

תיק פרויקט – Snake DQN

שם התלמיד: יואב גלעד

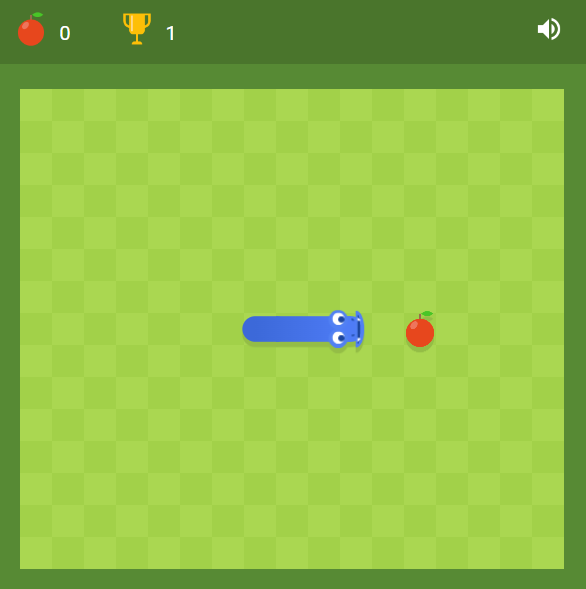
ת.ז. התלמיד: 216875922

בית ספר: תיכון המושבה

שם המנחה: ענבר מור

שם החלופה: למידת מכונה

תאירך הגשה: ?



תוכן עניינים

[תיק פרויקט – Snake DQN 1](#_Toc197430002)

[תוכן עניינים 2](#_Toc197430003)

[מבוא 3](#_Toc197430004)

[דגשים חשובים לקריאת תיק הפרויקט 4](#_Toc197430005)

[רקע תיאורטי 4](#_Toc197430006)

[מבנה הפרויקט 6](#_Toc197430007)

[ייצור הנתונים 7](#_Toc197430008)

[שדרוגים לסביבה Snake 8](#_Toc197430009)

[תיוג הנתונים 10](#_Toc197430010)

[בניית ואימון המודל 11](#_Toc197430011)

[בניית המודל 12](#_Toc197430012)

[היפר-פרמטרים וכוונון 14](#_Toc197430013)

[תוצאות אימון המודל בדוגמה 16](#_Toc197430014)

[ייעול אימונים 19](#_Toc197430015)

[יישום 24](#_Toc197430016)

[מדריך למפתח 25](#_Toc197430017)

[environments.py 26](#_Toc197430018)

[agents.py 34](#_Toc197430019)

[evaluation.py 40](#_Toc197430020)

[logs.py 44](#_Toc197430021)

[graphs.py 45](#_Toc197430022)

[emails.py 47](#_Toc197430023)

[mytime.py 48](#_Toc197430024)

[training.py 49](#_Toc197430025)

[testing.py 55](#_Toc197430026)

[view game.py 57](#_Toc197430027)

[מדריך למשתמש 58](#_Toc197430028)

[רפלקציה 59](#_Toc197430029)

[ביבליוגרפיה 60](#_Toc197430030)

[נספחים 61](#_Toc197430031)

מבוא

Reinforcement Learning (RL) הוא ענף בלמידת מכונה המתמקד בייעול התנהגות וקבלת החלטות בסביבה מסוימת, בפרויקט שלי במשחק Snake. היעד הוא להגיע למדיניות (policy) המתארת התנהגות אופטימלית בסביבה. הפרויקט כולל תשתית לאימון סוכן RL באלגוריתם DQN, ואת המשחק Snake כסביבה לדוגמה לשימוש. ניתן לבנות ולהוסיף סביבות נוספות. מעבר לכך, הפרויקט כולל גם אפשרויות שליטה על פרמטרים רבים ופיצ'רים נוספים להאצת התכנסות ושיפור ביצועים, שמירת והצגת נתוני אימון ויישום, ואנימציה לבחינה ויזואלית של יישום המודל בסביבה.

Snake הוא משחק מחשב לשחקן יחיד, בו השחקן שולט בנחש שנע במשטח דו ממדי בצורת "רשת" של ריבועים (grid). במשטח מפוזרים תפוחים ובכל פעם שהנחש מגיע לתפוח הוא אוכל אותו, גופו מתארך ונוצר תפוח חדש במקום אחר. מטרת השחקן היא לאכול כמה שיותר תפוחים מבלי להיתקע בעצמו או בקירות התוחמים את הלוח, שכן זוהי פסילה. התארכות הנחש מובילה לרמת קושי עולה ככל שהמשחק מתקדם, במיוחד אם גודל הלוח קטן.

בחרתי במשחק Snake בשל פשטותו, משום שהדבר מאפשר לי לבדוק את היעילות של שדרוגים רבים ומגוונים לגרסה הבסיסית של DQN ואת ההשפעה של פרמטרים שונים על ההתכנסות, תוך שמירה על עלות חישובית מינימלית ועל evaluation פשוט. הסביבה Snake מהווה צעד ראשוני בתהליך ההתקדמות לעבר שימוש ב-RL בכלל, וב-DQN בפרט, למקרים יותר ויותר פרקטיים ורלוונטיים לבעיות בעולם האמיתי.

קיימים מימושים רבים של DQN ופרויקטים רבים של המשחק Snake באלגוריתם זה, אך לא מצאתי אף פרויקט שמאפשר שליטה בפרמטרים ופיצ'רים בכמות שמגיעה למימוש שלי. מימשתי את האלגוריתם תוך לקיחת השראה מהמקורות קיימים, אך אני יצרתי את עיקר הקוד ואת הפיצ'רים הנוספים. התבססתי על [מאמר [1]](#reference_1) לערכים אופטימליים לחלק מהפרמטרים, כגון מקדם הנחה (gamma) וערכי התגמולים (rewards). ערכים אלו נתנו לי בסיס להתחיל את האימונים ממנו, ולהתמקד בפרמטרים ושדרוגים אחרים שהעדפתי להתעסק בהם.

קהל היעד של הפרויקט הוא מפתחים. הפרויקט מספק מימוש לדוגמה, או לפחות מימוש חלקי של DQN עם יותר פיצ'רים ואפשרויות ממרבית המימושים הקיימים. מפתחים יכולים להשתמש בסביבה הקיימת בשביל חקר ביצועים על קונפיגורציות של פרמטרים ופיצ'רים של DQN לפני שמנסים לשלב אותם בפרויקטים אחרים, או להשתמש בחלקים הדרושים מהמימוש.

חשוב לציין שמאחר ש-Snake הוא משחק פשוט, ניתן להגיע לביצועים מעולים באמצעות אלגוריתמיקה קלאסית, ללא למידת מכונה. אך יש לזכור שמטרת הבחירה במשחק Snake היא שישמש כסביבה לביצוע RL, לכן הפרויקט מתמקד בפתירתו באמצעות RL, וספציפית DQN.

דגשים חשובים לקריאת תיק הפרויקט

מלבד תוכן העניינים, לאורך הקובץ קיימות התייחסויות למקומות אחרים בו. לנוחותכם, הוספתי קישורים המסומנים ב-*italic* ולחיצה עליהם תוביל למקום המוזכר.

ידע קודם אודות פעולתן של רשתות נוירונים הכרחי להבנת קובץ זה, ולא יפורט בתיק. מומלץ ידע ב-Reinforcement Learning (RL) וב-DQN, אך הוספתי רקע תיאורטי בסיסי על כך וכן פירוט מעמיק יותר על שדרוגים במקומות הרלוונטיים בהמשך. מוזמנים גם להיכנס לקישורים ב[נספחים](#נספחים) להסברים מפורטים יותר לחיזוק ההבנה.

רקע תיאורטי

RL – הגדרה כללית

RL הוא ענף בלמידת מכונה המתמקד בייעול התנהגות וקבלת החלטות בסביבה מסוימת, כגון משחקי מחשב ורובוטיקה. בפרויקט שלי במשחק Snake. היעד הוא להגיע למדיניות (policy) המתארת התנהגות אופטימלית בסביבה. הלמידה נעשית על ידי ניסוי וטעיה של "סוכן" שפועל ב-"סביבה". באלגוריתם DQN הסוכן הוא רשת נוירונים עמוקה שנקראת Q-Network.

סביבה

לעיתים קרובות משתמשים בסימולציה ממוחשבת כסביבה, במיוחד עבור משחקי מחשב. סביבה שמאפשרת תהליך Reinforcement Learning באלגוריתם DQN צריכה לענות על מספר דרישות:

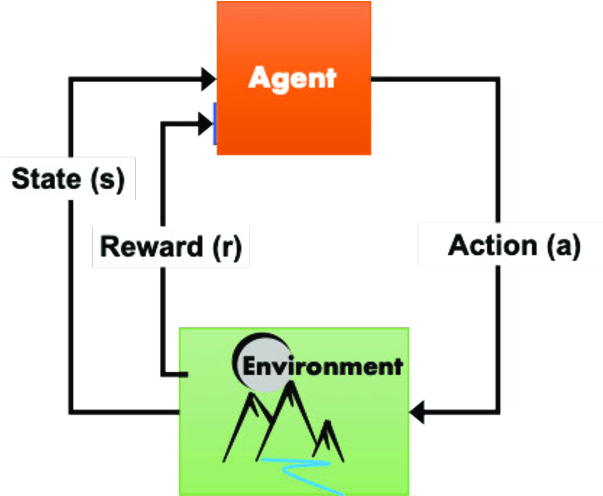
* מרחב פעולה סטטי – בכל מצב בסביבה יש אותן פעולות אפשריות (כל נוירון בשכבת הפלט של רשת ה-Q מייצג פעולה).
  + ב-Snake הפעולות הן ימינה, שמאלה, למעלה, למטה.
  + שחמט, לדוגמה, היא סביבה לא מתאימה, משום שכמות הפעולות משתנה בין מצבים, וכך גם המשמעות שלהן.
* ייצוג מצב קבוע – מבנה הקלט לרשת צריך להישאר קבוע.
* השפעת הפעולה הנוכחית תלויה רק במצב הנוכחי, לא באירועים קודמים.
* מערכת תגמולים מתאימה – תלוי במפתח.

סוכן

בכל איטרציה ב-RL, הסוכן מקבל את מצב הסביבה (state) ובוחר איזו פעולה (action) לבצע. ב-DQN מדובר ברשת נוירונים עמוקה, שהקלט שלה הוא מצב הסביבה, והפלט הוא ציוני איכות (ערכי Q – פירוט בהמשך) לכל הפעולות האפשריות במצב. הפעולה עם הציון הכי גבוה היא זו שהסוכן "בוחר".

תהליך אימון הסוכן נעשה באמצעות שני תהליכים נפרדים הפועלים לסירוגין או במקביל:

1. **איסוף מידע – צבירת "חוויות"**

בכל איטרציית ייצור מידע: הסוכן מקבל כקלט את המצב הנוכחי בסביבה (State), ובוחר פעולה (Action). לאחר ביצוע הפעולה הסוכן מקבל פידבק מהסביבה בצורה של "תגמול" (Reward), שנקבע על ידי המפתח עבור אירועים מסוימים. תגמול יכול להיות חיובי, שלילי או ניטרלי; גודל התגמול משפיע גם הוא על האימון. הפידבק כולל גם את המצב החדש בסביבה, שישמש בהמשך לתיוג החוויה שנצברה.

כל חוויה מאוחסנת בזיכרון שנקרא experience buffer, ובהמשך תשמש לאימון.

1. **תיוג ואימון**

בכל איטרציית אימון שולפים חוויות מהזיכרון ומתייגים אותן. ערך ה-Q הוא התגית, ומחושב באמצעות Bellman equation:

s – state, s’ – next state

a – action, a’ – next action

r – reward

gamma – (בד"כ 0.9-0.99) מקדם לחשיבות הפרסים העתידיים

Q(s’, a’) – מוערך באמצעות חיזוי הרשת

נשים לב שהגדרת היא רקורסיבית, משום שהיא כוללת את , שהוא ערך ה-Q של הפעולה הבאה. תיאורטית, הרקורסיה נעצרת כשאין עוד פעולה, כלומר כשנגמר המשחק. בפועל, תיוג המידע משתמש בחיזוי של הרשת הנוכחית עבור , ולכן התיוג לא מדויק לחלוטין. אך רכיב ה-reward מדויק, משום שהתקבל כפידבק מהסביבה, ולכן יש ערך לתיוג זה.

בכל איטרציה מתבצע תיוג מחדש יותר מדויק מהקודם, משום שחיזוי הרשת משתפר.

מבנה הפרויקט

נהוג לחלק את שלבי למידת המכונה לשלושה חלקים:

1. איסוף, הכנה וניתוח הנתונים – Collect, Prepare, and Analyze data
2. בניית ואימון המודל – Build and train deep learning model
3. יישום – Deployment

ב-Reinforcement Learning החלוקה פחות מוגדרת, משום שייצור הנתונים ותיוגם מתבצע במקביל לאימון המודל. לכן אסביר באמצעות חלוקה זו, אך לא מדובר בהכרח ברצף כרונולוגי.

להלן תמצית רצף האימון במימוש שלי, שנועדה להעניק מבט-על לפני שצוללים להסבר המעמיק על כל חלקי הפרויקט.

* תחילת הריצה: טעינת פרמטרים, בניית מחלקת הסוכן, בניית הסביבה.
* ייצור נתונים ראשוני לפני האימון, עד לכמות מינימלית.
* תחילת האימון בסבבים. כל סבב כולל:
  + איטרציות אימון:
    - קבלת המצב מהסביבה

ייצור נתונים

* + - חיזוי ובחירת הפעולה
    - ביצוע הפעולה בסביבה
    - קבלת פידבק
    - אחסון הפידבק בזיכרון
    - שליפת "חוויות" אקראיות מהזיכרון

תיוג נתונים

* + - תיוג החוויות

אימון הרשת

* + - אימון הרשת על החוויות המתויגות
    - עדכון פרמטרים דינמיים (פירוט בהמשך)
  + איטרציות מבחן:
    - קבלת המצב מהסביבה
    - חיזוי ובחירת הפעולה
    - ביצוע הפעולה בסביבה
  + שמירת ביניים של המודל ומדדי האימון והמבחן.
  + עדכון המשתמש בסטטוס האימון.
* שמירה סופית של המודל, מדדי האימון והמבחן, וייצור גרפים עבורם.
* עדכון המשתמש בסיום האימון.

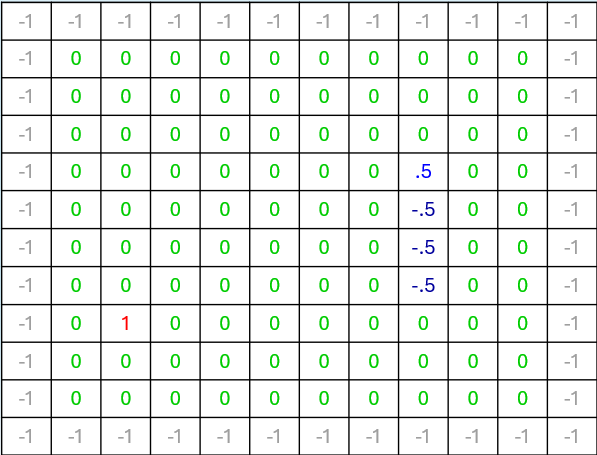
ייצור הנתונים

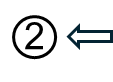
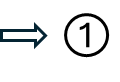
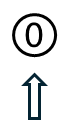
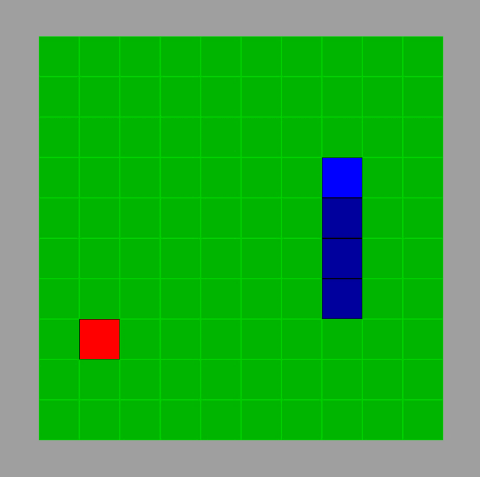
התחלתי מלבנות את המשחק Snake כמחלקה שתתפקד כסביבת אימון ויישום ממוחשבת, בה יתרחש ייצור הנתונים. שמתי דגש על מבנה מחלקה המתאים לתהליך אימון ב-DQN (ראה [רקע תיאורטי](#רקע_תיאורטי)).

בחרתי בתנועה יחסית, כלומר שלוש פעולות המסמלות תנועה ישר, פנייה ימינה ופנייה שמאלה ביחס לכיוון הנוכחי.

בחרתי לייצג את לוח המשחק כמטריצה עם 5 ערכים שונים שמייצגים תפוח, ראש הנחש, גוף הנחש, קיר ומשבצת ריקה. הערכים מנורמלים לטווח 1- עד 1.

השתמשתי בספרייה graphics.py, שהיא ספריית גרפיקה בסיסית ויחסית איטית, אך מספיק טובה לנקודת זמן זו.

הדגמת מצב המשחק והפעולות האפשריות:



|  |  |
| --- | --- |
| מאורע | תגמול |
| לאכול תפוח | 15+ |
| להיתקע בקיר | 10- |
| להיתקע בגוף | 10- |
| להתקרב לאוכל | 1+ |
| להתרחק מאוכל | 1- |

מימשתי את מערכת התגמולים המופיעה ב[מאמר [1]](#reference_1), משום שמערכת תגמולים טובה תאפשר לי להתמקד בבדיקת יעילות השדרוגים של DQN, במקום להתעסק בבניית ובדיקת מערכת מאפס. מערכת התגמולים מתוארת בטבלה:

התגמולים משמשים לתיוג, עליו אסביר בהמשך.

שדרוגים לסביבה Snake

לאחר שכבר אימנתי ובדקתי מספר מודלים, נתקלתי במספר צרכים שבעקבותיהם שיניתי את הסביבה.

**סביבה וקטורית**

עקב זמן אימון ממושך, יצרתי סביבה וקטורית כמחלקה VecSnake, שמאפשרת מספר משחקים במקביל על מנת להאיץ את ייצור הנתונים ואת מבחן המודל. לדוגמה, במקום לבצע 10 פעולות עוקבות בסביבה אחת, בכל איטרציית איסוף נתונים, אפשר לבצע פעולה אחת בכל 1 מתוך 10 סביבות מקבילות. ב-TensorFlow, עשרה חיזויים נפרדים איטיים הרבה יותר מחיזוי אחד על עשרה קלטים בבת אחת, לכן סביבה וקטורית הובילה לשיפור דרסטי בזמן האימון והמבחן.

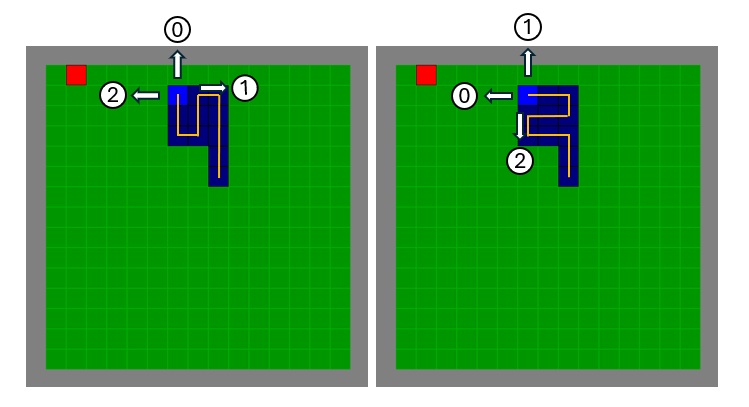
**קטיעת לולאות**

במודלים שעדיין לא התכנסו לביצועים טובים, לעיתים מתרחשת בעיה שהם חוזרים על רצף פעולות זהה המסתיים בנקודת ההתחלה, ונעים במעין "לולאה" אינסופית. במהלך האימון הלולאה נקטעת בסופו של דבר ע"י exploration כאשר epsilon גדול מ-0, אך בשלב המבחן לא מתבצע exploration. התמודדתי עם הבעיה על ידי זיהוי לולאות במשחק וקטיעה מוקדמת שלהן. אם מספר הפעולות הרצופות בהן לא נאכל תפוח או נגמר המשחק עובר סף מסוים – הסביבה תסיים את המשחק גם בלי פסילה. חשוב להשתמש בקטיעת לולאות רק בשלב המבחן *ולא* בשלב ייצור הנתונים, על מנת להימנע ממניפולציות שישבשו את הנתונים.

**Pygame**

זוהי ספריית אנימציה יותר נוחה שמאפשרת גם אינטראקציה מובנית עם משתמש אנושי, לכן עברתי להשתמש בה לבדיקת המודל ולמשחק אנושי בסביבה.

**תנועה מוחלטת**

שימוש בארבע פעולות אפשריות המסמלות כיוונים מוחלטים בלוח (למעלה, למטה, ימינה, שמאלה) במקום פנייה ביחס לנחש שהייתה לפני. תנועה מוחלטת שיפרה ביצועים ע"י התמודדות עם אי-ודאות במצבים זהים לכאורה, בהם מטריצת הלוח זהה אך השפעת הפעולות שונה. דוגמה למצבים זהים לכאורה היוצרים חוסר בהירות כאשר משתמשים בתנועה יחסית:

תוצאות מעבר לתנועה מוחלטת:

ניתן לראות התכנסות על רמת ביצועים גבוהה יותר עבור תנועה מוחלטת (4 פעולות) לעומת תנועה יחסית (3 פעולות).

ייתכן כי העיכוב הקטן בשיפור בתחילת האימון נגרם משום שאתחלתי חלק מהמשקולות באמצעות מודל מאומן על תנועה יחסית.

תיוג הנתונים

במהלך שלב הייצור נצברות חוויות, שנשמרות בצורה:

(state, action, reward, new state, done)

done הוא משתנה בוליאני על האם נגמר המשחק.

על החוויות מהזיכרון מבוצע experience replay: בכל איטרציית אימון שולפים חוויות מה-[experience buffer](#experience_buffer), מתייגים אותן, ומאמנים עליהן את המודל.

לכל חוויה מחושב ערך ה-Q ע"י Bellman equation (הסבר ברקע התיאורטי):

בכל חוויה נתון state אחד עם action אחת בלבד וה-reward עליו, לכן ניתן לחשב רק את ערך ה-Q מפעולה זו. אך בהינתן state, הרשת חוזה את ערכי ה-Q של כל ה-actions האפשריים במצב (במקרה הזה 4).

מאחר שנתון פידבק רק על פעולה אחת, התיוג מתבצע באופן חלקי, כך שרק הערך של הפעולה שקרתה מתויג.

נסמן = ערך מחיזוי, = ערך מחישוב. אם נבחרה הפעולה action=2, נקבל:

הקלט הוא:

חיזוי הרשת הוא:

התגית היא:

כך מתייגים כל חוויה ב-minibatch של מספר חוויות.

\*גם ערך ה-Q שמחושב כולל רכיב של חיזוי, שהוא Q(s’, a’). חיזוי זה לא מתבצע על ידי הרשת העיקרית של הסוכן, אלא על ידי target network. רשת זו היא גרסה מוקדמת יותר של המודל, והשימוש בה מייצב את האימון משום שכך חישוב התגית לרשת העיקרית לא תלוי בפלט של עצמה. רשת המטרה מסתנכרנת עם הרשת העיקרית כל כמות מסוימת של איטרציות אימון (התדירות נקבעת לפי פרמטר), כלומר מקבלת את המשקולות שלה. בסנכרון ניתן לבצע עדכון רך (soft update) שמרכיב את העדכון מהמשקולות של שתי הרשתות, לפי האחוזים הנקבעים בפרמטר tau. עדכון קשיח הוא העתקה מלאה של המשקולות העיקריות, ללא שימוש במשקולות של רשת המטרה.

בניית ואימון המודל

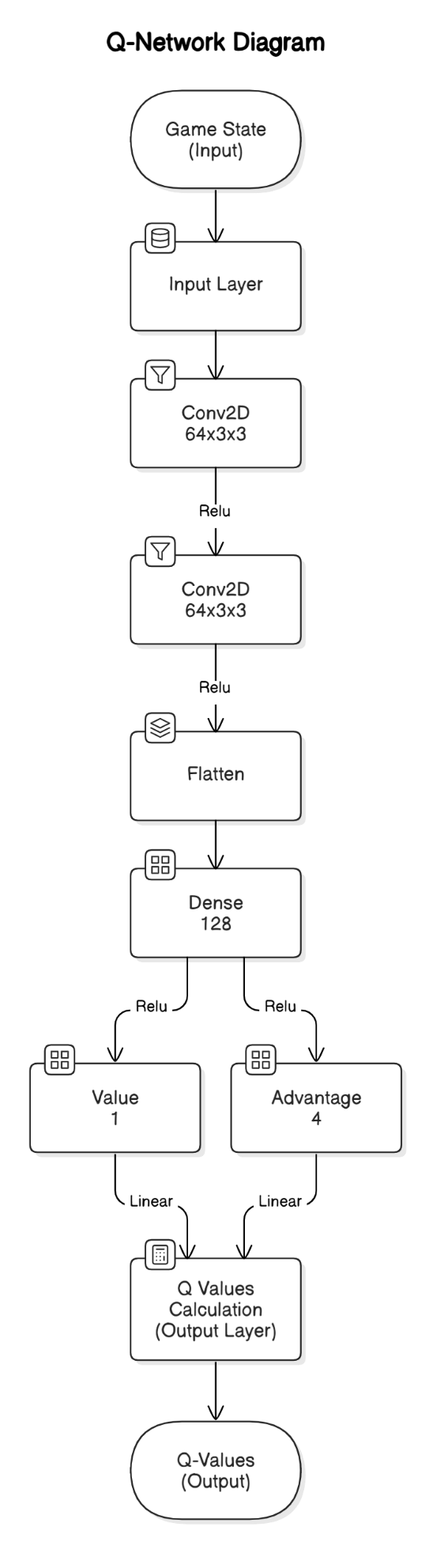
מימוש מחלקת הסוכן כלל הרבה אתגרים. היו לי בעיות בהתקנה ובקונפיגורציה של TensorFlow, אך הצלחתי לאחר מספר ימים. תהליך מימוש המחלקה הוביל אותי למסקנה שההבנה שלי לגבי האלגוריתם לא הייתה מספקת בשביל ליישם אותו לחלוטין, לכן חזרתי ללמוד תוך מחשבה על איך איישם בקוד כל חלק באלגוריתם.

בחירת מבנה הרשת והיפר-פרמטרים כללו שילוב של ערכים נפוצים ממקורות מהם למדתי, והרבה ניסוי וטעיה שלי לכוונון שלהם תוך שימוש בכלי מדידה והערכה שיפורטו בהמשך.

התחלתי מאימון רשתות MLP, אך לא ניכר שום שיפור ברמת המודל. לאחר debugging ממושך שמתי לב לתופעה מוזרה בנתונים: היו כפילויות רבות של מצבים זהים ב-[experience buffer](#experience_buffer). הבעיה נבעה משמירת פוינטר למטריצת המשחק במקום העתקה שלה. לאחר תיקון ניכר שיפור ברמת המודל, אך עדיין היה פער משמעותי מרמה אנושית סבירה.

התחלתי תהליך ארוך של חקר ביצועים שבאמצעותו ביצעתי hyper-parameter tuning, שיניתי את ארכיטקטורת הרשת, והוספתי שדרוגים לסוכן, לסביבה ולאלגוריתם שמטרתם לשפר את הביצועים ולהאיץ את האימון. להלן תמצית השינויים שהכנסתי לאורך הזמן (סדר לא כרונולוגי):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| היכן השינוי בא לידי ביטוי | שם השינוי | מטרת השינוי | | |
| שיפור ביצועים | האצת התכנסות | מדידה, בקרה והפקת תובנות ליישום |
| סוכן | [Experience replay](#experience_replay) | ✓ | ✓ |  |
| [CNN](#CNN) | ✓ |  |  |
| [Dueling network](#dueling_network) | ✓ |  |  |
| [Target network](#target_network) | ✓ | ✓ |  |
| [Soft updates](#soft_update) | ✓ | ✓ |  |
| [הגדלת מודל מאומן](#הגדלת_מודל_ואימון_מקדים) |  | ✓ |  |
| סביבה | [סביבה וקטורית](#סביבה_וקטורית) |  | ✓ |  |
| [קטיעת לולאות](#קטיעת_לולאות) |  |  | ✓ |
| [תנועה מוחלטת](#תנועה_מוחלטת) | ✓ |  |  |
| [ספריית גרפיקה](#Pygame) |  |  | ✓ |
| פרמטרים | [כוונון פרמטרים](#היפר_פרמטרים_וכוונון) | ✓ | ✓ |  |
| לולאת האימון | [Profiling](#Profiling) |  |  | ✓ |
| [שמירת מדדי אימון](#מדדי_אימון) |  |  | ✓ |
| [אימון מקדים למודלים מוגדלים](#אימון_מקדים) |  |  | ✓ |
| [Multi-Processing](#Multi_Processing) |  | ✓ |  |
| [Checkpoints](#Checkpoints) |  | ✓ |  |
| [אימיילים](#Emails) |  | ✓ |  |
| [Verbose](#Verbose) |  | ✓ |  |

בניית המודל

בסופו של דבר הגעתי לארכיטקטורת הרשת המוצגת, עליה אפרט בעמוד הבא.

* גודל שכבת הקלט יכול להשתנות בהתאם לגודל הלוח עבורו יוצרים את המודל.
* תחילה, הקלט עובר 2 שכבות קונבולוציה עוקבות, כל אחת עם 64 קרנלים בגודל 3x3 הנעים בפסיעות של 1, עם אקטיבציית ReLU.

שכבות הקונבולוציה נועדו לזיהוי דפוסים מרחביים במשחק, וכמות המשקולות בהן לא תלויה בגודל הקלט, לכן אני מבצע העתקה מלאה שלהן במעבר בין מודלים של גדלי לוח שונים.

* לאחר שיטוח, הקלט מועבר לשכבה לינארית עם 128 נוירונים ואקטיבציית ReLU.

כמות המשקולות בשכבות הלינאריות תלויה בגודל הקלט, לכן אני מבצע העתקה חלקית שלהן במעבר בין מודלים, ומשלים באמצעות ריפוד באפסים (ניתן להשתמש באינטרפולציה, אך ברוב המקרים תיווצר הטייה לפעולה אחת ויעילות ייצור הנתונים תפחת).

* הקלט מועבר לשני ערוצים נפרדים: Value ו-Advantage.

זוהי ארכיטקטורת Dueling Network (ראה [מאמר [5]](#reference_5)), שמשפרת את יעילות האימון כשלא כל הפעולות נחקרו בכל מצב, ומייצבת את הלמידה ע"י הפחתת רגישות לפידבקים "רועשים".

* + ערוץ ה-Value מעריך את התגמול הצפוי ממצב ללא תלות בפעולות, באמצעות נוירון 1 ואקטיבציה לינארית.
  + ערוץ ה-Advantage חוזה את התגמול מכל פעולה במצב, באמצעות 4 נוירונים ואקטיבציה לינארית.
* מחסרים מערכי ה-Advantage את הממוצע שלהם על מנת לשמור רק על היחסים בין הפעולות, וסוכמים עם חיזוי ערוץ ה-Value לקבלת ערכי Q.
* תוצאת צירוף הערוצים היא הפלט הסופי של הרשת.

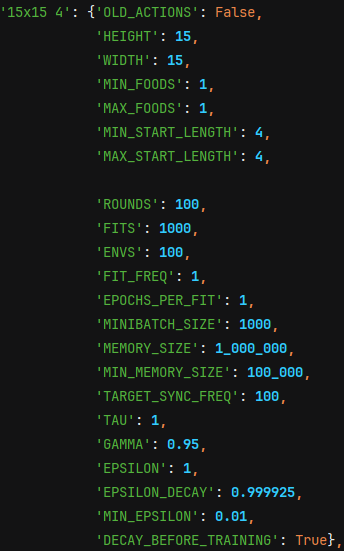
מאחר שחיזוי ערכי Q היא מטלת רגרסיה, אני משתמש בפונקציית השגיאה Mean Squared Error:

n שווה לכמות החוויות שנשלפות מהזיכרון בכל איטרציית אימון.

להגדרת y ו-ŷ ראה [תיוג הנתונים](#תיוג_הנתונים).

מאחר שאני רוצה להתמקד בפרמטרים ייחודיים ל-RL, עבור אופטימיזציה להתכנסות אני משתמש ב-Adam עם ערכי ה-default של keras לפרמטרים: learning\_rate=0.001, beta\_1=0.9, beta\_2=0.999, epsilon=1e-07

היפר-פרמטרים וכוונון



אימנתי מודל עם ארכיטקטורת הרשת הנ"ל ועם הפרמטרים המוצגים בתמונה. מצורפת טבלה המסבירה על המשמעות של הפרמטרים השונים, ולאחריה תוצאות האימון בקונפיגורציה זו.

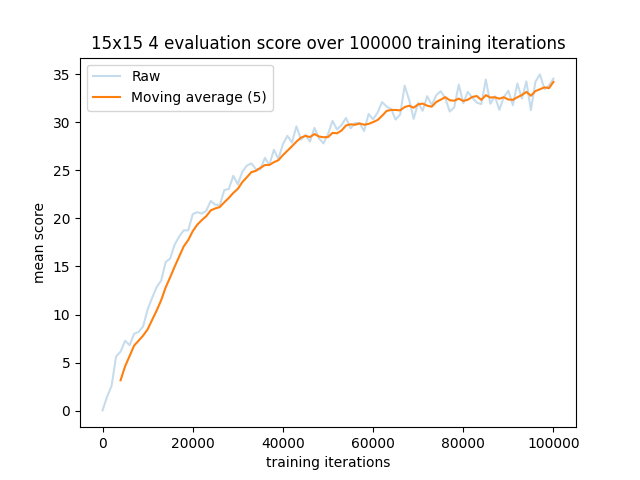
|  |  |  |
| --- | --- | --- |
| הערות | הסבר | פרמטרים |
|  | קובע אם להשתמש במנגנון התנועה הישן (תנועה יחסית) או בחדש (תנועה מוחלטת). | OLD ACTIONS |
|  | גודל הלוח הוא HEIGHT x WIDTH. | HEIGHT, WIDTH |
| הוספתי את האפשרות הזו לסביבה אך בפועל התמקדתי באימונים עם תפוח אחד בלבד. | קצוות הטווח לכמות התפוחים שתהיה בסביבה בכל רגע נתון. הכמות נקבעת כמספר אקראי בטווח. | MIN FOODS, MAX FOODS |
|  | קצוות הטווח לאורך הנחש בתחילת המשחק, האורך הוא מספר אקראי בטווח זה. | MIN START LENGTH, MAX START LENGTH |
| סה"כ יבוצעו ROUNDS x FITS איטרציות אימון.  בכל איטרציית אימון הרשת תעשה EPOCHS\_PER\_FIT אפוקים עלMINIBATCH SIZE חוויות.  סך כל החוויות שייצברו הוא  ROUNDS x FITS x ENVS x FIT FREQ | כמות סבבי האימון שיבוצעו, כמפורט ב[תמצית רצף האימון](#תמצית_רצף_האימון) תחת [מבנה הפרויקט](#מבנה_הפרויקט). | ROUNDS |
| איטרציות האימון בכל סבב. | FITS |
| כמות הסביבות המקבילות בהן יתבצע ייצור הנתונים. | ENVS |
| כמות איטרציות ייצור הנתונים על כל איטרציית אימון. במילים אחרות: כמות הצעדים הרצופים בכל אחת מהסביבות, שעבורם תתבצע איטרציית אימון אחת של הרשת. | FIT FREQ |
| כמות האפוקים באיטרציית אימון אחת של הרשת (model.fit) | EPOCHS PER FIT |
| כמות החוויות הנשלפות מהזיכרון בכל איטרציית אימון. | MINIBATCH SIZE |
| כשהזיכרון מלא, חוויות ישנות נדחפות החוצה ומפנות מקום לחוויות חדשות. | גודל הזיכרון (ה-[experience buffer](#experience_buffer)) המקסימלי. | MEMORY SIZE |
|  | כמות החוויות המינימלית לתחילת אימון הרשת. | MIN MEMORY SIZE |
|  | כל כמה איטרציות לעדכן את ה-target network במשקולות של הרשת העיקרית. | TARGET SYNC FREQ |
| 1=hard update | מקדם [soft update](#soft_update), ראה תחת [תיוג הנתונים](#תיוג_הנתונים). | TAU |
|  | מקדם הנחה, ראה [רקע תיאורטי](#רקע_תיאורטי). | GAMMA |
| פעולה אקראית נקראת exploration, ופעולה של הסוכן נקראת exploitation.  exploration חשוב על מנת לוודא שהסוכן חוקר הרבה אפשרויות, ולא "נתקע" על טקטיקה מסוימת שמובילה למקסימום מקומי של ביצועים. | ערך בין 0 ל-1, הקובע את ההסתברות שבמהלך ייצור הנתונים תתבצע פעולה אקראית במקום זו שהסוכן בחר. | EPSILON |
| המחשה לדעיכת epsilon לאורך האימון:  A graph of a person with a blue line  Description automatically generated | אנו רוצים ש-EPSILON ידעך לאורך האימון משום שהסוכן כבר חקר הרבה אפשרויות והשתפר. הדעיכה אקספוננציאלית ו-EPSILON DECAY הוא בסיס החזקה. | EPSILON DECAY |
| ערך מינימלי בו EPSILON מפסיק לדעוך, על מנת לוודא שתמיד יהיה "מחקר" של טקטיקות נוספות. | MIN EPSILON |
|  | האם להתחיל את דעיכת EPSILON במהלך ייצור הנתונים הראשוני, לפני תחילת אימון הרשת. | DECAY BEFORE TRAINING |

תוצאות אימון המודל בדוגמה

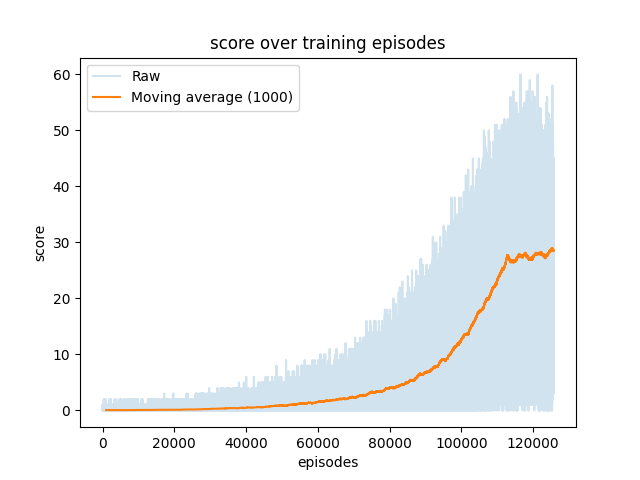
לאחר אימון ביצועי הסוכן עלו מ-0.03 ל-34.54 ממוצע תפוחים במשחק.

\*לגרפים המוצגים בפרק זה ביצעתי החלקה באמצעות [ממוצע נע](#ממוצע_נע) ברוחב המצוין במקרא כל גרף.

ניתוח תוצאות ומדדי אימון: ממוצע התפוחים במשחקי אימון ומבחן



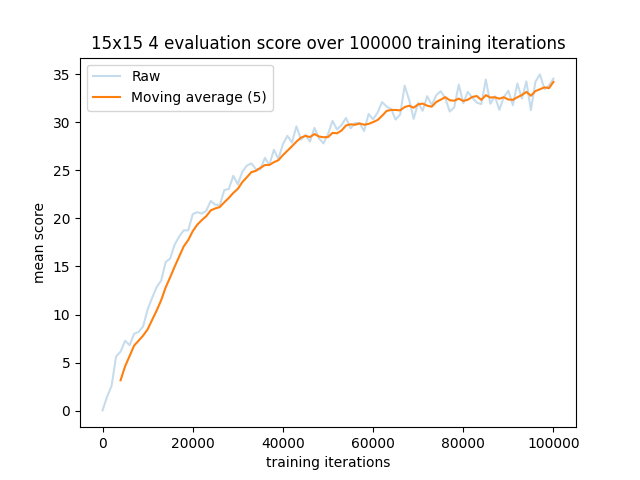
ממוצע תפוחים במשחקי מבחן



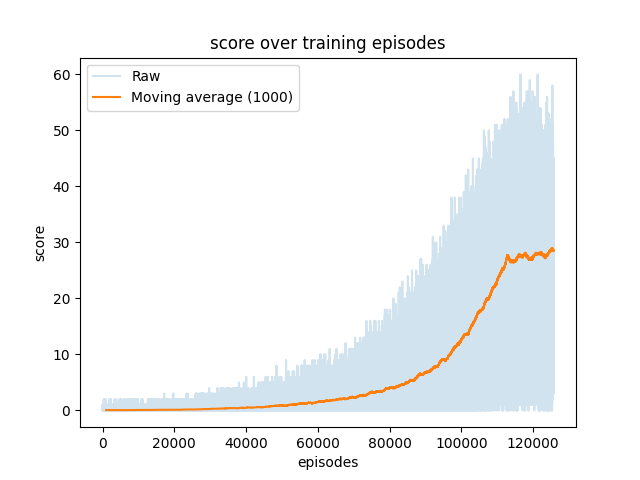
ממוצע תפוחים במשחקי אימון

במשחקי המבחן ניכר שיפור מהיר בביצועי המודל, בעוד שבמשחקי האימון ניכר שיפור מתון תחילה שלאחר מכן מאיץ. אני משער שתופעה זו נובעת מעצם העובדה שרק במהלך משחקי האימון מתבצע exploration, שעלול לגרום לפסילת הסוכן בטרם עת. כזכור, epsilon דועך אקספוננציאלית, לכן רוב ה-exploration מתבצע בתחילת האימון. בשלבי אימון מתקדמים epsilon נמוך ופחות מחבל בפעילות הסוכן. משחקי המבחן משקפים את היכולת האמיתית של המודל, משום שבהם אין exploration.

כעת נביט בגרפים לאחר שסימנתי עליהם מלבנים אדומים, המסמלים את אותו פרק אימון.



ממוצע תפוחים במשחקי מבחן



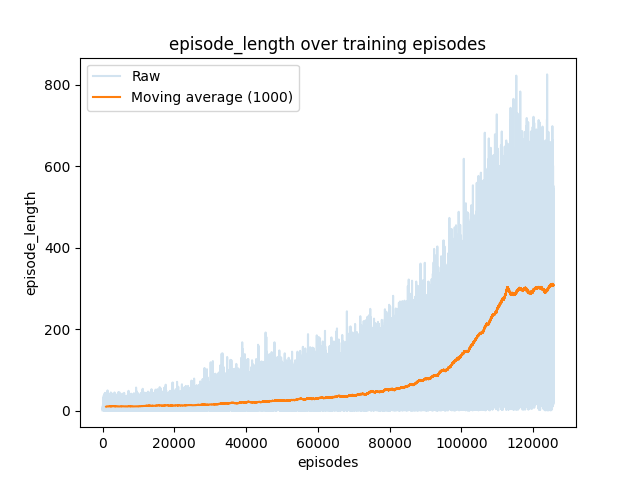
ממוצע תפוחים במשחקי אימון

(הסבר בעמ' הבא)

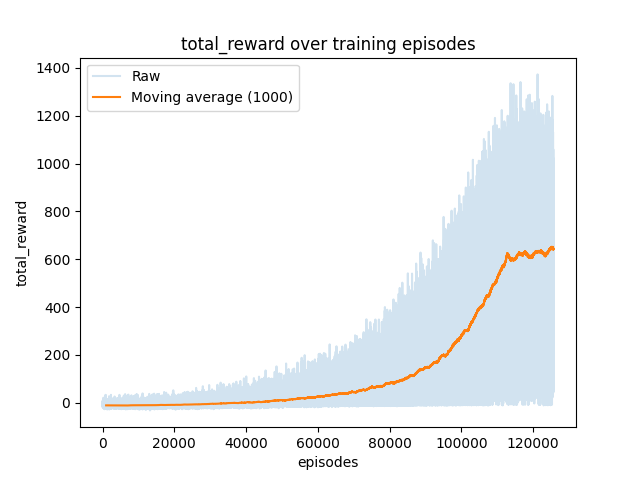
ההסבר ל-"דחיסות" השונה נעוץ ביחידות ששימשו למדידת הביצועים באימון ובמבחן. ציר ה-x בגרף האימון נמדד במשחקים, בעוד שבגרף המבחן הוא נמדד באיטרציות אימון. במספר קבוע של איטרציות אימון ניתן להספיק מעט משחקים ארוכים או הרבה משחקים קצרים. משמע, **משחקים ארוכים ידחסו את גרף האימון אך לא את גרף המבחן**. ככל שהמודל משתפר הוא נפסל פחות, לכן המשחקים מתארכים והגרף נדחס. בנוסף, משחקים שטרם נגמרו בסיום האימון לא נכנסו לגרף האימון אך כן נספרו לגרף המבחן (מדובר ב-100 משחקים בלבד לכן השפעתם זניחה).

ניתוח תוצאות ומדדי אימון: אורך משחק וסכום תגמולים למשחק

הגרף הימני מתייחס לסכום התגמולים שנצברו בכל משחק עד לפסילה, והגרף השמאלי מתייחס לכמות הפעולות עד לפסילה.

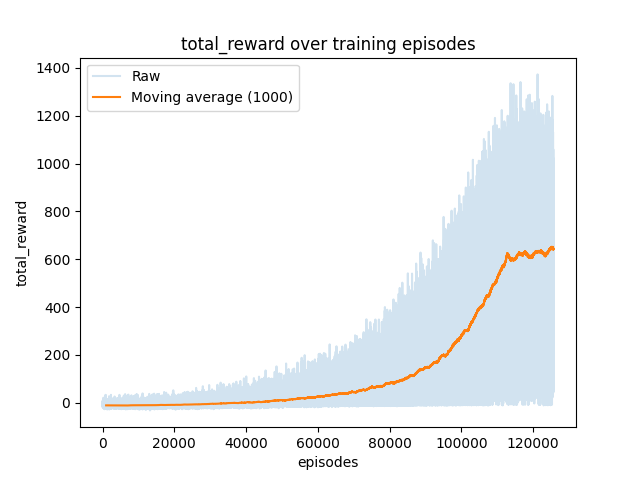
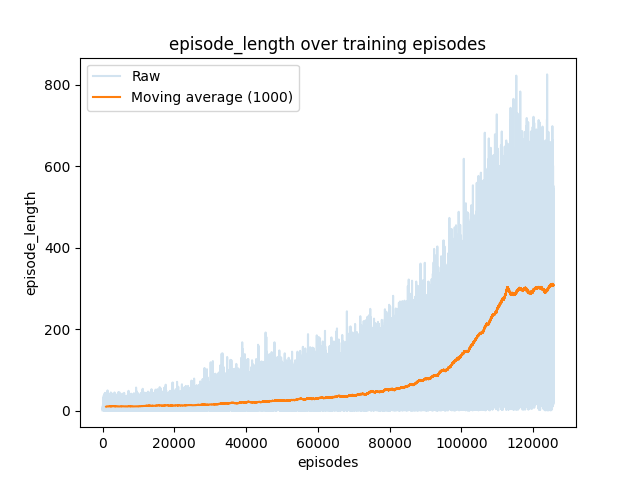


אורך משחקי אימון



סכום תגמולים למשחקי אימון

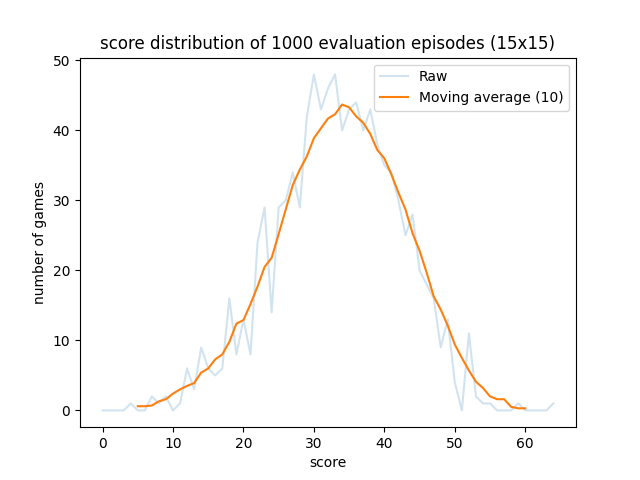
בגרף אורך המשחקים ניתן לראות "שפיצים" רבים שלא מופיעים בגרף סכום התגמולים, במיוחד באזורים המסומנים באדום:

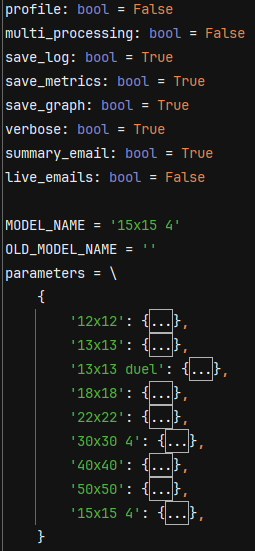


(הסבר בעמ' הבא)

אלו משחקים ארוכים במיוחד, אך עם ביצועים (פרסים מצטברים) סטנדרטיים. להערכתי, סביר כי מדובר במשחקים בהם המודל היה מספיק טוב בשביל להימנע מפסילה לזמן ממושך, אך לא מספיק טוב בשביל לאסוף תפוחים. במקרים מסוג זה ייתכנו "לופים" (ראה [קטיעת לולאות](#קטיעת_לולאות)).

ניתוח תוצאות ומדדי אימון: התפלגות ניקוד

ציר ה-x הוא כמות התפוחים שנאספו במשחק מבחן, וציר ה-y הוא כמות המשחקים (מתוך 1000) בהם הושגה תוצאה זו. לאחר החלקת הגרף עם ממוצע נע ברוחב 10, ניתן לראות שביצועי המודל מזכירים התפלגות נורמלית.

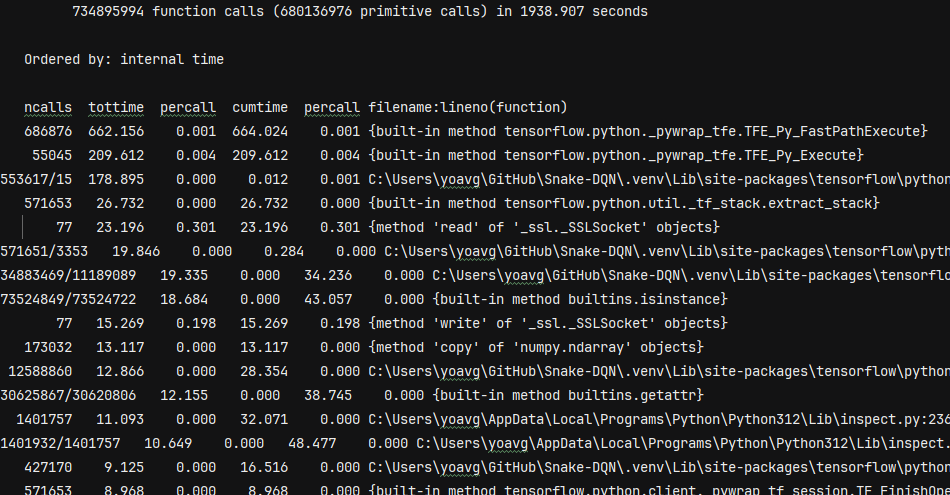
ייעול אימונים

כוח המחשוב שברשותי מוגבל ביותר (הקוד רץ על המעבד של הלפטופ שלי, ללא האצה בעזרת GPU) לכן ייחסתי חשיבות רבה להאצת האימון עצמו, ולקטיעה מוקדמת של אימונים בקונפיגורציות שמתבררות כלא יעילות כבר במהלך האימון. על מנת לעשות זאת, לאורך הזמן הוספתי אפשרויות לבקרת והאצת האימונים, שהשליטה בהן מוצגת בתמונה.

העמודים הקרובים מכילים פירוט על הכלים השונים.

**Profiling**

Profiling הוא כלי למדידת ביצועים של תוכנה בזמן הריצה, שעוזר לזהות חלקים שצורכים הרבה זמן חישוב. שימוש ב-profiling הוביל אותי לתובנות שאיפשרו לי לייעל את הקוד במקומות הנכונים ולהבדיל בין העיקר לתפל מבחינת השפעה על משך האימון. לדוגמה, בזכות profiling גיליתי שתחזית ערכי Q בשלב ייצור הנתונים דורשת הכי הרבה זמן, גילוי שבסופו של דבר הוביל אותי להוסיף [סביבה וקטורית](#סביבה_וקטורית).

בתמונה מופיע חלק מדו"ח profile של אימון מינימלי לדוגמה.

**Multi-Processing**

בדרך כלל ייצור הנתונים (אינטראקציה עם הסביבה) ואימון המודל על חוויות מתבצע לסירוגין, משום שרמת המודל לאחר אימון משפיעה על ייצור הנתונים. אך מאחר שאני משתמש ב-experience replay אין מניעה לבצע אותם במקביל, לכן הוספתי אפשרות להשתמש בספרייה multiprocessing לשם כך. ההבדל במשך האימון לא היה משמעותי, וכאשר התחלתי להשתמש גם ב-pygame גיליתי שנוצר קונפליקט בשילוב הספריות TensorFlow, Pygame, Multiprocessing שמאט את זמן הריצה. לכן העדפתי להפסיק להשתמש באופציה זו.

**מדדי אימון**

הוספתי אפשרות לשמור מדדי אימון כגון ממוצע תפוחים, אורך משחקים וסכום תגמולים למשחק. לשם הנוחות הוספתי פונקציה לייצור הגרפים של המדדים בסיום האימון, שמוצגים ב[תוצאות אימון המודל לדוגמה](#תוצאות_אימון_המודל_בדוגמה). התנודתיות הגבוהה של המדדים והכמות העצומה של הנתונים מהם מקשים על פרשנותם בצורתם המקורית, לכן אני מחליק אותם באמצעות ממוצע נע שמראה בגרף את המגמה הכללית של כל מדד. שקלול הערכים נעשה באופן אחיד, ע"י מתן משקל שווה לכל הנקודות שנכנסות לממוצע.

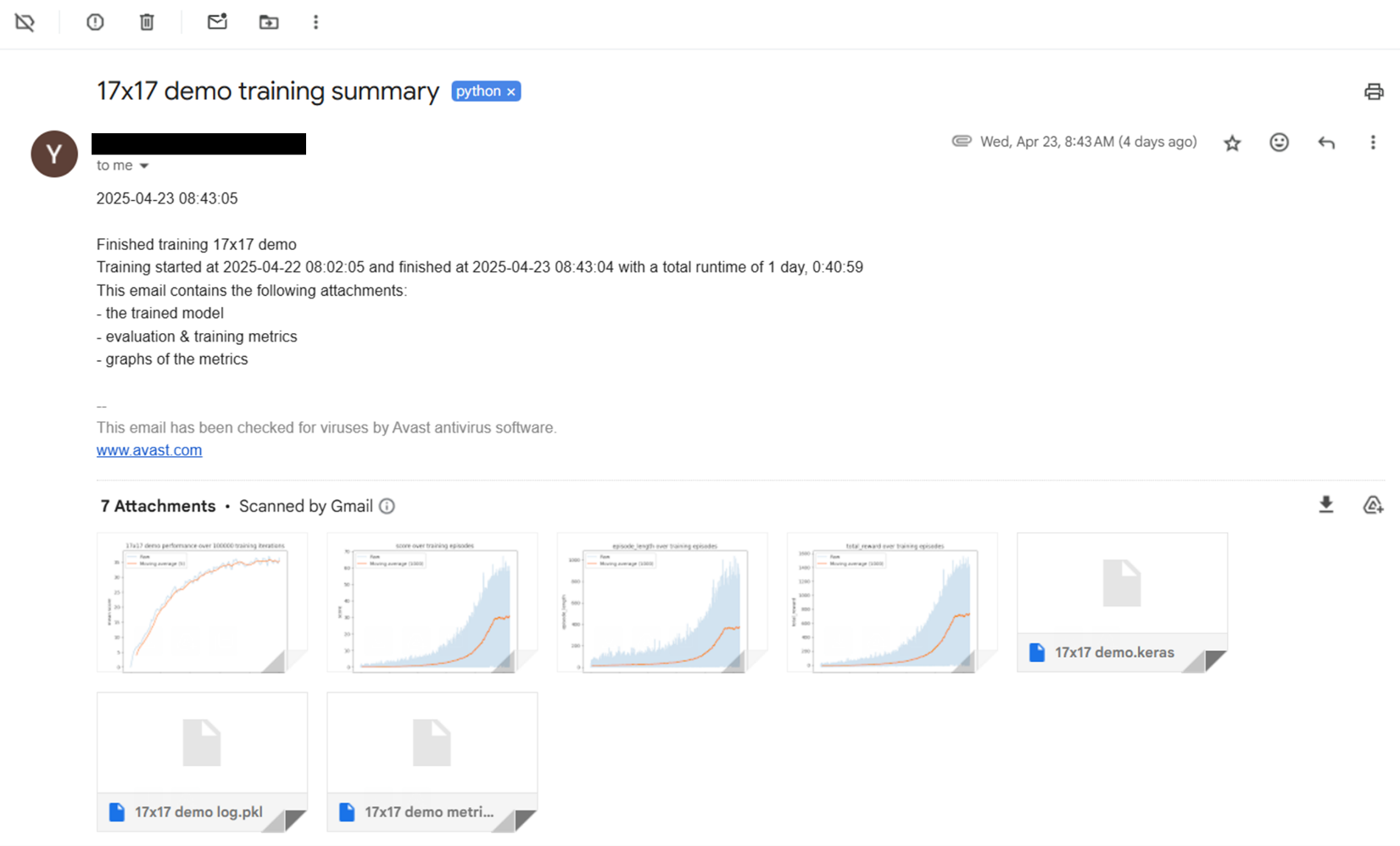
**Checkpoints**

בסיום כל סבב אימונים ומבחן (round), הוספתי checkpoint שמאפשרת לעצור את האימון (ולהמשיך אותו מאותה נקודה אם צריך) בלי לאבד את ההתקדמות עד לשלב זה, ע"י שמירת המודל, מדדי האימון ומדדי המבחן. נוסף לכך, ב-checkpoint מודפסים למשתמש תוצאת המבחן והערכת זמן לסיום האימון, ונשלח [עדכון באימייל](#אימייל_עדכון) עם נתוני המבחן ותחזית לזמן סיום האימון.

**Emails**

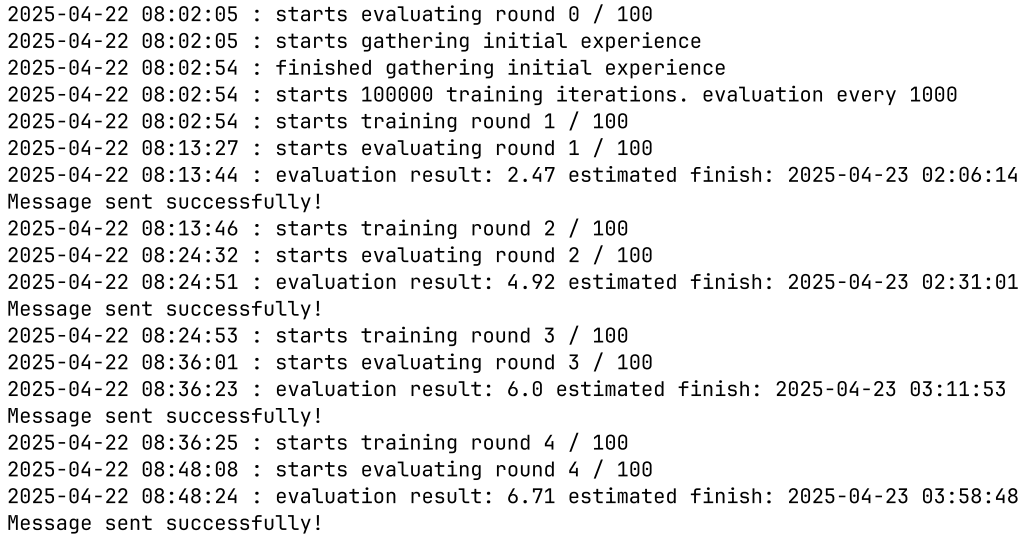
האימונים לוקחים זמן רב בו אני בדרך כלל לא ליד המחשב, לכן כאשר האימון נגמר או כשניתן לקטוע אותו באמצע בשל חוסר יעילות, אני "מבזבז זמן" כשאני לא מתחיל לאמן מודל נוסף או משנה פרמטרים וממשיך את האימון. על מנת לייעל את התהליך הוספתי אפשרות לשליחת אימייל סיכום אימון ואפשרות לשליחת עדכונים חיים באימייל. כאשר האפשרויות מופעלות, השליחה נעשית מהקוד באמצעות החבילות smtplib, email.

בעמוד הבא דוגמאות לשני סוגי האימיילים.

אימייל סיכום לדוגמה:

אימייל עדכון במהלך האימון:

**Verbose**

לשם בקרה על האימון הוספתי פלט מפורט המודפס על המסך, הכולל מידע מועיל כגון סטטוס האימון/המבחן, ביצועי הסוכן, והערכת זמן לסיום האימון. דוגמה לפלט מתחילת האימון:

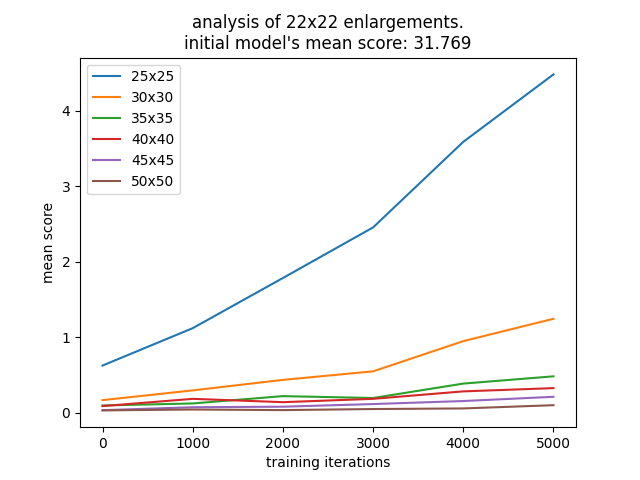
**הגדלת מודל ואימון מקדים**

A graph of a graph

AI-generated content may be incorrect.התנסיתי בלאמן מודלים על לוחות בגדלים שונים, וסיקרן אותי לראות אם למודל שאומן על לוח קטן יהיה יתרון בהתכנסות על לוח גדול יותר. האופן המדויק בו מתבצע המעבר מפורט תחת [בניית המודל](#בניית_המודל), אך עיקר הבעיה הוא שקלט גדול יותר לרשת דורש משקולות נוספות. בהתחלה ניסיתי להוסיף את המשקולות הנדרשות באמצעות אינטרפולציה לינארית, אך התוצאה היתה שהמודל ננעל על כיוון אחד ומתקדם כמעט ורק אליו, ללא תלות בקלט. לאחר מכן ניסיתי לרפד באפסים את המשקולות החסרות, והתוצאות היו מדהימות:

בעקבות ההצלחה של הגדלת מודל, עלתה לי השאלה:

**עד כמה אפשר להגדיל מודל בלי לאבד את יעילות ההגדלה?**

על מנת לענות על שאלה זו ניסיתי לבצע אימון מקדים הכולל פרק אימון קצר למספר רב של הגדלות למודל מסוים, ולפיו לנסות לחזות את היעילות של המשך האימונים. השתמשתי במודל מאומן בגודל 22x22 והגדלה בקפיצות של 5, ואלו התוצאות שקיבלתי:

בגרף ניכר כי במודל 30x30 ניכר שיפור יחסית מהיר, בעוד במודל 50x50 בקושי ניכר שיפור בפרק האימון המקדים. המשכתי רק את האימון של שני המודלים הללו (על מנת לחסוך במשאבי מחשוב), והגעתי לתוצאה הבאה:

לצערי נאלצתי להפסיק את האימונים בשל העלות החישובית הגבוהה שלהם (במיוחד בלוחות מאוד גדולים) ובשל משאבי המחשוב הדלים שברשותי. להערכתי לא ניתן להסיק מסקנות חד משמעיות מהמידע שנצבר ללא המשך ניסויים.

יישום

יישום המודל הוא להפעיל אותו בסביבה, כלומר להשתמש בו לשחק Snake.

המשתמש בוחר הגדרות כגון שם המודל, אורך הנחש, כמות התפוחים, קטיעת לולאות, fps, כמות משחקים וכו' שמפורטים ב[מדריך למשתמש](#מדריך_למשתמש).

ניתן לבחור ביישום ויזואלי של המודל המציג את משחק המודל באמצעות הספרייה Pygame, או לבחור ביישום המודל ללא אנימציה, אשר מריץ את המשחקים במקביל לשם יעילות מרבית. המשחק הטוב ביותר נשמר וניתן לצפות בו מחדש.

הפונקציה המשמשת ליישום לא ויזואלי היא אותה פונקציה המשמשת למבחן בזמן האימון, אך היישום הוויזואלי נעשה באמצעות פונקציה אחרת הדומה במבנה שלה.

משחקים לדוגמה של מודל ה-15x15 מהדוגמה ב[בניית המודל](#בניית_המודל):

לפני אימון:

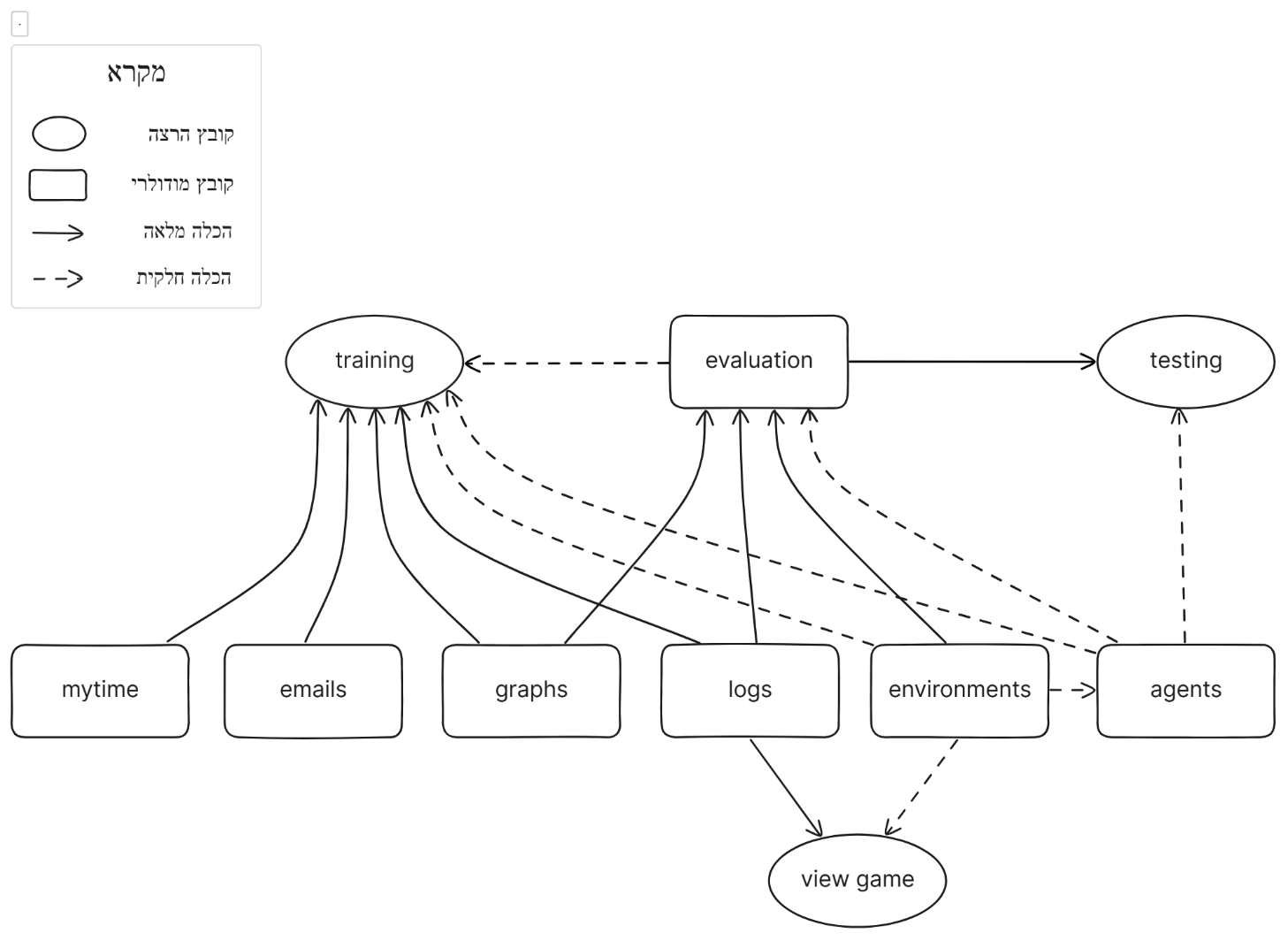
<https://vimeo.com/1081900645/b503552aeb>

אחרי אימון:

<https://vimeo.com/1081900675/fde202df00>

מדריך למפתח

הפרויקט כתוב ב-Python 3.12. אני עבדתי ב-PyCharm Community Edition ואת רוב הספריות הורדתי דרך pip באופן המובנה ב-IDE. עבור עבודה בסביבות אחרות אפשר להתקין ספריות ישירות מ-pip.

קבצי הקוד בפרויקט מחולקים בצורה מודולרית. להבנה מיטבית, תחילה אסביר על המודולים ולאחר מכן על קבצי ההרצה (שמשתמשים במודולים). להלן סיכום תמציתי של קבצי הפרויקט:

|  |  |
| --- | --- |
| תיאור | שם הקובץ (\_.py) |
| קבצי הרצה | |
| קובץ האימון הראשי | [training](#training_py) |
| קובץ היישום הראשי | [testing](#testing_py) |
| צפייה במשחק שמור | [view game](#view_game_py) |
| מודולים | |
| מחלקות הסביבה המשמשות לאימון ולמבחן | [environments](#environments_py) |
| מחלקת הסוכן, פעולות עזר ומחלקות עזר ליישום | [agents](#agents_py) |
| פונקציות מבחן ויזואלי ולא ויזואלי | [evaluation](#evaluation_py) |
| פונקציות לשמירת וקריאת נתוני אימון ומבחן | [logs](#logs_py) |
| פונקציות לייצור גרפים | [graphs](#graphs_py) |
| פונקציה לשליחת אימייל | [emails](#emails_py) |
| פונקציות להערכת זמן ריצה | [mytime](#mytime_py) |

environments.py

מודול המכיל את המחלקות Snake, VecSnake, Snake3, VecSnake3. הסיומת 3 מתייחסת ל[תנועה יחסית](#תנועה_יחסית) (3 פעולות), והמחלקות ללא סיומת מיישמות [תנועה מוחלטת](#תנועה_מוחלטת) (4 פעולות). הקידומת Vec מעידה על כך שזו [סביבה וקטורית](#סביבה_וקטורית). למחלקות מבנה דומה לכן אציג רק את חלקן.

תרשים UML של המחלקות Snake ו-VecSnake (קישורים לפעולות בלחיצה):

|  |
| --- |
| [Snake](#Snake) |
| + fps: int  + rewards: dict  + HEAD: float  + BODY: float  + FOOD: float  + WALL: float  + EMPTY: float  + HEIGHT: int  + WIDTH: int  + state: NDArray  + body: list  + foods: list  + score: int  + total\_reward: int  + episode\_length: int  + loop\_threshold: int  + loop\_counter: int  + is\_render: bool  + WINDOW\_SIZE: int  + clock: Clock  + screen: Surface |
| [+ **\_\_init\_\_**(shape: tuple, loop\_threshold: int = -1, render: bool = True)](#Snake_init)  [+ reset(start\_length: int = 4, foods: int = 1): NDArray](#Snake_reset)  [+ step(action: int): tuple](#Snake_step)  [+ render(): None](#Snake_render)  [+ close(): None](#Snake_close)  [+ generate\_food(num\_foods: int = 1): None](#Snake_generate_food) |

|  |
| --- |
| [VecSnake](#VecSnake_init) |
| + size: int  + envs: list  + continuous: bool  + active\_envs: list |
| [+ **\_\_init\_\_**(shape: tuple, loop\_threshold: int=-1, continuous: bool = True, render: bool=True)](#VecSnake_init)  [+ reset\_one(index: int, start\_length: int, foods: int): NDArray](#VecSnake_reset_one)  [+ reset\_all(start\_lengths: list, foods: list): NDArray](#VecSnake_reset_all)  [+ close\_one(index): None](#VecSnake_close_one)  [+ close\_all(): None](#VecSnake_close_all)  [+ get\_states(): NDArray](#VecSnake_get_states)  [+ scores(index: int | None = None): list | int](#VecSnake_scores)  [+ total\_rewards(index: int | None = None): list | int](#VecSnake_total_rewards)  [+ episode\_lengths(index: int | None = None): list | int](#VecSnake_episode_lengths)  [+ step(actions: list): tuple](#VecSnake_step) |

ייבוא ספריות לקובץ environments:

|  |
| --- |
| import random import numpy as np from numpy.typing import NDArray import os  os.environ['PYGAME\_HIDE\_SUPPORT\_PROMPT'] = '1' # Hide pygame's welcome output import pygame |

Snake – הגדרת המחלקה ותכונות סטטיות:

|  |
| --- |
| class Snake:  fps = **30** # fps limit; only relevant when rendering  rewards = {'food': **15,** 'hit wall': -**10,** 'hit body': -**10,** 'survive': **0,** 'move closer': **1,** 'move further': -**1**}  # move closer and move further rewards only apply if MAX\_FOOD=1  # representations for different tile types:  HEAD = **0.5** BODY = -**0.5** FOOD = **1** WALL = -**1** EMPTY = **0** |

Snake.\_\_init\_\_ – פעולה בונה:

|  |
| --- |
| def \_\_init\_\_(self**,** shape: tuple[int**,** int]**,** loop\_threshold: int = -**1,** render: bool = True):  *"""* ***:param*** *loop\_threshold: negative value = no loop declaration  """* # declaring variables  self.HEIGHT**,** self.WIDTH = shape  self.state: NDArray[float] = np.array([])  self.body: list[tuple[int**,** int]] = []  self.foods: list[tuple[int**,** int]] = []  # metrics  self.score: int = **0** self.total\_reward: int = **0** self.episode\_length: int = **0** # loop detection  self.loop\_threshold: int = loop\_threshold # consecutive moves without food before declaring loop  self.loop\_counter: int = **0** # rendering  self.is\_render: bool = render  if self.is\_render:  self.clock = pygame.time.Clock()  self.TILE\_SIZE = **750** // self.HEIGHT  pygame.init()  self.screen = pygame.display.set\_mode((self.HEIGHT \* self.TILE\_SIZE**,** self.WIDTH \* self.TILE\_SIZE))  pygame.display.set\_caption('Snake') |

Snake.reset – אתחול המשחק:

|  |
| --- |
| def reset(self**,** start\_length: int = **4,** foods: int = **1**) -> NDArray[float]:  # Reset variables  self.loop\_counter = **0** self.body = []  self.foods = []  self.score = **0** self.total\_reward = **0** self.episode\_length = **0** self.state = np.zeros(shape=(self.HEIGHT**,** self.WIDTH))  self.state[**0**] += self.WALL  self.state[-**1**] += self.WALL  for i in range(**1,** self.HEIGHT - **1**):  self.state[i][**0**] = self.WALL  self.state[i][-**1**] = self.WALL   # Choose random starting position and direction  start\_pos = None  look\_dir = random.randint(**1, 4**)  match look\_dir:  case **1**: # Up  start\_pos = (random.randint(**2,** self.HEIGHT - start\_length - **2**)**,** random.randint(**2,** self.WIDTH - **3**))  for i in range(**1,** start\_length):  self.state[start\_pos[**0**] + i**,** start\_pos[**1**]] = self.BODY  self.body.insert(**0,** (start\_pos[**0**] + i**,** start\_pos[**1**]))  case **2**: # Right  start\_pos = (random.randint(**2,** self.HEIGHT - **3**)**,** random.randint(start\_length + **1,** self.WIDTH - **2**))  for i in range(**1,** start\_length):  self.state[start\_pos[**0**]**,** start\_pos[**1**] - i] = self.BODY  self.body.insert(**0,** (start\_pos[**0**]**,** start\_pos[**1**] - i))  case **3**: # Down  start\_pos = (random.randint(start\_length + **1,** self.HEIGHT - **2**)**,** random.randint(**2,** self.WIDTH - **3**))  for i in range(**1,** start\_length):  self.state[start\_pos[**0**] - i**,** start\_pos[**1**]] = self.BODY  self.body.insert(**0,** (start\_pos[**0**] - i**,** start\_pos[**1**]))  case **4**: # Left  start\_pos = (random.randint(**2,** self.HEIGHT - **3**)**,** random.randint(**2,** self.WIDTH - start\_length - **2**))  for i in range(**1,** start\_length):  self.state[start\_pos[**0**]**,** start\_pos[**1**] + i] = self.BODY  self.body.insert(**0,** (start\_pos[**0**]**,** start\_pos[**1**] + i))   # Add head  self.body.append(start\_pos)  self.state[start\_pos[**0**]**,** start\_pos[**1**]] = self.HEAD   self.generate\_food(foods)  if self.is\_render: self.render()  return np.copy(self.state) |

Snake.step – צעד בסביבה:

|  |
| --- |
| def step(self**,** action: int) -> tuple[NDArray[float]**,** int**,** bool]:  done = False # Assign default value  # Update the snake's body based on action  match action:  case **0**: # Up  if self.body[-**1**][**0**] - **1** != self.body[-**2**][**0**]:  self.body.append((self.body[-**1**][**0**] - **1,** self.body[-**1**][**1**]))  else:  self.body.append((self.body[-**1**][**0**] + **1,** self.body[-**1**][**1**]))  case **1**: # Right  if self.body[-**1**][**1**] + **1** != self.body[-**2**][**1**]:  self.body.append((self.body[-**1**][**0**]**,** self.body[-**1**][**1**] + **1**))  else:  self.body.append((self.body[-**1**][**0**]**,** self.body[-**1**][**1**] - **1**))  case **2**: # Down  if self.body[-**1**][**0**] + **1** != self.body[-**2**][**0**]:  self.body.append((self.body[-**1**][**0**] + **1,** self.body[-**1**][**1**]))  else:  self.body.append((self.body[-**1**][**0**] - **1,** self.body[-**1**][**1**]))  case **3**: # Left  if self.body[-**1**][**1**] - **1** != self.body[-**2**][**1**]:  self.body.append((self.body[-**1**][**0**]**,** self.body[-**1**][**1**] - **1**))  else:  self.body.append((self.body[-**1**][**0**]**,** self.body[-**1**][**1**] + **1**))   # Determine reward, detect if done, and update state  match self.state[self.body[-**1**][**0**]**,** self.body[-**1**][**1**]]:  case self.EMPTY:  # Check if snake moved closer/further from the food  # Only applies with 1 food on the board  if len(self.foods) == **1**:  diff = (np.sqrt((self.body[-**2**][**0**] - self.foods[**0**][**0**]) \*\* **2** + (self.body[-**2**][**1**] - self.foods[**0**][**1**]) \*\* **2**) - np.sqrt((self.body[-**1**][**0**] - self.foods[**0**][**0**]) \*\* **2** + (self.body[-**1**][**1**] - self.foods[**0**][**1**]) \*\* **2**))  if diff > **0**:  reward = self.rewards['move closer']  elif diff < **0**:  reward = self.rewards['move further']  else:  reward = self.rewards['survive']  else:  reward = self.rewards['survive']  # Update state  self.state[self.body[-**1**][**0**]**,** self.body[-**1**][**1**]] = self.HEAD  self.state[self.body[-**2**][**0**]**,** self.body[-**2**][**1**]] = self.BODY  self.state[self.body[**0**][**0**]**,** self.body[**0**][**1**]] = self.EMPTY  self.body.pop(**0**)  # Apply loop detection  self.loop\_counter += **1** if self.loop\_counter == self.loop\_threshold:  done = True  case self.FOOD:  self.loop\_counter = **0** # Reset loop detection  reward = self.rewards['food']  # Update state  self.state[self.body[-**1**][**0**]**,** self.body[-**1**][**1**]] = self.HEAD  self.state[self.body[-**2**][**0**]**,** self.body[-**2**][**1**]] = self.BODY  # Update foods  self.foods.remove(self.body[-**1**])  self.generate\_food(**1**)  self.score += **1** case self.BODY:  reward = self.rewards['hit body']  done = True  case self.WALL:  reward = self.rewards['hit wall']  done = True  case \_:  raise ValueError('Invalid tile value reached')   # Update metrics  self.episode\_length += **1** self.total\_reward += reward  # Update animation  if self.is\_render:  self.render()  self.clock.tick(self.fps)  return np.copy(self.state)**,** reward**,** done # return feedback |

Snake.render – עדכון האנימציה:

|  |
| --- |
| def render(self):  # Iterate over tiles  for y in range(self.HEIGHT):  for x in range(self.WIDTH):  rect = pygame.Rect(x \* self.TILE\_SIZE**,** y \* self.TILE\_SIZE**,** self.TILE\_SIZE**,** self.TILE\_SIZE)  # Draw the tile and it's outline  match self.state[y**,** x]:  case self.EMPTY:  pygame.draw.rect(self.screen**,** (**0, 150, 0**)**,** rect)  pygame.draw.rect(self.screen**,** (**0, 175, 0**)**,** rect**, 1**)  case self.WALL:  pygame.draw.rect(self.screen**,** (**128, 128, 128**)**,** rect)  case self.BODY:  pygame.draw.rect(self.screen**,** (**0, 0, 125**)**,** rect)  pygame.draw.rect(self.screen**,** (**0, 0, 0**)**,** rect**, 1**)  case self.HEAD:  pygame.draw.rect(self.screen**,** (**0, 0, 255**)**,** rect)  pygame.draw.rect(self.screen**,** (**0, 0, 0**)**,** rect**, 1**)  case self.FOOD:  pygame.draw.rect(self.screen**,** (**255, 0, 0**)**,** rect)  pygame.draw.rect(self.screen**,** (**0, 0, 0**)**,** rect**, 1**)  pygame.display.flip() # Update the display |

Snake.close – סגירת האנימציה:

def close(self):  
 if self.is\_render:  
 pygame.quit()

Snake.generate\_food – יצירת אוכל חדש:

|  |
| --- |
| def generate\_food(self**,** num\_foods: int = **1**) -> None:  for i in range(num\_foods): # Repeat for the amount of required foods  # Choose a random empty tile  pos = random.randint(**0,** self.HEIGHT - **1**)**,** random.randint(**0,** self.WIDTH - **1**)  while self.state[pos[**0**]**,** pos[**1**]] != self.EMPTY:  pos = random.randint(**0,** self.HEIGHT - **1**)**,** random.randint(**0,** self.WIDTH - **1**)  # Assign tile as food  self.state[pos[**0**]**,** pos[**1**]] = self.FOOD  self.foods.append(pos) |

VecSnake.\_\_init\_\_ – פעולה בונה:

|  |
| --- |
| class VecSnake:  def \_\_init\_\_(self**,** shape: tuple[int**,** int**,** int]**,** loop\_threshold: int = -**1,** continuous: bool = True**,** render: bool = True):  *"""* ***:param*** *shape: (height, width, envs)* ***:param*** *loop\_threshold: negative value = no loop declaration* ***:param*** *continuous: whether to reset done envs without waiting for all*  *envs* ***:param*** *render:  """* self.size = shape[**2**]  self.envs = [Snake((shape[**0**]**,** shape[**1**])**,** loop\_threshold**,** render) for i in range(self.size)]  self.continuous = continuous  if not self.continuous:  self.active\_envs = self.envs.copy() |

VecSnake.reset\_one – איפוס סביבה אחת:

|  |
| --- |
| def reset\_one(self**,** index: int**,** start\_length: int**,** foods: int) -> NDArray[float]:  return self.envs[index].reset(start\_length**,** foods) |

VecSnake.reset\_all – איפוס כל הסביבות:

|  |
| --- |
| def reset\_all(self**,** start\_lengths: list[int]**,** foods: list[int]) -> NDArray[float]:  return np.array([self.envs[i].reset(start\_lengths[i]**,** foods[i]) for i in range(self.size)]) |

VecSnake.close\_one – סגירת סביבה אחת:

|  |
| --- |
| def close\_one(self**,** index):  if self.continuous:  self.envs[index].close()  else:  self.active\_envs[index].close() |

VecSnake.close\_all – סגירת כל הסביבות:

|  |
| --- |
| def close\_all(self):  for i in range(self.size):  self.envs[i].close() |

VecSnake.get\_states – אחזור המצבים בלוחות:

|  |
| --- |
| def get\_states(self) -> NDArray[float]:  return np.array([np.copy(self.envs[i].state) for i in range(self.size)]) |

VecSnake.scores – אחזור מדד הניקוד:

|  |
| --- |
| def scores(self**,** index: int | None = None) -> list[int] | int:  if index is None:  return [env.score for env in self.envs]  return self.envs[index].score |

VecSnake.total\_rewards – אחזור מדד התגמולים המצטברים:

|  |
| --- |
| def total\_rewards(self**,** index: int | None = None) -> list[int] | int:  if index is None:  return [env.total\_reward for env in self.envs]  return self.envs[index].total\_reward |

VecSnake.episode\_lengths – אחזור מדד אורך המשחק:

|  |
| --- |
| def episode\_lengths(self**,** index: int | None = None) -> list[int] | int:  if index is None:  return [env.episode\_length for env in self.envs]  return self.envs[index].episode\_length |

VecSnake.step – צעד בסביבות:

|  |
| --- |
| def step(self**,** actions: list[int]) -> tuple[list[NDArray[float]]**,** list[int]**,** list[bool]]:  # Initialize feedbacks  new\_states = []  rewards = []  dones = []  # Iterate over all active environments  for i in range(len(actions)):  # Perform step  if self.continuous:  feedback = self.envs[i].step(actions[i])  else:  feedback = self.active\_envs[i].step(actions[i])  # Save feedback  new\_states.append(feedback[**0**])  rewards.append(feedback[**1**])  dones.append(feedback[**2**])   return new\_states**,** rewards**,** dones # Return feedbacks |

agents.py

מודול המכיל את המחלקות [DQNAgent](#DQNAgent_init), [HumanPlayer](#HumanPlayer), [RandomPlayer](#RandomPlayer).

מכיל את הפונקציות [tensor\_reshape](#tensor_reshape), [unpack\_tensor\_reshape](#unpack_tensor_reshape), [mask\_loss](#mask_loss).

תרשים UML של המחלקות (קישורים לפעולות בלחיצה):

|  |
| --- |
| [DQNAgent](#DQNAgent_init) |
| + save\_name: str  + model: keras.Model  + target\_model: keras.Model  + target\_sync\_counter: int  + memory: deque |
| [+ **\_\_init\_\_**(save\_name: str, memory\_size: int, shape: tuple, old\_model=None)](#DQNAgent_init)  [+ create\_model(new\_shape: tuple, old\_model=None, action\_space: int = 4): keras.Model](#DQNAgent_create_model)  [+ fit\_minibatch(minibatch\_size: int, epochs: int, gamma: float, tau: float, target\_sync\_freq: int): None](#DQNAgent_fit_minibatch)  [+ update\_memory(experiences: list): None](#DQNAgent_update_memory)  [+ save(save\_name: str = None): None](#DQNAgent_save) |

|  |
| --- |
| [HumanPlayer](#HumanPlayer) |
| + input\_shape: tuple  + action: int |
| [+ **\_\_init\_\_**(input\_shape: tuple)](#HumanPlayer_init)  [+ predict(state=None, verbose=None): list](#HumanPlayer_predict) |

|  |
| --- |
| [RandomPlayer](#RandomPlayer) |
| + input\_shape: tuple |
| [+ **\_\_init\_\_**(input\_shape: tuple)](#RandomPlayer_init)  [+ predict(current\_state=None, verbose=None): list](#RandomPlayer_predict) |

ייבוא ספריות לקובץ agents:

|  |
| --- |
| import os  os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2' # Reduce tensorflow's verbose os.environ['PYGAME\_HIDE\_SUPPORT\_PROMPT'] = '1' # Hide pygame's welcome output import tensorflow as tf from tensorflow.keras.layers import Input**,** Flatten**,** Dense**,** Conv2D from environments import Snake import numpy as np from numpy.typing import NDArray import random from collections import deque import pygame |

DQNAgent.\_\_init\_\_ – פעולה בונה:

|  |
| --- |
| class DQNAgent:  def \_\_init\_\_(self**,** save\_name: str**,** memory\_size: int**,** shape: tuple[int**,** int]**,** old\_model=None):  # Creation of a new model if it doesn't exist already  if not os.path.exists(save\_name):  self.create\_model(shape**,** old\_model).save(save\_name)  self.save\_name = save\_name  # Load model  self.model = tf.keras.models.load\_model(save\_name)  # Initialize target network  self.target\_model = tf.keras.models.clone\_model(self.model)  self.target\_model.set\_weights(self.model.get\_weights())  self.target\_sync\_counter = **0** # Initialize experience buffer  self.memory = deque(maxlen=memory\_size) |

DQNAgent.create\_model – יצירת מודל חדש:

|  |
| --- |
| def create\_model(self**,** new\_shape: tuple[int**,** int]**,** old\_model=None**,** action\_space: int = **4**):  *"""* ***:param*** *action\_space: Output shape; either 3 or 4* ***:param*** *new\_shape: New model's input shape* ***:param*** *old\_model: If passed, weights will be transferred.* ***:return****: A new model  """* # New model definition. Should be identical to old model except input layer.  inputs = Input(shape=(new\_shape[**0**]**,** new\_shape[**1**]**, 1**))  x = Conv2D(filters=**64,** kernel\_size=(**3, 3**)**,** activation='relu')(inputs)  x = Conv2D(filters=**64,** kernel\_size=(**3, 3**)**,** activation='relu')(x)  x = Flatten()(x)  x = Dense(**128,** activation='relu')(x)  value = Dense(**1,** activation='linear')(x)  advantage = Dense(action\_space**,** activation='linear')(x)  mean\_advantage = tf.keras.ops.mean(advantage**,** axis=**1,** keepdims=True)  q\_values = value + (advantage - mean\_advantage)   new\_model = tf.keras.Model(inputs=inputs**,** outputs=q\_values)   # Transfer weights if given  if old\_model is not None:  for old\_layer**,** new\_layer in zip(old\_model.layers**,** new\_model.layers):  if isinstance(new\_layer**,** Dense):  # Zero padding to the end of the array.  old\_weights**,** old\_biases = old\_layer.get\_weights()  pad\_size = new\_layer.get\_weights()[**0**].shape[**0**] - old\_weights.shape[**0**]  padded\_weights = np.pad(old\_weights**,** ((**0,** pad\_size)**,** (**0, 0**))**,** mode='constant'**,** constant\_values=**0**)  new\_layer.set\_weights([padded\_weights**,** old\_biases])  elif isinstance(new\_layer**,** Conv2D):  # No padding  new\_layer.set\_weights(old\_layer.get\_weights())   new\_model.compile(loss='mean\_squared\_error'**,** optimizer='adam')  return new\_model |

DQNAgent.fit\_minibatch – איטרציית אימון לרשת:

|  |
| --- |
| def fit\_minibatch(self**,** minibatch\_size: int**,** epochs: int**,** gamma: float**,** tau: float**,** target\_sync\_freq: int) -> None:  minibatch = random.sample(self.memory**,** minibatch\_size) # Retrieve experiences  # Predict Q(s, a)  current\_states = tensor\_reshape(np.array([experience[**0**] for experience in minibatch])**,** is\_batch=True)  current\_qs\_minibatch = self.model.predict(current\_states**,** verbose=**0**)  # Predict Q(s', a')  new\_states = tensor\_reshape(np.array([experience[**3**] for experience in minibatch])**,** is\_batch=True)  future\_qs\_minibatch = self.target\_model.predict(new\_states**,** verbose=**0**)  # Initialize data structures  x = []  y = []  # Label data  for index**,** (current\_state**,** action**,** reward**,** new\_state**,** done) in enumerate(minibatch):  # Calculate Q(s, a) using Bellman equation  if done:  new\_q = reward  else:  new\_q = reward + gamma \* np.max(future\_qs\_minibatch[index])  # Combine prediction & calculation to create y labels  current\_qs = current\_qs\_minibatch[index]  current\_qs[action] = new\_q  # Append to training dataset  x.append(current\_state)  y.append(current\_qs)  # Train the model  self.model.fit(np.array(x)**,** np.array(y)**,** epochs=epochs**,** batch\_size=minibatch\_size**,** verbose=**0,** shuffle=False)  # Synchronize target network if needed  self.target\_sync\_counter += **1** if self.target\_sync\_counter >= target\_sync\_freq:  model\_weights = np.array(self.model.get\_weights()**,** dtype=object)  target\_weights = np.array(self.target\_model.get\_weights()**,** dtype=object)  new\_weights = tau \* model\_weights + (**1.0** - tau) \* target\_weights  self.target\_model.set\_weights(new\_weights)  self.target\_sync\_counter = **0** |

DQNAgent.update\_memory – הוספת חוויות ל-[experience buffer](#experience_buffer):

|  |
| --- |
| def update\_memory(self**,** experiences: list[tuple]) -> None:  self.memory.extend(experiences) |

DQNAgent.save – שמירת המודל:

|  |
| --- |
| def save(self**,** save\_name: str = None):  if save\_name is None:  self.model.save(self.save\_name)  else:  self.model.save(save\_name) |

מחלקת HumanPlayer משמשת רק עבור משחקי מבחן של שחקן אנושי, ומממשת חלק מהפעולות של מודל keras בגרסה שמאפשרת קלט אנושי.

HumanPlayer.\_\_init\_\_ – פעולה בונה:

|  |
| --- |
| class HumanPlayer:  def \_\_init\_\_(self**,** input\_shape: tuple[int**,** int**,** int**,** int]):  self.input\_shape = input\_shape  self.action = **0** |

HumanPlayer.predict – קליטת פעולה ממשתמש אנושי:

|  |
| --- |
| def predict(self**,** state=None**,** verbose=None) -> list[NDArray[float]]:  # Receive human input  events = pygame.event.get()  for event in events:  if event.type == pygame.KEYDOWN:  key = event.key  if key == pygame.K\_UP:  self.action = **0** if key == pygame.K\_RIGHT:  self.action = **1** if key == pygame.K\_DOWN:  self.action = **2** if key == pygame.K\_LEFT:  self.action = **3** # Imitate keras output structure  q\_values = np.zeros(**4**)  q\_values[self.action] = **1** return [q\_values] |

מחלקת RandomPlayer זהה ל-HumanPlayer בתפקידה ומימושה, מלבד זאת שהפלט שלה אקראי ולא קלט מהמשתמש.

RandomPlayer.\_\_init\_\_ – פעולה בונה:

|  |
| --- |
| class RandomPlayer:  def \_\_init\_\_(self**,** input\_shape: tuple[int**,** int**,** int**,** int]):  self.input\_shape = input\_shape |

RandomPlayer.predict – יצירת ערכי Q רנדומליים:

|  |
| --- |
| def predict(self**,** current\_state=None**,** verbose=None) -> list[NDArray[float]]:  # Imitate keras output structure with random values, for each parallel game  q\_values = [np.array([random.random() for action in range(**4**)]) for env in range(self.input\_shape[**0**])]  return q\_values |

פעולות עזר בקובץ agents:

tensor\_reshape – הכנת הקלט לרשת:

|  |
| --- |
| def tensor\_reshape(obs: NDArray[float]**,** is\_batch: bool = False) -> NDArray[float]:  *"""   Prepares observations for .predict* ***:param*** *obs: A single state or a batch of states* ***:param*** *is\_batch: Whether a batch or a single state were given* ***:return****: Observations ready for .predict  """* if is\_batch:  return obs.reshape((obs.shape[**0**]**,** obs.shape[**1**]**,** obs.shape[**2**]**, 1**)) # vectorized reshape  return obs.reshape((**1,** obs.shape[**0**]**,** obs.shape[**1**]**, 1**)) # single state reshape |

unpack\_tensor\_reshape – ההמרה ההפוכה מ-tensor\_reshape:

|  |
| --- |
| def unpack\_tensor\_reshape(obs: NDArray[float]**,** is\_batch: bool = False) -> NDArray[float]:  *"""   Undoes the function tensor\_reshape* ***:param*** *obs: A single state or a batch of states* ***:param*** *is\_batch: Whether a batch or a single state were given* ***:return****: Observations in their regular form  """* if is\_batch:  return obs.reshape((obs.shape[**0**]**,** obs.shape[**1**]**,** obs.shape[**2**])) # vectorized reshape  return obs.reshape((obs.shape[**1**]**,** obs.shape[**2**])) # single state reshape |

mask\_loss – מניעת פסילה מיידית (בפעולה הבאה בלבד):

|  |
| --- |
| def mask\_loss(state: NDArray[float]**,** body: list**,** qs: NDArray[float] | list[float]):  # Prepare variables  head\_i**,** head\_j = body[-**1**]  qs = qs[**0**]  old\_action = np.argmax(qs)  mask = min(qs) - **1** # It's impossible to move backwards, and trying so results in forward movement.  # Therefore, we mask every direction with an obstacle, unless the snake came from it:  # Up  if state[head\_i - **1,** head\_j] == Snake.BODY or state[head\_i - **1,** head\_j] == Snake.WALL and head\_i - **1** != body[-**2**][**0**]:  qs[**0**] = mask  # Right  if state[head\_i**,** head\_j + **1**] == Snake.BODY or state[head\_i**,** head\_j + **1**] == Snake.WALL and head\_j + **1** != body[-**2**][**1**]:  qs[**1**] = mask  # Down  if state[head\_i + **1,** head\_j] == Snake.BODY or state[head\_i + **1,** head\_j] == Snake.WALL and head\_i + **1** != body[-**2**][**0**]:  qs[**2**] = mask  # Left  if state[head\_i**,** head\_j - **1**] == Snake.BODY or state[head\_i**,** head\_j - **1**] == Snake.WALL and head\_j - **1** != body[-**2**][**1**]:  qs[**3**] = mask  # And if there's an obstacle straight ahead, mask the backwards direction as well:  if head\_i - **1** == body[-**2**][**0**] and qs[**2**] == mask: qs[**0**] = mask  if head\_j + **1** == body[-**2**][**1**] and qs[**3**] == mask: qs[**1**] = mask  if head\_i + **1** == body[-**2**][**0**] and qs[**0**] == mask: qs[**2**] = mask  if head\_j - **1** == body[-**2**][**1**] and qs[**1**] == mask: qs[**3**] = mask  # Notify user if masking changed the chosen action  if np.argmax(qs) != old\_action:  print('used mask')  return qs |

evaluation.py

מודול המכיל את הפונקציות [visual\_evaluation](#visual_evaluation), [score\_evaluation](#score_evaluation).

ייבוא ספריות לקובץ:

|  |
| --- |
| import graphs import logs import numpy as np import random from environments import VecSnake**,** Snake**,** Snake3**,** VecSnake3 from agents import tensor\_reshape**,** DQNAgent**,** HumanPlayer**,** mask\_loss from time import sleep import os  os.environ['PYGAME\_HIDE\_SUPPORT\_PROMPT'] = '1' import pygame |

visual\_evaluation – מבחן עם אנימציה. משתמש אנושי יכול לשחק באמצעות פונקציה זו. ללא סביבה וקטורית; המשחקים מתבצעים אחד אחרי השני.

|  |
| --- |
| def visual\_evaluation(model**,** episodes: int = -**1,** loop\_threshold: int = **200,** foods: tuple[int**,** int] = (**1, 1**)**,** start\_lengths: tuple[int**,** int] = (**4, 4**)**,** fps: int = **15,** save\_best: bool = True**,** old\_actions: bool = False**,** mask: bool = False) -> float:  *"""* ***:param*** *model: Either a Q-network or an agent* ***:param*** *episodes: -1 for endless evaluation* ***:param*** *loop\_threshold: Negative value for no loop detection* ***:param*** *foods: (min foods, max foods). A random int in this range will be chosen each episode* ***:param*** *start\_lengths: (min start length, max start length). A random int in this range is chosen each episode* ***:param*** *fps: FPS limit* ***:param*** *save\_best: Whether to save the best game* ***:param*** *old\_actions: True = relative movement, False = absolute movement* ***:param*** *mask: Whether to mask instantly-losing actions* ***:return****: Mean score of the evaluation episodes  """* # Extract network from an agent if given  if isinstance(model**,** DQNAgent):  model = model.model  # Construct the relevant class for the action space  if old\_actions:  env = Snake3((model.input\_shape[**1**]**,** model.input\_shape[**2**])**,** loop\_threshold if not isinstance(model**,** HumanPlayer) else -**1**)  else:  env = Snake((model.input\_shape[**1**]**,** model.input\_shape[**2**])**,** loop\_threshold if not isinstance(model**,** HumanPlayer) else -**1**)  env.fps = fps # Set fps  # Initialize score tracking infrastructure  scores = []  max\_score = **0** # Main game loops  for episode in range(**1,** episodes + **1**):  # Reset env and obtain the initial state  current\_state = env.reset(random.randint(start\_lengths[**0**]**,** start\_lengths[**1**])**,** random.randint(foods[**0**]**,** foods[**1**]))  done = False  if env.is\_render: sleep(**1**) # Wait before starting gameplay  if save\_best: game\_log = [current\_state] # Initialize game recording  # Game loop  while not done:  # Determine action  qs = model.predict(tensor\_reshape(current\_state)**,** verbose=**0**)  if mask: qs = mask\_loss(current\_state**,** env.body**,** qs)  action = np.argmax(qs**,** axis=-**1**)  # Clean user input  if env.is\_render: pygame.event.get()  new\_state**,** reward**,** done = env.step(action) # Perform action  current\_state = new\_state # Update state  if save\_best: game\_log.append(current\_state) # Append game recording  scores.append(env.score) # Save score  print(f'Game {episode}, Score: {scores[-**1**]}')  if save\_best:  game\_log.insert(**0,** env.score) # Add score to the record as metadata  # Save recording if it's a new high score and wasn't done by a human  if not isinstance(model**,** HumanPlayer) and game\_log[**0**] > max\_score:  max\_score = game\_log[**0**]  if not os.path.exists(f'{env.HEIGHT}x{env.WIDTH} best game.pkl') or \  logs.read\_log(f'{env.HEIGHT}x{env.WIDTH} best game')[**0**] < max\_score:  logs.log\_data(game\_log**,** f'{env.HEIGHT}x{env.WIDTH} best game')  print('saved!')  if env.is\_render: sleep(**1**) # Wait on final state   return np.mean(np.array(scores)) |

score\_evaluation – מבחן ללא אנימציה בסביבה וקטורית:

|  |
| --- |
| def score\_evaluation(model**,** episodes: int = **100,** loop\_threshold: int = **200,** foods: tuple[int**,** int] = (**1, 1**)**,** start\_lengths: tuple[int**,** int] = (**4, 4**)**,** save\_best: bool = False**,** graph\_save\_name: str | None = None**,** old\_actions: bool = False) -> float:  *"""* ***:param*** *model: Either a Q-network or an agent* ***:param*** *episodes: All episodes will run in parallel* ***:param*** *loop\_threshold: Negative value for no loop detection* ***:param*** *foods: (min foods, max foods). A random int in this range will be chosen each episode* ***:param*** *start\_lengths: (min start length, max start length). A random int in this range is chosen each episode* ***:param*** *save\_best: Whether to save a recording of the best game* ***:param*** *graph\_save\_name: Score distribution graph won't be saved if None* ***:param*** *old\_actions: True = relative movement, False = absolute movement* ***:return****: Mean score of the evaluation episodes  """* # Extract network from an agent if given  if isinstance(model**,** DQNAgent):  model = model.model  # Construct the relevant class for the action space  if old\_actions:  vec\_env = VecSnake3((model.input\_shape[**1**]**,** model.input\_shape[**2**]**,** episodes)**,** loop\_threshold**,** False**,** False)  else:  vec\_env = VecSnake((model.input\_shape[**1**]**,** model.input\_shape[**2**]**,** episodes)**,** loop\_threshold**,** False**,** False)  # Reset env and obtain the initial states  current\_states = vec\_env.reset\_all([random.randint(start\_lengths[**0**]**,** start\_lengths[**1**]) for i in range(episodes)]**,** [random.randint(foods[**0**]**,** foods[**1**]) for i in range(episodes)])  dones = [None] # Initialize with some value to enter the main loop  # Initialize recording if needed  if save\_best:  game\_logs = [[state] for state in current\_states]  best\_log = [**0**]  # Main loop of all parallel games  while dones:  # Determine and perform actions  actions = np.argmax(model.predict(tensor\_reshape(current\_states**,** is\_batch=True)**,** verbose=**0**)**,** axis=-**1**)  new\_states**,** rewards**,** dones = vec\_env.step(actions)  # Remove done environments  for i in range(len(dones) - **1,** -**1,** -**1**):  if dones[i]:  # Keep recording if it's a new high score, else discard it  if save\_best:  if vec\_env.active\_envs[i].score > best\_log[**0**]:  best\_log = [vec\_env.active\_envs[i].score] + game\_logs.pop(i)  else:  game\_logs.pop(i)  dones.pop(i)  new\_states.pop(i)  vec\_env.close\_one(i)  vec\_env.active\_envs.pop(i)   current\_states = np.array(new\_states) # Update states  if save\_best: # Append recordings  for i in range(len(game\_logs)): game\_logs[i].append(current\_states[i])  # Save recording of the best game if it's a new high score  if save\_best:  if not os.path.exists(f'{model.input\_shape[**1**]}x{model.input\_shape[**2**]} best game.pkl') or \  logs.read\_log(f'{model.input\_shape[**1**]}x{model.input\_shape[**2**]} best game')[**0**] < best\_log[**0**]:  logs.log\_data(best\_log**,** f'{model.input\_shape[**1**]}x{model.input\_shape[**2**]} best game')  print(f'saved a game with a score of {best\_log[**0**]}!')  # Save score distribution  if graph\_save\_name is not None:  scores\_dist = np.zeros(max(vec\_env.scores()) + **1**).astype(int)  for score in vec\_env.scores():  scores\_dist[score] += **1** # Smoothen data  moving\_average = **10** smooth\_scores\_dist = np.convolve(scores\_dist**,** np.ones(moving\_average) / moving\_average**,** mode='valid')  # Define graph configuration according to the format  graph\_data = [['score'**,** 'number of games'**,** f'score distribution of {episodes} evaluation episodes ({model.input\_shape[**1**]}x{model.input\_shape[**2**]})']**,** [[x for x in range(len(scores\_dist))]**,** scores\_dist**,** 'Raw'**, 0.2**]**,** [[x for x in range(moving\_average // **2,** len(smooth\_scores\_dist) + moving\_average // **2**)]**,** smooth\_scores\_dist**,** f'Moving average ({moving\_average})']]  # Generate and save graph  graphs.new\_graph(graph\_data**,** False**,** graph\_save\_name)   return np.mean(np.array(vec\_env.scores())) |

logs.py

מודול לשמירת נתוני האימון, עוטף את המודול pickle לשם הנוחות.

כולל את הפונקציות [read\_log](#read_log), [log\_data](#log_data).

ייבוא ספריות לקובץ logs:

|  |
| --- |
| import pickle |

read\_log – קריאה מקובץ:

|  |
| --- |
| def read\_log(log\_name: str):  with open(log\_name + '.pkl'**,** 'rb') as log\_file:  data = pickle.load(log\_file)  return data |

log\_data – שמירה בקובץ:

|  |
| --- |
| def log\_data(data**,** log\_name: str) -> None:  with open(log\_name + '.pkl'**,** 'wb') as log\_file:  pickle.dump(data**,** log\_file) |

graphs.py

מודול לייצור ושמירת גרפים לנתוני האימון, משתמש ב-matplotlib.

כולל את הפונקציות [new\_graph](#new_graph), [smoothen](#smoothen), [plot\_training\_metrics](#plot_training_metrics).

ייבוא ספריות לקובץ graphs:

|  |
| --- |
| import numpy as np from matplotlib import pyplot as plt   # Installing matplotlib on interpreter interferes with docstring display in the IDE, # causing :param & :return to be shown as plain text as well as formatted text. # Deleting matplotlib doesn't help, the only solution is to not install it at first place. # Matplotlib is the most comfortable graphs package I currently know, # but a decent alternative might be worth switching to. |

new\_graph – ייצור ושמירת גרף:

|  |
| --- |
| def new\_graph(data**,** view: bool**,** save\_name: str = None) -> None:  *"""  Required data format:   data = [general labels, line, ..., line]   general labels = [x-axis label, y-axis label, title]  line = [[xs], [ys], label, opacity (optional)]* ***:param*** *data: According to the specified format* ***:param*** *view: whether to display the graph on creation* ***:param*** *save\_name: None = won't save graph. other = will be saved as png with the given name  """* fig**,** ax = plt.subplots() # Create a new plot  # Set texts  labels = data[**0**]  ax.set\_xlabel(labels[**0**])  ax.set\_ylabel(labels[**1**])  ax.set\_title(labels[**2**])  # Add all the lines  for line in data[**1**:]:  if len(line) == **4**:  xs**,** ys**,** label**,** opacity = line  else:  opacity = **1** xs**,** ys**,** label = line  ax.plot(xs**,** ys**,** label=label**,** alpha=opacity)   ax.legend() # Add legend  if save\_name is not None: plt.savefig(save\_name + '.png') # Save if needed  if view: plt.show() # Display if needed  plt.close() |

smoothen – החלקת הנתונים באמצעות ממוצע נע:

|  |
| --- |
| def smoothen(data**,** filter\_size):  *"""* ***:param*** *data: Single dimensional sequence* ***:param*** *filter\_size: Size of Moving average filter* ***:return****:  """* return np.convolve(data**,** np.ones(filter\_size) / filter\_size**,** mode='valid') |

plot\_training\_metrics – ייצור ושמירת גרפים למדדי אימון:

|  |
| --- |
| def plot\_training\_metrics(metrics: list**,** filter\_size: int**,** save\_prefix: str | None = None):  *"""* ***:param*** *metrics: [[name, data], ...]* ***:param*** *filter\_size: Size of Moving average filter* ***:param*** *save\_prefix: Graphs will be saved as {prefix} + {metric} + .png* ***:return****:  """* for metric**,** data in metrics:  episodes = [game for game in range(len(data))] # Generate x values  # Define data according to the format  plot\_data = [['episodes'**,** metric**,** f'{metric} over training episodes']**,** [episodes**,** data**,** 'Raw'**, 0.2**]**,** [episodes[filter\_size - **1**:]**,** smoothen(data**,** filter\_size)**,** f'Moving average ({filter\_size})']]  # Generate graph and save if needed  if save\_prefix is None:  new\_graph(plot\_data**,** True)  else:  new\_graph(plot\_data**,** False**,** save\_prefix + ' ' + metric) |

emails.py

מודול לשליחת הודעות אימייל, משמש לשליחת עדכונים בזמן אימון.

מכיל רק את הפונקציה [send\_message](#send_message).

ייבוא ספריות לקובץ emails:

|  |
| --- |
| from mytime import now # own module import smtplib from email.mime.multipart import MIMEMultipart from email.mime.text import MIMEText from email.mime.base import MIMEBase from email import encoders import os |

send\_message – שליחת הודעת אימייל (השתמשתי במימושים של מקורות [[7]](#reference_7), [[8]](#reference_8)):

|  |
| --- |
| def send\_message(subject: str**,** body: str**,** filenames: list[str] = None):  # Define sender and recipient  receiver\_email = "example@gmail.com"  sender\_email = "example@gmail.com"  sender\_password = "example" # Use an App Password if 2FA is enabled  # Define email message object  msg = MIMEMultipart()  msg['From'] = sender\_email  msg['To'] = receiver\_email  msg['Subject'] = subject  # Attach body  msg.attach(MIMEText(f'{now()}\n\n' + body**,** 'plain'))  # Attach files if given  if filenames is not None:  for filename in filenames:  with open(filename**,** 'rb') as attachment:  part = MIMEBase('application'**,** 'octet-stream')  part.set\_payload(attachment.read())  encoders.encode\_base64(part)  part.add\_header(  'Content-Disposition'**,** f'attachment; filename= {os.path.basename(filename)}'**,** )  msg.attach(part)  # Send email  try:  server = smtplib.SMTP('smtp.gmail.com'**, 587**)  server.starttls()  server.login(sender\_email**,** sender\_password)  server.sendmail(sender\_email**,** receiver\_email**,** msg.as\_string())  server.quit()  print("Message sent successfully!")  except Exception as e:  print(f"Failed to send message: {e}") |

mytime.py

מודול זמן העוטף את המודול datetime. משמש להערכת זמן לסיום אימונים, לחישוב זמן ריצה, ולפלט והודעות בזמן אימון.

מכיל את הפונקציות [now](#now), [add\_time](#add_time), [runtime](#runtime).

ייבוא ספריות לקובץ mytime:

|  |
| --- |
| from datetime import datetime**,** timedelta |

now – תאריך ושעה נוכחיים:

|  |
| --- |
| def now() -> datetime:  *"""* ***:return****: Current datetime rounded-up to the next full second  """* time\_now = datetime.now()  return time\_now + timedelta(microseconds=**1\_000\_000** - time\_now.microsecond) |

add\_time – חישוב חותמת זמן עתידית:

|  |
| --- |
| def add\_time(delta: timedelta) -> datetime:  *"""* ***:param*** *delta: A time period* ***:return****: A future datetime, which is the sum of now and the given time delta  """* return now() + delta |

runtime – חישוב זמן ריצה מחותמת זמן:

|  |
| --- |
| def runtime(start: datetime) -> timedelta:  *"""* ***:param*** *start: A past timestamp* ***:return****: The duration since the given timestamp  """* return now() - start |

training.py

קובץ האימון הראשי המשתמש בכל המודולים שיצרתי.

מכיל את הפונקציות [exploration](#exploration), [update\_epsilon](#update_epsilon), [interact](#interact), [train\_agent](#train_agent), [main](#main).

ייבוא ספריות לקובץ training:

|  |
| --- |
| # Own modules import logs import emails import graphs from environments import VecSnake**,** VecSnake3 from agents import DQNAgent**,** tensor\_reshape from evaluation import score\_evaluation from mytime import now**,** add\_time**,** runtime  import random import numpy as np from numpy.typing import NDArray import os  os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2' # Reduce tensorflow's verbose import tensorflow as tf import multiprocessing as mp # For profiling import cProfile import pstats import io |

הגדרת קונפיגורציה לפרמטרים ופיצ'רים (פירוט ב[ייעול אימונים](#ייעול_אימונים) וב[היפר-פרמטרים וכוונון](#היפר_פרמטרים_וכוונון)):

|  |
| --- |
| profile: bool = True multi\_processing: bool = False # Causes retracing in TensorFlow when pygame is present save\_log: bool = True # Evaluation mean scores save\_metrics: bool = True # Training scores, episode lengths, and total rewards save\_graph: bool = True # Happens anyway if summary email is active verbose: bool = True summary\_email: bool = True live\_emails: bool = True  MODEL\_NAME = '15x15 4' OLD\_MODEL\_NAME = ''  # Parameters of only one model as an example for מדריך למפתח, the code contains more parameters = \  {  '15x15 4': {'OLD\_ACTIONS': False**,** 'HEIGHT': **15,** 'WIDTH': **15,** 'MIN\_FOODS': **1,** 'MAX\_FOODS': **1,** 'MIN\_START\_LENGTH': **4,** 'MAX\_START\_LENGTH': **4,** 'ROUNDS': **100,** 'FITS': **1000,** 'ENVS': **100,** 'FIT\_FREQ': **1,** 'EPOCHS\_PER\_FIT': **1,** 'MINIBATCH\_SIZE': **1000,** 'MEMORY\_SIZE': **1\_000\_000,** 'MIN\_MEMORY\_SIZE': **100\_000,** 'TARGET\_SYNC\_FREQ': **100,** 'TAU': **1,** 'GAMMA': **0.95,** 'EPSILON': **1,** 'EPSILON\_DECAY': **0.999925,** 'MIN\_EPSILON': **0.01,** 'DECAY\_BEFORE\_TRAINING': True}**,** } |

טעינת פרמטרים (פירוט ב[היפר-פרמטרים וכוונון](#היפר_פרמטרים_וכוונון)):

|  |
| --- |
| # Loading of training parameters OLD\_ACTIONS: bool = parameters[MODEL\_NAME]['OLD\_ACTIONS'] HEIGHT: int = parameters[MODEL\_NAME]['HEIGHT'] WIDTH: int = parameters[MODEL\_NAME]['WIDTH'] MIN\_FOODS: int = parameters[MODEL\_NAME]['MIN\_FOODS'] MAX\_FOODS: int = parameters[MODEL\_NAME]['MAX\_FOODS'] MIN\_START\_LENGTH: int = parameters[MODEL\_NAME]['MIN\_START\_LENGTH'] MAX\_START\_LENGTH: int = parameters[MODEL\_NAME]['MAX\_START\_LENGTH'] ROUNDS: int = parameters[MODEL\_NAME]['ROUNDS'] FITS: int = parameters[MODEL\_NAME]['FITS'] # For every .fit, {ENVS \* FIT\_FREQ} steps are done ENVS: int = parameters[MODEL\_NAME]['ENVS'] # Vectorized. Amount of parallel envs FIT\_FREQ: int = parameters[MODEL\_NAME]['FIT\_FREQ'] EPOCHS\_PER\_FIT: int = parameters[MODEL\_NAME]['EPOCHS\_PER\_FIT'] # 1 is probably always the best choice MINIBATCH\_SIZE: int = parameters[MODEL\_NAME]['MINIBATCH\_SIZE'] MEMORY\_SIZE: int = parameters[MODEL\_NAME]['MEMORY\_SIZE'] MIN\_MEMORY\_SIZE: int = parameters[MODEL\_NAME]['MIN\_MEMORY\_SIZE'] TARGET\_SYNC\_FREQ: int = parameters[MODEL\_NAME]['TARGET\_SYNC\_FREQ'] # Amount of .fits between updates TAU: float = parameters[MODEL\_NAME]['TAU'] # 1 = hard update GAMMA: float = parameters[MODEL\_NAME]['GAMMA'] epsilon: float = parameters[MODEL\_NAME]['EPSILON'] EPSILON\_DECAY: float = parameters[MODEL\_NAME]['EPSILON\_DECAY'] MIN\_EPSILON: float = parameters[MODEL\_NAME]['MIN\_EPSILON'] DECAY\_BEFORE\_TRAINING: bool = parameters[MODEL\_NAME]['DECAY\_BEFORE\_TRAINING']  # File management SAVE\_NAME = MODEL\_NAME + '.keras' LOG\_NAME = MODEL\_NAME + ' log'  # Define metrics globally for usage in interact() if os.path.exists(MODEL\_NAME + ' metrics.pkl'):  metrics = logs.read\_log(MODEL\_NAME + ' metrics') else:  metrics = [['score'**,** []]**,** ['total\_reward'**,** []]**,** ['episode\_length'**,** []]] |

exploration – האם לחקור פעולה חדשה:

|  |
| --- |
| def exploration() -> bool:  *"""   Applies epsilon-greedy policy* ***:return****: Whether to explore  """* return random.random() < epsilon |

update\_epsilon – הפחתת אפסילון:

|  |
| --- |
| def update\_epsilon() -> None:  *"""  Applies one exponential decay update to epsilon  """* global epsilon  if epsilon > MIN\_EPSILON:  epsilon \*= EPSILON\_DECAY |

interact – אינטראקציה עם הסביבה (ייצור נתונים):

|  |
| --- |
| def interact(interaction\_model**,** vec\_env: VecSnake**,** steps: int) -> list[  tuple[NDArray[float]**,** int**,** int**,** NDArray[float]**,** bool]]:  *"""* ***:param*** *interaction\_model: A keras neural network* ***:param*** *vec\_env: A vectorized environment with attribute continuous=True* ***:param*** *steps: Amount of interaction steps* ***:return****: A list of experiences  """* experiences = [] # Initialize interaction storage data structure  current\_states = vec\_env.get\_states() # Retrieve initial states  # Interaction loop  for step in range(steps):  qs = interaction\_model.predict(tensor\_reshape(current\_states**,** is\_batch=True)**,** verbose=**0**) # Predict Q values  actions = np.argmax(qs**,** axis=-**1**) # Extract the actions with maximal Q values  actions = [random.randint(**0, 3**) if exploration() else action for action in actions] # Explore  new\_states**,** rewards**,** dones = vec\_env.step(actions) # Perform interaction  experiences += [experience for experience in  zip(current\_states**,** actions**,** rewards**,** new\_states**,** dones**,** strict=True)] # Store experiences  current\_states = np.array(new\_states) # Update states for the next step  # Reset done environments  for i in range(vec\_env.size):  if dones[i]:  # Store game metrics  metrics[**0**][-**1**].append(vec\_env.scores(i))  metrics[**1**][-**1**].append(vec\_env.total\_rewards(i))  metrics[**2**][-**1**].append(vec\_env.episode\_lengths(i))  # Reset  current\_states[i] = vec\_env.reset\_one(i**,** random.randint(MIN\_START\_LENGTH**,** MAX\_START\_LENGTH)**,** random.randint(MIN\_FOODS**,** MAX\_FOODS))  return experiences |

train\_agent – לולאת אימון ראשית:

|  |
| --- |
| def train\_agent() -> None:  # ------------- Initialization Sequence -------------  # Load a trained model for enlargement  if OLD\_MODEL\_NAME != '':  old\_model = tf.keras.models.load\_model(OLD\_MODEL\_NAME + '.keras')  else:  old\_model = None  # Initialize agent  agent = DQNAgent(SAVE\_NAME**,** MEMORY\_SIZE**,** (HEIGHT**,** WIDTH)**,** old\_model)  interaction\_model = tf.keras.models.clone\_model(agent.model)  interaction\_model.set\_weights(agent.model.get\_weights())  # Initialize environment  if OLD\_ACTIONS:  vec\_env = VecSnake3((agent.model.input\_shape[**1**]**,** agent.model.input\_shape[**2**]**,** ENVS)**,** render=False)  else:  vec\_env = VecSnake((agent.model.input\_shape[**1**]**,** agent.model.input\_shape[**2**]**,** ENVS)**,** render=False)  vec\_env.reset\_all([random.randint(MIN\_START\_LENGTH**,** MAX\_START\_LENGTH) for i in range(ENVS)]**,** [random.randint(MIN\_FOODS**,** MAX\_FOODS) for i in range(ENVS)])  # Define evaluation eval\_data structure according to format. Used for human monitoring and logging.  if os.path.exists(LOG\_NAME + '.pkl'): # Continue an existing log  eval\_data = logs.read\_log(LOG\_NAME)  past\_iterations = eval\_data[**1**][**0**][-**1**]  eval\_data[**0**][-**1**] = f'{MODEL\_NAME} evaluation score over {past\_iterations + FITS \* ROUNDS} training iterations'  else: # Create a new log  past\_iterations = **0** if verbose: print(f'{now()} : starts evaluating round 0 / {ROUNDS}')  eval\_data = [['training iterations'**,** 'mean score'**,** f'{MODEL\_NAME} evaluation score over {FITS \* ROUNDS} training iterations']**,** [[**0**]**,** [score\_evaluation(agent.model)]**,** MODEL\_NAME]]  # Gather initial experience  if verbose: print(f'{now()} : starts gathering initial experience')  while len(agent.memory) < MIN\_MEMORY\_SIZE:  experiences = interact(interaction\_model**,** vec\_env**,** FIT\_FREQ) # Generate experiences  agent.update\_memory(experiences) # Append experience buffer  # Decay epsilon if needed  if DECAY\_BEFORE\_TRAINING:  update\_epsilon()  if verbose:  print(f'{now()} : finished gathering initial experience')  print(f'{now()} : starts {FITS \* ROUNDS} training iterations. evaluation every {FITS}')  # Initialize multi-processing if needed  if multi\_processing:  pool = mp.Pool(processes=**1**)   # ------------- Main Training Loop -------------  for r in range(**1,** ROUNDS + **1**):  round\_start = now() # Save start time for runtime estimation calculation  if verbose: print(f'{now()} : starts training round {r} / {ROUNDS}')  # Training iterations  for n in range(**1,** FITS + **1**):  if multi\_processing:  interaction = pool.starmap\_async(interact**,** [  (interaction\_model**,** vec\_env**,** FIT\_FREQ)]) # Start asynchronous interaction (data generation)  agent.fit\_minibatch(MINIBATCH\_SIZE**,** EPOCHS\_PER\_FIT**,** GAMMA**,** TAU**,** TARGET\_SYNC\_FREQ) # Train on previously collected experiences  experiences = interaction.get()[**0**] # Wait for interaction to finish  agent.update\_memory(experiences) # Append experience buffer  else:  experiences = interact(interaction\_model**,** vec\_env**,** FIT\_FREQ) # Generate experiences  agent.update\_memory(experiences) # Append experience buffer  agent.fit\_minibatch(MINIBATCH\_SIZE**,** EPOCHS\_PER\_FIT**,** GAMMA**,** TAU**,** TARGET\_SYNC\_FREQ) # Train  interaction\_model.set\_weights(agent.model.get\_weights()) # Sync models (important for multi-processing)  update\_epsilon() # Decay epsilon to manage exploration/exploitation  # Evaluate and store result  if verbose: print(f'{now()} : starts evaluating round {r} / {ROUNDS}')  eval\_data[**1**][**0**] += [r \* FITS + past\_iterations]  eval\_data[**1**][**1**] += [score\_evaluation(agent**,** old\_actions=OLD\_ACTIONS)]  # Checkpoint  agent.save(f'round {r} ' + SAVE\_NAME) # Save model  if save\_log: logs.log\_data(eval\_data**,** LOG\_NAME) # Save evaluation data  if save\_metrics: logs.log\_data(metrics**,** MODEL\_NAME + ' metrics') # Save training metrics  estimation = add\_time((ROUNDS - r) \* runtime(round\_start)) # Estimate run finish time  if verbose: print(f'{now()} : evaluation result: {eval\_data[**1**][**1**][-**1**]} estimated finish: {estimation}')  if live\_emails: # Send a training status email  graphs.new\_graph(eval\_data**,** False**,** LOG\_NAME)  emails.send\_message(f'round {r}/{ROUNDS} live update - {LOG\_NAME}'**,** f'Estimated finish: {estimation}\nRaw evaluation eval\_data so far:\n{eval\_data}'**,** [LOG\_NAME + '.png'])   # ------------- Finalization Sequence -------------  agent.save(SAVE\_NAME) # Save model  if save\_log: logs.log\_data(eval\_data**,** LOG\_NAME) # Save evaluation data  if save\_metrics: logs.log\_data(metrics**,** MODEL\_NAME + ' metrics') # Save training metrics  if summary\_email:  # Define email content  subject = MODEL\_NAME + ' training summary'  body = (f'Finished training {MODEL\_NAME}\n'  f'Training started at {start} and finished at {now()} with a total runtime of {runtime(start)}\n'  f'This email contains the following attachments:\n'  f'- the trained model\n'  f'- evaluation & training metrics\n'  f'- graphs of the metrics')  attachments = [SAVE\_NAME**,** LOG\_NAME + '.pkl'**,** f'{MODEL\_NAME} metrics.pkl'**,** f'{MODEL\_NAME} log.png'**,** f'{MODEL\_NAME} score.png'**,** f'{MODEL\_NAME} episode\_length.png'**,** f'{MODEL\_NAME} total\_reward.png']  # Smoothen evaluation data  eval\_data[**1**][**2**] = 'Raw'  eval\_data[**1**].append(**0.3**)  moving\_average\_size = **5** eval\_data.append(  [eval\_data[**1**][**0**][moving\_average\_size - **1**:]**,** graphs.smoothen(eval\_data[**1**][**1**]**,** moving\_average\_size)**,** f'Moving average ({moving\_average\_size})'])  # Generate & save graphs  graphs.new\_graph(eval\_data**,** False**,** LOG\_NAME)  graphs.plot\_metrics(metrics**, 1000,** MODEL\_NAME)  # Send email  emails.send\_message(subject**,** body**,** attachments)  elif save\_graph:  # Smoothen evaluation data  eval\_data[**1**][**2**] = 'Raw'  eval\_data[**1**].append(**0.3**)  moving\_average\_size = **5** eval\_data.append(  [eval\_data[**1**][**0**][moving\_average\_size - **1**:]**,** graphs.smoothen(eval\_data[**1**][**1**]**,** moving\_average\_size)**,** f'Moving average ({moving\_average\_size})'])  # Generate & save graphs  graphs.new\_graph(eval\_data**,** False**,** LOG\_NAME)  graphs.plot\_metrics(metrics**, 1000,** MODEL\_NAME)  if verbose: print(f'{now()} : done training') |

main ותחילת הריצה:

|  |
| --- |
| def main():  train\_agent()   if \_\_name\_\_ == '\_\_main\_\_':  start = now() # Save start time for runtime calculation  if profile:  # Profile using [*python's official documentation*](#reference_6)'s code example  profiler = cProfile.Profile()  profiler.enable()  main()  profiler.disable()  s = io.StringIO()  ps = pstats.Stats(profiler**,** stream=s).sort\_stats(pstats.SortKey.TIME)  ps.print\_stats()  # Save  with open(MODEL\_NAME + ' profile'**,** 'w') as save\_file:  save\_file.write(s.getvalue())  save\_file.close()  print(MODEL\_NAME + ' profile saved')  else:  main()   print(f'runtime was {runtime(start)}') # Calculate and output total runtime |

testing.py

קובץ הכולל תוכנית ליישום ומבחן מודלים, ומספק שליטה נוחה על הגדרות רבות.

ייבוא ספריות לקובץ testing:

|  |
| --- |
| from mytime import now**,** runtime import os  os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '2' from tensorflow.keras import models from evaluation import score\_evaluation**,** visual\_evaluation from agents import HumanPlayer**,** RandomPlayer |

שליטה בהגדרות היישום/מבחן:

|  |
| --- |
| # Deployment settings - only modify this segment # ------------------------------------------------------------------------------------ model\_name = '15x15 4' # 'random' for random player, 'human' for human player render = True # If render = False then all episodes will be played in parallel fps = **30** # FPS limit; may run slower for extremely large models episodes = **1000** # Amount of games. -1 for endless evaluation, requires render=True loop\_threshold = -**1** # Negative value for no loop detection foods = (**1, 1**) # A random int in this range is chosen each episode start\_lengths = (**4, 4**) # A random int in this range is chosen each episode save\_best = True # Whether to save the best game old\_actions = False # True = relative movement, False = absolute movement mask = True # Whether to mask instantly-losing actions height**,** width = **16, 16** # Only affects human & random player # ------------------------------------------------------------------------------------ |

טעינת המודל:

|  |
| --- |
| # Load model match model\_name.lower():  case 'random':  model = RandomPlayer((**1,** height**,** width**, 1**))  case 'human':  model = HumanPlayer((**1,** height**,** width**, 1**))  case \_:  model = models.load\_model(model\_name + '.keras') |

יישום:

|  |
| --- |
| # Save timestamp for runtime calculation start\_time = now() # Evaluate if render:  mean\_score = visual\_evaluation(model**,** episodes**,** loop\_threshold**,** foods**,** start\_lengths**,** fps**,** save\_best**,** old\_actions**,** mask) else:  mean\_score = score\_evaluation(model**,** episodes**,** loop\_threshold**,** foods**,** start\_lengths**,** save\_best**,** model\_name + ' distribution'**,** old\_actions) # Summarize print('mean score: '**,** mean\_score) print(f'runtime was {runtime(start\_time)}') |

view game.py

תוכנית לצפייה במשחקים שנשמרו במהלך מבחן.

ייבוא ספריות לקובץ view game:

|  |
| --- |
| import os import time  os.environ['PYGAME\_HIDE\_SUPPORT\_PROMPT'] = '1' # Hide pygame's welcome output import pygame import logs from environments import Snake |

קביעת הגדרות:

|  |
| --- |
| # Only modify these 2 lines: fps = **20** height**,** width = **15, 15** |

תוכנית לצפייה במשחק:

|  |
| --- |
| # Load game recording game = logs.read\_log(f'{height}x{width} best game') print('score:'**,** game[**0**]) print('shape:'**,** game[**1**].shape) # Initialize variables env = Snake(game[**1**].shape) clock = pygame.time.Clock() flag = True # Iterate over frames for frame in game[**1**:]:  env.state = frame  env.render()  clock.tick(fps)  pygame.event.get() # Clean user input  # Wait on first frame before starting  if flag:  time.sleep(**2**)  flag = False time.sleep(**3**) # Wait on final frame |

מדריך למשתמש

כפי שכבר ציינתי, קהל היעד של הפרויקט הוא בעיקר מפתחים ועבורם קיים [מדריך למפתח](#מדריך_למפתח). עם זאת, גם אנשים שאינם מפתחים יכולים להפעיל את המודלים המאומנים שישחקו את המשחק, ואפילו לשחק בעצמם במימוש שלי למשחק Snake. לאחר הורדת קבצי הפרויקט והספריות המופיעות במדריך למפתח, ניתן לערוך ולהריץ אותם בכל כלי שתמצאו לנכון, כגון PyCharm, Visual Studio Code, Notepad…

הקבצים לשימוש הם [testing.py](#testing_py) ו-[view game.py](#view_game_py).

בקובץ testing.py ניתן להפעיל מודלים שמורים ולצפות בהם משחקים, או לבחון אותם ללא אנימציה ולראות את התוצאות. נוסף לכך, ניתן לשחק באמצעות החיצים במקלדת כאשר שם המודל הוא 'human'. להסבר קצר על המשחק, ראה [מבוא](#מבוא). ניתן לשנות את ההגדרות בחלק זה של הקובץ בלבד:

|  |
| --- |
| # Deployment settings - only modify this segment # ------------------------------------------------------------------------------------ model\_name = '15x15 4' # 'random' for random player, 'human' for human player render = True # If render = False then all episodes will be played in parallel fps = **30** # FPS limit; may run slower for extremely large models episodes = **1000** # Amount of games. -1 for endless evaluation, requires render=True loop\_threshold = -**1** # Negative value for no loop detection foods = (**1, 1**) # A random int in this range is chosen each episode start\_lengths = (**4, 4**) # A random int in this range is chosen each episode save\_best = True # Whether to save the best game old\_actions = False # True = relative movement, False = absolute movement mask = True # Whether to mask instantly-losing actions height**,** width = **16, 16** # Only affects human & random player # ------------------------------------------------------------------------------------ |

בקובץ view game.py ניתן לצפות מחדש במשחקים שנשמרו מהמבחן בקובץ testing.py. שימו לב: רק המשחק הטוב ביותר נשמר וניתן לצפייה. משחק שיעקוף ניקוד זה בגודל לוח זהה יחליף את המשחק השמור הקודם. ניתן לשלוט בגודל הלוח בהקלטה ובמהירות הניגון (fps), ע"י שינוי השורות הבאות:

|  |
| --- |
| # Only modify these 2 lines: fps = **20** height**,** width = **15, 15** |

רפלקציה

נחשפתי לתחום Reinforcement Learning במקרה בסרטון ביוטיוב ומאוד התחברתי לשיטה, אז התחלתי להתעניין ולקרוא עוד על הנושא. השלמתי את סדרת הקורסים [ML Specialization](#reference_10) ב-Coursera (שכללו חלק על RL). המשכתי להעמיק בלמידה ממקורות שונים באינטרנט, והתמקדתי באלגוריתם DQN שהתחברתי אליו, במיוחד בשל הדמיון הרב בינו לבין למידה של בני אדם ובעלי חיים.

מאוד נהניתי ללמוד את החומר התיאורטי וליישם אותו, למרות כל האתגרים בהם נתקלתי והבאגים שנאבקתי לפתור, בין אם בפיתוח, הבנה ומימוש של אלגוריתמים, ובין אם בהתקנת ספריות, התמודדות עם ביצועים נמוכים, שינויים ותוספות לחלקים שמימשתי בעבר, תכנון המערכת וכו'. זהו פרויקט התוכנה בעל ההיקף הרחב ביותר שאי פעם עשיתי, ועבדתי עליו באופן עצמאי לחלוטין, לכן העבודה כללה היתקלות בקשיים גם מחוץ לעולם למידת המכונה באופן ספציפי – את החשיבות של סדר, תיעוד, חשיבה מערכתית, ומשוב עצמי שוטף נאלצתי להבין בדרך הקשה כשהייתי כבר עמוק בתוך הפרויקט. נוסף לכך, למדתי את החשיבות של סקירת הפתרונות הקיימים לפני שניגשים לפתור בעיה. מצאתי את עצמי משקיע זמן רב בחשיבה על פתרונות ושיטות, ואחר כך מגלה שהפתרון שחשבתי עליו כבר קיים ומקובל. זוהי מהות ההתנסות, אך אם לא מגבילים אותה היא באה על חשבון היקף הפרויקט וקצב ההתקדמות.

התשוקה שלי לתחום הפכה את העבודה על הפרויקט לחוויה מיוחדת במינה, ופעמים רבות מצאתי את עצמי יושב ועובד ימים שלמים בלי הפסקה, ולא מצליח לעצור וללכת לישון גם כשכבר לילה. אין לי ספק שההתנסות בסגנון עבודה זה, והכישורים שפיתחתי לאורך הדרך ישרתו אותי בהמשך חיי. למעשה, הפרויקט תרם לי רבות כבר בשלב זה – מצאתי את עצמי מספר עליו באריכות בראיונות למסלולים בצבא, ואני בטוח שהדבר תרם רבות לכך שבסופו של דבר התקבלתי.

אם הייתי מתחיל לעבוד על הפרויקט היום, הייתי משנה בעיקר את אופן הפיתוח למטרה ארוכת טווח – הייתי קודם כל מנסה להבין לאיזה סדר גודל אני מתכוון להגיע, ומתאים את הקוד שאני כותב להיות עמיד, מובן ומסודר בשביל לאפשר התרחבות ושילוב תוספות באופן הקל והמוצלח ביותר שניתן.

אשמח מאוד לראות המשך עבודה ומחקר בתחום, אפילו המשך של הפרויקט שלי. במיוחד לגבי הגדלת מודלים מאומנים, הייתי רוצה לראות אם באמת ניתן לחזות את ההגדלה הכי יעילה על בסיס אימונים מקדימים או שיטות מתמטיות. כזכור מהניסיונות שלי לעשות זאת ב[אימון מקדים](#אימון_מקדים), אין לי את משאבי המחשוב הדרושים לכך. בנוסף, בהמשך הפיתוח הייתי רוצה להוסיף ולבדוק פיצ'רים נוספים של DQN כגון prioritized experience replay, multi-step returns, distributional RL, multi-agent environments.

ביבליוגרפיה

**מאמרים:**

[1] Abraham, D. (2023). [Deep reinforcement learning for Snake: How well does a deep Q-network model for classic Snake perform with slight variations in the game rules?](https://pure.tue.nl/ws/portalfiles/portal/307051253/thesis_BDS_Juli_D._Abraham.pdf) [Bachelor’s thesis, Eindhoven University of Technology & Tilburg University].

[2] Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., & Silver, D. (2017). [Rainbow: Combining improvements in deep reinforcement learning](https://arxiv.org/abs/1710.02298v1) (arXiv:1710.02298v1). arXiv.

[3] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). [Playing Atari with deep reinforcement learning](https://arxiv.org/abs/1312.5602) (arXiv:1312.5602). arXiv.

[4] Sewak, M. (2019). [Deep Q Network (DQN), Double DQN, and Dueling DQN: A step towards general artificial intelligence. In Deep reinforcement learning](https://www.researchgate.net/publication/334070121_Deep_Q_Network_DQN_Double_DQN_and_Dueling_DQN_A_Step_Towards_General_Artificial_Intelligence) (pp. 95–108). Springer.

[5] Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., & de Freitas, N. (2016). [Dueling network architectures for deep reinforcement learning](https://arxiv.org/abs/1511.06581)(arXiv:1511.06581v3). arXiv.

**מדריכים ואתרים:**

[6] <https://docs.python.org/3/library/profile.html>

[7] <https://mailtrap.io/blog/smtplib/#Send-email-with-attachments>

[8] <https://www.geekslovecoding.com/blog/python-email-sending-tutorial/>

[9] <https://en.wikipedia.org/wiki/Q-learning#Deep_Q-learning>

[10] <https://www.coursera.org/specializations/machine-learning-introduction?utm_medium=sem&utm_source=gg&utm_campaign=b2c_emea_x_coursera_ftcof_courseraplus_cx_dr_bau_gg_sem_bd-ex_s1_en_m_hyb_24-10_x&campaignid=21836581617&adgroupid=351685084750&device=c&keyword=coursera&matchtype=e&network=g&devicemodel=&creativeid=1449957450621&assetgroupid=&targetid=kwd-36262515261&extensionid=&placement=&gad_source=1&gad_campaignid=21836581617&gbraid=0AAAAADdKX6aPSGzZ81mnjZ5VbFAGCbrBw&gclid=Cj0KCQjww-HABhCGARIsALLO6XwCdml_BBy7ItBaWL1lDhhxmtlB_BSW-6c5KdbOVBbpHLEXw3NCs6caAsa9EALw_wcB#courses>

נספחים

הסבר על RL ו-DQN:

<https://www.baeldung.com/cs/q-learning-vs-deep-q-learning-vs-deep-q-network>

העמקה על שדרוגים של DQN:

Sewak, M. (2019). [Deep Q Network (DQN), Double DQN, and Dueling DQN: A step towards general artificial intelligence. In Deep reinforcement learning](https://www.researchgate.net/publication/334070121_Deep_Q_Network_DQN_Double_DQN_and_Dueling_DQN_A_Step_Towards_General_Artificial_Intelligence) (pp. 95–108). Springer.