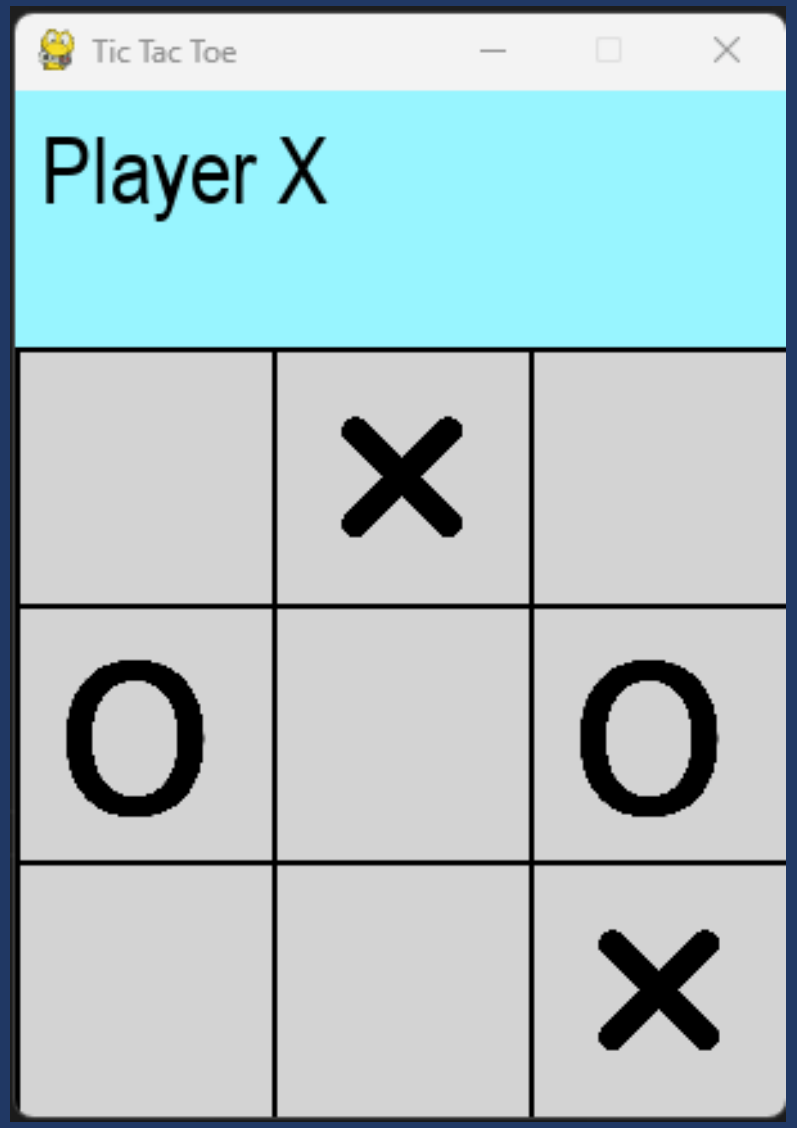



גלעד מרקמן

# REINFORCE MC

קישור ל GitHub

קריית החינוך
פארק המדע
בית לערכים
למצוינות ולחדשנות

# האלגוריתם REINFORCE MC

**REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)**

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G \leftarrow$ return from step $t$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t,\boldsymbol{\theta})$

- דגימה של הסביבה עד לסוף משחק (episode) ושמירה של שלישיות: State, Action, Reward לכל מצב t.
- חישוב ההחזרים G (סכום התגמולים עד סוף המשחק) לכל מצב t.
- עדכון רשת הנוירונים בסיום כל משחק בהתאם לנוסחה (SGD):

$$loss = -G_t * \ln \pi_\theta(a|s)$$

$$\theta = \theta - \alpha * \nabla loss(\theta)$$

קריית החינוך
פארק המדע
בית לערכים
למצוינות ולחדשנות

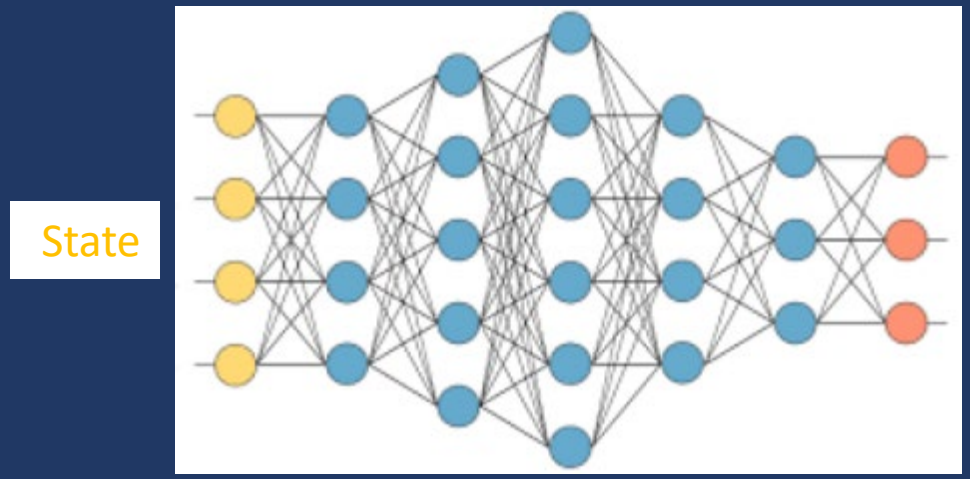
# שלבי המימוש

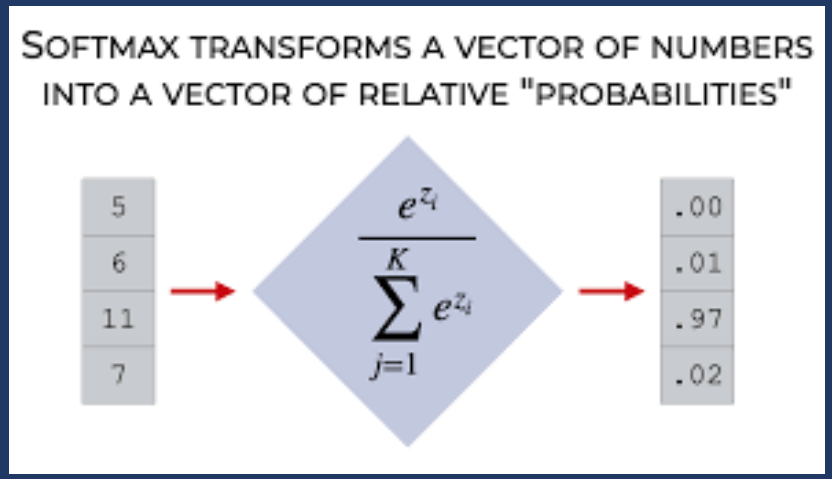
- בניית רשת הנוירונים
- בניית הזיכרון – שמירת הדגימות במהלך חקר הסביבה.
- בניית הסוכן והפונקציה העושה שימוש ברשת:
  - הפונקציה get_actions.
  - מיסוך פעולות לא חוקיות Masking.
  - הפונקציה remember
- אימון – לולאת האימון הדוגמת את הסביבה ומלמדת את הרשת.
- לימוד – בניית הפונקציה המבצעת את לימוד הרשת:
  - פונקציה compute_G המחשבת את ההחזרים $G_t$
  - פונקציה learn המעדכנת את הפרמטרים בהתאם לאלגוריתם.

# Policy Network

- רשת הנוירונים שלנו מקבלת state ומוציאה התפלגות לכל אחד מהפעולות (actions).
- ההתפלגות נעשית באמצעות פונקציית SoftMax.
- לספריית PyTorch יש מחלקה מיוחדת לטפל בהתפלגות בדידה:

`torch.distributions.categorical`

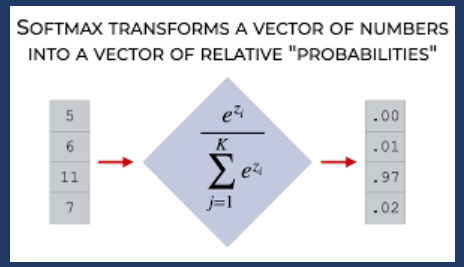


# torch.distributions.categorical

קריית החינוך פארק המדע
בית לערכים למצוינות ולחדשנות

- למחלקה שני מאפיינים (האובייקט נבנה עם אחד מהמאפיינים):
  - logits –טנסור של ערכים ללא נרמול (לפני הפעלת פונקציית softmax)
  - Probs – טנסור של הסתברויות (לאחר הפעלת softmax).
- בעת בניית האובייקט עם ה logits המחלקה מחשבת אוטומטית את ה probs באמצעות הפעלת softmax, כחלק מגרף חישוב.
- ה- logits מחושבים באמצעות log(p).
- הערכים המקוריים של ה logits איתם נבנה האובייקט לא נשמרים.
- המחלקה יכולה לטפל גם במספר דגימות (batch).




```
logits = torch.tensor([[1.2, -0.5, 1.3], [0.2, -1.1, 3.2]])
dist = Categorical(logits=logits)
print(dist.probs) # tensor([[0.4371, 0.0798, 0.4831], [0.0468, 0.0128, 0.9404]])
```

# torch.distributions.categorical



- פעולות המחלקה:
  - dist.sample() – מחזיר דגימה של ההסתברות. הפעולה בוחרת באופן הסתברותי אינדקס של אחת הדגימות, כאשר הסיכוי לבחירה תלוי בתוצאות. אם ההסתברות היא 0.8 לאינדקס מסויים, הסיכוי שהוא ייבחר הוא 80% אך ב 20% מהמקרים יתכן ויבחר אינדקס אחר.

  - dist.log_prob(index) – מחזיר לנו את ההתפלגות של ה action שבחרנו (במקרה של batch ניתן להעביר מספר אינדקסים).

```
probs = dist.log_prob(torch.tensor([2,1]))
```

  - Dist.entropy() – מחזיר לנו ערך המודד את מידת אי הודאות (האקראית) של הדגימה. נשתמש ונסביר בהמשך.

# Class REINFORCE_Net



- מימוש רשת הנוירונים לחישוב המדיניות:

```python
class REINFORCE_Network (nn.Module):
    def __init__(self, state_dim, action_dim, lr=0.001, fc_dims=256, chkpt=1, optim_step = 100, optim_gamma = 0.9):
        super().__init__()
        self.linear1 = nn.Linear(state_dim, fc_dims)
        self.Relu = nn.ReLU()
        self.linear2 = nn.Linear(fc_dims, fc_dims)
        self.linear3 = nn.Linear(fc_dims, action_dim)
        self.lr = lr
        self.optimizer = optim.Adam(self.parameters(), lr=lr)
        self.device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
        self.to(self.device)
        self.checkpoint_file = f'Data/REINFORCE{chkpt}.pth'

    def forward(self, state):
        x = self.linear1(state)
        x = self.Relu(x)
        x = self.linear2(x)
        x = self.Relu(x)
        x = self.linear3(x)
        dist = Categorical(logits=x)        # compute the softmax and more
        return dist

    def load_params(self):...

    def save_params(self):...
```

# בניית הזיכרון Memory

```python
class Memory:
    '''...
    def __init__(self, device):
        self.states = []
        self.actions = []
        self.rewards = []
        self.divice = device

    def store_memory (self, state, action, reward):
        self.states.append(state)
        self.actions.append(action)
        self.rewards.append(reward)

    def clear_memory (self):
        self.states = []
        self.actions = []
        self.rewards = []

    def get_tensors(self):
        states_tensor = torch.stack(self.states).to(self.divice)
        actions_tensor = torch.stack(self.actions).to(self.divice)
        rewards_tensor = torch.stack(self.rewards).to(self.divice)
        self.clear_memory()
        return states_tensor, actions_tensor, rewards_tensor
```

- אימון סוכן REINFORCE MC מחייב אותנו לדגום את הסביבות עד לסוף משחק (אפיסודה) ולשמור את: state, action, reward של כל צעד.
- המחלקה Memory תקל עלינו לטפל בשמירת הדגימות.
- הפעולה store_memory מקבלת tensors בלבד.

# המחלקה REINFORCE_Agent



```python
class REINFORCE_Agent:
    def __init__(self, chkpt, player = 1, state_dim = 9, action_dim=9  ):
        self.policy = REINFORCE_Network(chkpt=chkpt, state_dim=state_dim,
        self.player = player
        self.memory = Memory(self.policy.device)
        self.gamma = 0.95
        self.sum_loss = 0  # for logging
        self.sum_entropy = 0

    def remember (self, state, action, reward):···

    def save_model(self):···

    def load_model(self):···

    def get_action(self, state, events = None, train = None):···

    def get_masked_dist (self, states_tensor):···

    def to_action(self, action_index):···

    def mask_illegal (self, states_tensor, logits):···

    def compute_G (self, rewards):···

    def learn(self):···

    def __call__(self, *args, **kwds):···
```

המחלקה Agent היא הסוכן החכם שלנו, הכוללת:

- Policy – רשת הנוירונים למציאת הפעולה הבאה.
- memory – מקום לשמירת הדגימות במהלך הלימוד.
- get_action() – פעולה המקבלת state ומחזירה את ה action הנבחר, בעזרת ה poilcy.
- learn() – פעולה המבצעת את עדכון רשת הנוירונים במהלך הלימוד

# הפעולה get_action – ללא מיסוך



- בחירת הפעולה על ידי הסוכן נעשית באמצעות העברת state ל- policy וקבלת התפלגות של הפעולות (dist).
- בחירה (sample) מתוך ה dist בהתאם להתפלגות (פעולה בעלת התפלגות גבוהה הסיכוי שתבחר תהיה גדולה יותר).

```
def get_action_no_mask(self, state, events = None, train = None):
    state_tensor = state.toTensor().to(self.policy.device)
    with torch.no_grad():
        dist = self.policy(state_tensor)

    action = dist.sample().item()
    row, col = self.to_action(action)
    return (row, col), action
```

- בשלב זה עליכם לשחק עם הסוכן באמצעות רשת לא מאומנת ולבדוק את תפקודו.

# מיסוך פעולות לא חוקיות



- רשת הנוירונים מוציאה לנו התפלגות של כל הפעולות האפשריות, לרבות פעולות שהן לא חוקיות. לעיתים נרצה לבחור רק פעולות חוקיות. לפעולה זו קוראים מיסוך masking.
- מיסוך פעולות אי חוקיות יעשה באמצעות הוספת inf– לערכי הפעולות האי חוקיות (logits), כך שההסתברות לפעולות אילו תהיה 0 בעת הפעלת ה softmax.
- המיסוך נעשה בשני שלבים:
  - יצירת טנסור של הפעולות ל state: פעולה אי חוקית בערך inf–. פעולה חוקית בערך 0. הלוגיקה לפעולה אי חוקית תלויה במשחק שלנו.
  - עדכון ערכי ה logits : mask_logits = logits + mask.
  - בניית אובייקט התפלגות dist חדש.

# get_action – כולל מיסוך



```python
def mask_illegal (self, states_tensor, logits):
    mask = torch.zeros_like(logits, dtype=torch.float)
    mask[states_tensor != 0] = -torch.inf
    return mask
```

```python
def forward(self, state):
    x = self.linear1(state)
    x = self.Relu(x)
    x = self.linear2(x)
    x = self.Relu(x)
    x = self.linear3(x)
    # dist = Categorical(logits=x)
    return x
```

```python
def get_action(self, state, events = None, train = None):
    state_tensor = state.toTensor().to(self.policy.device)
    with torch.no_grad():
        logits = self.policy(state_tensor)

    mask = self.mask_illegal(state_tensor, logits)
    masked_logits = logits + mask
    dist = Categorical(logits=masked_logits)
    action = dist.sample().item()
    row, col = self.to_action(action)
    return (row, col), action
```

```python
def to_action(self, action_index):
    row = action_index // 3
    col = action_index % 3
    return row, col
```

- במשחק שלנו פעולה חוקית היא בחירה במשבצת ריקה.
- נשנה את הפעולה forward ברשת הנוירונים כך שתחזיר לנו logits ולא אובייקט התפלגות.
- בפעולה get_action נוסיף את המיסוך ל logits ונבנה אובייקט התפלגות חדש, לחישוב softmax עדכני.
- to_action: הפעולה המתקבלת מהרשת היא אינדקס אותו עלינו להמיר ל row, col במשחק שלנו.

# get_masked_dist



- הפונקציה get_action מחזירה פעולה בהתאם לרשת הנוירונים מבלי לחשב גרדיאנטים (with torch.no_grad()).
- לצורכי אימון הרשת אנחנו נבנה פעולה המחשבת התפלגות הפעולות עם מיסוך ונכללת בגרף החישוב.

```
def get_masked_dist (self, states_tensor):
    logits = self.policy(states_tensor)
    mask = self.mask_illegal(states_tensor, logits)
    masked_logits = logits + mask
    dist = Categorical(logits=masked_logits)
    return dist
```

- הפעולה get_action תשמש אותנו למשחק ולשלב הדגימה של הסביבה.
- הפעולה get_masked_dist תשמש אותנו לאימון המודל.

# הפונקציה remember



- הפונקציה remember של הסוכן מקבלת state, action, reward, ממירה אותם לטנסורים ושומרת באובייקט memory.

```
def remember (self, state, action, reward):
    self.memory.store_memory(state.toTensor(), torch.tensor(action), torch.tensor(reward))
```

# Class Trainer

- המחלקה Trainer מבצעת אימון של הסוכן באמצעות למידת חיזוק.
- בנוסף מטפלת המחלקה בשמירת נתוני האימון באפליקציה wandb, עליה הסברנו בהרצאות הקודמות (בנושא DDQN).
- הואיל ומתאמנים מול יריב – אחד המאפיינים של המחלקה הוא היריב.

```python
class REINFORCE_Trainer:
    def __init__(self,  chkpt):
        self.agent:REINFORCE_Agent = REINFORCE_Agent(chkpt=chkpt)
        self.env:TicTacToe = TicTacToe()
        self.opponent = Random_Agent(player=-1, env = self.env)
        # self.opponent = Random_Agent_Advanced(player=-1, env = self.env)
        self.chkpt = chkpt

    def train(self, epochs=100000, gamma = 0.99, lr=0.001):...

    def init_wandb (self, project_name, chkpt, resume=False):...

    def log_wandb(self, **kwargs):
        wandb.log(kwargs)
```

# הפעולה train

```python
def train(self, epochs=100000, gamma = 0.99, lr=0.001):
    self.init_wandb(project_name="REINFORCE",chkpt=self.chkpt,resume=False)
    score = 0
    for epoch in range(epochs):
        print(epoch, end="\r")
        state = self.env.reset()

        ########### sample environment ###########
        while not self.env.end_of_game(state):
            action, action_index = self.agent.get_action(state)
            after_state, reward1 = self.env.next_state(state, action)
            if self.env.end_of_game(after_state):
                self.agent.remember(state, action_index, reward1)
                break
            action2 = self.opponent.get_action(state=after_state)
            next_state, reward2 = self.env.next_state(after_state, action2)
            #reward2 += reward1
            self.agent.remember(state, action_index, reward2)
            state = next_state

        ########### train ###########
        self.agent.learn()

        ########### log ###########
        score += reward1+reward2
        if epoch !=0 and epoch % 100 == 0:
            self.log_wandb(score=score, loss = self.agent.sum_loss/100,
                           entropy = self.agent.sum_entropy/100)
            print (f'chkpt: {self.chkpt} epoch: {epoch} score: {score}
                   loss: {self.agent.sum_loss/100:.4e}
                   entropy{self.agent.sum_entropy/100:.4e}')
            score = 0
            self.agent.sum_loss = 0
            self.agent.sum_entropy = 0

    self.agent.save_model()
```

- האימון כולל שתי לולאות:
  - לולאה חיצונית –מספר משחקים קבוע מראש (epochs).
  - לולאה פנימית - לכל משחק אנחנו מבצעים לולאה עד לסוף המשחק– Monte Carlo.
  - בכל צעד שומרים state, action, reward ב- memory.
  - כיוון שהאימון הוא נגד יריב אנחנו מבצעים גם את מהליכי היריב.
  - בסיום כל משחק אנחנו מבצעים את האימון של ה policy, רשת הנוירונים, בהתבסס על הנתונים בזיכרון. האימון נעשה על ידי הסוכן בפונקציה agent.learn().

# אימון רשת הנוירונים

$$loss = -G_t * \ln \pi_\theta(a|s)$$
$$\theta = \theta - \alpha * \nabla loss(\theta)$$

```python
def learn(self):

    states, actions, rewards = self.memory.get_tensors()

    G = self.compute_G(rewards)

    dist = self.get_masked_dist(states)
    log_probs = dist.log_prob(actions)

    # Compute the loss using vectorized operations
    loss = -(G * log_probs).sum()

    # backward
    self.policy.optimizer.zero_grad()
    loss.backward()

    self.policy.optimizer.step()

    # logging
    self.sum_loss += loss.detach()
    self.sum_entropy += dist.entropy().detach().mean()
```

באימון נעשה עדכון אחד של רשת הנוירונים בהתבסס על כל המהלכים במשחק אחד.

- שולפים את states, actions, rewards כמספר המהלכים במשחק שבוצע.
- מחשבים את G – סכום התגמולים עד לסיום המשחק, עבור כל צעד.
- מחשבים log של ההסתברות של הפעולה log_prob, עבור כל צעד.
- מחשבים את ה loss בהתאם לאלגוריתם. מכפילה במינוס כיוון שאנחנו משתמשים ב SGD (המוצא מינימום ולא מקסימום).
- סוכמים את ההפסד כדי לקבל טעות אחת לכל המשחק.

# חישוב סכום התגמולים - G



- $G_t$ - טנסור הכולל את חישוב סכום התגמולים עבור כל צעד $t$ עד לסיום המשחק.

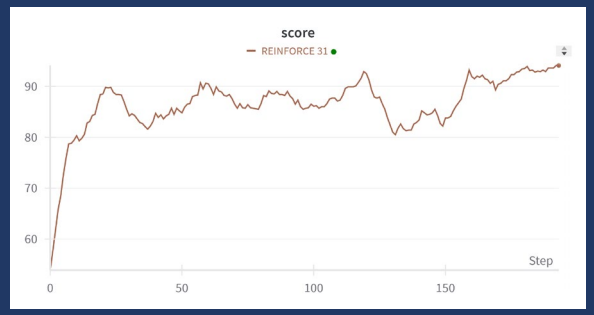
$$G_t = R_0 + \gamma R_1 + \gamma^2 R_2 + \gamma^3 R_3 + \cdots + \gamma^n R_n$$

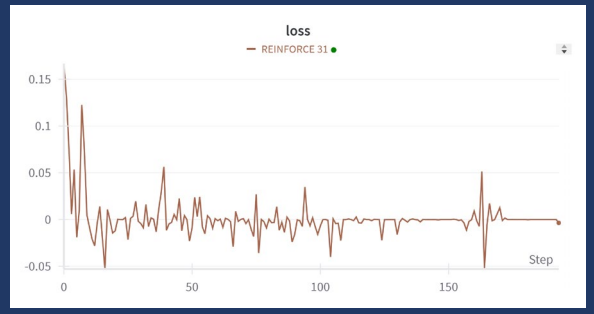
- החישוב נעשה מהסוף להתחלה, לפי הנוסחה הבאה:

$$G_t = R_t$$
$$G_{t-1} = R_{t-1} + \gamma\, G_t$$
$$G_{t-2} = R_{t-2} + \gamma\, G_{t-1}$$

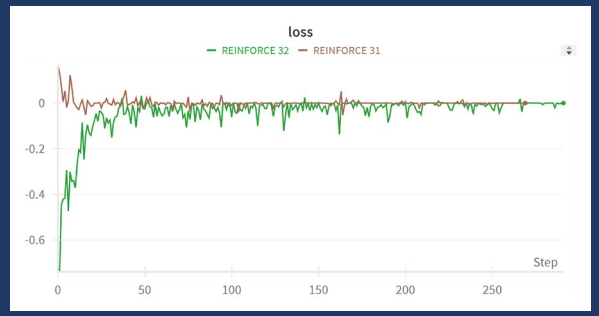
```
def compute_G (self, rewards):
    G_returns = torch.zeros_like(rewards)
    G = 0
    for i in range(rewards.size(0)-1, -1, -1):   # in reverse
        G = rewards[i] + self.gamma * G
        G_returns[i]= G

    return G_returns
```

# אימון הסוכן




- אימון הסוכן כנגד סוכן רנדומלי. חישוב לכל 100 משחקים (1 ניצחון, 0 תיקו, 1- הפסד):
  - עקומת לימוד מהירה ויפה.
  - התחיל עם ניקוד של 50 וסיים עם למעלה מ- 90.


- אימון הסוכן כנגד סוכן רנדומלי משופר
  - הסוכן חושב צעד אחד קדימה ובוחר צעד מנצח אם קיים או חסימה של היריב לסיים שלישייה, אחרת בוחר משחק רנדומלי.
  - עקומת לימוד יפה. הסוכן התחיל עם 50-(רוב הזמן הפסיד) וסיים על 30+.
  - עדיין הסוכן היריב שיחק טוב יותר. האלגוריתם מוגבל.