Τελική Εργασία – Κούτρας Δημήτριος

Σκοπός της εργασίας είναι να επιλύσουμε ένα παιχνίδι του Atari 2600 μέσω της εφαρμογής διάφορων τεχνικών των κυψελιδωτών αυτομάτων.

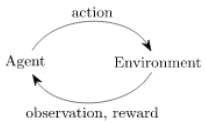
Περίληψη:
1. Εισαγωγή OpenAI Gym
2. Atari 2600 – Bank Heist
3. Προσεγγιση Προβλήματος

OpenAI Gym

Το gym αποτελεί ένα toolkit που χρησιμοποιείται για την αναπτυξη και την σύγκριση διαφόρων αλγορίθμων reinforcement learning. Υποστηρίζει παρα πολλα διαφορετικά περιβάλοντα (απο προβλήματα κλασσικου αυτόματαου ελεγχουμ, μέχρι και παιχνίδια Atari) τ αοποία έχουν ένα κοινό itrenface, προκειμένου οι αλγόριθμοι που αναπτύσονται να μπορούν να εφαρμοστούν εύκολα σε όλα. Το προαναφερθεν interface βασίζεται στον κλασσικό κύκλο Agent→ Action → Enviroment → observation,reward→ Agent. Ουσιαστικά στο πλαίσιο της εργασία χρησιμοποιήθηκε μόνο η συναρτηση observation (που αποτελεί τα pixel που εμφανίζονται στην οθόνη απο το Atari 2600) και η action (με την οποία στέλναμε την επόμενη κίνηση που θα εφαρμόσει ο agent μας - δηλαδή ποιά κουμπιά απο το joystick πατήσαμε).

- `observation` (**object**): an environment-specific object representing your observation of the environment. For example, pixel data from a camera, joint angles and joint velocities of a robot, or the board state in a board game.
- `reward` (**float**): amount of reward achieved by the previous action. The scale varies between environments, but the goal is always to increase your total reward.
- `done` (**boolean**): whether it's time to `reset` the environment again. Most (but not all) tasks are divided up into well-defined episodes, and `done` being `True` indicates the episode has terminated. (For example, perhaps the pole tipped too far, or you lost your last life.)
- `info` (**dict**): diagnostic information useful for debugging. It can sometimes be useful for learning (for example, it might contain the raw probabilities behind the environment's last state change). However, official evaluations of your agent are not allowed to use this for learning.

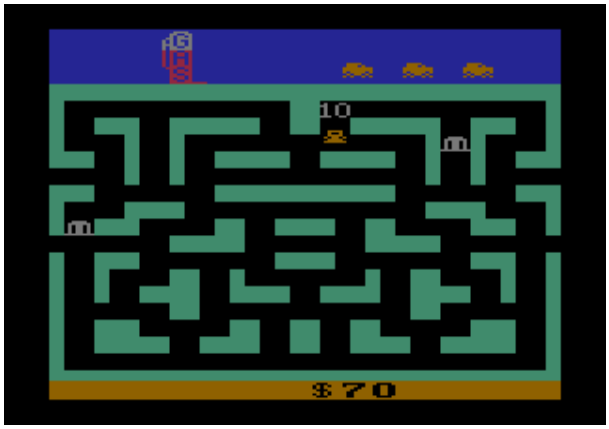
This is just an implementation of the classic "agent-environment loop". Each timestep, the agent chooses an `action`, and the environment returns an `observation` and a `reward`.

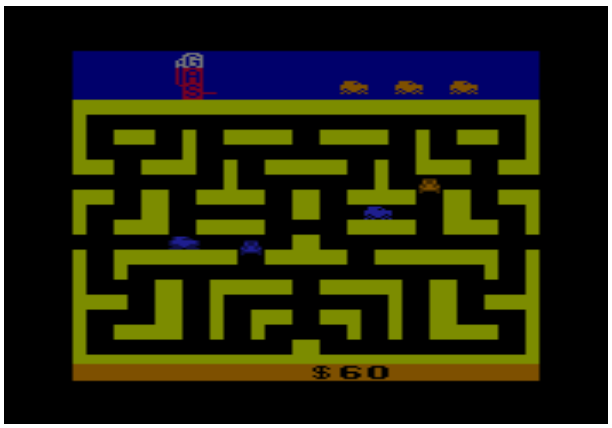


Atari 2600 – Bank Heist

Απ όλα τα διαθέσιμα παιχνίδια του Atari 2600 επέλεξα το Bank Heist, γιατί ποτελεί το πιο συναρπαστικό και ενδιαφέρον – προφανως και κάνω πλάκα - το επέλεξα γιατί τα στοιχεία που χρειάζονται για να πάρεις αποφάσεις είναι σαφώς ορισμένα, δεν αλλάζουν γρήγορα χρώμα (όπως για παραδειγμα οι τα “φαντασματάκια” που κυνηγάνε το Pac-Man) και η στρατιγική που απαιτείται να ακολουθήσει ο παίκτης είναι σχετικά απλή και η πολυπλοκόητητας αυξάνεται γραμμικά (δηλαδή στα αρχικά frames μπορούμε με την χρήση κάποιων κόνανων να την αναπαραστήσουμε).
Το παιχνίδια είναι αρκετά απλό, αποτελείται απο έναν λαμβύρινθό σαν το Pac-Mac (στο οποίο αναφέρομαι συχνά γιατι αρχικά αυτό είχα στο μυαλό μου να λύσω) στο οποίο υπάρχουν κάποιες τράπεζες. Ο παίκτης χειρίζεται ένα όχημα (gateway-car) και πρέπει να το κατευθύνει στις τράπεζες

τις οποίες οταν ακουμπήσει ουσιαστικά τις ληστεύει. Αφού ληστέψει μια τράπεζα αυτή εμφανίζει το ποσό που πήρε ο παίκτης απο αυτής (reward) και μετα απο κάποιο χρόνο στην θέση της εμφανίζεται ένα αστυνομικό όχημα, το οποίο προσπαθεί να πιάσει το gateaway car.





Προσεγγιση Προβλήματος

Επομένως, σε κάθε στιγμή ο παίκτης πρέπει να λάβει μια αποφαση για το που θα κινηθεί με βάση δύο παραγοντες: ποιες είναι οι θέσεις της κάθε τράπεζας και ποιες είναι οι θέσεις του εκάστοτε περιοπολικού. Η ποσεγγιση μου ήταν κάθε pixel να εκφράζει ένα κελί στο Κυψελιδωτό Αυτοματο μου και κάθε κελί να έχει μια τιμή χρησιμότητας (uitility): όσο πιο κοντά σε μια τράπεζα ειναι τοσο υψηλότερη θα ήταν η χρησιμότητα του, ενω οσο πιο κοντα σε ενα περιπολικό ήταν τοσο χαμηλότερη.

Therefore, at each moment our decision is based upon two factors: (1) the positions of the banks (positive sources) and (2) the positions of the police vehicles (negative sources). My approach is to use a 2-D Cellular Automata, where each cell corresponds to a pixel. The value of the cell represents a *utility* which indicates how beneficiary would be a move towards that direction. In other words, cells close to a police vehicle will have a low utility value and cells close to banks will have high utility value.

Θετική Χρησιμότητα – Bank Utility

Για τον υπολογισμό της θετικής χρησιμότητας κάθε κελιού χρησιμοποίησα τον παρακάτω κώδικα

```
155    def updateUtilityBanks(self):
156
157        if VERBOSE and INFO:
158            print('[INFO]: updating utility for banks')
159
160        rtn = np.zeros([self.rows, self.columns])
161
162        rtn[self.currentBankLocation[:,:,0]] = 1.0
163
164        rtn = self.clearBankNoise(rtn)
165
166        for i in range(BANK_RANGE):
167            (x,y) = np.where(rtn>0.)
168
169            for x2,y2 in zip(x,y):
170
171                ego = rtn[x2][y2]
172                down = rtn[x2+1][y2]
173                up = rtn[x2-1][y2]
174                right = rtn[x2][y2+1]
175                left = rtn[x2][y2-1]
176
177                avgMatrix = np.array([ego,down,left,right,up])
178                avgWeights = avgMatrix != 0
179                utilityValue = np.average(avgMatrix, weights=avgWeights)*UTILITYDECAY
180                # utilityValue = ego*UTILITYDECAY
181
182                rtn[x2][y2+1] = utilityValue if (rtn[x2][y2+1]==0. and self.freeSpace[x2][y2+1][0] ) else rtn[x2][y2+1]
183                rtn[x2][y2-1] = utilityValue if (rtn[x2][y2-1]==0. and self.freeSpace[x2][y2-1][0] ) else rtn[x2][y2-1]
184                rtn[x2-1][y2] = utilityValue if (rtn[x2-1][y2]==0. and self.freeSpace[x2-1][y2][0] ) else rtn[x2-1][y2]
185                rtn[x2+1][y2] = utilityValue if (rtn[x2+1][y2]==0. and self.freeSpace[x2+1][y2][0] ) else rtn[x2+1][y2]
186
187            self.printSingle(rtn,'Utility Banks {} [step]'.format(i),i)
188        self.utilityBanks = rtn
```

Αρχικά δημιουργώ έναν πίνακα ίσο με τις διαστάσεις της οθόνης του Atari 2600 (210X160 pixel) (γαρμμή 160) και θέτω τις τιμές ίσες με μηδέν. Στην συνέχεια αναγνωρίζω τις θέσεις που βρίσκονται οι τράπεζες και τις θέτω ίσες με ένα (η αναγνώριση γίνεται με βάση το χρώμα του κελιου, πχ οι τράζες έχουνε rgb(214, 214, 214), το περιπολικό rgb(24,26,167) κτλ). Εδω συνάντησα το εξή πρόβλημα: οταν το οχημα ληστεψει μια τραπεζα αυτή εμφανίζει τον αριθμό των χρηματων που είχε, αυτό ο αριθμος έχει ίδιο χρώμα με την τράπεζα, πρεπει με καποιο τρόπο να ξεχωρίζουμε τις τραπεζες απο αυτους τους αριθμούς, η συνάρτηση clearBankNoise (γραμμή 164) αντιμετωπίζει αυτό το πρόβλημα σβήνοντας τους αριθμούς.

Στην συνέχεια πρέπει με καποιο τρόπο-κανόνα να εξαπλώσουμε την χρησιμοτητα και στα υπολοιπα κελια. Αυτό το υλοποιούμε μέσω του For (γραμμή 166) – το Bank_Range εκφράζει το πόσο μακριά θα εξαπλωθεί η χρησοτητα απο μία τραπεζα (ιδανικά θα πρεπει να είναι ίσο με το μέγεθος του χωρου μας, ομως για πρακτικού λόγους, που σχετίζονται με την ταχύτητα εκτέλεσης, το θέτω λίγο μικρότερο). Για κάθε μή-μηδενικό στοιχείο υπολογίζω τον μέσο όρο των γύρο κελίών του (γραμμή 171-179) και αυτό το κελί αποδίδει σε γειτονικά κελιά τα οποία αποτελούν ελεύθερο χώρο προς τον οποίο μπορεί να κινηθεί το όχημα μας αυτή την τιμή μειωμένει κατα έναν παραγοντα UTILITYDECAY.

Σημείωση 1: η παραπανω τεχνική δεν είναι βέλτιστη ως προς τον υπολογιστικό χρόνο, δεδομένου οτι περιλαμβάνει παρα πολλους περιτούς υπολογισμούς, έχω σκεφτεί και καλύτερο με πίνακες που

θα κάνουνε roll προς κάθε κατεύθυνση και θα υπολογίζουμε μόνο τα οριακά κελία που έιναι ελευθερα, αλλα θα σας την αναλύσω αφού την δοκιμάσω και όντως έχουμε θετικά αποτελέσματα.

Σημείωση 2: η παραπανω τεχνική μου φαίνεται κατα κάοιο τρόπο αρκετα greedy και το οχημα θα κινείται ουσιαστικά προς την τραπεζα που είναι κοντά του. Ωστόσο, αυτό δν ξερω αν μας οδηγεί σε ένα βέλτιστο path για ολες τις τραπεζες. Μια άλλη προσεγγιση που έχω στο μυαλό μου είναι να αναγωρίζουμε ποσες ξεχωριστες τραπεζες υπαρχουν (με την χρηση πχ K – Nearest Neighbors) να υπολογίζουμε έναν πίνακα utility για κάθε τραπεζα ξεχωριστα και μετα να αθροίζουμε τους πολλούς πίνακες. Μια τετοια τεχνική θα αυξανε κατα πολύ το υπολογιστικό κόστος – παραπάνω απο γραμμικά σε σχέση με τον αριθμών των τραπεζων μας – αλλα ίσως είχε ενδιαφέρον να την μελετήσουμε …

Positve Sources – Bank Utility

```
169
170     def updateUtilityBanks(self):
171
172         if VERBOSE and INFO:
173             print('[INFO]: updating utility for banks')
174
175         rtn = np.zeros([self.rows, self.columns])
176
177         rtn[self.currentBankLocation[:,:,0]] = 1.0
178
179         rtn = self.clearBankNoise(rtn)
180
181         for i in range(BANK_RANGE):
182             (x,y) = self.boundaries(rtn)
183
184             for x2,y2 in zip(x,y):
185
186                 down = rtn[x2+1][y2]
187                 up = rtn[x2-1][y2]
188                 right = rtn[x2][y2+1]
189                 left = rtn[x2][y2-1]
190
191                 avgMatrix = np.array([down,left,right,up])
192                 avgWeights = avgMatrix != 0
193                 utilityValue = np.average(avgMatrix, weights=avgWeights)*UTILITYDECAY
194
195                 rtn[x2][y2] = utilityValue
196
197             # self.printSingle(rtn,'Utility Banks {} [step]'.format(i),i)
198         self.utilityBanks = rtn
199
200
```

First, we create a 2-D CA which values are all set equal to zero (line 175, integers rows and columns correspond to the size of the original CA, which is based on the Atari 2600 image). Then, we set the cells which consist a positive source (bank) equal to one. Note, that detection is based only on the coloring of the pixels of the original image and this is why we must clear any cells that have the same color with banks but are not banks (when getaway car touches a bank the latter will transform into a number with the same coloring as the banks). For a finite number of steps (BANK_RANGE) we will update the values of the cells, ideally the aforementioned number should cover all of our 2-D CA, however due to limited computational power I used a lower value. We obtain the *boundary cells*, those are cells which at least one of their neighbor is a cell with value

different than zero. For each one of the boundary cells we calculate the average value of their von Neumann neighborhood and multiply it with a calibration factor (UTILITYDECAY).

When the process is finished each cell of the generated 2-D CA will depict how close ta a positive source (bank) it is located.

Boundary Cells

After experimenting with many different techniques I concluded that the best approach is to use something like masking and end up with a 2-D CA which cells have boolean values and only boundary cells are *True* . Let me be more detailed, at any iteration of the BANK_RANGE I create 4 copies of the CA, each rolled towards a direction (up, down, left, right), named r1, r2, r3, r4.

```
153     def boundaries(self, rtn):
154
155         r1 = np.roll(rtn,1,axis=0)
156         r2 = np.roll(rtn,-1,axis=0)
157         r3 = np.roll(rtn,1,axis=1)
158         r4 = np.roll(rtn,-1,axis=1)
159
160         xor1 = np.logical_and( np.logical_xor(r1, rtn) , self.freeSpace[:,:,0])
161         xor2 = np.logical_and( np.logical_xor(r2, rtn) , self.freeSpace[:,:,0])
162         xor3 = np.logical_and( np.logical_xor(r3, rtn) , self.freeSpace[:,:,0])
163         xor4 = np.logical_and( np.logical_xor(r4, rtn) , self.freeSpace[:,:,0])
164
165         bound = np.logical_or( np.logical_or(xor1,xor2), np.logical_or(xor3,xor4))
166
167         return np.where(bound)
168
```

I create 4 new CA which each cell has the value of the logical xor apllied to the original CA and each one of the rolled CA s , named xor1, xor2, xor3, xor4 (I also apply a logical and to filter out any cells that can not be accessed by the gateaway car – they are not freeSpace). Each of the aforementioned xor CA indicates the boundary cell towards the corresponding direction. By applying a logical or to all the CA we end up with the final CA, from which we find the True cell which consist the boundary cells

Negative Source – Police Utility

We repeat the aforementioned process for each of the police vehicle. The only difference is that at in the end we multiply all the values of the CA with minus one,in order to prevent gateaway vehicle from moving towards that direction.

```
98      def updateUtilityPoliceVeh(self):
99          if VERBOSE and INFO:
100             print('[INFO]: updating utility for police veh')
101
102         rtn = np.zeros([self.rows, self.columns])
103
104         rtn[self.currentPoliceLocation[:,:,0]] = 1.0
105
106         for i in range(POLICE_RANGE):
107             (x,y) = self.boundaries(rtn)
108
109             for x2,y2 in zip(x,y):
110
111                 down = rtn[x2+1][y2]
112                 up = rtn[x2-1][y2]
113                 right = rtn[x2][y2+1]
114                 left = rtn[x2][y2-1]
115
116                 avgMatrix = np.array([down,left,right,up])
117                 avgWeights = avgMatrix != 0
118                 # utilityValue = np.average(avgMatrix, weights=avgWeights)*POLICE_DECAY*
119                 utilityValue = np.average(avgMatrix, weights=avgWeights)*POLICE_DECAY
120                 # utilityValue = ego*UTILITYDECAY
121
122                 rtn[x2][y2] = utilityValue
123
124         self.utilityPoliceVeh = rtn*(-1)
```

Αθροιστική Χρησιμότητα – Aggregated Utility

Αφού υπολογίσουμε τις δύο ειδών χρησιμότητες, τις αθροίζουμε προκειμένου να καταλήξουμε στην συνολική. Στην συνέχεια βρίσκουμε την κεντρική θέση του gate away οχήατος και υπολογίζουμε την μέση τιμή προς κάθε μια απο τις 4 κατευθύνσεις.

```
280     def updatesurroundingUtility(self,x,y):
281
282         upMatrix = self.utility[x-STRONGSIDE:x,y-WEAKSIDE:y+WEAKSIDE]
283         downMatrix = self.utility[x:x+STRONGSIDE,y-WEAKSIDE:y+WEAKSIDE]
284         rightMatrix = self.utility[x-WEAKSIDE:x+WEAKSIDE,y:y+STRONGSIDE]
285         leftMatrix = self.utility[x-WEAKSIDE:x+WEAKSIDE,y-STRONGSIDE:y]
286
287         surr_utility = {"RIGHT":np.sum(rightMatrix)/(STRONGSIDE*WEAKSIDE),
288                         "UP":np.sum(upMatrix)/(STRONGSIDE*WEAKSIDE),
289                         "LEFT":np.sum(leftMatrix)/(STRONGSIDE*WEAKSIDE),
290                         "DOWN":np.sum(downMatrix)/(STRONGSIDE*WEAKSIDE)}
291
292         if VERBOSE and SIM:
293             print('[SIM]: surrounding utility dict',surr_utility)
294
295         return max(surr_utility,key=surr_utility.get)
```

Προφανώς και το οχημα θα κινηθεί προς την κατεύθυνση με την υψηλότερη μέση τιμή.

Aggregated Utility

Once we have created the two CA grids (one for positive sources and one for negative) we will add the values of each indivudual cell. The result will be our final 2-D CA. Then we locate the X-Y coordinates of the gateaway car and calculate the average value towards all the directions (up, down, left, right). The variables STRONGSIDE indicates how far towards a direction we will average the cells and the WEAKSIDE how many cells right and left of that direction we will take into account.

Ανανέωση Utility

Το ερώτημα έιναι τώρα κάθε πότε θα πρέπει να εφαρμόζουμε την υπολογιστικά κοστοβόρα διαδικασία ανανέσωσης της χρησιμότητας (περίπου 100 φορές πιο κοστοβόρα). Η θετική χρησιμότητα ανανεώνεται όποτε υπάρχει διαφορά στην θέση έστω και μιας τράπεζας (κατι που δν συνμβαίνει και τοσοσ συχνά), απ την αλλη η αρνητική χρησιμότητα ανανενώνεται συχνότερα αφου πρέεπι να ξεουμε ποσο κοντά μας είανι τα αστυνομικά οχήματα.

```
315     if (self.currentBankLocation != self.previousBankLocation).any() and (self.updateStepBank <= self.simStep):
316
317         if VERBOSE and SIM:
318             print('[SIM]: Difference in Bank Locations')
319
320         self.utility = np.zeros([self.rows, self.columns])
321
322         self.updateUtilityBanks()
323         self.aggregateUtility()
324
325         self.previousBankLocation = self.currentBankLocation
326         self.updateStepBank = self.simStep + UPDATE_INTERVAL_BANK
327
328         self.print(self.utility, title="Aggregated Utility")
```

```
330         if (self.currentPoliceLocation != self.previousPoliceLocation).any() and (self.updateStepPolice <= self.simStep):
331 
332             xPolice, yPolice = self.locatePoliceVeh()
333             dist = self.distance(x,y,xPolice, yPolice)
334 
335             if dist<DIST:
336 
337                 if VERBOSE and SIM:
338                     print('[SIM]: Difference in Police Car Locations')
339 
340                 self.utility = np.zeros([self.rows, self.columns])
341 
342                 self.updateUtilityPoliceVeh()
343                 self.aggregateUtility()
344 
345                 self.previousPoliceLocation = self.currentPoliceLocation
346                 self.updateStepPolice = self.simStep + UPDATE_INTERVAL_POLICE
347 
348                 self.print(self.utility, title="Aggregated Utility")
```