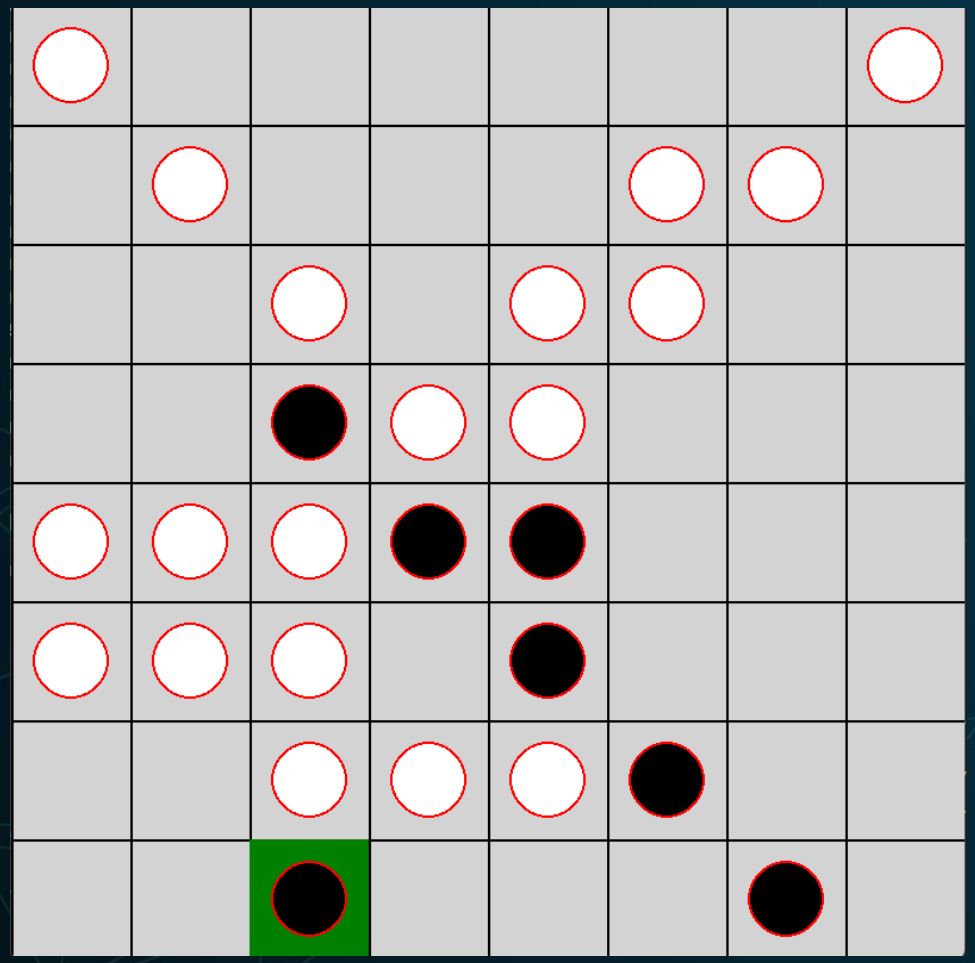


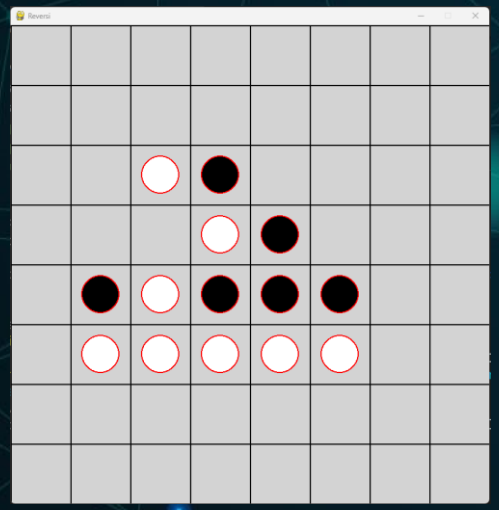
הדגמת משחק
Reversi / Othelo

DQN
Deep Q-learning Neural Network

קריית החינוך
פארק המדע
בית לערכים
למצוינות ולחדשנות

תוכן השיעור

- נדגים כיצד לבנות סוכן AI הלומד לשחק Reversi באמצעות למידת חיזוק.
- נממש את האלגוריתם DQN לצורך בניית הבינה המלאכותית.
- נדגים כיצד ניתן לאמן את המודל שלנו באמצעות משחק כנגד סוכנים שונים:
  - אימון כנגד סוכן אקראי – רנדומלי
  - אימון כנגד סוכן המשחק לפי כללים קובעים מראש – fix_Agent.
- אימון רשת הנוירונים לשחק את שני השחקנים, תוך כדי משחק עצמי.



- הקוד המלא:
- https://github.com/MarkmanGilad/Reversi_AI_new

המחלקה state

```python
class State:
    def __init__(self, board= None, player = 1, legal_actions = []) -> None:
        self.board = board
        self.player = player
        self.legal_actions = legal_actions

    def get_opponent (self):···

    def switch_player(self):···

    def score (self):···

    def __eq__(self, other) ->bool:···

    def __hash__(self) -> int:···

    def copy (self):···

    def reverse (self):···

    def toTensor (self, device = torch.device('cpu')) -> tuple:···

    [staticmethod]
    def tensorToState (state_tuple, player):···
```

- המצב מוגדר באמצעות לוח המשחק המקבל ערכים 0, 1, -1.
  - 0 = ריק
  - 1 = לבן
  - -1 = שחור
- בעת יצירת מצב אנחנו יוצרים רשימה של כל הפעולות החוקיות באותו מצב (לא הכרחי אך חוסך לנו זמן ריצה בהמשך).

המחלקה Graphics

```
class Graphics:
    def __init__(self): ···

    def draw(self, board): ···

    def draw_Lines(self): ···

    def draw_all_pieces(self, board): ···

    def draw_piece(self, row_col, player): ···

    def calc_pos(self, row_col): ···

    def calc_base_pos(self, row_col): ···

    def calc_row_col(self, pos): ···

    def calc_color(self, player): ···

    def draw_square(self, row_col, color): ···

    def blink(self, row_col, color, board): ···
```

- המחלקה Graphics נבנתה על מנת להדפיס את לוח המשחק על המסך.
- הפעולה העיקרית היא ()draw המקבלת את הלוח מתוך המצב state.board ומדפיסה אותו בצורה גרפית על המסך.
- בנוסף למחלקה פעולות המחשבות את השורה והעמודה שנבחרת על ידי המשתמש באמצעות העכבר ()calc_row_col.

המחלקה Reversi

```python
class Reversi:
    def __init__(self, state:State = None) -> None:
        if state is None:
            self.state = self.get_init_state((ROWS, COLS))
        else:
            self.state = state

    def get_init_state(self, Rows_Cols = (ROWS, COLS)):...

    def is_inside(self, row_col, state: State):...

    def flip_piece(self, row_col, state: State):...

    def check_legal_line(self, start_row_col: tuple[int, int], dir_row_col: tuple[ir

    def is_legal_move(self, row_col, state: State):...

    def reverse_line (self, row_col, dir_row_col, count, state: State) -> None:...

    def move(self, action: tuple[int, int], state: State) -> bool:...

    def set_legal_actions(self, state: State):...

    def get_legal_actions(self, state:State) -> list:...

    def is_end_of_game(self, state: State) -> bool:...

    def get_next_state(self, action, state:State) -> State:...

    def get_all_next_states (self, state: State) -> tuple:...

    def toTensor (self, list_states, device = torch.device('cpu')) -> tuple:...

    def reward (self, state : State, action = None) -> tuple:...
```

- המחלקה Reversi היא הסביבה שלנו.
- המאפיין state מתאר את מצב המשחק הנוכחי.
- הפעולה move מקבלת מצב ופעולה ומשנה את המצב.
- הפעולה get_next_state מקבלת מצב ופעולה ומחזירה state חדש.
- העולה reward מחזירה:
  - לבן ניצח = 1
  - שחור ניצח = -1
  - תיקו = 0
  - כל פעולה שאינה סוף משחק = 0

המחלקה Human_Agent

קריית החינוך
פארק המדע
בית לערכים
למצוינות ולחדשנות

- המחלקה נועדה לשמש ממשק למשחק של המשתמש עם העכבר.
- המחלקה מחזירה את row, col אם המשתמש בחר פעולה חוקית. אחרת מחזירה None.

```
class Human_Agent:

    def __init__(self, player: int) -> None:
        self.player = player

    def get_Action (self, events= None, graphics: Graphics = None, state : State = None,
        for event in events:
            if event.type == pygame.MOUSEBUTTONDOWN:
                pos = pygame.mouse.get_pos()
                row_col = graphics.calc_row_col(pos)
                if row_col in state.legal_actions:
                    return row_col
        return None
```

המחלקה Game

```
def main ():
    run = True
    clock = pygame.time.Clock()
    graphics.draw(env.state.board)
    player = player1

    while(run):
        clock.tick(FPS)
        events = pygame.event.get()
        for event in events:
            if event.type == pygame.QUIT:
                run = False
                exit()
        action = player.get_Action(events=events, graphics=
        if action:
            env.move(action, env.state)
            graphics.blink(action, GREEN, env.state.board)
            player = switchPlayers(player)
        graphics.draw(env.state.board)
        pygame.display.update()
        if env.is_end_of_game(env.state):
            score = env.state.score()
            print ("score = ", score)
            pygame.time.delay(DELAY)
            env.state = env.get_init_state()
            player = player1
            graphics.draw(env.state.board)
    pygame.time.delay(500)
    pygame.quit()
    print("End of game")
    score = env.state.score()
    print ("score = ", score)

def switchPlayers(player):···
```

- המחלקה Game הינה הלולאה הראשית של המשחק.
- אנו מגדירים שני שחקנים: player1, player2 ובכל תור מחליפים ביניהם.
- בכל איטרציה מקבלים מהשחקן שתורו לשחק פעולה חוקית (אם none אז לא בחר פעולה וממתינים).
- בכל איטרציה בודקים אם הגענו למצב סופי.

המחלקה Random_Agent

קריית החינוך
פארק המדע
בית לערכים
למצוינות ולחדשנות

- המחלקה Random_Agent מייצגת סוכן הבוחר פעולה חוקית באופן אקראי.
- המחלקה יכולה לשמש לאימון את רשת הנוירונים, וכן לבדיקה לאיכות רשת הנוריונים שיצרנו.

```python
class Random_Agent:
    def __init__(self, env : Reversi, player = None) -> None:
        self.env = env

    def get_Action (self, events = None, graphics=None, state: State = None, epoch = 0, train = None):
            action = random.choice(state.legal_actions)
            return action
```

קריית החינוך
פארק המדע
בית לערכים
למצוינות ולחדשנות

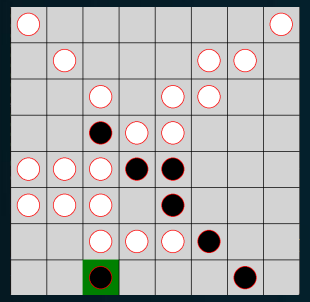
המחלקה Fix_Agent

```python
class Fix_Agent:
    def __init__(self, env, player = 1, train = False, random = 0.10) -> None:
        self.env  = env
        self.player = player
        self.train = train
        self.random = random

    def value(self, state: State):
        v = np.array([[100, -25, 10, 5, 5, 10, -25, 100],
                      [-25, -25, 2, 2, 2, 2, -25, -25],
                      [10, 2, 5, 1, 1, 5, 2, 10],
                      [5,2,1,2,2,1,2,5],
                      [5,2,1,2,2,1,2,5],
                      [10, 2, 5, 1, 1, 5, 2, 10],
                      [-25, -25, 2, 2, 2, 2, -25, -25],
                      [100, -25, 10, 5, 5, 10, -25, 100]])
        board = state.board
        return (board*v).sum()

    def get_Action (self, events = None, graphics=None, state: State = None, epoch = 0, train = True):
        legal_actions = state.legal_actions
        if self.train and train and random.random() < self.random:
            return random.choice(legal_actions)
        next_states, _ = self.env.get_all_next_states(state)
        values = []
        for next_state in next_states:
            values.append(self.value(next_state))
        if self.player == 1:
            maxIndex = values.index(max(values))
            return legal_actions[maxIndex]
        else:
            minIndex = values.index(min(values))
            return legal_actions[minIndex]
```

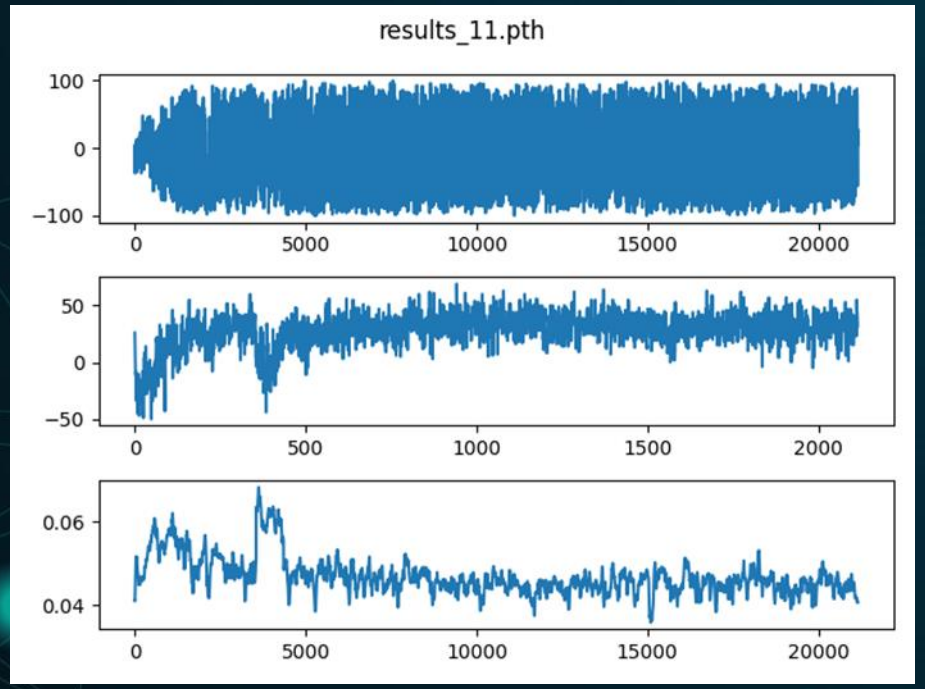
- סוכן אשר משתמש בטבלת ערכים קבועה לחישוב ערך המצב.
- כל משבצת מעניקה ניקוד.
- הסוכן בודק את כל הפעולות האפשריות ובוחר את הפעולה שערך המצב לאחריה יהיה הגבוה ביותר.

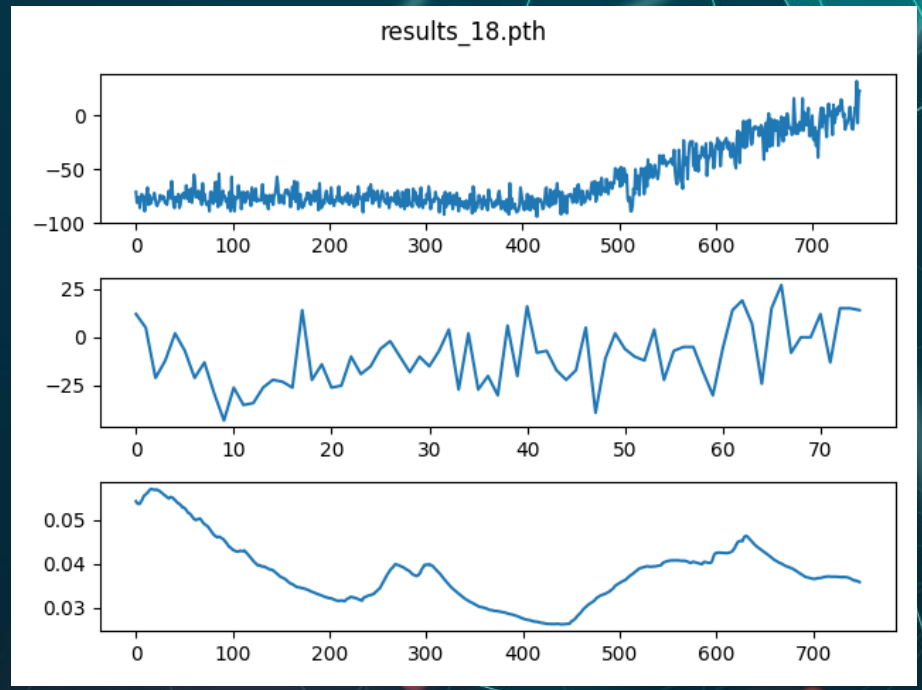


קריית החינוך
פארק המדע
בית לערכים
למצוינות ולחדשנות

רברסי באמצעות האלגוריתם DQN

Deep Q-learning Neural Network





Deep Q-learning (DQN)

Initialize Q(s,a,w) with arbitrary w.

Initialize $\hat{Q}$(s,a,w^-) with arbitrary w^-.

Initialize replay-buffer RB with size N.

For epoch in epochs (for each game):

- Initialize s
- While s is not Terminal:
  - Choose A From S using Q and e-greedy
  - Interact with environment and get s, a, r, s'
  - Store transition (s, a, r, s') in RB
  - $s = s'$
  - sample random minibatch from RB
  - Calculate $Q(s, a, w)$ - forward
  - Calculate loss with MSELoos :
    - If s' is Terminal: $loss = \left(r - Q(s, a, w)\right)^2$
    - else: $loss = \left(r + \gamma \cdot max_a \hat{Q}(s', a, w^-) - Q(s, a, w)\right)^2$
  - Calculate gradients (backwards): $loss.backward()$
  - Update W : $Optim.SGD.step()$
- Every C epochs $w^- \leftarrow w$

Replay_Buffer

```python
capacity = 100000
end_priority = 2

class ReplayBuffer:
    def __init__(self, capacity= capacity, path = None) -> None:
        if path:
            self.buffer = torch.load(path).buffer
        else:
            self.buffer = deque(maxlen=capacity)

    def push (self, state : State, action, reward, next_state: State, done):
        self.buffer.append((state.toTensor(), torch.from_numpy(np.array(action)), torch.tensor(reward), next_state.toTensor(), torch.tensor(done)))
        if done:
            for i in range(end_priority):
                self.buffer.append((state.toTensor(), torch.from_numpy(np.array(action)), torch.tensor(reward), next_state.toTensor(), torch.tensor(done)))

    def sample (self, batch_size):
        if (batch_size > self.__len__()):
            batch_size = self.__len__()
        state_tensors, action_tensor, reward_tensors, next_state_tensors, dones = zip(*random.sample(self.buffer, batch_size))
        state_boards, state_actions = zip(*state_tensors)
        states = torch.vstack(state_boards), state_actions
        actions= torch.vstack(action_tensor)
        rewards = torch.vstack(reward_tensors)
        next_board, next_actions = zip(*next_state_tensors)
        next_states = torch.vstack(next_board), next_actions
        done_tensor = torch.tensor(dones).long().reshape(-1,1)
        return states, actions, rewards, next_states, done_tensor

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

DQN

```python
# Parameters
input_size = 66 # state: board = 8 * 8 = 64 + action (2)
layer1 = 64
output_size = 1 # Q(state, action)
gamma = 0.99


class DQN (nn.Module):
    def __init__(self, device = torch.device('cpu')) -> None: ...

    def forward (self, x):
        x = self.linear1(x)
        x = F.leaky_relu(x)
        x = self.output(x)
        return x

    def loss (self, Q_value, rewards, Q_next_Values, Dones ):
        Q_new = rewards + gamma * Q_next_Values * (1- Dones)
        return self.MSELoss(Q_value, Q_new)

    def load_params(self, path): ...

    def save_params(self, path): ...

    def copy (self): ...

    def __call__(self, states, actions): ...
```

- רשת הנוירונים שנבחרה כוללת:
- שכבה קלט של 66 מאפיינים:
  - לוח 64 מאפיינים.
  - פעולה 2 מאפיינים.
- שכבה נסתרת של 64 נוירונים.
- אקטיבציה LeakyRELU
- שכבת פלט 1 – הערך.

- ביצענו גם ניסויים עם רשת נוירונים עמוקה הכוללת שתי שכבות נסתרות: 64, 128

DQN_Agent

- המחלקה מממשת סוכן המשתמש ב Deep Q-Learning ללימוד המשחק.
- המחלקה כוללת את הפעולות הבאות (ראה פירוט בהמשך)

```
class DQN_Agent:
    def __init__(self, player = 1, parametes_path = None, train = True, env= None):···

    def setTrainMode (self):···

    def get_Action (self, state:State, epoch = 0, events= None, train = True, graphics = None, black_state = None) -> tuple:···

    def get_Actions (self, states_tensor: State, dones) -> torch.tensor:···

    def epsilon_greedy(self,epoch, start = epsilon_start, final=epsilon_final, decay=epsiln_decay):···

    def loadModel (self, file):···

    def save_param (self, path):···

    def load_params (self, path):···

    def __call__(self, events= None, state=None):
        return self.get_Action(state)
```

DQN_Agent: get_Action()

קריית החינוך
פארק המדע
בית לערכים
למצוינות ולחדשנות

```python
def get_Action (self, state:State, epoch = 0, events= None, train = True, graphics = None, black_state = None) -> tuple:
    actions = state.legal_actions
    if self.train and train:
        epsilon = self.epsilon_greedy(epoch)
        rnd = random.random()
        if rnd < epsilon:
            return random.choice(actions)

    if self.player == 1:
        state_tensor, action_tensor = state.toTensor()
    elif not black_state:
        black_state = state.reverse()
        state_tensor, action_tensor = black_state.toTensor()
    else:
        state_tensor, action_tensor = black_state.toTensor()

    expand_state_tensor = state_tensor.unsqueeze(0).repeat((len(action_tensor),1))

    with torch.no_grad():
        Q_values = self.DQN(expand_state_tensor, action_tensor)
    max_index = torch.argmax(Q_values)
    return actions[max_index]
```

DQN_Agent: get_Actions()

קריית החינוך
פארק המדע
בית לערכים
למצוינות ולחדשנות

```python
def get_Actions (self, states_tensor: State, dones) -> torch.tensor:
    actions = []
    boards_tensor = states_tensor[0]
    actions_tensor = states_tensor[1]
    for i, board in enumerate(boards_tensor):
        if dones[i].item():
            actions.append((0,0))
        else:
            actions.append(self.get_Action(State.tensorToState(state_tuple=(boards_tensor[i],actions_tensor[i]),player=self.player), train=False))
    return torch.tensor(actions)
```

DQN_Agent: epsilon_greedy()

```python
def epsilon_greedy(self,epoch, start = epsilon_start, final=epsilon_final, decay=epsiln_decay):
    res = final + (start - final) * math.exp(-1 * epoch/decay)
    return res
```

Trainer

- הקוד Trainer נועד לאמן את רשת הנוירונים לפי האלגוריתם DQN, באמצעות שימוש ב Replay_Buffer.
- בנינו קבצים נפרדים לסוגי אימון שונים:
- Trainer_White – מאמן שחקן ראשון (שחקן לבן – כלי מס' 1), באמצעות משחק כנגד סוכן Fix_Agent. ניתן לשנות בקלות לאימון כנגד סוכן רנדומלי.
- Trainer_Black – מאמן שחקן שני באמצעות "היפוך" הלוח, כך שהשחקן משחק אף הוא עם כלי מס' 1. האימון נעשה באמצעות סוכן Fix_Agent.
- Trainer_BW – אימון רשת אחת לשחק את שני השחקנים, באמצעות משחק האחד כנגד השני. השחקן השני מבצע "היפוך" של הלוח.

Trainer_white

```python
def main ():

    player1 = DQN_Agent(player=1, env=env,parametes_path=path_load)
    player_hat = DQN_Agent(player=1, env=env, train=False)
    Q = player1.DQN
    Q_hat = Q.copy()
    Q_hat.train = False
    player_hat.DQN = Q_hat

    # player2 = Fix_Agent(player=-1, env=env, train=True, random=0)   #0.1
    player2 = Random_Agent(player=-1, env=env)
    buffer = ReplayBuffer(path=buffer_path) # None

    results_file = torch.load(results_path)
    results = results_file['results'] # []
    avgLosses = results_file['avglosses']     #[]
    avgLoss = avgLosses[-1] #0
    loss = torch.Tensor([0])
    res = 0
    best_res = -200
    loss_count = 0
    tester = Tester(player1=player1, player2=Random_Agent(player=-1, env=env), env=env)
    tester_fix = Tester(player1=player1, player2=player2, env=env)
    random_results = torch.load(random_results_path)   # []
    best_random = max(random_results)


    # init optimizer
    optim = torch.optim.Adam(Q.parameters(), lr=learning_rate)
    scheduler = torch.optim.lr_scheduler.StepLR(optim,100000*30, gamma=0.90)
    # scheduler = torch.optim.lr_scheduler.MultiStepLR(optim,[30*50000, 30*100000, 30*25000
```

- הקוד מאמן שחקן לבן (שחקן ראשון).
- האימון נעשה על פי האלגוריתם DQN, באמצעות שתי רשתות נוירונים.
- בכל C צעדים מעודכנת רשת הנוירונים השניה (Q_hat).

Trainer_white

```python
for epoch in range(start_epoch, epochs):
    print(f'epoch = {epoch}', end='\r')
    state_1 = env.get_init_state()
    while not env.is_end_of_game(state_1):
        # Sample Environement
        action_1 = player1.get_Action(state_1, epoch=epoch)
        after_state_1 = env.get_next_state(state=state_1, action=action_1)
        reward_1, end_of_game_1 = env.reward(after_state_1)
        if end_of_game_1:
            res += reward_1
            buffer.push(state_1, action_1, reward_1, after_state_1, True)
            break
        state_2 = after_state_1
        action_2 = player2.get_Action(state=state_2)
        after_state_2 = env.get_next_state(state=state_2, action=action_2)
        reward_2, end_of_game_2 = env.reward(state=after_state_2)
        if end_of_game_2:
            res += reward_2
        buffer.push(state_1, action_1, reward_2, after_state_2, end_of_game_2)
        state_1 = after_state_2

        if len(buffer) < MIN_Buffer:
            continue
```

- לולאת האימון כוללת את השלבים הבאים:
- State_1 – מצב.
- Action_1 – פעולה.
- After_state_1 – המצב לאחר הפעולה.
- State_2 <- after_state_1
- State_2 – מצב היריב.
- Action_2 – הפעולה של היריב.
- After_state_2 – המצב הבא.
- State_1 <- after_state_2

Trainer_white

```
    # Train NN
    states, actions, rewards, next_states, dones = buffer.sample(batch_size)
    Q_values = Q(states[0], actions)
    next_actions = player_hat.get_Actions(next_states, dones) #fixed bug
    with torch.no_grad():
        Q_hat_Values = Q_hat(next_states[0], next_actions) #todo: use the value

    loss = Q.loss(Q_values, rewards, Q_hat_Values, dones)
    loss.backward()
    optim.step()
    optim.zero_grad()

    scheduler.step()
    if loss_count <= 1000:
        avgLoss = (avgLoss * loss_count + loss.item()) / (loss_count + 1)
        loss_count += 1
    else:
        avgLoss += (loss.item()-avgLoss)* 0.00001

if epoch % C == 0:
        Q_hat.load_state_dict(Q.state_dict())

if (epoch+1) % 100 == 0:
    print(f'\nres= {res}')
    avgLosses.append(avgLoss)
    results.append(res)
    if best_res < res:
        best_res = res
        if best_res > 75 and tester_fix(1) == (1,0):
            player1.save_param(path_best)
    res = 0
```

- האימון מתחלק לשני שלבים:
  - דגימת הסביבה ושמירה ב Buffer.
  - אימון רשת הנוירונים.
- ערכי המצב הנוכחי מחושבים באמצעות Q
- ערכי המטרה בנוסחת בלמן מחושבים באמצעות Q_hat.
- Q.hat <- Q בכל C צעדים.

Trainer_white

קריית החינוך
פארק המדע
בית לערכים
למצוינות ולחדשנות

- שמירת המודל והדפסות שונות.
- הבדיקה של ביצועי המודל נעשית גם במדידת 100 משחקים מול סוכן רנדומלי.

```
if (epoch+1) % 1000 == 0:
    test = tester(100)
    test_score = test[0]-test[1]
    if best_random < test_score and tester_fix(1) == (1,0):
        best_random = test_score
        player1.save_param(path_best_random)
    print(test)
    random_results.append(test_score)

if (epoch+1) % 5000 == 0:
    torch.save({'epoch': epoch, 'results': results, 'avglosses':avgLosses}, results_path)
    torch.save(buffer, buffer_path)
    player1.save_param(path_Save)
    torch.save(random_results, random_results_path)
if len(buffer) > MIN_Buffer:
    print (f'epoch={epoch} loss={loss:.5f} Q_values[0]={Q_values[0].item():.3f} avgloss={avgLoss:.5f}', end=" ")
    print (f'learning rate={scheduler.get_last_lr()[0]} path={path_Save} res= {res} best_res = {best_res}')
```

Trainer_black

```python
for epoch in range(start_epoch, epochs):
    print(f'epoch = {epoch}', end='\r')
    state = env.get_init_state()
    action = player2.get_Action(state=state)
    state_1 = env.get_next_state(state=state, action=action)
    state_1_R = state_1.reverse()
    while not env.is_end_of_game(state_1_R):
        # Sample Environement
        action_1_R = player1.get_Action(state_1_R, epoch=epoch, black_state=state_1_R) #
        after_state_1_R = env.get_next_state(state=state_1_R, action=action_1_R)
        reward_1_R, end_of_game_1_R = env.reward(after_state_1_R)
        if end_of_game_1_R:
            res += reward_1_R
            buffer.push(state_1_R, action_1_R, reward_1_R, after_state_1_R, True)
            break
        state_2 = after_state_1_R.reverse()
        action_2 = player2.get_Action(state=state_2)
        after_state_2 = env.get_next_state(state=state_2, action=action_2)
        after_state_2_R = after_state_2.reverse()
        reward_2_R, end_of_game_2 = env.reward(state=after_state_2_R)
        if end_of_game_2:
            res += reward_2_R
        buffer.push(state_1_R, action_1_R, reward_2_R, after_state_2_R, end_of_game_2)
        state_1_R = after_state_2_R

        if len(buffer) < MIN_Buffer:
            continue
```

- האלגוריתם דומה לאימון השחקן הלבן בשינויים הבאים:
- השחקן משחק אף הוא עם החיילים הלבנים (ערך 1 בלוח המצב).
- לצורך כך מבצעים הפיכה של הלוח, כך שהחיילים השחורים הופכים להיות לבנים.
- אין התנגשות שכן שחקן לבן תמיד מספר חיילים זוגי ושחקן שחור אי זוגי.
- במקרים מתאימים ניתן להוסיף למצב גם פרמטר של מי התור (1 או -1), ולהגדיל את הקלט ל- 77 פרמטרים.

Trainer_BW

```python
def main ():

    player1 = DQN_Agent(player=1, env=env,parametes_path=path_load)
    player2 = DQN_Agent(player=-1, env=env,parametes_path=None)
    player_hat = DQN_Agent(player=1, env=env,parametes_path=None)


    Q = player1.DQN
    player2.DQN = Q
    Q_hat = Q.copy()
    Q_hat.train = False
    player_hat.DQN = Q_hat
```

```python
# init optimizer
optim = torch.optim.Adam(Q.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.StepLR(optim,100000*30, gamma=0.90)
# scheduler = torch.optim.lr_scheduler.MultiStepLR(optim,[30*50000, 30*100000
```

- אימון של הרשת לשחק את שני השחקנים.
- שני השחקנים "מסתכלים" על הלוח של המצב באותה צורה – החייל שלהם הוא 1 והיריב הוא 1-.
- לצורך כך אנחנו מחזיקים מצבים כפולים:
  - מצב רגיל state – לשחקן הלבן
  - מצב הופכי state_R – לשחקן השחור לאחר שהוחלפו החיילים והשחקן השחור משחק גם כן עם חייל מספר 1.

Traine_BW

```python
for epoch in range(start_epoch, epochs):
    print(f'epoch = {epoch}', end='\r')
    state_1 = env.get_init_state()
    state_2, action_2, after_state_2 = None, None, None
    state_2_R, after_state_2_R = None, None
    while not env.is_end_of_game(state_1):
    # Sample Environement
        # white
        action_1 = player1.get_Action(state_1, epoch=epoch)
        action_1_R = action_1
        after_state_1 = env.get_next_state(state=state_1, action=action_1)
        after_state_1_R = after_state_1.reverse()
        reward_1, end_of_game_1 = env.reward(after_state_1)
        reward_1_R, end_of_game_1_R = -reward_1, end_of_game_1
        if state_2: # not the first action in a game
                buffer.push(state_2_R, action_2_R, reward_1_R, after_state_1_R, end_of_game_1_R)
        if end_of_game_1:
            res += reward_1
            buffer.push(state_1, action_1, reward_1, after_state_1, True) # for white
            state_1 = after_state_1
        else:
            # Black
            state_2 = after_state_1
            state_2_R = after_state_1_R
            action_2 = player2.get_Action(state=state_2, epoch=epoch, black_state=state_2_R)
            action_2_R = action_2
            after_state_2 = env.get_next_state(state=state_2, action=action_2)
            after_state_2_R = after_state_2.reverse()
            reward_2, end_of_game_2 = env.reward(state=after_state_2)
            reward_2_R, end_of_game_2_R = -reward_2, end_of_game_2
            if end_of_game_2:
                res += reward_2
                buffer.push(state_2_R, action_2_R, reward_2_R, after_state_2_R, True) # for black
            buffer.push(state_1, action_1, reward_2, after_state_2, end_of_game_2)
            state_1 = after_state_2

        if len(buffer) < MIN_Buffer:
            continue
```

- בכל תור אנחנו שומרים ב Buffer גם את הנתונים עבור השחקן הלבן וגם עבור השחקן השחור.

State1, action1, reward1, after_state1

State2 <- after_state1

State2, action2, reward2, after_state2

State1 <- after_state2

State1, action1, reward1, after_state1

Push not end state:

State1, action1, reward2, after_state2

State2_R, action2_R, reward1_R, after_state_1_R

- לולאת האימון לא משתנה. כיוון ששני השחקנים משחקים עם חייל מס' 1 האימון הוא זהה לשניהם.

Tester

```python
class Tester:
    def __init__(self, env, player1, player2) -> None:
        self.env = env
        self.player1 = player1
        self.player2 = player2


    def test (self, games_num):
        env = self.env
        player = self.player1
        player1_win = 0
        player2_win = 0
        games = 0
        while games < games_num:
            action = player.get_Action(state=env.state, train = False)
            env.move(action, env.state)
            player = self.switchPlayers(player)
            if env.is_end_of_game(env.state):
                score = env.state.score()
                if score > 0:
                    player1_win += 1
                elif score < 0:
                    player2_win += 1
                env.state = env.get_init_state()
                games += 1
                player = self.player1
        return player1_win, player2_win
```

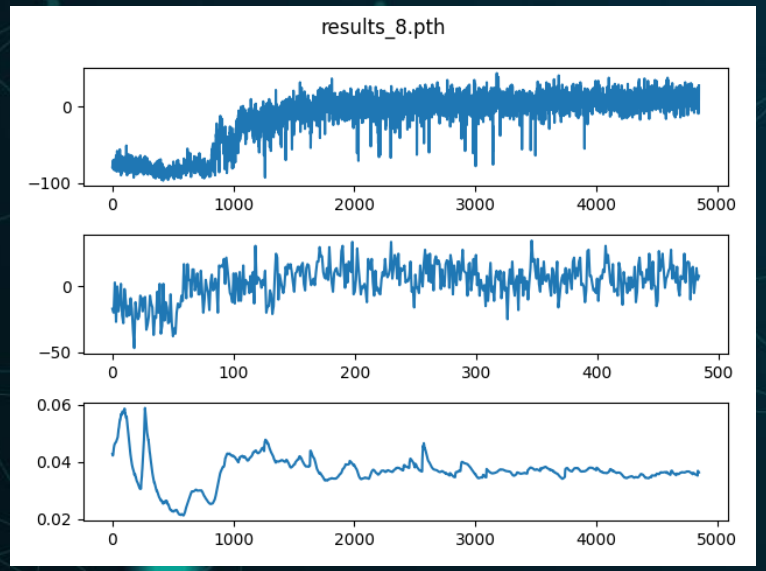
- המחלקה Tester מבצעת מספר קבוע של משחקים בין שני סוכנים ומחזירה את מספר הניצחונות של כל אחד מהסוכנים.
- המחלקה נועדה כדי לבדוק אם התקדמות האימון של המודל, מול סוכנים אחרים ולאו דווקא הסוכן מולו המודל מתאמן.

תוצאות אימון ומסקנות

- עומק רשת הנוירונים
- שמירה של הנתונים. שמירה של מודל מיטבי.
- שינוי קצב הלימוד ופרמטרים נוספים.
- אימון כנגד סוכן סוכן קבוע – בעיית overfitting. הפתרון אימון כנגד סוכן קבוע שבחלק מהמקרים (כל 5%-10%) מבצע צעד רנדומלי.
- אימון כנגד סוכן רנדומלי – קושי ללמוד אסטרטגיה מנצחת.
- אימון כנגד סוכן AI אחד עם אותה רשת – התוצאות הטובות ביותר.

תוצאות אימון – תרשים 8

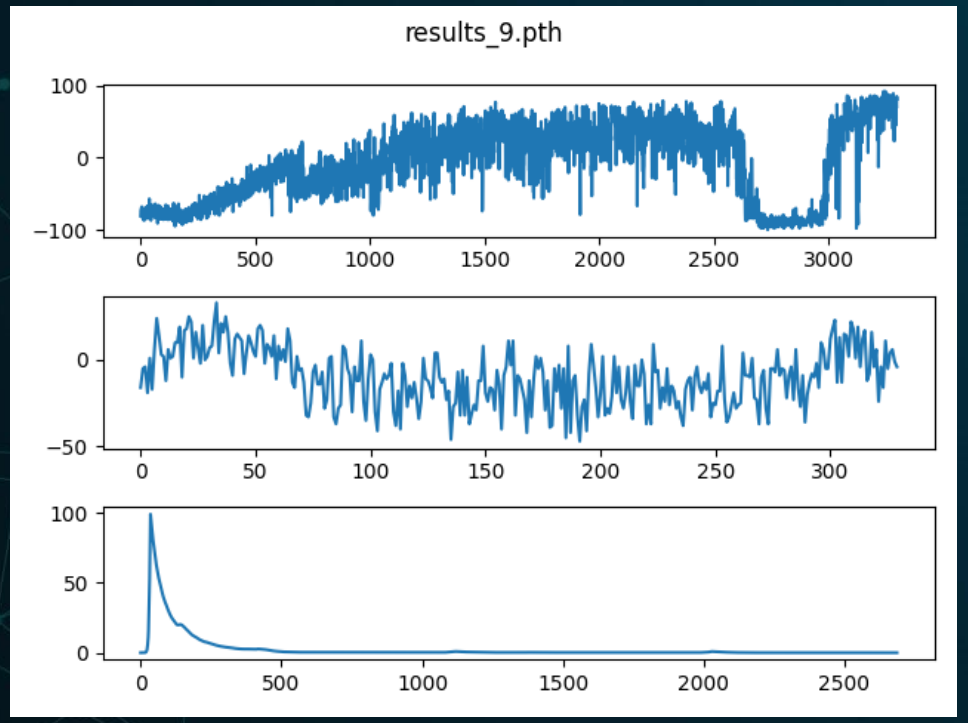
- אימון שחקן לבן מול fix_Agent עם 5% צעדים רנדומליים. C=350; LR = 0.01 עם הפחתה של 0.5 לאחר 25000, 50000, 100000, 250000, 500000 משחקים. המודל עם שכבה נסתרת אחת של 64 נוירונים.
  - תרשים ראשון – תוצאות המשחקים מול המודל המתחרה (fix Agent 5% random).
  - תרשים שני – תוצאות 100 משחקים מול שחקן רנדומלי.
  - תרשים שלישי – טעות ממוצעת Average loss.

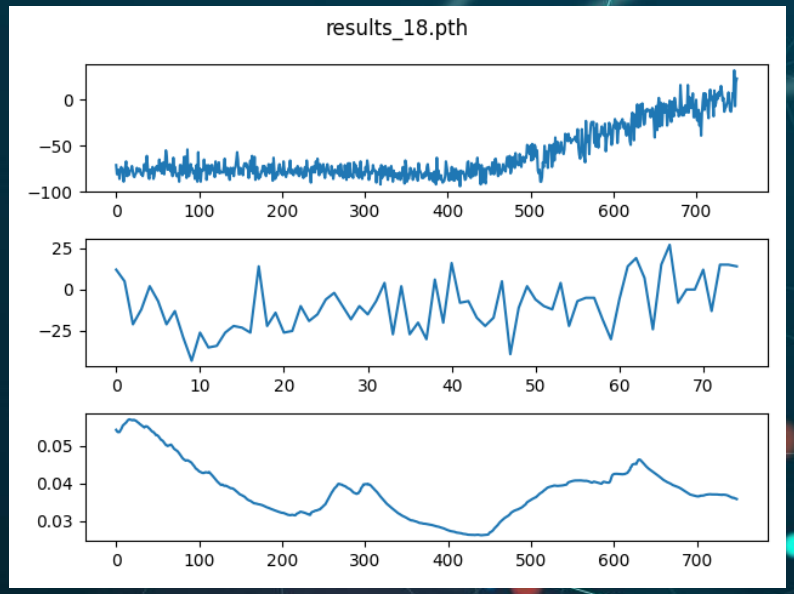


- ניתן לראות כי המודל למד לשחק מול היריב חלקית.
- המודל השתפר מעט במשחק מול שחקן רנדומלי.
- השיפור החל לאחר 100000 משחקים.

תוצאות אימון – תרשים 9, 18

- אימון שחקן לבן מול fix_Agent ללא צעדים רנדומליים. C=50, 200. קצב הלימוד כמו בתרשים הקודם. המודל עם שכבה נסתרת אחת של 64 נוירונים.
- המודל הצליח ללמוד לנצח את השחקן הקבוע באופן מובהק, יחד עם זאת לא הצליח לנצח שחקן רנדומלי (overfitting).
- שיפר בתוצאות החל לאחר 30000 משחקים.

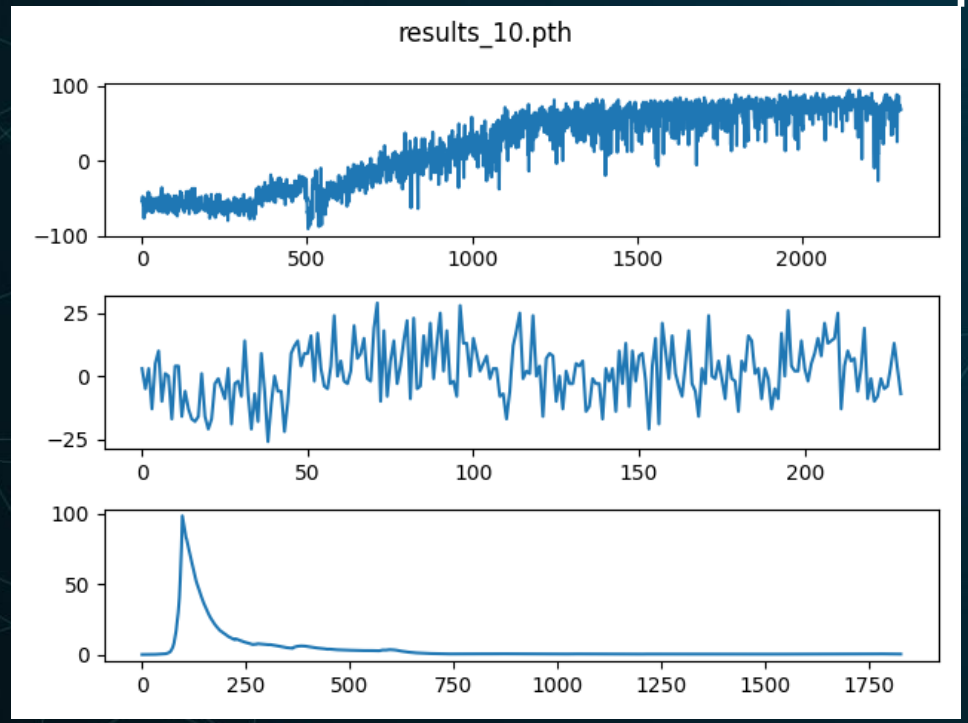




- <u>תרשים 18:</u>
- C=200
- Decay כמו הקודם
- שיפור החל 450000
- שכבה נסתרת אחת

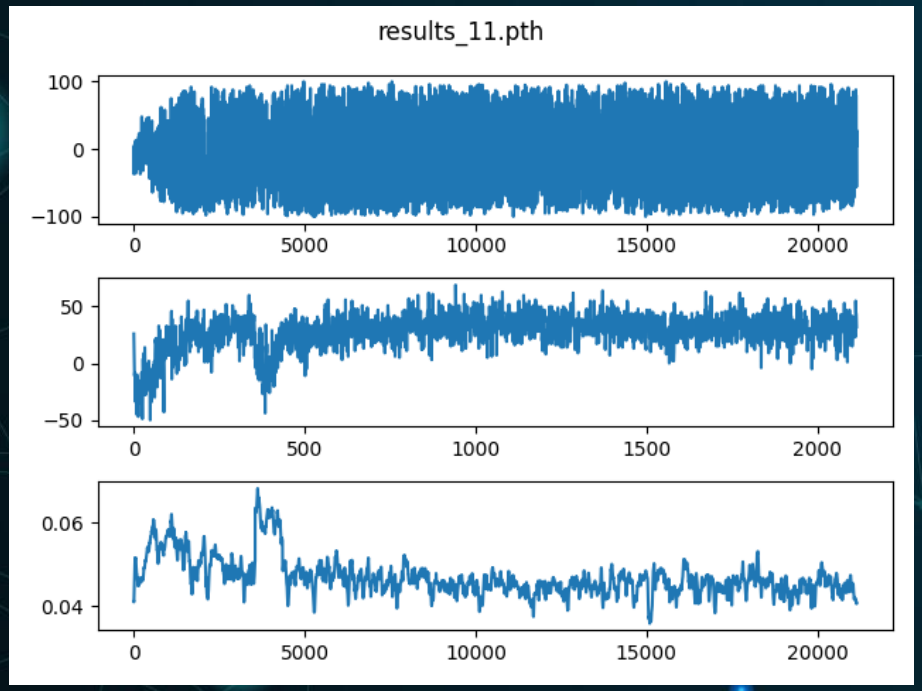
תוצאות אימון – תרשים 10

- אימון שחקן שחור מול fix_Agent ללא צעדים רנדומליים. C=100. קצב הלימוד משתנה בדומה לתרשים 8. המודל עם שכבה נסתרת אחת של 64 נוירונים.
- שיפור התוצאות החל לאחר 50000 משחקים.
- המודל הצליח באופן מובהק לנצח את השחקן הקבוע.
- המודל אינו מנצל שחקן רנדומלי (overfitting).



תוצאות אימון – תרשים 11

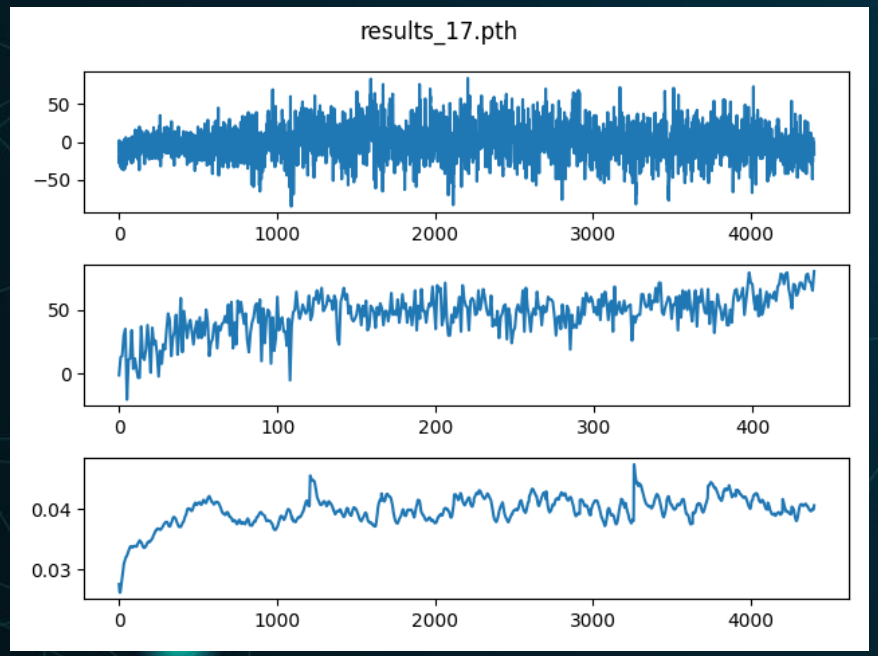
- אימון רשת לשחק את שני השחקנים באמצעות משחק בין שני סוכנים DQN. המודל כולל שכבה נסתרת אחת של 64 נוירונים. C=100. קצב הלמידה כמו במקרים הקודמים.
  - התרשים העליון מתאר את תוצאות המשחקים בין הסוכנים. אין יתרון לסוכן אחד.
  - התרשים השני מתאר תוצאות 100 משחקים של המודל מול שחקן רנדומלי שנמדדו כל 1000 משחקים (המודל היה שחקן מס' 1). יש למידה משמעותית.
  - התרשים השלישי – מתאר את הטעות הממוצעת.



- שיטת אימון זו הביאה את התוצאות הטובות ביותר. המודל טוב יותר משמעותית משחקן רנדומלי.
- המודל החל להשתפר לאחר 25000 משחקים.

תוצאות אימון – תרשים 17

- אימון רשת לשחק את שני השחקנים באמצעות משחק בין שני סוכנים DQN. המודל כולל שתי שכבות נסתרות (64, 128 נוירונים). C=100. קצב הלמידה מתחיל 0.01 ומוכפל ב 0.9 כל 100,000 משחקים.
  - התרשים העליון מתאר את תוצאות המשחקים בין הסוכנים. אין יתרון לסוכן אחד.
  - התרשים השני מתאר תוצאות 100 משחקים של המודל מול שחקן רנדומלי שנמדדו כל 1000 משחקים (המודל היה שחקן מס' 1). יש למידה משמעותית.
  - התרשים השלישי – מתאר את הטעות הממוצעת.



- שיטת אימון זו הביאה את התוצאות הטובות ביותר. המודל טוב יותר משמעותית משחקן רנדומלי.
- המודל החל להשתפר לאחר 25000 משחקים.
- האימון כולל 440,000 משחקים בלבד (לעומת 2 מיליון בתרשים הקודם). יתכן שאימון נוסף היה משפר את המודל.

סיכום

- אימון המודל מחייב לנסות פרמטרים רבים:
  - מבנה הרשת ועומקה. קשה לאמן רשת עמוקה ולכן מומלץ להתחיל עם רשת רדודה.
  - הפרמטרים C (מספר משחקים לעדכון רשת המטרה), Learning Rate (כולל שינוי קצב הלמידה עם התקדמות האימון).
  - גודל ה Replay Buffer.
- יש לבחור את המודל כנגדו יבוצע האימון. ניתן לשקול לבצע אימון כנגד מספר מודלים, ולהחליף את המודל תוך כדי למידה.
- שיפור המודל אינו לינארי. יש קפיצות ושינויים תוך כדי האימון. המודל הטוב ביותר אינו בהכרח המודל האחרון. על כן, יש לעקוב ולשמור את המודל הטוב ביותר.
- יש לתעד את הניסויים שאנו עורכים ולשמור את התוצאות שמתקבלות. לא אחת שינוי שעשינו במודל עלול לקלקל את המודל ולהביא לתוצאות פחות טובות.