SIV Project: Extracting Significant Features in Atari Games

Richichi Andrea
Sorrentino Francesco

A.A. 2024-2025

Table of Contents

Introduction

The whole idea behind this project stems from Héctor Moreno Ferrando's work that studied how image processing applied to Atari video games can extract some useful symbolic features. In the original work the algorithms applied were implemented in MATLAB, using the Arcade Learning Environment as a backbone for the acquisition of the images needed.

The goal of this project is to understand the techniques in the study and adapt them for the Python language. This is also done to evaluate the code in a different language, studying if Python can be properly used in this context, giving us also a more complete idea on the usefulness of the features extracted.

The future goal behind the extraction of useful features in games is to create a solid base behind the use of AI for solving Atari games. The importance of image processing remains relevant even in the current world of AI because if a problem can be solved with simpler processing techniques we have a great advantage in terms of complexity and explainability.

Project Domain

Atari video games

The choice of using Atari video games for testing image processing techniques is motivated by various reasons. Firstly the simplicity of these games both in terms of graphics and gameplay makes them ideal for various research purposes. The minimalism of the different game frames also prove to be good datasets for image processing.
Along with the low resolution, this old games also have the advantage of having iconic objects and clearly defined trajectories on the screen (i.e. Pac-Man, Donkey Kong…) that bring out manageable challenges and opportunities for symbolic abstraction.

While the study of reference tested the processing pipeline on various titles, we chose two games:

1. Breakout: A simpler game environment useful for testing the various steps of the processing phase without too many unexpected scenarios.
2. Skiing: A more complex game that has more moving objects than Breakout, chosen as a test for the processing algorithm.

Atari Learning Environment

The Atari Learning Environment (ALE) is a widely recognized framework developed to assist AI research. The modules are built on top of the open-source Stella Atari 2600 emulator that emulates behaviour of classic games with good accuracy. ALE allows access to the peripheral controls of the games in order to easily code an agent that plays them autonomously. Within the module are already included more than 900 games to be studied with multiple or single-agent environments.

ALE offers two sources of data: game's RAM state and pixel-based frames that represent the screen of the game at each time step. This particular project will only use the frames to study them further. The

project will use the Python interface of the ALE module to collect frames as the episode progresses further.

Acquisition

The project files include a directory appropriately named */acquisition* that contains the code for acquiring the frames of the two chosen games.

Firstly a random agent has been set to play the two different games following the pseudocode:

1. *actions ← ale.getLegalActionSet()*
2. ***while*** *not ale.game_over()*
    a. *random_action ← actions[rand_range(len(actions)]*
    b. *ale.act(random_action)*

This has the clear advantage of completely automating the acquisition process, making it very fast with the drawback of not having very interesting gameplay frames with low scores at the end of the episodes.
Since we could also wish to study the games' behaviour with a more competent player, a Python script called *human_acquisition.py* has been added, giving the possibility of playing the games via keyboard. This acquisition method is slower but gives out much more informative frames and helps to conclude the games with a higher score.

We must also keep in mind that the duration of the acquisition also depends on the game's constraints. Breakout ends when the player loses all lives and so will end first with the random agent because the ball will fall under the screen very easily. On the other hand, skiing lasts as long as the player hasn't passed through all flags, meaning that a human agent will end the game earlier while the random agent will take longer.

Attacked Problem

The project tackles the problem of extracting properly structured data from frames. This fundamental problem exists because of the gap between raw pixel data and meaningful information that can be used for AI development.
An argument could be made against employing image processing, since of the widespread employment of deep learning techniques that do not require this kind of processing on data.
However these solutions are very expensive and have difficult interpretation when dealing with game states.
We must also keep in mind that agents that will use the symbolic information obtained from processing could still provide a satisfying performance even if very simple.

Let us now enter in detail about what is the information that will need to be extracted from the pixel-based frames.

Scale-Invariant Feature Transform

The first type of data to be extracted are so-called Scale-Invariant Feature Transform (SIFT) keypoints and their descriptors.

SIFT is an image processing technique that identifies and describes local features in a given image offering scale and rotation invariance. This technique proves very useful in object detection and pattern recognition.
This method also proves well-suited for Atari environments where consistent features could help to provide even more information for an agent playing the game.
After finding keypoints and descriptors for a single frame we can try to match them between successive frames in order to find out static or dynamic locations in the different images.

Connected components information

When identifying objects in the image, some useful information could be all data regarding connected components identified in a given frame.
For every connected component found in the image we should know:

- x,y coordinates in the pixel-based frame
- width and height
- area
- centroids.

Pixel lists

The previous information however is lacking because we don't know with precision the location of the frames of a given object in a frame. Pixel lists encode this information providing a collection of the pixels' coordinates x,y for every pixel in a given object in the frame.

Partial distances

Dealing with mobile objects requires also keeping track of their pixel distance from the starting position. This allows us to plot a graph of the evolution of the distance of the mobile objects through the frames.
Also if objects share patterns in the relationship between distance and time we can assume they have some sort of interaction inside the game.

Methodologies

After introducing both the problem and the environment, a procedure must be defined in order to extract the required information from the frames.
The steps that must be performed are:

1. Load the frames
2. Find the SIFT keypoints
3. Feature matching
4. Detect connected components
5. Classify objects as mobile or static
6. Track moving objects.

Load frames

To start the processing a procedure must be created to load in the frames in both BGR and gray. This is performed by the function:
load_frames(game, step=100) return imgs, gray_imgs.
The *game* parameter specifies which game to load with a string and *step* is the number of frames to skip when loading the frames for processing.

Since the number of initial frames could be very high and also could present very little differences between each other, the function skips 100 frames before loading the next one, reducing the images that will be actually processed. Let us keep in mind that the number of frames depends also on the length of the acquisition episode and so generally longer games will have more frames recorded.

The processing will use both images in BGR and gray levels, therefore the procedure returns both to be used later on.

Find SIFT keypoints

The processing properly starts by detecting features with the SIFT algorithm. SIFT actually works in four steps, let's see them.

1. Space-scale Extrema Detection: In order to detect properly the keypoints scale-space filtering is used. Laplacian of Gaussian (LoG) is found for a given image with different parameters σ. LoG acts as a blob detector with σ as a scaling parameter. The output of this step is a list of values (x, y, σ) that are potential keypoints. However since computation of LoG is costly, Difference of Gaussians (DoG) is used as an approximation that is simply the difference of Gaussian blurring of an image with two different σ. Then once DoGs are found we search for keypoints over scale and space.
2. Keypoint Localization: Once possible keypoints are found a refining step must be added in order to find only relevant ones. A Taylor series expansion of scale space is used to get a more accurate location of extrema, then if intensity is less than a threshold we reject that keypoint. DoG has a higher response to edges and so they must be removed. The output of this step is the keypoints of step 1 without low-contrast and edge ones.
3. Orientation Assignment: An orientation is then assigned to each keypoint to achieve rotation invariance. To determine the keypoint's direction, we look at a small region around it, sized according to its scale. We then measure how the image brightness changes (gradients) in that area. These changes are grouped into a circular histogram with 36 sections, each covering 10° of rotation. The values in the histogram are weighted to give more importance to strong changes and nearby pixels. The highest peak in the histogram sets the keypoint's main direction, but if other peaks are at least 80% as strong, they're also considered. This means a single keypoint can have multiple orientations, which helps make feature matching more reliable, even if the image is rotated.
4. Keypoint Descriptor: To describe the keypoint in detail, we focus on a 16×16 patch around it. This area is split into 16 smaller squares, each 4×4 in size. Within each square, we analyze how the brightness changes in different directions and store these changes in an 8-bin histogram. In the end, we gather 128 values, which together form a unique fingerprint for the keypoint. To make sure this fingerprint stays reliable even if the image gets brighter, darker,

or rotated, extra steps are taken to adjust for these changes, making feature matching more accurate and stable.

This procedure is the same implemented in the OpenCV library that is used in the project to assist with most steps of processing.

Feature matching

Having found keypoints for every frame of the dataset, we can now find matches between consecutive images given a step of frames to be skipped.
Matches can be found by employing searching for the nearest neighbour method for every keypoint. Lowe's ratio test is then applied in order to filter out bad matches.

For each keypoint in the first image, we find the best matching keypoints in the second image. We keep the two closest matches based on how similar they are. To ensure the match is reliable, we use Lowe's test, which compares the best match with the second-best. If the two matches are too similar, the keypoint is discarded because it might not be a strong or unique match. This helps remove ambiguous points and improves accuracy in feature matching.
Then if a minimum number of matches is found homography is also applied. Homography is a technique to morph the first image according to the second one in order to have a more accurate matching result. In the test case of the project there aren't any significant perspective changes, however this could prove important if the algorithm is applied to more complex games.

Connected component detection

With the previous steps we have proven if two consecutive frames might have any matching patterns or objects. From this step onwards the reasoning will be performed on objects present in the frames. In order to find these objects we have to detect connected components in every frame.
In order to enhance detection some thresholding is applied to the gray_imgs in order to have objects with white pixels and black background.

Then we threshold again the gray images in order to have different classes of pixels to be labeled. After obtaining this thresholded images we simply apply the function *connectedComponentsWithStats* in the OpenCV library that gives us all components with their relative information.

However this function does not return pixel lists and so a procedure must be added to obtain them. This can be done simply by saving the coordinates of the pixels of the different labels in an image for each object in each image.

A filtering step is also added in order to ignore huge components that are part of the background by saving only objects with an area lower than a certain threshold.

Object classification

Once the objects have been found we have to define an algorithm to classify them as static or mobile. This can be done by taking n-1 frames where n is the number of frames in the dataset and for every frame i we study the next i+1 frame according to the following pseudocode:

1. *class_list ← [] # Let us define 0 as static and 1 as mobile*

```
    for i in range(len(stats) - 1): # Iterate on every frame
        frame_class ← []
        for j in range(len(stats[i])): # Iterate on every object
            object_class ← 1 # Set class mobile
            for n in range(len(stats[i + 1])): # Iterate on every object of successive frames
                inter ← find_intersection(pixels_lists[i][j], pixels_lists[i + 1][n])
                area ← stats[i][j, cv.CC_STAT_AREA]
                ratio ← len(inter)/area
                if ratio >= thresh: # Check ratio must be higher than a threshold value
                    object_class ← 0 # Set class still
            frame_class.append(object_class)
        class_list.append(frame_class)
    return class_list
```

The algorithm works by taking shared pixels between objects in the two frames and if there is a match the object is static, otherwise It is mobile. The match is checked by seeing if the ratio of intersection divided by area of the object is bigger than a certain threshold value.

Object tracking

The final step of the processing is to track the objects through the frames. This can be done by taking only the objects classified as mobile in every frame and searching for a match in the successive frames.

```
tracked ← [] # Initalize a bool array to save tracked mobile objects in frames
for i in range(len(mobile_obj)): # Set the tracked list correctly
    tracked.append(np.full(len(mobile_obj[i]), False))
apparitions = [] # To save appearances of objects
for i in range(len(mobile_obj)): # Iterate through frames - 1
    for j in range(len(mobile_obj[i])): # Iterate through mobile objects
        apparition ← []
        if not tracked[i][j]: # If the current object is not already tracked
            apparition.append((i, mobile_obj[i][j]))
            area ← stats[i][mobile_obj[i][j], cv.CC_STAT_AREA] # Init area of object j in frame i
            for k in range(i + 1, len(mobile_obj)): # Go to next frame to search for same obj
                for n in range(len(mobile_obj[k])): # Iterate through mobile objects of next frame
                    if not tracked[k][n]:
                        area_next = stats[k][mobile_obj[k][n], cv.CC_STAT_AREA]
                        # Here we must find an object similiar to our starting one
                        if area == area_next - thresh:
                            apparition.append((k, mobile_obj[k][n]))
                            tracked[k][n] = True
                            break # Added to stop iterating in a frame after first similiar object found
            tracked[i][j] = True
        if len(apparition) > 0: # Let us save the appearances if we have found them
            apparitions.append(apparition)
```

The algorithm works by keeping track if a mobile object has been already tracked and also saving the list of locations for a given object.

When a new untracked object is found we search for a similar mobile object in every next frame and if we find any we set them as tracked. This is iterated for every n-1 frame giving us a list of appearances for every different object found.

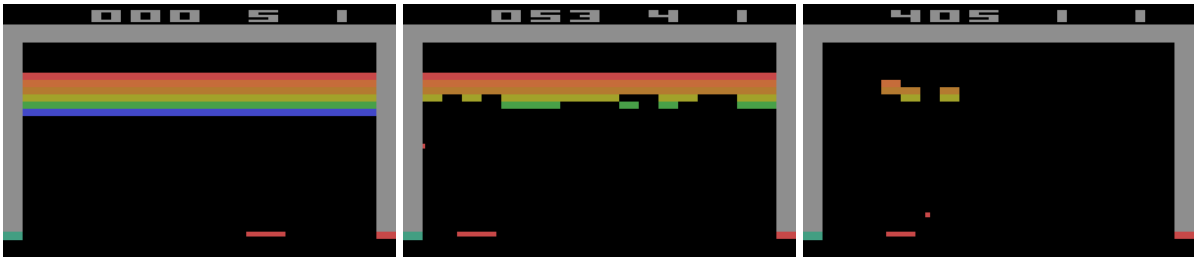
It is trivial to compute partial distances given a list of positions and then we just have to plot the changes in distances over frames for every different object found.
Since some objects could have been found due to noise or could be uninformative to the game state we filter out the partial distances of objects with less than a minimum number of apparitions.

Results

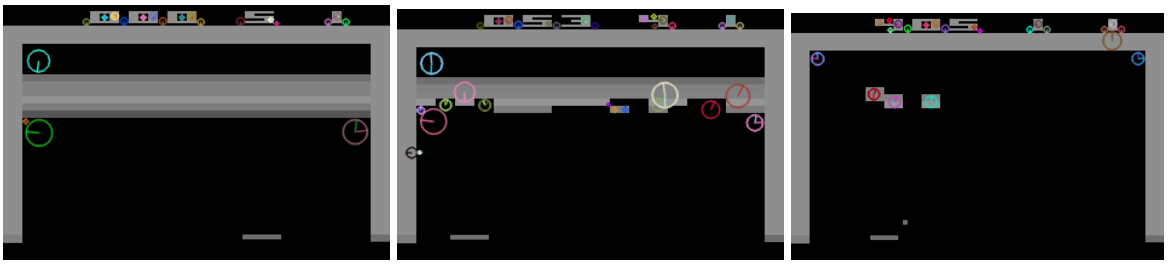
The results of the project along with the implementation of the algorithm are contained in the file */processing/processing.ipynb*.
The results presented are obtained from processing a dataset generated by executing the file *human_acquisition.py* with a collected total of 9099 frames for Breakout and 2293 frames for Skiing.
For every step previously presented let's see 3 images: first, middle and last image of the result of the processing in that specific step.

Breakout Analysis

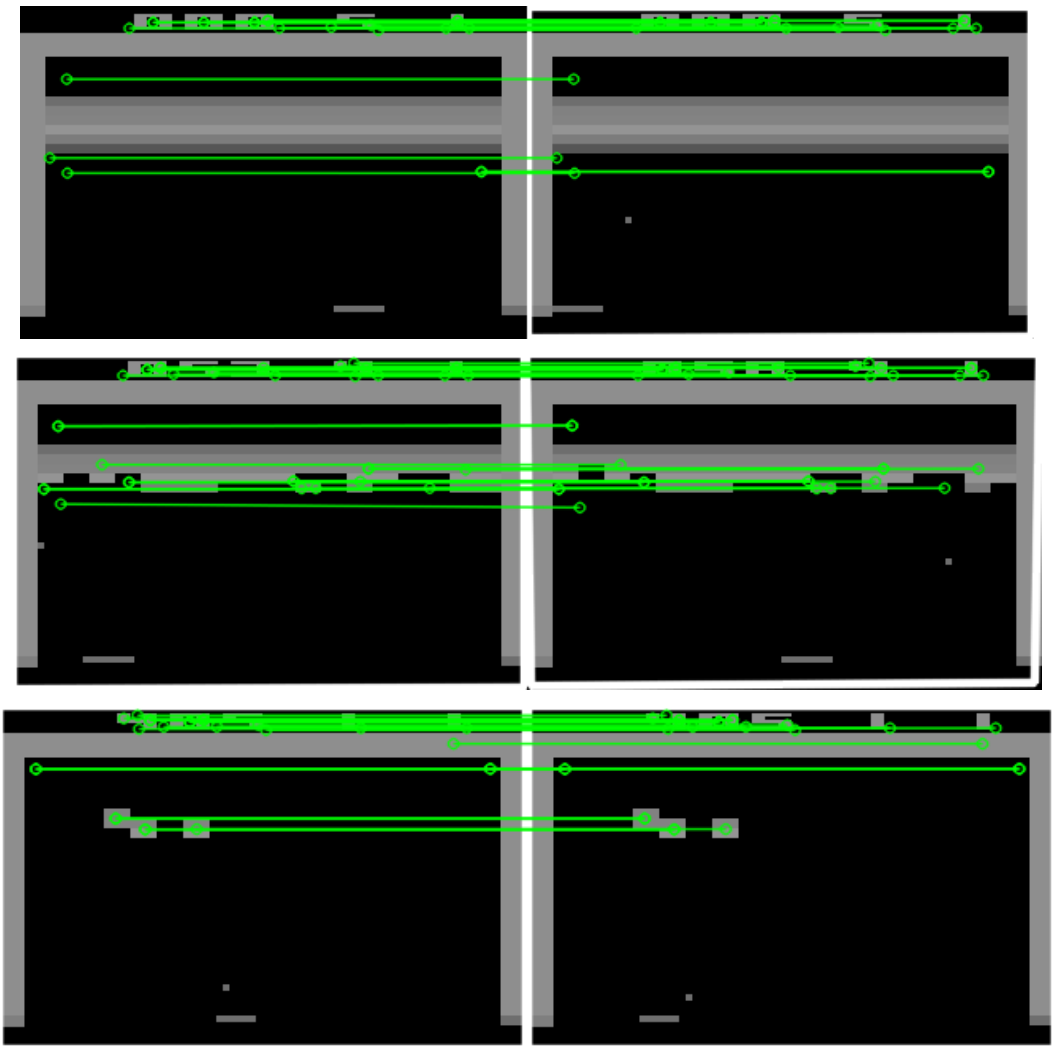
The first step is to load in the images from the dataset.



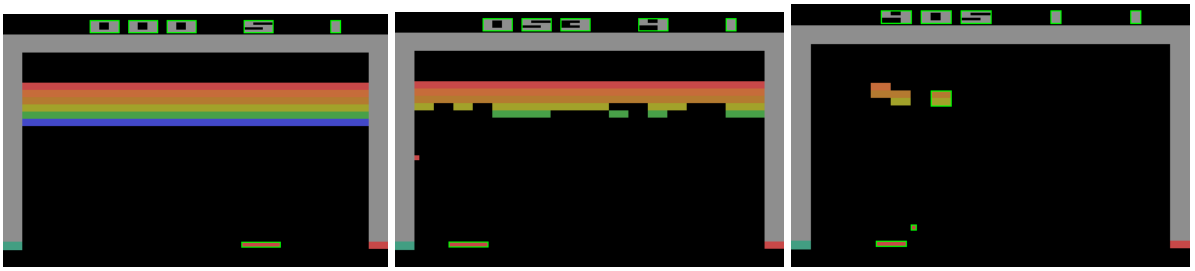
The first step then takes these images in grayscale and computes relevant keypoints with the technique explained before. The features are mainly detected around the score, the lives and sometimes also the bricks that have to be destroyed.



Given keypoints and their descriptors, we can compute matches between frames. The matches are successful given that this environment does not have big changes from one frame to the next and the perspective does not change massively.

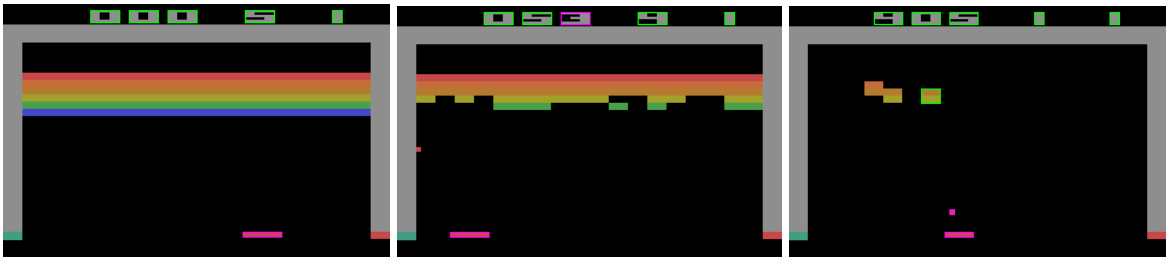


The processing, aftering having matched successfully the features, focuses on detecting the objects. We can see the detected objects surrounded by their green bounding boxes. The algorithm successfully detects the ball, the bar and also the score and lives values at the top.
In some scenarios if the ball or the bar is in contact with the right or left wall the detection fails. However we can ignore those detections since we already have enough detections to approximately track the object.

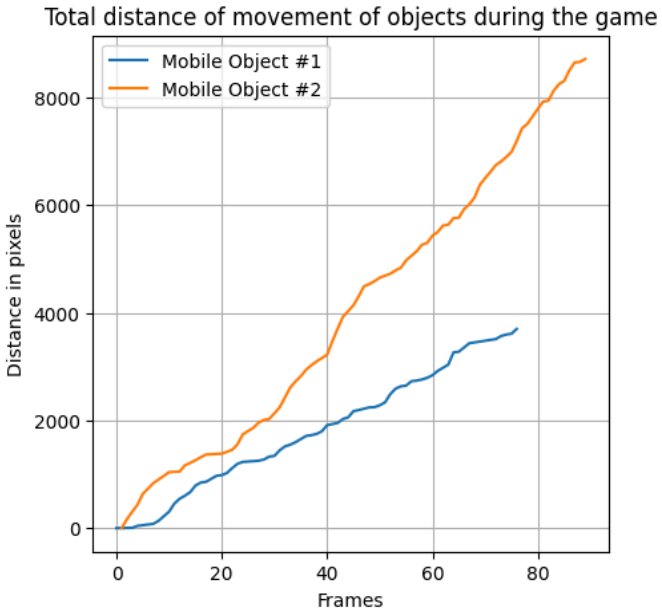


The classification step then sets an object as mobile or static. We can see static objects with a pink bounding box and static with a green one.

Notice that changes in the score in the next frame also classify the figure that will change as mobile even if they don't move because their area changes. This will be filtered out in the tracking step by filtering out objects that appear only a few times in the frames.



The last step then tracks the objects and plots the partial distances over the number of frames. After setting the fact that an object to be plotted must have appeared at least 30 times on 90 frames, we draw It.



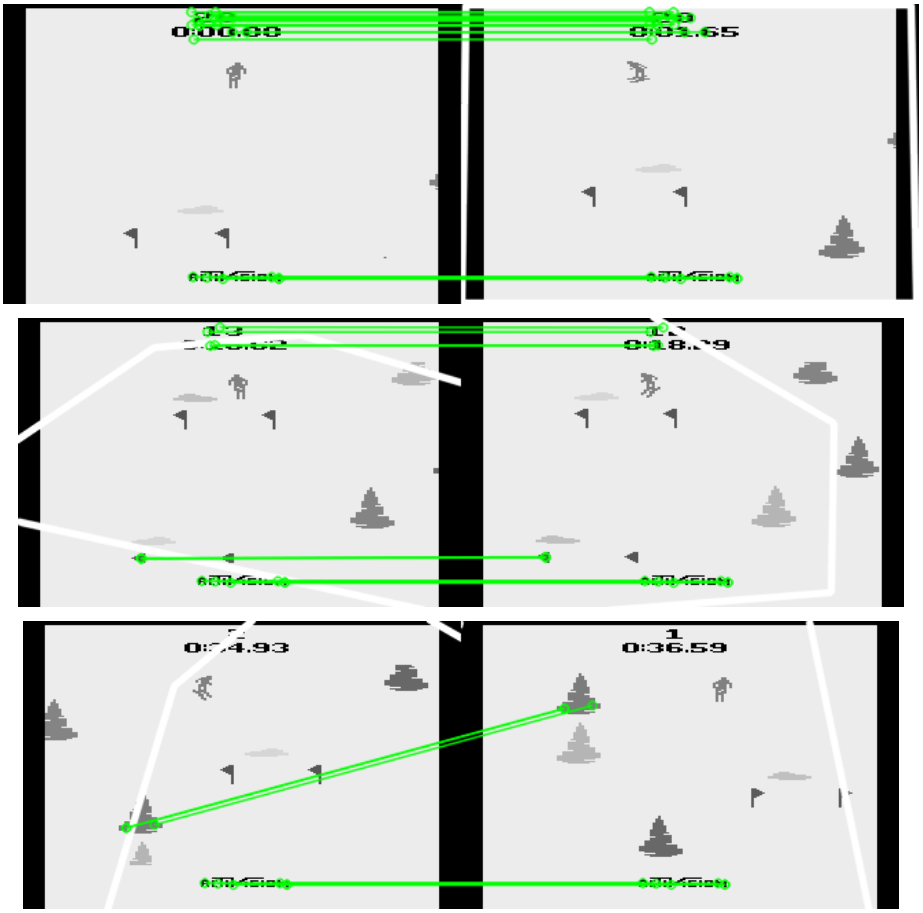
The algorithm obviously can't name the objects so we have to manually check what are "Mobile Object #1" and "2#" in order to properly understand the graph. After checking manually we find out that object #2 is the ball that moves intuitively a lot more than the bar and #1 is the bar itself. The bar also stops being detected at a certain point because as the game goes on it gets noticeably smaller.

Skiing Analysis

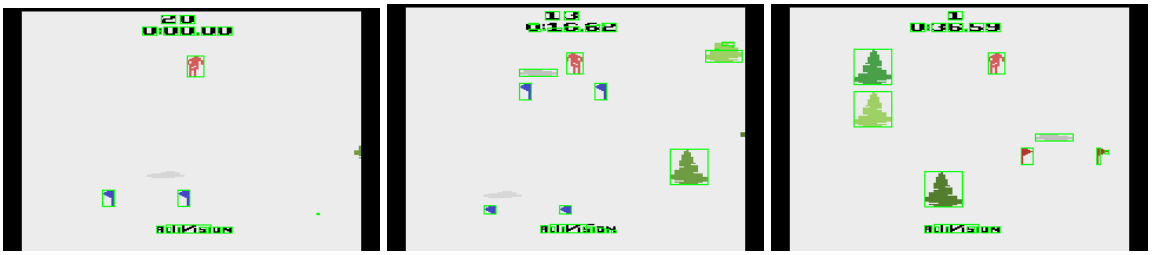
Firstly the frames are loaded for processing.

Keypoints are detected and computed, proving to be much more informative than with the Breakout since they also cover the player, flags and the trees.

Feature matching is performed and it's mostly successful with some keypoints that have too weak matches. However trees and flags can be successfully matched giving some information on the location of obstacles and goals.



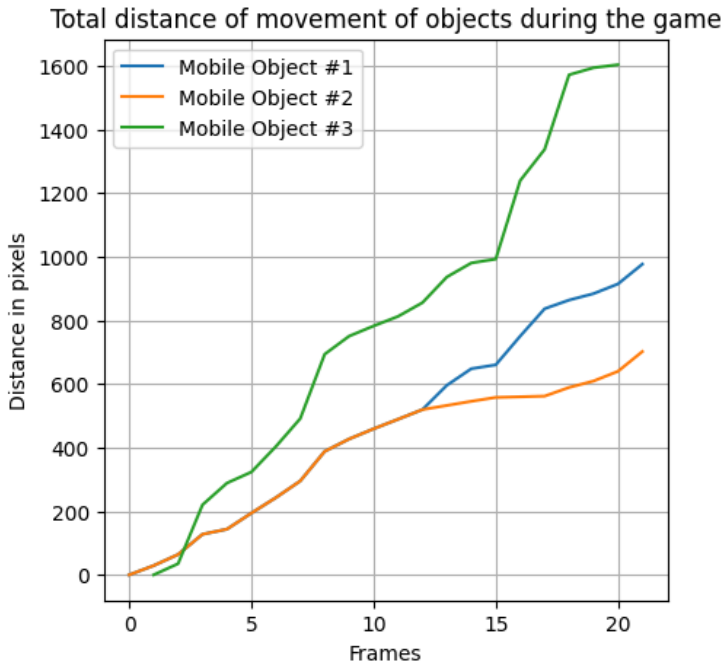
Object detection is next and is successful in finding player position along with flags and trees.



Classification of objects is then performed and we can see that the timer and flag count can be classified as mobile if we have a change in the next frame. Also all objects that move upwards are classified as mobile.

It can be noticed that there are some wrongly detected objects in the corners of the images, however they will be filtered out in the tracking step.

Finally the las step tacks the objects and draws the plot of partial distances over frames.



The threshold of appearances here is set at a minimum of 16 over 22 in order to be more consistent because of the high number of objects on screen and some detections. The objects in the plot are respectively:

- #1 → Right flag
- #2 → Left flag
- #3 → Player

As It can be expected, the player covers more distance than the flags that just move vertically and also have the same distance covered for most of the frames. Around frame 11 there are multiple flags on screen and this can change the distances between right and left flag depending on the first flag found.

Conclusion

The whole project reached the goal of implementing in Python the idea proposed in the reference thesis of Moreno et al. and testing with two chosen games.

The results prove the importance of image processing in the age of AI since important features can be extracted from images to create more sustainable solutions.

Future research could study the processing with more games and try to find better solutions to extract features integrating them or substituting some of the steps proposed in the project.

References

1. Moreno Ferrando, Héctor. "Image processing techniques to extract symbolic features from Atari video games." (2017).
2. M. G. Bellemare, Y. Naddaf, J. Veness and M. Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents, Journal of Artificial Intelligence Research, Volume 47, pages 253-279, 2013.