operation shows us that minimum row, maximum row, minimum column and maximum column indices of the corners should be around $[25, 222, 82, 284]$.

Then, we use the following function to fix our position:

```
def fix_pos():
    time.sleep(1)
    ss = pg.screenshot()
    ss = np.array(ss)
    img = ss[90:350, 500:850]
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
    gray = np.float32(gray)
    gray = cv.GaussianBlur(gray,(5,5),0)
    dst = cv2.cornerHarris(gray,2,3,0.04)
    dst = cv2.dilate(dst,None)
    ret, dst = cv2.threshold(dst,0.01*dst.max(),255,0)
    dst = np.uint8(dst)
    row_pos, col_pos = np.where(dst != 0)
    r1, r2, c1, c2 = min(row_pos), max(row_pos), min(col_pos), max(col_pos)

    if np.sum(np.abs(np.array([r1, r2, c1, c2]) - np.array([25, 222, 82, 284]))) > 20:
        if c1 > 82:
            move(R, (c1 - 82)/100)
        else:
            move(L, (82 - c1)/100)
        if r1 > 25:
            move(D, (r1-25)/100)
        else:
            move(U, (25-r1)/100)

    return r1, r2, c1, c2
```

4. Playing the Game

We have implemented a kind of a depth first search. Game is played by the `State` objects.

`next_states` method determines if moves are possible. This method is called in the `probe` method.

`probe` method first fixes the position of the agent, then goes near and comes back to all possible squares around the agent. If the face becomes worried near a square, it is marked as an impossible square in the world.

> ❗ Inside the probe method, there is a variable called first_move. In our system, after the initial move, screen freezes for a while and if there is a worried face, it is possible to miss it in the screenshot. Therefore, we are waiting for 2 seconds after our first move.

`play` method tries all the moves one by one and creates new `State` objects. If the following states do not yield good results, it backtracks.

```
class State():
    def __init__(self, cur_pos, world):
        self.cur_pos = cur_pos
        self.world = deepcopy(world)
        if self.world[cur_pos] == 1:
            print("Win!")
            pg.press("esc")
            sys.exit(0)
        self.world[cur_pos] = -100
        self.move_coords = []

    def next_states(self):
        self.right_pos = (self.cur_pos[0], self.cur_pos[1] + 1)
        self.left_pos = (self.cur_pos[0], self.cur_pos[1] - 1)
        self.up_pos = (self.cur_pos[0] - 1, self.cur_pos[1])
        self.down_pos = (self.cur_pos[0] + 1, self.cur_pos[1])


        candidates = [self.right_pos, self.left_pos, self.up_pos,
self.down_pos]
        self.move_coords = candidates
        dirs = []
        for dr, pos in enumerate(candidates):
            if self.world[pos[0], pos[1]] > -1:
                dirs.append(dr)

        return dirs

    def probe(self):
        global first_move
        fix_pos()
        self.moves = self.next_states()
        ss = np.array(pg.screenshot())

        time.sleep(2)
        if R in self.moves:
            move(R)
            if first_move:
                first_move = False
                time.sleep(2)
            ss = pg.screenshot()
            move(L)
            if is_worried(np.array(ss)):
                self.moves.remove(R)
                self.world[self.right_pos[0], self.right_pos[1]] = -100

        if L in self.moves:
            move(L)
            ss = pg.screenshot()
            move(R)
            if is_worried(np.array(ss)):
                self.moves.remove(L)
                self.world[self.left_pos[0], self.left_pos[1]] = -100

        if U in self.moves:
            move(U)
            ss = pg.screenshot()
```

```
            move(D)
            if is_worried(np.array(ss)):
                self.moves.remove(U)
                self.world[self.up_pos[0], self.up_pos[1]] = -100

        if D in self.moves:
            move(D)
            ss = pg.screenshot()
            move(U)
            if is_worried(np.array(ss)):
                self.moves.remove(D)
                self.world[self.down_pos[0], self.down_pos[1]] = -100

    def play(self):
        if len(self.moves) == 0:
            return -1

        for m in self.moves:
            st = State(self.move_coords[m], self.world)
            move(m, 2.0)
            st.probe()
            p = st.play()

            if p == -1:
                if m == R or m == U:
                    move(m+1, 2.0)
                else:
                    move(m-1, 2.0)

        return -1
```

We create the maze by putting -100 into impossible squares and 1 into goal squares. Rest are 0's.

```
maze = np.zeros((7, 12))
maze[1:4, -2] = 1
maze[:, -1] = -100
maze[4:,4:] = -100
maze[0, :] = -100
maze[:, 0] = -100
maze[-1, :] = -100
```

Finally, we play the game:

```
a = State((5, 2), maze)
time.sleep(3)
a.probe()
a.play()
```

Example Output in the commandline:

```
(deep) ugur@ugur-HP-Notebook:~/Belgeler/ITU Course Resources/20 Fall/Computer Vision/Homeworks/Term Project/Part2$ python part2.py
Starting to move in 3 seconds
2
1
Go!
Win!
(deep) ugur@ugur-HP-Notebook:~/Belgeler/ITU Course Resources/20 Fall/Computer Vision/Homeworks/Term Project/Part2$
```