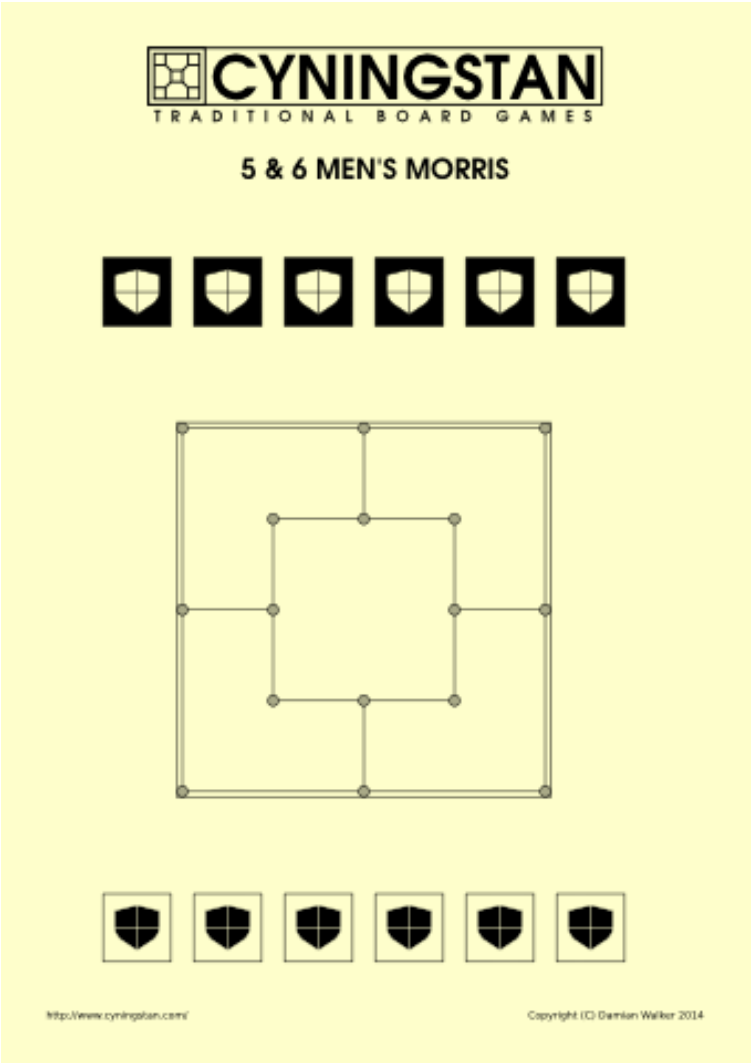
נושא הפרויקט: 5 Men's Morris

שם המגיש: אלון אוסדצ'י

שם המורה: אורנה אברך

ת.ז: 216754267

תיכון הריאלי העברי בחיפה



תוכן עניינים:

תוכן עניינים	**2**
מבוא	3
תיאור המשחק	3-4
הרקע לפרויקט	4
תהליך מחקר המשחק	5
אתגרים מרכזיים ופתרונות	5
שלב איסוף נתונים	6
תרשים UML לתיאור המחלקות	6-10
תיאור המילון	11
שלב אימון ובנית המודל הלומד	12
מטרת האימון	12
תיאור גרפי של המודל הראשוני	12-15
ארכיטקטורת המודל הראשוני	15-18
תוצאות המודל הראשוני	18-23
דיון במודל שנבחר ובמהלכים שנעשו לשיפור האימון	24
תיאור גרפי המודל הסופי	25-27
תוצאות המודל הסופי	28-32
יישום	33
מבוא	33
ממשק משתמש	33-34
מתודולוגיית MVC ותרשים UML	34-41
רפלקציה	42
הקוד ליצירת המילונים	43-67
הקוד לגרפיקה	67-90
הקוד ליצירת המודלים (דוגמה אחת מתוך חמש)	90-93

מבוא:

תיאור המשחק:

סקירה כללית:

5 Men's Morris הוא מחשק אסטרטגיה לשני שחקנים. מטרת המשחק היא לייצר טחנות, שהן רצף של 3 חיילים צמודים בשורה או טור, כדי להוציא חיילים של היריב, לצמצם אותם ל2 ולנצח.

רכיבים:

1. **לוח המשחק:** מורכב מ-2 ריבועים של 3 על 3, אחד חיצוני ואחד פנימי, וחיבורים ביניהם היוצרים 16 נקודות על הלוח עליהן ניתן להניח חיילים. הנקודות ממוספרות על ידי אותיות כמו שמתואר בתמונה מתחת להסברים.
2. **חיילים:** לכל שחקן יש 5 חיילים שיכולים לנוע בין הנקודות המחוברות בקו בלבד.

מטרת המשחק:

במשחק קיימות 2 דרכים לנצח:

1. לצמצם את מספר החיילים של היריב ל-2 בלבד על ידי יצירת טחנה.
2. לחסום את החיילים של היריב כך שאין לו אף מהלך חוקי לעשות.

טחנה:

כדי להוציא שחקן של היריב מהמשחק יש ליצור טחנה שהיא רצף של 3 חיילים בטור או שורה. כאשר שחקן יוצר תחנה הוא יכול להוציא חייל של היריב מהמשחק.

חיילים שנמצאים בטחנה מוגנים מפני הוצאה של היריב, אלא אם אין אף חייל אחר להוציא, כלומר אם נשארו רק 3 חיילים במשחק והם בטחנה, היריב יכול להוציא אחד מהם אם יצר טחנה משלו.

מהלך המשחק:

המשחק מחולק ל-2 שלבים: **שלב ההנחה** ו**שלב ההזזה.**

שלב ההנחה:

הלוח מתחיל ריק. השחקן שמתחיל יכול לשים חייל אחד איפה שהוא רוצה. אחר כך השחקן השני שם חיל שלו איפה שפנוי, וכך ממשיך עד שכל החיילים על הלוח.

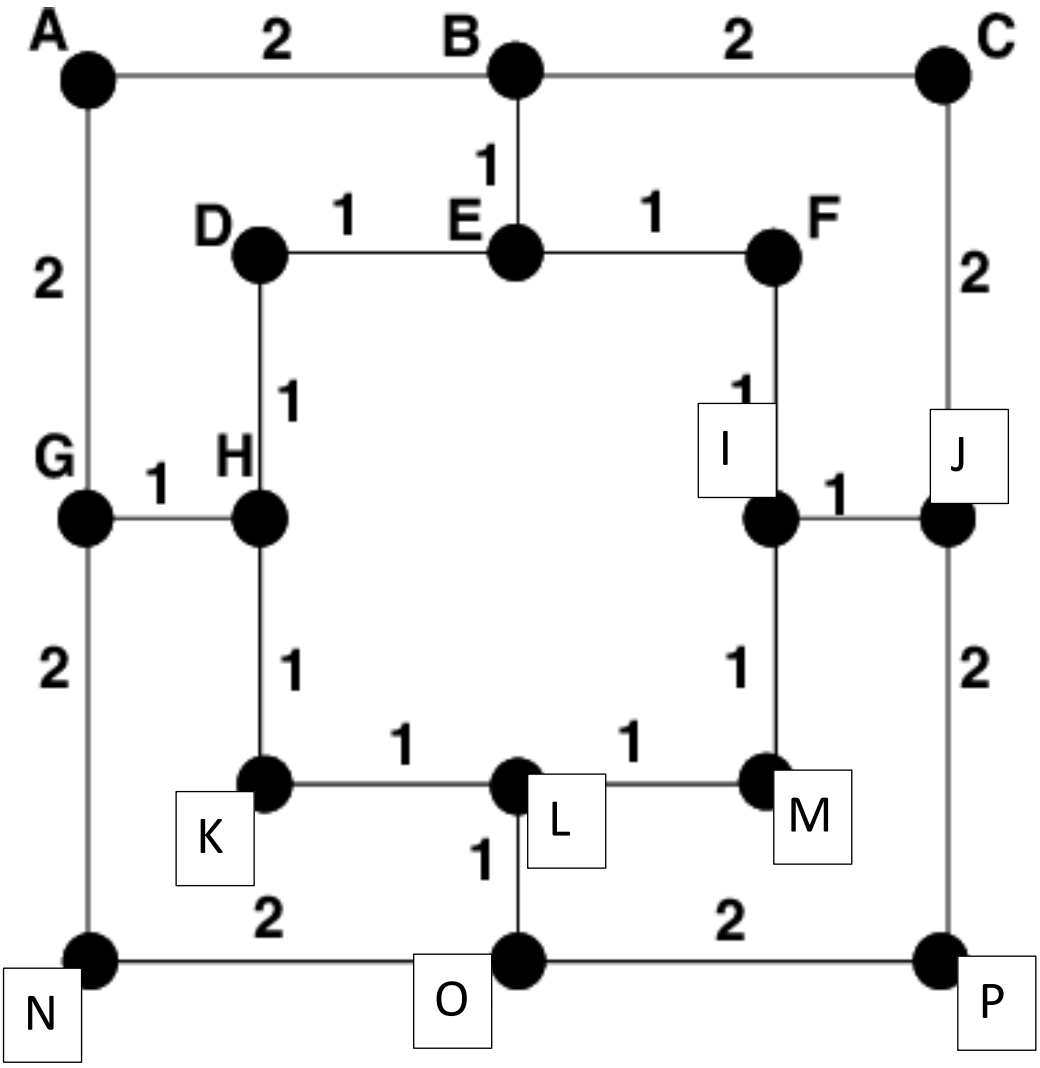
שלב ההזזה:

בשלב זה כל שחקן בתורו רשאי להזיז אחד מחייליו לנקודה פנויה סמוכה המחוברת לנקודה עלייה הוא נמצא בקו.

כללים מיוחדים:

1. **תעופה:** כאשר לשחקן נשארו רק 3 חיילים, הוא רשאי להזיז את החיילים שלו לכל נקודה פנויה על הלוח, גם אם היא לא מחוברת בקו לנקודה עליו נמצא החייל.
2. לא ניתן להוציא חייל שלוקח חלק בטחנה אלא אם זאת האפשרות היחידה.

מבנה הלוח:



הרקע לפרויקט:

הפרויקט עוסק בלמידת מכונה ולמידה עמוקה, וכולל בתוכו ממשק גרפי למשחק 5 Men's Morris בעיצוב נוח לשימוש לשחקן. הפרויקט משלב אלמנטים של למידה על ידי חיזוקים (reinforcement learning) ורשתות נוירונים (ANN) במטרה לאמן סוכן חכם במספר דרכים ולפתח אסטרטגיית משחק חכמה.

תהליך מחקר למשחק:

במהלך מחקרי חיפשתי משחק לוח שישלב אתגר ממשי יחד עם פשטות מספקת, על מנת שיתאפשר אימון יעיל ומוצלח של מודלים חכמים. בתום המחקר מצאתי את המשחק 5 Men's Morris, ובחרתי בו למרות הקשיים הכרוכים במשחק זה אשר אני אפרט עליהם בהמשך הפרק.

מצאתי באינטרנט אתר בו אפשר לשחק את המשחק נגד סוכן, ושם שיחקתי עשרות משחקים במטרה להכיר את כל חוקי המשחק ולהכיר את אסטרטגיות המשחק, וזאת כדי להעשיר את הידע שלי על המשחק ולהכין אותי להמשך הפרויקט.

אתגרים מרכזיים ופתרונות:

נתקלתי במספר אתגרים במהלך עבודה על הפרויקט, בהם היו 2 מרכזיים:

<u>צורת הלוח:</u> הלוח מורכב מ-16 נקודות המפוזרות על הלוח כשני ריבועים בגודל 3X3 ללא מרכזים, וביניהם יש קוים המחברים נקודות צמודות בריבועים ואת האמצעים של צלעות הריבועים. לוח זה אינו לוח שניתן לייצג על ידי רשימה פשוטה, כלומר קשה לשמור את הלוח כמבנה נתונים פשוט ונוח. כדי להתמודד עם בעיה זו יצרתי עצם בשם Sector המייצג נקודה על הלוח המכיל את התכונות הבאות:

1. Taken_by - שומר את בעלות הנקודה (1:סוכן, 0:ריק, 1-:שחקן).
2. All_mills - שומר את כל הטחנות האפשריות שניתן ליצור מהנקודה.
3. Legal_moves – שומר את כל המהלכים החוקיים בשלב התזוזה מהנקודה.

את הלוח ייצגתי כמילון של 16 ערכים. המפתח הוא האות המייצגת את הנקודה (לפי האיור בעמוד 4) והערך הוא עצם מסוג Sector.

<u>חלקי המשחק:</u> במשחק זה יש מספר שלבים שונים, אשר יש להם תכונות וחוקים שונים (הנחה, תזוזה ותעופה). ריבוי השלבים גרם לכפילויות של לוחות שיש להם תכונות שונות, אשר השפיעו על הניקוד של הלוחות בצורה לא תקינה, לדוגמה אותו לוח בדיוק יכול להופיע גם בשלב התזוזה וגם בשלב ההנחה, אך הניקוד של שתי הלוחות עשוי להיות שונה מאוד. כדי להתמודד עם בעיה זו פיצלתי את המילונים שלי ואת המודלים שלי לפי שלבי המשחק, מה שמבטיח ייצוג אמין ועצמאי של ניקודי הלוחות.

שלב איסוף נתונים:

תרשים UML לתיאור המחלקות:

מחלקה: Sector

מטרה – מייצגת נקודה על הלוח

תכונות:

שם	טיפוס	הסבר
taken_by	int	מייצג מי מחזיק בסקטור (1=סוכן, 1-=שחקן, 0=ריק)
all_mills	list[tuple[str]]	רשימת כל השלשות (mills) שאפשר ליצור מסקטור זה
legal_moves	tuple[str,…]	רשימת הסקטורים אליהם מותר לזוז מהסקטור הנוכחי

פעולות:

שם	טענת כניסה	טענת יציאה / תיאור פעולה
__init__(self, mills, legal_moves)	mills: list[tuple[str]] legal_moves: tuple[str,…]	יוצר עצם Sector, מאתחל את taken_by ל-0, ואת שאר התכונות לפי הפרמטרים שהתקבלו.

מחלקה: Game

מטרה – מייצג משחק יחיד

תכונות:

שם	טיפוס	הסבר
board	dict[str, Sector]	לוח המשחק
board_list	list[dict{str:Sector}]	היסטוריית לוחות משלב התזוזה
board_list_place	list[dict{str:Sector}]	היסטוריית לוחות משלב ההצבה
max_pieces	int	מקסימום חיילים (5)
player_sectors	int	מספר הנקודות שתפס השחקן
agent_sectors	int	מספר הנקודות שתפס הסוכן
active_mills	list[tuple[str]]	טחנות פעילות
agent_taken_sectors	list[str]	רשימת נקודות שתפס הסוכן

player_taken_sectors	list[str]	רשימת נקודות שתפס השחקן
empty_sectors	list[str]	רשימת נקודות פנויות

פעולות:

שם	טענת כניסה	טענת יציאה / תיאור פעולה
set_board(self)	–	מאתחלת את כל התכונות לערכי ברירת־מחדל של משחק חדש.
put_soldier_agent(self)	–	ממקמת חייל אקראי לסוכן, מעדכנת את התכונות המתאימות ומחזירה את המיקום.
put_soldier_player(self)	–	ממקמת חייל אקראי לשחקן, מעדכנת את התכונות המתאימות ומחזירה את המיקום.
check_legal_moves(self, sec)	sec: str	מחזירה True אם קיימים מהלכים חוקיים מסקטור sec ו־False אחרת.
check_jam_win_agent(self)	–	מחזירה True אם הסוכן תקוע (לא יכול לזוז בכלל) ו־False אחרת.
check_jam_win_player(self)	–	מחזירה True אם השחקן תקוע (לא יכול לזוז בכלל) ו־False אחרת.
take_sector_agent(self)	–	מזיזה חייל של הסוכן לנקודה חוקית אקראית, מעדכנת את התכונות המתאימות ומחזירה את המיקום של הנקודה אליה החייל זז.
take_sector_player(self)	–	מזיזה חייל של השחקן לנקודה חוקית אקראית, מעדכנת את התכונות המתאימות ומחזירה את המיקום של הנקודה אליה החייל זז.
check_mill_with_board(self, s, board)	s: str, board: dict{str:Sector}	מחזירה 1 אם לסוכן ו-1 אם לשחקן נוצרה טחנה על־פי המצב בלוח, ו-0 אחרת.
check_mill(self, s)	s: str	בודקת אם יש טחנה בלוח, מעדכנת את הטחנות הפעילות בהתאם ומחזירה 1\0\1- בהתאם.
double_mill_checker(self, s, owner)	s: str, owner: int	מחזירה True אם בנקודה s נוצרו שתי טחנות ו־false אחרת.

sector_in_active_mills(self, s, owner)	s: str, owner: int	מחזירה True אם s נמצא בתוך טחנה פעילה השייכת ל־owner.
remove_soldier(self, winner)	winner: int	מסירה חייל של המפסיד (אם winner=1מסירה שחקן, אם winner=-1 מסירה סוכן).
fly_soldier_agent(self)	–	מעיפה (ללא תלות בקווים המחברים את הנקודות) חייל של הסוכן לנקודה חוקית אקראית, מעדכנת את התכונות המתאימות ומחזירה את המיקום של הנקודה אליה החייל זז.
fly_soldier_player(self)	–	מעיפה (ללא תלות בקווים המחברים את הנקודות) חייל של השחקן לנקודה חוקית אקראית, מעדכנת את התכונות המתאימות ומחזירה את המיקום של הנקודה אליה החייל זז.
check_win_regular(self)	–	מחזירה 1 אם השחקן נותר עם 2 חיילים (הסוכן ניצח) -1, אם הסוכן נותר עם 2 (השחקן ניצח), ואחרת 0.
choose_dict_from_board(self, board)	board: dict{str:Sector}	בוחרת באיזה מילון להשתמש בשביל הסוכן החכם.
move_blocks_opponent(self, board, move_sector, opponent)	board: dict[str,Sector], move_sector: str, opponent: int	מחזירה True אם מהלך ל־ move_sector חוסם טחנה אפשרית של opponent.
smart_place(self, value_dicts)	value_dicts: dict[str,dict]	ממקמת חייל חכם לסוכן על פי מילון, מעדכנת את התכונות המתאימות ומחזירה את המיקום.
smart_player_place(self)	–	הצבת חייל חכמה של השחקן (בודקת טחנות אפשריות) ומחזירה את הנקודה.
smart_move(self, value_dicts)	value_dicts: dict[str,dict]	מזיזה באופן חכם על פי מילונים חייל של הסוכן לנקודה חוקית, מעדכנת את התכונות המתאימות ומחזירה את המיקום של הנקודה אליה החייל זז.

smart_player_move(self)	–	הזזה חכמה של השחקן (בודקת טחנות אפשריות), ומחזירה את היעד.
smart_fly(self, value_dicts)	value_dicts: dict{str:dict}	מעיפה באופן חכם על פי מילונים חייל של הסוכן לנקודה חוקית, מעדכנת את התכונות המתאימות ומחזירה את המיקום של הנקודה אליה החייל זז.
smart_player_fly(self)	–	תעופה חכמה של השחקן (בודקת טחנות אפשריות), ומחזירה את היעד.
smart_remove(self, winner, value_dicts)	winner: int, value_dicts: dict[str,dict]	הסרה חכמה של חייל את השחקן על פי מילונים.
smart_play_one_game(self, value_dicts)	value_dicts: dict[str,dict]	מדמה משחק שלם (חכם vs רנדומלי). מחזיר 1 אם הסוכן מנצח ו1- אם הוא מפסיד.
play_one_game(self)	–	מדמה משחק שלם (רנדומלי vs רנדומלי). מחזיר 1 אם הסוכן מנצח ו 1- אם הוא מפסיד.
smart_play_one_game_heuristic(self, value_dicts)	value_dicts: dict[str,dict]	מדמה משחק שלם (חכם vs יוריסטיקה). מחזיר 1 אם הסוכן מנצח ו1- אם הוא מפסיד.

<u>מחלקה: Games</u>

מטרה – מדמה מספר גדול של משחקים כדי ליצור מילונים של למידה מחיזוקים

תכונות:

שם	טיפוס	הסבר
gamma	float	זהו המשתנה בו אני מכפיל את הניקוד של הלוח שבא אחרי לוח נתון כדי לתת ללוח הנתון ניקוד.
place_dict	dict	מילון ערכי שלב ההנחה
no_fly_dict	dict	מילון ערכי שלב התזוזה ללא תעופה
agent_fly_dict	dict	מילון ערכי שלב התזוזה עם תעופת סוכן
player_fly_dict	dict	מילון ערכי שלב התזוזה עם תעופת שחקן
all_fly_dict	dict	מילון ערכי שלב התזוזה עם תעופת סוכן ושחקן

פעולות:

שם	טענת כניסה	טענת יציאה / תיאור פעולה
load_dicts(self)	–	טוענת קבצי JSON קיימים לתוך המילונים, או מאתחל מילונים ריקים במקרה של שגיאות.
choose_dict(self, board)	board: dict[str,Sector]	מחזירה מספר (1–4) המייצג את סוג המילון המתאים לפי מצב הלוח.
add_to_dict(self, grade, board_str, dict_obj)	grade: float board_str: str dict_obj: dict	מוסיף או מעדכן ערך במילון dict_objעבור המצב board_str עם הציון grade.
simulate(self)	–	מדמה מיליון משחקים רנדומליים ושומרת את המילונים קקובץ JSON.
smart_simulate(self)	–	מדמה 1,000 משחקים חכמים נגד שחקן רנדומלי, מדפיס שיעורי ניצחון/תיקו/הפסד כדי לבדוק את איכות המילונים
smart_dict_simulate(self)	–	מדמה מיליון משחקים חכמים ושומרת את המילונים קקובץ JSON.
smart_dict_simulate_explore(self)	–	מדמה מיליון משחקים explore-explicit (80% חכם, 20% רנדומלי) ושומרת את המילונים קקובץ JSON.
smart_dict_simulate_explore_heuristic(self)	–	כמו הקודמת, אך משתמש ביוריסטיקות לשחקן במקום שחקן רנדומלי.

פעולות חיצוניות:

שם	טענת כניסה	טענת יציאה
convert_taken_by_to_string()	sector_dict: dict[str,Sector]	מחזירה מחרוזת של 16 תווים-(O,X,שמתארת מצב לוח.

תיאור המילונים:

<u>פיצול המילונים:</u>

כמו שציינתי בפרק האתגרים והפתרונות, אני פיצלתי את המילונים שלי לשלבי המשחק: שלב ההנחה, שלב התזוזה ללא תעופה, שלב התזוזה עם תעופת שחקן, שלב התזוזה עם תעופת הסוכן ושלב התזוזה עם תעופה של שני השחקנים.

ערכי המילון:

מפתח - מחרוזת של אותיות המייצגות לוח נתון ('X' - שחקן, 'O' - סוכן, '-' – ריק) המסודרים על פי סדר הabc (עמ 4).

ערך - מערך המכיל את הניקוד של הלוח ואת כמות המופעים שלו במילון.

אופן מתן הניקוד ללוח:

הניקוד הניתן לניצחון הוא 100, לתיקו הוא 50 ולהפסד הוא 0. במהלך משחק אני שומר את כל הלוחות של אותו משחק ב-2 רשימות (אחת לתזוזה ואחת להנחה) ובסדר הפוך אני נותן ניקוד. הלוח הסופי מקבל ניקוד על פי תוצאת המשחק, וכל לוח אחרי זה מקבל את הניקוד של הלוח שבא אחריו כפול הגאמא, שאצלי מאותחל ל0.9.

הרצות המילון

1. 1,000,000 הרצות של סוכן רנדומלי נגד שחקן רנדומלי.
2. 1,000,000 הרצות של סוכן חכם על פי המילון נגד שחקן רמדומלי
3. 1,000,000 הרצות של סוכן explore explicit (80% חכם 20% רנדומלי) נגד שחקן רנדומלי.
4. 2,000,000 הרצות של סוכן explore explicit נגד שחקן רנדומלי עם היוריסטיקת טחנות (יוצר טחנות כאשר יכול).

שמירת קבצי המילון:

המילונים נשמרו כ-5 קבצי JSON נפרדים.

שלב אימון ובניית המודל:

מבוא:

בפרק זה נתעמק בתהליך הבנייה והאימון של הרשתות נוירונים המלאכותיות (ANN) שבניתי עבור הסוכן ה"גאון" במשחק. אני אפרט על מטרות האימון שלי, על הגרסאות השונות של המודלים שלי בעזרת נתונים מספריים וגרפיים המתארים את התפלגות הניקוד במילונים ואת שיפור הloss של המודל, ואסביר על השינויים שביצעתי בין המודלים במטרה לשפר את תפקודם.

חלוקת הרשתות:

בדומה למילונים שלי, אני פיצלתי את הרשת שלי ל-5 רשתות נפרדות, רשת אחד לכל מילון, כלומר רשת אחת לכל שלב של המשחק. פיצול זה נעשה מכיוון שניקוד של לוח מושפע מאוד על ידי השלב הנוכחי של המשחק בו הלוח נמצא, לכן היה עליי לחלק את הרשת.

מטרת האימון:

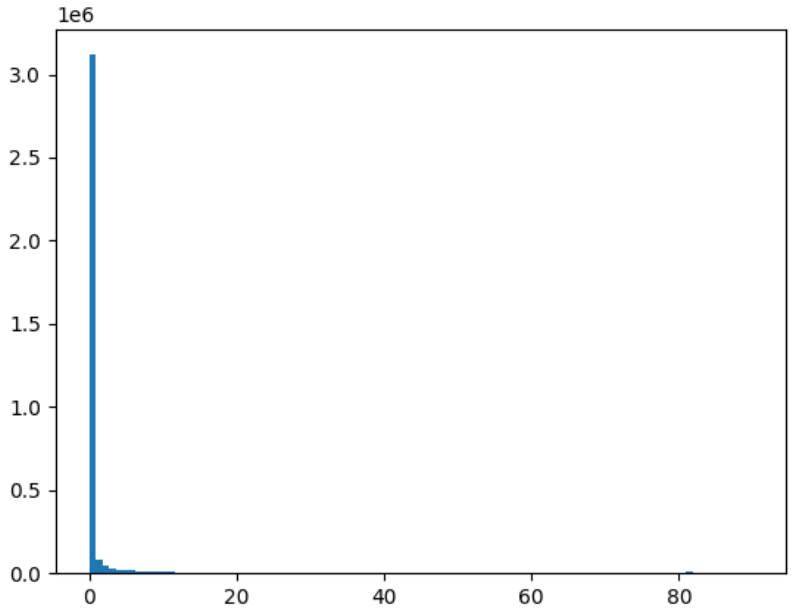
לאימון המודלים יש 2 מטרות עיקריות:

1. לימוד פונקצית ערך: המודל לומד לחזות את הניקוד של כל לוח אפשרי במשחק במטרה שהסוכן ה"גאון" יפעל כדי לנצח את השחקן על ידי פיתוח אסטרטגיית משחק אופטימלית.
2. כלליות: מטרת המודל הינה גם לנבא ניקוד של כל לוח אפשרי, גם אם אינו הופיע בשלב אימון המודל, או במילון כלל. בניגוד למילון, הרשת יכולה לזהות כל לוח ולנקד אותו על פי פרמטרים שהיא מחשבת בשלב האימון, ולא רק על פי ניסיון קודם כמו במילון.

תיאור רשתות ראשוניות:

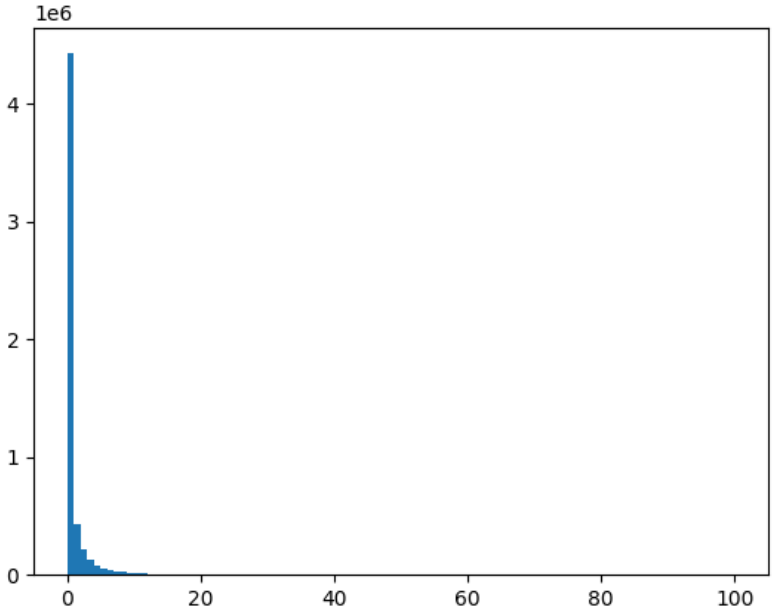
תחילה כדי להחליט לגבי כל מיני פרטים של ארכיטקטורת המודל, בדקתי את התפלגות הניקוד בנתונים שלי. להלן גרפים המתארים את ההתפלגויות:

Place_model:



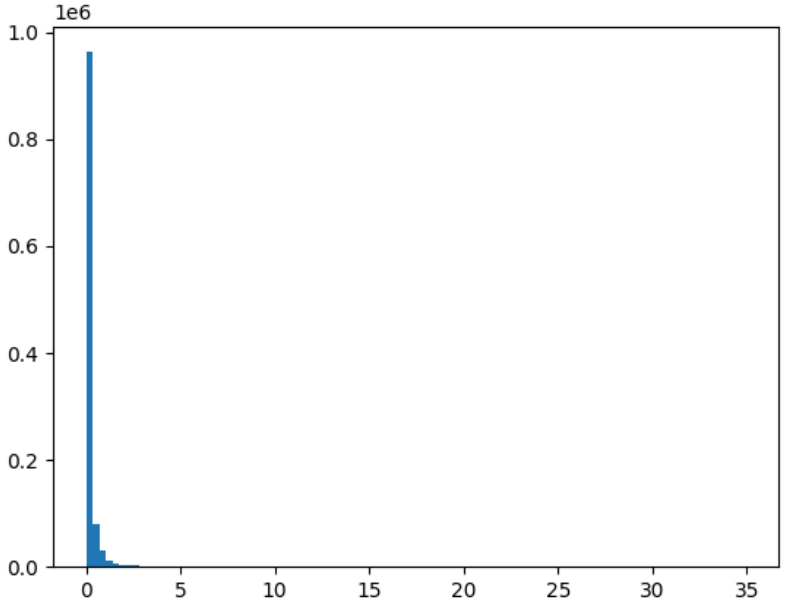
כפי שניתן לראות הרוב הגדול של הניקודים הוא 0 או שואף קרוב מאוד ל-0. התפלגות כזו הינה בעייתית מאוד ללמידת המודל ממספר סיבות כגון לימוד לקוי של מקרים נדירים, לימוד לא מאוזן ולחיזוי קבוע של ניקוד 0. ניתן גם לראות שיש ערכים בעלי ערך גבוה (מעל 80), מה שרק מגדיל את הפער בין הנתונים השכיחים לנדירים, מה יגרום למודל לא להתחשב במצבים טובים ולא לנבא ניקודים גבוהים.

:No_fly_model



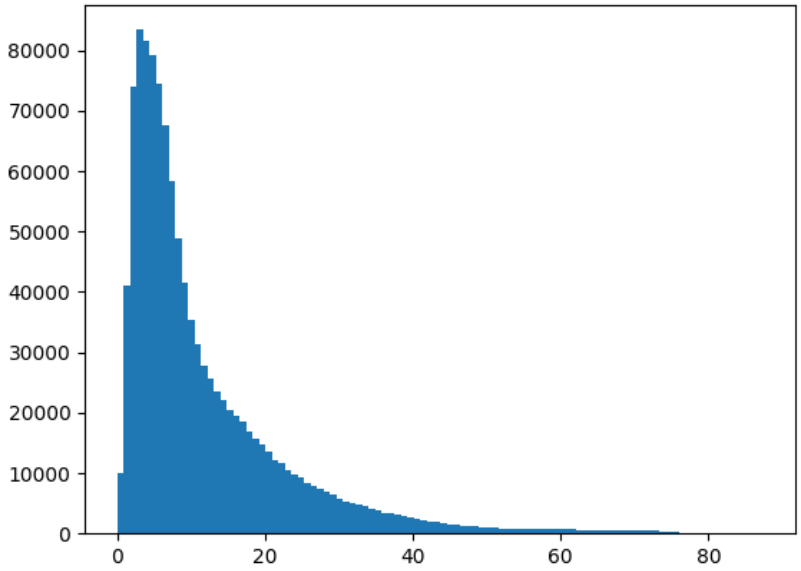
כפי שניתן לראות בגרף זה, אותה בעיה של התפלגות מרוכזת סביב ה-0 עולה גם מנתוני התזוזה ללא תעופה. ניתן היה לשער שהמודל לא יצליח ללמוד את הפונקציה והוא פשוט יחזיר 0 לכל לוח (יהיה ניתן לראות זאת בפרק התוצאות).

:Agent_fly_model



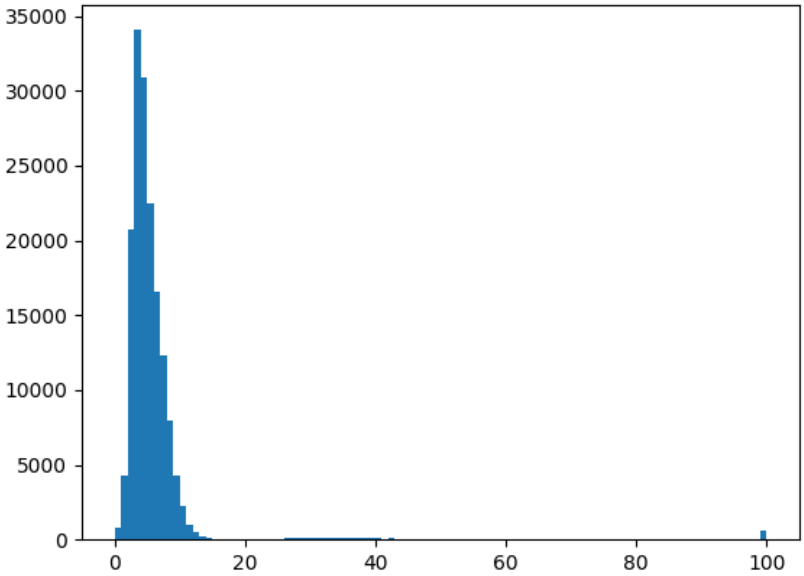
ניתן לראות שאותה בעיה נוכחת גם בנתונים של תזוזה עם תעופת סוכן.

:Player_fy_model



ניתן לראות שבנתונים של תזוזה עם תעופת שחקן התפלגות הנתונים יותר נוחה למודל ללמוד. בהתפלגות אידיאלית זו רוב הדוגמאות מרוכזות בטווח 0–20, מה שמבטיח שיש למודל כמות מספקת של דוגמאות "קלות" כדי ללמוד תבניות בסיסיות של מצב הלוח.
הירידה ההדרגתית בהיסטוגרמה מעבר ל-20 מספקת דווקא דיווח על המקרים ה"קשים" יותר, כך שהדוגמאות עם ערכים גבוהים עדיין משאירות חותם בלמידה ולא נעלמות בגלל שכל השאר קרוב לאפס. התוצאה היא loss מאוזן יותר לאורך כל טווח הערכים, ויכולת הכללה משופרת לחיזוי מדויק גם של מצבים נדירים וחשובים.

:All_fly_model



ניתן לראות שההתפלגות של הנתונים עדיין מאוד מרוכזת סביב ה-0, אך היא יותר רחבה משאר המודלים עם בעיה זו, לכן השערתי הייתה שהמודל יחזיר ערכים שהם לא 0, אך המודל אינו יהיה חכם כמו שהוא יכול להיות.

ארכיטקטורת המודלים:

ארכיטקטורת המודלים הייתה זהה בין כל חמש המודלים:

שכבות:

1. שכבה של 128 נוירונים עם פונקציית אקטיבציה Relu שהיא ליניארית בערכים גדולים מ-0 ושווה 0 בערכים שליליים.
2. שכבה של 64 נוירונים עם פונקציית אקטיבציה Relu.
3. שכבה של 32 נוירונים עם פונקציית אקטיבציה Relu.
4. שכבת פלט בעלת נוירון 1 עם פונקציית אקטיבציה ליניארית.

קומפילציה:

- Optimizer: adam
- Loss: mean absolute error

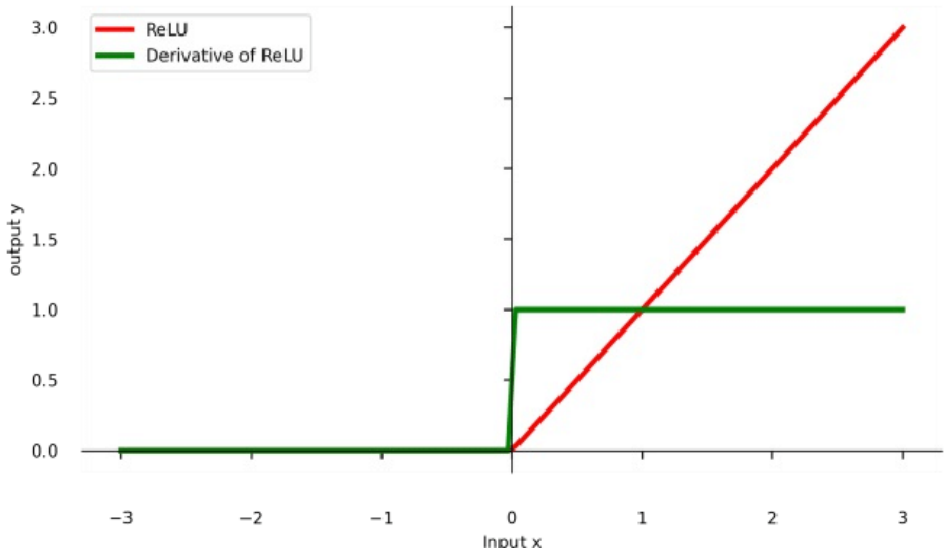
אימון:

- Epochs: 7 אפוקים
- Batch_size: 32
- Callbacks: עצירה מוקדמת של המודל אחרי שהloss לא משתפר 3 אפוקים ברציפות (Early stopping).

הסברים על הארכיטקטורה:

פונקציית Relu: relu היא פונקציית אקטיבציה שמוגדרת כך: Relu(x) = max(0, x), כלומר היא ליניארית לגמרי עם שיפוע 1 בערכים הגדולים מ-0 והיא שווה ל-0 בערכים שליליים.
הליניאריות של פונקציית ה-relu מונעת מהערכים ברשת להתקרב ל-0 ולהחזיר ערכים בעלי משקל מזערי לאורך רשת גדולה.

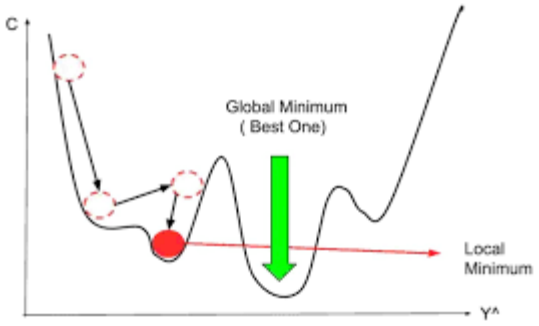
בנוסף, איפוס כל הערכים השליליים גורם לכך שבערך חצי מהנוירונים מעשית כבויים, מה שגורם ל"דילול" (sparsity) של הרשת. היתרונות בדילול זה הם שזה גורם לרשת להיות פחות רגישה ל"רעש רקע" בנתונים, ומשפר את כלליות (generalization) הרשת משום שהכיבוי משמש כרגולציה (regularization) שעוזרת למנוע overfitting.



<u>Adam:</u> adam הוא הoptimizer בו בחרתי להשתמש בכל המודלים שלי. המטרה של הoptimizer היא למצוא את המינימום הגלובלי של פונקציית ה-loss, כלומר למצוא את הערכים של המשקולות שייצרו את השגיאה הקטנה ביותר שהמודל יכול להגיע אליה.

בoptimizer ליניארי יש 2 בעיות עיקריות:

1. הlearning rate הקבוע עלול לגרום למודל "להיתקע" במקומות בהם הגרדיאנט אפסי (plateau) או במינימום מקומי.
2. השיפוע של הפונקציה על פי כל פרמטר היא שונה, לכן יש לייחס learning rate שונה לכל פרמטר, מה שoptimizer ליניארי לא עושה.2 הבעיות האלה נפתרות על ידי האלגוריתם של adam. מה שadam עושה הוא מייחס learning rate נפרד לכל משקולת על ידי חילוקו בגרדיאנטים קודמים בריבוע, כלומר כאשר הגרדיאנט היה מאוד קרוב ל-0 ("שטוח"), הlearning rate יהיה גדול מאוד, ולהיפך.



Loss: פונקציית loss היא דרך למדוד את האיכות של מודל במטרה לשפר אותו כך שהשגיאה מינימלית. הפונקציה מחשבת את הפער בין הערך שחוזה על ידי הרשת לבין הערך האמיתי. המטרה בלמידה היא לשנות את המשקולות ואת biasו כך שהloss בסוף הוא נמוך ככל האפשר, כלומר הכי קרוב לתוצאה האמיתית שניתן. בלמידה, המודל משתמש בנגזרת הפונקציה כדי לעדכן את הפרמטרים במטרה למצוא את המינימום של הפונקציה.

פונקציית הloss בה אני השתמשתי היא mean absolute error, שהיא מחשבת שגיאה על ידי חישוב הערך המוחלט של ההפרש בין הערך שהרשת חזתה לבין הערך האמיתי. בחרתי בפונקציה זו משום שערך השגיאה בה הוא ליניארי, כלומר שגיאה גדולה לא "תעניש" את הרשת באופן מוגבר באופן מעריכי. במילונים שלי רוב הערכים קרובים ל-0, אך יש גם ערכים גבוהים (כמו 100 ללוח ניצחון) לכן פונקציה כמו mean squared error לא הייתה מתאימה לנתונים שלי.

$$\mathrm{MAE} = \underbrace{\frac{1}{n}\sum_{i=1}^{n}}_{\text{test set}} \left|\underbrace{y_i}_{\text{predicted vaue}} - \underbrace{\hat{y}_i}_{\text{actual value}}\right|$$

Epoch: אפוק מסמל מעבר מלא אחד על כל הנתונים בdataset באימון הרשת. בepoch אחד הרשת עוברת על כל הדוגמאות פעם אחת ולבסוף מעדכנת את פרמטרים בהתאם.

אימון על מספר גבוה מדי של אפוקים עלול לגרום לoverfitting, כלומר להתלבשות פונקציית הרשת באופן מדויק מדי רק לנתוני האימון כך שהloss על נתוני הוולידציה גדל. השתמשתי ב20 אפוקים משום שלא דאגתי מoverfitting כתוצאה מהearly stopping שהכנסתי לרשת.

Batch size: מספר שורות הנתונים שעוברים ברשת לפני שהיא מבצעת חישוב של הloss ומעדכנת את המשקולות. כלומר כאשר ה batch size הוא 32 כמו במודל שלי, אחרי כל 32 שורות נתונים, הloss מתעדכן והמשקולות מתעדכנות גם הן.

Early stopping: אלגוריתם עצירה ללימוד הרשת שמטרתו היא למנוע overfitting של המודל. אלגוריתם זה עוצר את למידת המודל אחרי כמות קבועה של פעמים רצופות בהן הloss לא משתפר, ושומר את המודל הטוב ביותר בין האפוקים שכן הושלמו.

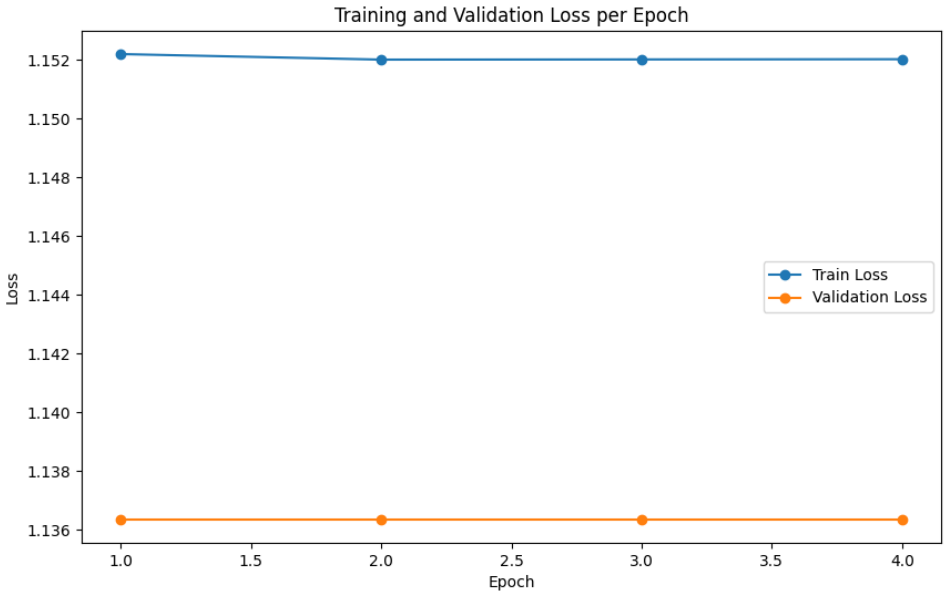
תוצאות:

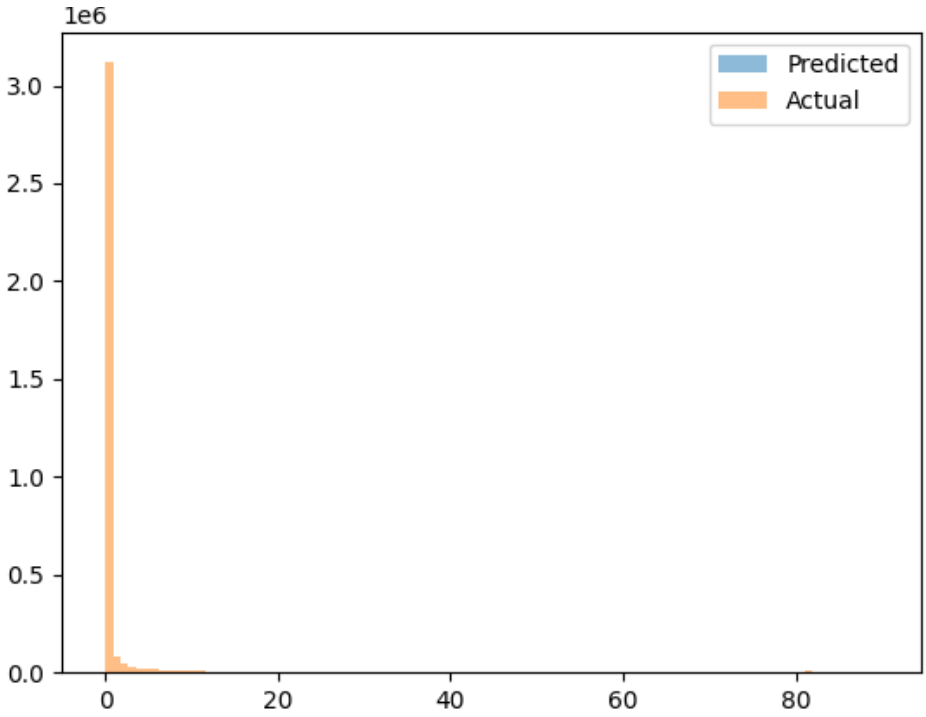
את התוצאות של המודלים אני ייצגתי באמצעות 2 גרפים: גרף של התפתחות הloss וגרף של השוואה בין התפלגות הניקודים של הtrain לבין התפלגות הפלטים של הרשת על הtest.

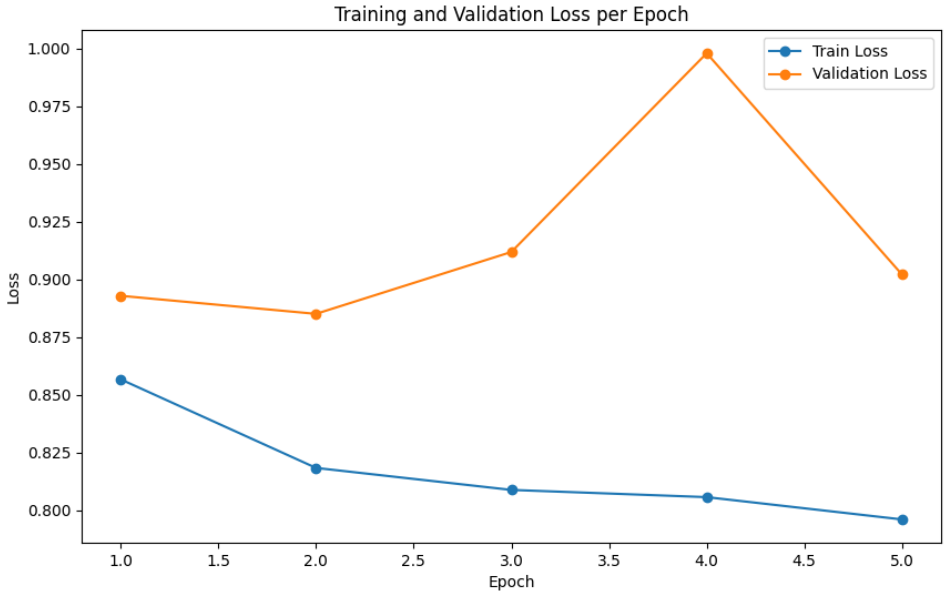
:Place model

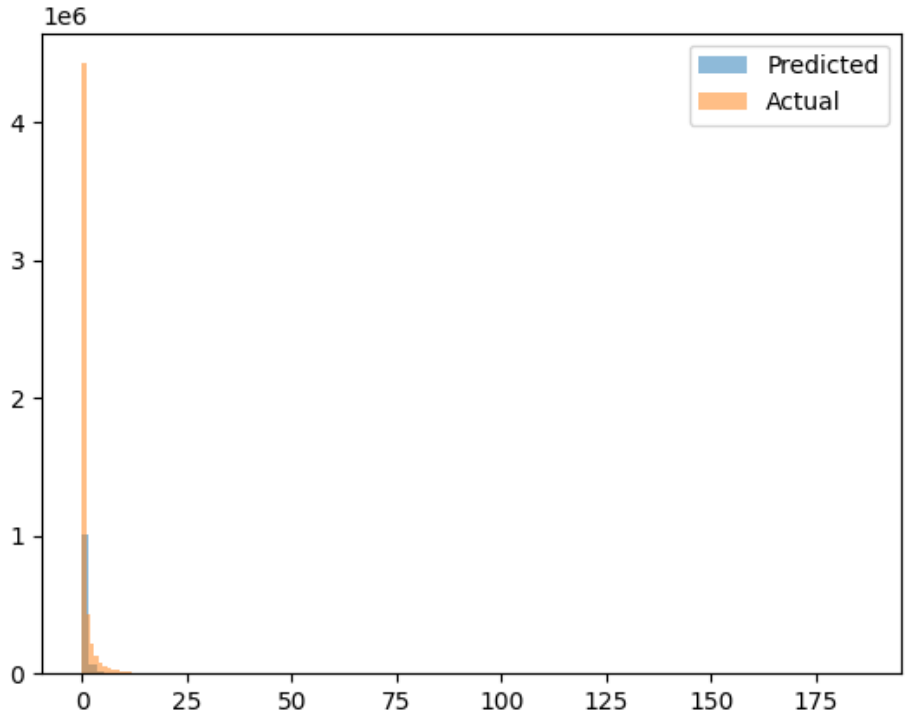
ניתן לראות מהגרף הראשון שהloss לא השתנה, ושהמודל למד רק 4 אפוקים בלבד. כלומר המודל לא באמת למד משהו בשלב האימון. כאשר ניסיתי את המודל, על כל לוח שקיבל הוא החזיר ניקוד 0. המודל למד שעל רוב הערכים שהוא מקבל, הוא צריך להחזיר 0, אך הוא לא למד להתחשב בערכי קיצון (שונים מ-0).

:No Fly Model



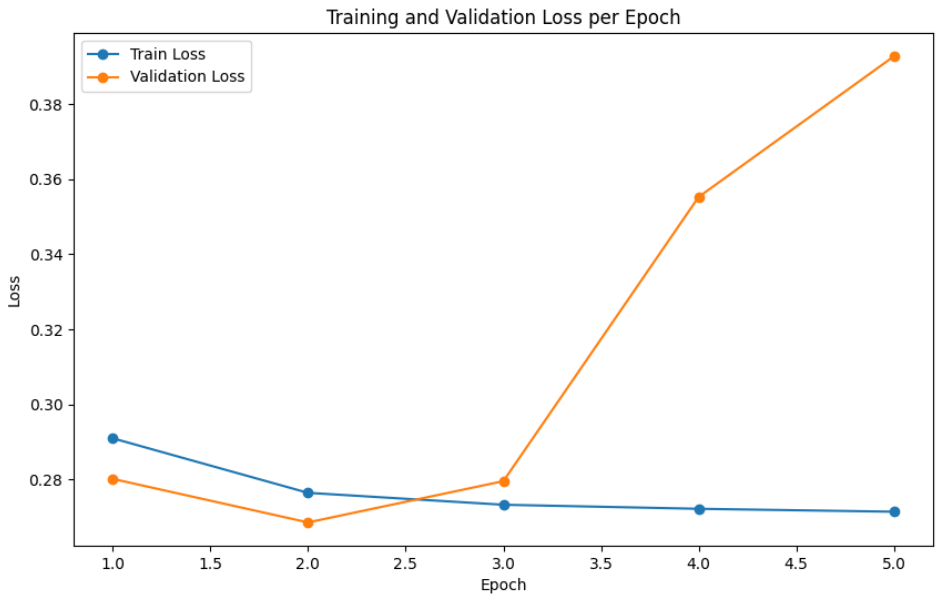


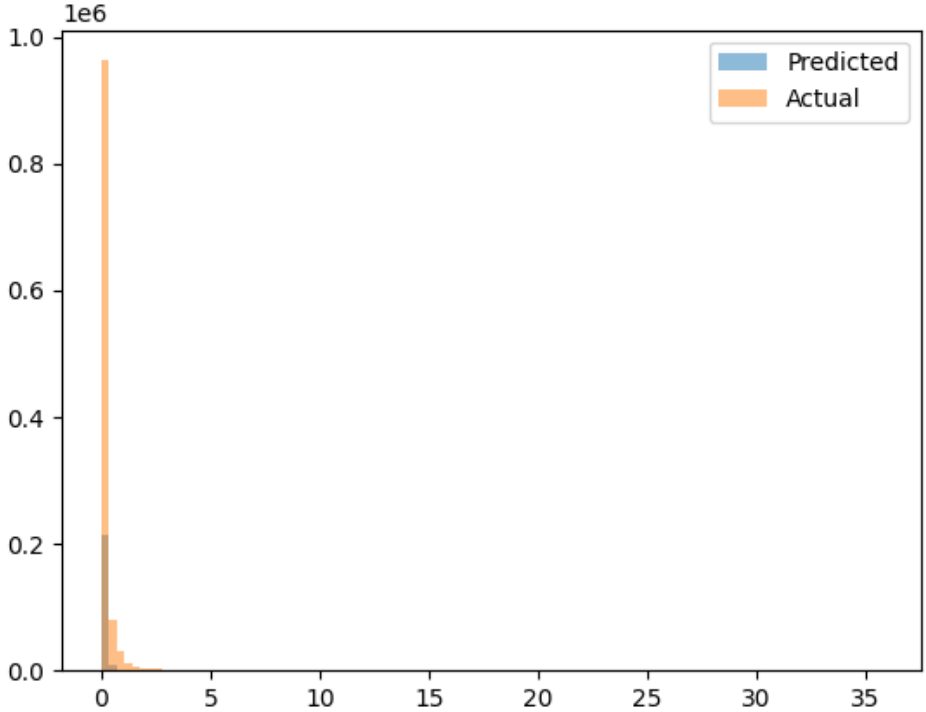




ניתן לראות על פי הגרפים האלה שהמודל הצליח להקטין את הloss של נתוני האימון, אך הוא לא הצליח להוריד את הloss של הtest באופן משמעותי, כלומר הוא אינו מדוייק. לפי הגרף השני ניתן לראות שהתפלגות הניקודים של הpredict על נתוני הtest דומים להתפלגות הניקודים של הtrain, אך הוא עדיין לא יודע להתמודד טוב עם מקרי קיצון. ניתן גם לראות שהמודל ניבא ניקודים גבוהים מ-100, למרות שהניקוד המקסימלי בנתוני האימון הוא רק 100.

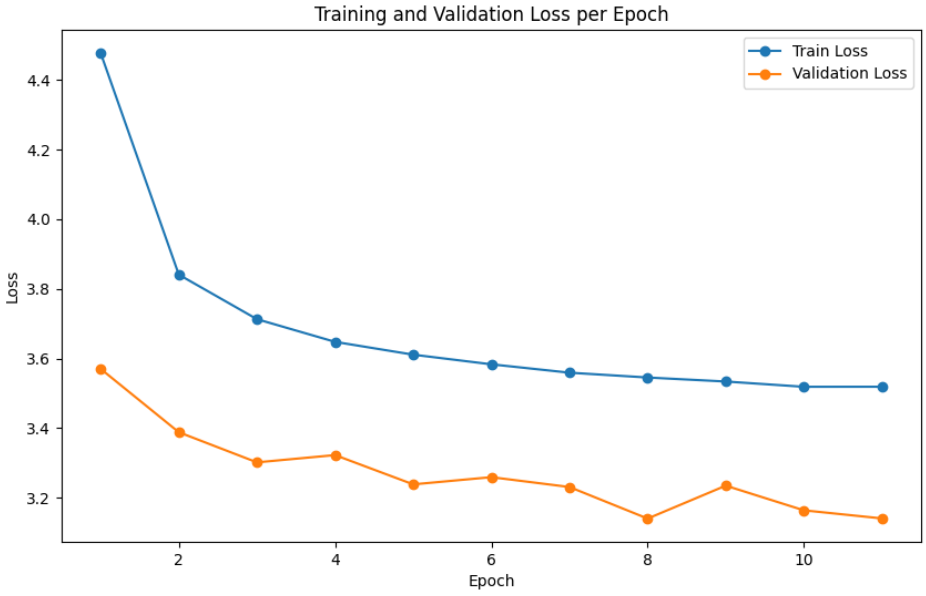
:Agent Fly Model

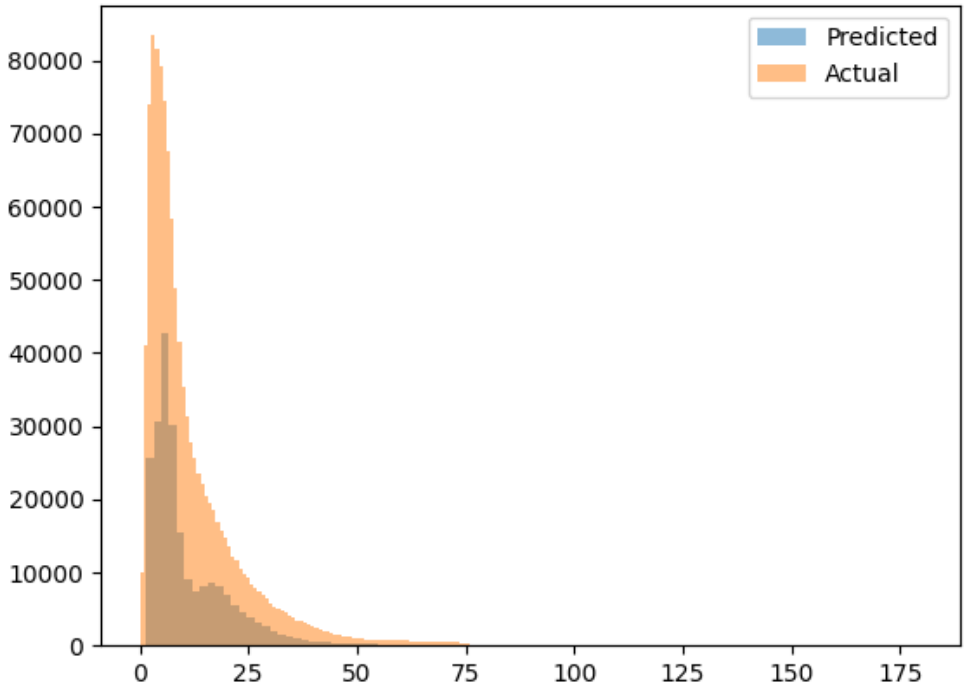




בדומה לגרפים הקודמים, המודל למד טוב את הtrain, אבל הloss של הtest לא מאוד השתפר עם האפוקים. המודל יודע להחזיר 0 על הרוב הגדול של הערכים שהוא מקבל, מה ששוקף גם בבדיקות שלי על המודל, בהן הוא החזיר 0 על כל ערך שקיבל.

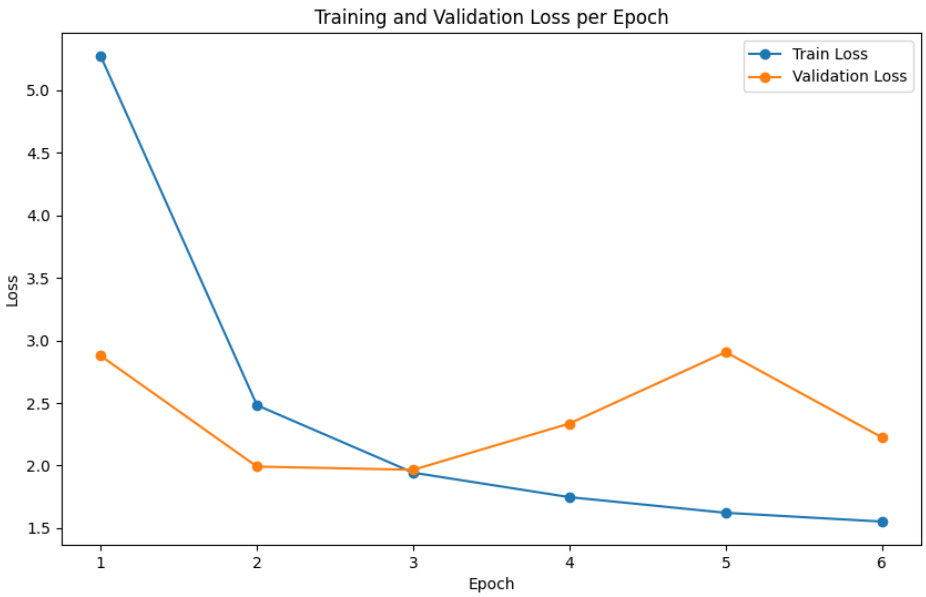
:Player Fly Model

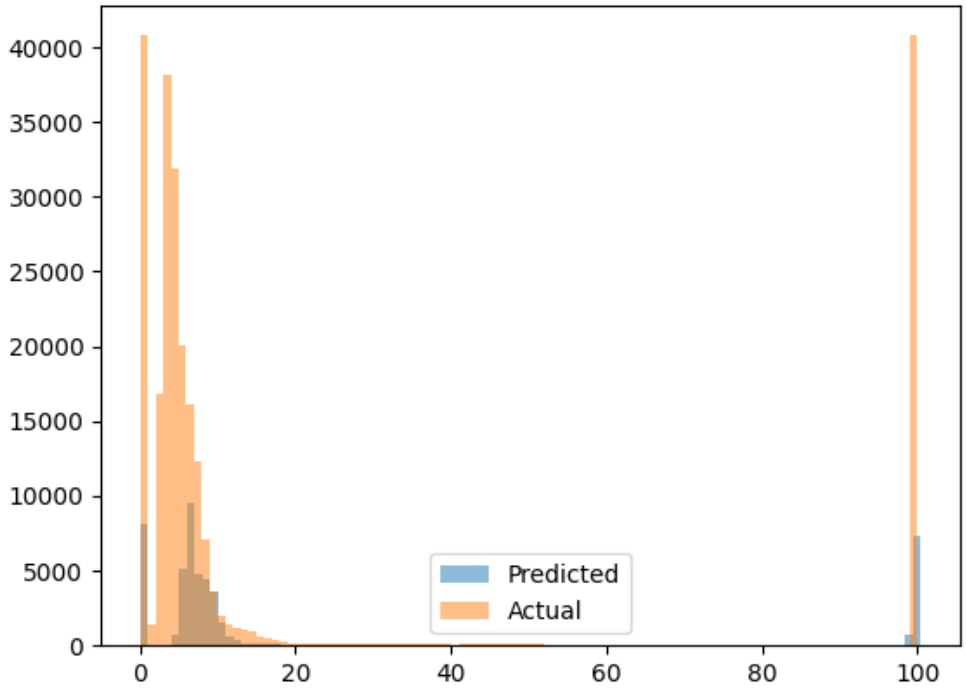




בגרפים אלה ניתן לראות שהמודל הצליח ללמוד את הפומקציה של המילון הזה. הloss של גם הtrain וגם הtest ירד בהדרגתיות, והתפלגות הנתונים של הpredict דומה לנתוני האימון.

:All Fly Model





בגרף זה, בדומה לגרפים אחרים המודל לומד טוב על נתוני האימון, אך הloss על נתוני הtest ממשיך להיות לא יציב ולא לרדת כמו שהוא אמור. לעומת זאת, התפלגות ה-predict על הtest מאוד דומה להתפלגות הtrain, מה שהתבטא גם בבדיקות שלי בהן המודל החזיר ניקוד שהוא יחסית מדויק.

מסקנות:

ההתפלגות הקיצונית מסביב ל-0 ברוב המודלים, גרמו לכך שאותם מודלים החזירו ניקוד 0 על כל לוח שקיבלו. ניתן להסיק שזו הייתה הבעיה משום שבמודלים שלמדו על נתונים בעלי התפלגות יותר מפוזרת ומסודרת, הצליחו ללמוד יותר טוב, והחזירו ערכים הגיוניים ששונים מ-0.

כדי לשפר את המודל הבא צריך לגרום להתפלגות הנתונים להיות יותר מסודרת כדי להקל על למידת המודל. בנוסף, אפשר גם להוריד את כל הנתונים בהן הניקוד הוא 0 משום שערכים אלה לא תורמים למודל לנצח את השחקן, אלא הם רק פוגעים בלמידת המודל. נוסף על זאת, הרחבת הbatch size עשויה לשפר את הכלליות של הרשת (generalization) ולהפחית overfitting.

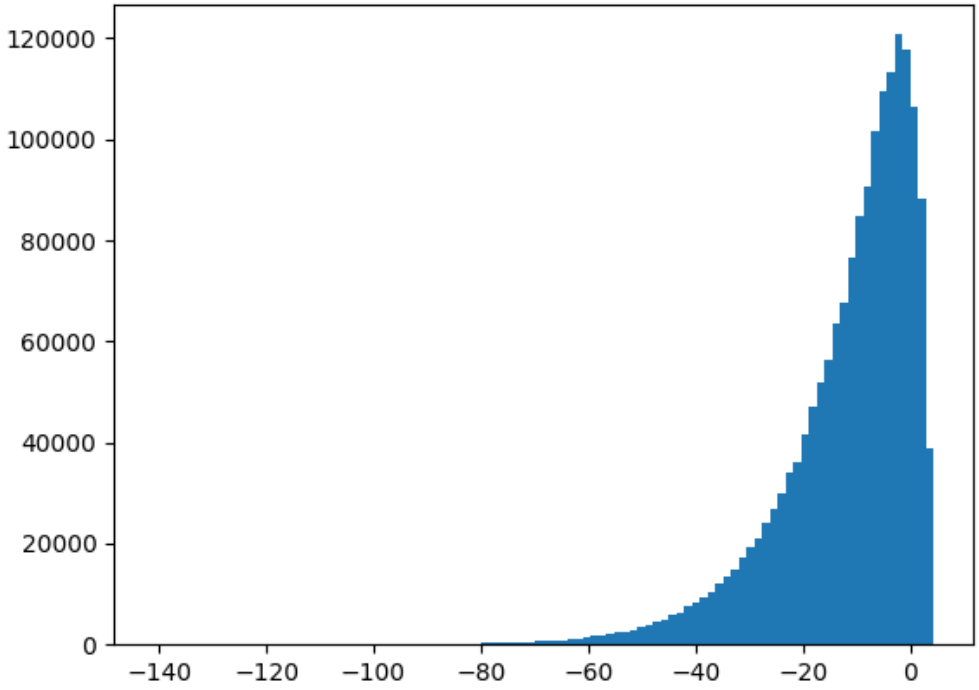
<u>הרשת הסופית:</u>

<u>השינויים בין הרשתות:</u>

1. הורדתי את כל השורות הנתונים במילונים בהן הניקוד הוא 0 על מנת שהם לא יבלבלו את המודל ויגרמו לו להחזיר 0 משום ששורות אלה מאוד שכיחות בנתונים שלי.
2. בניתי מילונים מיוחדים בשביל הרשת במטרה לאזן את התפלגות הניקוד. במקום לשמור את הניקוד כמו שהוא, שמרתי את הlog (בסיס e) של הניקוד, מה שאיזן את התפלגות הנתונים והקל על למידת המודל.
3. העליתי את הbatch size ל128.

תיאור הרשתות:

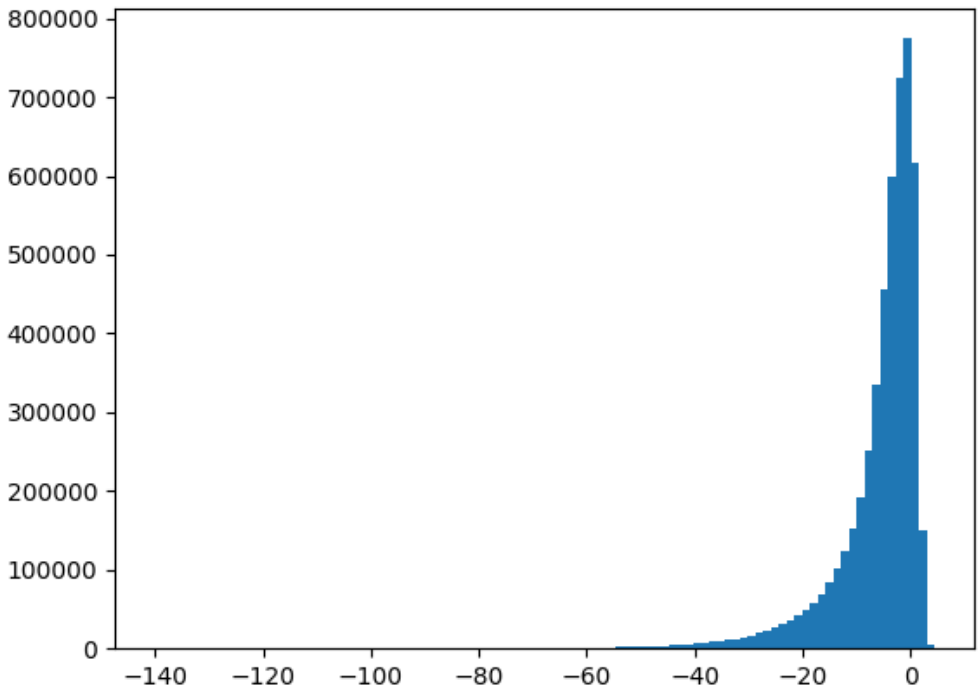
Place Model:



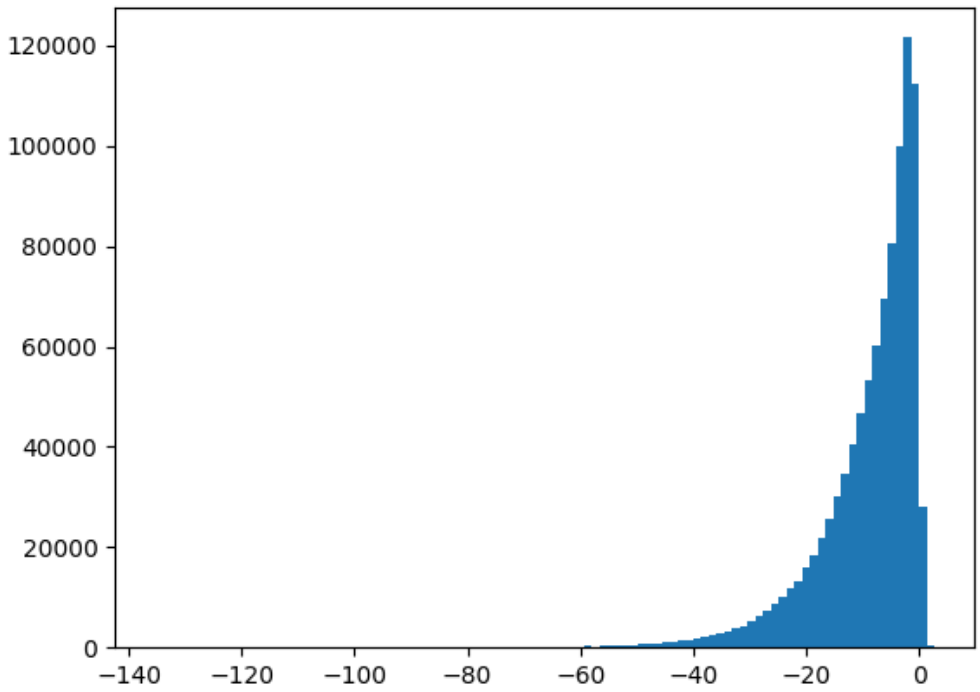
כפי שניתן לראות, התפלגות הנתונים הרבה יותר נוחה ללמידה. כל הנתונים יוצר מפוזרים ולא מרוכזים סביב מספר אחד. יש ערכים שליליים בגרף משום שlog לפי בסיס e הוא שלילי כאשר הx קטן מ-1. זאת לא בעיה משום שהlog היא פונקציה רציפה שתמיד עולה, לכן הסדר בין הניקודים נשמר.

תובנות אלו תקפות לשאר הגרפים בחלק זה של תיאור הרשת הסופית.

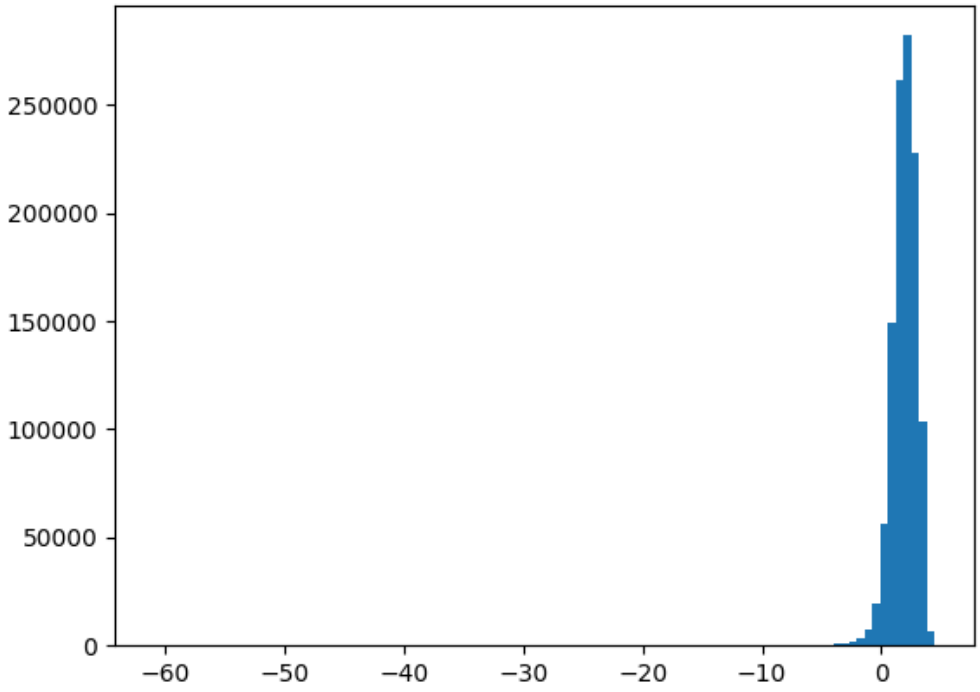
:No Fly Model



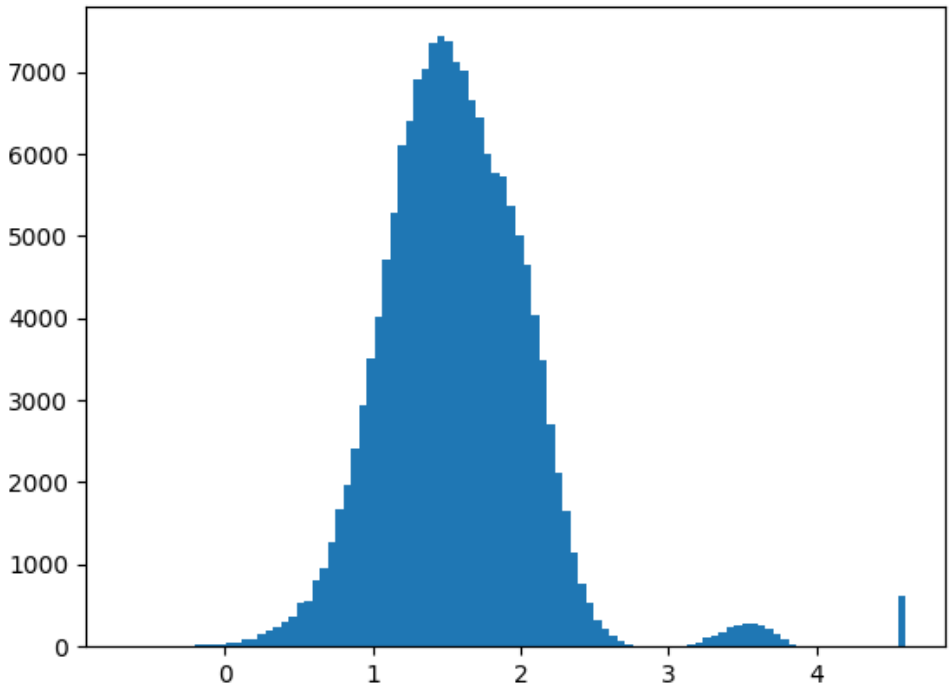
:Agent Fly Model



:Player Fly Model

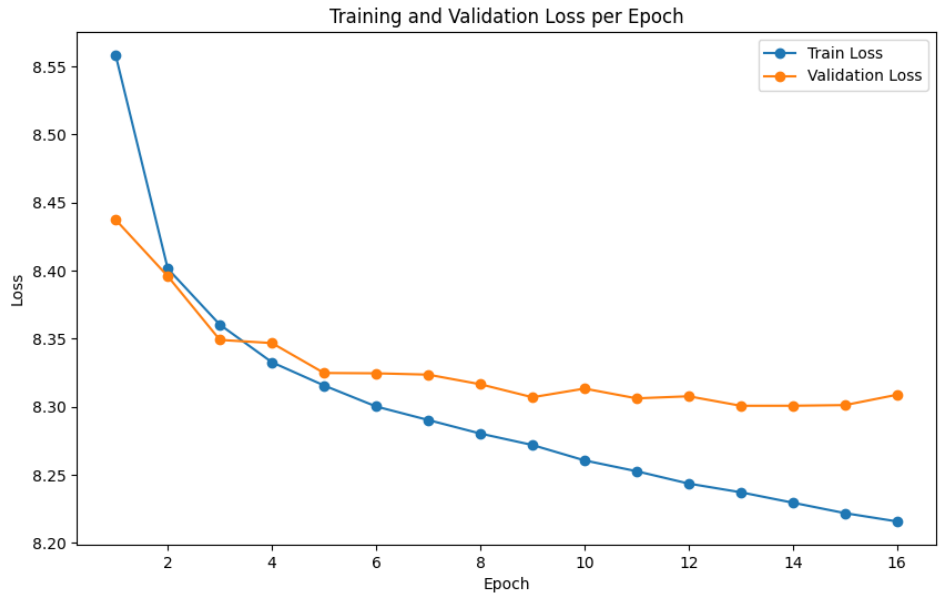


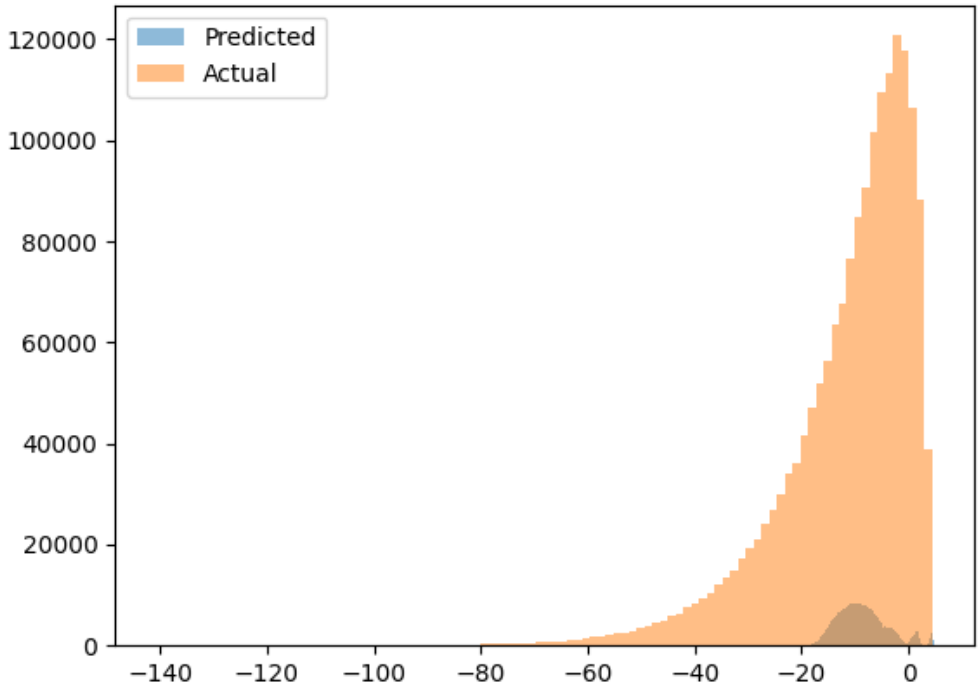
:All Fly Model



תוצאות המודל:

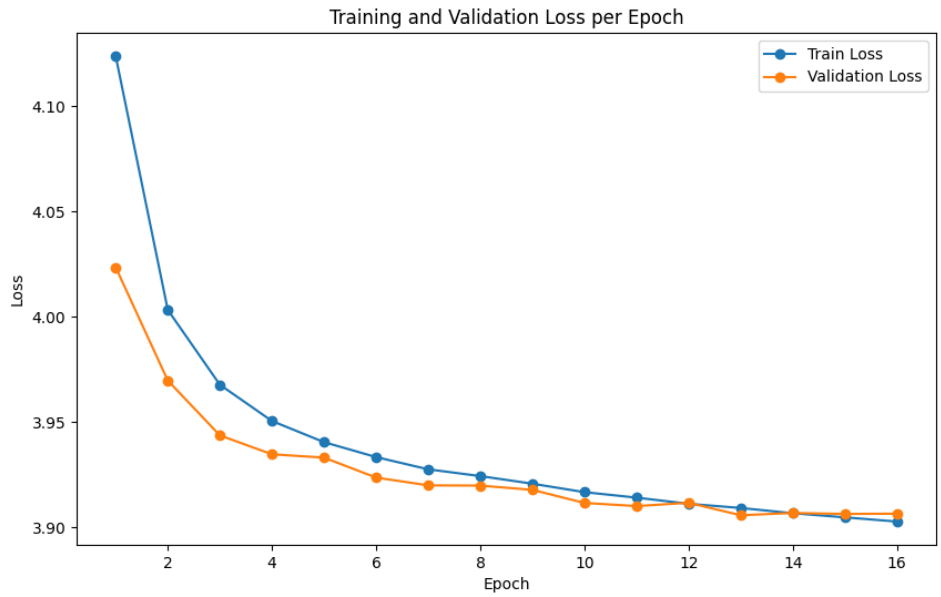
Place Model:

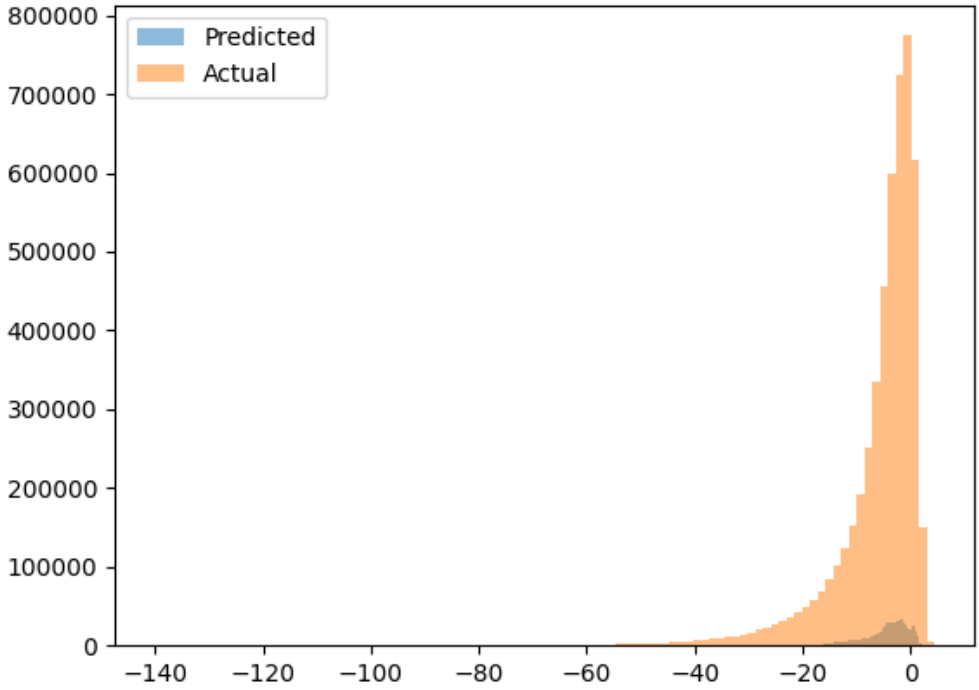




ניתן לראות שבזכות השינויים שעשיתי, הרשת הצליחה ללמוד את הפונקציה משום שהloss של הtest ירד. ניתן לראות גם שהמודל מחזיר ערכים שונים בהתפלגות דומה לנתוני האימון.

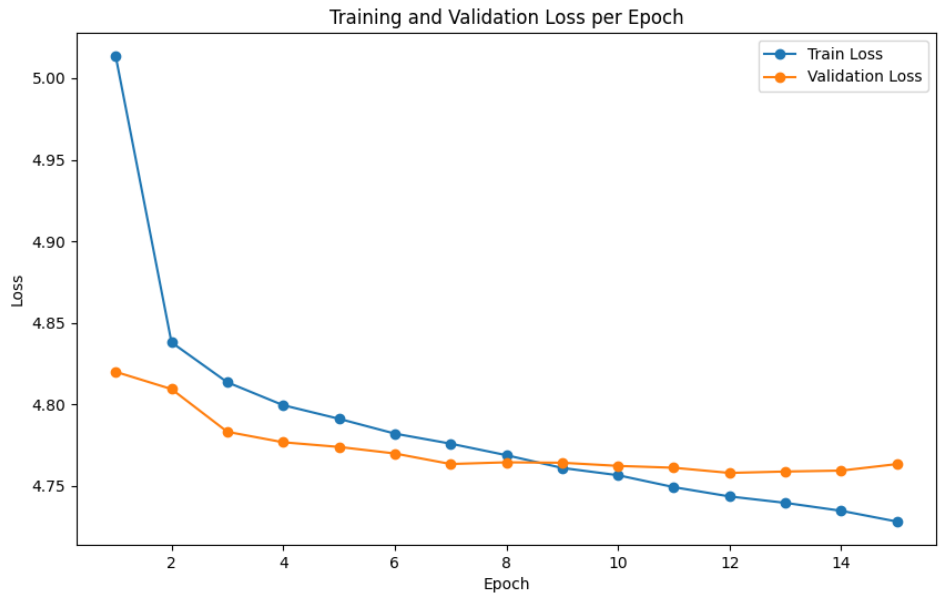
:No Fly Model

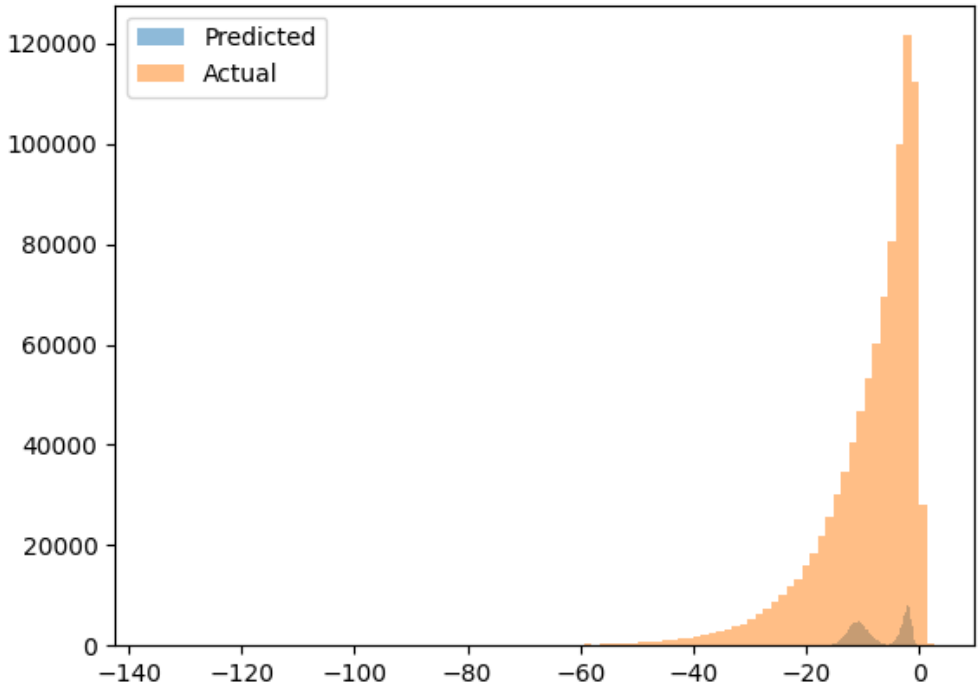




גם ברשת זו המודל הצליח ללמוד את הלוגיקה מאחורי שיעור הניקוד. הloss של הtest ירד יפה, וההתפלגות של הניקוד דומה להתפלגות המקורית בנתוני האימון.

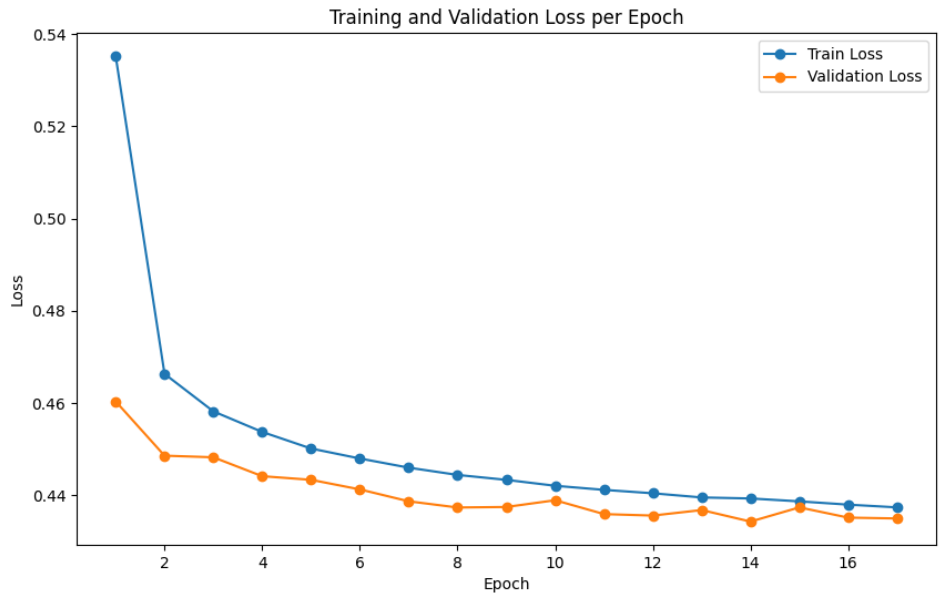
:Agent Fly Model

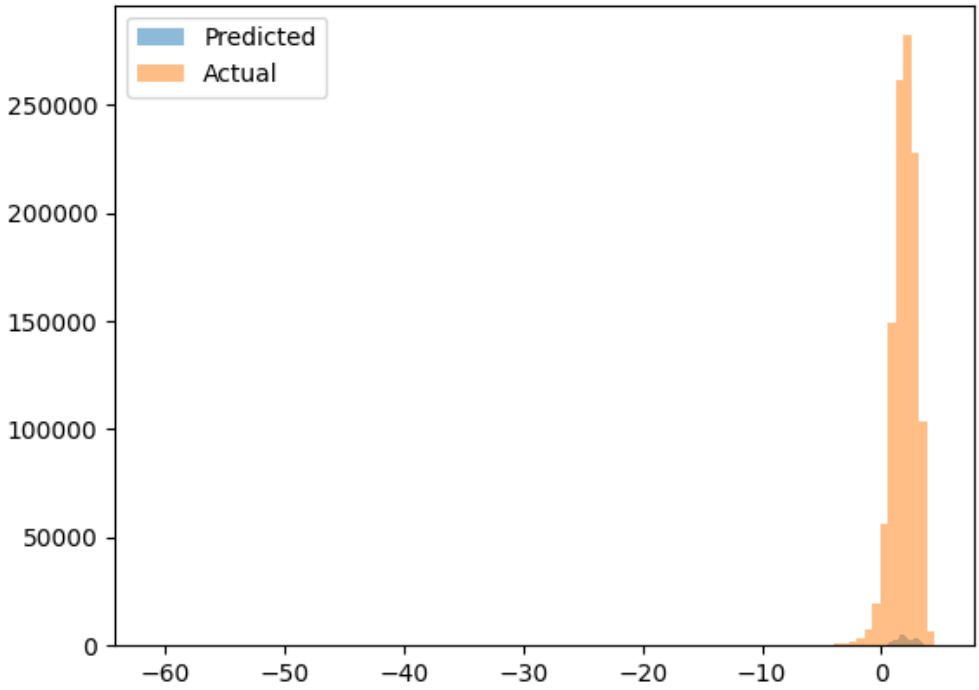




ניתן לראות שהתפלגות הנקודות של הpredict מרוכזת סביב 2 נקודות עיקריות. להשערתי, הניקוד הגבוה יותר שייך למצבי לוח בהם לשחקן יש 4 חיילים, והנמוך שייך למצבי לוח בהם לשחקן יש 5 חיילים.

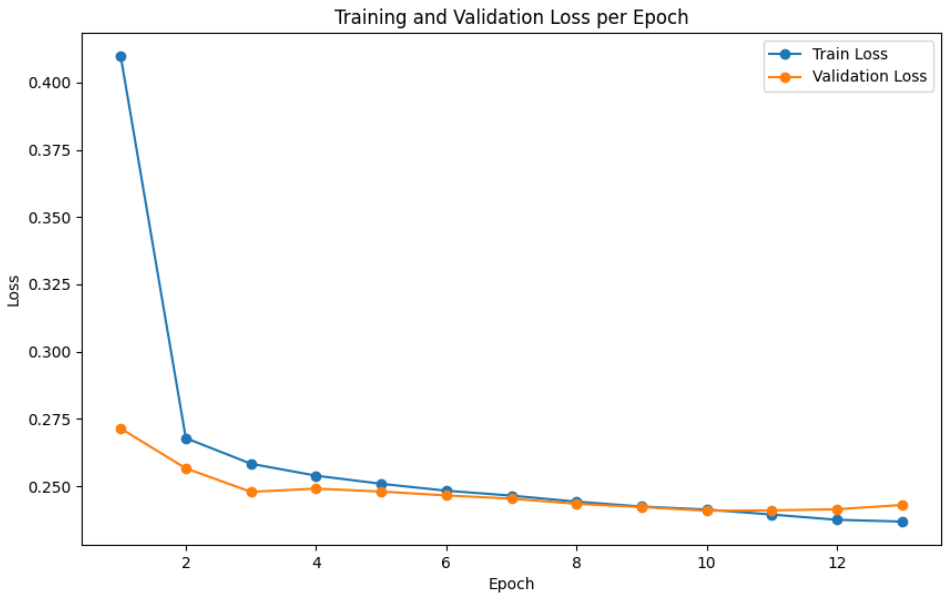
:Player Fly

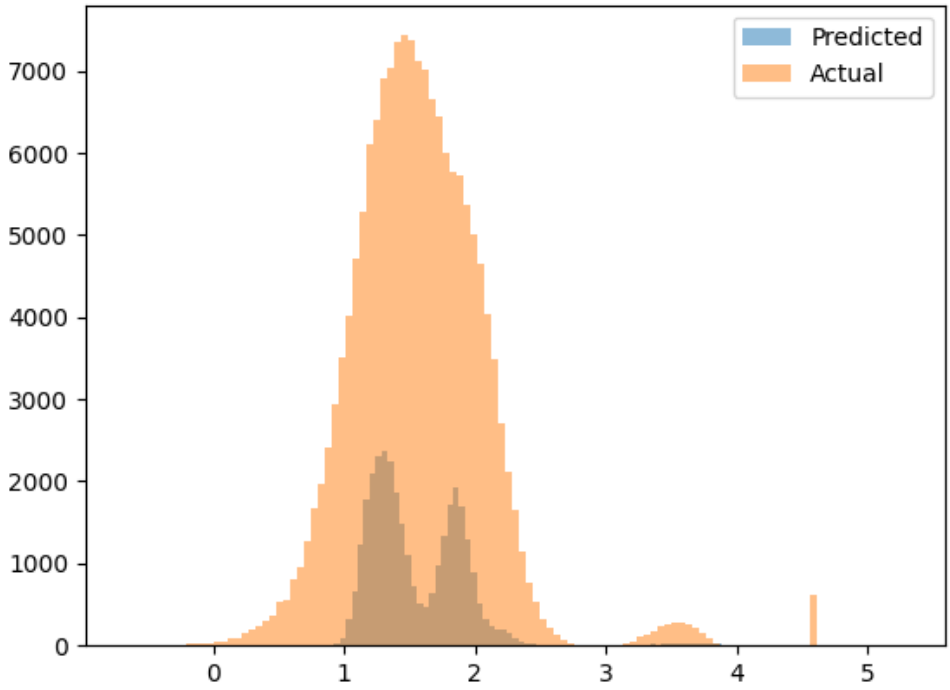




אפשר לראות שלמרות שרוב הניקודים יחסית מרוכזים סביב 0, המודל עדיין מחזיר ערכים שונים בהתפלגות דומה לזו של נתוני האימון, מה שמשוקף על ידי הloss הנמוך.

:All Fly Model





כמו בשאר הגרפים, הloss נמוך והתפלגות הניקודים של הpredict והtrain דומים, לכן ניתן להסיק שהמודל כן למד את הלוגיקה של מתן הניקוד ללוח.

יישום

מבוא:

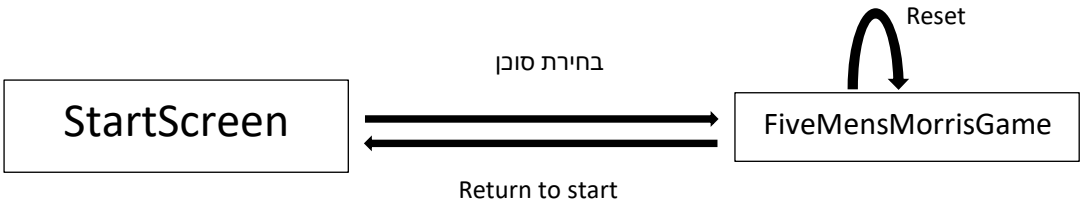
המטרה של היישום היא לאפשר משחק אינטראקטיבי של "5 Men's Morris" עם שחקן אנושי מול יריב מבוסס בינה מלאכותית. המערכת מאפשרת לבחור בין שלושה מצבי סוכן (רנדומלי, חכם, גאון) כאשר החכם מבוסס על למידה מחיזוקים ממילון (reinforcement learning), והגאון מתבסס על מודלים מאומנים בלמידה עמוקה (ANN).

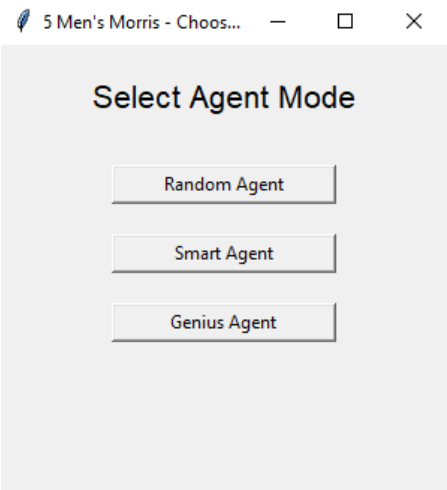
המשחק מתבסס על מודל :**MVC (Model-View-Controller)**

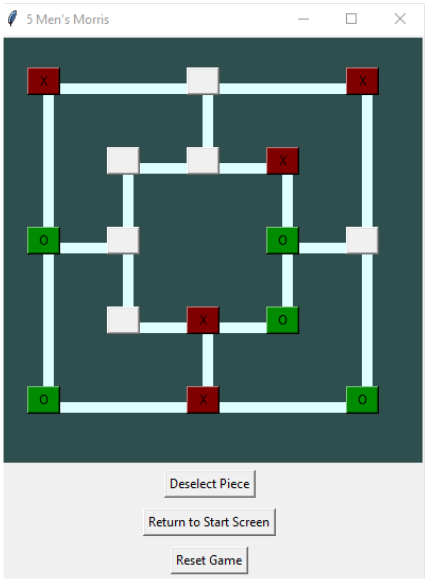
- **Model (Logic)**: אחראי על חוקי המשחק, מצב הלוח, תזוזות, ובדיקות ניצחון.
- **View (GUI - Tkinter)**: אחראי על הצגת הלוח, ממשק המשתמש, וכפתורים.
- **Controller**: משמש כמתווך בין הממשק הגרפי לבין הלוגיקה.

ממשק משתמש:

תרשים מסכים בגרפיקה:







הסבר על כל הכפתורים:

Random Agent: עובר למשחק נגד סוכן רנדומלי.

Smart Agent: עובר למשחק נגד סוכן רנדומלי

Genius Agent: עובר למשחק נגד סוכן רנדומלי

Return to Start Screen: חוזר מהמסך הראשי למסך הראשוני כדי לבחור סוכן

Reset Game: מאפס את המשחק הנתון

Deselect Piece: מבטל לחיצה על חייל

<u>תרשימי UML:</u>

<u>מחלקה: StartScreen</u>

תכונות:

שם	טיפוס	הסבר
root	tk.Tk	חלון השורש של Tkinter שבו מוצג המסך
label	tk.Label	כותרת חלון הפתיחה
random_btn	tk.Button	כפתור לבחירת "Random Agent"
smart_btn	tk.Button	כפתור לבחירת "Smart Agent"
genius_btn	tk.Button	כפתור לבחירת "Genius Agent"

פעולות:

שם	טענת כניסה	טענת יציאה / תיאור פעולה
__init__(self, root)	root: tk.Tk	בונה מסך התחלה בו בוחרים את סוג הסוכן.
start_game(self, mode)	mode: str("random", "smart", "genius")	סוגר את מסך ההתחלה, פותח חלון חדש, ומאתחל משחק במצב שהמשתמש בחר .

מחלקה: Graphics

מטרה – מנהלת את הגרפיקה של המשחק.

תכונות:

שם	טיפוס	הסבר
root	tk.Tk	חלון השורש של Tkinter שבו רץ המשחק
canvas	tk.Canvas	קנבס לציור הלוח
Player_sign	Char	הסימן שמסמל את השחקן
Agent_sign	char	הסימן שמסמל את הסוכן
Player_color	str	הצבע של השחקן
Agent_color	str	הצבע של הסוכן
controller	Controller	בקר המשחק
selected_piece	int או None	החייל שנבחר כרגע להזזה.
positions	list[tuple[int,int]]	רשימת קואורדינטות הכפתורים
buttons	list[tk.Button]	רשימת כל כפתורי התאים
reset_btn	tk.Button	כפתור לאיפוס המשחק
return_btn	tk.Button	כפתור לחזרה למסך ההתחלתי
Deselect_btn	Tk.Button	כפתור לביטול לחיצה

פעולות:

שם	טענת כניסה	טענת יציאה
__init__(self,root,agent_mode)	Root: tk.Tk Agent_mode: str	מאתחל את חלון הגרפיקה
Deselect_piece(self)	-	מבטל את הלחיצה על חייל
Reset_game(self)	-	מתחיל משחק חדש
Draw_board(self)	-	מצייר את לוח המשחק
Return_to_opening(self)	-	חוזר למסך ההתחלתי
Put_soldier_agent(self)	-	מניח חייל של הסוכן בגרפיקה וקורא לפעולה בלוגיקה על פי מצב סוכן.
Put_soldier_player(self,pos)	Pos: int	מניח חייל של השחקן בגרפיקה וקורא לפעולה בלוגיקה.
Handle_click(self, pos)	Pos: int	מגיב ללחיצה של השחקן
Move_soldier_agent(self,pos)	-	מזיז חייל של הסוכן בגרפיקה וקורא

		לפעולה בלוגיקה על פי מצב סוכן.
Move_soldier_player(self,pos)	Pos: int	מזיז חייל של השחקן בגרפיקה וקורא לפעולה בלוגיקה.
Remove_soldier_player(self,pos)	Pos: int	מוציא חייל של היריב בגרפיקה וקורא לפעולה בלוגיקה.
Check_regular_win(self)	-	בודק ניצחון על פי מספר חיילים
Check_jam_win(self)	-	בודק ניצחון על פי חסימה
Disable_buttons(self)	-	מכבה את כל הכפתורים על הלוח.

מחלקה: Controller

מטרה – משמשת כ - API בין הגרפיקה ללוגיקה.

תכונות:

שם	טיפוס	הסבר
Agent_mode	Str	מצב סוכן (רנדומלי, חכם מילון, חכם רשת)
Logic	Logic	עצם המנהל את לוגיקת המשחק.

פעולות:

שם	טענת כניסה	טענת יציאה
__init__(self,agent_mode)	Agent_mode: str	מאתחלת את המחלקה.
Get_game_phase(self)	-	מחזירה את שלב המשחק הנוכחי
Put_soldier_agent(self)	-	קוראת להנחת חייל בלוגיקה לפי מצב הסוכן ומחזיר את הנקודה לגרפיקה.
Restart(self)	-	קוראת לאיפוס הלוגיקה
Put_soldier_player(self,pos)	Pos – int	קוראת להנחת חייל של השחקן בלוגיקה ומחזירה את הנקודה.
Move_soldier_agent(self)	-	קוראת לתזוזת סוכן בלוגיקה על פי מצב סוכן ומחזירה את הצעד לגרפיקה.

Move_soldier_player(self,pos)	Pos: int	קוראת לתזוזת חייל של השחקן בלוגיקה ומחזירה את הצעד.
Remove_soldier_player(pos)	Pos: int	קוראת להוצאת חייל סוכן ומחזירה את הנקודה ממנה הוסר החייל.
Check_regular_win(self)	-	קורא לבדיקת ניצחון בלוגיקה ומחזירה את התוצאה לגרפיקה.
Check_jam_win_general(self)	-	קורא לבדיקת ניצחון חסימה בלוגיקה ומחזירה את התוצאה לגרפיקה.
Get_sector_owner(self, selected_piece)	Selected_piece: int	מחזיר את הבעלות של הנקודה.

מחלקה: Sector:

מטרה – מייצגת נקודה על הלוח

תכונות:

שם	טיפוס	הסבר
taken_by	int	מייצג מי מחזיק בסקטור (1=סוכן, -1=שחקן, 0=ריק)
all_mills	list[tuple[str]]	רשימת כל הטחנות שאפשר ליצור מנקודה זו
legal_moves	tuple[str,…]	רשימת הסקטורים אליהם מותר לזוז מהסקטור הנוכחי

פעולות:

שם	טענת כניסה	טענת יציאה / תיאור פעולה
__init__(self, mills, legal_moves)	mills: list[tuple[str]]legal_moves: tuple[str,…]	יוצר אובייקט Sector, מאתחל taken_by = 0, ושומר imills־ legal_moves

מחלקה: Logic

מטרה: מנהלת את לוגיקת המשחק (חוקים, מצב לוח וכו'...)

תכונות:

שם	טיפוס	הסבר
Board	Dict	ייצוג הלוח
Game_phase	Int	שלב המשחק (הנחה, הזזה, הוצאה)

מספר הנקודות של השחקן	Int	Player_sectors
מספר הנקודות של הסוכן	Int	Agent_sectors
רשימה של נקודות השחקן	List	Player_taken_sectors
רשימה של נקודות הסוכן	List	Agent_taken_sectors
רשימה של נקודות ריקות	List	Empty_sectors
מספר הטור	Int	Turn
מילונים של למידה על ידי חיזוקים לסוכן "חכם"	Dict	…_dict
מודלים של ANN לסוכן "גאון"	Keras.Model	…_model

פעולות:

שם	טענת כניסה	טענת יציאה
__init__(self,agent_mode)	Agent_mode: str	מאתחל את לוגיקת המשחק לפי מצב סוכן
Restart(self)	-	מאתחל משחק חדש
Check_mill(self, s)	S: char	בודק עם יש טחנה בנקודה S
Get_removable_opponent_sectors(self, remover)	Remover: int	מחזיר רשימה של כל הנקודות שאפשר להוציא מהן חייל של היריב.
Put_soldier_agent(self)	-	מניח חייל של הסוכן באופן רנדומלי.
Put_soldier_player(self, pos)	Pos: char	מניח חייל של השחקן בנקודה pos.
Move_soldier_agent(self)	-	מזיז חייל של הסוכן באופן רנדומלי
Move_soldier_player(self, from_letter, to_letter)	From letter: char To_letter: char	מזיז חייל של השחקן מנקודה from_sector לנקודה to_sector.
Remove_soldier_player(self,pos)	Pos: char	מוציא מנקודה S חייל של הסוכן.
Check_regular_win(self)	-	בודק עם יש ניצחון לפי הוצאת חיילים (1: סוכן, 0: אין, -1: שחקן)
Check_legal_moves(self, sec)	Sec: char	מחזיר true עם יש מהלכים חוקיים

מנקודה sec ו false אחרת.		
בודק עם הסוכן חסום לגמרי (הפסד בחסימה)	-	Check_jam_win_agent(self)
בודק עם השחקן חסום לגמרי (הפסד בחסימה)	-	Check_jam_win_player(self)
בודק עם יש ניצחון לפי חסימה (1: סוכן, 0: אין, 1-: שחקן)		Check_jam_win_general(self)
מחזיר ייצוג של מצב לוח כמחרוזת (סוכן: O, ריק: -, שחקן: X)	Sector_dict: dict	Convert_taken_by_to_string(self, sector_dict)
בודק עם יש טחנה בנקודה S בלוח board.	S: char Board: dict	Check_mil_with_board(self,s, board)
בודק עם לשים חייל בנקודה move_sector יחסום טחנה פוטנציאלית של היריב.	Board: dict Move_sector: char	Move_blocks_opponent(self, board, move_sector)
מניח חייל של הסוכן על פי מילון למידה מחיזקים.	-	Smart_put_soldier_agent(self)
מזיז חייל של הסוכן על פי מילוני למידה מחיזקים	-	Smart_move_soldier_agent(self)
מוציא חייל של השחקן על פי מילוני למידה מחיזקים	-	Smart_remove(self)
מחזיר את מספר המילון המתאים לפי מצב הלוח (0-place, 1-no_fly, 2-agent_fly, 3-player_fly, 4-all_fly)	Board: dict	Choose_dict_from_board(self, board)
הופך מחרוזת לוח לרשימת numpy (1:X , -: 0, O: 1-)	S: str	Convert_string(self,s)
מניח חייל של הסוכן על פי מודל ANN.	-	Genius_put_soldier_agent(self)

Genius_move_soldier_agent(self)	-	מזיז חייל של הסוכן על פי מודלים של ANN.
Genius_remove(self)	-	מוציא חייל של השחקן על פי מודלים של ANN.

מתודולוגיית MVC:

View (הממשק הגרפי)

StartScreen:

- מציג חלון לבחירת מצב סוכן
- בלחיצה – סוגר את המסך ההתחלתי ופותח מסך משחק (FiveMensMorrisGame) על פי מצב הסוכן הנבחר על ידי המשתמש.

FiveMensMorrisGame:

- מציג את חלון משחק ובונה את הלוח
- הלוח מורכב מ-16 נקודות המפוזרות על הלוח כשני ריבועים בגודל 3X3 ללא מרכזים, וביניהם יש קוים המחבריאת הנקודות הצמודות בריבועים ואת האמצעים של צלעות הריבועים.
- מציג את סימוני החיילים כך שהסוכן הוא עיגול ירוק והשחקן הוא איקס אדום.
- אחריות ה־ View מוגבלת להצגת צבעים, תמונות, ולטיפול בלחיצה על הכפתורים – אך לא לבחירת מהלכים או אימות חוקיות.

Controller (המתווך בין הלוגיקה לגרפיקה):

קישור:

- בונה עצם Logic (ה-Model)
- מספק API פשוט ל־View: הView יכול לפנות לModel רק על ידי פנייה מעצם הController על ידי שימוש בפומקציות המובנות בו.

בידוד:

- ה־ View **לא יודעת** על מבנה הנתונים הפנימי של ה-Model.
- ה־ Model **לא יודעת** על מבנה הנתונים הפנימי של ה-View.

ניהול זרימה:

- ה־ View שולח אירועי לחיצה ל־ Controller.
- ה־ Controller מעביר את האירוע ל־ Logic.
- לאחר עדכון ה-Model, הController מחזיר את המידע הרלוונטי על מצב הלוח בחזרה לView, אשר מעדכן את התצוגה כהוגן.

Model (הלוגיקה)

מבנה נתונים:

- Self.board: מילון של 16 עצמים מטיפוס Sector.

ניהול מצב:

- הלוגיקה אחראית על ניהול שלבי המשחק (הנחה, תזוזה, תעופה) אצל השחקן והסוכן.
- היא שומרת את מספר החיילים של כל שחקן בנפרד.

למידת מכונה:

- במצב "חכם" נטענים מילונים מקבצי JSON שמשמשים לחיפוש המהלכים הכי טובים (ניקוד הכי גבוה)
- במצב "גאון" נטענים מודלי ANN שמשמשים לבחירת המהלכים הכי טובים (מחשבת ניקוד לכל מצב לוח אפשרי).

האינטראקציה בין העצמים:

1. המשתמש בוחר מצב שחקן בחלון ה-StartScreen ומועבר למשחק.
2. השחקן לוחץ על כפתור בממשק הגרפי, הView מעביר את הלחיצה לController.
3. הController מעביר את הלחיצה לModel.
4. הModel מבצע את הפעולה הנדרשת ומעדכן את התכונות הנדרשות.
5. הController מקבל את תוצאות הפעולה ומעבירה אותה לView.
6. הView מעדכן את הגרפיקה בהתאם לפעולות שבוצעו.

יתרונות הגישה

- **הפרדה ברורה** :שינויים בלוגיקה לא דורשים שינוי בממשק הגרפי.
- **תחזוקה קלה** :כל מחלקה אחראית על תחום אחד בלבד.

רפלקציה:

במהלך עבודה על הפרויקט, צברתי ידע מעמיק במגוון של נושאים כמו למידה מחיזוקים, רשתות נוירונים ומתודולוגיית MVC. למדתי על הקשיים הכרוכים בפני מפתחים במהלך בניית ממשק גרפי נוח למשתמש וחיבורו ללוגיקה העצמאית. הלמידה כללה למידה מעמיקה של מגוון ספריות שונות כגון tkinter, tensorflow וכו', בנוסף למבני הנתונים המובנים בשפה המשמשים ללמידת המכונה.

במהלך עבודתי על הפרויקט, למדתי את ספריית tkinter מ-0 עד לרמה של יצירת GUI נגיש ונוח למשתמש למשחק. הקושי המרכזי שנתקלתי בו בחלק זה הוא האינטגרציה של הלוגיקה לתוך הGUI, תוך כדי שימור העצמאות של כל מחלקה. במהלך משחק, הנקודות הפתוחות ללחיצת השחקן משתנות באופן דינמי מאוד, ומה שקובע אם כפתור צריך להיות דלוק או כבוי הוא הלוגיקה, מה שמוביל למעבר מידע רציף לאורך כל המשחק בין הגרפיקה ללוגיקה. כדי להתמודד עם בעיה זו מימשתי הרבה פעולות תיווך במחלקת הcontroller שעזרו לי להעביר את המידע מהלוגיקה לגרפיקה בצורה נוחה, תוך שימור על עצמאות המחלקות בקוד.

נוסף על כך, התעמקתי בפרויקט על למידה מחיזוקים ועל רשתות נוירונים. בפרויקט מימשתי 2 סוגי סוכנים המובנים על 2 סוגים שונים של למידת מכונה. הסוכן ה"חכם" מובנה על למידה על ידי חיזוקים, כלומר הסוכן מקבל תגמול על צעדים טובים שמובילים לניצחון. סוכן זה מסתמך על משחקים קודמים, ולא יכול לתת ניקוד ללוח שלא ראה לפני כן. הסוכן ה"גאון" מובנה על רשתות נוירונים. הרשתות למדו את המילונים ו"הסיקו" מסקנות בהתאם.
הרשתות יכולות לנבא את הניקוד של כל לוח, גם אם לא נתקל בו בשלב האימון שלו. הקושי המרכזי שלי בשלב זה היה למצוא דרך לשנות את נתוני האימון כך שהרשת תצליח ללמוד את לוגיקת המשחק. ברוב שורות הנתונים במילון, הניקוד מאוד קרוב ל-0, מה שמלמד את הרשת להחזיר 0 על כל ערך כדי להקטין את הloss כמה שאפשר. כדי לתקן את בעיה זו חשבתי על פונקציה כלשהי שתמיד עולה ורציפה, אך עולה בקצב איטי יותר מליניארית, ולבסוף הגעתי לlog.

הקוד ליצירת המילונים:

```
import os
import random
import json
import copy

class Sector:
    def __init__(self, mills, legal_moves):
        self.taken_by = 0
        self.all_mills = mills  # list of tuples that
represent possible mills
        self.legal_moves = legal_moves

class Game:
    def __init__(self):
        self.board = {
            'a': Sector([('a', 'b', 'c'), ('a', 'g',
'n')], ('b', 'g')),
            'b': Sector([('a', 'b', 'c')], ('a', 'e',
'c')),
            'c': Sector([('a', 'b', 'c'), ('c', 'j',
'p')], ('b', 'j')),
            'd': Sector([('d', 'e', 'f'), ('d', 'h',
'k')], ('e', 'h')),
            'e': Sector([('d', 'e', 'f')], ('b', 'd',
'f')),
            'f': Sector([('d', 'e', 'f'), ('f', 'i',
'm')], ('e', 'i')),
            'g': Sector([('a', 'g', 'n')], ('a', 'h',
'n')),
            'h': Sector([('d', 'h', 'k')], ('d', 'g',
'k')),
            'i': Sector([('f', 'i', 'm')], ('f', 'j',
'm')),
            'j': Sector([('c', 'j', 'p')], ('c', 'i',
'p')),
            'k': Sector([('k', 'l', 'm'), ('d', 'h',
'k')], ('h', 'l')),
            'l': Sector([('k', 'l', 'm')], ('k', 'o',
'm')),
            'm': Sector([('k', 'l', 'm'), ('f', 'i',
'm')], ('l', 'i')),
            'n': Sector([('n', 'o', 'p'), ('a', 'g',
'n')], ('g', 'o')),
            'o': Sector([('n', 'o', 'p')], ('n', 'l',
'p')),
            'p': Sector([('n', 'o', 'p'), ('c', 'j',
'p')], ('o', 'j')),
```

```
        }
        self.board_list = []
        self.board_list_place = []
        self.max_pieces = 5
        self.player_sectors = 0
        self.agent_sectors = 0
        self.ACTIVE_MILLS = []
        self.agent_taken_sectors = []
        self.player_taken_sectors = []
        self.empty_sectors = list('abcdefghijklmnop')

    def set_board(self):
        self.board = {
            'a': Sector([('a', 'b', 'c'), ('a', 'g',
'n')], ('b', 'g')),
            'b': Sector([('a', 'b', 'c')], ('a', 'e',
'c')),
            'c': Sector([('a', 'b', 'c'), ('c', 'j',
'p')], ('b', 'j')),
            'd': Sector([('d', 'e', 'f'), ('d', 'h',
'k')], ('e', 'h')),
            'e': Sector([('d', 'e', 'f')], ('b', 'd',
'f')),
            'f': Sector([('d', 'e', 'f'), ('f', 'i',
'm')], ('e', 'i')),
            'g': Sector([('a', 'g', 'n')], ('a', 'h',
'n')),
            'h': Sector([('d', 'h', 'k')], ('d', 'g',
'k')),
            'i': Sector([('f', 'i', 'm')], ('f', 'j',
'm')),
            'j': Sector([('c', 'j', 'p')], ('c', 'i',
'p')),
            'k': Sector([('k', 'l', 'm'), ('d', 'h',
'k')], ('h', 'l')),
            'l': Sector([('k', 'l', 'm')], ('k', 'o',
'm')),
            'm': Sector([('k', 'l', 'm'), ('f', 'i',
'm')], ('l', 'i')),
            'n': Sector([('n', 'o', 'p'), ('a', 'g',
'n')], ('g', 'o')),
            'o': Sector([('n', 'o', 'p')], ('n', 'l',
'p')),
            'p': Sector([('n', 'o', 'p'), ('c', 'j',
'p')], ('o', 'j')),
        }
        self.board_list = []
        self.board_list_place = []
        self.max_pieces = 5
        self.player_sectors = 0
        self.agent_sectors = 0
```

```
        self.ACTIVE_MILLS = []
        self.agent_taken_sectors = []
        self.player_taken_sectors = []
        self.empty_sectors = list('abcdefghijklmnop')

    # ---------- Original random methods ----------
    def put_soldier_agent(self):
        letters = "abcdefghijklmnop"
        sector = random.choice(letters)
        while self.board[sector].taken_by != 0:
            sector = random.choice(letters)
        self.board[sector].taken_by = 1
        self.agent_taken_sectors.append(sector)
        self.agent_sectors += 1
        self.empty_sectors.remove(sector)
        return sector

    def put_soldier_player(self):
        letters = "abcdefghijklmnop"
        sector = random.choice(letters)
        while self.board[sector].taken_by != 0:
            sector = random.choice(letters)
        self.board[sector].taken_by = -1
        self.player_taken_sectors.append(sector)
        self.player_sectors += 1
        self.empty_sectors.remove(sector)
        return sector

    def check_legal_moves(self, sec):
        legal_move = False
        for move in self.board[sec].legal_moves:
            if move in self.empty_sectors:
                legal_move = True
        return legal_move

    def check_jam_win_agent(self):
        jammed = True
        for sec in self.agent_taken_sectors:
            if self.check_legal_moves(sec):
                jammed = False
        return jammed

    def check_jam_win_player(self):
        jammed = True
        for sec in self.player_taken_sectors:
            if self.check_legal_moves(sec):
                jammed = False
        return jammed

    def take_sector_agent(self):
        sector = random.choice(self.agent_taken_sectors)
```

```
        while not self.check_legal_moves(sector):
            sector =
random.choice(self.agent_taken_sectors)
        sector_move =
random.choice(self.board[sector].legal_moves)
        while self.board[sector_move].taken_by != 0:
            sector_move =
random.choice(self.board[sector].legal_moves)
        self.board[sector].taken_by = 0
        self.board[sector_move].taken_by = 1
        self.agent_taken_sectors.remove(sector)
        self.agent_taken_sectors.append(sector_move)
        self.empty_sectors.append(sector)
        self.empty_sectors.remove(sector_move)
        for mill in self.board[sector].all_mills:
            if mill in self.ACTIVE_MILLS:
                self.ACTIVE_MILLS.remove(mill)
        return sector_move

    def take_sector_player(self):
        sector = random.choice(self.player_taken_sectors)
        while not self.check_legal_moves(sector):
            sector =
random.choice(self.player_taken_sectors)
        sector_move =
random.choice(self.board[sector].legal_moves)
        while self.board[sector_move].taken_by != 0:
            sector_move =
random.choice(self.board[sector].legal_moves)
        self.board[sector].taken_by = 0
        self.board[sector_move].taken_by = -1
        self.player_taken_sectors.remove(sector)
        self.player_taken_sectors.append(sector_move)
        self.empty_sectors.append(sector)
        self.empty_sectors.remove(sector_move)
        for mill in self.board[sector].all_mills:
            if mill in self.ACTIVE_MILLS:
                self.ACTIVE_MILLS.remove(mill)
        return sector_move
    def check_mill_with_board(self, s, board):
        sector = board[s]
        for mill in sector.all_mills:
            if (board[mill[0]].taken_by == 1 and
                    board[mill[1]].taken_by == 1 and
                    board[mill[2]].taken_by == 1):

                return 1
            elif (board[mill[0]].taken_by == -1 and
                  board[mill[1]].taken_by == -1 and
                  board[mill[2]].taken_by == -1):
                return -1
```

```
        return 0

    def check_mill(self, s):
        sector = self.board[s]
        for mill in sector.all_mills:
            if (self.board[mill[0]].taken_by == 1 and
                self.board[mill[1]].taken_by == 1 and
                self.board[mill[2]].taken_by == 1 and
                mill not in self.ACTIVE_MILLS):
                self.ACTIVE_MILLS.append(mill)
                return 1
            elif (self.board[mill[0]].taken_by == -1 and
                  self.board[mill[1]].taken_by == -1 and
                  self.board[mill[2]].taken_by == -1 and
                  mill not in self.ACTIVE_MILLS):
                return -1
        return 0

    def double_mill_checker(self, s, owner):
        sector = self.board[s]
        if len(sector.all_mills) != 2:
            return False
        if (sector.all_mills[0] in self.ACTIVE_MILLS and
            sector.taken_by == owner and
            sector.all_mills[1] in self.ACTIVE_MILLS):
            return True
        return False

    def sector_in_active_mills(self, s, owner):
        for mill in self.ACTIVE_MILLS:
            if s in mill and self.board[s].taken_by ==
owner:
                return True
        return False

    def remove_soldier(self, winner):
        # Random removal fallback
        if winner == 1:
            sector =
random.choice(self.player_taken_sectors)
            if (not self.sector_in_active_mills(sector, -
1) or
                self.player_sectors == 3 or
                self.double_mill_checker(sector, -1)):
                self.board[sector].taken_by = 0
                self.player_taken_sectors.remove(sector)
                self.player_sectors -= 1
                self.empty_sectors.append(sector)
        elif winner == -1:
            sector =
random.choice(self.agent_taken_sectors)
```

```
            if (not self.sector_in_active_mills(sector,
1) or
                self.player_sectors == 3 or
                self.double_mill_checker(sector, 1)):
                self.board[sector].taken_by = 0
                self.agent_taken_sectors.remove(sector)
                self.agent_sectors -= 1
                self.empty_sectors.append(sector)

    def fly_soldier_agent(self):
        sector = random.choice(self.agent_taken_sectors)
        sector_move = random.choice(self.empty_sectors)
        self.board[sector].taken_by = 0
        self.board[sector_move].taken_by = 1
        self.agent_taken_sectors.remove(sector)
        self.agent_taken_sectors.append(sector_move)
        self.empty_sectors.append(sector)
        self.empty_sectors.remove(sector_move)
        for mill in self.board[sector].all_mills:
            if mill in self.ACTIVE_MILLS:
                self.ACTIVE_MILLS.remove(mill)
        return sector_move

    def fly_soldier_player(self):
        sector = random.choice(self.player_taken_sectors)
        sector_move = random.choice(self.empty_sectors)
        self.board[sector].taken_by = 0
        self.board[sector_move].taken_by = -1
        self.player_taken_sectors.remove(sector)
        self.player_taken_sectors.append(sector_move)
        self.empty_sectors.append(sector)
        self.empty_sectors.remove(sector_move)
        for mill in self.board[sector].all_mills:
            if mill in self.ACTIVE_MILLS:
                self.ACTIVE_MILLS.remove(mill)
        return sector_move

    def check_win_regular(self):
        if self.player_sectors == 2:
            return 1
        elif self.agent_sectors == 2:
            return -1
        else:
            return 0

    def _choose_dict_from_board(self, board):
        count_agent = 0
        count_player = 0
        for sec in board.values():
            if sec.taken_by == 1:
                count_agent += 1
```

```
            elif sec.taken_by == -1:
                count_player += 1
        if count_agent > 3 and count_player > 3:
            return 1
        if count_agent <= 3 and count_player > 3:
            return 2
        if count_agent > 3 and count_player <= 3:
            return 3
        if count_agent <= 3 and count_player <= 3:
            return 4
        return 4


    # ------ Helper: check if a move blocks an opponent
mill ------
    def move_blocks_opponent(self, board, move_sector,
opponent=-1):
        """
        Given a board state (dictionary) and a move
candidate (sector),
        return True if placing a soldier there would
block an opponent mill.
        We check each mill involving move_sector and if
two cells are occupied
        by the opponent and the third is empty, that move
blocks the mill.
        """
        sec_obj = board[move_sector]
        for mill in sec_obj.all_mills:
            # Count opponent pieces in this mill.
            opponent_count = sum(1 for pos in mill if
board[pos].taken_by == opponent)
            empty_count = sum(1 for pos in mill if
board[pos].taken_by == 0)
            if opponent_count == 2 and empty_count == 1:
                return True
        return False

    # ------ Smart methods using dictionaries and
blocking bonus ------
    def smart_place(self, value_dicts):
        # Use "place" dictionary; add bonus if move
blocks opponent mill.
        best_sector = None
        best_score = -float('inf')
        BLOCK_BONUS = 1000
        for sector in self.empty_sectors:
            board_copy = copy.deepcopy(self.board)
            board_copy[sector].taken_by = 1
            if self.check_mill_with_board(sector,
board_copy) == 1:
```

```
                best_sector = sector
                break
            board_str =
convert_taken_by_to_string(board_copy)
            # Look up dictionary value for placing moves.
            place_dict = value_dicts.get("place", {})
            score = place_dict.get(board_str, [0, 1])[0]
            # Check if placing here blocks an opponent
mill.
            if self.move_blocks_opponent(board_copy,
sector, opponent=-1):
                score += BLOCK_BONUS
            if score > best_score:
                best_score = score
                best_sector = sector
        if best_sector is None:
            return self.put_soldier_agent()
        self.board[best_sector].taken_by = 1
        self.agent_taken_sectors.append(best_sector)
        self.agent_sectors += 1
        self.empty_sectors.remove(best_sector)
        return best_sector

    def smart_player_place(self):
        best_sector = None
        for sector in self.empty_sectors:
            board_copy = copy.deepcopy(self.board)
            board_copy[sector].taken_by = 1
            if self.check_mill_with_board(sector,
board_copy) == 1:
                best_sector = sector
                break
        if best_sector is None:
            return self.put_soldier_player()
        self.board[best_sector].taken_by = -1
        self.player_taken_sectors.append(best_sector)
        self.player_sectors += 1
        self.empty_sectors.remove(best_sector)
        return best_sector


    def smart_move(self, value_dicts):
        mill_found = False
        best_origin = None
        best_destination = None
        best_score = -float('inf')
        BLOCK_BONUS = 1000
        for origin in self.agent_taken_sectors:
            if mill_found:
                break
            if not self.check_legal_moves(origin):
```

```
                continue
            for destination in
self.board[origin].legal_moves:
                if self.board[destination].taken_by != 0:
                    continue
                board_copy = copy.deepcopy(self.board)
                board_copy[origin].taken_by = 0
                board_copy[destination].taken_by = 1
                if
self.check_mill_with_board(destination, board_copy) == 1:
                    best_origin = origin
                    best_destination = destination
                    mill_found = True
                    break
                board_str =
convert_taken_by_to_string(board_copy)
                dict_num =
self._choose_dict_from_board(board_copy)
                if dict_num == 1:
                    dict_used = value_dicts.get("no_fly",
{})
                elif dict_num == 2:
                    dict_used =
value_dicts.get("agent_fly", {})
                elif dict_num == 3:
                    dict_used =
value_dicts.get("player_fly", {})
                else:
                    dict_used =
value_dicts.get("all_fly", {})
                score = dict_used.get(board_str,
[0,1])[0]
                # Add bonus if the move blocks an
opponent mill.
                if self.move_blocks_opponent(board_copy,
destination, opponent=-1):
                    score += BLOCK_BONUS
                if score > best_score:
                    best_score = score
                    best_origin = origin
                    best_destination = destination
        if best_origin is None or best_destination is
None:
            return self.take_sector_agent()
        self.board[best_origin].taken_by = 0
        self.board[best_destination].taken_by = 1
        self.agent_taken_sectors.remove(best_origin)
        self.agent_taken_sectors.append(best_destination)
        self.empty_sectors.append(best_origin)
        self.empty_sectors.remove(best_destination)
        for mill in self.board[best_origin].all_mills:
```

```
            if mill in self.ACTIVE_MILLS:
                self.ACTIVE_MILLS.remove(mill)
        return best_destination

    def smart_player_move(self):
        mill_found = False
        best_origin = None
        best_destination = None
        for origin in self.player_taken_sectors:
            if mill_found:
                break
            if not self.check_legal_moves(origin):
                continue
            for destination in
self.board[origin].legal_moves:
                if self.board[destination].taken_by != 0:
                    continue
                board_copy = copy.deepcopy(self.board)
                board_copy[origin].taken_by = 0
                board_copy[destination].taken_by = -1
                if
self.check_mill_with_board(destination, board_copy) == -
1:
                    best_origin = origin
                    best_destination = destination
                    mill_found = True
                    break
        if best_origin is None or best_destination is
None:
            return self.take_sector_player()
        self.board[best_origin].taken_by = 0
        self.board[best_destination].taken_by = -1
        self.player_taken_sectors.remove(best_origin)

self.player_taken_sectors.append(best_destination)
        self.empty_sectors.append(best_origin)
        self.empty_sectors.remove(best_destination)
        for mill in self.board[best_origin].all_mills:
            if mill in self.ACTIVE_MILLS:
                self.ACTIVE_MILLS.remove(mill)
        return best_destination

    def smart_fly(self, value_dicts):
        mill_found = False
        best_origin = None
        best_destination = None
        best_score = -float('inf')
        BLOCK_BONUS = 1000
        for origin in self.agent_taken_sectors:
            if mill_found:
                break
```

```
            for destination in self.empty_sectors:
                board_copy = copy.deepcopy(self.board)
                board_copy[origin].taken_by = 0
                board_copy[destination].taken_by = 1
                if
self.check_mill_with_board(destination, board_copy) == 1:
                    best_origin = origin
                    best_destination = destination
                    mill_found = True
                    break
                board_str =
convert_taken_by_to_string(board_copy)
                dict_num =
self._choose_dict_from_board(board_copy)
                if dict_num == 1:
                    dict_used = value_dicts.get("no_fly",
{})
                elif dict_num == 2:
                    dict_used =
value_dicts.get("agent_fly", {})
                elif dict_num == 3:
                    dict_used =
value_dicts.get("player_fly", {})
                else:
                    dict_used =
value_dicts.get("all_fly", {})
                score = dict_used.get(board_str,
[0,1])[0]
                # Add bonus if flying to destination
blocks an opponent mill.
                if self.move_blocks_opponent(board_copy,
destination, opponent=-1):
                    score += BLOCK_BONUS
                if score > best_score:
                    best_score = score
                    best_origin = origin
                    best_destination = destination
        if best_origin is None or best_destination is
None:
            return self.fly_soldier_agent()
        self.board[best_origin].taken_by = 0
        self.board[best_destination].taken_by = 1
        self.agent_taken_sectors.remove(best_origin)
        self.agent_taken_sectors.append(best_destination)
        self.empty_sectors.append(best_origin)
        self.empty_sectors.remove(best_destination)
        for mill in self.board[best_origin].all_mills:
            if mill in self.ACTIVE_MILLS:
                self.ACTIVE_MILLS.remove(mill)
        return best_destination
```

```
    def smart_player_fly(self):
        mill_found = False
        best_origin = None
        best_destination = None
        for origin in self.player_taken_sectors:
            if mill_found:
                break
            for destination in self.empty_sectors:
                board_copy = copy.deepcopy(self.board)
                board_copy[origin].taken_by = 0
                board_copy[destination].taken_by = -1
                if
self.check_mill_with_board(destination, board_copy) == -
1:
                    best_origin = origin
                    best_destination = destination
                    mill_found = True
                    break
        if best_origin is None or best_destination is
None:
            return self.fly_soldier_player()
        self.board[best_origin].taken_by = 0
        self.board[best_destination].taken_by = -1
        self.player_taken_sectors.remove(best_origin)

self.player_taken_sectors.append(best_destination)
        self.empty_sectors.append(best_origin)
        self.empty_sectors.remove(best_destination)
        for mill in self.board[best_origin].all_mills:
            if mill in self.ACTIVE_MILLS:
                self.ACTIVE_MILLS.remove(mill)
        return best_destination

    def smart_remove(self, winner, value_dicts):
        # Greedy removal: if the agent (winner == 1)
formed a mill,
        # choose the opponent piece whose removal yields
highest dictionary score,
        # with a penalty if that piece is in an active
mill.
        BLOCK_BONUS = 1000  # not used here but you could
incorporate further incentives.
        if winner == 1:
            best_sector = None
            best_score = -float('inf')
            for sector in self.player_taken_sectors:
                if self.sector_in_active_mills(sector, -
1) and not self.double_mill_checker(sector, -1):
                    continue
                board_copy = copy.deepcopy(self.board)
                board_copy[sector].taken_by = 0
```

```
                board_str =
convert_taken_by_to_string(board_copy)
                dict_num =
self._choose_dict_from_board(board_copy)
                if dict_num == 1:
                    dict_used = value_dicts.get("no_fly",
{})
                elif dict_num == 2:
                    dict_used =
value_dicts.get("agent_fly", {})
                elif dict_num == 3:
                    dict_used =
value_dicts.get("player_fly", {})
                else:
                    dict_used =
value_dicts.get("all_fly", {})
                score = dict_used.get(board_str,
[0,1])[0]

                if score > best_score:
                    best_score = score
                    best_sector = sector
            if best_sector is None:
                best_sector =
random.choice(self.player_taken_sectors)
            self.board[best_sector].taken_by = 0
            self.player_taken_sectors.remove(best_sector)
            self.player_sectors -= 1
            self.empty_sectors.append(best_sector)
            return best_sector

    # ------ Smart play one game (smart agent vs. random
player) ------
    # value_dicts is a container mapping keys ("place",
"no_fly", "agent_fly", "player_fly", "all_fly")
    def smart_play_one_game(self, value_dicts):
        self.set_board()
        # Placing Stage
        for i in range(5):
            # Smart agent places soldier using dictionary
"place"
            sector = self.smart_place(value_dicts)

self.board_list_place.append(copy.deepcopy(self.board))
            if self.check_mill(sector) == 1:
                self.smart_remove(1, value_dicts)

self.board_list_place.append(copy.deepcopy(self.board))
                if self.check_win_regular() != 0:
                    return self.check_win_regular()
            # Random player places soldier
```

```
            sector = self.put_soldier_player()

self.board_list_place.append(copy.deepcopy(self.board))
            if self.check_mill(sector) == -1:
                self.remove_soldier(-1)

self.board_list_place.append(copy.deepcopy(self.board))
                if self.check_win_regular() != 0:
                    return self.check_win_regular()
        # Moving Stage
        while self.agent_sectors > 2 and
self.player_sectors > 2:
            if self.agent_sectors != 3:
                if self.check_jam_win_agent():
                    return -1
                sector = self.smart_move(value_dicts)

self.board_list.append(copy.deepcopy(self.board))
            else:
                sector = self.smart_fly(value_dicts)

self.board_list.append(copy.deepcopy(self.board))
            if self.check_mill(sector) == 1:
                self.smart_remove(1, value_dicts)

self.board_list.append(copy.deepcopy(self.board))
                if self.check_win_regular() != 0:
                    return self.check_win_regular()
            if self.player_sectors != 3:
                if self.check_jam_win_player():
                    return 1
                sector = self.take_sector_player()

self.board_list.append(copy.deepcopy(self.board))
            else:
                sector = self.fly_soldier_player()

self.board_list.append(copy.deepcopy(self.board))
            if self.check_mill(sector) == -1:
                self.remove_soldier(-1)

self.board_list.append(copy.deepcopy(self.board))
                if self.check_win_regular() != 0:
                    return self.check_win_regular()
        if self.player_sectors == 2:
            return 1
        elif self.agent_sectors == 2:
            return -1
        else:
            return 0
```

```
    def play_one_game(self):
        self.set_board()
        for i in range(5):
            sector = self.put_soldier_agent()

self.board_list_place.append(copy.deepcopy(self.board))
            if self.check_mill(sector):
                self.remove_soldier(1)

self.board_list_place.append(copy.deepcopy(self.board))
                if self.check_win_regular() != 0:
                    return self.check_win_regular()
            sector = self.put_soldier_player()

self.board_list_place.append(copy.deepcopy(self.board))
            if self.check_mill(sector):
                self.remove_soldier(-1)

self.board_list_place.append(copy.deepcopy(self.board))
                if self.check_win_regular() != 0:
                    return self.check_win_regular()
        while self.agent_sectors > 2 and
self.player_sectors > 2:
            if self.agent_sectors != 3:
                if self.check_jam_win_agent():
                    return -1
                sector = self.take_sector_agent()

self.board_list.append(copy.deepcopy(self.board))
            else:
                sector = self.fly_soldier_agent()

self.board_list.append(copy.deepcopy(self.board))
            if self.check_mill(sector):
                self.remove_soldier(1)

self.board_list.append(copy.deepcopy(self.board))
                if self.check_win_regular() != 0:
                    return self.check_win_regular()
            if self.player_sectors != 3:
                if self.check_jam_win_player():
                    return 1
                sector = self.take_sector_player()

self.board_list.append(copy.deepcopy(self.board))
            else:
                sector = self.fly_soldier_player()

self.board_list.append(copy.deepcopy(self.board))
            if self.check_mill(sector):
                self.remove_soldier(-1)
```

```
self.board_list.append(copy.deepcopy(self.board))
                if self.check_win_regular() != 0:
                    return self.check_win_regular()
        if self.player_sectors == 2:
            return 1
        elif self.agent_sectors == 2:
            return -1
        else:
            return 0

    def smart_play_one_game_heuristic(self, value_dicts):
        self.set_board()
        # Placing Stage
        for i in range(5):
            # Smart agent places soldier using dictionary
"place"
            sector = self.smart_place(value_dicts)

self.board_list_place.append(copy.deepcopy(self.board))
            if self.check_mill(sector) == 1:
                self.smart_remove(1, value_dicts)

self.board_list_place.append(copy.deepcopy(self.board))
                if self.check_win_regular() != 0:
                    return self.check_win_regular()
            # Random player places soldier
            sector = self.smart_player_place()

self.board_list_place.append(copy.deepcopy(self.board))
            if self.check_mill(sector) == -1:
                self.remove_soldier(-1)

self.board_list_place.append(copy.deepcopy(self.board))
                if self.check_win_regular() != 0:
                    return self.check_win_regular()
        # Moving Stage
        while self.agent_sectors > 2 and
self.player_sectors > 2:
            if self.agent_sectors != 3:
                if self.check_jam_win_agent():
                    return -1
                sector = self.smart_move(value_dicts)

self.board_list.append(copy.deepcopy(self.board))
            else:
                sector = self.smart_fly(value_dicts)

self.board_list.append(copy.deepcopy(self.board))
            if self.check_mill(sector) == 1:
                self.smart_remove(1, value_dicts)
```

```
self.board_list.append(copy.deepcopy(self.board))
                if self.check_win_regular() != 0:
                    return self.check_win_regular()
            if self.player_sectors != 3:
                if self.check_jam_win_player():
                    return 1
                sector = self.smart_player_move()

self.board_list.append(copy.deepcopy(self.board))
            else:
                sector = self.smart_player_fly()

self.board_list.append(copy.deepcopy(self.board))
            if self.check_mill(sector) == -1:
                self.remove_soldier(-1)

self.board_list.append(copy.deepcopy(self.board))
                if self.check_win_regular() != 0:
                    return self.check_win_regular()
        if self.player_sectors == 2:
            return 1
        elif self.agent_sectors == 2:
            return -1
        else:
            return 0
def convert_taken_by_to_string(sector_dict):
    result = []
    for sector in sector_dict.values():
        if sector.taken_by == 1:
            result.append("O")
        elif sector.taken_by == 0:
            result.append("-")
        elif sector.taken_by == -1:
            result.append("X")
    return "".join(result)

class Games:
    def __init__(self):
        self.gamma = 0.9
        self.place_dict = {}
        self.no_fly_dict = {}
        self.agent_fly_dict = {}
        self.player_fly_dict = {}
        self.all_fly_dict = {}

    def load_dicts(self):
        # Use try/except in case files don't exist or are
empty/corrupt.
        try:
            with open('place_dict_6.json', 'r') as
```

```
json_file:
                self.place_dict = json.load(json_file)
        except (FileNotFoundError, json.JSONDecodeError):
            self.place_dict = {}
        try:
            with open('no_fly_dict_6.json', 'r') as
json_file:
                self.no_fly_dict = json.load(json_file)
        except (FileNotFoundError, json.JSONDecodeError):
            self.no_fly_dict = {}
        try:
            with open('agent_fly_dict_6.json', 'r') as
json_file:
                self.agent_fly_dict =
json.load(json_file)
        except (FileNotFoundError, json.JSONDecodeError):
            self.agent_fly_dict = {}
        try:
            with open('player_fly_dict_6.json', 'r') as
json_file:
                self.player_fly_dict =
json.load(json_file)
        except (FileNotFoundError, json.JSONDecodeError):
            self.player_fly_dict = {}
        try:
            with open('all_fly_dict_6.json', 'r') as
json_file:
                self.all_fly_dict = json.load(json_file)
        except (FileNotFoundError, json.JSONDecodeError):
            self.all_fly_dict = {}

    def choose_dict(self, board):
        count_agent = 0
        count_player = 0
        for sec in board.values():
            if sec.taken_by == 1:
                count_agent += 1
            if sec.taken_by == -1:
                count_player += 1
        if count_agent > 3 and count_player > 3:
            return 1
        if count_agent <= 3 and count_player > 3:
            return 2
        if count_agent > 3 and count_player <= 3:
            return 3
        if count_agent <= 3 and count_player <= 3:
            return 4

    def add_to_dict(self, grade, board_str, dict_obj):
        if board_str not in dict_obj:
            dict_obj[board_str] = [grade, 1]
```

```
        else:
            # Calculate new average grade
            total_grade = dict_obj[board_str][0] *
dict_obj[board_str][1] + grade
            dict_obj[board_str][1] += 1
            dict_obj[board_str][0] = total_grade /
dict_obj[board_str][1]

    def simulate(self):
        # Original simulation method (if needed)
        for iteration in range(1000000):
            print(iteration)
            current_game = Game()
            winner = current_game.play_one_game()
            grade = 100 if winner == 1 else 50 if winner
== 0 else 0



            current_game.board_list.reverse()
            current_game.board_list_place.reverse()
            save_i = 0
            for i in range(len(current_game.board_list)):
                dict_num =
self.choose_dict(current_game.board_list[i])
                board_str =
convert_taken_by_to_string(current_game.board_list[i])
                grade_dict = grade * (self.gamma ** i)
                save_i = i
                if dict_num == 1:
                    self.add_to_dict(grade_dict,
board_str, self.no_fly_dict)
                elif dict_num == 2:
                    self.add_to_dict(grade_dict,
board_str, self.agent_fly_dict)
                elif dict_num == 3:
                    self.add_to_dict(grade_dict,
board_str, self.player_fly_dict)
                else:
                    self.add_to_dict(grade_dict,
board_str, self.all_fly_dict)

            for i in
range(len(current_game.board_list_place)):
                board_str =
convert_taken_by_to_string(current_game.board_list_place[
i])
                grade_dict = grade * self.gamma ** (i +
save_i + 1)
                self.add_to_dict(grade_dict, board_str,
self.place_dict)
```

```
        # After all iterations, write the final
dictionaries to JSON files.
        for file_path, d in [
            ('place_dict_1.json', self.place_dict),
            ('agent_fly_dict_1.json',
self.agent_fly_dict),
            ('player_fly_dict_1.json',
self.player_fly_dict),
            ('all_fly_dict_1.json', self.all_fly_dict),
            ('no_fly_dict_1.json', self.no_fly_dict)
        ]:
            with open(file_path, 'w') as json_file:
                json.dump(d, json_file, indent=4)

    def smart_simulate(self):
        win_count = 0
        draw_count = 0
        loss_count = 0
        self.load_dicts()
        value_dicts = {
            "place": self.place_dict,
            "no_fly": self.no_fly_dict,
            "agent_fly": self.agent_fly_dict,
            "player_fly": self.player_fly_dict,
            "all_fly": self.all_fly_dict
        }
        for i in range(1000):
            print(i)
            current_game = Game()
            winner =
current_game.smart_play_one_game(value_dicts)
            if winner == 1:
                win_count += 1
            elif winner == 0:
                draw_count += 1
            else:
                loss_count += 1
        total_games = win_count + draw_count + loss_count
        win_rate = (win_count / total_games) * 100
        draw_rate = (draw_count / total_games) * 100
        loss_rate = (loss_count / total_games) * 100
        print(f"Win rate: {win_rate:.2f}%")
        print(f"Draw rate: {draw_rate:.2f}%")
        print(f"Loss rate: {loss_rate:.2f}%")



    def smart_dict_simulate(self):
        # Load existing dictionaries from disk into the
attributes
```

```
        self.load_dicts()
        value_dicts = {
            "place": self.place_dict,
            "no_fly": self.no_fly_dict,
            "agent_fly": self.agent_fly_dict,
            "player_fly": self.player_fly_dict,
            "all_fly": self.all_fly_dict
        }
        for iteration in range(1000000):
            if iteration%100 == 0:
                print(iteration)
            current_game = Game()
            winner =
current_game.smart_play_one_game(value_dicts)
            grade = 100 if winner == 1 else 50 if winner
== 0 else 0

            current_game.board_list.reverse()
            save_i = 0
            for i in range(len(current_game.board_list)):
                dict_num =
self.choose_dict(current_game.board_list[i])
                board_str =
convert_taken_by_to_string(current_game.board_list[i])
                grade_dict = grade * self.gamma ** i
                save_i = i
                if dict_num == 1:
                    self.add_to_dict(grade_dict,
board_str, self.no_fly_dict)
                elif dict_num == 2:
                    self.add_to_dict(grade_dict,
board_str, self.agent_fly_dict)
                elif dict_num == 3:
                    self.add_to_dict(grade_dict,
board_str, self.player_fly_dict)
                else:
                    self.add_to_dict(grade_dict,
board_str, self.all_fly_dict)

            current_game.board_list_place.reverse()
            for i in
range(len(current_game.board_list_place)):
                board_str =
convert_taken_by_to_string(current_game.board_list_place[
i])
                grade_dict = grade * self.gamma ** (i +
save_i + 1)
                self.add_to_dict(grade_dict, board_str,
self.place_dict)

        # After all iterations, write the final
```

```
dictionaries to JSON files.
        for file_path, d in [
            ('place_dict_2.json', self.place_dict),
            ('agent_fly_dict_2.json',
self.agent_fly_dict),
            ('player_fly_dict_2.json',
self.player_fly_dict),
            ('all_fly_dict_2.json', self.all_fly_dict),
            ('no_fly_dict_2.json', self.no_fly_dict)
        ]:
            with open(file_path, 'w') as json_file:
                json.dump(d, json_file, indent=4)
    def smart_dict_simulate_explore(self):
        # Load existing dictionaries from disk into the
attributes
        self.load_dicts()
        value_dicts = {
            "place": self.place_dict,
            "no_fly": self.no_fly_dict,
            "agent_fly": self.agent_fly_dict,
            "player_fly": self.player_fly_dict,
            "all_fly": self.all_fly_dict
        }
        for iteration in range(1000000):
            if iteration%100 == 0:
                print(iteration)
            current_game = Game()
            n = random.randint(1,5)
            if n%5 != 0:
                winner =
current_game.smart_play_one_game(value_dicts)
            else:
                winner = current_game.play_one_game()
            grade = 100 if winner == 1 else 50 if winner
== 0 else 0

            current_game.board_list.reverse()
            save_i = 0
            for i in range(len(current_game.board_list)):
                dict_num =
self.choose_dict(current_game.board_list[i])
                board_str =
convert_taken_by_to_string(current_game.board_list[i])
                grade_dict = grade * self.gamma ** i
                save_i = i
                if dict_num == 1:
                    self.add_to_dict(grade_dict,
board_str, self.no_fly_dict)
                elif dict_num == 2:
                    self.add_to_dict(grade_dict,
board_str, self.agent_fly_dict)
```

```
                elif dict_num == 3:
                    self.add_to_dict(grade_dict,
board_str, self.player_fly_dict)
                else:
                    self.add_to_dict(grade_dict,
board_str, self.all_fly_dict)

            current_game.board_list_place.reverse()
            for i in
range(len(current_game.board_list_place)):
                board_str =
convert_taken_by_to_string(current_game.board_list_place[
i])
                grade_dict = grade * self.gamma ** (i +
save_i + 1)
                self.add_to_dict(grade_dict, board_str,
self.place_dict)

        # After all iterations, write the final
dictionaries to JSON files.
        for file_path, d in [
            ('place_dict_5.json', self.place_dict),
            ('agent_fly_dict_5.json',
self.agent_fly_dict),
            ('player_fly_dict_5.json',
self.player_fly_dict),
            ('all_fly_dict_5.json', self.all_fly_dict),
            ('no_fly_dict_5.json', self.no_fly_dict)
        ]:
            with open(file_path, 'w') as json_file:
                json.dump(d, json_file, indent=4)
    def smart_dict_simulate_explore_heuristic(self):
        # Load existing dictionaries from disk into the
attributes
        self.load_dicts()
        value_dicts = {
            "place": self.place_dict,
            "no_fly": self.no_fly_dict,
            "agent_fly": self.agent_fly_dict,
            "player_fly": self.player_fly_dict,
            "all_fly": self.all_fly_dict
        }
        for iteration in range(1000000):
            if iteration%100 == 0:
                print(iteration)
            current_game = Game()
            n = random.randint(1,10)
            if n != 1 and n != 2 and n != 3:
                winner =
current_game.smart_play_one_game_heuristic(value_dicts)
            else:
```

```
                winner = current_game.play_one_game()
            grade = 100 if winner == 1 else 50 if winner
== 0 else 0

            current_game.board_list.reverse()
            save_i = 0
            for i in range(len(current_game.board_list)):
                dict_num =
self.choose_dict(current_game.board_list[i])
                board_str =
convert_taken_by_to_string(current_game.board_list[i])
                grade_dict = grade * self.gamma ** i
                save_i = i
                if dict_num == 1:
                    self.add_to_dict(grade_dict,
board_str, self.no_fly_dict)
                elif dict_num == 2:
                    self.add_to_dict(grade_dict,
board_str, self.agent_fly_dict)
                elif dict_num == 3:
                    self.add_to_dict(grade_dict,
board_str, self.player_fly_dict)
                else:
                    self.add_to_dict(grade_dict,
board_str, self.all_fly_dict)

            current_game.board_list_place.reverse()
            for i in
range(len(current_game.board_list_place)):
                board_str =
convert_taken_by_to_string(current_game.board_list_place[
i])
                grade_dict = grade * self.gamma ** (i +
save_i + 1)
                self.add_to_dict(grade_dict, board_str,
self.place_dict)

        # After all iterations, write the final
dictionaries to JSON files.
        for file_path, d in [
            ('place_dict_7.json', self.place_dict),
            ('agent_fly_dict_7.json',
self.agent_fly_dict),
            ('player_fly_dict_7.json',
self.player_fly_dict),
            ('all_fly_dict_7.json', self.all_fly_dict),
            ('no_fly_dict_7.json', self.no_fly_dict)
        ]:
            with open(file_path, 'w') as json_file:
                json.dump(d, json_file, indent=4)
my_games = Games()
```

```
# my_games.simulate()
# my_games.smart_dict_simulate()
# my_games.smart_dict_simulate_explore()
# my_games.smart_dict_simulate_explore()  x
# my_games.smart_dict_simulate_explore()  x
# my_games.smart_dict_simulate_explore_heuristic()
my_games.smart_dict_simulate_explore_heuristic()
```

הקוד של הגרפיקה:

```
import copy
import random
import tkinter as tk
from tkinter import messagebox
import json
import numpy as np
from tensorflow.keras.models import load_model  # or from
keras.models import load_model

def load_data(filename):
    """
       Load data from a JSON file.

       Args:
           filename (str): Path to the JSON file.

       Returns:
           dict: Parsed JSON data.
       """
    with open(filename, 'r') as file:
        return json.load(file)


class StartScreen:
    """
       Displays the initial window for agent selection.
    """
    def __init__(self, root):
        """
                Initialize the start screen, set up the
window, and add agent selection buttons.

                Args: root: Tkinter root window.
        """
        self.root = root
        self.root.title("5 Men's Morris - Choose Agent")
        self.root.geometry("300x300")


        label = tk.Label(root, text="Select Agent Mode",
font=("Helvetica", 16))
```

```
        label.pack(pady=20)

        random_btn = tk.Button(root, text="Random Agent",
width=20, command=lambda: self.start_game("random"))
        random_btn.pack(pady=10)

        smart_btn = tk.Button(root, text="Smart Agent",
width=20, command=lambda: self.start_game("smart"))
        smart_btn.pack(pady=10)

        genius_btn = tk.Button(root, text="Genius Agent",
width=20, command=lambda: self.start_game("genius"))
        genius_btn.pack(pady=10)



    def start_game(self, mode):
        """
                Handles agent selection, closes the start
screen, and opens the main game window.

                Args:
                    mode (str): Selected agent mode
("random", "smart", "genius").
        """
        messagebox.showinfo("Info", f"{mode.capitalize()}
Agent selected.")
        self.root.destroy()  # Close the start screen
        game_window = tk.Tk()
        Graphics(game_window, agent_mode=mode)
        game_window.mainloop()


class Graphics:
    """
      Handles the main game GUI, including the board,
buttons, and user interactions.
      """
    def __init__(self, root, agent_mode):
        """
                Initialize the game window, board, and
control buttons.

                Args:
                    root: Tkinter window.
                    agent_mode (str): Selected agent
mode.
                """
        self.selected_piece = None
        self.root = root
        self.root.title("5 Men's Morris")
```

```
        self.canvas = tk.Canvas(root, width=400,
height=400, bg="dark slate gray")
        self.canvas.pack()

        self.agent_sign = "O"
        self.player_sign = "X"
        self.agent_color ="green4"
        self.player_color = "maroon"

        # GUI now only accesses logic via the controller
API.
        self.controller = Controller(agent_mode)
        self.draw_board()

        # Define positions for buttons (sectors); mapping
index to (x, y) coordinates.
        self.positions = [
            (50, 50), (200, 50), (350, 50),  # Top row
(outer square)
            (125, 125), (200, 125), (275, 125),  # Inner
square (top row)
            (50, 200), (125, 200), (275, 200), (350,
200),  # Middle row
            (125, 275), (200, 275), (275, 275),  # Inner
square (bottom row)
            (50, 350), (200, 350), (350, 350),  # Bottom
row (outer square)
        ]
        self.buttons = []
        for i, (x, y) in enumerate(self.positions):
            btn = tk.Button(root, text="", width=3,
height=1,
                            command=lambda pos=i:
self.handle_click(pos))
            # Place the button centered on the
coordinates.
            btn.place(x=x - 20, y=y - 20)
            self.buttons.append(btn)

        # Add a reset button and a "Return to Start
Screen" button.
        self.reset_btn = tk.Button(root, text="Reset
Game", command=self.reset_game)
        self.reset_btn.pack(side=tk.BOTTOM, pady=5)
        self.return_btn = tk.Button(root, text="Return to
Start Screen", command=self.return_to_opening)
        self.return_btn.pack(side=tk.BOTTOM, pady=5)
        self.deselect_btn = tk.Button(root, text =
"Deselect Piece", command=self.deselect_piece)
        self.deselect_btn.pack(side=tk.BOTTOM, pady=5)
```

```
        self.put_soldier_agent()
    def deselect_piece(self):
        """Deselect the currently selected piece."""
        if self.selected_piece is not None:
            owner =
self.controller.get_sector_owner(self.selected_piece)
            if owner == -1:

self.buttons[self.selected_piece].config(relief=tk.RAISED
, bg=self.player_color)
            elif owner == 1:

self.buttons[self.selected_piece].config(relief=tk.RAISED
, bg=self.agent_color)
            else:

self.buttons[self.selected_piece].config(relief=tk.RAISED
, bg="SystemButtonFace")
            self.selected_piece = None
    def reset_game(self):
        """ Reset the game to its initial state."""
        self.controller.restart()
        for btn in self.buttons:
            btn.config(text="", relief=tk.RAISED, bg =
"SystemButtonFace")
        self.draw_board()
        self.put_soldier_agent()
        self.selected_piece = None  # Reset selected
piece
        for btn in self.buttons:
            btn.config(state=tk.NORMAL)

    def draw_board(self):
        """Draw a stylish board with colored lines."""
        self.canvas.delete("all")
        self.canvas.create_rectangle(50, 50, 350, 350,
width=10, outline="light cyan")
        self.canvas.create_rectangle(125, 125, 275, 275,
width=10, outline="light cyan")
        self.canvas.create_line(200, 50, 200, 125,
width=10, fill="light cyan")
        self.canvas.create_line(200, 275, 200, 350,
width=10, fill="light cyan")
        self.canvas.create_line(50, 200, 125, 200,
width=10, fill="light cyan")
        self.canvas.create_line(275, 200, 350, 200,
width=10, fill="light cyan")
        self.canvas.create_line(125, 200, 125, 275,
width=10, fill="light cyan")
        self.canvas.create_line(275, 200, 275, 275,
width=10, fill="light cyan")
```

```
    def return_to_opening(self):
        """Return to the opening screen."""
        self.root.destroy()
        start_screen = tk.Tk()
        StartScreen(start_screen)
        start_screen.mainloop()

    def put_soldier_agent(self):
        """ Place a soldier for the agent and update the
button."""
        agent_result =
self.controller.put_soldier_agent()
        sector_num, removed_sector = agent_result
        if agent_result is not None:

self.buttons[sector_num].config(text=self.agent_sign,
bg=self.agent_color)

        if removed_sector is not None:
            rem_index =
"abcdefghijklmnop".index(removed_sector)
            self.buttons[rem_index].config(text="", bg=
"SystemButtonFace")

    def put_soldier_player(self, pos):
        """ Place a soldier for the player and update the
button."""
        if
self.controller.put_soldier_player("abcdefghijklmnop"[pos
]):

self.buttons[pos].config(text=self.player_sign,
bg=self.player_color)
            if self.controller.get_game_phase() == 0:
                self.put_soldier_agent()
            elif self.controller.get_game_phase() == 1:
                self.move_soldier_agent()
                result_jam = self.check_jam_win()
                result_reg = self.check_regular_win()

                if result_reg != 0 or result_jam != 0:
                    return
        else:
            messagebox.showerror("Error", "Invalid move.
Try again.")

    def handle_click(self, pos):
        """Handle button clicks for player moves."""
        if self.controller.get_game_phase() == 0:
            self.put_soldier_player(pos)
```

```
        elif self.controller.get_game_phase() == 1:
            self.move_soldier_player(pos)
            result = self.check_jam_win()
            if result != 0:
                return
        elif self.controller.get_game_phase() == 2:
            self.remove_soldier_player(pos)

    def move_soldier_agent(self):
        """ Move a soldier for the agent and update the
buttons."""
        agent_result =
self.controller.move_soldier_agent()
        if agent_result is not None:
            from_sector, to_sector, from_index, to_index,
removed_sector = agent_result
            self.buttons[from_index].config(text="",
bg="SystemButtonFace")

self.buttons[to_index].config(text=self.agent_sign,
bg=self.agent_color)

            if removed_sector is not None:
                rem_index =
"abcdefghijklmnop".index(removed_sector)
                self.buttons[rem_index].config(text="",
bg="SystemButtonFace")

    def move_soldier_player(self, pos):
        """ Move a soldier for the player and update the
buttons."""
        if self.selected_piece is None:
            if self.controller.get_sector_owner(pos) != -
1:
                messagebox.showerror("Error", "You can
only select your own pieces.")
                return
            self.selected_piece = pos
            self.buttons[pos].config(relief=tk.SUNKEN,
bg="yellow")  # Highlight selected piece
        else:
            player_result =
self.controller.move_soldier_player("abcdefghijklmnop"[se
lf.selected_piece], "abcdefghijklmnop"[pos])
            if player_result:

self.buttons[self.selected_piece].config(text="",relief =
tk.RAISED, bg="SystemButtonFace")

self.buttons[pos].config(text=self.player_sign,relief =
tk.RAISED,  bg=self.player_color)
```

```
                self.selected_piece = None

                if self.controller.get_game_phase() != 2:
                    self.move_soldier_agent()
                    result_jam = self.check_jam_win()
                    result_reg = self.check_regular_win()

                    if result_reg != 0 or result_jam !=
0:
                        return


    def remove_soldier_player(self, pos):
        """ Remove a soldier from the agent and update
the buttons."""
        if
self.controller.remove_soldier_player("abcdefghijklmnop"[
pos]):
            self.buttons[pos].config(text="",
bg="SystemButtonFace")
            result = self.check_regular_win()  # Check
for win condition after removal
            if result != 0:
                return
            if self.controller.get_game_phase() == 0:
                self.put_soldier_agent()
            else:
                self.move_soldier_agent()
                result_jam = self.check_jam_win()
                result_reg = self.check_regular_win()

                if result_reg != 0 or result_jam != 0:
                    return
    def check_regular_win(self):
        """ Check for regular win conditions and display
messages."""
        result = self.controller.check_regular_win()
        if result == 1:
            messagebox.showinfo("Game Over", "Agent wins
by force!")
            self.disable_buttons()

        elif result == -1:
            messagebox.showinfo("Game Over", "Player wins
by force!")
            self.disable_buttons()

        return result

    def check_jam_win(self):
        """ Check for jam win conditions and display
```

```
messages."""
        result = self.controller.check_jam_win_general()
        if result == 1:
            messagebox.showinfo("Game Over", "Agent wins
by jam!")
            self.disable_buttons()
        elif result == -1:
            messagebox.showinfo("Game Over", "Player wins
by jam!")
            self.disable_buttons()
        return result


    def disable_buttons(self):
        """Disable all buttons to prevent further
interaction."""
        for btn in self.buttons:
            btn.config(state=tk.DISABLED)


class Controller:
    """ Controller class to manage the game logic and
interactions between the GUI and the game state."""
    def __init__(self, agent_mode):
        """ Initialize the controller with the game logic
and agent mode."""
        self.agent_mode = agent_mode
        self.logic = Logic(agent_mode)

    def get_game_phase(self):
        """Return the current game phase."""
        return self.logic.game_phase

    def put_soldier_agent(self):
        """Place a soldier for the agent based on the
selected agent mode."""
        if self.agent_mode == "random":
            return self.logic.put_soldier_agent()
        elif self.agent_mode == "smart":
            return self.logic.smart_put_soldier_agent()
        else:
            return self.logic.genius_put_soldier_agent()

    def restart(self):
        """Restart the game by resetting the logic."""
        self.logic.restart()

    def put_soldier_player(self, pos):
        """Place a soldier for the player at the
specified position."""
        return self.logic.put_soldier_player(pos)
```

```
    def move_soldier_agent(self):
        """Move a soldier for the agent based on the
selected agent mode."""
        if self.agent_mode == "random":
            return self.logic.move_soldier_agent()
        elif self.agent_mode == "smart":
            return self.logic.smart_move_soldier_agent()
        else:
            return self.logic.genius_move_soldier_agent()

    def move_soldier_player(self, from_index, to_index):
        """Move a soldier for the player from one index
to another."""
        return self.logic.move_soldier_player(from_index,
to_index)

    def remove_soldier_player(self, pos):
        """Remove a soldier for the player at the
specified position."""
        return self.logic.remove_soldier_player(pos)

    def check_regular_win(self):
        """Check for regular win conditions."""
        return self.logic.check_regular_win()

    def check_jam_win_general(self):
        """Check for jam win conditions."""
        return self.logic.check_jam_win_general()

    def get_sector_owner(self, selected_piece):
        """Return the owner of the sector at the given
index."""
        sector = "abcdefghijklmnop"[selected_piece]
        return self.logic.board[sector].taken_by


class Sector:
    """    Represents a sector on the game board with its
properties."""
    def __init__(self, mills, legal_moves):
        """
                Initialize a sector with its mills and
legal moves.

                Args:
                    mills (list): List of tuples
representing possible mills.
                    legal_moves (tuple): Legal moves for
the sector.
        """
```

```
        self.taken_by = 0  # 0: empty, -1: player, 1:
agent
        self.all_mills = mills
        self.legal_moves = legal_moves


class Logic:
    """    Contains the game logic for managing the
board, player actions, and game phases."""
    def __init__(self, agent_mode):
        """
                Initialize the game logic, including the
game board and agent mode.

                Args:
                    agent_mode (str): Selected agent mode
("random", "smart", "genius").
        """
        self.game_phase = 0
        self.board = {
            'a': Sector([('a', 'b', 'c'), ('a', 'g',
'n')], ('b', 'g')),
            'b': Sector([('a', 'b', 'c')], ('a', 'e',
'c')),
            'c': Sector([('a', 'b', 'c'), ('c', 'j',
'p')], ('b', 'j')),
            'd': Sector([('d', 'e', 'f'), ('d', 'h',
'k')], ('e', 'h')),
            'e': Sector([('d', 'e', 'f')], ('b', 'd',
'f')),
            'f': Sector([('d', 'e', 'f'), ('f', 'i',
'm')], ('e', 'i')),
            'g': Sector([('a', 'g', 'n')], ('a', 'h',
'n')),
            'h': Sector([('d', 'h', 'k')], ('d', 'g',
'k')),
            'i': Sector([('f', 'i', 'm')], ('f', 'j',
'm')),
            'j': Sector([('c', 'j', 'p')], ('c', 'i',
'p')),
            'k': Sector([('k', 'l', 'm'), ('d', 'h',
'k')], ('h', 'l')),
            'l': Sector([('k', 'l', 'm')], ('k', 'o',
'm')),
            'm': Sector([('k', 'l', 'm'), ('f', 'i',
'm')], ('l', 'i')),
            'n': Sector([('n', 'o', 'p'), ('a', 'g',
'n')], ('g', 'o')),
            'o': Sector([('n', 'o', 'p')], ('n', 'l',
'p')),
            'p': Sector([('n', 'o', 'p'), ('c', 'j',
```

```
'p')], ('o', 'j')),
        }
        self.player_sectors = 0
        self.agent_sectors = 0
        self.player_taken_sectors = []
        self.agent_taken_sectors = []
        self.empty_sectors = list("abcdefghijklmnop")
        self.turn = 0
        if agent_mode == "smart":
            self.place_dict =
load_data("place_dict_7.json")
            self.no_fly_dict =
load_data("no_fly_dict_7.json")
            self.player_fly_dict =
load_data("player_fly_dict_7.json")
            self.agent_fly_dict =
load_data("agent_fly_dict_7.json")
            self.all_fly_dict =
load_data("all_fly_dict_7_fixed.json")
        elif agent_mode == "genius":
            self.place_model =
load_model("final_model_place.h5")
            self.no_fly_model =
load_model("final_model_no_fly.h5")
            self.player_fly_model =
load_model("final_model_player_fly.h5")
            self.agent_fly_model =
load_model("final_model_agent_fly.h5")
            self.all_fly_model =
load_model("final_model_all_fly.h5")

    def restart(self):
        """Reset the game state to its initial
configuration."""
        self.game_phase = 0
        self.board = {
            'a': Sector([('a', 'b', 'c'), ('a', 'g',
'n')], ('b', 'g')),
            'b': Sector([('a', 'b', 'c')], ('a', 'e',
'c')),
            'c': Sector([('a', 'b', 'c'), ('c', 'j',
'p')], ('b', 'j')),
            'd': Sector([('d', 'e', 'f'), ('d', 'h',
'k')], ('e', 'h')),
            'e': Sector([('d', 'e', 'f')], ('b', 'd',
'f')),
            'f': Sector([('d', 'e', 'f'), ('f', 'i',
'm')], ('e', 'i')),
            'g': Sector([('a', 'g', 'n')], ('a', 'h',
'n')),
            'h': Sector([('d', 'h', 'k')], ('d', 'g',
```

```
'k')),
            'i': Sector([('f', 'i', 'm')], ('f', 'j',
'm')),
            'j': Sector([('c', 'j', 'p')], ('c', 'i',
'p')),
            'k': Sector([('k', 'l', 'm'), ('d', 'h',
'k')], ('h', 'l')),
            'l': Sector([('k', 'l', 'm')], ('k', 'o',
'm')),
            'm': Sector([('k', 'l', 'm'), ('f', 'i',
'm')], ('l', 'i')),
            'n': Sector([('n', 'o', 'p'), ('a', 'g',
'n')], ('g', 'o')),
            'o': Sector([('n', 'o', 'p')], ('n', 'l',
'p')),
            'p': Sector([('n', 'o', 'p'), ('c', 'j',
'p')], ('o', 'j')),
        }
        self.player_sectors = 0
        self.agent_sectors = 0
        self.player_taken_sectors = []
        self.agent_taken_sectors = []
        self.empty_sectors = list("abcdefghijklmnop")
        self.turn = 0



    def check_mill(self, s):
        """ Check if a mill is formed at the given
sector."""
        player = self.board[s].taken_by
        if player == 0:
            return False
        for mill in self.board[s].all_mills:
            if all(self.board[pos].taken_by == player for
pos in mill):
                print(f"Mill found for player {player} at
{mill}")

                return True
        return False

    def get_removable_opponent_sectors(self, remover):
        """ Get a list of sectors that can be removed by
the opponent."""
        opponent = -remover
        removable = []
        for pos, sector in self.board.items():
            if sector.taken_by == opponent:
                if not self.check_mill(pos):
                    removable.append(pos)
```

```
        if len(removable) == 0:
            for pos, sector in self.board.items():
                if sector.taken_by == opponent:
                    removable.append(pos)
        return removable


    def put_soldier_agent(self):
        """ Place a soldier for the agent and check for
mills."""
        removed_sector = None
        if self.agent_sectors < 5:
            numbers = list(range(16))
            sector_num = random.choice(numbers)
            sector = "abcdefghijklmnop"[sector_num]
            while self.board[sector].taken_by != 0:
                sector_num = random.choice(numbers)
                sector = "abcdefghijklmnop"[sector_num]
            self.board[sector].taken_by = 1
            self.agent_taken_sectors.append(sector)
            self.agent_sectors += 1
            print(f"Agent placed at {sector}, total:
{self.agent_sectors}")
            self.empty_sectors.remove(sector)
            if self.check_mill(sector):
                # Preserve agent mill info before removal
check.
                print(f"Mill formed at {sector} by the
agent!")
                removable =
self.get_removable_opponent_sectors(1)
                if removable:
                    removed_sector =
random.choice(removable)
                    self.board[removed_sector].taken_by =
0
                    self.player_sectors -= 1
                    if removed_sector in
self.player_taken_sectors:

self.player_taken_sectors.remove(removed_sector)
                    if removed_sector not in
self.empty_sectors:

self.empty_sectors.append(removed_sector)
                    print(f"Agent removed player's
soldier at {removed_sector}")

            return sector_num, removed_sector
        return None
```

```
    def put_soldier_player(self, pos):
        """ Place a soldier for the player at the
specified position."""
        if self.board[pos].taken_by == 0:
            self.board[pos].taken_by = -1
            self.player_taken_sectors.append(pos)
            self.player_sectors += 1
            print(f"Player placed at {pos}, total:
{self.player_sectors}")
            self.empty_sectors.remove(pos)
            if self.check_mill(pos):
                print(f"Mill formed at {pos} by the
player!")
                self.game_phase = 2
            self.turn += 1
            if self.turn == 5 and self.game_phase != 2:
                self.game_phase = 1
            return True
        return False

    def move_soldier_agent(self):
        """ Move a soldier for the agent and check for
mills."""

        random.shuffle(self.agent_taken_sectors)
        for from_sector in self.agent_taken_sectors:
            if self.agent_sectors == 3:
                possible_moves = list(self.empty_sectors)
            else:
                possible_moves =
list(self.board[from_sector].legal_moves)
            random.shuffle(possible_moves)
            for to_sector in possible_moves:
                if self.board[to_sector].taken_by == 0:
                    self.board[from_sector].taken_by = 0
                    self.board[to_sector].taken_by = 1

self.agent_taken_sectors.remove(from_sector)

self.agent_taken_sectors.append(to_sector)

self.empty_sectors.append(from_sector)
                    self.empty_sectors.remove(to_sector)
                    from_index =
"abcdefghijklmnop".index(from_sector)
                    to_index =
"abcdefghijklmnop".index(to_sector)
                    removed_sector = None
                    if self.check_mill(to_sector):
                        # Preserve agent mill info before
removal check.
```

```
                        print(f"Mill formed at
{to_sector} by the agent move!")
                        removable =
self.get_removable_opponent_sectors(1)
                        if removable:
                            removed_sector =
random.choice(removable)

self.board[removed_sector].taken_by = 0
                            self.player_sectors -= 1
                            if removed_sector in
self.player_taken_sectors:

self.player_taken_sectors.remove(removed_sector)
                            if removed_sector not in
self.empty_sectors:

self.empty_sectors.append(removed_sector)
                            print(f"Agent removed
player's soldier at {removed_sector}")
                    return from_sector, to_sector,
from_index, to_index, removed_sector
        return None

    def move_soldier_player(self, from_letter,
to_letter):
        """ Move a soldier for the player from one sector
to another."""
        if self.board[from_letter].taken_by == -1 and
self.board[to_letter].taken_by == 0:
            if len(self.player_taken_sectors) == 3 or
to_letter in self.board[from_letter].legal_moves:
                self.board[from_letter].taken_by = 0
                self.board[to_letter].taken_by = -1

self.player_taken_sectors.remove(from_letter)

self.player_taken_sectors.append(to_letter)
                self.empty_sectors.append(from_letter)
                self.empty_sectors.remove(to_letter)
                if self.check_mill(to_letter):
                    print(f"Mill formed at {to_letter} by
the player move!")
                    self.game_phase = 2

                return True
        return False

    def remove_soldier_player(self, pos):
        """ Remove a soldier for the player at the
specified position."""
```

```
        if self.board[pos].taken_by == 1:
            removable =
self.get_removable_opponent_sectors(-1)
            if pos in removable:
                self.board[pos].taken_by = 0
                self.agent_sectors -= 1
                if pos in self.agent_taken_sectors:
                    self.agent_taken_sectors.remove(pos)
                if pos not in self.empty_sectors:
                    self.empty_sectors.append(pos)
                if self.turn == 5:
                    self.game_phase = 1
                else:
                    self.game_phase = 0
                return True
        return False

    def check_regular_win(self):
        """ Check for regular win conditions based on the
number of sectors taken."""
        print(f"player: {self.player_sectors}, agent:
{self.agent_sectors}")
        if self.game_phase != 0:

            if self.player_sectors < 3:
                print("Agent wins by force!")
                return 1
            elif self.agent_sectors < 3:
                print("Player wins by force!")
                return -1
        return 0
    def check_legal_moves(self, sec):
        """ Check if there are legal moves available for
the given sector."""
        legal_move = False
        for move in self.board[sec].legal_moves:
            if move in self.empty_sectors:
                legal_move = True
        return legal_move

    def check_jam_win_agent(self):
        """ Check if the agent is jammed (no legal moves
available)."""
        jammed = True
        for sec in self.agent_taken_sectors:
            if self.check_legal_moves(sec):
                jammed = False
        return jammed

    def check_jam_win_player(self):
        """ Check if the player is jammed (no legal moves
```

```
available)."""
        jammed = True
        for sec in self.player_taken_sectors:
            if self.check_legal_moves(sec):
                jammed = False
        return jammed
    def check_jam_win_general(self):
        """ Check for jam win conditions for both
players."""
        if self.check_jam_win_agent():
            print("Player wins by jam!")
            return -1
        elif self.check_jam_win_player():
            print("Agent wins by jam!")
            return 1
        return 0

    def convert_taken_by_to_string(self, sector_dict):
        """ Convert the taken_by status of sectors to a
string representation."""
        result = []
        for sector in sector_dict.values():
            if sector.taken_by == 1:
                result.append("O")
            elif sector.taken_by == 0:
                result.append("-")
            elif sector.taken_by == -1:
                result.append("X")
        return "".join(result)

    def check_mill_with_board(self, s, board):
        """ Check if a mill is formed at the given sector
using a specific board state."""
        player = board[s].taken_by
        if player == 0:
            return False
        for mill in board[s].all_mills:
            if all(board[pos].taken_by == player for pos
in mill):

                return True

        return False

    def move_blocks_opponent(self, board, move_sector):
        """
        Given a board state (dictionary) and a move
candidate (sector),
        return True if placing a soldier there would
block an opponent mill.
        """
```

```
        sec_obj = board[move_sector]
        for mill in sec_obj.all_mills:
            # Count opponent pieces in this mill.
            opponent_count = sum(1 for pos in mill if
board[pos].taken_by == -1)
            empty_count = sum(1 for pos in mill if
board[pos].taken_by == 0)
            if opponent_count == 2 and empty_count == 1:
                return True
        return False

    def smart_put_soldier_agent(self):
        """ Place a soldier for the agent using dicts."""
        removed_sector = None
        if self.agent_sectors < 5:
            best_sector = None
            best_score = -float('inf')
            block_bonus = 100
            mill_bonus = 200
            for sector in self.empty_sectors:
                score = 0
                board_copy = copy.deepcopy(self.board)
                if self.move_blocks_opponent(board_copy,
sector):

                    score += block_bonus
                board_copy[sector].taken_by = 1
                board_str =
self.convert_taken_by_to_string(board_copy)
                score += self.place_dict.get(board_str,
[0, 0])[0]  # Get score from place_dict
                if self.check_mill_with_board(sector,
board_copy):
                    score += mill_bonus
                if score > best_score:
                    best_score = score
                    best_sector = sector
            if best_sector is None:
                return self.put_soldier_agent()
            self.board[best_sector].taken_by = 1
            self.agent_taken_sectors.append(best_sector)
            self.agent_sectors += 1
            self.empty_sectors.remove(best_sector)
            if self.check_mill(best_sector):
                print(f"Mill formed at {best_sector} by
the agent!")
                removed_sector = self.smart_remove(1)
            return "abcdefghijklmnop".index(best_sector),
removed_sector
        return None
```

```
    def smart_remove(self, winner):
        """ Remove a soldier from the opponent's sectors
using dicts."""
        if winner == 1:
            best_sector = None
            best_score = -float('inf')
            for sector in
self.get_removable_opponent_sectors(1):
                board_copy = copy.deepcopy(self.board)
                board_copy[sector].taken_by = 0
                board_str =
self.convert_taken_by_to_string(board_copy)
                dict_num =
self.choose_dict_from_board(board_copy)
                if dict_num == 0:
                    dict_used = self.place_dict
                if dict_num == 1:
                    dict_used = self.no_fly_dict
                elif dict_num == 2:
                    dict_used = self.agent_fly_dict
                elif dict_num == 3:
                    dict_used = self.player_fly_dict
                else:
                    dict_used = self.all_fly_dict
                score = dict_used.get(board_str, [0,
1])[0]

                if score > best_score:
                    best_score = score
                    best_sector = sector
            if best_sector is None:
                best_sector =
random.choice(self.player_taken_sectors)
            self.board[best_sector].taken_by = 0
            self.player_taken_sectors.remove(best_sector)
            self.player_sectors -= 1
            self.empty_sectors.append(best_sector)
            return best_sector

    def smart_move_soldier_agent(self):
        """ Move a soldier for the agent using dicts."""
        best_origin = None
        best_destination = None
        best_score = -float('inf')
        block_bonus = 100
        mill_bonus = 200
        for from_sector in self.agent_taken_sectors:

            if len(self.agent_taken_sectors) == 3:
                possible_moves = list(self.empty_sectors)
            else:
```

```
                possible_moves = list(filter(lambda x:
self.board[x].taken_by == 0,
self.board[from_sector].legal_moves))
            for to_sector in possible_moves:
                score = 0
                board_copy = copy.deepcopy(self.board)
                if self.move_blocks_opponent(board_copy,
to_sector):
                    score += block_bonus
                board_copy[from_sector].taken_by = 0
                board_copy[to_sector].taken_by = 1
                if self.check_mill_with_board(to_sector,
board_copy):
                    score += mill_bonus

                board_str =
self.convert_taken_by_to_string(board_copy)
                dict_num =
self.choose_dict_from_board(board_copy)
                if dict_num == 1:
                    dict_used = self.no_fly_dict
                elif dict_num == 2:
                    dict_used = self.agent_fly_dict
                elif dict_num == 3:
                    dict_used = self.player_fly_dict
                else:
                    dict_used = self.all_fly_dict

                score += dict_used.get(board_str, [-1,
1])[0]
                if score > best_score:
                    best_score = score
                    best_origin = from_sector
                    best_destination = to_sector
        self.board[best_origin].taken_by = 0
        self.board[best_destination].taken_by = 1
        self.agent_taken_sectors.remove(best_origin)
        self.agent_taken_sectors.append(best_destination)
        self.empty_sectors.append(best_origin)
        self.empty_sectors.remove(best_destination)
        from_index =
"abcdefghijklmnop".index(best_origin)
        to_index =
"abcdefghijklmnop".index(best_destination)
        removed_sector = None
        if self.check_mill(best_destination):


            print(f"Mill formed at {best_destination} by
the agent!")
            removed_sector = self.smart_remove(1)
```

```
        return best_origin, best_destination, from_index,
to_index, removed_sector

    def choose_dict_from_board(self, board):
        """ Choose the appropriate dictionary based on
the board state."""
        if self.game_phase == 0:
            return 0
        count_agent = 0
        count_player = 0
        for sec in board.values():
            if sec.taken_by == 1:
                count_agent += 1
            elif sec.taken_by == -1:
                count_player += 1
        if count_agent > 3 and count_player > 3:
            return 1
        if count_agent <= 3 < count_player:
            return 2
        if count_agent > 3 >= count_player:
            return 3
        if count_agent <= 3 and count_player <= 3:
            return 4
        return 4

    def convert_string(self, s):
        """
        Converts a 16-character string into a numpy
array,
        mapping 'x' to -1, '-' to 0, and 'o' to 1.
        """
        mapping = {'X': -1, '-': 0, 'O': 1}
        return np.array([mapping[char] for char in s])

    def genius_put_soldier_agent(self):
        """ Place a soldier for the agent using neural
networks."""
        removed_sector = None
        if self.agent_sectors < 5:
            best_sector = None
            best_score = -float('inf')
            block_bonus = 100
            mill_bonus = 200
            for sector in self.empty_sectors:
                score = 0
                board_copy = copy.deepcopy(self.board)
                if self.move_blocks_opponent(board_copy,
sector):
                    score += block_bonus
                board_copy[sector].taken_by = 1
                board_str =
```

```
self.convert_taken_by_to_string(board_copy)
                score +=
self.place_model.predict(self.convert_string(board_str).r
eshape(1, -1))
                if self.check_mill_with_board(sector,
board_copy):
                    score += mill_bonus
                if score > best_score:
                    best_score = score
                    best_sector = sector
            if best_sector is None:
                return self.put_soldier_agent()
            self.board[best_sector].taken_by = 1
            self.agent_taken_sectors.append(best_sector)
            self.agent_sectors += 1
            self.empty_sectors.remove(best_sector)
            if self.check_mill(best_sector):
                print(f"Mill formed at {best_sector} by
the agent!")
                removed_sector = self.smart_remove(1)
            return "abcdefghijklmnop".index(best_sector),
removed_sector
        return None

    def genius_remove(self, winner):
        """ Remove a soldier from the opponent's sectors
using neural networks."""
        if winner == 1:
            best_sector = None
            best_score = -float('inf')
            for sector in
self.get_removable_opponent_sectors(1):
                board_copy = copy.deepcopy(self.board)
                board_copy[sector].taken_by = 0
                board_str =
self.convert_taken_by_to_string(board_copy)
                model_num =
self.choose_dict_from_board(board_copy)
                if model_num == 0:
                    model_used = self.place_model

                if model_num == 1:
                    model_used = self.no_fly_model
                elif model_num == 2:
                    model_used = self.agent_fly_model
                elif model_num == 3:
                    model_used = self.player_fly_model
                else:
                    model_used = self.all_fly_model
                score =
model_used.predict(self.convert_string(board_str).reshape
```

```
(1, -1))

                if score > best_score:
                    best_score = score
                    best_sector = sector
            if best_sector is None:
                best_sector =
random.choice(self.player_taken_sectors)
            self.board[best_sector].taken_by = 0
            self.player_taken_sectors.remove(best_sector)
            self.player_sectors -= 1
            self.empty_sectors.append(best_sector)
            return best_sector

    def genius_move_soldier_agent(self):
        """ Move a soldier for the agent using neural
networks."""
        best_origin = None
        best_destination = None
        best_score = -float('inf')
        block_bonus = 100
        mill_bonus = 200
        for from_sector in self.agent_taken_sectors:

            if len(self.agent_taken_sectors) == 3:
                possible_moves = list(self.empty_sectors)
            else:
                possible_moves = list(
                    filter(lambda x:
self.board[x].taken_by == 0,
self.board[from_sector].legal_moves))
            for to_sector in possible_moves:
                score = 0
                board_copy = copy.deepcopy(self.board)
                if self.move_blocks_opponent(board_copy,
to_sector):
                    score += block_bonus
                board_copy[from_sector].taken_by = 0
                board_copy[to_sector].taken_by = 1
                if self.check_mill_with_board(to_sector,
board_copy):
                    score += mill_bonus

                board_str =
self.convert_taken_by_to_string(board_copy)
                model_num =
self.choose_dict_from_board(board_copy)
                if model_num == 1:
                    model_used = self.no_fly_model
                elif model_num == 2:
                    model_used = self.agent_fly_model
```

```
                elif model_num == 3:
                    model_used = self.player_fly_model
                else:
                    model_used = self.all_fly_model

                score +=
model_used.predict(self.convert_string(board_str).reshape
(1, -1))
                if score > best_score:
                    best_score = score
                    best_origin = from_sector
                    best_destination = to_sector
        self.board[best_origin].taken_by = 0
        self.board[best_destination].taken_by = 1
        self.agent_taken_sectors.remove(best_origin)
        self.agent_taken_sectors.append(best_destination)
        self.empty_sectors.append(best_origin)
        self.empty_sectors.remove(best_destination)
        from_index =
"abcdefghijklmnop".index(best_origin)
        to_index =
"abcdefghijklmnop".index(best_destination)
        removed_sector = None
        if self.check_mill(best_destination):
            print(f"Mill formed at {best_destination} by
the agent!")
            removed_sector = self.genius_remove(1)
        return best_origin, best_destination, from_index,
to_index, removed_sector


def main():
    """Main function to start the game."""
    start_screen = tk.Tk()
    StartScreen(start_screen)
    start_screen.mainloop()

if __name__ == "__main__":
    main()
```

הקוד של המודל:

```
import numpy as np
import matplotlib.pyplot as plt
from keras.src.layers import BatchNormalization
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import Callback,
EarlyStopping, ModelCheckpoint
```

```
from sklearn.model_selection import train_test_split



# class LogTargetTransformer:
#     """
#     Applies log(y) for regression targets and provides
inverse exp(y).
#     Assumes all y > 0.
#     """
#     def __init__(self):
#         self.eps = 1e-30  # to prevent log(0) if needed
#
#     def fit(self, y):
#         y = np.asarray(y)
#         invalid = y[y <= 0]
#         if invalid.size > 0:
#             print("Invalid target values detected (must
be > 0):", invalid[:10], "...")
#             raise ValueError("All target values must be
> 0 for log transform.")
#         return self
#
#     def transform(self, y):
#         y = np.asarray(y)
#         # return np.log(y + self.eps)
#         return y
#
#     def inverse_transform(self, y_log):
#         y_log = np.asarray(y_log)
#         # return np.exp(y_log) - self.eps
#         return y_log


# Load dataset
X_agent_fly = np.load("X_agent_fly.npy")
Y_agent_fly = np.load("Y_agent_fly.npy")
# tf = LogTargetTransformer().fit(Y_agent_fly)
# y_agent_fly_log = tf.transform(Y_agent_fly)


print(np.mean(Y_agent_fly))

plt.hist(Y_agent_fly, bins=100)
plt.show()

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test =
train_test_split(X_agent_fly, Y_agent_fly, test_size=0.2,
random_state=42)
```

```
# Determine input dimension from the data
input_dim = X_train.shape[1]

# Build the model
model = Sequential()
model.add(Dense(256, input_shape=(input_dim,),
activation='relu'))
model.add(BatchNormalization())
model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(64, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(32, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(1))  # For regression, you may also
consider a linear activation

# Compile the model
model.compile(optimizer='adam',
loss='mean_absolute_error', metrics=['mae'])

# Custom Callback to record loss at the end of each epoch
and plot the history at the end
class EpochLossHistory(Callback):
    def __init__(self):
        super().__init__()
        self.epochs = []
        self.train_losses = []
        self.val_losses = []

    def on_epoch_end(self, epoch, logs=None):
        logs = logs or {}
        self.epochs.append(epoch + 1)  # Epoch number
starting from 1
        self.train_losses.append(logs.get("loss"))
        self.val_losses.append(logs.get("val_loss"))

    def on_train_end(self, logs=None):
        plt.figure(figsize=(10, 6))
        plt.plot(self.epochs, self.train_losses,
marker='o', label='Train Loss')
        plt.plot(self.epochs, self.val_losses,
marker='o', label='Validation Loss')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.title('Training and Validation Loss per
Epoch')
        plt.legend()
        plt.show()
```

```
# Instantiate callbacks
epoch_loss_history = EpochLossHistory()
early_stopping = EarlyStopping(monitor='val_loss',
patience=3, verbose=1, restore_best_weights=True)
checkpoint = ModelCheckpoint("final_model_agent_fly.h5",
monitor='val_loss', mode='min', save_best_only=True,
verbose=1)

# Train the model with the callbacks
history = model.fit(
    X_train, y_train,
    epochs=30,
    batch_size=128,
    validation_data=(X_test, y_test),
    callbacks=[epoch_loss_history, early_stopping,
checkpoint]
)

# Optionally, save the final model (the checkpoint
callback already saved the best model)
model.save("final_model_agent_fly.h5")

# Predict using the model and plot histograms of
predictions vs. actual values
y_predict = model.predict(X_test)
# y_predict = tf.inverse_transform(y_predict_tf)
plt.hist(y_predict, bins=100, alpha=0.5,
label='Predicted')
plt.hist(Y_agent_fly, bins=100, alpha=0.5,
label='Actual')
plt.legend()
plt.show()

# Evaluate the model
loss, mae = model.evaluate(X_test, y_test)
print(f"Loss: {loss}, MAE: {mae}")
```