

Τελική Εργασία – Κούτρας Δημήτριος

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

Περίληψη:

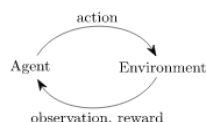
1. Εισαγωγή OpenAI Gym
2. Atari 2600 – Bank Heist
3. Προσεγγιση Προβλήματος

OpenAI Gym

Το gym αποτελεί ένα toolkit που χρησιμοποιείται για την ανάπτυξη και την σύγκριση διαφόρων αλγορίθμων reinforcement learning. Υποστηρίζει παρα πολλά διαφορετικά περιβάλλοντα (απο προβλήματα κλασσικού αυτόματου ελεγχου, μέχρι και παιχνίδια Atari) τα οποία έχουν ένα κοινό interface, προκειμένου οι αλγόριθμοι που αναπτύσσονται να μπορούν να εφαρμοστούν εύκολα σε όλα. Το προαναφερθέν interface βασίζεται στον κλασσικό κύκλο Agent → Action → Environment → observation, reward → Agent. Ουσιαστικά στο πλαίσιο της εργασίας χρησιμοποιήθηκε μόνο η συνάρτηση 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.

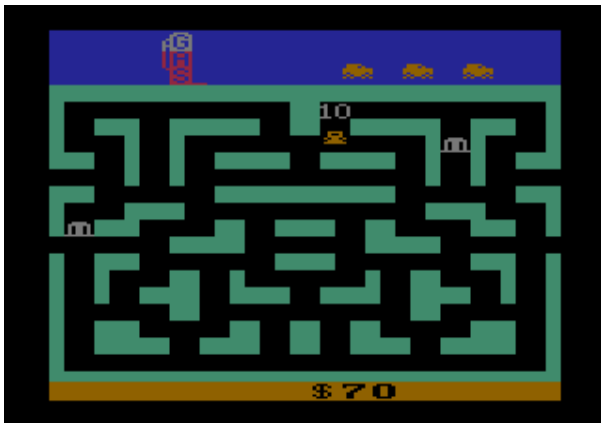
This is just an implementation of the classic "agent-environment loop". Each timestep, the agent chooses an **action**, and the environment returns an **observation** and a **reward**.



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 στρατηγική που απαιτείται να ακολουθήσει ο παίκτης είναι σχετικά απλή και η πολυπλοκότητας αυξάνεται γραμμικά (δηλαδή στα αρχικά frames μπορούμε με την χρήση κάποιων κόνανων να την αναπαραστήσουμε). Το παιχνίδι είναι αρκετά απλό, αποτελείται απο έναν λαμβύρινό σαν το Pac-Mac (στο οποίο αναφέρομαι συχνά γιατί 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 όταν ακουμπήσει ουσιαστικά τις ληστεύει. Αφού ληστεύσει μια τράπεζα αυτή εμφανίζει το ποσό που πήρε ο παίκτης από αυτής (reward) και μετά από κάποιο χρόνο στην θέση της εμφανίζεται ένα αστυνομικό όχημα, το οποίο προσπαθεί να πιάσει το gateway 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 περιπολικού. Η προσεγγιση μου ήταν κάθε pixel να εκφράζει ένα κελί στο Κυψελιδωτό Αυτοματο μου και κάθε κελί να έχει μια τιμή χρησιμότητας (utility): όσο πιο κοντά σε μια τράπεζα είναι τόσο υψηλότερη θα ήταν η χρησιμότητα του, ενώ όσο πιο κοντά σε ένα περιπολικό ήταν τόσο χαμηλότερη.

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
189 ~

```

Αρχικά δημιουργώ έναν πίνακα ίσο με τις διαστάσεις της οθόνης του Atari 2600 (210X160 pixel) (γραμμή 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 κελιού, πχ οι τράζες έχουν rgb(214, 214, 214), το περιπολικό rgb(24,26,167) κτλ). Εδώ συνάντησα το εξή πρόβλημα: όταν το οχήμα ληστεψει μια τραπεζα αυτή εμφανίζει τον αριθμό των χρημάτων που είχε, αυτό ο αριθμός έχει ίδιο χρώμα με την τράπεζα, πρέπει με κάποιο τρόπο να ξεχωρίζουμε τις τραπεζες από αυτούς τους αριθμούς, η συνάρτηση clearBankNoise (γραμμή 164) αντιμετωπίζει αυτό το πρόβλημα σβήνοντας τους αριθμούς.

Στην συνέχεια πρέπει με κάποιο τρόπο-κανόνα να εξαπλώσουμε την χρησιμότητα και στα υπολοιπα κελιά. Αυτό το υλοποιούμε μέσω του For (γραμμή 166) – το Bank_Range εκφράζει το πόσο μακριά θα εξαπλωθεί η χρησιμότητα από μία τραπεζα (ιδανικά θα πρέπει να είναι ίσο με το μέγεθος του χώρου μας, όμως για πρακτικού λόγους, που σχετίζονται με την ταχύτητα εκτέλεσης, το θέτω λίγο μικρότερο). Για κάθε μή-μηδενικό στοιχείο υπολογίζω τον μέσο όρο των γύρο κελιών 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 μειωμένη κατά έναν παραγοντα UTILITYDECAY.

Σημείωση 1: η παραπάνω τεχνική δεν είναι βέλτιστη ως προς τον υπολογιστικό χρόνο, δεδομένου ότι περιλαμβάνει παρα πολλούς περιττούς υπολογισμούς, έχω σκεφτεί και καλύτερο με πίνακες που θα κάνουν roll προς κάθε κατεύθυνση και θα υπολογίζουμε μόνο τα οριακά κελιά που είναι ελεύθερα, αλλά θα σας την αναλύσω αφού την δοκιμάσω και όντως έχουμε θετικά αποτελέσματα.

Σημείωση 2: η παραπάνω τεχνική μου φαίνεται κατα κάποιο 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 αναγνωρίζουμε 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 ...

Αρνητική Χρησιμότητα – Police Utility

Η αρνητική χρησιμότητα υπολογίζεται ακριβώς με την ίδια λογική με την μόνη διαφορά να βρίσκεται στο τέλος όπου ο συνολικός πίνακας πολλαπλασιάζεται με μείον ένα.

```
96 def updateUtilityPoliceVeh(self):~
97     if VERBOSE and INFO:~
98         print(['INFO]: updating utility for police veh')~
99     ~
100     rtn = np.zeros([self.rows, self.columns])~
101     ~
102     rtn[self.currentPoliceLocation[:,0]] = 1.0~
103     ~
104     for i in range(POLICE_RANGE):~
105         (x,y) = np.where(rtn>0.)~
106         ~
107         for x2,y2 in zip(x,y):~
108             ~
109             ego = rtn[x2][y2]~
110             down = rtn[x2+1][y2]~
111             up = rtn[x2-1][y2]~
112             right = rtn[x2][y2+1]~
113             left = rtn[x2][y2-1]~
114             ~
115             avgMatrix = np.array([ego,down,left,right,up])~
116             avgWeights = avgMatrix != 0~
117             # utilityValue = np.average(avgMatrix, weights=avgWeights)*POLICE_DECAY*(((POLICE_RANGE-i)/POLICE_RANGE)**2)~
118             utilityValue = np.average(avgMatrix, weights=avgWeights)*POLICE_DECAY~
119             # utilityValue = ego*UTILITYDECAY~
120             ~
121             rtn[x2][y2+1] = utilityValue if (rtn[x2][y2+1]==0. and self.freeSpace[x2][y2+1][0] ) else rtn[x2][y2+1]~
122             rtn[x2][y2-1] = utilityValue if (rtn[x2][y2-1]==0. and self.freeSpace[x2][y2-1][0] ) else rtn[x2][y2-1]~
123             rtn[x2-1][y2] = utilityValue if (rtn[x2-1][y2]==0. and self.freeSpace[x2-1][y2][0] ) else rtn[x2-1][y2]~
124             rtn[x2+1][y2] = utilityValue if (rtn[x2+1][y2]==0. and self.freeSpace[x2+1][y2][0] ) else rtn[x2+1][y2]~
125             ~
126             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 οχήματος 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.

Ανανέωση Utility

Το ερώτημα είναι τώρα κάθε πότε θα πρέπει να εφαρμόζουμε την υπολογιστικά κοστοβόρα διαδικασία ανανέωσης 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 τοσσοσ συχνά), απ την αλλη η αρνητική χρησιμότητα ανανεώνεται συχνότερα αφού πρέπει να ξεουμε ποσο κοντά μας είναι 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~
~
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")~
```