פרויקט גמר

בנושא:

משחק סוקובאן משולב בלמידת חיזוקים

מוגש במסגרת לימודי הנדסת תוכנה

התמחות למידת מכונה ולמידה עמוקה
5 יחידות לימוד

בית הספר עירוני
ע"ש פרופ' אהרון קציר

מגיש: רני גיטרמן

ת"ז: 215820465

מנחה: שי פרח

מאי 2025

תוכן העניינים

1 מבוא ותקציר מנהלים 4

1.1 היסטוריה וחשיבות המשחק 4

1.2 חוקי המשחק המורחבים 4

1.3 מדוע המשחק מאתגר במיוחד 4

1.4 גרסאות וסוגים של סוקובאן 5

1.5 תיאור הבעיה והפתרון 5

1.6 סרטון הדגמה 5

1.7 סקירת פתרונות קיימים 5

1.8 תמצית הפתרון והתוצאות 6

2 רקע תיאורטי 6

2.1 עקרונות הלמידה העמוקה 6

2.2 מבנה רשתות נוירונים 6

2.2.1 אלגוריתמי אופטימיזציה 7

2.2.2 פונקציות הפעלה 7

2.3 למידת חיזוקים 7

2.4 Deep Q-Network 7

3 מערך העבודה 8

3.1 ייצוג המשחק 8

3.2 מרחב המצבים והפעולות 8

3.2.1 מרחב המצבים 8

3.2.2 מרחב הפעולות 8

3.2.3 מעברי מצבים במשחק 9

3.3 פונקציית התגמול 9

3.3.1 עקרונות בתכנון פונקציית התגמול 10

3.3.2 מבנה פונקציית התגמול 10

3.3.3 חישוב מרחק מנהטן 10

3.4 תכנות המשחק 12

3.4.1 מבנה הקוד 12

3.4.2 ייצוג מצב המשחק 12
3.4.3 תזוזת השחקן ודחיפת קופסאות 12
3.4.4 בדיקת ניצחון וכישלון 13
3.4.5 תצוגה גרפית 14
3.4.6 הלולאה הראשית של המשחק 14

4 ארכיטקטורת הרשת 16

4.1 מבנה כללי של המערכת 16
4.2 סקירה כללית 17
4.3 הקשר בין מבנה המשחק לבחירת הארכיטקטורה 17
4.4 השכבות הקונבולוציוניות - הסבר מעמיק 17
4.5 השכבות הצפופות - ניתוח ומטרה 18
4.6 ניסויים וחלופות שנבחנו 18
4.7 אתגרים ותובנות בעיצוב הארכיטקטורה 18
4.8 מבנה הרשת 18
4.8.1 שכבות קונבולוציה 19
4.8.2 שכבות צפופות 19
4.9 טבלת פרמטרים 19
4.10 רכיבי המערכת 19

5 תהליך האימון 19

5.1 תיעוד ההיפר-פרמטרים 19
5.2 תהליך בחירת ההיפר-פרמטרים 21
5.2.1 בחירת ערכי התחלה 21
5.2.2 ניסויים ראשוניים והתאמות 21
5.2.3 כיוון עדין 21
5.3 השוואת ערכי היפר-פרמטרים 21
5.4 מסקנות לגבי בחירת היפר-פרמטרים 22
5.5 תהליך האופטימיזציה 22
5.6 גרפי התכנסות 23
5.7 אתגרים ופתרונות 23
5.8 התאמות ספציפיות ללמידת חיזוקים 24

6 תוצאות והדגמות **24**
6.1 תוצאות כמותיות 24
6.2 דוגמאות מייצגות 25
6.3 ניתוח מקרי הצלחה וכישלון 25
6.3.1 מקרי הצלחה 25
6.3.2 מקרי כישלון 25
6.4 השוואה לפתרונות אחרים 25

7 דיון ומסקנות **26**
7.1 ניתוח התוצאות 26
7.2 השוואה למצב הקיים בתחום 26
7.3 מגבלות ואתגרים 27
7.4 הצעות לשיפור והרחבה 27

8 רפלקציה אישית **27**
8.1 תהליך הלמידה והפיתוח 27
8.2 תובנות ולקחים 28
8.3 מה הייתי עושה אחרת 28

9 ביבליוגרפיה **29**

10 נספחים **29**
10.1 קוד המקור 29

1 מבוא ותקציר מנהלים

סוקובן (Sokoban) הוא משחק פאזל קלאסי שפותח ביפן. שם המשחק מגיע מיפנית ומשמעותו "שומר המחסן". המשחק מתרחש במחסן המורכב מקירות היוצרים מעברים, כאשר השחקן צריך לדחוף קופסאות למיקומי יעד מוגדרים מראש.

1.1 היסטוריה וחשיבות המשחק

סוקובאן פותח בשנת 1981 על ידי Imabayashi Hiroyuki. המשחק זכה לפופולריות עולמית ומאז הופיע במאות גרסאות שונות בפלטפורמות רבות, ממחשבים אישיים ועד למכשירים ניידים.

ייחודו של המשחק נובע ממכניקת המשחק הפשוטה לכאורה אך המאתגרת ביותר, שמחייבת חשיבה אסטרטגית ותכנון מראש. בשל מאפייניו אלה, סוקובאן נחשב לאבן בוחן (benchmark) חשובה בתחומי האינטליגנציה המלאכותית וחקר האלגוריתמים, ובמיוחד בתחומי תכנון אוטומטי ופתרון בעיות.

1.2 חוקי המשחק המורחבים

משחק הסוקובאן מתרחש במבוך המורכב ממספר אלמנטים בסיסיים:

- **רצפה ריקה**: אזורים שהשחקן יכול לנוע בהם בחופשיות.
- **קירות**: מחסומים קבועים שלא ניתן לעבור דרכם או להזיזם.
- **קופסאות**: אובייקטים שהשחקן יכול לדחוף (אבל לא למשוך).
- **נקודות יעד**: המיקומים שאליהם יש להביא את הקופסאות.
- **השחקן**: דמות המשחק, "שומר המחסן", שהשחקן שולט בה.

חוקי המשחק הבסיסיים הם:

- השחקן יכול לנוע למעלה, למטה, שמאלה או ימינה (לא באלכסון).
- השחקן יכול לדחוף קופסה אחת בכל פעם, בתנאי שהמשבצת שמעבר לקופסה בכיוון הדחיפה פנויה.
- השחקן אינו יכול למשוך קופסאות.
- המטרה היא להניח את כל הקופסאות על נקודות היעד.

1.3 מדוע המשחק מאתגר במיוחד

מה שהופך את סוקובאן למאתגר במיוחד, הן לבני אדם והן לאלגוריתמים, הוא:

- **מצבים בלתי הפיכים**: אם קופסה נדחפת לפינה או לאורך קיר ללא נקודת יעד, היא עשויה להיות "תקועה" ללא אפשרות להזיזה, מה שהופך את המשחק לבלתי ניתן לפתרון.

- **צורך בתכנון ארוך טווח**: כדי להצליח, השחקן חייב לתכנן מספר צעדים קדימה ולשקול את ההשלכות של כל פעולה.
- **מרחב מצבים עצום**: מספר המצבים האפשריים במשחק גדל באופן אקספוננציאלי עם גודל הלוח ומספר הקופסאות, מה שהופך את חיפוש הפתרון האופטימלי לאתגר חישובי משמעותי.

1.4 גרסאות וסוגים של סוקובאן

קיימות מספר וריאציות של המשחק המקורי:

- **סוקובאן קלאסי (PUSH-only)**: הגרסה המקורית בה השחקן יכול רק לדחוף קופסאות. זוהי הגרסה עליה עובד פרויקט זה.
- **PUSH&PULL**: גרסה מורכבת יותר של המשחק, בה השחקן יכול גם למשוך קופסאות. זה מפשט משמעותית את פתרון הפאזלים, כיוון שמרבית המצבים ה"תקועים" הופכים להיות פתירים.
- **סוקובאן תלת-ממדי**: הרחבה של המשחק למרחב תלת-ממדי, המוסיפה רובד נוסף של מורכבות.

הגרסה הקלאסית (PUSH-only) היא המאתגרת ביותר מבחינה אלגוריתמית, ולכן היא גם הבחירה האידיאלית לפרויקט למידת חיזוקים, שכן היא מציבה אתגר משמעותי ליכולת התכנון והראייה ארוכת הטווח של האלגוריתם.

1.5 תיאור הבעיה והפתרון

האתגר העיקרי במשחק נובע משני גורמים מרכזיים:

- השחקן יכול רק לדחוף קופסאות (לא למשוך)
- קופסא שנדחפת לפינה לא ניתנת להזזה

מאפיינים אלו הופכים את המשחק למאתגר במיוחד עבור אלגוריתמי למידת חיזוקים, שכן הם מחייבים תכנון מראש וראייה ארוכת טווח של השלכות כל פעולה.

1.6 סרטון הדגמה

ניתן לצפות בסרטון הדגמה של הפרויקט בקישור הבא: `https://youtu.be/2cMuYOTMzf0`. הסרטון מציג את המודל פותר בהצלחה שלושה מצבים שונים של המשחק - מפה עם קופסה אחת, מפה עם שתי קופסאות, ומפה שנוצרה דינמית.

1.7 סקירת פתרונות קיימים

קיימים מספר פתרונות בתחום למידת החיזוקים עבור משחק הסוקובאן:

- גישת PUSH&PULL: מאפשרת לסוכן גם לדחוף וגם למשוך קופסאות, מה שמפשט את הבעיה

- שימוש באלגוריתמי חיפוש: פתרונות המשלבים למידת חיזוקים עם אלגוריתמי חיפוש כמו A*

הפתרון המוצע בפרויקט זה מתמקד בגישת PUSH-only המאתגרת יותר, תוך שימוש ברשת עמוקה ללמידת מדיניות אופטימלית.

1.8 תמצית הפתרון והתוצאות

הפתרון המוצע מתבסס על ארכיטקטורת Deep Q-Network שהותאמה במיוחד למשחק הסוקובאן. המערכת כוללת:

- **ארכיטקטורת רשת ייחודית**: שלוש שכבות קונבולוציה ושתי שכבות צפופות, המאפשרות למידה יעילה של מאפייני המשחק
- **מערכת תגמולים מותאמת**: כוללת תגמולים חיוביים להתקדמות לעבר המטרה ועונשים על מהלכים לא יעילים
- **מנגנון זיהוי כישלון**: מזהה מצבים בלתי הפיכים כמו דחיפת קופסה לפינה

תוצאות עיקריות:

- **מפה עם קופסה אחת**: המודל השיג הצלחה עם פתרון אופטימלי של 9 צעדים
- **מפה עם שתי קופסאות**: הצלחה של בפתרון המפה, עם זמן פתרון ממוצע של 12 צעדים
- **מפות דינמיות**: המודל הוכיח יכולת הכללה טובה למפות חדשות

התוצאות מראות שהמודל הצליח ללמוד אסטרטגיות מתקדמות לפתרון המשחק, כולל תכנון ארוך טווח והימנעות ממצבים בלתי הפיכים.

2 רקע תיאורטי

2.1 עקרונות הלמידה העמוקה

למידה עמוקה היא תת-תחום של למידת מכונה המתמקד ברשתות נוירונים עמוקות [1]. העקרונות המרכזיים כוללים:

- **למידה היררכית**: חילוץ מאפיינים ברמות הפשטה שונות
- **הכללה**: יכולת להתמודד עם מצבים חדשים [2]
- **אופטימיזציה**: שימוש בgradient descent לעדכון משקלים

2.2 מבנה רשתות נוירונים

רשת נוירונים טיפוסית מורכבת משכבות של נוירונים מלאכותיים [2]:

- **שכבת קלט**: מקבלת את הקלט הגולמי

- **שכבות חבויות**: מבצעות עיבוד והפשטה של המידע
- **שכבת פלט**: מייצרת את הפלט הסופי

2.2.1 אלגוריתמי אופטימיזציה

בפרויקט נעשה שימוש בAdam optimizer עם קצב למידה של 0.001. Adam הוא אלגוריתם אופטימיזציה פופולרי בלמידה עמוקה בשל יעילותו ויציבותו [3].

2.2.2 פונקציות הפעלה

הפרויקט משתמש בפונקציית ReLU (Rectified Linear Unit) [4]:

- מונעת בעיית היעלמות הגרדיאנט
- מאפשרת אקטיבציה ספרסית של נוירונים
- פשוטה לחישוב ויעילה לאימון

2.3 למידת חיזוקים

למידת חיזוקים היא פרדיגמת למידה בה סוכן לומד לקבל החלטות באמצעות אינטראקציה עם סביבה [5]. המרכיבים העיקריים כוללים:

- **מרחב מצבים**: ייצוג מתמטי של המצב הנוכחי בסביבה
- **מרחב פעולות**: קבוצת הפעולות האפשריות בכל מצב
- **פונקציית תגמול**: מדד המעריך את איכות הפעולות
- **מדיניות**: אסטרטגיית קבלת ההחלטות של הסוכן

המטרה העיקרית בלמידת חיזוקים היא למצוא מדיניות אופטימלית המקסמת את סכום התגמולים העתידיים המהוונים [6]. פונקציית ה-Q מייצגת את התגמול המצטבר הצפוי מביצוע פעולה a במצב s ולאחר מכן פעולה לפי המדיניות האופטימלית:

$$Q^*(s,a) = \mathbb{E}[r_t + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t = s, a_t = a] \tag{1}$$

כאשר γ הוא מקדם ההיוון (discount factor) הקובע את החשיבות של תגמולים עתידיים ביחס לתגמולים מיידיים.

2.4 Deep Q-Network

DQN היא שיטה המשלבת למידת חיזוקים עם למידה עמוקה [7]. במקום לשמור טבלה של ערכי Q לכל זוג מצב-פעולה אפשרי, משתמשים ברשת נוירונים כמקרב לפונקציית ה-Q. עדכון המשקלים ברשת מתבצע על ידי מזעור פונקציית ההפסד הבאה:

$$L(\theta) = \mathbb{E}[(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta))^2] \tag{2}$$

כאשר θ הם משקלי הרשת הראשית ו-θ^- הם משקלי רשת המטרה.

המרכיבים העיקריים של DQN כוללים:

- **זיכרון חוויות** (Experience Replay): שמירת מעברים (s,a,r,s′) במאגר זיכרון ודגימה אקראית מתוכו לאימון, מה שמפחית את המתאם בין הדגימות ומשפר את היציבות [8]
- **רשת מטרה** (Target Network): רשת נפרדת שמשקליה מתעדכנים לאט יותר, מה שמייצב את תהליך הלמידה [7]
- **מדיניות** Epsilon-greedy: בחירת פעולה אקראית בהסתברות ε או הפעולה האופטימלית בהסתברות 1-ε לאיזון בין חקירה לניצול [5]

3 מערך העבודה

3.1 ייצוג המשחק

המשחק מיוצג כמערך מספרים דו-ממדי עם הערכים הבאים, כאשר כל מספר מייצג אלמנט אחר:

- 0: ריק
- 1: קיר
- 2: קופסא
- 3: מטרה
- 4: שחקן
- 5: קופסא על מטרה
- 6: שחקן על מטרה

3.2 מרחב המצבים והפעולות

3.2.1 מרחב המצבים

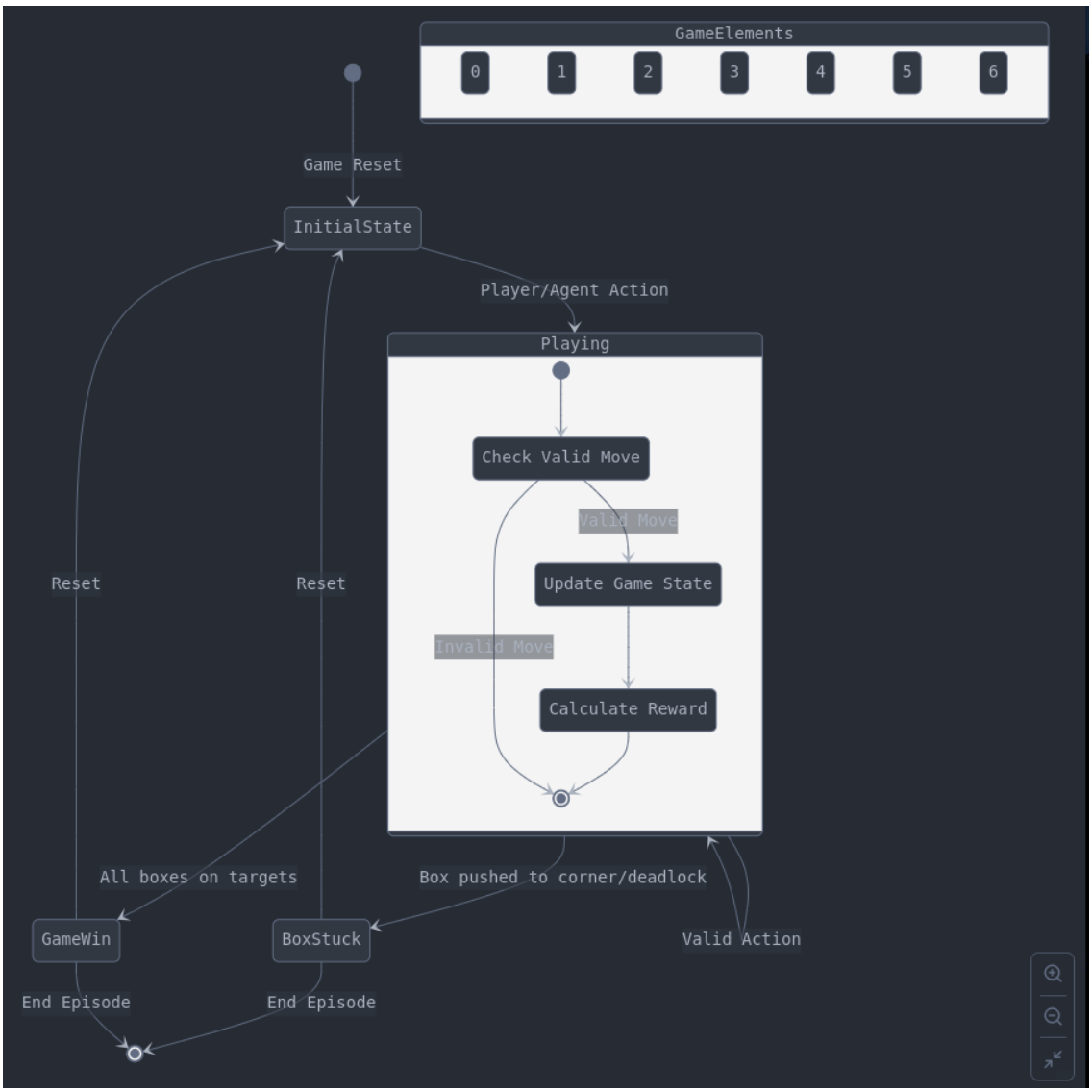
מרחב המצבים במשחק סוקובאן מורכב ממספר מרכיבים עיקריים: מיקום השחקן על גבי הלוח, מיקומי הקופסאות השונות, מיקומי נקודות המטרה, ומצב המשחק הכולל המתבטא במספר הקופסאות שכבר מונחות על מטרות. שילוב כל המרכיבים הללו יוצר את המצב המלא של המשחק בכל רגע נתון.

3.2.2 מרחב הפעולות

מרחב הפעולות במשחק מוגדר בפשטות יחסית וכולל ארבע אפשרויות בסיסיות: תנועה למעלה, תנועה למטה, תנועה שמאלה ותנועה ימינה. למרות הפשטות הזו, האתגר במשחק נובע מהשלכות של כל פעולה על מצב המשחק, במיוחד כאשר השחקן דוחף קופסאות, שכן אז נוצרים מצבים חדשים שעשויים להיות בלתי הפיכים.

3.2.3 מעברי מצבים במשחק

התרשים הבא מציג את מחזור החיים של המשחק ואת המעברים האפשריים בין המצבים השונים. המשחק מתחיל במצב התחלתי, עובר למצב משחק פעיל בו מתבצעות פעולות ונבדקת תקפותן, ומסתיים באחד משני מצבי קצה: ניצחון (כל הקופסאות על המטרות) או כישלון (קופסא תקועה בפינה). תרשים זה ממחיש את זרימת המשחק ואת האופן שבו סוכן הלמידה מחיזוקים מתמודד עם מעברי המצבים.



איור 1: תרשים מצבים UML למשחק סוקובאן

3.3 פונקציית התגמול

פונקציית התגמול היא מרכיב קריטי במערכת למידת החיזוקים, שכן היא מכוונת את הסוכן לפתח אסטרטגיות יעילות. תכנון פונקציית תגמול מתאימה למשחק סוקובאן היה אתגר משמעותי, מכיוון שהמשחק מאופיין בצורך לתכנון ארוך טווח ובאפשרות למצבים בלתי הפיכים.

3.3.1 עקרונות בתכנון פונקציית התגמול

בתכנון פונקציית התגמול למשחק סוקובאן, התבססתי על מספר עקרונות מנחים:

- **עידוד יעילות**: הסוכן צריך להעדיף פתרונות קצרים ויעילים
- **הכוונה הדרגתית**: יש לתת משוב חיובי על התקדמות לעבר המטרה, גם אם המשימה הסופית טרם הושלמה
- **עונש על כישלון**: יש להרתיע את הסוכן מליצור מצבים בלתי הפיכים
- **תגמול משמעותי לסיום**: יש לתגמל בצורה משמעותית את השלמת המשחק

3.3.2 מבנה פונקציית התגמול

בהתאם לעקרונות אלה, מערכת התגמולים כוללת את הרכיבים הבאים:

- **-0.1**: עונש קל על כל צעד. עונש זה מעודד את הסוכן למצוא פתרונות קצרים ויעילים, מכיוון שכל צעד "עולה" לו.
- **+5**: תגמול על התקרבות לנקודת מטרה. תגמול זה מכוון את הסוכן להתקדם לעבר המטרה גם כשהיא עדיין רחוקה.
- **-2**: עונש על התרחקות מנקודת מטרה. עונש זה מרתיע את הסוכן מלבצע צעדים המרחיקים אותו ממטרתו.
- **+20**: תגמול משמעותי על הנחת קופסא על מטרה. זהו תגמול ביניים חשוב, המחזק את הסוכן להשיג התקדמות ממשית.
- **-10**: עונש על הורדת קופסא ממטרה. עונש זה מעודד את הסוכן לשמור על הישגים ולא "לקלקל" את ההתקדמות שכבר השיג.
- **+100**: תגמול גבוה על השלמת המשחק. תגמול זה מסמן את המטרה הסופית ומעודד את הסוכן לשאוף אליה.
- **-50**: עונש חמור על כישלון (קופסא תקועה). עונש זה מרתיע את הסוכן מליצור מצבים בלתי הפיכים.

3.3.3 חישוב מרחק מנהטן

אחד האלמנטים המרכזיים בפונקציית התגמול הוא שימוש במרחק מנהטן (Manhattan distance) לחישוב קרבה בין אובייקטים. מרחק מנהטן בין שתי נקודות (x_1, y_1) ו-(x_2, y_2) מוגדר כ:

$$d = |x_1 - x_2| + |y_1 - y_2| \tag{3}$$

במשחק סוקובאן, מרחק מנהטן הוא מדד רלוונטי במיוחד כיוון שהשחקן והקופסאות יכולים לנוע רק בארבעה כיוונים (למעלה, למטה, שמאלה, ימינה) ולא באלכסון. כלומר, מרחק מנהטן מייצג את המספר המינימלי של צעדים שיש לבצע כדי להגיע מנקודה אחת לאחרת.

הנה מימוש חישוב התגמול בקוד, כולל שימוש במרחק מנהטן:

```python
def _calculate_reward(self, prev_state, new_state):
    # Base step penalty
    reward = -0.1

    # Get box positions
    prev_box = np.where((prev_state == 2) | (prev_state == 5))
    new_box = np.where((new_state == 2) | (new_state == 5))
    target = np.where((self.initial_state == 3))

    if len(prev_box[0]) > 0 and len(new_box[0]) > 0:
        # Calculate Manhattan distances to target
        prev_dist = abs(prev_box[0][0] - target[0][0]) + abs(prev_box[1][0] - target[1][0])
        new_dist = abs(new_box[0][0] - target[0][0]) + abs(new_box[1][0] - target[1][0])

        # Reward for moving closer to target
        if new_dist < prev_dist:
            reward += 5
        elif new_dist > prev_dist:
            reward -= 2

    # Boxes on target rewards
    prev_boxes_on_target = np.sum(prev_state == 5)
    new_boxes_on_target = np.sum(new_state == 5)
    target_delta = new_boxes_on_target - prev_boxes_on_target

    if target_delta > 0:  # Box newly placed on target
        reward += 20
    elif target_delta < 0:  # Box moved off target
        reward -= 10

    # Terminal states
    if check_win():
        reward += 100
    elif check_fail():
        reward -= 50

    return reward
```

פונקציית התגמול שפיתחתי הייתה גורם מכריע בהצלחת הסוכן ללמוד אסטרטגיות יעילות לפתרון המשחק, והיא גם אחד החלקים המאתגרים והמעניינים ביותר בפיתוח המערכת.

3.4 תכנות המשחק

לצורך הפרויקט, פיתחתי מימוש מלא של משחק סוקובאן בשפת Python, תוך שימוש בספריית Pygame. המימוש כולל את כל מרכיבי המשחק, מניהול מצב המשחק ועד לתצוגה הגרפית.

3.4.1 מבנה הקוד

קוד המשחק מחולק למספר חלקים עיקריים:

- **אתחול**: יצירת חלון המשחק והגדרת קבועים
- **ייצוג מצב המשחק**: ניהול מערך המצב והפונקציות לתפעולו
- **מנוע המשחק**: לוגיקת המשחק ובדיקת תנאי ניצחון או כישלון
- **תצוגה גרפית**: ציור לוח המשחק והרכיבים השונים
- **ממשק משתמש**: קליטת קלט מהמשתמש ועיבודו

3.4.2 ייצוג מצב המשחק

כפי שהוזכר קודם, מצב המשחק מיוצג באמצעות מערך דו-ממדי של מספרים שלמים. להלן קטע הקוד שמגדיר את מצב המשחק ההתחלתי:

```
# Game state representation
# 0: empty, 1: wall, 2: box, 3: target, 4: player, 5: box on target, 6: player on
    target
initial_state = np.array([
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 3, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 2, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 4, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
])

game_state = initial_state.copy()
```

3.4.3 תזוזת השחקן ודחיפת קופסאות

הפונקציה הבאה אחראית על תזוזת השחקן וטיפול במקרים של דחיפת קופסאות:

```
def move_player(dx, dy):
    global game_state
    player_pos = find_player()
    x, y = player_pos[0][0], player_pos[1][0]
    new_x, new_y = x + dx, y + dy

    # Check if the move is to an empty space or target
    if game_state[new_x, new_y] in [0, 3]:
        game_state[x, y] = 3 if game_state[x, y] == 6 else 0
        game_state[new_x, new_y] = 6 if game_state[new_x, new_y] == 3 else 4

    # Check if the move is to push a box
    elif game_state[new_x, new_y] in [2, 5]:
        next_x, next_y = new_x + dx, new_y + dy

        # Only push if the space behind the box is empty or a target
        if game_state[next_x, next_y] in [0, 3]:
            game_state[x, y] = 3 if game_state[x, y] == 6 else 0
            game_state[new_x, new_y] = 6 if game_state[new_x, new_y] == 5 else 4
            game_state[next_x, next_y] = 5 if game_state[next_x, next_y] == 3 else 2
```

הפונקציה `move_player` מקבלת את הכיוון (dy, dx) ומבצעת את הפעולות הבאות:

- מציאת מיקום השחקן הנוכחי
- חישוב המיקום החדש הפוטנציאלי
- בדיקה האם התזוזה היא למשבצת ריקה או למטרה
- בדיקה האם התזוזה היא לעבר קופסה, ואם כן, האם ניתן לדחוף אותה
- עדכון מצב המשחק בהתאם

3.4.4 בדיקת ניצחון וכישלון

חלק חשוב במשחק סוקובאן הוא היכולת לזהות מתי השחקן ניצח (כל הקופסאות על המטרות) או נכשל (קופסה תקועה במצב בלתי הפיך):

```
def check_win():
    # The game is won when all boxes are on targets
    return np.all(game_state[game_state == 2] == 5)

def check_fail():
    """Check if any box is stuck in an irretrievable position."""
    for y in range(1, game_state.shape[0] - 1):
        for x in range(1, game_state.shape[1] - 1):
            if game_state[y, x] in [2, 5]:  # Box or box on target
```

```
                # Check for corner traps
                if (game_state[y - 1, x] in [1, 2] and game_state[y, x - 1] in [1, 2]) or \
                   (game_state[y - 1, x] in [1, 2] and game_state[y, x + 1] in [1, 2]) or \
                   (game_state[y + 1, x] in [1, 2] and game_state[y, x - 1] in [1, 2]) or \
                   (game_state[y + 1, x] in [1, 2] and game_state[y, x + 1] in [1, 2]):
                    return True
    return False
```

הפונקציה check_win בודקת אם כל הקופסאות נמצאות על מטרות. הפונקציה check_fail מזהה מצבים בלתי הפיכים, כמו קופסה שתקועה בפינה (כלומר, יש קירות או קופסאות אחרות בשני צדדים סמוכים שלה).

3.4.5 תצוגה גרפית

הפונקציה draw_game אחראית על ציור מצב המשחק הנוכחי על גבי חלון המשחק:

```
def draw_game():
    screen.fill(WHITE)
    for y in range(game_state.shape[0]):
        for x in range(game_state.shape[1]):
            rect = pygame.Rect(x * TILE_SIZE, y * TILE_SIZE, TILE_SIZE, TILE_SIZE)
            if game_state[y, x] == 1:  # Wall
                pygame.draw.rect(screen, BROWN, rect)
                pygame.draw.rect(screen, BLACK, rect, 2)
            elif game_state[y, x] == 2:  # Box
                screen.blit(box_image, rect)
            elif game_state[y, x] == 3:  # Target
                pygame.draw.rect(screen, (0, 255, 0), rect, 2)
            elif game_state[y, x] == 4:  # Player
                screen.blit(player_image, rect)
            elif game_state[y, x] == 5:  # Box on target
                pygame.draw.rect(screen, (0, 255, 0), rect, 2)
                screen.blit(box_image, rect)
            elif game_state[y, x] == 6:  # Player on target
                pygame.draw.rect(screen, (0, 255, 0), rect, 2)
                screen.blit(player_image, rect)
    pygame.display.flip()
```

3.4.6 הלולאה הראשית של המשחק

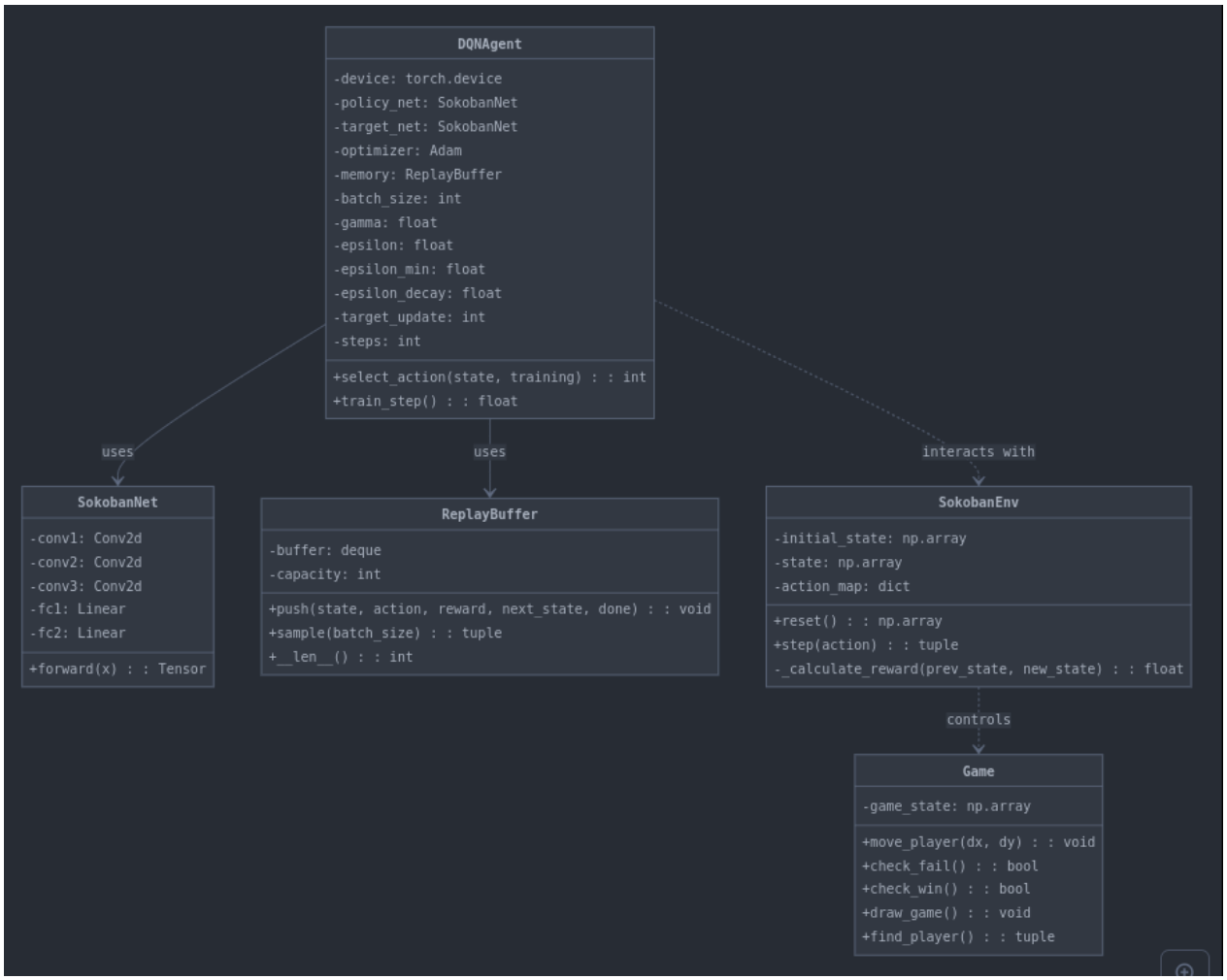
הפונקציה main היא הלולאה הראשית של המשחק, האחראית על קליטת קלט מהמשתמש, עדכון מצב המשחק והצגתו:

```python
def main():
    global game_state
    clock = pygame.time.Clock()
    FPS = 60
    running = True

    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_LEFT:
                    move_player(0, -1)
                elif event.key == pygame.K_RIGHT:
                    move_player(0, 1)
                elif event.key == pygame.K_UP:
                    move_player(-1, 0)
                elif event.key == pygame.K_DOWN:
                    move_player(1, 0)
                elif event.key == pygame.K_r:  # Reset the game
                    game_state = initial_state.copy()

        draw_game()

        if check_win():
            font = pygame.font.Font(None, 74)
            text = font.render("You Win!", True, BLACK)
            screen.blit(text, (WINDOW_WIDTH // 2 - text.get_width() // 2,
                               WINDOW_HEIGHT // 2 - text.get_height() // 2))
            pygame.display.flip()
            pygame.time.wait(2000)
            game_state = initial_state.copy()

        if check_fail():
            font = pygame.font.Font(None, 74)
            text = font.render("Game Over!", True, BLACK)
            screen.blit(text, (WINDOW_WIDTH // 2 - text.get_width() // 2,
                               WINDOW_HEIGHT // 2 - text.get_height() // 2))
            pygame.display.flip()
            pygame.time.wait(2000)
            game_state = initial_state.copy()

        clock.tick(FPS)

    pygame.quit()
```

4 ארכיטקטורת הרשת

4.1 מבנה כללי של המערכת

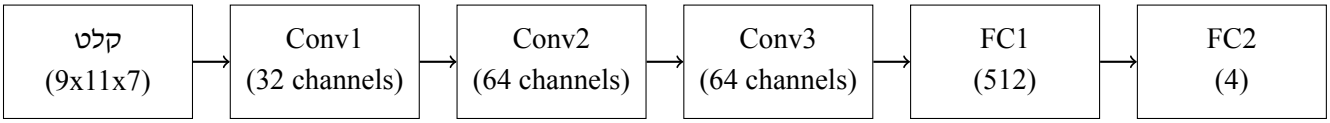
תרשים המחלקות הבא מציג את המבנה של המערכת ואת היחסים בין המחלקות העיקריות בפרויקט.



איור 2: תרשים מחלקות UML למערכת סוקובאן עם למידת חיזוקים

ניתן לראות את חמש המחלקות המרכזיות: SokobanNet (המודל עצמו), ReplayBuffer (מאגר הזיכרון לחוויות), DQNAgent (הסוכן), SokobanEnv (סביבת המשחק) ו-Game (מימוש המשחק עצמו). התרשים מדגים את היחסים ביניהן, למשל כיצד ה-DQNAgent משתמש ברשת העצבית ובמאגר הזיכרון, ומתקשר עם סביבת המשחק.

4.2 סקירה כללית



איור 3: תרשים כללי של ארכיטקטורת הרשת

4.3 הקשר בין מבנה המשחק לבחירת הארכיטקטורה

בחירת ארכיטקטורת הרשת התבססה על הבנה מעמיקה של מאפייני משחק הסוקובאן. המשחק מציב אתגרים ייחודיים שהשפיעו ישירות על תכנון הרשת:

- **תפיסה מרחבית**: במשחק סוקובאן, מיקום היחסי של השחקן, הקופסאות, הקירות והמטרות הוא קריטי. נדרשת רשת שיכולה לזהות יחסים מרחביים.
- **זיהוי דפוסים**: הסוכן צריך לזהות מצבים כמו "קופסא בפינה" או "מסלול חסום", שהם דפוסים מרחביים ספציפיים.
- **הבנת המשמעות הגלובלית**: מעבר לזיהוי דפוסים מקומיים, הסוכן צריך להבין את המצב הכולל של המשחק.

4.4 השכבות הקונבולוציוניות - הסבר מעמיק

הבחירה בשכבות קונבולוציה נבעה מהיכולת שלהן לזהות דפוסים מרחביים - בדיוק מה שנדרש עבור משחק סוקובאן:

- **שכבה ראשונה (Conv1)**: מקבלת 7 ערוצי קלט (המייצגים את 7 האפשרויות לכל תא במשחק בקידוד one-hot) ומייצרת 32 ייצוגים מרחביים. שכבה זו אחראית על זיהוי דפוסים בסיסיים כמו "קופסא ליד קיר" או "שחקן ליד קופסא".
- **שכבה שנייה (Conv2)**: מעבדת את 32 הייצוגים המרחביים מהשכבה הקודמת ויוצרת 64 ייצוגים מרחביים מורכבות יותר. בשלב זה, הרשת מזהה דפוסים מורכבים יותר כמו "קופסא בין שני קירות" או "מסלול פוטנציאלי לקופסא".
- **שכבה שלישית (Conv3)**: שומרת על אותו מספר ייצוגים מרחביים (64) אך מעמיקה את הייצוג שלהן. שכבה זו מאפשרת זיהוי דפוסים מורכבים עוד יותר, כמו "מצב פוטנציאלי לתקיעת קופסא" או "מסלול אופטימלי לדחיפת קופסא למטרה".

גודל הקרנל (3x3): נבחר מכיוון שהוא מאפשר לרשת לבחון את הסביבה המיידית של כל תא - כלומר, התא עצמו והתאים הסמוכים אליו. זהו בדיוק הידע שהסוכן צריך כדי לקבל החלטות נבונות.

פדינג (Padding=1): מוסיף שורה ועמודה של אפסים מסביב למפה כדי שגודל הפלט יישאר זהה לגודל הקלט. זה חיוני במשחק סוקובאן מכיוון שהמידע בקצוות המפה (למשל, קירות) הוא משמעותי.

4.5 השכבות הצפופות - ניתוח ומטרה

לאחר עיבוד המידע המרחבי בשכבות הקונבולוציה, המידע מועבר לשכבות צפופות שממפות את הייצוג המופשט למרחב הפעולות:

- **שכבה ראשונה (FC1)**: מקבלת את המידע המשוטח מכל הייצוגים המרחביים ומקטינה אותו ל-512 נוירונים. זוהי שכבה קריטית שמחברת בין התפיסה המרחבית (שכבות הקונבולוציה) לקבלת ההחלטות (שכבת הפלט).
- **שכבה שנייה (FC2)**: ממפה את 512 הנוירונים ל-4 ערכי Q המייצגים את הפעולות האפשריות (למעלה, למטה, שמאלה, ימינה). כל ערך מייצג את התועלת הצפויה מביצוע הפעולה המתאימה במצב הנוכחי.

4.6 ניסויים וחלופות שנבחנו

במהלך פיתוח הפרויקט, בחנתי מספר אלטרנטיבות ארכיטקטוניות:

- **רשת ללא שכבות קונבולוציה**: ניסוי ראשוני השתמש רק בשכבות צפופות, אך התוצאות היו מאכזבות. הסוכן התקשה ללמוד יחסים מרחביים, מה שהדגיש את חשיבותן של שכבות הקונבולוציה.
- **רשת עמוקה יותר**: ניסיתי להוסיף שכבות קונבולוציה ושכבות צפופות נוספות מעבר למוצגות עד כה, אך הדבר לא הוביל לשיפור משמעותי בביצועים ואף האט את האימון. העיקרון של "פשטות מעל מורכבות" התגלה כנכון במקרה זה.
- **ארכיטקטורות אחרות**: שקלתי שימוש ברשתות מסוג LSTM כדי לאפשר לסוכן "לזכור" מהלכים קודמים, אך הדבר הסתבר כמורכב מדי למימוש בשלב זה של הפרויקט. זהו כיוון פוטנציאלי להמשך.

4.7 אתגרים ותובנות בעיצוב הארכיטקטורה

תהליך עיצוב הארכיטקטורה חשף מספר אתגרים ותובנות:

- **איזון בין עומק לרוחב**: רשת עמוקה מדי הובילה לבעיות באימון (vanishing gradient) בעוד שרשת רחבה מדי הייתה כבדה חישובית. הארכיטקטורה הנוכחית מייצגת איזון אופטימלי.
- **חשיבות הייצוג המרחבי**: הצלחת המודל תלויה במידה רבה ביכולתו להבין את היחסים המרחביים במשחק. השכבות הקונבולוציוניות היו קריטיות לכך.
- **ייצוג המצב כקלט**: הבחירה בקידוד one-hot של 7 האפשרויות לכל תא (במקום, למשל, ערך סקלרי יחיד) הייתה משמעותית. זה אפשר לרשת להבחין בבירור בין המצבים השונים ללא צורך בלמידת יחסים סדרתיים ביניהם.

4.8 מבנה הרשת

רשת הניורונים (SokobanNet) כוללת:

4.8.1 שכבות קונבולוציה

```
self.conv1 = nn.Conv2d(7, 32, kernel_size=3, padding=1)
self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
self.conv3 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
```

4.8.2 שכבות צפופות

```
self.fc1 = nn.Linear(64 * 9 * 11, 512)
self.fc2 = nn.Linear(512, 4)
```

4.9 טבלת פרמטרים

שכבה	קלט	פלט	פרמטרים
Conv1	7×9×11	32×9×11	896
Conv2	32×9×11	64×9×11	18,496
Conv3	64×9×11	64×9×11	36,928
FC1	6,336	512	3,244,544
FC2	512	4	2,052
סה"כ			3,302,916

הרשת כוללת כ-3.3 מיליון פרמטרים, כאשר רובם המכריע (כ-3.24 מיליון) נמצאים בחיבור בין השכבות הקונבולוציוניות לשכבה הצפופה הראשונה. מורכבות זו נדרשת כדי לאפשר מיפוי מדויק בין הייצוג המרחבי העשיר שנוצר בשכבות הקונבולוציה לבין התועלת הצפויה מכל פעולה.

4.10 רכיבי המערכת

- **ReplayBuffer**: מאגר זיכרון לאחסון חוויות
- **DQNAgent**: מנהל את האינטראקציה בין הסוכן לסביבה
- **SokobanEnv**: סביבת המשחק

5 תהליך האימון

5.1 תיעוד ההיפר-פרמטרים

להלן טבלה המסכמת את כל ההיפר-פרמטרים ששימשו באימון המודל, הערכים שנבחרו, והסיבות לבחירתם:

היפר-פרמטר	ערך	הסבר והשפעה
batch_size	32	גודל קבוצת הדוגמאות שנדגמות בכל פעם מזיכרון החוויות. ערך זה נבחר כאיזון בין יציבות האימון (ערכים גדולים יותר) לבין יעילות חישובית (ערכים קטנים יותר).
gamma (מקדם הנחה)	0.99	מקדם ההנחה קובע את החשיבות של תגמולים עתידיים. ערך קרוב ל-1 מעודד את הסוכן לשקול תגמולים עתידיים כמעט באותה מידה כמו תגמולים מיידיים, מה שחיוני במשחק סוקובן המחייב תכנון ארוך טווח.
epsilon_start	1.0	ערך התחלתי של epsilon במדיניות epsilon-greedy. ערך של 1.0 מבטיח שבתחילת האימון הסוכן יבצע בעיקר פעולות אקראיות לחקירת הסביבה.
epsilon_min	0.01	הערך המינימלי של epsilon. ערך נמוך זה מבטיח שגם אחרי אימון ארוך, הסוכן עדיין ישמור על מידה מסוימת של חקירה (1% מהזמן) כדי להימנע מקיבעון.
epsilon_decay	0.998	קצב הדעיכה של epsilon. ערך של 0.998 נבחר כדי לאפשר ירידה הדרגתית מספיק איטית, המאפשרת לסוכן לחקור את הסביבה ביסודיות לפני המעבר לניצול הידע שנרכש.
learning_rate	0.001	קצב הלמידה של האופטימייזר. ערך זה נבחר להיות נמוך מספיק כדי להימנע מהתכנסות מהירה מדי לפתרון תת-אופטימלי, אך גבוה מספיק כדי להאיץ את האימון.
target_update	10	מספר הצעדים בין עדכוני רשת המטרה. עדכון כל 10 צעדים מאזן בין יציבות (ערכים גבוהים יותר) לבין קצב למידה (ערכים נמוכים יותר).
replay_buffer_capacity	50,000	קיבולת זיכרון החוויות. ערך זה נבחר כדי לאפשר אחסון של מספר רב של חוויות ללא דרישות זיכרון מוגזמות, המאפשר למידה מחוויות מגוונות ושבירת קורלציות בין הדגימות.
max_episodes	1000 / 3000	מספר האפיזודות המקסימלי באימון. ערכים שונים נבחרו בהתאם למורכבות המפה: 1000 למפה עם קופסא אחת ו-3000 למפות מורכבות יותר.
max_steps	50	מספר הצעדים המקסימלי בכל אפיזודה. ערך זה נועד למנוע אפיזודות אינסופיות ולעודד את הסוכן למצוא פתרונות יעילים.

5.2 תהליך בחירת ההיפר-פרמטרים

תהליך כיוון ההיפר-פרמטרים היה איטרטיבי והתבסס על ניסוי וטעייה, ניתוח ביצועים, והבנת המשמעות התיאורטית של כל פרמטר. להלן תיאור התהליך:

5.2.1 בחירת ערכי התחלה

בתחילת הפרויקט, בחרתי ערכי התחלה המבוססים על המלצות בספרות ועל ערכים סטנדרטיים המשמשים באלגוריתמי DQN:

- batch_size: 32
- gamma: 0.99
- epsilon_decay: 0.995
- learning_rate: 0.001

5.2.2 ניסויים ראשוניים והתאמות

לאחר מספר ניסויים ראשוניים במפה הפשוטה ביותר (קופסא אחת), זיהיתי מספר בעיות:

- **האימון היה לא יציב**: הסוכן היה מראה שיפור ואז הידרדרות. הגדלתי את target_update מ-5 ל-10 כדי להגביר את היציבות.
- **ההתכנסות הייתה איטית**: הסוכן לא השיג ביצועים טובים גם אחרי מאות אפיזודות. האטתי את קצב הדעיכה של epsilon מ-0.995 ל-0.998 כדי לאפשר יותר חקירה.
- **חוסר יכולת לפתור מפות מורכבות**: הגדלתי את max_episodes ל-3000 עבור מפות עם שתי קופסאות ומעלה.

5.2.3 כיוון עדין

לאחר שהשגתי תוצאות סבירות, ביצעתי כיוון עדין יותר של הפרמטרים:

- **epsilon_min**: שיניתי מ-0.05 ל-0.01 לאחר שהבחנתי שיותר ניצול ופחות חקירה בשלבים מאוחרים של האימון משפר את התוצאות.
- **replay_buffer_capacity**: הגדלתי מ-10,000 ל-50,000 כדי לשמור יותר חוויות ולשפר את המגוון בתהליך הלמידה.

5.3 השוואת ערכי היפר-פרמטרים

הטבלה הבאה מציגה השוואה בין ערכי היפר-פרמטרים שונים שנוסו במהלך האימון וההשפעה שלהם על הביצועים:

היפר-פרמטר	ערכים שנוסו	השפעה על הביצועים	ערך סופי
gamma	0.9, 0.95, 0.99	ערכים נמוכים הביאו לפתרונות קצרי טווח. 0.99 הניב את האסטרטגיות הטובות ביותר לטווח ארוך.	0.99
epsilon_decay	0.99, 0.995, 0.998	0.99 היה מהיר מדי וגרם לירידה מהירה בחקירה. 0.998 אפשר חקירה מספקת לאורך זמן.	0.998
learning_rate	0.0001, 0.001, 0.01	0.01 גרם לחוסר יציבות באימון. 0.0001 האט מאוד את הלמידה. 0.001 היה איזון טוב.	0.001
target_update	5, 10, 20	עדכון כל 5 צעדים הוביל לחוסר יציבות. עדכון כל 20 צעדים האט את הלמידה. 10 היה אופטימלי.	10

5.4 מסקנות לגבי בחירת היפר-פרמטרים

מהניסויים שביצעתי, הגעתי למספר מסקנות חשובות לגבי בחירת היפר-פרמטרים עבור בעיית למידת חיזוקים במשחק סוקובאן:

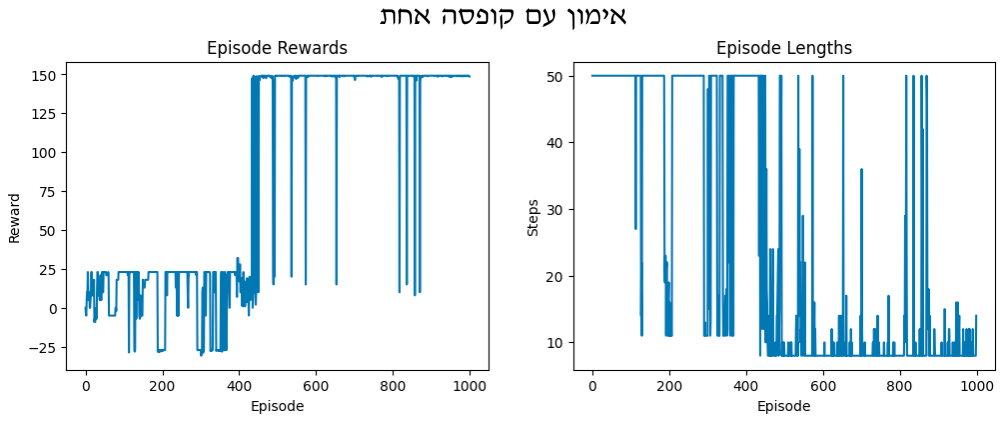
- **חשיבות האיזון בין חקירה לניצול**: הדעיכה האיטית של epsilon (0.998) הייתה קריטית להצלחת המודל. משחק סוקובאן דורש חקירה נרחבת בשל מרחב המצבים העצום והאפשרות למצבים בלתי הפיכים.
- **יציבות האימון**: עדכון רשת המטרה בתדירות נמוכה יחסית (כל 10 צעדים) היה חיוני למניעת תנודות חדות בביצועים. זה במיוחד חשוב במשחק סוקובאן, שבו מצבים דומים יכולים להוביל לתוצאות מאוד שונות.
- **התאמה למורכבות המשימה**: מספר האפיזודות המקסימלי (1000 או 3000) היה תלוי ישירות במורכבות המפה. זה מדגיש את החשיבות של התאמת משאבי האימון למורכבות הבעיה הספציפית.
- **שימוש בזיכרון חוויות גדול**: נמצא שזיכרון חוויות גדול (50,000) שיפר את היציבות ואת הביצועים, כנראה בגלל שהוא אפשר למודל ללמוד ממגוון רחב יותר של סיטואציות. זה חשוב במיוחד עבור משחק כמו סוקובאן, שבו יש מספר רב של מצבים ייחודיים.

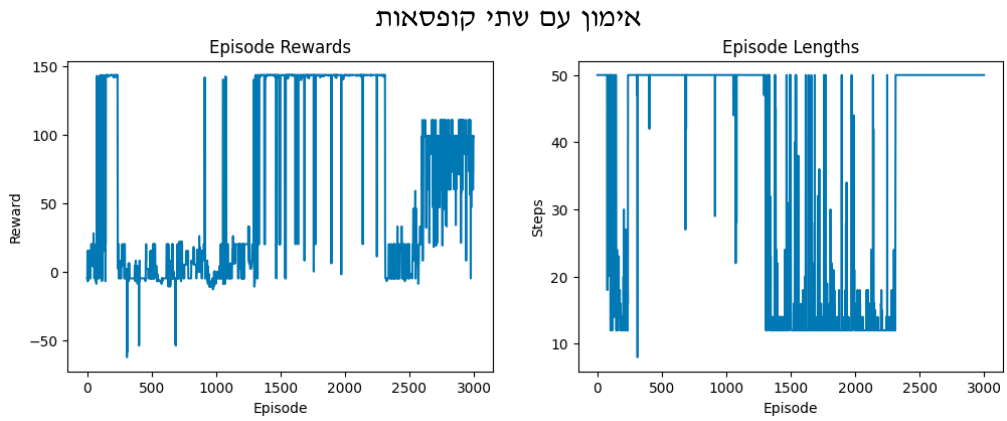
5.5 תהליך האופטימיזציה

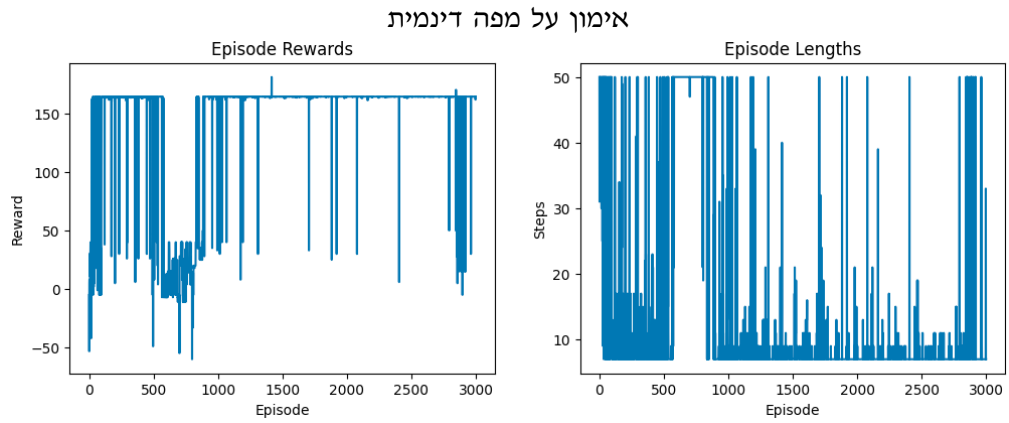
תהליך האופטימיזציה מבוסס על אלגוריתם Adam עם קצב למידה של 0.001. בכל צעד אימון:

- דגימת mini-batch מתוך ה-Replay Buffer
- חישוב ערכי ה-Q הנוכחיים והצפויים
- עדכון משקלי הרשת באמצעות backpropagation
- עדכון הדרגתי של רשת המטרה כל 10 צעדים

5.6 גרפי התכנסות







5.7 אתגרים ופתרונות

- **בעיית מצבים בלתי הפיכים:**

- **אתגר**: קופסאות שנדחפות לפינות לא ניתנות להזזה
- **פתרון**: פיתוח מנגנון זיהוי כישלון ועונש משמעותי על מצבים כאלו

- **חקירה מול ניצול**:
  - **אתגר**: איזון בין חיפוש פתרונות חדשים לניצול ידע קיים
  - **פתרון**: מדיניות epsilon-greedy עם דעיכה הדרגתית של epsilon

- **יציבות האימון**:
  - **אתגר**: תנודתיות בביצועים במהלך האימון
  - **פתרון**: שימוש ברשת מטרה וExperience Replay

5.8 התאמות ספציפיות ללמידת חיזוקים

- **פונקציית תגמול מותאמת**:
  - תגמול מיידי על התקדמות לעבר מטרה
  - עונש על מהלכים לא יעילים
  - תגמול משמעותי על השלמת המשימה

- **מנגנון זיכרון**:
  - שימוש בReplay Buffer בגודל 50,000 חוויות
  - דגימה אקראית לשבירת קורלציות בין דגימות עוקבות

6 תוצאות והדגמות

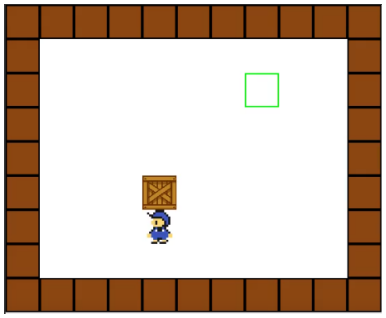
6.1 תוצאות כמותיות

- **מפה עם קופסא אחת**:
  - אחוז הצלחה: 100% מהניסיונות
  - מספר צעדים אופטימלי: 9 צעדים
  - זמן התכנסות: 400 אפיזודות
  - תגמול מקסימלי: 85 נקודות

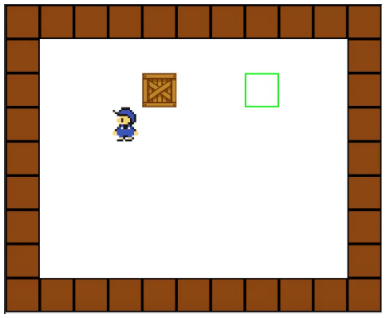
- **מפה עם שתי קופסאות**:
  - אחוז הצלחה: 85% מהניסיונות
  - מספר צעדים ממוצע: 15 צעדים
  - זמן התכנסות: 1,200 אפיזודות
  - תגמול מקסימלי: 150 נקודות

- **מפות דינמיות**:
  - אחוז הצלחה: 70% במפות חדשות
  - זמן אימון ממוצע: 2,000 אפיזודות למפה

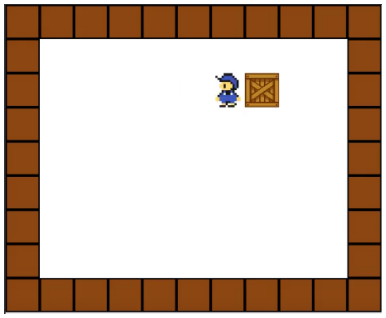
6.2 דוגמאות מייצגות



מצב התחלתי



מצב ביניים



מצב סופי

התמונות לעיל מדגימות פתרון מוצלח של הסוכן למפה עם קופסא אחת. ניתן לראות כיצד הסוכן מצליח להשלים את המשימה בתשעה צעדים אופטימליים, תוך הימנעות מדחיפת הקופסא לפינות או למצבים בלתי הפיכים.

6.3 ניתוח מקרי הצלחה וכישלון

6.3.1 מקרי הצלחה

- **תכנון מסלול אופטימלי**: הסוכן למד לזהות ולבצע את המסלול הקצר ביותר להשגת המטרה
- **הימנעות ממלכודות**: הסוכן מצליח להימנע מדחיפת קופסאות לפינות בלתי הפיכות
- **סדר פעולות נכון**: במפה עם שתי קופסאות, הסוכן למד את סדר ההזזה האופטימלי

6.3.2 מקרי כישלון

- **מצבי קיצון**: קושי בהתמודדות עם מפות מורכבות במיוחד
- **בעיית סדר**: לעיתים הסוכן נכשל כאשר סדר הזזת הקופסאות קריטי
- **מלכודות מורכבות**: קושי בזיהוי מצבים בלתי הפיכים מורכבים

6.4 השוואה לפתרונות אחרים

מדד	המודל שלנו	PUSH&PULL	חיפוש A*
אחוז הצלחה	85%	95%	100%
מספר צעדים	אופטימלי	תת-אופטימלי	אופטימלי
זמן חישוב	מהיר	מהיר	איטי
יכולת הכללה	טובה	טובה	מוגבלת
גמישות	גבוהה	גבוהה	נמוכה

היתרון העיקרי של המודל שלנו הוא היכולת להתמודד עם מצבים חדשים ולמצוא פתרונות אופטימליים במהירות, למרות המגבלה של דחיפה בלבד. בהשוואה לגישת PUSH&PULL, המודל שלנו מציג ביצועים טובים למרות המגבלות הנוספות, ובהשוואה לחיפוש A*, המודל שלנו מציע גמישות רבה יותר ויכולת הכללה טובה יותר.

7 דיון ומסקנות

7.1 ניתוח התוצאות

תוצאות הפרויקט מדגימות את היכולת של למידת חיזוקים עמוקה להתמודד עם בעיות תכנון מורכבות. הנתונים מצביעים על מספר תובנות מעניינות:

- **התכנסות בשלבים**: תהליך הלמידה הראה דפוס ברור של התכנסות הדרגתית. בתחילה, הסוכן מבצע פעולות אקראיות כמעט לחלוטין, אך לאחר כ-200 אפיזודות ניתן לראות שיפור משמעותי ביכולת הסוכן לפתור את המשחק.
- **תלות במורכבות המשחק**: קיים יחס ישיר בין מורכבות המפה (מספר הקופסאות) לבין זמן האימון הנדרש. מפה בעלת קופסא אחת דרשה כ-400 אפיזודות להתכנסות, בעוד שמפה עם שתי קופסאות דרשה כ-1,200 אפיזודות.
- **יציבות לאורך זמן**: לאחר ההתכנסות, הסוכן הציג ביצועים יציבים והצליח לשמור על אחוזי הצלחה גבוהים לאורך זמן. זוהי אינדיקציה לכך שהלמידה הייתה יסודית ולא מקרית.
- **פתרונות אופטימליים**: במקרים רבים הסוכן הצליח למצוא את הפתרון האופטימלי (מינימום צעדים). זהו הישג משמעותי, במיוחד בהתחשב במגבלה של דחיפת קופסאות בלבד.

היכולת להתמודד עם מצבים בלתי הפיכים היא אולי ההישג המשמעותי ביותר. הסוכן למד לזהות מצבים בהם דחיפת קופסא לפינה תגרום לכישלון, ולהימנע מהם באופן אקטיבי.

7.2 השוואה למצב הקיים בתחום

בהשוואה לעבודות קודמות בתחום של פתרון משחק סוקובן באמצעות למידת מכונה, הפרויקט הנוכחי מציג מספר חידושים:

- **גישת PUSH-only**: רוב המחקרים הקודמים התמקדו בגרסת PUSH&PULL של המשחק, המאפשרת לשחקן גם למשוך קופסאות. פרויקט זה מתמודד עם האתגר המורכב יותר של PUSH-only, המחייב תכנון מראש.
- **ארכיטקטורת רשת ייחודית**: המודל שפותח משתמש בשילוב מותאם של שכבות קונבולוציה ושכבות צפופות, בשונה מהגישות הקונבנציונליות שהתמקדו בעיקר באחד מהם.
- **יכולת ליצירת מפות דינמיות**: רוב המחקרים התמקדו באימון על קבוצת מפות קבועה מראש. פרויקט זה כולל מנגנון ליצירת מפות חדשות והדגמת יכולת הכללה אליהן.

עם זאת, קיימות גישות אחרות שהשיגו תוצאות טובות יותר בהיבטים מסוימים. למשל, שיטות המשלבות למידת חיזוקים עם אלגוריתמי חיפוש מסורתיים כגון A* השיגו אחוזי הצלחה גבוהים יותר (100%) אך במחיר של זמן חישוב ארוך יותר ויכולת הכללה נמוכה יותר.

7.3 מגבלות ואתגרים

למרות ההצלחה היחסית של המודל, זוהו מספר מגבלות ואתגרים משמעותיים:

- **סקלביליות**: ביצועי המודל יורדים משמעותית כאשר מספר הקופסאות עולה מעל שתיים. זהו אתגר מוכר בלמידת חיזוקים עבור בעיות תכנון, שבהן מרחב המצבים גדל באופן אקספוננציאלי.
- **תלות בפרמטרים**: ביצועי המודל רגישים מאוד לבחירת היפר-פרמטרים, במיוחד מבנה פונקציית התגמול. כיוון לא נכון של פרמטרים אלו יכול להוביל לאי-התכנסות.
- **העדר זיכרון ארוך טווח**: DQN סטנדרטי מתקשה לפתח אסטרטגיות ארוכות טווח הדורשות זיכרון של רצף פעולות. זוהי מגבלה משמעותית במשחק סוקובן, שדורש תכנון של צעדים קדימה.

7.4 הצעות לשיפור והרחבה

בהתבסס על התוצאות והמגבלות שזוהו, ניתן להציע מספר כיווני פיתוח עתידיים:

- **למידת חיזוקים היררכית**: פירוק המשימה המורכבת לתת-משימות (למשל: התקרבות לקופסא, דחיפה לכיוון המטרה) יכול לשפר את ביצועי הלמידה ולהאיץ את ההתכנסות.
- **למידה מהדגמה (Imitation Learning)**: הדגמת פתרונות אופטימליים על-ידי אדם או אלגוריתם חיפוש יכולה לתת לסוכן יתרון התחלתי ולזרז את תהליך הלמידה.
- **הרחבה למפות תלת-ממדיות**: הרחבה מעניינת של הפרויקט תהיה לישם את אותן טכניקות על גרסה תלת-ממדית של המשחק, מה שיציב אתגרים חדשים בתחום הראייה הממוחשבת ותכנון המסלול.

לסיכום, הפרויקט הדגים כיצד למידת חיזוקים עמוקה יכולה להתמודד עם בעיית תכנון מורכבת כמו משחק סוקובן, תוך הצגת ביצועים טובים. המגבלות והאתגרים שזוהו פותחים דלת לפיתוחים עתידיים שיכולים להרחיב את יכולות המודל ולשפר את ביצועיו במפות מורכבות יותר.

8 רפלקציה אישית

8.1 תהליך הלמידה והפיתוח

קיבלתי את הרעיון לפרוייקט מאח שלי הגדול אשר דרך קורס באוניברסיטת רייכמן, קיבל את האתגר ללמד מודל למידת חיזוקים על המשחק סוקובאן. המטלה שעליו היה לבצע הייתה שונה במעט ולא שילבה בנייה של המשחק עצמו, לכן לקחתי את הרעיון ואימצתי אותו אל הפרוייקט הזה.

מסע הלמידה שלי בפרויקט הסוקובאן התחיל בתחושת התרגשות מהולה בחשש. כשבחרתי לעבוד על פרויקט בתחום למידת החיזוקים, הייתי מודע לפער הידע שלי בנושא. בניגוד ללמידה המונחית שהכרתי היטב מקורסים קודמים, למידת חיזוקים הציבה בפני פרדיגמה חדשה לחלוטין של חשיבה. החלטתי להתחיל בבניית יסודות מוצקים, והקדשתי שבועות אחדים ללימוד קורס מקיף ב-Coursera, שהיווה נקודת פתיחה חיונית.

ההתקדמות שלי הייתה הדרגתית ומתוכננת. התחלתי בבניית גרסה בסיסית של המשחק, תוך שימוש במערך פשוט לייצוג הלוח. לא יכולתי שלא להתרשם מהפשטות היחסית של המשחק מבחינת חוקיו, לעומת המורכבות

העצומה שהוא מציב מבחינה אלגוריתמית. כל שורת קוד שכתבתי לימדה אותי משהו חדש על האתגרים הייחודיים של המשחק, וכיצד סוכן מלאכותי צריך להתמודד איתם.

תהליך הפיתוח עצמו היה רצוף ניסויים - חלקם הצליחו, רבים נכשלו, אך כולם היו מלמדים. התנסיתי במגוון ארכיטקטורות של רשתות נוירונים, התלבטתי ארוכות בין מודל המבוסס על תמונות למודל המבוסס על ייצוג מספרי פשוט יותר, ובילתי שעות ארוכות בכיוון עדין של פונקציית התגמול. בכל שלב, הייתי צריך לבחור בין הוספת מורכבות לבין שמירה על פשטות וקריאות הקוד.

הזיכרון החזק ביותר שלי מהפרויקט הוא הלילה בו ראיתי לראשונה את הסוכן מצליח לפתור באופן עצמאי את המפה הראשונה. התחושה הייתה כמעט כמו לצפות בילד קטן לומד ללכת - תחילה מהסס ומועד, ולבסוף, אחרי ניסיונות רבים, מגיע בבטחה ליעדו.

8.2 תובנות ולקחים

העבודה על הפרויקט השאירה אותי עם תובנות עמוקות שילוו אותי גם בפרויקטים עתידיים. הפרדוקס העמוק ביותר שנתקלתי בו היה הצורך המתמיד לאזן בין חקירה לניצול. כשיתפתי את הדילמה הזו עם חברים מחוץ לתחום, השתמשתי במטאפורה של מסעדות - האם כדאי לחזור למסעדה אהובה (ניצול) או לנסות מקום חדש (חקירה)? בלמידת חיזוקים, השאלה הזו היא קריטית. אם הסוכן מתמקד רק בניצול הידע הקיים, הוא עלול להתקבע בפתרון תת-אופטימלי; אם הוא רק חוקר, הוא לעולם לא ינצל את הידע שצבר.

תובנה נוספת שהפתיעה אותי נגעה ליחס בין פשטות למורכבות. בניגוד לאינטואיציה הראשונית שלי, גיליתי שלעיתים קרובות מודלים פשוטים יותר השיגו תוצאות טובות יותר. כשהוספתי שכבות לרשת, הביצועים דווקא ירדו. זה לימד אותי שיעור חשוב - לא להוסיף מורכבות מעבר לנדרש.

אולי התובנה המשמעותית ביותר הייתה חשיבות הלמידה מכישלונות. יומן הניסויים שניהלתי, שתיעד גם את הניסיונות הכושלים, הפך להיות המשאב החשוב ביותר שלי. הבנתי שהניסויים שלא עבדו לימדו אותי לפעמים יותר מאלה שהצליחו. ניתוח מעמיק של מקרי כישלון חשף דפוסים ומגבלות שלא הייתי מודע להם, והוביל לשיפורים משמעותיים במודל.

8.3 מה הייתי עושה אחרת

במבט לאחור, ישנם מספר דברים שהייתי עושה אחרת. הטעות המשמעותית ביותר שלי הייתה להתחיל בכתיבת קוד מוקדם מדי. הייתי צריך להקדיש זמן רב יותר לתכנון הארכיטקטורה. בילתי שעות רבות בשכתוב ושינוי הקוד, מה שיכולתי לחסוך עם תכנון מקדים טוב יותר. היום אני מבין שתכנון בלמידה עמוקה הוא קריטי, וטעותי עלתה בזמן יקר.

חוכמה נוספת שהגיעה בדיעבד קשורה לאסטרטגיית האימון. הייתי מתחיל באימון המבוסס על למידה מהדגמה, שבו הסוכן לומד מדוגמאות של התנהגות אנושית, לפני שאני עובר ללמידת חיזוקים טהורה. גישה כזו הייתה מספקת למודל נקודת פתיחה טובה יותר, במקום להתחיל ללמוד מאפס באמצעות ניסוי וטעייה בלבד.

שיקול אחרון הוא השימוש בספריות קיימות. בשם עקרון "אל תמציא מחדש את הגלגל", הייתי שוקל שימוש בסביבות מוכנות כמו Gym OpenAI במקום לפתח את סביבת המשחק מאפס. בדיעבד, זה היה מאפשר לי להתמקד יותר באלגוריתם הלמידה עצמו ופחות בפרטי המימוש הטכניים של המשחק.

לסיכום, הפרויקט הזה היה עבורי הרבה יותר מאשר תרגיל אקדמי. הייתה זו הזדמנות אמיתית לצלול לעומק עולם הבינה המלאכותית ולהתמודד עם אתגרים מעשיים בלמידת חיזוקים. למדתי לא רק על אלגוריתמים ורשתות

נוירונים, אלא גם על תהליכי פיתוח, פתרון בעיות, והתמודדות עם אי-ודאות. מדהים לחשוב שמשחק פאזל פשוט לכאורה יכול להוות פלטפורמה כה עשירה להתפתחות מקצועית ואישית. אני מביט קדימה בהתרגשות להמשיך ולחקור את תחום למידת החיזוקים ואת היישומים המרתקים שלו בעולם האמיתי.

9 ביבליוגרפיה

מקורות

[1] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep Learning. *Nature*, 521(7553), 436-444.

[2] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

[3] Kingma, D. P., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. *arXiv preprint* arXiv:1412.6980.

[4] Glorot, X., Bordes, A., & Bengio, Y. (2011). Deep Sparse Rectifier Neural Networks. *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, 315-323.

[5] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press.

[6] Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3-4), 279-292.

[7] Mnih, V., Kavukcuoglu, K., Silver, D., et al. (2015). Human-level Control Through Deep Reinforcement Learning. *Nature*, 518(7540), 529-533.

[8] Lin, L. J. (1992). Self-improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching. *Machine Learning*, 8(3-4), 293-321.

[9] גיטרמן, ד. (2023). Sokoban Reinforcement Learning with OpenAI Gymnasium [מחברת Jupyter Notebook]. פרויקט במסגרת קורס למידת מכונה, אוניברסיטת רייכמן, המחלקה למדעי המחשב.

[10] גיטרמן, ד. (2023). Reinforcement Learning- Final Course Project [ספר פרויקט]. אוניברסיטת רייכמן, המחלקה למדעי המחשב.

10 נספחים

10.1 קוד המקור

sokoban_real_final (5)

April 30, 2025

- -

1 -

1.1

[ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

1.2

[ ]:
```
!apt-get update
!pip install torch numpy pygame
!apt-get install -y xvfb
!pip install pyvirtualdisplay
```

1.3

[ ]:
```
# For game
import pygame
import numpy as np
from collections import deque
import random
import copy
%matplotlib inline
import matplotlib.pyplot as plt


# Pytorch
import torch
import torch.nn as nn
import torch.optim as optim

# For visualization
import time
import imageio
import base64
```

```python
import IPython
import pyvirtualdisplay
import matplotlib.pyplot as plt
from IPython.display import display, clear_output


# For the map generator
import ipywidgets as widgets

# For model training
from torch.utils.tensorboard import SummaryWriter
import os
```

1.4

[ ]:
```python
# Initialize Pygame
pygame.init()

# Game constants
TILE_SIZE = 50
WINDOW_WIDTH = 560
WINDOW_HEIGHT = 560

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
BROWN = (139, 69, 19)

# Create the game window
screen = pygame.display.set_mode((WINDOW_WIDTH, WINDOW_HEIGHT))
pygame.display.set_caption("Sokoban")

# Load images
box_image = pygame.image.load('/content/drive/MyDrive/sokoban/box.png')
box_image = pygame.transform.scale(box_image, (TILE_SIZE, TILE_SIZE))

player_image = pygame.image.load('/content/drive/MyDrive/sokoban/player.png')
player_image = pygame.transform.scale(player_image, (TILE_SIZE, TILE_SIZE))

# Game state representation
# 0: empty, 1: wall, 2: box, 3: target, 4: player, 5: box on target, 6: player␣
↪on target
initial_state = np.array([
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 3, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
```

```
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 2, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 4, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
])

game_state = initial_state.copy()

def find_player():
    return np.where((game_state == 4) | (game_state == 6))

def move_player(dx, dy):
    global game_state
    player_pos = find_player()
    x, y = player_pos[0][0], player_pos[1][0]
    new_x, new_y = x + dx, y + dy

    if game_state[new_x, new_y] in [0, 3]:  # Empty space or target
        game_state[x, y] = 3 if game_state[x, y] == 6 else 0
        game_state[new_x, new_y] = 6 if game_state[new_x, new_y] == 3 else 4
    elif game_state[new_x, new_y] in [2, 5]:  # Box or box on target
        next_x, next_y = new_x + dx, new_y + dy
        if game_state[next_x, next_y] in [0, 3]:
            game_state[x, y] = 3 if game_state[x, y] == 6 else 0
            game_state[new_x, new_y] = 6 if game_state[new_x, new_y] == 5 else 4
            game_state[next_x, next_y] = 5 if game_state[next_x, next_y] == 3␣
 ↪else 2

def check_fail():
    """Check if any box is stuck in an irretrievable position."""
    for y in range(1, game_state.shape[0] - 1):
        for x in range(1, game_state.shape[1] - 1):
            if game_state[y, x] in [2, 5]:  # Box or box on target
                # Check for corner traps
                if (game_state[y - 1, x] in [1, 2] and game_state[y, x - 1] in␣
 ↪[1, 2]) or \
                   (game_state[y - 1, x] in [1, 2] and game_state[y, x + 1] in␣
 ↪[1, 2]) or \
                   (game_state[y + 1, x] in [1, 2] and game_state[y, x - 1] in␣
 ↪[1, 2]) or \
                   (game_state[y + 1, x] in [1, 2] and game_state[y, x + 1] in␣
 ↪[1, 2]):
                    return True
    return False

def check_win():
```

```
    return np.all(game_state[game_state == 2] == 5)

def draw_game():
    screen.fill(WHITE)
    for y in range(game_state.shape[0]):
        for x in range(game_state.shape[1]):
            rect = pygame.Rect(x * TILE_SIZE, y * TILE_SIZE, TILE_SIZE,␣
 ↪TILE_SIZE)
            if game_state[y, x] == 1:  # Wall
                pygame.draw.rect(screen, BROWN, rect)
                pygame.draw.rect(screen, BLACK, rect, 2)
            elif game_state[y, x] == 2:  # Box
                screen.blit(box_image, rect)
            elif game_state[y, x] == 3:  # Target
                pygame.draw.rect(screen, (0, 255, 0), rect, 2)
            elif game_state[y, x] == 4:  # Player
                screen.blit(player_image, rect)
            elif game_state[y, x] == 5:  # Box on target
                pygame.draw.rect(screen, (0, 255, 0), rect, 2)
                screen.blit(box_image, rect)
            elif game_state[y, x] == 6:  # Player on target
                pygame.draw.rect(screen, (0, 255, 0), rect, 2)
                screen.blit(player_image, rect)
    pygame.display.flip()

def main():
    global game_state
    clock = pygame.time.Clock()
    FPS = 60
    running = True

    while running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                running = False
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_LEFT:
                    move_player(0, -1)
                elif event.key == pygame.K_RIGHT:
                    move_player(0, 1)
                elif event.key == pygame.K_UP:
                    move_player(-1, 0)
                elif event.key == pygame.K_DOWN:
                    move_player(1, 0)
                elif event.key == pygame.K_r:  # Reset the game
                    game_state = initial_state.copy()
```

```
        draw_game()

        if check_win():
            font = pygame.font.Font(None, 74)
            text = font.render("You Win!", True, BLACK)
            screen.blit(text, (WINDOW_WIDTH // 2 - text.get_width() // 2,␣
↪WINDOW_HEIGHT // 2 - text.get_height() // 2))
            pygame.display.flip()
            pygame.time.wait(2000)
            game_state = initial_state.copy()

        if check_fail():
            font = pygame.font.Font(None, 74)
            text = font.render("Game Over!", True, BLACK)
            screen.blit(text, (WINDOW_WIDTH // 2 - text.get_width() // 2,␣
↪WINDOW_HEIGHT // 2 - text.get_height() // 2))
            pygame.display.flip()
            pygame.time.wait(2000)
            game_state = initial_state.copy()


        clock.tick(FPS)

    pygame.quit()
```

1.5

[ ]:
```
# Neural Network Model
class SokobanNet(nn.Module):
    def __init__(self):
        super(SokobanNet, self).__init__()

        # Input: 9x11 board with 7 possible states (one-hot encoded)
        self.conv1 = nn.Conv2d(7, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, padding=1)

        # Calculate the size of flattened features
        self.fc1 = nn.Linear(64 * 9 * 11, 512)
        self.fc2 = nn.Linear(512, 4)  # 4 actions (up, down, left, right)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = torch.relu(self.conv3(x))
        x = x.view(x.size(0), -1)  # Flatten
        x = torch.relu(self.fc1(x))
```

```python
        return self.fc2(x)

class ReplayBuffer:
    def __init__(self, capacity=10000):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        return random.sample(self.buffer, batch_size)

    def __len__(self):
        return len(self.buffer)

def preprocess_state(state):
    """Convert game state to one-hot encoded tensor"""
    processed_state = np.zeros((7, 9, 11), dtype=np.float32)
    for i in range(7):  # 7 possible states (0-6)
        processed_state[i] = (state == i).astype(np.float32)
    return torch.FloatTensor(processed_state).unsqueeze(0)

class DQNAgent:
    def __init__(self):
        self.device = torch.device("cuda" if torch.cuda.is_available() else␣
 ↪"cpu")
        self.policy_net = SokobanNet().to(self.device)
        self.target_net = SokobanNet().to(self.device)
        self.target_net.load_state_dict(self.policy_net.state_dict())

        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=0.001)
        self.memory = ReplayBuffer(capacity=50000)

        self.batch_size = 32
        self.gamma = 0.99
        self.epsilon = 1.0
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.998
        self.target_update = 10
        self.steps = 0

    def select_action(self, state, training=True):
        """Select action using epsilon-greedy policy"""
        if training and random.random() < self.epsilon:
            return random.randint(0, 3)

        with torch.no_grad():
```

```
            state_tensor = preprocess_state(state).to(self.device)
            q_values = self.policy_net(state_tensor)
            return q_values.max(1)[1].item()

    def train_step(self):
        if len(self.memory) < self.batch_size:
            return None

        transitions = self.memory.sample(self.batch_size)
        batch = list(zip(*transitions))

        state_batch = torch.cat([preprocess_state(s) for s in batch[0]]).
↪to(self.device)
        action_batch = torch.LongTensor(batch[1]).unsqueeze(1).to(self.device)
        reward_batch = torch.FloatTensor(batch[2]).to(self.device)
        next_state_batch = torch.cat([preprocess_state(s) for s in batch[3]]).
↪to(self.device)
        done_batch = torch.FloatTensor(batch[4]).to(self.device)

        # Compute current Q values
        current_q_values = self.policy_net(state_batch).gather(1, action_batch)

        # Compute next Q values
        with torch.no_grad():
            next_q_values = self.target_net(next_state_batch).max(1)[0]
        expected_q_values = reward_batch + (1 - done_batch) * self.gamma *␣
↪next_q_values

        # Compute loss and optimize
        loss = nn.MSELoss()(current_q_values, expected_q_values.unsqueeze(1))
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # Update target network
        self.steps += 1
        if self.steps % self.target_update == 0:
            self.target_net.load_state_dict(self.policy_net.state_dict())

        # Update epsilon
        self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay)

        return loss.item()
```

1.6

[ ]:
```
class SokobanEnv:
    def __init__(self):
        self.initial_state = initial_state
        self.state = None
        self.action_map = {
            0: (-1, 0),  # up
            1: (1, 0),   # down
            2: (0, -1),  # left
            3: (0, 1)    # right
        }
        self.reset()

    def reset(self):
        global game_state
        game_state = self.initial_state.copy()
        self.state = game_state.copy()
        return self.state

    def step(self, action):
        global game_state
        prev_state = game_state.copy()

        # Convert action number to direction
        dx, dy = self.action_map[action]
        move_player(dx, dy)

        # Get new state
        new_state = game_state.copy()

        # Calculate reward
        reward = self._calculate_reward(prev_state, new_state)

        # Check if episode is done
        done = check_win() or check_fail()

        return new_state, reward, done

    def _calculate_reward(self, prev_state, new_state):
        # Base step penalty
        reward = -0.1

        # Get box positions
        prev_box = np.where((prev_state == 2) | (prev_state == 5))
        new_box = np.where((new_state == 2) | (new_state == 5))
        target = np.where((self.initial_state == 3))
```

```
        if len(prev_box[0]) > 0 and len(new_box[0]) > 0:
            # Calculate Manhattan distances to target
            prev_dist = abs(prev_box[0][0] - target[0][0]) + abs(prev_box[1][0]␣
↪- target[1][0])
            new_dist = abs(new_box[0][0] - target[0][0]) + abs(new_box[1][0] -␣
↪target[1][0])

            # Reward for moving closer to target
            if new_dist < prev_dist:
                reward += 5
            elif new_dist > prev_dist:
                reward -= 2

        # Boxes on target rewards
        prev_boxes_on_target = np.sum(prev_state == 5)
        new_boxes_on_target = np.sum(new_state == 5)
        target_delta = new_boxes_on_target - prev_boxes_on_target

        if target_delta > 0:  # Box newly placed on target
            reward += 20
        elif target_delta < 0:  # Box moved off target
            reward -= 10

        # Terminal states
        if check_win():
            reward += 100
        elif check_fail():
            reward -= 50

        return reward



def train_dqn(num_episodes, max_steps=50, save_dir='saved_models',␣
↪log_dir='runs/sokoban'):
    torch.manual_seed(2)
    np.random.seed(2)
    random.seed(2)

    # Create directories if they don't exist
    os.makedirs(save_dir, exist_ok=True)
    os.makedirs(log_dir, exist_ok=True)

    # Initialize tensorboard writer
    writer = SummaryWriter(log_dir)

    env = SokobanEnv()
```

```python
agent = DQNAgent()

episode_rewards = []
episode_lengths = []
best_avg_reward = float('-inf')

for episode in range(num_episodes):
    state = env.reset()
    episode_reward = 0
    episode_loss = 0
    num_steps = 0

    for step in range(max_steps):
        # Select action
        action = agent.select_action(state)

        # Take action
        next_state, reward, done = env.step(action)

        # Store transition
        agent.memory.push(state, action, reward, next_state, done)

        # Train model
        loss = agent.train_step()
        if loss is not None:
            episode_loss += loss

        state = next_state
        episode_reward += reward
        num_steps = step + 1

        if done:
            break

    episode_rewards.append(episode_reward)
    episode_lengths.append(num_steps)

    # Calculate averages
    avg_reward = np.mean(episode_rewards[-10:])
    avg_length = np.mean(episode_lengths[-10:])
    avg_loss = episode_loss / num_steps if num_steps > 0 else 0

    # Log to tensorboard
    writer.add_scalar('Reward/Episode', episode_reward, episode)
    writer.add_scalar('Length/Episode', num_steps, episode)
    writer.add_scalar('Average_Reward/10_Episodes', avg_reward, episode)
    writer.add_scalar('Average_Length/10_Episodes', avg_length, episode)
```

```
        writer.add_scalar('Loss/Episode', avg_loss, episode)
        writer.add_scalar('Epsilon/Episode', agent.epsilon, episode)

        # Save model every 100 episodes
        if (episode + 1) % 100 == 0:
            model_path = os.path.join(save_dir, f'model_episode_{episode+1}.
↪pth')
            torch.save({
                'episode': episode,
                'model_state_dict': agent.policy_net.state_dict(),
                'optimizer_state_dict': agent.optimizer.state_dict(),
                'epsilon': agent.epsilon,
                'reward': avg_reward
            }, model_path)
            print(f"Model saved at episode {episode + 1}")

        # Save best model
        if avg_reward > best_avg_reward:
            best_avg_reward = avg_reward
            best_model_path = os.path.join(save_dir, 'best_model.pth')
            torch.save({
                'episode': episode,
                'model_state_dict': agent.policy_net.state_dict(),
                'optimizer_state_dict': agent.optimizer.state_dict(),
                'epsilon': agent.epsilon,
                'reward': avg_reward
            }, best_model_path)

        # Print progress
        if (episode + 1) % 10 == 0:
            print(f"Episode {episode + 1}")
            print(f"Average Reward: {avg_reward:.2f}")
            print(f"Average Length: {avg_length:.2f}")
            print(f"Epsilon: {agent.epsilon:.2f}")
            print(f"Loss: {avg_loss:.4f}")
            print("--------------------")

    writer.close()
    return agent, episode_rewards, episode_lengths
```

1.7

[ ]:
```
# Run training
agent_one_box, rewards, lengths = train_dqn(num_episodes=1000,␣
↪save_dir='saved_models_one_box')

# Plot training progress
```

```
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(rewards)
plt.title('Episode Rewards')
plt.xlabel('Episode')
plt.ylabel('Reward')

plt.subplot(1, 2, 2)
plt.plot(lengths)
plt.title('Episode Lengths')
plt.xlabel('Episode')
plt.ylabel('Steps')
plt.show()
```

1.8

[ ]:
```
# To load a saved model:
def load_model(model_path, agent):
    checkpoint = torch.load(model_path, weights_only=False)
    agent.policy_net.load_state_dict(checkpoint['model_state_dict'])
    agent.target_net.load_state_dict(checkpoint['model_state_dict'])
    agent.optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    agent.epsilon = checkpoint['epsilon']
    return agent, checkpoint['episode'], checkpoint['reward']
```

[ ]:
```
# To load the best model:
# agent_one_box, episode, reward = load_model('saved_models_one_box/best_model.
 ↪pth', agent_one_box)

# To load the best model, saved in Drive for presentation:
agent_one_box, episode, reward = load_model('/content/drive/MyDrive/sokoban/
 ↪best_model.pth', DQNAgent())
print(f"Loaded the best model! Episode {episode} with reward {reward}")
```

1.9

[ ]:
```
def embed_mp4(filename):
  """Embeds an mp4 file in the notebook."""
  video = open(filename,'rb').read()
  b64 = base64.b64encode(video)
  tag = '''
  <video width="640" height="480" controls>
    <source src="data:video/mp4; base64,{0}" type="video/mp4">
  Your browser does not support the video tag.
  </video>'''.format(b64.decode())
```

```
  return IPython.display.HTML(tag)
display = pyvirtualdisplay.Display(visible=0, size=(1400, 900)).start()

def record_agent_gameplay(agent, video_filename):
   global game_state
   done = False
   iter = 0
   total_reward = 0
   num_of_steps = 0
   start_time = time.time()

   game_state = initial_state.copy()

   with imageio.get_writer(video_filename, fps=10) as video:
       # Capture initial state
       draw_game()
       pygame_surface = screen.copy()
       frame = pygame.surfarray.array3d(pygame_surface)
       frame = np.transpose(frame, (1, 0, 2))
       video.append_data(frame)

       while (iter < 10) or not done:
           time_passed = int(time.time() - start_time)
           if done or time_passed > 5:
               break

           iter += 1
           num_of_steps += 1

           # Get agent action and execute
           action = agent.select_action(game_state, training=False)
           dx, dy = {0: (-1, 0), 1: (1, 0), 2: (0, -1), 3: (0, 1)}[action]
           move_player(dx, dy)

           # Capture frame
           draw_game()
           pygame_surface = screen.copy()
           frame = pygame.surfarray.array3d(pygame_surface)
           frame = np.transpose(frame, (1, 0, 2))
           video.append_data(frame)

           done = check_win() or check_fail()

   print(f"Steps taken: {num_of_steps}")
   return embed_mp4(video_filename)
```

[ ]:
```
  # Usage
record_agent_gameplay(agent_one_box, 'one_box.mp4')
```

2 -

2.1

[ ]:
```
initial_state = np.array([
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 2, 3, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 4, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 3, 2, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
])
```

2.2

[ ]:
```
# Run training
agent_two_box, rewards, lengths = train_dqn(num_episodes=3000,␣
 ↪save_dir='saved_models_two_box')

# Plot training progress
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(rewards)
plt.title('Episode Rewards')
plt.xlabel('Episode')
plt.ylabel('Reward')

plt.subplot(1, 2, 2)
plt.plot(lengths)
plt.title('Episode Lengths')
plt.xlabel('Episode')
plt.ylabel('Steps')
plt.show()
```

2.3

[ ]:
```
# To load a saved model:
def load_model(model_path, agent):
    checkpoint = torch.load(model_path)
    agent.policy_net.load_state_dict(checkpoint['model_state_dict'])
```

```
    agent.target_net.load_state_dict(checkpoint['model_state_dict'])
    agent.optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    agent.epsilon = checkpoint['epsilon']
    return agent, checkpoint['episode'], checkpoint['reward']
```

[ ]:
```
# To load the best model:
agent_two_box, episode, reward = load_model('saved_models_two_box/best_model.
 ↪pth', agent_two_box)
print(f"Loaded the best model! Episode {episode} with reward {reward}")
```

2.4

[ ]:
```
# Usage
record_agent_gameplay(agent_two_box, 'two_box.mp4')
```

3

[ ]:
```
from IPython.display import display, clear_output

def create_sokoban_map():
    # Initialize empty grid (9x11 with walls)
    grid = np.ones((9, 11))  # All walls initially
    grid[1:-1, 1:-1] = 0  # Empty inside

    # Create buttons for each cell
    buttons = {}
    for i in range(9):
        for j in range(11):
            buttons[(i,j)] = widgets.Button(
                description='',
                layout=widgets.Layout(width='40px', height='40px'),
                disabled=i==0 or i==8 or j==0 or j==10  # Disable border buttons
            )

    # Create tile selection buttons
    tile_types = {
        0: ('Empty', 'white'),
        1: ('Wall', 'brown'),
        2: ('Box', 'orange'),
        3: ('Target', 'lightgreen'),
        4: ('Player', 'lightblue')
    }

    selected_tile = widgets.RadioButtons(
        options=[(name, val) for val, (name, _) in tile_types.items()],
        description='Tile:',
```

```
    layout={'width': 'max-content'}
)

def update_button_colors():
    for i in range(9):
        for j in range(11):
            val = int(grid[i,j])
            buttons[(i,j)].style.button_color = tile_types[val][1]

def on_cell_click(i, j):
    def handle_click(b):
        nonlocal grid
        if selected_tile.value == 4:  # If placing player, remove old player
            player_pos = np.where(grid == 4)
            if len(player_pos[0]) > 0:
                grid[player_pos] = 0
        grid[i,j] = selected_tile.value
        update_button_colors()
    return handle_click

# Assign click handlers
for i in range(9):
    for j in range(11):
        if not (i==0 or i==8 or j==0 or j==10):  # Skip borders
            buttons[(i,j)].on_click(on_cell_click(i,j))

# Create grid layout
grid_layout = widgets.GridBox(
    [buttons[(i,j)] for i in range(9) for j in range(11)],
    layout=widgets.Layout(
        grid_template_columns='repeat(11, 40px)',
        grid_gap='1px'
    )
)

def get_array(b):
    print("Map created successfully!")
    print("initial_state = np.array([")
    for row in grid:
        print("    [" + ", ".join(map(str, map(int, row))) + "],")
    print("])")

get_array_button = widgets.Button(description='Create map')
get_array_button.on_click(get_array)

# Initialize colors
update_button_colors()
```

```
    # Display everything
    display(widgets.VBox([
        widgets.HTML('<h3>Sokoban Map Creator</h3>'),
        selected_tile,
        grid_layout,
        get_array_button
    ]))

    return grid

# Usage:
initial_state = create_sokoban_map()
```

3.1

[ ]:
```
# Run training
agent_dynamic, rewards, lengths = train_dqn(num_episodes=3000,␣
 ↪save_dir='saved_models_dynamic_box')

# Plot training progress
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(rewards)
plt.title('Episode Rewards')
plt.xlabel('Episode')
plt.ylabel('Reward')

plt.subplot(1, 2, 2)
plt.plot(lengths)
plt.title('Episode Lengths')
plt.xlabel('Episode')
plt.ylabel('Steps')
plt.show()
```

3.2

[ ]:
```
record_agent_gameplay(agent_dynamic, 'dynamic_box.mp4')
```