שם בית הספר: בסמת החדש

שם העבודה: Rubik's Crasher

שם התלמיד: רועי אביעד

ת.ז: 324890995

שם המנחה: סבטלנה דודקין

שם החלופה: למידת מכונה

תאריך ההגשה: 27.5.2022

תוכן עניינים

מבוא

הרקע לפרויקט

מטרת הפרויקט היא אימון המחשב עד כדי יכולתו לקבל קובייה הונגרית מבולגנת, ולהצליח בכוחות עצמו להביא אותה למצב פתור. את האימון אבצע באמצעות אלגוריתם למידה מחיזוקים, במסגרתו תקבל המכונה חיזוקים חיוביים בעבור צעדים טובים שהיא עושה, ושליליים בעבור טעויות. בדרך זו תלמד המכונה לפעול בצורה הנכונה ביותר על מנת להגיע לפתרון.

קהל היעד המרכזי של הפרויקט הוא אנשים אשר רוצים ללמוד לפתור קובייה הונגרית ואת הדרכים הטובות ביותר לעשות זאת. קהל היעד המשני של הפרויקט הוא אנשים אשר קיבלו קובייה הונגרית במתנה, בלגנו אותה, וכעת אינם יכולים להחזירה למצבה המקורי משום שאינם יודעים לפתור את הקובייה.

אופן הפעולה של המכונה בעת הלמידה יהיה כאמור דרך למידה מחיזוקים, במסגרתה תקבל אלפי קוביות ותלמד את הדרכים הטובות ביותר לפתור אותן.

אופן הפעולה של המכונה בעת העבודה עמה יהיה קלט של מצב של קובייה הונגרית מסוימת (בדרך ידנית או בדרך ויזואלית באמצעות GUI). המכונה תהפוך את הקלט למידע עמו יודעת לעבוד, ותשתמש במודל המוכן לאחר האימון על מנת לפתור את הקובייה. לאחר כל שלב בפתרון תזכור המכונה את הצעד שביצעה, ובסוף הפתרון תדפיס למשתמש את הדרך המלאה לפתרון – כך שיוכל לפתור את הקובייה בעצמו.

בחרתי בפרויקט זה של פתרון קובייה הונגרית, משום שסקרן אותי כיצד ניתן לפתור קובייה הונגרית בצורה מהירה ובמספר שלבים מועט. אני יודע מספר אלגוריתמים באמצעותם ניתן לפתור את הקובייה, אך אלגוריתמים אלו מסובכים מאוד, ונדרשים צעדים רבים (מעל מאה בדרך כלל) על מנת לפתור את הקובייה. לשם השוואה, הזמן הלוקח לי לפתור קובייה הונגרית בעצמי הוא בין 40 שניות לדקה וחצי.

תהליך המחקר

לפני תחילת העבודה על הפרויקט, חיפשתי פתרונות אחרים לבעיה הקיימים בשוק. ישנם מספר אתרים המציעים פתרון מהיר לקובייה ההונגרית (כגון rubiks-cube-solver.com), אך אתרים שכאלו מתבססים במרבית המקרים על בסיס נתונים עצום המכיל מספר רב של מצבים של הקובייה, או על אלגוריתמים ארוכים לפתירת הקובייה, שנמצאו על ידי האדם.

את הפרויקט רציתי לבצע ללא כל בסיס נתונים השומר מצבי קובייה, וללא אלגוריתמים לפתירת הקובייה מעשי ידי אדם. בחרתי להשתמש באלגוריתם מסוג למידה מחיזוקים, משום שהבעיה של הקובייה ההונגרית לא ניתנת לפתרון באמצעים בדרך של מונחית (משום שאין בסיס נתונים להשוואה ולמידה ממנו), ומשום שבמהלך פתירת הקובייה המכונה צריכה ללמוד לבצע החלטות – דבר הניתן לבצע רק באמצעות למידה מחיזוקים.

מכיוון שלמידה מחיזוקים אינה נמצאת בתוכנית הלימודים, היה עלי להשתמש במקורות מידע שונים על מנת להצליח לבנות את המודל. כדי ללמוד כיצד לפתח מודל של למידה מחיזוקים, ומשום שאין מדריך לפתירת קובייה הונגרית בצורה זו, השתמשתי במדריך לאימון מכונה למשחק "אטארי". למדתי כיצד להשתמש בספרייה ובאפשרויותיה וליישם אותם על הפרויקט

שלי. השתמשתי במדריך נוסף על מנת ללמוד כיצד לבנות סביבת אימון למודל של למידה מחיזוקים, ובניתי בעצמי סביבת אימון לפתירת קובייה הונגרית.

אתגרים מרכזיים

במהלך הפיתוח של הפרויקט אצטרך להתמודד עם מספר בעיות כגון:

- למידה מחיזוקים אינה חלק מתוכנית הלימודים ולכן אצטרך ללמוד את עקרונותיה לבד.
- אין פרויקטים הזהים או דומים לפרויקט שלי, ועל כן אצטרך לקבץ מידע ממספר מקורות שונים, ולהתאים אותם כך שאוכל להשתמש בהם בפרויקט שלי.

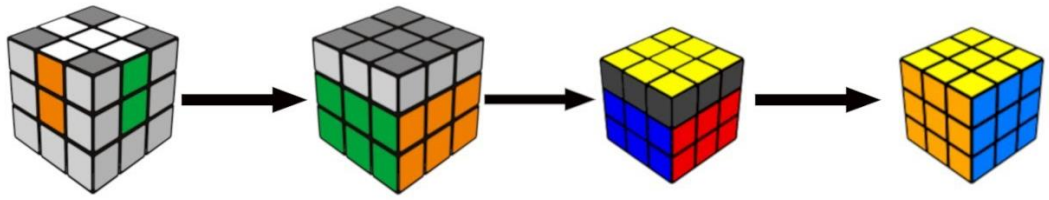
הפרויקט מתעסק בפתירה מהירה של הקובייה ההונגרית, ועונה על הצורך לפתור אותה במהירות, במספר צעדים מועט, וללא שימוש בידע המוקדם של האדם על הקובייה.

פתרונות אחרים לבעיה

ישנם מספר פתרונות אחרים של למידת מכונה לבעיה, אך הם מסחריים ועל כן לא אוכל להציג את המודל בו השתמשו.

למרות זאת, אציג את האלגוריתם המרכזי כיום לפתירת הקובייה – CFOP: האלגוריתם פותר את הקובייה באמצעות 4 שלבים:

יצירת צורת צלב בשכבה הראשונה ← פתירת 2 השכבות התחתונות ← פתירת השכבה העליונה ← השלמת השכבה הנותרת.



מבנה הפרויקט

שלב בניית סביבת האימון

משום שהמודל עובד על למידה מחיזוקים, אין מבנה נתונים עליו הוא מתבסס. לכן, בשלב זה אסביר על סביבת האימון של המודל.

סביבת האימון בנויה משתי מחלקות, שאת תיעוד הקוד שלהן אתאר במדריך למפתח:

- מחלקת Cube – מחלקה המייצגת קובייה הונגרית בגודל 2x2x2:
  מחלקה זו מחזיקה את מבנה הנתונים המייצג קובייה הונגרית:

```
self._cube['U'] = np.array([[0, 0], [0, 0]])
self._cube['D'] = np.array([[1, 1], [1, 1]])
self._cube['F'] = np.array([[2, 2], [2, 2]])
self._cube['B'] = np.array([[3, 3], [3, 3]])
self._cube['L'] = np.array([[4, 4], [4, 4]])
self._cube['R'] = np.array([[5, 5], [5, 5]])
```

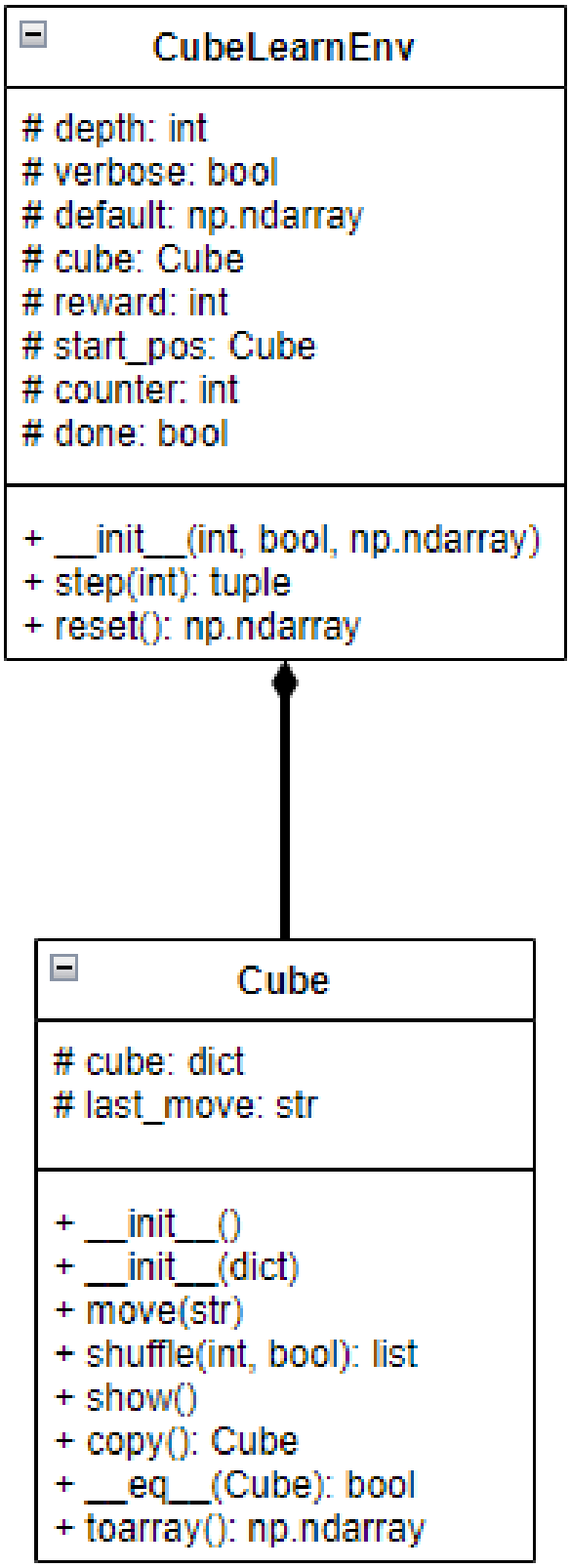
  מבנה הנתונים הוא מסוג מילון השומר מערך דו־מימדי של ספריית numpy. לכל צד בקובייה (Up, Down, Front, Back, Left, Right) המערך הדו־מימדי שומר את הצבעים הממוקמים עליו (לכל צבע מספר המייצג אותו מ0-5).

  במחלקה גם ממומשות פונקציות השולטות בסיבוב הקובייה:

  - פונקציית move – מקבלת מחרוזת המייצגת פעולה של סיבוב הקובייה, ומסובבת את הקובייה בהתאם לפעולה שקיבלה.
  - פונקציית shuffle – פונקציה המקבלת מספר ומערבבת את הקובייה באופן רנדומלי בהתאם למספר הסיבובים שקיבלה.
  - פונקציית show – מדפיסה את המצב הנוכחי של הקובייה באמצעות ספריית matplotlib.

- מחלקת CubeLearnEnv – סביבת האימון עליה עובד המודל:
  מחלקה זו מנהלת קובייה בודדת (עצם מסוג Cube), וממשת את הפונקציות הדרושות לאימון של מודל DQNAgent של ספריית keras-rl המשמשת לצורך למידה מחיזוקים.
  הפונקציות הללו הן:
  - פונקציית step – פונקציה המטפלת בצעד אחד של המודל. היא מקבלת מספר בתחום 0-11 המייצג את הפעולה לביצוע. בהתאם למצב הקובייה ולפעולה שקיבלה, היא מחליטה על החיזוק המתאים שעליה לתת למודל. לבסוף היא מחזירה למודל את המצב של הקובייה, את החיזוק שמקבל בעבור הצעד שביצע, וערך בוליאני הקובע האם הקובייה נפתרה.
  - פונקציית reset – פונקציה המאתחלת את הסביבה ויוצרת לה קובייה חדשה מבולגנת, ומחזירה את המצב החדש כדי שהמודל ינסה לפתור אותה.

תרשים UML של המחלקות בסביבת האימון:



שלב בנייה ואימון המודל

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten1 (Flatten)          (None, 24)                0

 dense1 (Dense)              (None, 72)                1800

 dense2 (Dense)              (None, 60)                4380

 dense3 (Dense)              (None, 48)                2928

 dense4 (Dense)              (None, 36)                1764

 dense5 (Dense)              (None, 24)                888

 dense6 (Dense)              (None, 12)                300

=================================================================
Total params: 12,060
Trainable params: 12,060
Non-trainable params: 0
```

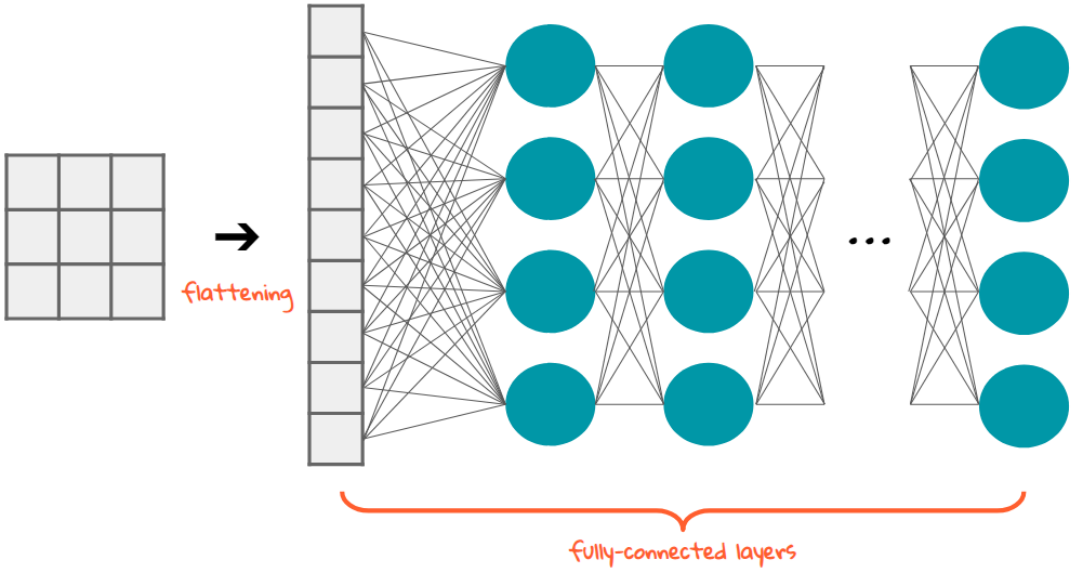
כפי שניתן לראות, המודל הוא מודל סדרתי (Sequential) המאפשר שימוש במספר שכבות נוירונים הפועלות אחת אחרי השנייה, והוא בנוי מ7 שכבות בסך הכול:

בתחילה ישנה שכבה Flatten שתפקידה לקחת את הנתונים של הקובייה ולהפוך אותם למערך חד־מימדי בגודל 24 (קוביית 2x2x2 מכילה בסך הכול 24 חלקים חיצוניים שונים).

לאחר מכן ישנן מספר שכבות מסוג Dense שהן שכבות צפופות בהן כל נוירון מחובר לשאר הנוירונים. לשכבות אלה פונקציית אקטיבציה מסוג ReLU (f(x)=max(0, x)) המאפשרת אימון של רשתות עמוקות ומורכבות יותר.

לבסוף ישנה שכבת הפלט, המחזירה פלט בגודל 12, כמספר הפעולות האפשריות לבצע על הקובייה.

תרשים מוקטן של השכבות:



כפי שניתן לראות – שכבת Flatten "המשטחת" את המידע, ולאחר מכן שכבות Dense בהן הנוירונים מחוברים באופן מלא, עד שכבת הפלט. התרשים בגודל שונה מאשר השכבות האמתיות (קלט 24, פלט 12).

על מודל השכבות הנ"ל פועל סוכן DQN (Deep Q Network), בעל מדיניות Greedy. סוכן זה מאפשר את הלמידה מחיזוקים, ותפקידו להחליט בכל צעד על הפעולה המתאימה ביותר לבצע. המדיניות Greedy אומרת לסוכן לפעול במטרה לקבל רווח גבוה ככל הניתן. בדרך זו הסוכן לומד בכל צעד לבחור את הפעולה שתביא אותו לרווח הגבוה ביותר.

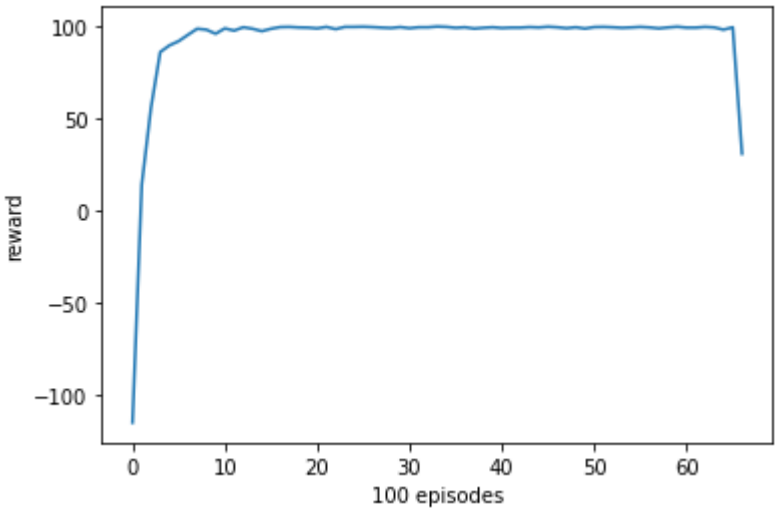
בשלב הבא נקמפל את הסוכן על המודל שיצרנו, ובעזרת אלגוריתם האופטימיזציה Adam.

את האימון בחרתי לבצע בשלבים לפני מרחק הקובייה מהפתרון - מרחק הקובייה מהפתרון הוא מספר הסיבובים המינימלי שיש לבצע על מנת להביא את הקובייה למצב ההתחלתי הפתור.

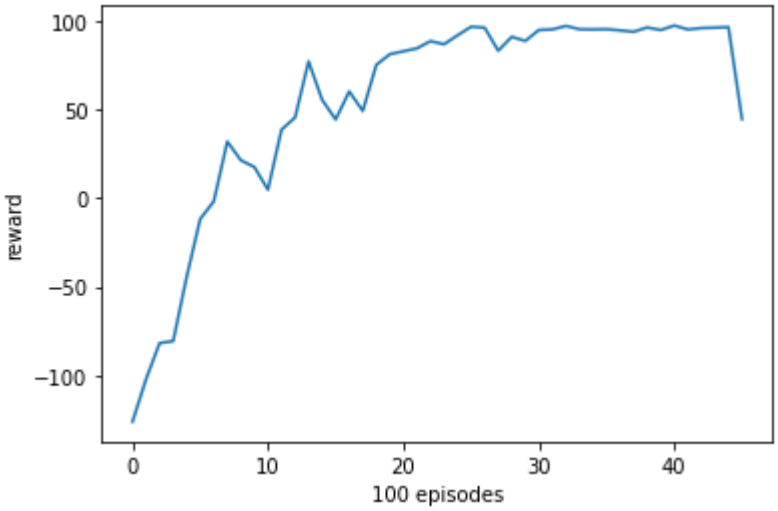
את המודל אימנתי על שלושת השלבים הראשונים (מרחק 1, 2 ו-3) מקובייה פתורה, כאשר בכל אחד מהם הוגרלו למודל אלפי קוביות הונגריות מבולגנות, והיה עליו ללמוד כיצד לפתור אותן בעזרת סביבת האימון ועקרונות הלמידה מחיזוקים.

תוצאות שלב האימון

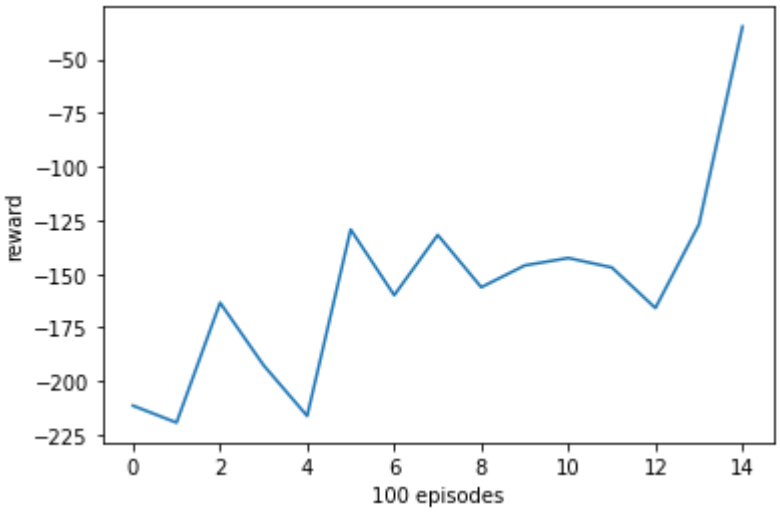
- גרף שלב האימון בעבור מרחק 1 מקובייה פתורה:



- גרף שלב האימון בעבור מרחק 2 מקובייה פתורה:



- גרף שלב האימון בעבור מרחק 3 מקובייה פתורה:



דיוק ושגיאה

את הדיוק והשגיאה חישבתי באמצעות הגרלת 1000 קוביות לכל אחת מהרמות וחישוב השגיאה האבסולוטית הממוצעת, ואחוז ההצלחה בפתרון.

- בעבור הרמה הראשונה:
  - דיוק: 100%
  - שגיאה אבסולוטית ממוצעת: 0.0
- בעבור הרמה השנייה:
  - דיוק: 98.4%
  - שגיאה אבסולוטית ממוצעת: 5.22
- בעבור הרמה השלישית:
  - דיוק: 57.2%
  - שגיאה אבסולוטית ממוצעת: 95.28

בשלב זה החלטתי לבדוק האם המודל באמת הצליח ללמוד, והאם יצליח לפתור קוביות גם בעבור הרמה הרביעית. לאחר הבדיקה קיבלתי שהדיוק שלו בעבור קוביות מהרמה הרביעית הוא 26.1%. כלומר, המודל אכן למד ומצליח לפתור קוביות שלא נתקל בהן בעבר.

משתני האימון ושינוים לקבלת תוצאות טובות יותר

בלמידה מחיזוקים, מלבד קצב הלמידה, מספר החזרות וגודל רשת הנוירונים, יש להתאים גם את סביבת האימון עצמה ללוגיקה הטובה ביותר בעבור המודל.

בתחילה, השתמשתי ברשת נוירונים בעלת 3 שכבות, ובעבור כל רמה נתתי למודל 10,000 צעדים. לוגיקת סביבת האימון הייתה:

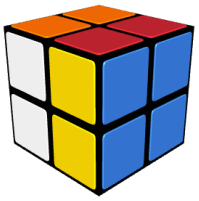
- שמירת הצעדים הדרושים לפתירת הקובייה, ובעבור צעד נכון שמבצע המודל לתת לו 10 נקודות.
- בעבור טעות להוריד לו 30 נקודות.
- עם פתרון נכון של הקובייה לתת לו 50 נקודות.

ראיתי שהמודל מצליח לאמן טוב קוביות מהרמה הראשונה, אך בעבור קוביות ברמות גבוהות יותר הוא נכשל. החלטתי להגדיל את רשת הנוירונים ולהפוך אותה ל7 השכבות הקיימות כעת, ושיניתי את מספר החזרות לפי הנוסחה (n**2)*10,000 כאשר n הוא הרמה הנוכחית.

לאחר השינוי הנ"ל ראיתי כי התוצאות אכן משתפרות, אך כי התוצאות ברמות 2 ומעלה עדיין לא מאוד טובות. בשלב זה הבנתי שבלוגיקת סביבת האימון ישנה בעיה:

ישנן מספר דרכים שונות לפתור 2 קוביות הונגריות זהות, וגם אם המודל יעשה את הצעדים הנכונים, הסביבה תוריד לו נקודות אם הוא אינו בוחר בדרך מסוימת עליה היא יודעת.

לדוגמה, את הקובייה הזו:



ניתן לפתור כך: R' L'

וגם כך: L' R'

לאחר שהבנתי זאת, החלטתי לשנות את לוגיקת הסביבה:

- בעבור הצלחה בפתרון יקבל המודל 100 נקודות – הגדלתי את המספר כדי שהמודל ידע שזו המטרה שלו בוודאות.
- הסביבה נותנת למודל n צעדים כדי לפתור קובייה מרמה n. אם לאחר n הצעדים הצליח המודל לפתור, יקבל את 100 הנקודות. אם נכשל בפתרון, ירדו לו 10 נקודות.
- בכל צעד תרד למודל נקודה אחת, כדי להנחות אותו לפתור את הקובייה במספר צעדים מועט ככל הניתן, ולא לעשות צעדים מיותרים.

לאחר השינוי הזה, התוצאות השתפרו בהרבה, והגעתי כמעט לדיוק הסופי. כאשר הגדלתי שוב את מספר הצעדים ומספר החזרות שהמודל יבצע, ראיתי שיפור בתוצאות, ואני סבור כי אם אתן למודל עוד זמן וחזרות אימון יצליח לפתור את הקוביות בדיוק גבוה מאוד.

שלב היישום

השימוש במודל נעשה על ידי יצירה של קובייה פתורה, וקליטה של רצף פעולות מהמשתמש. לאחר קליטת רצף הפעולות, תבולגן הקובייה ברצף זה בדיוק, ולאחר מכן תיפתר על ידי המודל. את התוצאות יראה המשתמש באמצעות פונקציית show של המחלקה Cube, שתקרא בכל צעד שמבצע המודל, וכך יראה המשתמש את הדרך המלאה לפתרונה של הקובייה.

את הקוביות החזותיות יוצרת המחלקה Cube באמצעות שימוש באפשרות תלת־המימד של הספרייה matplotlib. הפונקציה יוצרת 2 גרפים, שבכל אחד מהם 12 מלבנים בצבעים שונים, כאשר כל אחד מהגרפים מראה צד אחר של הקובייה – ניתן לראות רק 3 פאות שלה במקביל.

מדריך למפתח

קישור לקוד המלא

https://drive.google.com/file/d/1yEQfTw4UdVnIt0OPKI_a1Qgp41r16Ogi/view?usp=sharing

תא קוד 1 (ייבוא ספריות)

```
import numpy as np
```

ייבוא של ספריית numpy המשתמשת לביצוע פעולות מתמטיות ואלגברה ליניארית. בספרייה זו גם מספר פונקציות בהן השתמשתי לסיבוב הקובייה.

```
import random
```

ספרייה בה אשתמש לטובת ערבוב הקובייה באופן רנדומלי.

```
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
import mpl_toolkits.mplot3d.art3d as art3d
```

ייבוא אובייקטים של ספריית matplotlib. Art3d מאפשר עבודה עם גרפים בתלת־מימד, וRectangle מאפשר יצירת מלבנים, בהם אשתמש לטובת ציור הצבעים בקובייה.

```
from keras.models import Sequential
from keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import Adam
```

ייבוא אובייקטים של ספריית keras ליצירת המודל הסדרתי בו יש שכבת שיטוח ושכבות צפופות, אשר מקומפל באמצעות אלגוריתם האופטימיזציה Adam.

```
from sklearn.metrics import mean_absolute_error
```

פונקציה המחשבת את השגיאה האבסולוטית הממוצעת.

```
from rl.agents.dqn import DQNAgent
from rl.policy import EpsGreedyQPolicy
import rl.policy
from rl.memory import SequentialMemory
import rl.callbacks
```

ייבוא של אובייקטים מספריית keras-rl – ניתן לראות כאן את הייבוא של אובייקט הסוכן ושל המדיניות Greedy עליהם הסברתי.

```
ACTIONS = ["U", "U'", "D", "D'", "L", "L'", "R", "R'", "F", "F'", "B",
"B'"]
```

זוהי רשימה השומרת את כל הפעולות האפשריות על הקובייה. באמצעות רשימה זו מתרגמת סביבת האימון את מספר הפעולה לשמה.

תא קוד 2 (מחלקת Cube)

במחלקה זו קוד רב ולכן לא אתעד כאן את כולו, אלא רק את החלקים החשובים בו:

```
def __init__(self, new_cube: dict=None)
```

הפונקציה הבונה של המחלקה. אם הפרמטר new_cube מועבר, האובייקט ישכפל את הקובייה שהועברה לו. אם הפרמטר לא הועבר, האובייקט ייווצר כקובייה פתורה.

פונקציית move המבצעת סיבוב אחד של הקובייה:

```
self._last_move = action
```

שמירת המהלך שנשלח לקובייה (נועד למנוע מהלך הפוך שיחזיר את הקובייה לאותו המצב).

עתה הפונקציה בודקת מה המהלך שנשלח, ומבצעת את הסיבוב של הקובייה, לדוגמה עבור הסיבוב של החלק העליון בכיוון השעון (U) תבצע:

```
if action == "U":
      self._cube['F'][0], self._cube['R'][0], self._cube['B'][0],
      self._cube['L'][0] = self._cube['R'][0].copy(),
      self._cube['B'][0].copy(), self._cube['L'][0].copy(),
      self._cube['F'][0].copy()
```

בסיבוב זה (עליון) של הקובייה, משתנות הפאות הצדדיות (F, B, L, R), וניתן לראות כאן את ההחלפה של הערכים ביניהן בצורה החכמה של פייתון (x, y = y, x).

לאחר מכן יש לסובב את החלק העליון של הקובייה פעם אחת בכיוון השעון, ולשם כך נשתמש בפונקציה rot90 של numpy המסובבת מערך דו־מימדי נגד כיוון השעון:

```
if "'" in action:
    self._cube[action[0]] = np.rot90(self._cube[action[0]])
else:
    for _ in range(3):
        self._cube[action[0]] = np.rot90(self._cube[action[0]])
```

זוהי בדיקה – אם הסיבוב הוא נגד כיוון השעון (יש גרש בפעולה) יש לסובב פעם אחת נגד כיוון השעון, ואם הסיבוב עם כיוון השעון, יש לסובב 90 מעלות 3 פעמים נגד כיוון השעון (מה שיביא לסיבוב של 90 עם כיוון השעון).

פונקציית shuffle המערבבת את הקובייה:

```
while i < length:
```

בעבור length פעמים (הפרמטר הקובע את הרמה אליה תסובב הקובייה).

```
valid_actions = [actions_map[i] for i in range(len(actions_map))
if actions_map[i] != self._last_move[0]]
```

(זוהי שורה אחת). חישוב כל הפעולות האפשריות (להוציא את הפעולה האחרונה שבוצעה).

```
action = valid_actions[random.randint(0, len(valid_actions) - 1)]
+ "'" * random.randint(0, 1)
```

חישוב סוג הפעולה בעזרת ספריית random, ולאחר מכן סיכוי של 50% האם תהיה עם או נגד כיוון השעון.

```
times = 2 if random.randint(0, 10) == 0 else 1
```

פעולות מקובלות נוספות על הקובייה הן לדוגמה U2, המבצעת שני סיבובים של החלק העליון. מסיבה זו, ישנו סיכוי של 1 ל11 שנסובב את הקובייה פעמיים, אך לא ניתן לסובב יותר מפעמיים – U3=U' ולכן זוהי כבר פעולה אחרת.

לבסוף, לאחר חישוב הפעולה שנבצע ומספר הפעמים שנבצע אותה, נקרא לפונקציה move:

```
self.move(action)
```

פונקציית show המדפיסה את המצב הנוכחי של הקובייה:

בתחילה מועתקת הקובייה, והעותק של הקובייה מסובב במספר פאותיו לשם ההצגה (לדוגמה, הפאות L, D, B אשר נמצאות בגב הקובייה נראות על ידינו כתמונת מראה ויזואלית, ועל כן יש להשתמש בפונקציה flip של numpy על מנת להראות אותן).

לאחר מכן נאתחל את הגרפים באמצעות הקריאות:

```
fig = plt.figure()
```

(יצירת תרשים matplotlib).

והוספת שני הגרפים התלת־מימדיים אליו:

```
fig.add_subplot(1, 2, i + 1, projection='3d')
```

```
for side in range(len(special_order)):
    for i in range(2):
        for j in range(2):
```

(בעבור כל אחת מ6 הפאות, ובעבור כל ריבוע בכל פאה):

```
r = Rectangle(...)
```

נוצר המלבן המתאים בצבע המתאים (שורה ארוכה).

```
axs[1 ^ (side & 1) ^ ((side & 2) >> 1)].add_patch(r)
```

המלבן נוסף לגרף הנכון (0 או 1). הפעולות הבינאריות יוצרות מצב בו בעבור פאות 034 המייצגות את הפאות האחוריות ייצא 1 (הגרף השני), ובעבור הפאות 125 ייצא 0 (הגרף הראשון).

לבסוף מחשבים את הציר בו המלבן ממוקם (xyz) ומדפיסים אותו: `plt.show()`.

תא קוד 3 (סביבת האימון)

המחלקה הזו היא הסביבה בה פועל המודל ולומד לפתור את הקובייה ההונגרית. במחלקה 2 פונקציות שחייבות להיות ממומשות על מנת שהמודל יוכל להשתמש בה:

פונקציית step הנקראת בכל צעד של המודל:

- בתחילה נזיז את הקובייה לפי הצעד שהתקבל מהמודל:

```
self._cube.move(ACTIONS[action])
```

- לאחר מכן נבדוק האם הקובייה הושלמה, ואם כן נוסיף למודל 100 נקודות ונסמן לו שסיים עם הקובייה הזו:

```
if self._cube == Cube():
    self._reward = 100
    self._done = True
```

(בקובייה ממומש אופרטור ==, ולכן ניתן להשוות לקובייה פתורה).

- כעת נבדוק האם המודל ניסה לפתור את הקובייה מספר פעמים ולא הצליח:

```
elif self._counter % self._depth == 0:
    self._reward = -10
    self._cube = self._start_pos.copy()
```

נראה כי אם המודל ניסה לפתור 3 פעמים קובייה מרמה 3, תרדנה לו 10 נקודות, והוא יחזור לנקודת ההתחלה.

- לבסוף נוריד למודל נקודה כדי שיידע לנסות לפתור את הקובייה כמה שיותר מהר (באיבוד נקודות נמוך ככל האפשר), ונחזיר לו את המידע שלו הוא זקוק כדי ללמוד:

```
return np.reshape(list(self._cube.copy().cube.values()), (24,)),
self._reward, self._done, {}
```

הסביבה מחזירה לו את המצב החדש של הקובייה (24 החלקים בה), את הרווח שהתקבל או נלקח בעבור הצעד שביצע, והאם הוא סיים.

פונקציית reset המאתחלת את הסביבה כאשר המודל מסיים עם קובייה מסוימת:

שני החלקים המרכזיים של הפונקציה הם:

- יצירת קובייה חדשה:

```
self._cube = Cube()
```

- ערבוב הקובייה החדשה:

```
self._cube.shuffle(self._depth, verbose=self._verbose)
```

תא קוד 4 (יצירת המודל)

```
model = Sequential()
model.add(Flatten(input_shape=(1, 24), name="flatten1"))
model.add(Dense(72, activation='relu', name="dense1"))
model.add(Dense(60, activation='relu', name="dense2"))
model.add(Dense(48, activation='relu', name="dense3"))
model.add(Dense(36, activation='relu', name="dense4"))
model.add(Dense(24, activation='relu', name="dense5"))
model.add(Dense(12, activation='linear', name="dense6"))
print(model.summary())
```

על שכבות המודל הסברתי מעלה, ניתן לראות בחלק זה את הוספת כל השכבות ולבסוף את הדפסת סיכום המודל (את הפלט ניתן לראות מעלה).

תא קוד 5 (יצירת הסוכן)

```
dqn = DQNAgent(model=model, nb_actions=nb_actions, memory=memory,
nb_steps_warmup=1000, \
    target_model_update=1e-2, policy=policy)
dqn.compile(Adam(learning_rate=1e-3), metrics=['accuracy'])
```

בתא זה ניצור את סוכן הלמידה (Deep Q Network Agent) בעזרת מדיניות Greedy, ונקמפל אותו באמצעות אלגוריתם האופטימיזציה Adam.

לאחר מכן נבדוק האם קובץ המשקולות weights_v2.h5 קיים, ובמידה וכן נטען את המשקולות למודל:

```
if os.path.exists("weights_v2.h5"):
    dqn.load_weights("weights_v2.h5")
```

תא קוד 6 (אימון המודל)

בעבור כל אחת מהרמות עליהן נרצה לאמן את המודל:

```
for i in range(1, depth + 1):
```

ניצור את סביבת האימון של המודל:

```
    env = CubeLearnEnv(i)
```

נאמן את המודל באמצעות:

```
    dqn.fit(env, nb_steps=steps, visualize=False, verbose=2,
    nb_max_episode_steps=i ** 2 * 20)
```

ונשמור את היסטוריית האימון (כל הרווחים השקולים של כל episode באימון):

```
    json.dump(rewards, h_file)
```

תא קוד 7 (שמירת המודל)

נשמור את המודל כדי שלא נצטרך לאמן אותו מההתחלה בכל פעם:

```
dqn.save_weights("weights_v2.h5", overwrite=True)
```

תא קוד 8 (הדפסת גרף רווחים)

נקרא מהקובץ את רשימת הרווחים:

```
rewards = json.load(h_file)
```

ונדפיס למסך את הגרף:

```
rewards = [(sum(rewards[i:i+n]) / n) for i in range(0, len(rewards),
n)]
```

הפלט של קוד זה נמצא מעלה בתוצאות האימון.

תא קוד 9 (הדפסת דיוק ושגיאה)

נבצע בדיקה על מודל הלמידה מחיזוקים:

```
history = dqn.test(test_env, nb_episodes=n, visualize=False, verbose=0,
nb_max_episode_steps=depth * 10)
rewards = history.history["episode_reward"]
accuracy = [(reward > 0) for reward in rewards].count(True) * (100 / n)
```

הפונקציה dqn.test מבצעת ריצה על 1000 קוביות שונות, ושומרת את תוצאות הריצה למשתנה history. לאחר מכן נחלץ את הרווחים מההיסטוריה של הבדיקה, ונחשב את דיוק התוצאות.

לאחר מכן נחשב את השגיאה:

```
mse = np.array(rewards)
print("mae: %.2f" % mean_absolute_error(mse, np.full(mse.size, 100)))
```

אנו מחשבים את השגיאה האבסולוטית הממוצעת לפי רשימה מלאה בערכי 100 (הערך המקסימלי האפשרי של הרווח המתקבל במידה ולא הייתה טעות בפתירה). כך נחשב את ההפסד השקול של המודל.

תא קוד 10 (הכנסת קובייה)

קוד זה מבקש מהמשתמש תחילה את הרצף בו ירצה לערבב את הקובייה:

```
path = input("Enter shuffle sequence: ")
```

ולאחר מכן מערבב את הקובייה ברצף זה:

```
for action in path.split(" "):
    cube.move(action)
```

ניצור מחלקה חדשה (EpisodeLogger) היורשת ממחלקת rl.callbacks.Callback. מחלקה זו משמשת להצגת התוצאות של הבדיקה. באמצעות dqn.test נבצע בדיקה על episode אחד (קובייה אחת). נכניס כפרמטר את הEpisodeLogger שיצרנו, השומר את מצבי הקובייה השונים בדרך לפתרון.

דוגמה להרצה אתן במדריך למשתמש.

מדריך למשתמש

דרישות התקנה

- ספריית numpy
  התקנה באמצעות: pip install numpy
- ספריית matplotlib
  התקנה באמצעות: pip install matplotlib
- ספריית tensorflow
  התקנה באמצעות: pip install tensorflow
- ספריית keras (ככל הנראה כבר נמצאת אם tensorflow מותקנת)
  התקנה באמצעות: pip install keras
- ספריית sklearn
  התקנה באמצעות: pip install sklearn
- ספריית keras-rl
  התקנה באמצעות pip install keras-rl2

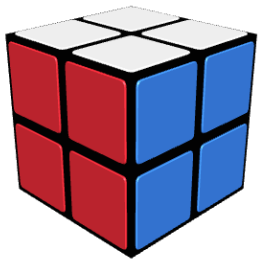
לאחר ההתקנות, הקובץ היחיד שנדרש הוא קובץ המחברת של פייתון הנקרא **Rubik's_Crasher_RL_v2.ipynb**. כדי שממשק המשתמש יעבוד (תא הקוד האחרון), יש להריץ תחילה את כל התאים הקודמים. אם ישנו קובץ בשם weights_v2.h5 המכיל את משקולות האימון של המודל המאומן, אין צורך לאמן את המודל שוב (תא קוד 6).

אם קובץ המשקולות לא קיים, יש לאמן מחדש את המודל באמצעות הרצת תא קוד 6.

לבסוף יש להריץ את תא קוד 10, ולהכניס רצף פעולות לערבוב קובייה, כדי שהמודל יוכל לפתור אותה.

הוראות הרצה והפעלה של התוכנה

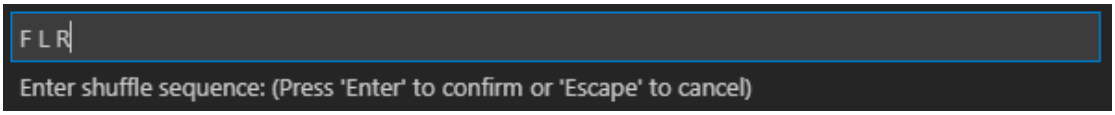
- יש להחזיק קובייה הונגרית בגודל 2x2x2 לצד המחשב, כאשר היא במצב פתור והצד הפונה לכיוון המשתמש הוא פינת לבן – כחול – אדום (לבן מעלה):

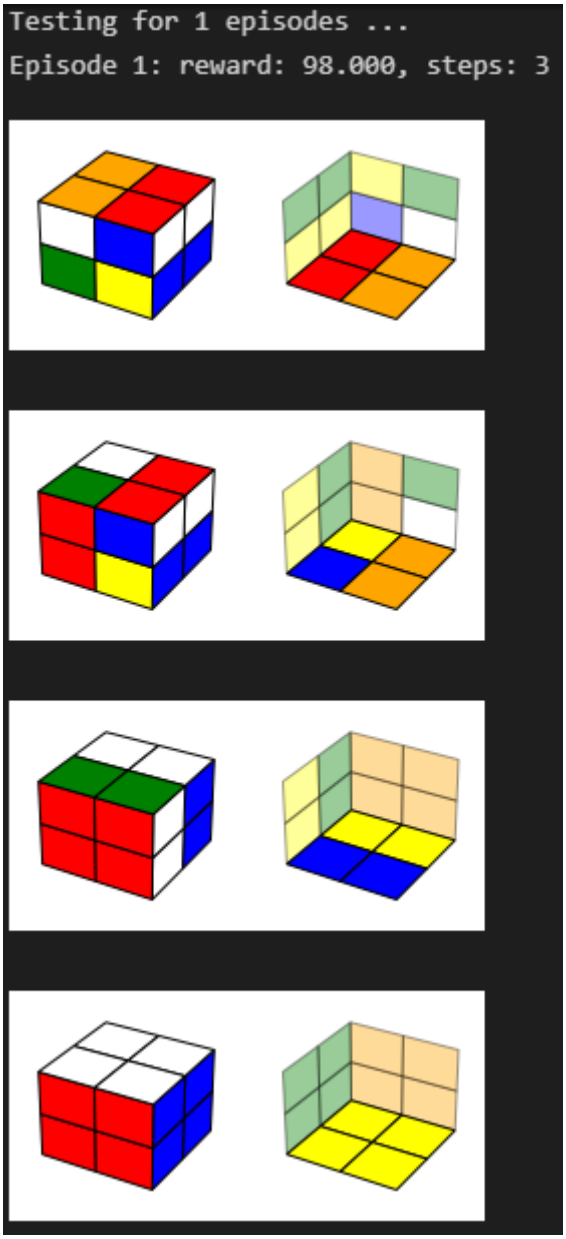


- כעת יש לסובב את הקובייה לפי הכללים הבאים:
  - U – חלק עליון, D – חלק תחתון, F – חלק קדמי, B – חלק אחורי, L – חלק שמאלי, R – חלק ימני.
  - יש להוסיף גרש (') לאחר האות כדי להורות על סיבוב כנגד כיוון השעון.
  - בין כל פעולה יש להוסיף רווח.
- לבסוף המשתמש יכניס את הרצף שבו ערבב לתוכנה, והמודל ידפיס לו את השלבים לפתרון.

דוגמה להרצת המודל

בעבור הקלט "F L R":





כפי שניתן לראות, המשתמש מקבל את כל השלבים מהקובייה המעורבבת שהכניס (אמורה להיראות בדיוק כמו הגרף הראשון), עד לקובייה הפתורה שבסוף.

רפלקציה

העבודה על הפרויקט בעבורי הייתה מאוד מהנה, ולמדתי רבות ממנה. אני מאוד מתחבר לנושא של למידת מכונה משום שהוא מעניין מאוד בעבורי, ומשום שזה תמיד מדהים אותי לראות כיצד המחשב מצליח לפתור פעולות מורכבות מאוד במהירות יחסית.

זו הסיבה שבשלה בחרתי שבפרויקט הסיום אתמקד בפתרון הקובייה ההונגרית, תהליך מורכב ומסובך, שרציתי לראות כיצד מודל של למידת מכונה יצליח להתמודד עמו.

אני סבור כי במהלך העבודה צברתי ידע רב בתחום למידת המכונה, ובמיוחד על עולם הלמידה מחיזוקים בו בחרתי להתמקד. אני חושב שבעתיד, בצבא, בעבודה ואף בלימודים אם אחליט להמשיך בתחום הנדסת התוכנה, הידע שצברתי במהלך העבודה יתרום לי מאוד. לדעתי, העובדה שאגיע עם בסיס מוכן בעולם זה למקומות הללו תסייע לי מאוד.

האתגר המרכזי שעמד בפני בכתיבת הפרויקט היה החקר אודות תחום הלמידה מחיזוקים, עליו לא למדנו במהלך השיעורים והוא אינו נמצא בתוכנית הלימודים. תחום זה עניין אותי במיוחד שכן הוא פותח אפשרויות רבות שאינן קיימות בתחומים האחרים של למידת מכונה (כמו למידה מונחית). היה קשה במיוחד לחקור אודות הספרייה keras-rl המתמקדת בלמידה מחיזוקים, שכן התיעוד שלה דל יחסית לספריות אחרות. לבסוף מצאתי מספר מדריכים טובים שהסבירו בצורה נאותה את אופן השימוש בספריה.

בנוסף, אתגר אותי מאוד למצוא פתרונות לבעיות שנקלעתי אליהן במהלך העבודה. בעיות כמו דיוק נמוך של המודל מאוד קשות לפתרון בתחום הלמידה מחיזוקים, שכן גם סביבת האימון נבנתה על ידי. אני שמח כי הצלחתי לקבל תוצאות טובות מאוד ומודל מדויק יחסית, אך מצר על כך שלא הצלחתי להביא את המודל לפתרון קוביות ברמות גבוהות יותר.

כעת, לסיכום העבודה על הפרויקט, אני יכול לומר כי אני שמח על שבחרתי בפרויקט זה ובמיוחד על שבחרתי בתחום הלמידה מחיזוקים שמאוד עניין אותי. כשארצה בעתיד לשפר את המודל, אתחיל משיפור העבודה של המודל עם רמות גבוהות כך שיידע לפתור קוביות מסובכות יותר וקשות יותר.

ביבליוגרפיה

Chapman, J., & Lechner, M. (2020, May 23). *Keras documentation: Deep Q-Learning for Atari Breakout*. Keras.Io. Retrieved May 22, 2022, from https://keras.io/examples/rl/deep_q_network_breakout/

Dalton, M. (2020, August 11). *Solving a Rubik's Cube with Reinforcement Learning (Part 1)*. Medium. Retrieved May 22, 2022, from https://medium.com/analytics-vidhya/solving-a-rubiks-cube-with-reinforcement-learning-part-1-4f0405dd07f2

[Python] Easy Reinforcement Learning (DQN) with Keras-RL. (n.d.). Linuxtut.Com. Retrieved May 22, 2022, from https://www.linuxtut.com/en/e63ade6f21766c7c2393/

Tonia, L. (2022, February 2). *Building a Reinforcement Learning Environment using OpenAI Gym*. Engineering Education (EngEd) Program | Section. Retrieved May 22, 2022, from https://www.section.io/engineering-education/building-a-reinforcement-learning-environment-using-openai-gym/