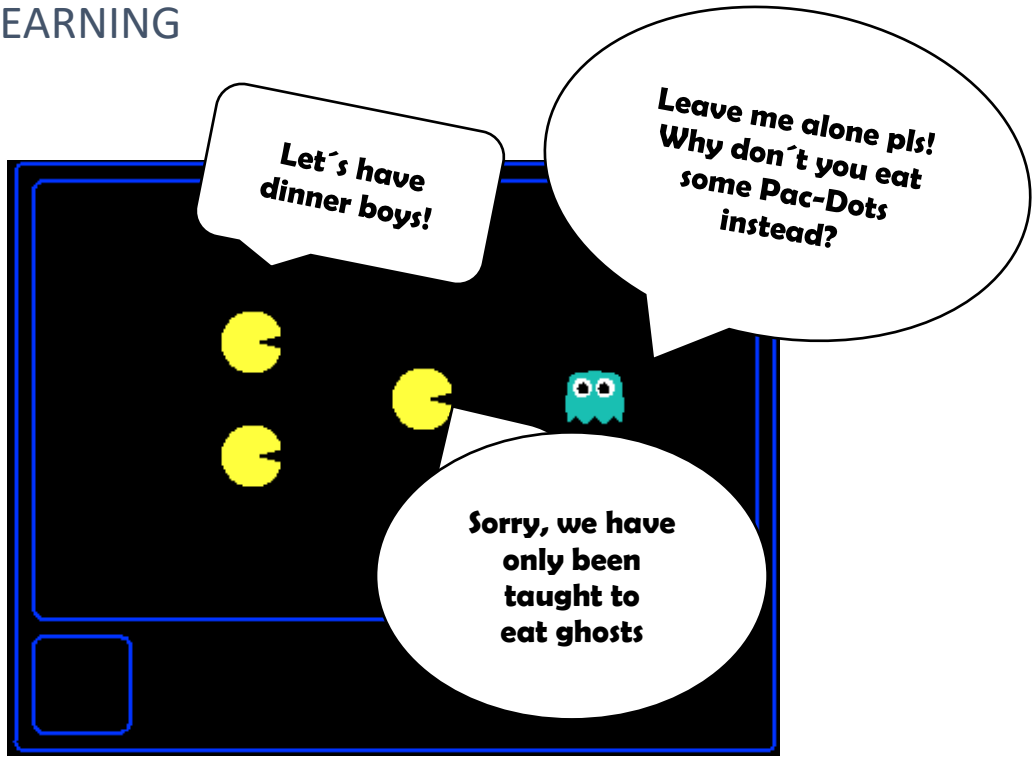
Universidad Carlos III de Madrid

Assignment 2

MACHINE LEARNING



Claudio Sotillos (100409401) | Daniel de las Cuevas (100406666)
2º DATA SCIENCE AND ENGINEERING | UNIVERSIDAD CARLOS III MADRID

Introduction

The objective of this project is to build a Pac-Man agent that will maximize the score obtained by applying the Q-learning algorithm. For that, we will follow a process of intuition and "trial and error", until we obtain our optimal agent. We want to clarify that we have focused on eating the ghosts as soon as possible, Pac-Dots are not a target for the Pac-Man.
Even though we will learn as we get more familiar with the mechanisms, we decided to organize all our ideas and experiments in the corresponding categories.

First, we will explain the main functions that we have programmed (for the selection of attributes, the different reward functions and extra methods) and also explain the modifications that we have done in methods inherited from other Tutorials. We will also talk about the general training process that we have followed for the training of the agents (parameter selection and training strategies at the different maps).

Secondly, we will explain the different agents, focusing in the combination of attributes, their corresponding reward function and our impressions after training each agent. We will keep in mind the resulting dimensions of the Q-table depending on the attribute selection and also explain the mappings we have done for assigning to each state a row of the q-table.

Finally, we will perform an evaluation of our agents with different maps and configurations. We will show the results graphically using the final score and the ticks which have been required. We will also explain qualitatively the results. After comparing all the agents, we will choose our best candidate.

Auxiliary methods

Before getting into action, we wanted to explain all the different functions that we used in different parts of the process. In the code you will appreciate that we have divided them in two sections. The functions imported from other Tutorials which have been slightly modified and functions that we have coded from scratch.

Modified functions

Function that initializes the q-table and other auxiliary functions

On top of all the basic functions inherited from Tutorial 4, we needed to define two special ones for this Pac-Man situation. The "*registerInitialState*" is a method that is used in the bustersAgent class to initialize some basic parameters of the game. Thus, in our QLearningAgent class, we defined an adapted version of it that first calls that original method and then sets the parameters for Q-Learning. These are epsilon, alpha, gamma, the list of actions (which maps the 4 possible coordinates to numbers between 0 and 3), and the q-table.
Initially in the list of actions there was an extra action which was the stop action, but we decided to get rid of it since we want our Pac-Man to be in constant motion.

For the q-table, if the *initialize* parameter that we defined is set to True, it will reset the table and set all the values to 0. To do that, we defined another function called "*initQTable*", which made the grid with dimensions according to our selection of attributes. The number of rows is selected manually, which will correspond to the attribute combination (the different states). There will be always four columns corresponding to the four main movements Pac-Man carries out. The dimensions of the q-tables that we have used are of 9x4, 18x4 and 144x4.
Also, the q-table is initialized if the dimensions in the "*initQTable*" are modified.

Another modification we did was in "*computePosition*". Since this function computes the row of the q-table for a given state, it depends on the number of states and attributes. Therefore, we needed to adjust the mapping for every combination of attributes. We will talk about the mappings when we explain each of the approaches.

Function that updates the q-table

As an important part of the Q-Learning algorithm, we need to update the q-table at every tick of the game. Thus, the "*update*" method was thought of in a similar way than in Tutorial 4. First, it finds the position of a given state in the q-table, using the "*computePosition*" mapping described before; and the action in numeric value.

This was done using a dictionary that matched “North”, “East”, “South” and “West” to values 0, 1, 2 and 3; respectively.
Then, we update the q-table with the indexes of the state and action using the Q-Learning formulas.
The update function is used only in the *game* file. It is used for all the ticks except for tick 0 (as we don´t have information about the next tick).

Coded functions

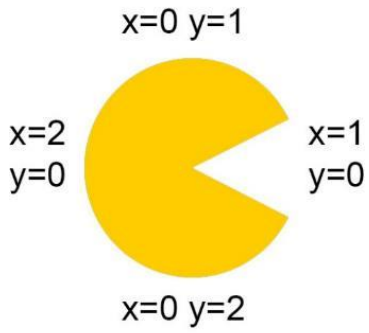
Mytuple: Function that stores the learning tuples

For the learning tuples, we defined a function to store this information, which we called “*mytuple*”. It generates the learning tuples *(state tick, action, state next tick, reward)* that will be used in the update function.
In terms of the functioning, we implemented a similar structure than in the previous assignment. This method is called from the game file, where it updates the tuple in every game run. In this while, for the tick 0 there is the particularity that the function is called, but nothing is stored in the tuple, since the information of the next tick hasn´t been generated yet (there are None’s). For the rest of the ticks, it calls the method, generates the tuple from which the 4 parameters will be extracted and used on the update function

Closestg: Function that computes the relative position of the closest ghost

This function, “*closestg*”, will be used to compute the relative position of the nearest ghost with respect to the Pac-Man. The output of the function will be two attributes (relative position according the X and Y axis) of three levels (0,1,2).
First, we obtain the distances of the ghosts with respect the Pac-Man. Then, we want to take the index of the smallest distance in the list, but before doing this we must substitute the “None” values (a distance is None when a ghost dies) by a huge number. By doing this we won’t have problem finding the smallest distance. Once we have the index of the smallest distance, we can obtain the coordinates of the nearest ghost.
Finally, we subtract the coordinates of the ghost and the Pac-Man, and with the difference that we obtain we can classify and obtain the relative position.

In the X axis, if the difference is smaller than zero, we will classify as 2 (West), if it is bigger as 1 (East) and we classify as 0 if the Pac-Man and the Ghosts are aligned according to the X axis.
In the Y axis, if the difference is smaller than zero, we will classify as 2 (South), if it is bigger as 1 (North) and we classify as 0 if the Pac-Man and the Ghosts are aligned according to the Y axis.



Closestg_wall: **Computes relative position of the Ghost and if there is wall**
This method computes the relative position of the closest ghost exactly in the same way as the before method. The only difference is that when classifying the relative position of the ghost we also check if in that direction there is a wall (just on the next block where Pac-Man may move). If there is a wall in any of the relative directions the attribute "W" will be 1, and if there is no wall then it will be 0.

Functions that return the reward

We had to design reward functions that maximized the score of a game. This was essential for the agent to know which actions have a positive impact in the finishing of the game and which are useless. All the reward functions that we have coded take as input the state and the next state. In the next sections, we will explain each strategy for their design and also train and test the agent using them.

Training process

Now, we will explain the process we followed to train our agents. This process was used in general for all our approaches, with little modifications.
Along the learning process we have done check points of the q-table. For example when the agent learned to play on the first three maps we saved the q-table just

in case we overfitted it by training to much on the next maps (it happened to us that after training on the 4th and 5th maps the agent didn´t work on the before maps).

Parameters

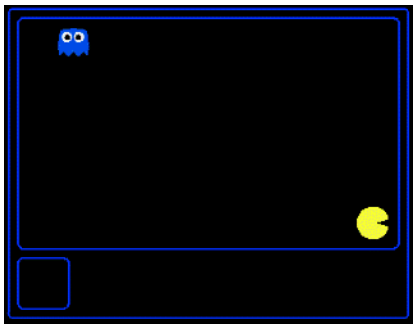
- *Epsilon:* Since it administrates the percentage of random movements the Pac-Man does, we vary it´s value from 1 to 0 throughout the episodes.
- *Alpha:* (Learning rate) We set this to a small value (0.2) since in general we want to give more importance to immediate actions (we will reward for approaching the ghost). For checking if the Pac-Man works effectively in the maps we set its value to 0.
- *Gamma:* This is the discount parameter which we set to 0.8 as it was set in the Tutorial 4.

We want to clarify that we did not spent much time making combinations of this parameters since we thought that the key for the well-functioning of the Pac-Man had more to do with finding a good combination of attributes.

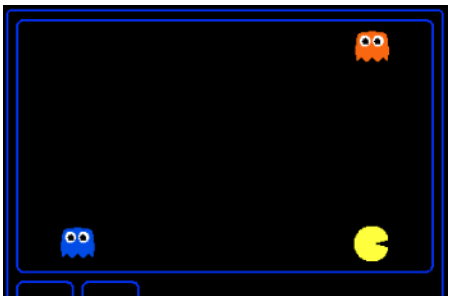
Training process

- **Maps labAA1 and labAA2 (Difficulty: Low)**

The training in the first two maps was very similar since it is the same map but with one extra ghost in the labAA2.
We started setting the value of epsilon to 1 and using Random ghosts. Once the q-table had some values so that the Pac-Man could make a decision form it we reduced the epsilon gradually, thus little by little it would take more decision based on the q-table than randomly.



labAA 1

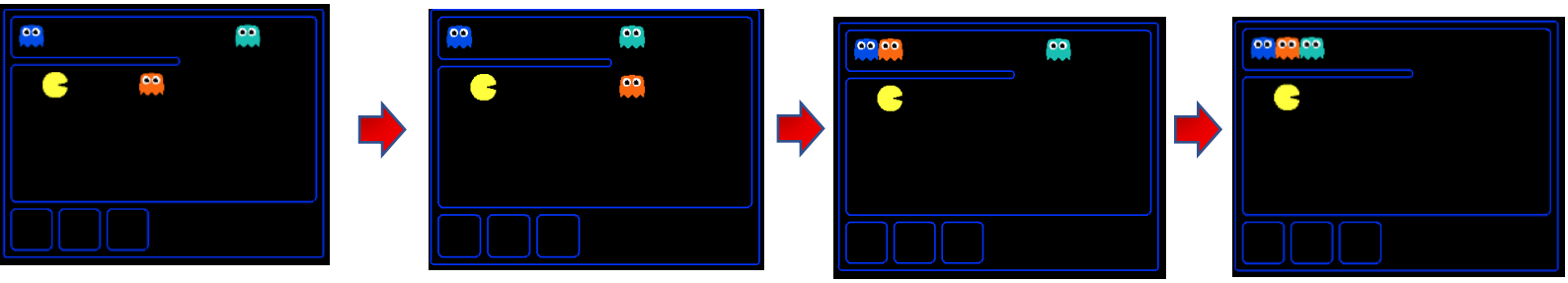


labAA 2

After some episodes with random ghosts (so it had learnt a bit how to move) we started to chain episodes using Random and Static Ghosts. Once the Pac-Man performed perfectly on both maps (setting epsilon and alpha to zero) we went on to train in the next map (labAA3).

- **Map labAA3 (Difficulty: Medium)**

The third map was a little bit trickier because of the wall in the left and 3 ghosts instead of 2. As our Pac-Man already knew to eat ghosts in a map without walls (that is, it had a useful q-table) we started using an epsilon of 0.75 and decreasing it gradually. As in the before maps we started training with random ghosts and making different combinations. After some episodes we started using static ghosts and also placing ghosts in a way that would benefit the learning for going around the wall. The photos below show the progression of the learning strategy.



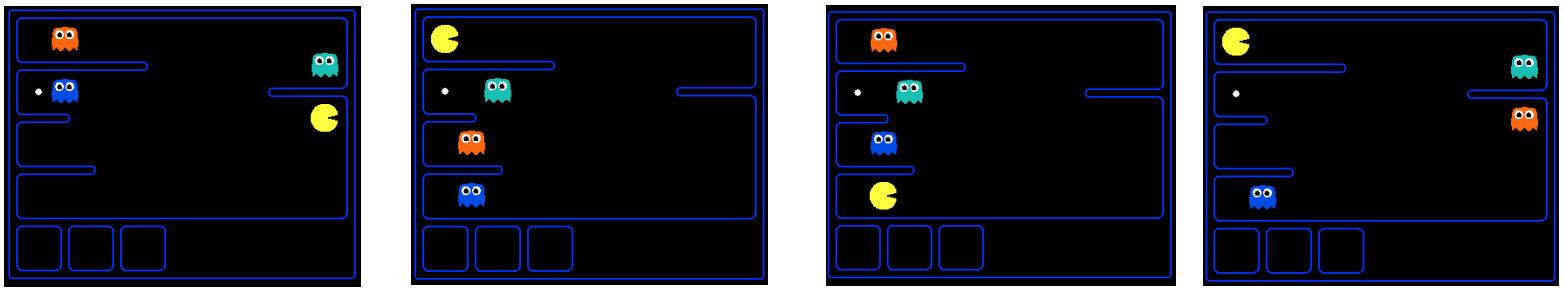
Using this approach will help the agent to learn quicker how to behave in this kind of situations.

Since we didn't get all the agents to work on this map, setting ghosts in static, we continued training on the next map (labAA4) when we considered it worked sufficiently good using random ghosts.

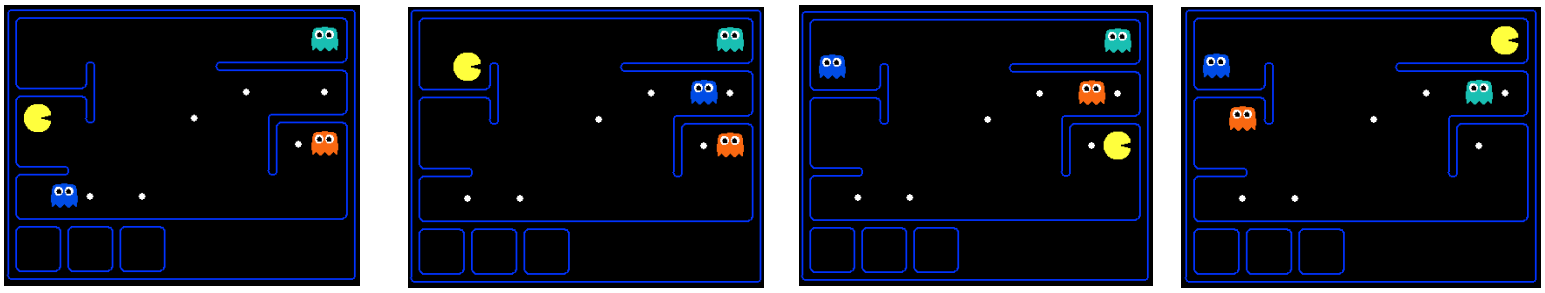
- **Map labAA4 and labAA5 (Difficulty: High)**

These two maps are kind of similar in the sense that they have walls coming from the left and the right. The unique difference is that the labAA5 map has also vertical walls. The process in these maps is very similar to the one followed for the third map. After playing with random ghosts we placed the ghost and the Pac-Man in different positions so that it could learn to go around horizontal and vertical walls. Although it was a good strategy not all our agents learnt to go around the wall in every situation (mainly due to the selection of attributes and reward). The photos show different layouts that we used in the learning process.

- *labAA4:*



- *labAA5:*



Strategies

In this section we will go over the different attribute's selections and reward functions that we built. We will analyze the experiments we performed using those combinations and the particularities of the training process in the different situations.

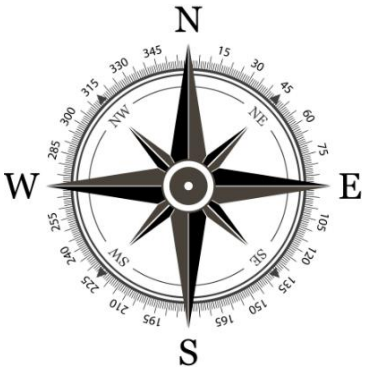
Regarding the attribute selection, we needed to balance a reasonable small q-table size with representative state information. We did not want to include attributes that would be really dependent on the map we are playing (like distance attributes for instance since that would depend on the dimensions of the map).

For the reward functions, we implemented them according to each approach and its state information (attributes). The goal we wanted to achieve was encouraging the Pac-Man to get closer to a ghost and eat it without getting stuck or doing random movements.

1st approach (closesg + getReward1)

Attributes (X, Y)

In terms of the attributes, our first idea was to select the relative position between the Pac-Man and the ghost. This was done using the function "*closestg*".



A good metaphor for understanding this selection of attributes is to imagine that the Pac-Man has a compass which tells him at every tick the directions that it must follow to reach the closest ghost.

We believed that this was a really powerful combination of attributes, since we were interested in the Pac-Man trying to eat the ghost as fast as possible, and this strategy in theory would encourage the agent to chase the closest ghost and finish quickly. However, we will see that after training this agent it has troubles caused by the walls.

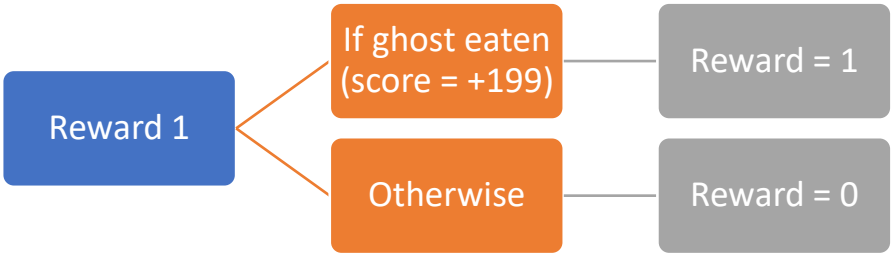
In this approach, there will be two attributes (relative position of the ghost in the X and Y axis) of 3 levels (0,1,2). The mapping of the "*computePosition*" method is given by this formula → $State\ number = 3 * X + Y$, being X and Y the relative position in the respective axes. The dimensions of the q-table are 9x4.

Reward

For this first approach we implemented a very basic reward system, if that would be enough for the Pac-Man to perform well.

We returned 1 if the difference between the score of two consecutive ticks is 199 (it has eaten a ghost); or 0 otherwise. We did not return a negative reward because we were aware that it did not work properly.

With this strategy, the agent would not try to eat the food, but since the score it gives is insignificant compared to the one obtained by eating a ghost, we thought it might work consistently.



Training

Let's analyze how this version of the agent behaved during the training process. For the first two maps (labs), it was really easy, since there were no walls or obstacles. From the third one on, each map was more difficult than the previous one.

The reason why it didn´t work in the maps with walls is because it doesn´t know how to go around these, it follows the directions where the Pac-Man is and if there is a static ghost behind a wall, this agent won´t be able to learn.

This represented a real issue, since the tendency of the Pac-Man could not be changed with any more training or tuning of the parameters (alpha, epsilon, omega). That is why for the upcoming approaches we thought of adding attributes in relation with the presence of walls.

2nd approach (closesg_wall + getReward3)

This approach was motivated by the willing to increase the frequency where we give the Pac-Man a reward. In other words, we wanted to make the process of learning from the rewards obtained more usual, to try and make the agent learn easier and quicker. We also wanted it to learn how to behave with walls, that is why we added an extra attribute in relation with this.

Attributes (X, Y, W)

For the state information, we kept the same pair of attributes: the closest ghost information, and we added an extra attribute which was 1 if there is an immediate wall in any of the direction the Pac-Man is moving, and 0 otherwise.

The mapping in this case is done by this formula → $State\ number\ = 6 * X + 2 * Y + W$, being W the attribute which refers to the wall presence. The size of the q-table in this approach is 18x4.

Reward (getReward3, an improvement of getReward2)

This system gives small rewards when the Pac-Man gets close to the nearest ghost (this is what geteward2 does, we haven´t explained it because getReward3 includes its functionality). It is done by comparing the distance between the closest ghost and the Pac-Man in the current and following tick. If it eats a ghost (difference of scores equal to 199), it returns 1. If it approaches the nearest ghost two things can happen. If when approaching it hits a wall which has a ghost behind it, the reward

is 0 (this is the detail which getReward3 includes), and if there is no wall, then it would be 0.3.

The reason why the reward for approaching is smaller is because we are giving this whenever the Pac-Man approaches the ghost and that will happen in an episode a lot of times. This idea tries to encourage the Pac-Man to approach the closest ghost.

Training

This agent performed perfectly in the three first maps, either with Random or Static ghosts. It learned to surpass walls which emerge from the let border. The problem was that when we started training this agent in the labAA4 and labAA5, and we tried to teach him how to overcome walls that emerge from the right border it unlearnt how to overcome walls from the left border since in the state representation there is no difference for the agent between these two kinds of walls. That is why we thought on the next approach, which would distinguish the five main kind of walls and act accordingly.

3nd approach (closesg_wall2 + wallchecker + getReward4)

This is our best and more complex agent. We have to mention that it was an afterthought, we programed and trained this agent a day before the submission of the project and it turned out to be the best one as you will see. The variation that we have done in this respect with respect the one before is the distinction of the five main kinds of walls. Depending from which border the wall emerges we have a

different kind of wall (right, left, up, down) and the fifth class is the walls which do not emerge from the borders.

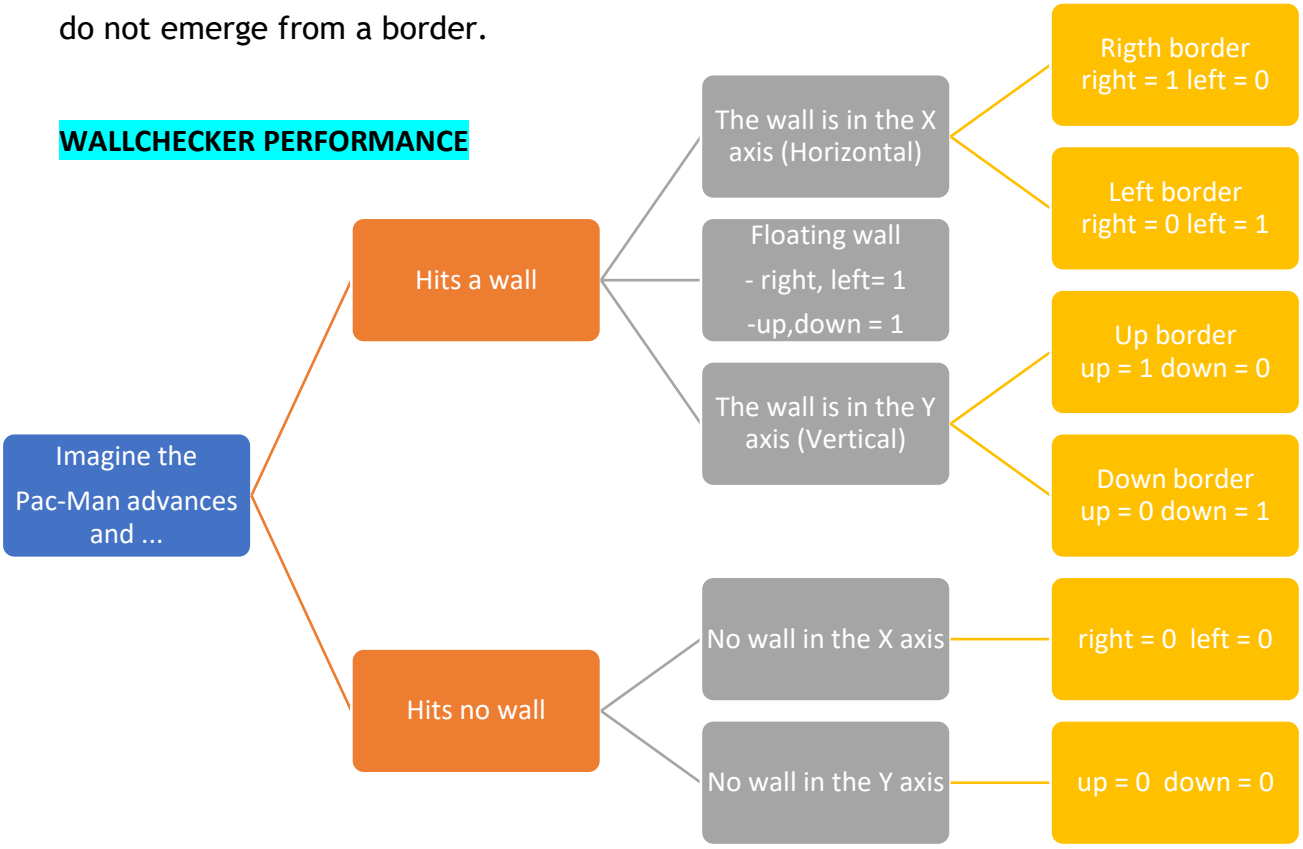
As you may have noticed the methods *"closesg_wall2"* (used for the attribute selection) and *"wallchecker"* (used for detecting which kind of wall we are facing) haven´t been explained in the **"auxiliary methods"** section of our report. This is because they are kind of messy functions to explain in detail so we will give a summarized explanation of these while we explain the methodology of this final agent.

Attributes (X, Y, right, left, up, down)

The position arguments (x, y) are obtained following the same procedure as in the before approaches. Additionally, inside the "closesg_wall2" we call the "wallchecker" which will tell the kind of wall the Pac-Man has hit. We will explain how this selection of attributes works conceptually so it is easier to understand.

Clarify that when we refer to a "floating wall" we are referring to the walls which do not emerge from a border.

WALLCHECKER PERFORMANCE



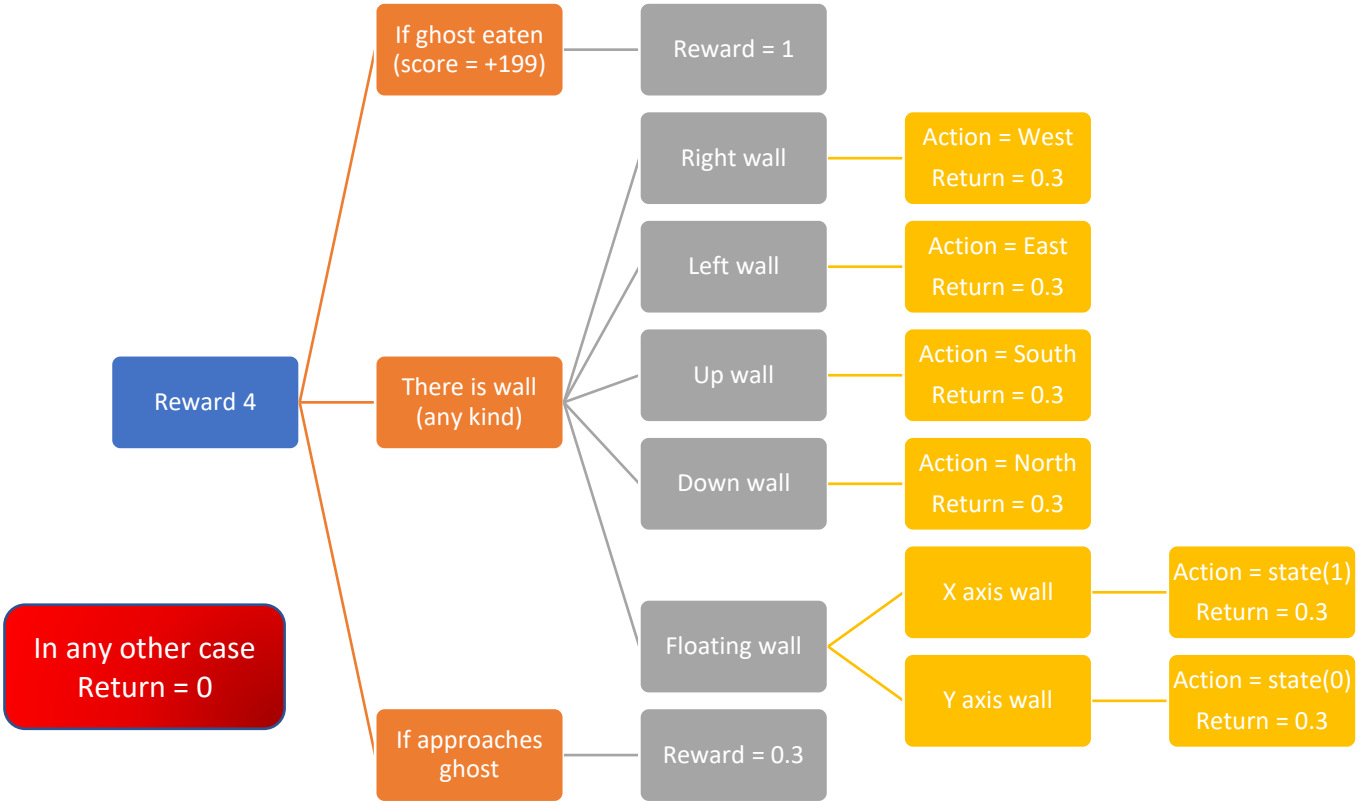
The mapping for this approach is larger since the q-table is of 144x4.

$$State\ number\ = 48 * X + 16 * Y + 8 * right + 4 * left + 2 * up + down$$

Reward (getReward4, an improvement of the getReward3)

The concept is the same as in the "getReward3" with the difference that now we refer as different situations whether the Pac-Man is approaching a ghost or if he hits a wall (since he probably needs to move away for being able to eat the ghost afterwards). As we have done before, we will explain the functioning of the "getReward4" conceptually so it is easier to understand.

GETREWARD4 PROCEDURE



Training

We expected that the training of this agent would take much more time since it has a bigger q-table, however it did not. This is because if you take a look to the q-table the majority of the rows are only zeros. This is because the maps proposed for the training of the agents (labAA) didn't cover all the situations that this model needs to be complete (like for example up or down walls). Despite not being complete it is our best agent. It was able to work perfectly on the first four maps (using random and static ghosts) and works pretty good in the fifth map (using just random).

We haven´t been able to make it work in the fifth map since the vertical walls which don´t emerge from a border are a problem (don´t really know how to overcome these).

Results

After having explained all the approaches we followed, we will move on to testing the final agents in the 5 maps. In addition, if one performs really well in these labs, we will try to test the agent in some other maps.
We will report the results both qualitatively, explaining the general conclusions for each agent in terms of its behavior, positive and negative aspects; and quantitively, with some graphical representations of the score obtained in different games.

Qualitative Analysis

In order to perform this type of analysis, we used the trained q-table that let the agent work as smoothly as possible in all labs. For this reason, we changed the alpha (learning rate) to 0; since we now do not want to update the q-table in every game. Furthermore, we switched epsilon to 0, to let the agent have complete control of the actions, and avoid randomness.

1st approach

Recall that this approach uses our most basic agent. This agent uses the pair attribute with the relative position to the closest ghost, and a reward function that gives 1, if it eats a ghost), and 0, otherwise.
We will a quick summary about the testing of the maps:

- On lab 1 and 2, it has no problem finishing the map with ease with static and random ghost
- On lab 3:
    - Static: it gets stuck and cannot finish the game
    - Random: it is able to eat all the ghosts
- On lab 4:
    - Static: this time, again, it is not able to eat all the ghosts
    - Random: it eventually catches all the ghosts and finishes the game
- On lab 5:
    - Static: cannot finish the game
    - Random: it eventually completes it

In conclusion, this agent performs terribly on static ghost configurations. However, it gets the job done decently with random ghosts. This is not something to brag about, since it is thanks to the randomness of the ghosts, that end up going right to the Pac-Man to be eaten.
We noticed how the Pac-Man has a similar behavior than the agent from Tutorial 1, since it finds the closest ghost like if it was programmed to do so, but it cannot avoid walls or learn how to eat the ghosts consistently.

2nd approach

This approach incorporates a new attribute, that indicates if there is a wall between the ghost and the Pac-Man. It uses Reward 3. We will hope to see a big improvement in the evaluation performance, since we could tell it would be a lot better from the training process.
We will give a quick summary about the testing of the maps:

- On lab 1, it eats all the ghosts quickly
- On lab 2, the same happens. However, in certain moments it got a little bit stuck
- On lab 3:
    - Static: for the first time, it can complete this map with this configuration. It works very well
    - Random: it can eat all the ghosts consistently, just as other agents
- On lab 4:
    - Static: again, it is able to finish the games consistently
    - Random: works smoothly
- On lab 5:
    - Static: unfortunately, after eating two of the ghosts it gets stuck between two walls
    - Random: it is quicker to eat all the ghosts and it has no problems.

We can see how this agent outperforms the previous ones, although it has some important issues. We think that its design helps in avoiding getting stuck with the walls. Therefore, it could work very consistently in unknown situations, especially with random ghosts.

3th approach

As we explain in the strategies section, this last approach changes the dynamic of the detection of the close walls in order to be more precise about the action the Pac-Man should perform. Since this is our best candidate, we will also try it on some other harder maps.

We will give a quick summary about the testing of the maps:

- On lab1 and lab2: it had no problem, once again
- On lab3: similarly, as the previous agent, it worked well in both modes
- On lab 4: it was very consistent with the ticks needed and score obtained
- On lab 5:
    - Static: it ate all the ghosts except one and it got stuck in a corner, so it did not finish it
    - Random: there was a big variability between games, but it managed to complete it. However, it took considerably longer to finish.
- Other maps, we tested it on:
    - bigHunt: it could only finish it in random mode. It took the Pacman a lot of ticks but it ended up eating all the ghosts
    - smallHunt, oneHunt: for these two maps, the results were pretty good in terms of score and ticks needed. It completed the game in both modes

To sum up, this agent is the most powerful of all of the agents. It is very consistent in al the maps and even works well in other new maps.

Conclusions

A similar conclusion we can extract from all the games is that we score obtained in random and static ghosts is similar, but the ticks needed have a big variability.

Quantitative Analysis

Now we will proceed will the analysis of the score obtained and the number of ticks needed. In our “game.py”, we coded a few lines to print the score and ticks after finishing each game.

We will test each map with both static and random ghosts, 3 times each, and we will compute the averages. For the static ghost games, it obviously did not change from game to game, but we wanted to be consistent and homogeneous with the

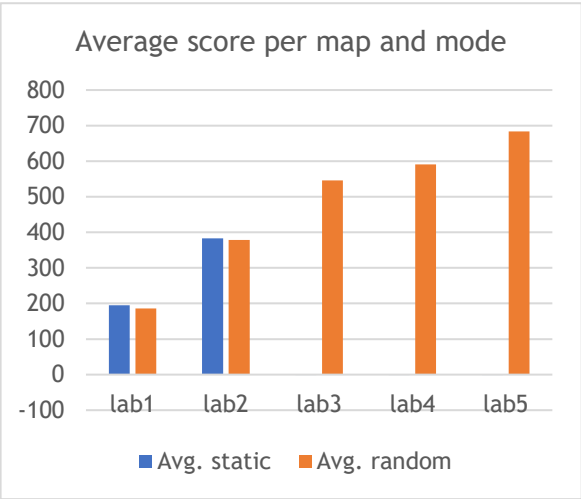
data. For the case of random ghosts, there is definitely a big variability due to randomness.

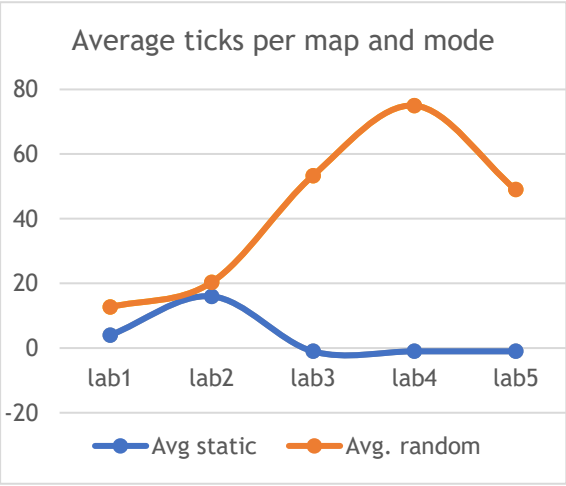
Note: if the Pac-Man could not finish the game in a reasonable period of time, i.e. it got stuck or could not eat all the ghosts, or if we did not even try that map because it was not good enough, we set the score = -1 and the ticks needed = -1 However, in these cases we did not take these -1's into account for the average, we set it to 0. This represented a kind of "disqualification" in those maps.

1st approach

Score (rounded to units)					
Type	lab1	lab2	lab3	lab4	lab5
Static	195	383	-1	-1	-1
Static	195	383	-1	-1	-1
Static	195	383	-1	-1	-1
Avg. static	195	383	-1	-1	-1
Random	188	376	557	575	703
Random	175	381	548	629	675
Random	196	379	532	568	672
Avg. random	186	379	546	591	683
Average	191	381	546	591	683

Ticks needed (rounded to units)					
Type	lab1	lab2	lab3	lab4	lab5
Static	4	16	-1	-1	-1
Static	4	16	-1	-1	-1
Static	4	16	-1	-1	-1
Avg. static	4	16	-1	-1	-1
Random	11	23	42	24	96
Random	24	18	51	170	24
Random	3	20	67	31	27
Avg. random	13	20	53	75	49
Average	8	18	53	75	49

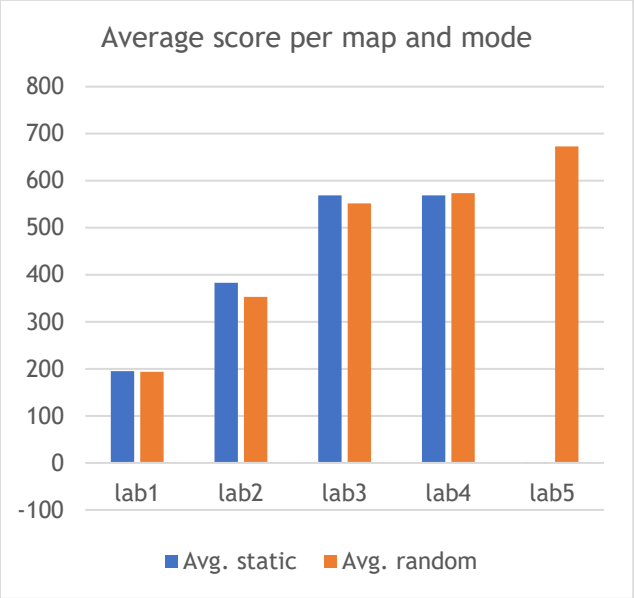


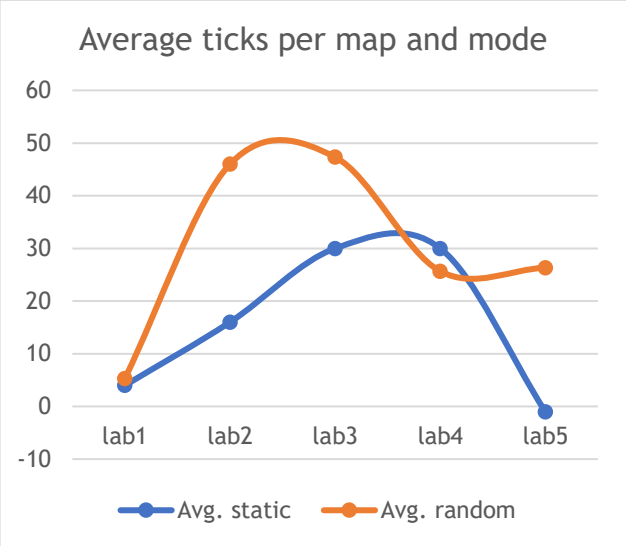


2nd approach

Score (rounded to units)					
Type	lab1	lab2	lab3	lab4	lab5
Static	195	383	569	569	-1
Static	195	383	569	569	-1
Static	195	383	569	569	-1
Avg. static	195	383	569	569	-1
Random	192	382	553	571	574
Random	192	305	570	582	678
Random	197	372	532	567	766
Avg. random	194	353	552	573	673
Average	194	368	560	571	673

Ticks needed (rounded to units)					
Type	lab1	lab2	lab3	lab4	lab5
Static	4	16	30	30	-1
Static	4	16	30	30	-1
Static	4	16	30	30	-1
Avg. static	4	16	30	30	-1
Random	7	17	46	28	25
Random	7	94	29	17	21
Random	2	27	67	32	33
Avg. random	5	46	47	26	26
Average	5	31	39	28	26

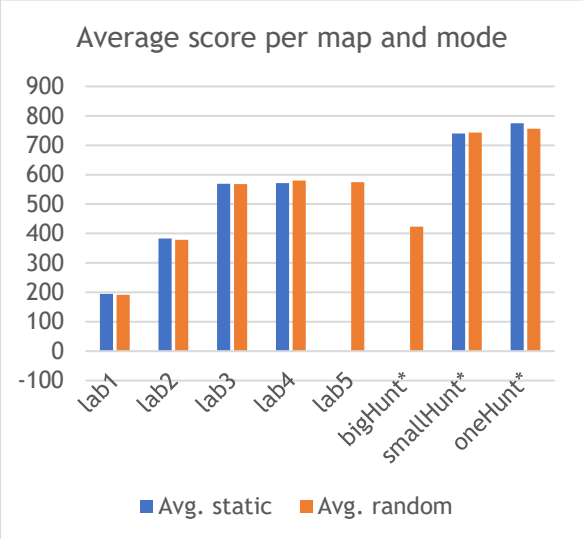


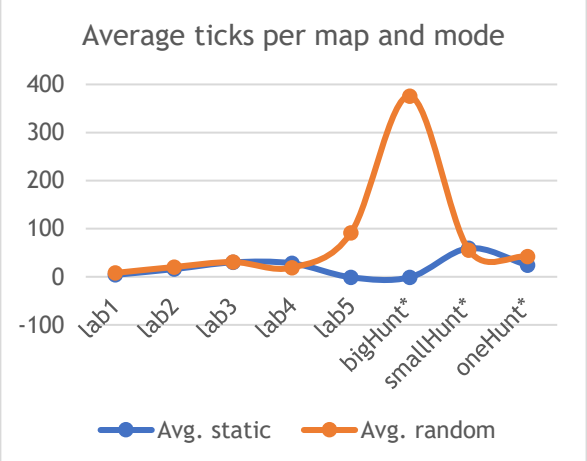


3th approach

Score (rounded to units)								
Type	lab1	lab2	lab3	lab4	lab5	bigHunt*	smallHunt*	oneHunt*
Static	195	383	569	571	-1	-1	729	775
Static	195	383	569	571	-1	-1	733	775
Static	195	383	569	571	-1	-1	757	775
Avg. static	195	383	569	571	-1	-1	740	775
Random	188	371	569	582	593	269	739	764
Random	189	380	569	579	503	564	776	752
Random	196	385	567	579	627	436	715	754
Avg. random	191	379	568	580	574	423	743	757
Average	193	381	569	576	287	269	739	764

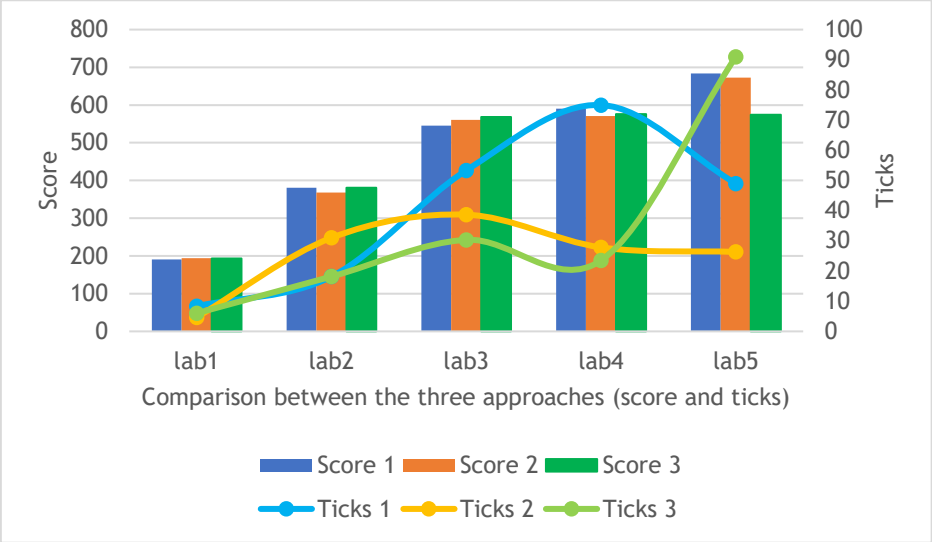
Ticks needed (rounded to units)								
Type	lab1	lab2	lab3	lab4	lab5	bigHunt*	smallHunt*	oneHunt*
Static	4	16	30	28	-1	-1	70	24
Static	4	16	30	28	-1	-1	66	24
Static	4	16	30	28	-1	-1	42	24
Avg. static	4	16	30	28	-1	-1	59	24
Random	11	28	30	17	106	530	60	35
Random	10	19	30	20	96	235	23	47
Random	3	14	32	20	72	363	84	45
Avg. random	8	20	31	19	91	376	56	42
Average	6	18	30	24	91	376	56	42





Final comparison

As a quick visual summary, below we show the average score and ticks needed in each approach. The good performance of the last approach cannot be so clearly seen graphically, but it is really noticeable when evaluating the lab maps, especially with static ghosts. Not to mention that the first approach looks better than what it really is, when it actually cannot complete 3 maps in static mode.



**If the agent could not finish the map in static mode, the final average did only take into account the random ghost configuration*

Conclusions

All in all, we have decided to choose the third agent as the one we will present on the competition, taking into account that is the one which has worked perfectly on the first four maps (that means it never gets stuck on those, it does not matter if you use random or static ghost) and quite well on the fifth.

We have learnt many things while going through the different stages of this project. The most relevant result is that reinforcement learning works much better (at least for us), than the previous models about classification/regression. However, the automatic agent has some serious issues sometimes. Thus, we think that the ideal way of automating Pacman would be by combining it with a more powerful model, like a Neural Network.