למידה מונחית חיזוקים - Reinforcement Learning

סמסטר קיץ תשפ"א
ד"ר עמוס עזריה (שימוש במצגות הקורס שלו)
סיכום של הילה שושן

שיעור 1 - 19.7

מצגת 1 - Introduction

למידה מונחית חיזוקים (עמוקה, בשנים האחרונות) היא תחום חם מאוד, ואפשר לעשות איתה דברים מגניבים. דוגמאות: הטסות של הליקופטר, רכבים אוטונומיים, ניצחון במשחקים (גם מאוד מסובכים כמו GO), בניית ארכיטקטורת ה-TPU's של גוגל - מעין בינה מלאכותית שבונה את עצמה.

<u>איך Reinforcement Learning עובד?</u>

יש סוכן שיוצר אינטראקציה עם הסביבה, עושה פעולות ומקבל עליהן תגמול. עם הזמן הסוכן צריך ללמוד לקבל כמה שיותר תגמולים חיובים וכמה שפחות תגמולים שליליים. כל פעם שהוא עושה משהו טוב - הוא מקבל "תמורה" (reward) חיובית, ואז הוא יודע שזה טוב לעשות את זה, וכל פעם שהוא עושה משהו לא טוב - מקבל reward שלילי ואז הוא מבין שלא טוב לעשות זאת.

ה-reward יכול להיות תלוי גם ב-action וגם ב-state (במצב העולם) שבו נמצאים, או רק ב-state.
מה שחשוב לשים לב שה-reward לא יכול להיות תלוי רק ב-action! כי אז זו כבר לא הייתה בעיה של RL - (רק נצטרך לדעת מה הפעולה הכי טובה לבצע ללא תלות במשימה), אלא הוא חייב להיות תלוי ב-state.
כלומר, לא תמיד צריך לעשות פעולה מסוימת, זה תלוי במצב העולם, והסוכן צריך ללמוד את זה.

דוגמה: מבוך 4X4.

S			0 (end)
			+10 (end)

הפעולות: תזוזה לכל אחד מהמשבצות הסמוכות (לא אלכסון).
תמיד מתחילים במצב S, ומצב העולם כולו לא משתנה אף פעם.
ישנם שני מצבים סופיים.
כל פעם שהרובוט מבצע תזוזה, יורדת לו נקודה.
נשים לב שעדיף להגיע ל-0 במקום להיתקע בלולאה אינסופית.
ישנו גם מצב שחור שאסור לעבור דרכו.
המטרה: למצוא את ה-policy (מדיניות) שתביא לנו את ה-reward הגבוה ביותר.

- בהקשר למה שאמרנו קודם: לא בכל פעם שעושים פעולה מסוימת, נניח למעלה, מקבלים reward חיובי, אלא זה תלוי היכן נמצאים.

ההבדל המהותי בין למידה מונחית חיזוקים לבין למידת מכונה סטנדרטית:

- נשים לב ש-RL היא לא כמו Supervised Learning (בלמידת מכונה רגילה). למרות שמקבלים reward כך שבסופו של דבר יודעים האם אנו פועלים טוב או לא.
- ב-ML יש label שאומר עבור כל מצב מה הדבר הנכון לעשות, רוצים לדעת לנבא. כאן אין.
- ישנם מקרים שכן אפשר להתייחס לבעיה של RL כבעיה של Supervised Learning, נראה בהמשך.
- בלמידת מכונה ה-data צריכה להיות ב"ת - כלומר הדוגמאות לא תלויות אחת בשנייה, מה שלא קורה ב-RL, שם כל פעם שנעשה פעולה כנראה נשאר קרובים לנקודה הקודמת שהייתה לנו.
- ב-RL, הסוכן מקבל את ה-reward signal, (מתי הרוויח ומתי הפסיד) שזה משהו שלא היה ב-ML.
- RL מבוסס על trial error - עושים פעולה וכל פעם מצליחים או נכשלים.
- ה-reward מתקבל בעתיד, ולא מיד לאחר ביצוע הפעולה. לא ידוע לנו ישר האם זה היה נכון או לא נכון, אלא בעתיד צריך לזהות מה עשינו שגרם למשהו לא בסדר.

Reward Oriented Learning

הרעיון הוא שלומדים על מנת למקסם את התגמול שאנו מקבלים.
בד"כ מקבלים הרבה תגמולים תוך כדי, ואז קל יותר ללמוד.

- הערה: אם בשחמט היה סוכן שמתחיל לשחק רנדומלית והיה מקבל תגמול רק בסוף המשחק (האם הוא ניצח או הפסיד), הוא לעולם לא היה מנצח ולעולם לא היה יכול ללמוד. לכן במשחקים כאלו בד"כ משתמשים בעוד טכניקות מעבר ל-RL. אחת מהן היא ששני סוכנים, ששניהם בהתחלה לא יודעים לשחק, משחקים זה מול זה ולומדים יחד (דומה ל-GAN).

sacrifice = מפסידים עכשיו כדי לנצח אחר כך. הרעיון הזה קיים רק ב-RL, ב-ML אין שום משמעות לכך.

הרבה פעמים כאשר אנו רוצים למקסם כמה דברים (לדוגמה: ברכב אוטונומי רוצים גם להגיע בבטחה, גם לנסוע על המסלול הקצר ביותר, גם לא לקבל קנסות, לנסוע ע"פ החוקים...).
מכיוון שאפשר למקסם רק דבר אחד, נצטרך לשים את כל התגמולים בסקאלה אחת - של נקודות.
בד"כ אנו מגדירים בעצמנו את ה-reward function. הרבה פעמים אנו יודעים אותו, אבל לא תמיד משתמשים בו, ולעיתים די רחוקות זה נובע מהסביבה. נראה כמה שיטות של RL וכיצד זה מתבטא.
גם את ה-reward signal אנו בד"כ מגדירים בעצמנו.

r = reward, r_t = reward at time t
a (or a_t) = action
o (or o_t) = observation (לאחר ביצוע פעולה **רואים** את מצב העולם כך)
s (or s_t) = world state (זהו מצב העולם **האמיתי**, בד"כ לא ידוע במלואו)

הערות:
- נרצה שסכום ה-rewards יהיה מקסימלי לאורך כל הדרך.
- במשחקים בד"כ נקבל רוב הזמן reward של 0 (חוץ מהסוף - ניצחון או הפסד, 1 או מינוס 1 בהתאמה).
- בסימולציה יותר סביר שנדע מה מצב העולם מאשר בעולם האמיתי - שם נצטרך להסתפק במה שאנו רואים מהחיישנים בלבד.
- לכן נרצה שהמידע שסוכן מקבל בסימולציה יהיה כמה שיותר קרוב למידע במציאות.

- לאורך הקורס נשתמש בד"כ בסימון של s, אך נדע שזה לא באמת מצב העולם האמיתי, אלא רק מה שרואים.
- לאחר ביצוע פעולה, נקבל reward מסוים ונראה מצב עולם מסוים.

history = מסומן ב-h_t, והוא המסלול שעשינו עד עכשיו, כולל הפעולות וה-observations(a_i, o_i) עד זמן t.

$h_{x,y}$ = כל הפעולות וה-observations שהיו לנו מזמן x עד זמן y.

באופן דומה: $s_{x,y}$ = כל ה-states שהיו לנו

$a_{x,y}$ = כל הפעולות שהיו לנו

<u>אבחנה בין States לבין Observations:</u>

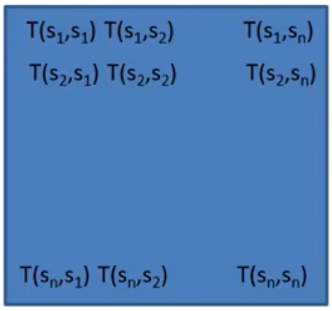
State הוא ממש מצב העולם כולו - כל מה שמשפיע ולו במעט על התוצאה. יכול להיות שנצטרך לדעת גם איפה ממוקמת השמש, האם יפלו אסטרואידים בעתיד… כמעט כל היקום, כי יש לכך השפעה כלשהי.
מכיוון שזה כמובן לא פיזיבילי, נעבוד עם מה שאנו יכולים לדעת באמת.

התנאי (הלא אמיתי) שנבקש אותו: $P(s_{t+1}, |s_t, a_t) = P(s_{t+1}|s_{0:t}, a_{0:t}, o_{0:t})$
כלומר: אם יודעים באיזה state ובאיזה action נמצאים, ניתן לקבוע את ההסתברות של המצב הבא בו נהיה, שהיא בפועל תלויה גם בכל הפעולות, המצבים וה-observations שהיו עד עכשיו, אבל זו דרישה שאי אפשר לעבוד איתה.

הדרישה מה-state היא שהוא יסכם לנו את כל ההיסטוריה עד הרמה שאנו צריכים. נניח שמה שאנו רואים זה מספיק כדי לדעת מה ה-state שאנו נמצאים בו.
יש מצב שבו רואים את ה-observations אבל הם לא מאפשרים לנו להבין מה ה-state (לא כ"כ נדבר על זה).

כעת נבנה את הבעיה הכללית של ה-MDP בצורה מדורגת.

<u>Markov Process - Chain</u>



יש רק מצבים (S) ומעברים בין המצבים (T - transition function). אין סוכן, actions או rewards. ישנם מצבים סופיים. מזכיר אוטומט.
אין משמעות ל-states עדיפים יותר או פחות, אלא רק תיאור של התקדמות הדברים.

התכונה המרקובית קיימת גם כאן: ה-state הבא שנגיע אליו תלוי רק ב-state הנוכחי, ולא ב-states קודמים.

ה-transition function (בציור) היא מטריצה T שאומרת בכל תא $T(s_i, s_j)$ מה הסיכוי שנעבור מ-s_i ל-s_j.

- השורות סוכמות ל-1.
- העמודות לא סוכמות למשהו מוגדר, אבל אם יש לעמודה j סכום גבוה מאוד, זה אומר שיש סיכוי גבוה שנעבור למצב s_j מהרבה states.

- המעברים לא תלויים בהיסטוריה - המעבר ל-state הבא תלוי רק ב-state הנוכחי (כאן אפילו לא ב-action).
- כשיש הרבה מאוד states לא נוכל להשתמש במטריצה הזו.

<u>Markov Reward Process - MRP</u>

כמו Markov Process רק שלכל אחד מה-states יש גם reward. (עדיין אין פעולות - אי אפשר לבחור מה עושים).

ה-reward function מסומנת ב-R.

יש כאן גם discount factor, המסומן ב-γ. הוא מועלה בחזקה בגודל מספר הצעדים שהתקדמנו מההתחלה ומוכפל ב-reward המתקבל בכל פעם. המשמעות שלו היא שדברים שנקבל עכשיו טובים יותר מדברים שנקבל בעתיד.
ה-discount factor הוא ערך בין 0 ל-1 (וקטן ממש מ-1). בד"כ קרוב מאוד ל-1, למשל 0.999.

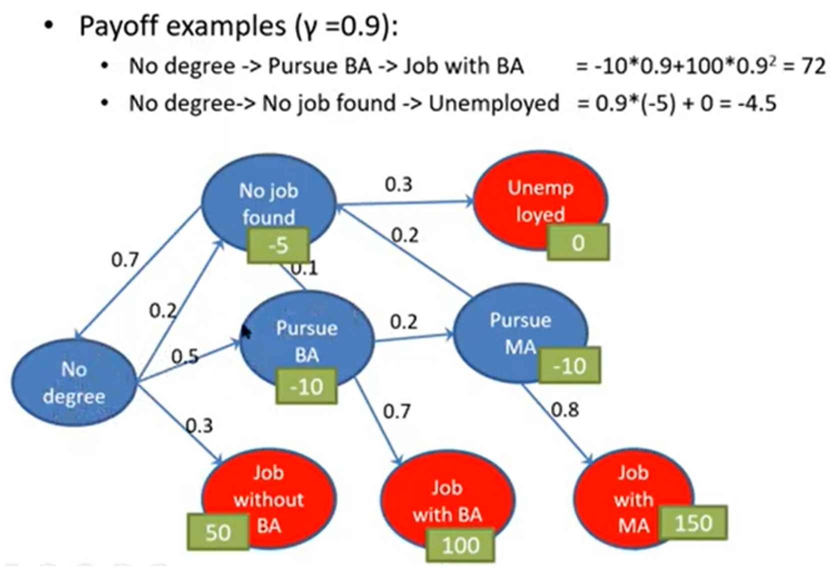
המוטיבציה עבור ה-discount factor:
- להגיע כמה שיותר מהר לסוף.
- אם נקבל סכום כסף עכשיו, ניתן לשים אותו בבנק ובעתיד נרוויח יותר כי נקבל ריבית.
- הסיכוי שבאמת נקבל משהו עכשיו גדול יותר מהסיכוי שנקבל משהו בעתיד, כי לא יודעים מה יקרה בדרך ומה עלול להשתנות.
- ה-discount factor עוזר לנו לפתור את הבעיה - גורם להתכנסות הטור.
  לדוגמה, אם היינו מקבלים בכל תור 10, אז הטור $\sum_{i=0}^{inf} \gamma^{i} 10$ כידוע מתכנס, וללא ה-γ מתבדר.
- אנשים (וגם חיות) מעדיפים לקבל תגמול עכשיו מאשר בעתיד. למרות שאנשים פועלים יותר לפי hyperbolic discounting (לא חשוב לקורס).

ישנן אלטרנטיבות ל-discount factor:
1. finite horizon - מצב בו מסתכלים רק t צעדים קדימה, ואז זה גם כן תמיד מתכנס.
2. average reward - הממוצע שמקבלים במשך t צעדים, גם כן לא מתבדר כי זה סכום סופי.

לסיכום, $MRP =< S, T, R, \gamma >$

total payoff

ניתן לחשב מה ה-reward המתקבל במסלול מסוים, כך: ה-reward של כל מצב כפול γ בחזקה המתאימה.
לדוגמה:



נרצה לדעת עבור כל אחד מה-states עד כמה זה טוב להיות בו, כלומר תוחלת הרווח עד סוף המשחק.
נסמן ב-$V(S)$ את ה-expected total payoff כשמתחילים ב-S.
השאלה: אם מתחילים במצב מסוים, מה יהיה הצפי שלנו ל-total payoff עד סוף המשחק?

- עבור termination states זה קל - פשוט הערך של ה-reward במצב הזה. (כי אנחנו במצב בו ה-reward מוגדר עבור state בלבד, בלי action).
- עבור states שאינם סופיים נראה בהמשך.

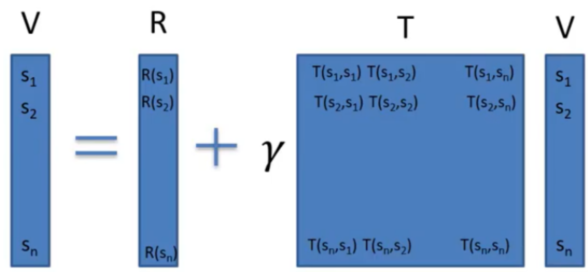
ה-value function של ה-MRP:
Bellman Equation: דרך לבטא את ה-expected total payoff באמצעות כל ה-states שקדמו לו, או הפוך - מ-state מסוים באמצעות ה-state הבא שאליו נעבור.
הנוסחה (הגדרה רקורסיבית):

$$V(s_i) = R(s_i) + \gamma \sum_{s_j \in S} T(s_i, s_j)\, V(s_j)$$

(כרגע מניחים שכל ההסתברויות נתונות).

הפתרון הישיר:

$$V = R + \gamma\, T\, V$$



$$V - \gamma\, T\, V = R$$
$$(I - \gamma\, T)V = R$$
$$V = (I - \gamma\, T)^{-1} R$$

אנו נשתמש במקום בפתרון איטרטיבי:
נאתחל את כל ה-Vים ל-0, ואז בצורה איטרטיבית נעדכן את כל ה-states לפי ה-Bellman eq.

- הסיבה שנעדיף פתרונות איטרטיביים היא שהם גמישים יותר ויש אפשרות להפעיל אותם בכל מיני מצבים.
- כמו כן, אם יש מטריצה מאוד גדולה יכול להיות שיהיה קשה לחשב את ההופכית שלה.

שאלה: למה זה נכון להשתמש בערך שלא יודעים אותו כדי לעדכן ערך אחר שגם אותו לא יודעים?
נשים לב שהמשוואה נכונה קדימה (כלומר אם היינו מבודדים את V_j ומחשבים אותו לפי הנוסחה, זה כנראה לא היה נכון). הכיוון הזה הגיוני מפני שככל שנתקרב לסוף זה יהיה יותר מדויק. כלומר, מעדכנים על סמך ערכים שיהיו בהמשך הזמן וזה הגיוני כי הם מדויקים יותר. בכיוון ההפוך זה היה מתבדר לערכים לא נכונים.

נשים לב שבכל עדכון מנצלים את האינפורמציה החדשה שקיבלנו, ה-R.

- גם אם אין לנו מצבי סיום זה הגיוני, כי ככל שהמצב מרוחק יותר בעתיד פחות אכפת לנו ממנו. לכן ניתן להגביל את מספר הצעדים שנרצה להסתכל עליהם למשל.

הוכחת התכנסות של הפתרון האיטרטיבי:

Bellman Operator

- $F(v(s_i)) = R(s_i) + \gamma \sum_{s_j \in S} T(s_i, s_j) v(s_i)$
- $F(v) = R + \gamma\, Tv$

מקבל את ה-V של איזשהו state, ומעריך את ה-state על סמך כל ה-states הבאים (מפעיל את ה-Bellman Equation).

<u>Contraction Mapping Theorem</u>

הגדרה: $\gamma - contraction$

אופרטור בתוך מרחב וקטורי המקיים:

$\|F(x) - F(y)\| \le \gamma \|x - y\|$, for all x and y,
where $0 < \gamma < 1$

(כאשר מפעילים את האופרטור על שני וקטורים, המרחק ביניהם קטן מזה שהיה בין הוקטורים המקוריים, עם פקטור של γ, או שווה לו).

יהיו v, u שני וקטורים (המבטאים value function מסוים).
נשתמש בנורמת אינסוף, שזה בעצם המרחק הכי גדול שיש בין 2 קורדינטות של 2 וקטורים.
נרצה להראות שהמרחק בין שני וקטורים שנעבור ביניהם קטן.

- $\|F(v) - F(u)\|_\infty = \|R + \gamma\, Tv - R + \gamma\, Tu\|_\infty$
- $= \|\gamma T\,(v - u)\|_\infty$
- $\le \gamma \|v - u\|_\infty$

T is a stochastic matrix, i.e. all rows sum to 1. So the maximum is obtained when T gives all its weight (i.e. 1) to where v is most different from u.

(בכך שהתעלמנו מ-T רק הקטנו את הביטוי, כי T רק יכולה להגדיל אותו).

<u>Markov Decision Process - MDP</u>

כאן מתווסף A - החלטות שהסוכן יכול להחליט באיזה action לבחור.
ה-transition function תלוי גם ב-action עכשיו. אם בחרנו לזוז ימינה, יש סיכוי מסוים שנזוז באמת ימינה (זה לא בוודאות, תלוי בהגדרה. יכול להיות שיש "החלקות").

המטריצה כעת היא תלת מימדית. כל תא $T(s_i, a, s_j)$ מכיל את הסיכוי שאם היינו ב-s_i ובחרנו לעשות את action a הגענו ל-s_j. צריך ש-$\sum T(s, a, ?) = 1$.

עדיין יש הנחה מרקובית: ה-state הנוכחי תלוי אך ורק ב-state הקודם וב-action שביצענו. לכן:
$MDP =< S, T, A, R, \gamma >$

- לפעמים נגדיר את ה-reward function שיהיה תלוי גם ב-γ. זה שקול.

Policy: מסומן ב-π. אומר לנו בהינתן state, איזה action נעשה. יכול להיות דטרמיניסטי, למשל $\pi(s_5) = a_2$ (תמיד כשנהיה במצב s_5 נבצע את הפעולה a_2), או סטוכסטי, למשל $\pi(s_5, a_2) = 0.3$.

- בדוגמאות שראינו עכשיו אין משמעות ל-policy סטוכסטי, לכל פתרון שממקסם את תוחלת הרווח, קיים גם פתרון דטרמיניסטי שממקסם את הערך.

בהינתן policy נרצה לדעת עד כמה היא טובה. בסופו של דבר נרצה למצוא את ה-policy האופטימלית.

<u>הגדרת RL</u>

בהינתן MDP נרצה למצוא את ה-policy שתמקסם לנו את תוחלת הרווח. כלומר, סכום ה-discounted של כל ה-rewards שאנו מקבלים.

- כרגע מניחים שהכל נתון לנו, אבל זה לא תמיד נכון. לפעמים ה-T וה-reward לא נתונים לנו.

בד"כ ב-RL מסתכלים על סוכנים אחרים כחלק מהסביבה. כלומר, אם רוצים לדעת מה מצב העולם לאחר שנעשה פעולה מסוימת, אז אם נסתכל רק על עצמנו זה ברור (למשל בשחמט, בחרנו לזוז ימינה אז זה זז ימינה), אבל אם נתחשב גם בפעולה של הסוכנים האחרים (היריב במשחק), אז מצב העולם בפעם הבאה משתנה בהתאם לפעולה שלו.

לפני שנדבר על מציאת ה-policy, נדבר על בעיה פשוטה יותר:
בהינתן policy נרצה לדעת מה תוחלת הרווח (עד סוף המשחק) לכל מצב בו אנו נמצאים. כך הבעיה חוזרת להיות בעיית MRP (אין בחירה ברגע שה-policy נתונה, ה-action מקובע).
בבעיית MRP אנו יודעים איך למצוא את $V(S)$ עבור כל state.

ה-value function כעת תלוי גם ב-policy (בנוסף ל-state, נניח כעת שלא תלוי ב-action).

- Bellman Equation עם policy דטרמיניסטי ו-transition סטוכסטי:

$$V_\pi(s_i) = R(s_i) + \gamma \sum_{s_j \in S} T\big(s_i, \pi(s_i), s_j\big) V_\pi\big(s_j\big)$$

- Bellman Equation עם policy סטוכסטי ו-transition סטוכסטי:

$$V_\pi(s_i) = R(s_i) + \gamma \sum_{a \in A} \pi(s_i, a) \cdot \sum_{s_j \in S} T\big(s_i, a, s_j\big) V_\pi\big(s_j\big)$$

ואם ה-reward תלוי גם ב-action:

$$V_\pi(s_i) = \sum_{a \in A} \pi(s_i, a)\Big(R(s_i, a) + \gamma \sum_{s_j \in S} T\big(s_i, a, s_j\big) V_\pi\big(s_j\big)\Big)$$

- זה שקול ל-reward שתלוי רק ב-state מכיוון שתמיד אפשר להוסיף למצב העולם שדה שאומר כיצד הגענו אליו, ואז ה-reward יהיה פר state.
- אבל אם ניצחנו במשחק בד"כ לא כ"כ מעניין אותנו כיצד ניצחנו אותו, ה-reward יהיה אותו דבר.

Q function

כמו ה-value function, רק שהוא לוקח בחשבון גם את ה-state וגם את ה-action (נניח שה-reward תלוי גם ב-state וגם ב-action).

- policy דטרמיניסטי:

$$q_\pi(s_i,a) = R(s_i,a) + \gamma \sum_{s_j\in S} T(s_i,a,s_j) q(s_j,\pi(s_j))$$

הסבר: ה-reward שמקבלים עכשיו ועוד גמה כפול מה שאנו צפויים לקבל בהמשך (בשביל זה צריך לעבור על כל ה-states האפשריים וההסתברות שנעבור אליהם), ועבור כל state נעשה את ה-action שצריך לעשות לפי ה-policy.

- policy סטוכסטי:

$$V_\pi(s_i) = \sum_{a\in A} \pi(s_i,a)(R(s_i,a) + \gamma \sum_{s_j\in S} T(s_i,a,s_j) V_\pi(s_j))$$

$$q_\pi(s_i,a) = R(s_i,a) + \gamma \sum_{s_j\in S} T(s_i,a,s_j) \sum_{a'\in A} \pi(s_j,a') q(s_j,a')$$

מצגת 2 - Model-Based Reinforcement Learning

<u>בעיית ה-RL Planning:</u>

עד כה רק שערכנו עד כמה policy מסוימת היא טובה, עכשיו רוצים למצוא את הpolicy הכי טובה (לפתור את ה-MDP).

בהינתן $MDP =< S, T, A, R, \gamma >$, נרצה למצוא את ה-policy שממקסמת את תוחלת הרווח (כאשר לוקחים בחשבון את ה-discount factor).

<u>בעיית חישוב ה-reward:</u>

זהה לבעיית ה-Markov-reward process, צריך להכניס את כל הפעולות ל-transmissions.

מחשבים את $V_\pi(s_i)$ (שהוגדר קודם). ניתן לחשב בצורה ישירה, אבל בד"כ נעבוד עם פתרונות רקורסיביים.

<u>אלגוריתם</u> לחישוב $V_\pi(s_i)$ בהינתן policy:

- If s_i is a terminating state:
  - $V_\pi(s_i) = \sum_{a\in A} \pi(s_i,a) R(s_i,a)$
- Start with: V := 0.
- Repeat until convergence:

$$V_\pi(s_i) := \sum_{a\in A} \pi(s_i,a)(R(s_i,a) + \gamma \sum_{s_j\in S} T(s_i,a,s_j) V_\pi(s_j))$$

(ההוכחה שזה מתכנס זהה להוכחת התכנסות ה-MRP).

שאלה: איך אפשר להשתמש בידע הזה כדי לשפר את ה-policy?

תשובה: לכל אחד מה-states, נבחר ב-action שמביא אותנו ל-state עם ה-V המקסימלי.

דרך המטרה: למצוא את V^*, q^*, π^*

π^* הוא ה-policy האופטימלי (לכל state, איזה action ימקסם לנו את תוחלת הרווח).

V^* אומר בהינתן כל אחד מה-states מה ה-payoff תחת ההנחה של ה- π^*

q^* אומר בהינתן זוג של state ו-action (לאו דווקא אופטימלי), מה ה-payoff בהנחה שמעתה והלאה נפעל לפי ה-policy האופטימלי.

פורמלית:

$$V^*(s_i) = max_\pi V_\pi(s_i)$$

- אי אפשר לחשב ישירות את הערך הזה, כי יש המון policies, אפילו דטרמיניסטיים בלבד. סדר גודל של $|A|^{|S|}$.

$$q^*(s_i, a) = max_\pi q_\pi(s_i, a)$$

$$\pi^*(s_i, a) = argmax_\pi V_\pi(s_i)(s_i, a) = argmax_\pi q_{\pi^*}(s_i, a)(s_i, a)$$

בגלל בעיית החישוב, נשתמש ב-**Policy Iteration**.
זה האלגוריתם הראשון שפותר בעיית RL עם MDP (בהנחה שהכל נתון לנו).

- Start with any policy π
- While π changes:
  - Evaluate V_π with the new π
  - Update π greedily, according to V_π
- That is, repeat until convergence:
  - Repeat until convergence:
    - $V_\pi(s_i) := R(s_i) + \gamma \sum_{s_j \in S} T(s_i, a, s_j)\, v(s_j)$
    - Or $V_\pi := R + \gamma T_\pi V_\pi$
  - Update:
    - $\pi'(s_i) := argmax_{a \in A}(R(s_i) + \gamma \sum_{s_j} T(s_i, a, s_j)\, v_\pi(s_j))$

Or $\pi'(s_i) := argmax_a(\sum_{s_j} T(s_i, a, s_j)\, v(s_j))$ since $R(s_i)$ and γ are identical

בגדול: מתחילים עם policy כלשהי, משערכים את ה-values לפי ה-policy הזה ומעדכנים את π להיות הכי טוב שאפשר בהנחה שה-values שלנו נכונים.

- הערה: כדי שהתהליך יתכנס, כאשר יש לנו שני actions עם אותו הערך, נקבע איזשהו סדר עליהם כדי להכריע.

שיעור 2 - 26.7

נוכיח את <u>התכנסות ואופטימליות ה-Policy Iteration</u>:

נשים לב שלאחר כל איטרציה, ה-policy דטרמיניסטית.
ראינו ש-V מתכנס לערך הנכון (ב-MRP).

אם האלגוריתם עוד לא הסתיים, אז ה-policy השתנה. נסמן ב-π את ה-policy שהייתה קודם, וב-'π את ה-policy החדשה. נרצה להראות ש-'π יותר טובה מ-π (בצורה מובהקת, גדולה ממש).
כמו כן, מאחר שמספר ה-policies הדטרמיניסטים הוא סופי, וכל הזמן משתפרים ממש, אז בהכרח נגיע למקסימום בסוף התהליך.

ה-policy השתנתה ← קיים k עבורו: $V_{\pi'}(s_k) \neq V_{\pi}(s_k)$.

אבל: $V_{\pi'}(s_k) = r + \gamma max_a T(s_k, a, s_j) V_\pi R > r + \gamma \pi(s_k) T(s_k, a, s_j) V_\pi R = V_\pi(s_k)$
כלומר, בכל איטרציה V משתפר, לפחות בכניסה אחת.

- הפעולה הטובה ביותר בפרט טובה יותר מהפעולה הקודמת שהייתה לנו באותו ה-state, זו $\pi(s_k)$.
- זה גדול ממש ולא גדול/ שווה מכיוון שאמרנו שיש קורדינטה אחת שונה לפחות.

ברגע שבו π נשארת ללא שינוי, $\pi'(s_i, a) = argmax_\pi(s_i)(s_i, a)$.

לכן $\pi' = \pi^*$ כאשר π^* הוא ה-policy האופטימלי, ומכיוון ש-π תמיד דטרמיניסטית, גם π^*.

שיפור: <u>Modify Policy Iteration</u>

הרעיון הוא שבמקום לחכות עד להתכנסות, נריץ k איטרציות ובהן נשערך את ה-policy הנתונה כרגע, ואז נעדכן את ה-policy, כי אפשר לבצע עדכונים בכל פעם.
כלומר, בד"כ לא צריך לשערך הכל עד הסוף, עד שהכל מתעדכן.

- עבור k=1, יש שם מיוחד לשיטה: **Value Iteration**. ההבדל הוא שכאן לא מאתחלים עם policy רנדומלי, אלא לא צריך בכלל policy. הסיבה היא שאין חזרה עד ההתכנסות, אלא כל פעם מבצעים את העדכון:
$$V_\pi(s_i) := R(s_i) + \gamma \sum_{s_j \in S} T(s_i, a, s_j)\, v(s_j)$$
פעם אחת בלבד. לכן לוקחים את ה-action שממקסם את הסכום הזה.
- המטרה שלו היא בעצם למצוא את V^*, ואז נדע גם את π^*, כי תמיד נבחר ב-action שממקסם לנו את ה-value.
- בשיטה זו כאילו עושים גם שערוך של ה-policy וגם עדכון שלה ביחד (כי ה-policy תמיד ממקסמת את ה-values).

- Initialize: $V := 0$
- Repeat until convergence (for every s_i):
  - $V(s_i) := R(s_i) + \gamma \max_a T(s_i, a, s_j)\, V(s_j)$
- There is no implicit policy.
- Finally, we choose the policy:
  - $\pi = argmax_a V$

על מה מסתמכים כשבאים לעדכן?
- אפשר לעדכן בצורה סינכרונית - כל פעם מעדכנים את כולם ע"פ האיטרציה הקודמת.
- או לעדכן "in place" - כל פעם מעדכנים על סמך הערך המעודכן ביותר שיש אצלנו (זה גם יכול לחסוך איטרציות בד"כ, וגם נוח יותר גם מבחינה תכנותית).
- אפשר לקבוע סדר מסוים לעדכון, שיכול להשפיע גם כן על מספר האיטרציות.
- אם נראה שיש states שמתעדכנים הרבה, נוכל לבחור לעדכן אותם לעיתים קרובות יותר.

- הערה: כל אלו משפיעים רק על הזמן עד להתכנסות, כולם מתכנסים ל-policy האופטימלית!

מה קורה כשלא יודעים את הדינמיקה של המערכת (מאיזה state עוברים לאיזה state)?
כלומר מצב בו עושים פעולה אך לא יודעים בוודאות איפה נהיה בצעד הבא, יש "החלקות". לא יודעים את ההסתברויות של ה-transitions, ויכול להיות שיש סוכנים אחרים בסביבה.

- עדיין ניתן להניח שקיימת transition function, שבהינתן מצב ופעולה אומרת מה הסיכוי שכל אחד מה-states הבאים יקרה, אבל בד"כ לא יודעים אותה.

פתרון נאיבי: frequency count
לראות מאילו states עוברים לאילו states בעולם. כך נגלה את הסיכויים.
ואם יש לנו יותר מידי states זה לא פתרון הגיוני.
גם אם יש לנו את ה-transition function אנחנו עדיין בבעיה, כי לא נוכל להחזיר את ה-value function לכל אחד מה-states.

לכן צריך פתרון אחר: Model Free. (משתמשים בו כשאין לנו את ה-transition function או כשיש יותר מידי states).

מצגת 3 - Model-Free Reinforcement Learning

בהינתן $MDP =< S, T, A, R, \gamma >$:
A - בד"כ ידוע (יודעים אילו actions אנחנו יכולים לעשות)
S - גם ידוע, כי הגדרנו את ה-states בעצמנו
γ - ה-discount factor. גם כן אנחנו הגדרנו (בד"כ הוא אפילו לא חלק מהמציאות, אבל חייב אותו)
R - פונקציית ה-reward. אנחנו גם כן מגדירים בד"כ, אבל לפעמים חסרה לנו כי לא יודעים את החוקים מראש, ומגלים אותם תוך כדי המשחק (למשל מה טוב ומה רע וכיצד מוגדר הניקוד - חלק מהסביבה).
T - פונקציית המעברים. הכי בעייתי לדעת אותה כי לא יודעים מה הדינמיקה, איך עובד העולם ...

ב-Model-Free Reinforcement Learning יודעים את A, S, γ אבל את T, R לא צריכים.

הרעיון הכללי הוא פשוט לעשות פעולה ולראות לאן הגענו. ואז תהיה לנו מעין דוגמית של ה-transition function (אבל כמובן לא הפונקציה ממש, כי לא יודעים את כל הסיכויים).
לאחר ששיחקנו כמה פעמים, יהיה לנו את ה-Q-function, ואז נרצה לעשות פעולות שצפויות לתת לנו ערכים גבוהים יותר.

<u>Policy Evaluation</u>

נניח שיש לנו policy, ונרצה לחשב את הערכים עבור כל state, אבל מכיוון שאין לנו את ה-T, זה לא יהיה שקול ל-MRP ואז לא נוכל להשתמש באותו תהליך של value evaluation שהפעלנו קודם.

Monte-Carlo Policy Evaluation

שיטה להערכת ה-values של ה-states. הרעיון הוא להגריל רנדומלית.
ה-policy כן אומרת לנו איזה action לעשות מכל state, לכן נריץ אותה מספר רב של פעמים, עד שזה מסתיים. ואז לכל מסלול שיש לנו, נוכל לחשב כמה טוב היה לנו להיות בכל אחד מה-states, לפי ה-reward שקיבלנו. נשמור זאת ב-G (אלו הערכים עבור משחק מסוים, כלומר מסלול ספציפי).

ה-V אמור להחזיר את ה-expected return, ונחשב אותו בקירוב לפי ממוצע על פני כל ערכי ה-G עבור ה-state הזה.

- הנחות סמויות: עברנו על כל ה-states, יש לנו מספר סביר של states.
- אם עברנו באותו state מספר פעמים במסלול, זה לא בהכרח אומר שהערכים יהיו אותו דבר. לכן אפשר להסתכל רק על הפעם הראשונה שעברנו בו ולהתעלם מכל הפעמים הבאות, או להסתכל על הכל אבל לעשות ממוצע של יותר ערכים (במקום מספר המשחקים, מספר הביקורים).

במקום לשמור את כל ה-Gים ורק בסוף לעשות ממוצע, אפשר:

1. לשמור תוך כדי סכום שאומר כמה היה לנו עד עכשיו, לעשות counter ששומר כמה פעמים ביקרנו בכל אחד מה-states, ולבסוף לחשב את הממוצע. זה נקרא Incremental updates.
2. או לזכור את ה-V בכל פעם שראינו את ה-state, ובכל פעם לעדכן אותו קצת לפי ה-V החדש. כלומר במקום לשמור את הסכום שהיה לנו עד עכשיו, נשמור את הממוצע ונקבל ערך דומה. זה נקרא Incremental Mean:

$$V_k(s_i) = \frac{1}{k}\sum_{j=1}^{k} G_j(s_i) \quad = \frac{1}{k} G_k(s_i) + \frac{k-1}{k} V_{k-1}(s_i)$$
$$= V_{k-1}(s_i) + \frac{1}{k}\left(G_k(s_i) - V_{k-1}(s_i)\right)$$

- כל V מבוסס על ה-V הקודם עם איזשהו שינוי קטן בהתאם למידע החדש שיש לנו.
- אפשר להתייחס ל-$\frac{1}{k}$ כקבוע learning rate, α, ואז:

$$(1-\alpha)V_{k-1}(s_i) + \alpha\, G_k(s_i)$$

זה טוב להתייחס לזה ככה כי כך לא נצטרך לעקוב אחרי ה-kים, וגם דברים חדשים שנראה ישפיעו יותר מדברים ישנים, חוץ מהערך הראשון. בעתיד, נרצה לעדכן את ה-policy ושם הערכים החדשים שנראה יהיו רלוונטים יותר וזה יתרון נוסף.
בעיה באלפא קבועה: יש משמעות לאיפה התחלנו, כלומר אם נאתחל את הכל ל-0 זה עלול להשפיע מאוד על הממוצע. לכן אפשר לתקן את זה באמצעות bias correction term:

$$\widehat{v_k} = \frac{v_k}{1-(1-\alpha)^k}$$

אבל נשים לב שבתיקון קיבלנו חזרה את ה-k, אז בד"כ מתעלמים מה-bias הזה, שבד"כ לא מזיק, ואפילו יכול להיות הגיוני.

Temporal Differences (TD) policy evaluation

שיטה אחרת להערכת ה-values של ה-states. הרעיון הוא להסתכל צעד אחד קדימה (או מספר קטן של צעדים), ולעדכן את ה-value function מיד. זה "ההפך" ממונטה קרלו, שם הרצנו את ה-policy עד הסוף (הרבה פעמים).
יתרונות: לא צריך לחכות עד סוף ה-episode כדי לעדכן, הרבה פעמים זה יתכנס יותר מהר.

- העניין של ה-bias וה-variance: במונטה קרלו רצים כל פעם עד הסוף, לכן ל-samples שלנו יש variance מאוד גדול, אבל אין bias כי לא משנה מאיפה התחלנו.
לעומת זאת, כאן הערך ההתחלתי שלנו משפיע מאוד על ההערכה בהמשך.

השיטה $TD(0)$:

- מאתחלים את $V = 0$
- עבור כל $s \xrightarrow[a]{} s'[r]$:

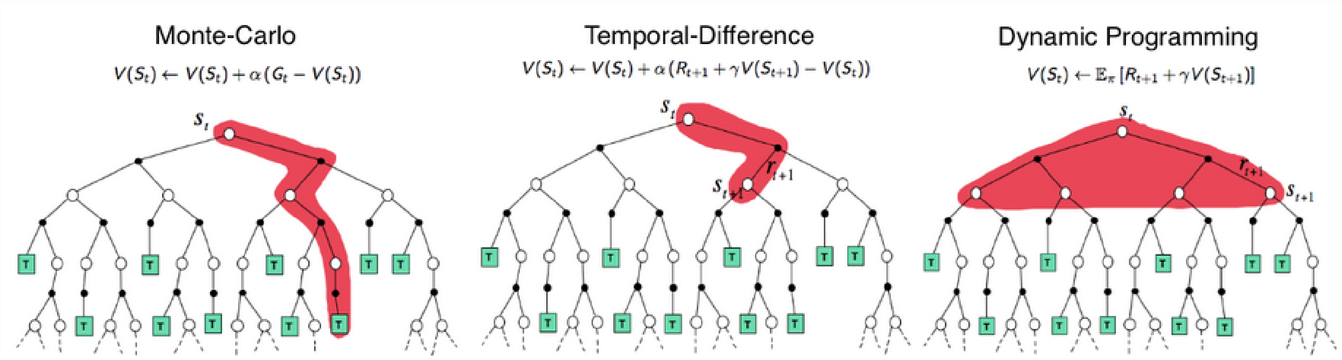
מעדכנים את $v(s)$ לערך $r\ +\ \gamma v(s')$ (ה-TD Target).
כלומר: $v(s) := v(s)\ +\ \alpha(r\ +\ \gamma v(s')\ -\ v(s))$. זה נקרא "לעדכן לקראת r".

הערך $\delta_t = r_{t+1} + \gamma v(s_{t+1}) - v(s_t)$ נקרא $\delta - error$ או TD error (בזמן t). הוא אומר עד כמה טעינו. בנוי מהערך שאנו מעדכנים פחות מה שהיה לנו.

<u>ההבדלים בין מונטה קרלו ל-TD (אפס):</u>

- המונטה קרלו משתמש במסלול שלם, לכן לא משנה מאיפה התחלנו (אלא אם כן משתמשים ב-α),
  אין שום bias, אבל יש variance, כי בכל משחק ייתכן שהתוצאות יהיו שונות לגמרי.
  בנוסף, יכול להיות שעשינו פעולות ממש טובות לאורך כמעט כל המשחק, אבל בסוף הפסדנו, אז הכל יתעדכן לקראת הפסד.
- לעומת זאת, ב-TD, אם עשינו פעולות טובות שהובילו אותנו ל-state טוב, אז זה יתבטא בערך שנעדכן.
- זה נכון גם בכיוון השני, "almost happens". למשל ברכב אוטונומי - אם כמעט הייתה תאונה, וברגע האחרון הצלחנו להימנע ממנה, מבחינת Monte Carlo הערך הסופי יהיה חיובי, ולא נתחשב בזה שכמעט מתנו. (למרות שאם נגיע למצב הזה הרבה פעמים, הגיוני שיש פעמים בהן היו תאונות וכאלו שלא ולכן ה-V של המצב כבר כן יתעדכן לערך הנכון).
- לעומת זאת ב-TD נעדכן את הערך לערך שלילי, וזה גם ישפיע ויעדכן את הערך של כל המצבים הקודמים שהובילו אותנו למצב זה).

<u>השוואה בין Temporal Differences, Monte-Carlo ו-Dynamic Programming (שראינו ב-Model-Based):</u>



העדכון המדויק ביותר הוא ב-DP, אבל שם מניחים שאנו יודעים את ה-transition function.
גם העדכון במונטה קרלו יחסית מדויק, כי מחכים עד הסוף על מנת לעדכן.
ב-TD העדכון פחות מדויק, אבל הוא הרבה יותר זול (כי לא צריך להסתכל על כל ה-states).

אם ב-TD נסתכל על יותר מצעד אחד קדימה, כלומר זה כבר לא יהיה $TD(0)$, אז מצד אחד זה יקטין לנו את ה-bias כי מתבססים על פידבק אמיתי יותר, אבל מצד שני זה יגדיל לנו את ה-variance כי זה תלוי במה עשינו במסלול הנוכחי שאנו נמצאים בו.

$TD(\lambda)$:
משהו בין $TD(0)$ לבין מונטה קרלו, כך שנותנים משקל לכל אחד מהמצבים.
פרמטר λ הוא המשקל שרוצים לתת לצעד הבא, כלומר:

- הערך של הצעד הבא יוכפל ב-$(1-\lambda)$
- הצעד אחריו ב-$(1-\lambda)\lambda$
- הצעד הבא ה-i יוכפל ב-$(1-\lambda)\lambda^{i}$
- בהנחה שהצעד האחרון יקרה בעוד T צעדים, ניתן לו משקל של λ^{T-1}

נשים לב שככל הצעד רחוק יותר, הוא פחות חשוב (בפקטור של λ), ושאם נסכום את הכל נקבל 1:

סכום סדרה הנדסית: $\sum_{i=0}^{inf} \lambda^{i} = \frac{1}{1-\lambda}$

אם נכפול את הכל במכנה: $(1-\lambda)\sum_{i=0}^{inf} \lambda^{i} = 1$

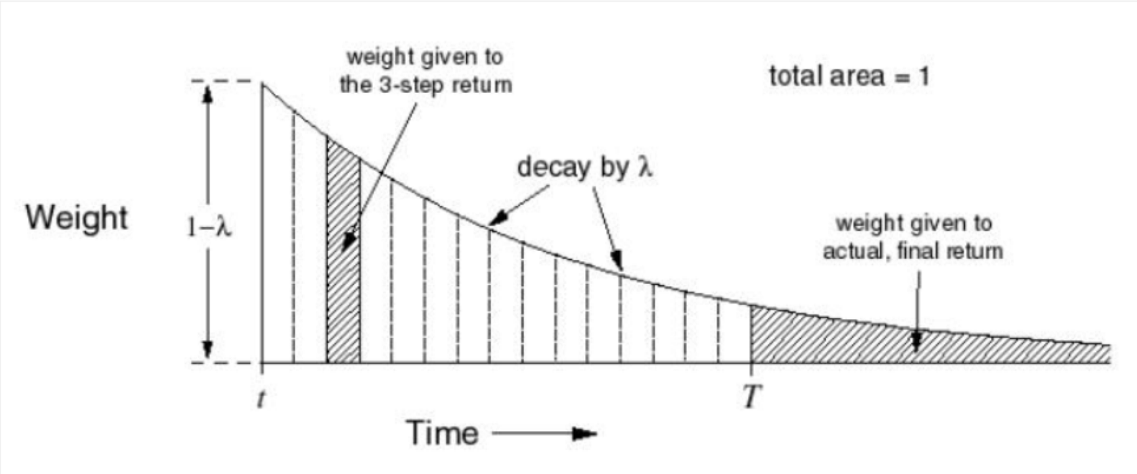
אבל מכיוון שמספר הצעדים הוא סופי: $(1-\lambda)\sum_{i=0}^{T-2} \lambda^{i} + (1-\lambda)\sum_{i=T-1}^{inf} \lambda^{i} = 1$

נפתח את הסכום השמאלי: $(1-\lambda) + (1-\lambda)\lambda + (1-\lambda)\lambda^{2} + ... + (1-\lambda)\sum_{i=T-1}^{inf} \lambda^{i}$

נרצה לדעת מה המשקל שניתן לאחרון כדי שהכל ייסכם ל-1. יש לנו שוב סכום של סדרה הנדסית אינסופית:

$$(1-\lambda)\sum_{i=T-1}^{inf} \lambda^{i} = (1-\lambda)\frac{\lambda^{T-1}}{1-\lambda} = \lambda^{T-1}$$

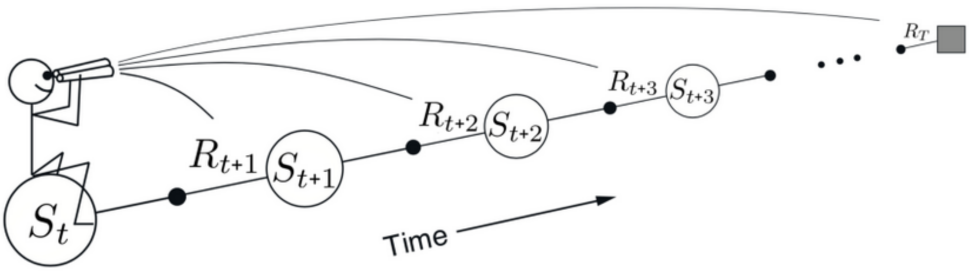
יוצא שהאחרון מקבל משקל גדול יחסית:



- λ הוא ערך בין 0 ל-1, בד"כ קרוב ל-1.
- המקרה הפרטי בו $\lambda = 0$ ייתן לנו בדיוק את $TD(0)$.
- והמקרה הפרטי בו $\lambda = 1$ ייתן בדיוק את MC.

forward view

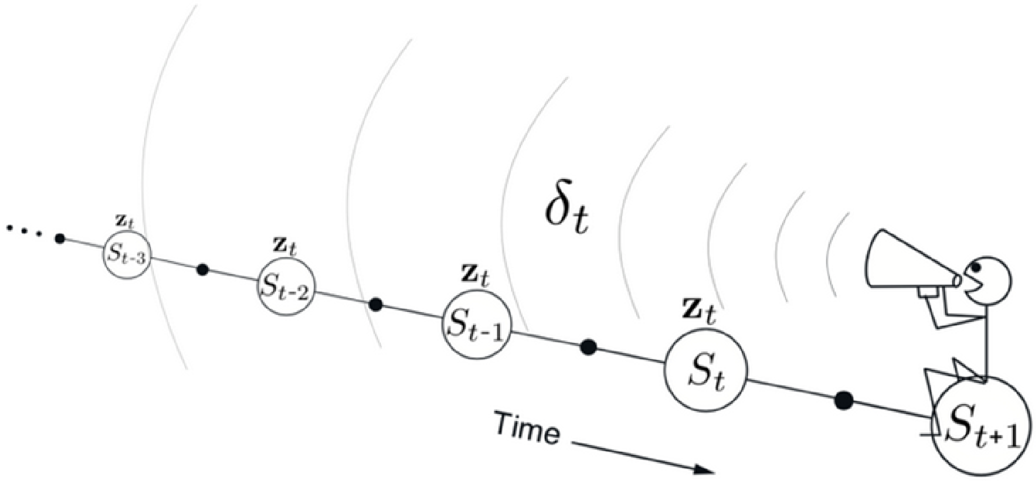
מסתכלים על כל ה-states הבאים, ומעדכנים באמצעות ה-values שלהם את ה-value של ה-state הנוכחי.
כלומר רצים עד הסוף וחוזרים אחורה בשביל העדכון.



In Forward View we look ahead n steps for future rewards

backward view

מה שבד"כ משתמשים בו בפועל. כל state משדר אחורה לכל ה-states האחרים מה הייתה הטעות בו, ואלו מעדכנים בהתאם לכך.



Backward View propagates the error δ to previous states

שיעור 3 - 2.8

- הערה מהבוחן בתחילת השיעור: סוכן ב-model free RL טוב יותר מסוכן שפועל ע"פ policy שנלמדה ע"י policy iteration כאשר ה-transition function אינה מדויקת.
  נשים לב שאם היא לא הייתה קיימת, אז לא היה ניתן לעשות בכלל policy iteration. כמו כן, אם היא הייתה טובה, אז policy iteration הייתה נותנת את התוצאה האופטימלית.

מה מרווחים מה-backward view?

שכל פעם אנחנו יכולים לעדכן אחורנית ולא חייבים לחכות להגיע עד הסוף כדי לעדכן.

מבחינת סיבוכיות, זה לא ממש עוזר, כי בכל מקרה כל אחד מה-states מתעדכן על סמך כל ה-states הבאים, גם ב-forward view.

גם ב-backward view וגם ב-forward view רוצים שהטעויות של ה-states שקרובים יותר לסוף יהיו מדויקות יותר. ככל שהם רחוקים יותר, הטעות תשפיע עליהם פחות.

eligibility traces

אנו נרצה שככל שאנחנו יותר קרובים ל-state, נעדכן יותר. במילים אחרות, ככל שמתרחקים ממנו המשקל קטן. לכן נגדיר מהו ה- eligibility trace של state: כמה זמן עבר מאז שראינו אותו. זה דועך לפי ה-λ שקבענו.

הדוגמה הפשוטה ביותר: כאשר ראינו את ה-state פעם אחת בלבד, אז נתחיל מ-1, ואז כל פעם נכפול את זה ב-λ (וגם ב-discount המובנה במערכת, הלוא הוא γ), וה-eligibility trace של אותו state ידעך ל-0.

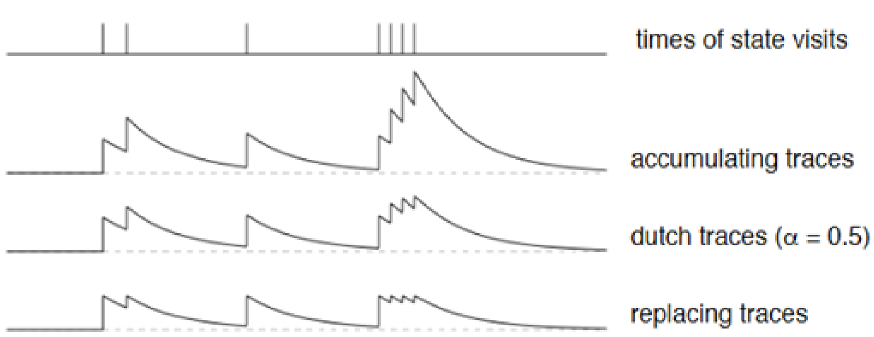
ואז עבור כל אחד מה-states נעדכן לפי ה-eligibility trace שלו כך: מחברים לערך שהיה לנו קודם מכפלה של ה-learning rate שלנו (α), עם ה-eligibility trace $E(S)$ ועם ה-δ שחישבנו, כלומר הטעות $\delta(t)$ שחישבנו עכשיו.

התהליך עצמו:

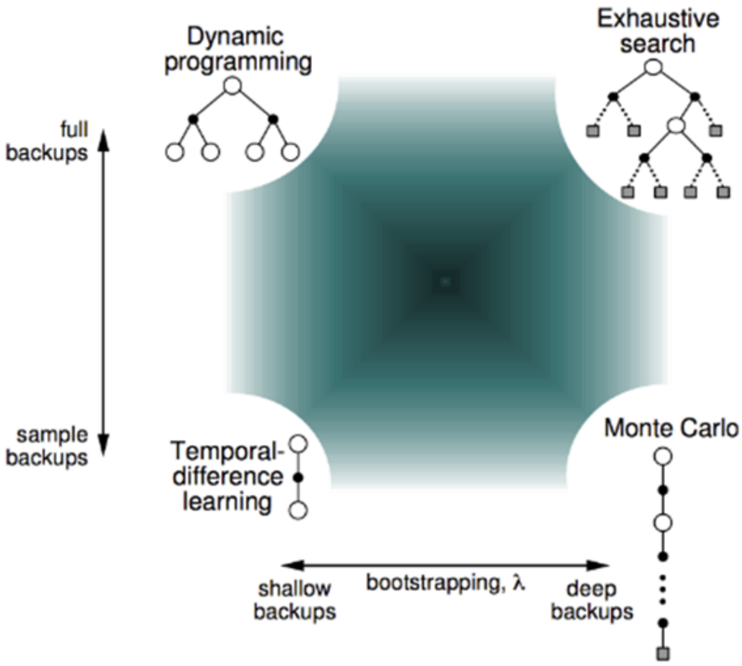
Initialize $V(s)$ arbitrarily (but set to 0 if s is terminal)
Repeat (for each episode):
    Initialize $E(s) = 0$, for all $s \in \mathcal{S}$
    Initialize S
    Repeat (for each step of episode):
        $A \leftarrow$ action given by π for S
        Take action A, observe reward, R, and next state, S'
        $\delta \leftarrow R + \gamma V(S') - V(S)$
        $E(S) \leftarrow E(S) + 1$ (accumulating traces)
        or $E(S) \leftarrow (1-\alpha)E(S) + 1$ (dutch traces)
        or $E(S) \leftarrow 1$ (replacing traces)
        For all $s \in \mathcal{S}$:
            $V(s) \leftarrow V(s) + \alpha\delta E(s)$
            $E(s) \leftarrow \gamma\lambda E(s)$
        $S \leftarrow S'$
    until S is terminal

אם מבקרים באותו state כמה פעמים, יש הבדלים בין השיטות:

- accumulating traces - כל פעם שמבקרים ב-state מוסיפים לערך שלו 1+. כלומר אם ביקרנו ב-state ממש לא מזמן, אז ה-eligibility trace עדיין לא הספיק לדעוך ל-0, ולכן יהיה לנו ערך גדול מ-1 בעדכון. בפרט, אם ביקרנו הרבה פעמים ברצף, זה ממש יעלה.
- dutch traces - לוקחים ממה שצברנו עד עכשיו רק $(1-\alpha) - fraction$ ואז מוסיפים 1.
- replacing traces - לא מתחשב במה שצברנו עד כה, פשוט כל פעם שביקרנו זה עולה ל-1.



עד כה דיברנו רק על policy evaluation, הנה סיכום של כל מה שראינו עד כה:



- full backup מתייחס לרוחב, ו-deep backup מתייחס לעומק.

<u>Model Free Control:</u>

כעת נשתמש בשיטות שלמדנו עבור policy evaluation כדי לעדכן את ה-policy (ולא רק כדי להעריכה).

Monte-Carlo Control

דומה למה שראינו עם policy iteration במובן שגם כאן נשתמש ב-**Monte-Carlo Evaluation** כדי לעשות את ההערכה, כלומר נריץ כמה trajectories עם ה-policy הנתונה, ואז נעדכן אותה בהתאם.
[לא צריך את ה-transition function, כי מריצים בעולם האמיתי].

- מתחילים עם random policy
- חוזרים על הפעולות הבאות עד התכנסות:
  - מריצים את ה-policy פעם אחת או מספר פעמים (עד הסוף)
  - מעדכנים את הממוצע של ה-expected return לכל state (שעברנו בו במסלול). העדכון מתבצע לאחר שמגיעים לסוף ומעדכנים לאחור.
  - מעדכנים את ה-policy כדי למקסם את V (ה-average expected return).

בעיה 1:
לא ניתן לדעת איך לעדכן את ה-policy כדי למקסם את V אם לא נתונה לנו ה-transition function, כי לא יודעים לאיזה מצב תוביל אותנו כל פעולה. כלומר זה לא ממש עוזר שיודעים עד כמה טוב להיות בכל state אם אין לנו אפשרות לדעת מאיזה state ו-action מגיעים לאיזה state.

$\pi' := argmax_a(\sum_{s_j} T(s_i, a, s_j)v(s_j))$ השתמש ב-T.

פתרון: Q-values

$$\pi' := argmax_a Q(s_i, a)$$

כאשר $Q(s_i, a)$ הוא ה-expected return עד סוף המשחק כאשר נמצאים ב-state מסוים ועושים action מסוים, ע"פ policy מסוימת, π (אבל a הוא לא לפי ה-π, כלומר בצעד הבא עושים איזושהי פעולה, ומכאן ואילך פועלים לפי π).
זה יעבוד מכיוון שכאן לא אכפת לנו לאיזה state עוברים, אפשר לעדכן את ה-policy ללא זה. בוחרים את ה-action שנותן לנו את ה-Q-value הגבוה ביותר.

תיקון ה-Monte Carlo Control:
- מתחילים עם random policy
- חוזרים על הפעולות הבאות עד התכנסות:
  - מריצים את ה-policy פעם אחת או מספר פעמים (עד הסוף)
  - מעדכנים את הממוצע של ה-expected return לכל state-action pair. העדכון מתבצע לאחר שמגיעים לסוף ומעדכנים לאחור (אבל הפעם זה לא מחלחל ל-states, אלא ל-state-action pairs).
  - מעדכנים את ה-policy כדי למקסם את Q.

הערות:
- הגדלנו את הטבלה, עכשיו היא לא רק בגודל ה-states, אלא בגודל states×actions.
- לא חייב לשמור את ה-policy, אלא אפשר פשוט בכל התלבטות מה נעשה לבחור במה שממקסם לנו את ה-Q.

בעיה 2:
ייתכן שיש מקומות שלא נבקר בהם בכלל (גם במקרה של policy סטוכסטי וגם במקרה של דטרמיניסטי).
במילים אחרות: אין exploration.

Exploration vs. Exploitation
exploration משמעותו לחקור מקומות חדשים, exploitation משמעותו לנצל ידע קיים.
- בד"כ בטסטים, כשנרצה לבדוק כמה השיטה שלנו טובה, נוריד את ה-exploration ונשאיר רק את ה-exploitation.
- במהלך הלמידה חשוב לעשות גם exploitation ולא רק exploration כי אז לא נלמד. נעשה תמיד פעולות רנדומליות, מה שימנע מאיתנו להגיע ל-states הטובים. למרות שלא באמת חשוב לנו למקסם את הרווח בזמן זה, חשוב לעשות exploitation מכיוון שנרצה להגיע ל-states שבאמת נהיה בהם במציאות (למשל רכב אוטונומי צריך לדעת להגיע לכביש, ולא להיתקע בחנייה).

פתרון: $\epsilon - greedy$
שיטה בה בוחרים פעולה רנדומלית (exploration) בהסתברות ϵ, ואחרת בוחרים את הפעולה הטובה ביותר (exploitation).

<u>אלגוריתם מתוקן</u>(אפשר להריץ אותו בסביבה אמיתית והוא לומד את ה-policy):

$Monte - Carlo\ \epsilon - greedy\ Control$

- מורכב משני חלקים: Monte-Carlo Evaluation בשביל ה-policy, ומ-$\epsilon - greedy$ בשביל ה-exploration.

- מתחילים עם random policy
- חוזרים על הפעולות הבאות עד התכנסות:
- מריצים $\epsilon - greedy$ עם ה-policy שיש לנו עכשיו (פעם אחת או יותר).
- מעדכנים את הממוצע של ה-expected return לפי ה-Q-values, לקראת ה-return שקיבלנו בפועל. הנוסחה: $Q(s,a) := Q(s,a) + \alpha(G - Q(s,a))$
- מעדכנים את ה-policy כדי למקסם את Q.

GLIE Property

התנאי בעצם מחולק ל-2 תכונות: greedy in the limit, ו-infinite exploration.
אם נענה על התנאי הזה, מובטח לנו שאם נריץ מספיק פעמים, בסופו של דבר נתכנס למקסימום.

greedy in the limit
באינסוף נתנהג בצורה גרידית (ולא כמעט גרידית, למשל שאפסילון מהפעמים נעשה פעולות אחרות). צריכים את זה כדי להבטיח שנבחר בסופו של דבר את ה-policy האופטימלית.

infinite exploration
בסופו של דבר נבקר בכל ה-states, יהיה לנו את כל ה-exploration באינסוף. לא רק זה, גם נבקר בכל state אינסוף פעמים (כדי שלא נסתמך רק על ביקור אחד/ ביקורים מעטים).

$\epsilon - greedy$ אינו GLIE, כי זה לא מתכנס ל-policy שהוא greedy in the limit.
פתרון: ϵ ידעך עם הזמן, למשל מכפלה ב-$\frac{1}{k}$ או $\frac{\epsilon}{\sqrt{k}}$.
(אבל צריך לוודא שאנו לא מתכנסים מהר מידי, כי אז התנאי השני של ביקור בכל מצב אינסוף פעמים לא יתקיים).
לפעמים גם יורדים עד איזשהו ערך קבוע, כדי לשמור קצת exploration (עדיין לא GLIE כי לא יורדים עד ה-0 באינסוף, אבל מקובל), או מאתחלים ϵ רנדומלי ומשאירים אותו קבוע בפרק זמן מסוים, ורק אז מתחילים לרדת וכדו'.

פתרון נוסף: Softmax Exploration
בוחרים לפי פונקציית ה-softmax: ככל שפעולה טובה יותר, יש סיכוי גדול יותר שנבחר אותה (אבל היא לא תיבחר בוודאות).

$$\pi(s,a) \propto e^{Q(s,a)} \text{ i.e: } \pi(s,a) = \frac{e^{Q(s,a)}}{\sum_{a' \in A} e^{Q(s,a')}}$$

הערה: הרבה פעמים כשממדלים התנהגות של בני אדם, מגלים שהם משתמשים בזה. זה משפר מודלים שהניחו באופן שגוי בהתחלה שאנשים תמיד בוחרים בפעולה המקסימלית, כלומר בפועל הם פשוט בוחרים בה בהסתברות גבוהה יותר, אבל יש סיכוי שיבחרו בפעולות האחרות.
(אפשר לכפול את החזקה של ה-e בקבוע c כלשהו, שככל שיהיה יותר גדול, יהיה פחות exploration, יותר דטרמיניסטי).
יש גם Temperature decay: עם הזמן נהיים יותר ויותר גרידיים.

$$\pi(s,a) \propto e^{\frac{1}{T}Q(s,a)}$$

$TD(0)\ \epsilon - greedy\ Control$

(זה מה שאנחנו מכירים כ-Q learning מהקורס בלמידה עמוקה).
אותו הדבר כמו $Monte - Carlo\ \epsilon - greedy\ Control$, רק שבמקום Monte-Carlo policy evaluation משתמשים ב-$TD(0)$.

הנוסחה היא: $Q(s,a) := Q(s,a) + \alpha(G - Q(s,a))$ (ממקודם), רק ש-G מוחלף ב-$r + \gamma Q(s', a')$.
אם מריצים עד הסוף זה נקרא on policy. אפשר לקחת בתור a' את הפעולה שאנו יודעים שביצענו, וזה בדיוק SARSA שנראה בהמשך.
אם לא מריצים עד הסוף, כלומר off policy, אפשר להניח שבצעד הבא נבחר את הפעולה שממקסמת את ה-Q value, ואז קיבלנו את ה-Q-learning.

מעדכנים לקראת המידע המדויק יותר שיש לנו עכשיו לגבי ה-Q value של ה-state הנוכחי. הנוסחה:

$$Q(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t,a_t)}_{\text{old value}} \right)$$

Wikipedia

OpenAI Gym

יש שם כל מיני משחקים שמתאימים ל-RL.
דוגמה: משחק ה-frozen lake - יש לוח קבוע, נקודת ההתחלה היא S והמטרה היא להגיע ל-G ו"לנצח את המשחק".
הנקודות שהן H הן חורים, "נופלים" ומסיימים את המשחק עם 0 נקודות. הנקודות שהן F הן "frozen", כלומר אם עושים פעולה מסוימת במטרה להגיע לשם, יש סיכוי כלשהו של החלקה, כלומר ה-transition function הוא סטוכסטי.

```
SFFF
FHFH
FFFH
HFFG
  (Up)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Down)
SFFF
FHFH
FFFH
HFFG
  (Right)
```

דוגמה לטבלת ה-Q values של כל אחד מה-states:

```
            Left         Down         Right        Up
>>> Q
array([[ 0.13211114,  0.13004507,  0.1300684 ,  0.12543337],
       [ 0.10225738,  0.1031522 ,  0.07597036,  0.11599257],
       [ 0.11020248,  0.10681383,  0.10797833,  0.10796757],
       [ 0.08812456,  0.08083084,  0.06962331,  0.1027171 ],
       [ 0.16079886,  0.11567742,  0.06060604,  0.10754974],
       [ 0.        ,  0.        ,  0.        ,  0.        ],
       [ 0.07352861,  0.08338232,  0.14634421,  0.05314348],
       [ 0.        ,  0.        ,  0.        ,  0.        ],
       [ 0.13517999,  0.1316702 ,  0.13380257,  0.22340567],
       [ 0.16513649,  0.34041771,  0.20438269,  0.13474098],
       [ 0.3378061 ,  0.23741361,  0.12737932,  0.20518666],
       [ 0.        ,  0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ],
       [ 0.29020822,  0.28142782,  0.51664485,  0.26141714],
       [ 0.46086019,  0.73573282,  0.48198415,  0.48449962],
       [ 0.        ,  0.        ,  0.        ,  0.        ]])
```

(זה חושב עם transition function מסוים, הסתברויות מסוימות שנקבעו מראש, למשל: ההסתברות שנבחר לזוז ימינה ונישאר במקום/ שנזוז למעלה או למטה במקום וכד'...).
נשים לב שהערכים כולם הם בין 0 ל-1, כי זה מייצג את תוחלת הרווח עד סוף המשחק. כמו כן, הסכום לא 1.

<u>On Policy RL</u>

עד עכשיו (ב-Q learning) דיברנו על Off Policy, שאמר שלא מסתכלים על מה שה-policy אומרת לנו לעשות כדי לעדכן את ה-learn value, אלא מניחים שנפעל לפי ה-max למשל. כלומר יכול להיות שנעשה בפועל משהו מסוים, אבל הנחנו שנעשה אחרת.
יש בעצם מעין שתי policies, אחת הנתונה לנו (שהיא בעצמה סטוכסטית - $\epsilon - greedy$), שתמיד נפעל על פיה. אבל כאשר באים לעדכן, מניחים שבפעם הבאה נפעל על פי הpolicy הנוספת, שהיא "max policy".

כעת נדבר על On Policy: משתמשים בדיוק ב-policy שאנו פועלים על פיה כדי לעדכן את הערכים.
ה-On Policy של Q-learning נקראית State Action Reward State Action) SARSA).
הנוסחה זהה לזו של ה-Q-learning, רק שבמקום לפעול ע"פ ה-max מסתכלים על ה-state וה-action הבאים שנעשה (באמת, יודעים זאת כי קודם נבחר את ה-action לפי הpolicy, שיכול להיות שהיא סטוכסטית, ואז בדיעבד לאחר שיודעים מה בחרנו מעדכנים).

$$Q(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t,a_t)}_{\text{old value}} \right)$$

Wikipedia

$$\boxed{Q(s_{t+1}, a_{t+1})}$$

הבדלים בין On Policy ל-Off Policy:

ישנן שתי גישות: או שאומרים שמכיוון שאנחנו בסימולציה ואחר כך נוריד את ה-exploration אז לא אכפת לנו ש"נפלנו" עכשיו אבל בפועל זה לא יקרה,זה מתאים ל-Q-learning, כלומר Off Policy. או שנרצה להשאיר את ה-exploration גם אחר כך ואז נעדיף לעשות פעולות "בטוחות" יותר, מה שייתן לנו SARSA, כלומר On Policy.

ה-On Policy בד"כ שמרנים יותר, מפחדים, וה-exploration הוא חלק מהותי שם. כלומר, למרות שיש exploration (אפילו שזה זמן לימוד) הוא ישתדל להיות זהיר יותר. כלומר, אם רוצים להוריד אחר כך את ה-exploration אז לא כדאי להשתמש ב-On Policy, כי Off Policy אמורים לעבוד טוב עם ה-policy שבפועל צריך להריץ אותה.

התכנסות SARSA:

מובטח ש-SARSA מתכנס ל-policy האופטימלי, אם הוא:
- GLIE
- Robbins-Monro sequence של ה-learning rate, α, שזה אומר שני דברים:

1. $\sum_{t=1}^{\infty} \alpha_t = \infty$ [יכול ללמוד בתחום כולו]
2. $\sum_{t=1}^{\infty} \alpha_t^{\ 2} < \infty$ [בסופו של דבר מפסיק להשתנות]

כלומר צריך לדעוך מהר מספיק, אבל עדיין שואף לאינסוף. למשל $\frac{1}{t}$ זו דוגמה טובה.

בד"כ משתמשים ב-learning rate קבוע, ולא מתייחסים לתנאי הראשון.

הערה: אמרנו שרוצים להשתמש ב-learning rate קבוע כדי לתת משקל יותר גדול לצעדים האחרונים, מה ש-learning rate דועך "מקלקל". זה מנוגד.

כל הדברים שדיברנו עליהם קודם, ניתן לדבר עליהם גם כאן:

$n - step\ TD\ \ and\ \epsilon - greedy$

נקרא גם $n - step\ SARSA$.

במקום לעדכן לקראת צעד אחד קדימה, מעדכנים לקראת n צעדים קדימה.

זה ה-tradeoff בין ה-bias של SARSA ל-variance של MC. אפשר לשלוט עליו באמצעות n-step TD ו-$TD(\lambda)$.

(אפשר להשתמש ב-n-step TD וב-$\epsilon - greedy$ לצורך policy evaluation וגם ל-policy control).

- Consider the following n-step returns for $n = 1, 2, \infty$:

$$\begin{array}{lll} n = 1 & (Sarsa) & q_t^{(1)} = R_{t+1} + \gamma Q(S_{t+1}) \\ n = 2 & & q_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 Q(S_{t+2}) \\ \vdots & & \vdots \\ n = \infty & (MC) & q_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T \end{array}$$

- Define the n-step Q-return

$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n})$$

- n-step Sarsa updates $Q(s, a)$ towards the n-step Q-return

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(q_t^{(n)} - Q(S_t, A_t) \right)$$

$TD(\lambda)\ \ and\ \epsilon - greedy$

נקרא גם $SARSA(\lambda)$.

במקום לעדכן לקראת ערך מסוים, מעדכנים לקראת צירוף של כולם. הצירוף הזה הוא:

$$q_t^{\ \lambda} = (1 - \lambda) q_t^{\ (1)} + (1 - \lambda)\lambda q_t^{\ (2)} + (1 - \lambda)\lambda^2 q_t^{\ (3)} + ... + \lambda^{T-1} q_t^{\ (\infty)}$$

כלומר $Q(s, a) := Q(s, a) - \alpha(q_t^{\ \lambda} - Q(s, a))$

$SARSA(\lambda)\ backward\ view$

ה-eligibility traces מורכבים עכשיו גם מה-state וגם מה-action, ולא רק מה-state.

מתחילים ב-0, ומעדכנים בדרך כלשהי מבין הדרכים שהזכרנו קודם. כאן נשתמש ב-accumulate:

$$E_t(s,a) = \gamma\lambda E_{t-1}(s,a) + 1\{S_t = s \wedge A_t = a\}$$

ה-1 הוא אינדיקטור - אם מה שבפנים נכון (אם ביקרנו ב-state s ועשינו action a) אז זה מחזיר 1, אחרת 0.
את מה שהיה קודם אנו מכפילים ב-λ, כי רוצים להתייחס יותר ל-states קרובים ופחות לעתידיים, וגם ב-γ, כי כל מה שקורה בעתיד פחות קריטי לנו.

ה-Q values מתעדכנים לקראת ה-TD error, δ_t ומוכפלים בערך של ה-eligibility trace, $E_t(s,a)$:

$\delta_t = (R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})) - Q(S_t, A_t)$.

כל צעד t עבור כל הפעולות והמצבים: $Q(s,a) := Q(s,a) + \alpha\delta_t E_t(s,a)$

- האלגוריתם:

Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
  $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
  Initialize S, A
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$
    $E(S,A) \leftarrow E(S,A) + 1$
    For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
      $E(s,a) \leftarrow \gamma\lambda E(s,a)$
    $S \leftarrow S'$; $A \leftarrow A'$
  until S is terminal

Dyna

שילוב של Model-based ו-Model-free.
כלומר אנחנו מדברים על מצב שאין לנו את המודל, אז איך אפשר לעשות Model-based?
לבנות אותו תוך כדי התהליך. כלומר, ללמוד את פונקציות ה-transition ואת ה-reward מאינטראקציה אמיתית עם הסביבה, לעשות סימולציות ואז לעדכן את ה-Q function (בהתבסס על הניסוי האמיתי וגם ה-simulated).

האלגוריתם:

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
  (a) $S \leftarrow$ current (nonterminal) state
  (b) $A \leftarrow \varepsilon$-greedy(S, Q)
  (c) Take action A; observe resultant reward, R, and state, S'
  (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S,A)\right]$
  (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
  (f) Loop repeat n times:
    $S \leftarrow$ random previously observed state
    $A \leftarrow$ random action previously taken in S
    $R, S' \leftarrow Model(S,A)$
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S,A)\right]$

- עד d זה Q-learning רגיל.
- בעדכון לפי הסימולציה: ה-state שמניחים שהיינו בו נבחר בצורה רנדומלית מתוך ה-states שראינו, וגם ה-action שעושים רנדומלי, שתואם את ה-s הזה. לפי זה עדכנו את ה-Q.
- החישובים לא קשורים ל-state שאנו נמצאים בו עכשיו, אבל כן קשורים לאיזשהו state שראינו בעבר, כי זה model based אז צריך שתהיה למידה על סמך המודל.
- נשתמש ב-Dyna כאשר האינטראקציה עם הסביבה היא יקרה. כאשר היא זולה, אין טעם להשתמש ב-Dyna כי אין צורך לעשות סימולציות במקום לבדוק את מה שקורה באמת.

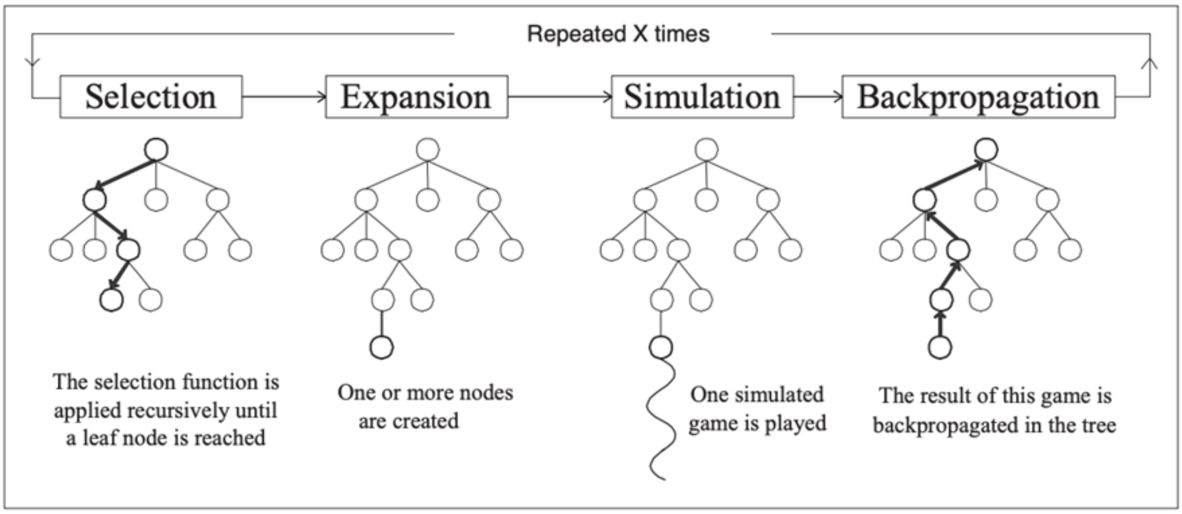
הרעיון הכללי ב-Simulated-Based Search:
מתחזקים מודל של העולם, ורוצים להשתמש בו כדי לדעת איזו פעולה לבצע עכשיו.

הדוגמה הכי פשוטה: Simple MC Search
- אין כאן למידה.
- דורש מודל, כלומר מניחים שקיימים T ו-R כלשהם. (לפעמים כן לומדים אותם).
- נתונה policy כלשהי, π.
- עבור כל state: מסתכלים על כל אחד מה-actions שניתן לבצע ומריצים מסלולים עד הסוף. נבחר את ה-action שנותן את תוחלת הרווח הממוצעת הגדולה ביותר.
- הערה: ה-action הבא נבחר על ידינו, אבל כל שאר הפעולות עד הסוף נקבעות ע"פ ה-policy.

MC Tree Search
הנתונים זהים לשיטה הקודמת, וגם כאן מריצים סימולציות עד הסוף.
רק שכאן מעדכנים לכל תת עץ את הממוצע של הרווח הצפוי לנו.
יש לנו מעין שתי policies: הנתונה, איתה מתנהלים באופן כללי, ו-policy נוספת, שבוחרת את הפעולות הטובות ביותר בכל תת עץ. נפעל ע"פ ה-policy השנייה כל עוד יש לנו מידע, ומרגע שאין לנו עוד מידע, נמשיך לפעול ע"פ π.
הבעיה עם זה שתמיד הולכים למקום שראינו שהוא טוב יותר היא: שנוכל להתבסס על מידע לא אמין מספיק. כלומר, אם פעם אחת עשינו מסלול מסוים וניצחנו, לא נרצה תמיד לעשות את הפעולה שהובילה אותנו לשם, כי יכול להיות שזה גם יכול להוביל אותנו להפסד. כלומר נרצה לקחת בחשבון גם את מספר הפעמים שהגענו לתת העץ הזה (יחד עם מספר הפעמים שניצחנו משם).
משתמשים בשיטה זו בד"כ בשביל לנצח במשחק.
האפשרויות לבחירת תת העץ:

1. $\frac{w}{n}$ - לבחור תמיד את זה שהיחס של מספר הנצחונות למספר הביקורים שלו הוא הגדול ביותר. אבל הבעיה היא שאז יהיה לנו רק exploitation ולא exploration.
2. $\epsilon - greedy$
3. UCB (Upper Confidence Bound) - הערכה על כמה טוב לבחור בכל אחת מהאופציות, והיא: $\frac{w}{n}c\sqrt{\frac{logN}{n}}$. כאשר N = מספר האיטרציות שהרצנו בסה"כ, n = מספר הסימולציות שהרצנו, ו-w = מספר הניצחונות. כלומר ככל שהרצנו פחות איטרציות, יש לכך עדיפות גבוהה יותר (זה ה-exploration).

סיכום: MC Tree Search



Importance sampling

אם רוצים לשערך ערך מסוים, אפשר פשוט לדגום מספיק דגימות, וזה אמור להתכנס לגבול האמיתי.

שיעור 4 - 9.8

Importance sampling

המטרה: להעריך ממוצע של ערך מסוים (למשל הכנסה) באוכלוסיה מסוימת המחולקת לקבוצות. ההנחה היא שאנו יודעים את התפלגות הקבוצות באוכלוסיה.

הרעיון של Importance sampling הוא שיש לנו דגימות מהתפלגות מסוימת, אבל ההתפלגות שבאמת מעניינת אותנו היא אחרת.

נסתכל על כמה מקרים:

- המקרה הפשוט: ניתן לדגום מכל הקבוצות, מאותה התפלגות כמו שקיימת באוכלוסיה. אז ניתן פשוט לעשות ממוצע על הדגימות שקיבלנו.
- מקרה קצת פחות פשוט: ניתן לדגום מכל הקבוצות אך לפי התפלגות אחידה (ולאו דווקא לפי זו הקיימת באוכלוסיה). במקרה זה אפשר לקחת את ממוצע הדגימות מכל קבוצה ולמשקל אותה לפי ההתפלגות האמיתית שאנו יודעים.
- המקרה המורכב: יש לנו המון קבוצות, או שאנו לא יכולים לדגום מההתפלגות האמיתית מסיבה מסוימת (למשל: ערך רציף כמו גובה, או פשוט קבוצות רבות כמו state-action pairs שלא ניתן לקבל את כולם בסימולציות ריצה). נניח שבגלל האילוצים הללו, תכונה מסוימת שמתפלגת גאוסיאנית נדגמת בצורה הפוכה (יש לנו יותר דגימות מהקיצוניים דווקא). אז הפתרון:

אם אנו יודעים את ההתפלגות ממנה דגמנו - $d(x)$, את ההתפלגות האמיתית - $t(x)$ ואת ה-labels של הדגימות (פונקציה $(f(x)$, נוכל לחלק את ה-f ב-sampling(d), ולהכפיל בערך האמיתי (t).

למה זה נכון?

- $E_{x\sim r}[f(X)] = \sum_x t(x)f(x) =$
- $= \sum_x \frac{t(x)f(x)d(x)}{d(x)} = E_{x\sim d}[f(x) \cdot \frac{t(x)}{d(x)}]$

$\sim \frac{1}{n}\sum_{i=1}^{n} f(x_{d,i})\frac{t(x)}{d(x)}$

הסיבה המרכזית שראינו את זה: נשתמש במידע שהשגנו מ-policy אחד בשביל לדעת policy אחר. אבל נשים לב שלא נוכל להשתמש בערכים שקיבלנו כפי שהם כדי לעשות ממוצע ולשערך את ה-Q values, אלא נצטרך לבצע עליהם חישובים מסוימים לפני.

מצגת 4 - Deep Reinforcement Learning

(אנחנו עדיין ב-Model free RL).
עד כה התמקדנו במצב בו אפשר להציג את הבעיה בצורת טבלה (מס' states קטן יחסית).
כשיש לנו המון מצבים, למשל תמונה, לא נוכל לתחזק אותה בטבלה, בטח לא כשנוסיף actions.
בעיה נוספת היא שהסיכוי שנראה את אותו המצב פעמיים, או שנראה מצבים שהם זהים הוא אפסי.

<u>הפתרון:</u> **Function Approximation**
נשתמש בפונקציה שתשערך לנו את ה-Q value. באופן דומה למה שעושים בלמידה עמוקה באופן כללי: בהינתן כל מיני דוגמאות, מאמנים איזושהי מערכת. גם כאן ניקח המון דוגמאות שלכל אחת מהן יש ערך אמיתי של Q value. נאמן מערכת (למשל NN) שבהינתן state ו-action מסוימים, תנבא את ה-Q value. ניתן להשתמש במערכת הזו כדי להריץ את ה-policy (בוחרים את ה-action שהמערכת שלנו ניבאה שיש לו את ה-Q value המקסימלי).

- כרגע מדובר רק על מצב שיש מעט actions.

איך נשיג את הרשת הזו?
נלמד אותה. בשביל כך צריך לדעת מה ה-Q value האמיתי (ה-label, כדי לאמן את הרשת).
הדרך הכי פשוטה להשיג את ה-label הזה היא להריץ מונטה-קרלו: בהינתן policy מסוימת שאנו משתמשים בה (בהתחלה זו בד"כ תהיה רנדומלית), נריץ כמה trajectories עד הסוף וכך נשיג את ה-Q values האמיתיים עבור ה-policy הנוכחית.

<u>דוגמה:</u>
נניח שישנם 5 פיצ'רים בינאריים המייצגים את המצבים, 5 פעולות, $\gamma = 1$.
ונניח שיש את ה-trajectory הבא:

$$(01001{,}3) \rightarrow_{+2} (10010{,}2) \rightarrow_{-4} (01010{,}1) \rightarrow_{+3} (00011{,}2) \rightarrow_{+6} (end)$$

אז ה-Q value האמיתי של להיות במצב (00011) ולעשות את action 2 הוא 6, ה-Q value של להיות במצב (01001) ולעשות action 3 הוא 7=2-4+3+6. כך בעצם יש לנו training data שניתן לאמן רשת של רגרסיה לפיה:

- $input: (01001{,}3)\ label: 7$
- $input: (10010{,}2)\ label: 5$
- $input: (01010{,}1)\ label: 9$
- $input: (00011{,}2)\ label: 6$

*אם הפעולות סטוכסטיות אפשר להריץ כמה פעמים. כמו כן, אם אפשר להתחיל מכמה מצבי התחלה שונים. (אם יש מצב התחלה אחד ופעולות דטרמיניסטיות, וכן ה-policy שאנו פועלים על פיה דטרמיניסטית, אין טעם להריץ כמה פעמים).

הרבה פעמים משתמשים ב-$TD(0)$ במקום במונטה קרלו, כדי למנוע ריצה חוזרת עד הסוף, ולעדכן את ה-Q-value רק ע"פ הצעד הבא. כמו כן, במונטה קרלו יש variance גבוה, ועם $TD(0)$ יהיה variance נמוך.
עושים זאת ע"י שימוש ברשת פעמיים - כדי לשערך את ה-Q value של הפעם הבאה ולדעת מה ה-action הטוב ביותר.

ה-label של $Q(s, a)$ לפי Q Learning הוא $r + \gamma\, max_{a'} Q(s', a')$

מציאת ערכי ה-Q value "האמיתיים" עבור $TD(0)$:
- עושים action מסוים (רנדומלי או שממקסם את ה-Q value ברשת הנוכחית).
- מקבלים reward, r.
- מקבלים את ה-state החדש, הבא, s_n.
- מחשבים את ה-Q values של s_n (עם אותה הרשת), עבור כל אחד מה-actions. נסמן ב-q_n את ה-Q המקסימלי מבין כל ה-actions.
- ה-Q value עבור ה-state-action pair הקודם מתעדכן ל-$r + \gamma * q_n$.
- (לגבי שאר הפעולות שלא ביצענו - נשאיר את ה-Q values כפי שהיו קודם).
- מחשבים את ה-loss.

- הערה: במקום שהרשת תקבל גם את s וגם את a, עושים רשת אחת שמקבלת רק את s ויש לה מספר outputs כמספר ה-actions שיש לנו. זה לצורך חישוב מהיר.

Random Player

הפעולה שאנו בוחרים לעשות נדגמת רנדומלית, וכך גם ה-state שנגיע אליו.
חשוב לשים לב שהסוכן הרנדומי גם כן צריך להצליח לפעמים, כדי שהוא ילמד מה טוב לעשות (או שייכשל לפעמים, כדי ללמוד מה רע לעשות ואז ילמד לדחות את ה-rewards השליליים). אחרת הוא לא יוכל ללמוד. (לדוגמה במשחק ה-pong הוא מצליח לפעמים להבקיע).

במצגת יש קוד של רשת עבור משחק ה-pong. הרשת היא של Linear Regression. אבל במקרה של תמונות עדיף לעבוד עם Deep CNN Regression.
שיפורים:
- בהתחלה עדיף שהסוכן ישחק רנדומלית, כדי שלא ילמד לעשות פעולה אחת לא כ"כ טובה ולא ישתפר לעולם.
- ה-ϵ יתחיל מערך רנדומלי ויקטן לאט לאט, עד t מסוים.
- אפשר להשתמש במשהו שנקרא "replay buffer" שהוא שילוב של שני דברים: הראשון דומה ל-Dyna, למידה מסימולציה, כלומר מה-data הישן. אם יש לנו דוגמאות שכבר דגמנו בעבר, אפשר ללמוד מכל דוגמה מספר פעמים בשביל לחסוך בריצות נוספות (לא לשכוח לעשות shuffle). כמו כן, נשתמש ב-mini batch על מנת ללמוד על כמה דוגמאות יחד ולא על כל דוגמה בנפרד.
- 2 רשתות DQNN: במקום להשתמש באותה הרשת פעמיים, נשתמש ב-2 רשתות במובן הבא: בעיקרון יש לנו רשת אחת (main), והרשת ה"שנייה" (target), היא אותה הרשת לפני מספר איטרציות. הבעיה היא שאם עשינו סט פעולות שבמקרה היה לנו טוב, ה-bias ישפיע עלינו פעמיים - הוא יכול לתחזק יותר מידי את הפעולה הנוכחית שביצענו כך שבעתיד תמיד נחזור על אותן פעולות. זה נכון גם לכיוון השני - סט פעולות שבמקרה היה לנו רע.
פתרון: כשבוחרים פעולה, בוחרים לפי הרשת המעודכנת הנוכחית, אבל בשביל חישוב ה-Q value נשתמש ברשת הקודמת (שהיא תעודכן כל מספר episodes מסוים).

בשביל $TD(\lambda)$ function approximation זה נהיה מסובך, ויש הרבה עדכונים, לכן לא משתמשים בזה כ"כ.
כל state-action pair מתעדכן לקראת ה-λ שחוזר (q_t^{λ} עבור ה-forward view).
אבל אנחנו רוצים backward view, כדי לעדכן אחורה את כל ה-states שראינו עד עכשיו. בשביל זה צריך לשמור eligibility trace עבור כל אחד מהפיצ'רים שיש לנו ב-state (מגדילים את הערך כל פעם שראינו את הפיצ'ר הזה). מכיוון שיש לנו מספר גדול של states, נשמור וקטור עם ערך לכל משקולת ברשת שלנו (במקום לכל state).

עדכון המשקולות מתבצע כך (מניחים שהעדכון ל-eligibility trace הוא accumulate):

- $E_t = \gamma\lambda E_{t-1} + \nabla_w q(S_t, A_t, w)$
- $\delta_t = R_{t+1} + \gamma q(S_{t+1}, A_{t+1}, w) - q(S_t, A_t, w)$
- $w := w - \alpha\delta_t E_t$

עבור כל פעולה שעושים צריך לעדכן את כל המשקולות.

עד עכשיו התמקדנו ב-Value Based Method, בו למדנו את ה-Q values ולא היה policy (מלבד ה-policy המרומז של למקסם את ה-Q values).

עכשיו נדבר על **Policy Based Methods**

היתרונות שלו: העדכונים פחות רועשים, נוטים יותר להתכנס, אבל בד"כ לוקח להם יותר זמן.

המטרה: לבנות רשת שבהינתן state מסוים מה הסיכוי שנבחר בכל אחד מה-actions. אפשר לפתור זאת גם כאשר ה-actions רציפים (למשל ברכב אוטונומי, אפשר לסובב את ההגה ימינה טיפה/ ימינה הרבה וכו').

יש משהו שנקרא "system one, system two": מדמים את ה-policy based method ל-S1, דברים שהם יותר אינסטינקטיביים ואת ה-value based method ל-S2, דברים שצריכים יותר מחשבה.
בפועל, נשתמש בעיקר ב-actor critic שהוא משלב בין השניים, אך עדיין נחשב יותר policy based.
בגדול: אם שינויים קטנים מובילים לתוצאות שונות, אז value based יותר מתאים, ואם הם מובילים לתוצאות קרובות, אז policy based מתאים יותר.

בעיה מרכזית ב-Value Based היא שנתבקש לשערך מה קורה עד סוף המשחק, ולפעמים זה קשה מאוד כי יכולים להיות תרחישים רבים ולא נדע כמה נרוויח עד סוף המשחק.
בעיה נוספת: ככל שאנו נהיים טובים יותר, ה-Q value עולה עבור כל אחד מה-actions כי משחקים יותר טוב.

ה-Policy Based הוא ממש ההגדרה של הבעיה שלנו - רוצים לדעת איזו פעולה לעשות באיזה מצב (את ה-policy, ולא ממש מעניין אותנו כמה נרוויח, נריץ ונראה בממוצע…).
כמו כן, יש משחקים שבכלל אי אפשר לפתור באמצעות Value Based, כמו אבן, נייר ומספריים, כי אין policy דטרמיניסטי שהוא אופטימלי למשחק זה.
ב-Policy Based לומדים את ה-policy בצורה ישירה, ולא מחכים שזה ינבע מה-value (Q) function.

Softmax
אחת הדרכים הכי פשוטות של Policy Based - בהינתן state אומרים מה הסיכוי שנפעל ע"פ כל אחד מה-actions.
ואז ה-policy הוא: להריץ את ה-softmax (עד סוף המשחק) ולהחליט לפיו איזו פעולה לעשות.
עדכון ה-policy הוא לפי מונטה-קרלו, שנותן לנו את ה-return values: אם עשינו action טוב (שנותן לנו כמה שיותר rewards חיוביים, ולבסוף return גבוה), נגדיל את הסיכוי לעשות אותו, ואם עשינו action לא טוב, נקטין את הסיכוי לעשות אותו.
האלגוריתם המתואר נקרא REINFORCE (או Monte-Carlo Policy Gradient):

- Update parameters by stochastic gradient ascent
- Using policy gradient theorem
- Using return v_t as an unbiased sample of $Q^{\pi_\theta}(s_t, a_t)$

$$\Delta\theta_t = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$$

function REINFORCE
  Initialise θ arbitrarily
  for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ **do**
    for $t = 1$ to $T - 1$ **do**
      $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
    end for
  end for
  return θ
end function

נשים לב שאם v_t חיובי, מגדילים את הסיכוי לקחת את ה-action הזה, אחרת - מקטינים.

בעזרת Gradient Ascent נרצה למקסם את תוחלת הרווח במשחק (כולו).
One step MDP: ה-MDP הכי פשוט שיש, מתחילים באיזשהו state, עושים action מסוים, מקבלים reward ונגמר המשחק.

נסמן ב-$J(\theta)$ את תוחלת הרווח שלנו. $J(\theta) = E[r] = \sum_{s \in S} p(s) \sum_{a \in A} \pi_\theta(s, a) R(s. a)$
כאשר: $p(s)$ הוא הסיכוי שנתחיל ב-state (ממוצע על כל ה-states שלנו), $R(s, a)$ הוא ה-reward שנקבל כאשר אנו ב-s state ונעשה את פעולה a, $\pi_\theta(s, a)$ הוא ההסתברות שנבחר ב-a action לפי ה-policy שלנו.

נחשב את הגרדיאנט: $\nabla J(\theta) = \sum_{s \in S} p(s) \sum_{a \in A} \nabla \pi_\theta(s, a) R(s, a)$
נרצה לעדכן את ה-θות (שנמצאות בתוך ה-π, המשקלים) צעד בכיוון הנגזרות האלה.
לא נוכל לחשב את כל הגרדיאנט כי יש הרבה states ו-actions (בגלל זה אנו משתמשים בכלל ב-Policy-Based). כמו כן נשים לב שאם היינו ב-state מסוים, עשינו action מסוים וקיבלנו reward, לא יודעים מה היה קורה ב-action אחר.
את $p(s)$ אפשר להעריך לפי Sampling (כמו שעשינו עד עכשיו): אנו משתמשים ב-states שראינו בפועל, לכן נדע אותו בקירוב לפי דגימות. אבל הבעיה היא עם $\pi_\theta(s, a)$ - צריך להשתמש במעין policy אחידה (יוניפורמית), שבוחרים את כל אחד מה-actions באותה הסתברות (לכולם משקל זהה), אבל זו בעיה כי רוצים להשתמש ב-policy האמיתית שלנו (שהיא לא יוניפורמית), כדי שנגיע לשלבים המתקדמים יותר ונוכל ללמוד מה יקרה בפועל.

נשים לב כי: $\nabla J(\theta) = \sum_{s \in S} p(s) \sum_{a \in A} \pi_\theta(s, a) \nabla log(\pi_\theta(s, a)) R(s, a)$ (מהגדרת נגזרת של log).
הרווחנו מהמעבר הזה שיש לנו את ה-$\pi_\theta(s, a)$ האמיתית, ואז יש לנו את ההסתברות האמיתית שנבחר בפעולה a במצב s, ואז נוכל להסתכל על הדוגמאות האמיתיות שאנו מקבלים בפועל (שמוטות לפי ה-policy האמיתי) ועל ה-rewards האמיתיים, ונעדכן את המשקולות בהתאם.
$\nabla log(\pi_\theta(s, a))$ נקרא score function.

לסיכום: $\nabla J(\theta) = E_{\pi_\theta}[\nabla log(\pi_\theta(s, a)) r]$.

Policy Gradient Theorem

אם רוצים שזה יהיה נכון במקרה הכללי (ולא רק ב-one step MDP) נשנה כך:
.$\nabla J(\theta) = E_{\pi_\theta}[\nabla log(\pi_\theta(s,a))Q(s,a)]$

חישוב ה-Score of Softmax:

$$\pi_W(x,a) = \frac{e^{W_a x + b_a}}{\sum_{a' \in A} e^{W_{a'} x + b_{a'}}}$$

$$\frac{\partial log\pi_W(x,a)}{\partial w_{ia}} = \frac{\partial}{\partial w_{ia}} \log\left(\frac{e^{W_a x + b_a}}{\sum_{a' \in A} e^{W_{a'} x + b_{a'}}}\right)$$

$$= \frac{\partial}{\partial w_{ia}}(W_a x + b_a) - \frac{1}{\sum_{a' \in A} e^{W_{a'} x + b_{a'}}} \frac{\partial}{\partial w_{ia}} \left(\sum_{a' \in A} e^{W_{a'} x + b_{a'}}\right)$$

$$= x_{ia} - \frac{e^{W_a x + b_a}}{\sum_{a' \in A} e^{W_{a'} x + b_{a'}}} \cdot x_{ia}$$

$$= x_{ia}(1 - \pi_W(x,a))$$

$$\nabla_W log\pi_W(x,a) = x - x^T \pi_W(x,a)$$

שיעור 5 - 16.8

Actor Critic

יש שתי רשתות:רשת אחת מייצגת את ה-actor שמבצע את כל הפעולות, והרשת השנייה מייצגת את ה-critic, שזה ה-Q values שמתקנים את ה-actor.
ההבדל הוא שב-REINFORCE השתמשנו לצורך העדכון ב-v_t האמיתי, וכאן משתמשים ב-Q value, שיש לנו מרשת אחרת $Q(s_t, a)$.

ה-loss function שנרצה למזער (בעצם רוצים למקסם ערך מסוים, אז ממזערים את המינוס):

REINFORCE: $-\ log\ \pi_W(x,a) \cdot v_t$

Actor-Critic: $-\ log\ \pi_\theta(s_t, a_t) \cdot Q_W(s_t, a_t)$ [policy loss, for actor]

$(r + \gamma Q_W(s_{t+1}, a')) - Q_W(s_t, a))^2$ [value loss, for critic]

ב-value loss השתמשנו ב-MSE, ה-loss function הדיפולטיבי לבעיית רגרסיה.

- כאשר מעדכנים את המשתנים של הרשת חשוב לשים לב לא לגעת במשתנים של הרשת השנייה, כי אז זו רמאות, כי במקום למצוא את הפעולות הטובות נגיד שמה שעשינו זה טוב, כאילו רצינו מלכתחילה למקסם משהו אחר, כי שינינו את המשקלים. היה לנו את אותו העניין ב-GAN.

Actor Critic Algorithm

- Using linear value fn approx. $Q_w(s,a) = \phi(s,a)^\top w$
  Critic Updates w by linear TD(0)
  Actor Updates θ by policy gradient

```
function QAC
    Initialise s, θ
    Sample a ~ π_θ
    for each step do
        Sample reward r = R^a_s; sample transition s' ~ P^a_{s,·}
        Sample action a' ~ π_θ(s', a')
        δ = r + γQ_w(s', a') − Q_w(s, a)
        θ = θ + α∇_θ log π_θ(s, a)Q_w(s, a)
        w ← w + βδφ(s, a)
        a ← a', s ← s'
    end for
end function
```

מצגת 5 - Advanced RL Algorithms

נראה אלגוריתמים מורכבים יותר של RL, חלקם מהשנים האחרונות.

DDPG

אלגוריתם המאפשר לנו לעבוד עם action space רציף. זה שימושי לרכבים, כי יש הרבה פעולות רציפות (פנייה ימינה הרבה/קצת, לחיצה חלשה/חזקה על הגז וכד').
לכן עכשיו ה-policy שלנו הוא בעיית רגרסיה, ולא קלסיפיקציה. הוא מקבל את ה-state ומחזיר מספר שמייצג את הפעולה (אפשר להרחיב את זה בקלות לוקטורים).

איך בוחרים את הפעולה? מהרשת שמקבלת את ה-policy (ה-actor) יוצא action מסוים, אבל לא נרצה לעשות אותו תמיד כי נרצה להוסיף גם exploration, לכן נוסיף רעש של ϵ מסוים שמתפלג גאוסיאנית.

$$a = \pi_\theta(s) + N(0, \sigma^2)$$

נשתמש ב-experience replay buffer שלוקח בחשבון את (s, a, r, s').
גם כאן יש לנו 4 רשתות, אחת ל-critic, השנייה ל-actor ולכל אחת מהן יש גם את ה-target. העדכונים של ה-target network מתבצעים כל הזמן, באופן רציף.
ה-loss של ה-critic הוא כרגיל (W = המשתנים):

$$\frac{1}{|B|} \sum_{(s,a,r,s') \in B} \left(r + \gamma Q_{target}(s', \pi_{target}(s')) - Q_W(s, a)\right)^2$$

האלגוריתם:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Deterministic Policy Gradient Theorem

תזכורת: Stochastic Policy Gradient Theorem

$$\nabla_\theta J(\theta) = E_{s\sim p^\pi, a\sim\pi(s)}[\nabla_\theta log(\pi_\theta(s,a))Q(s,a)]$$

ה-states וה-actions (לכל state) הם לפי מה שרואים ב-policy π.

- אם היינו רוצים להשתמש ב-samples מ-policy אחרת, היינו צריכים לעשות את הנורמליזציה לפי importance sampling. זה מקשה על השימוש ב-replay buffer - כמו ב-actor critic או ב-REINFORCE.

Deterministic Policy Gradient Theorem

$$\nabla_\theta J(\theta) = E_{s\sim p^\pi}[\nabla_\theta \pi_\theta(s)\nabla_a Q_\pi(s,a)|_{a=\pi(s)}]$$

הפעולות כאן הן דטרמיניסטיות, לכן לא צריך לדאוג ל-actions שהם לפי ה-policy.
זה בעצם נגזר ע"י שימוש בכלל השרשרת על הביטוי $\nabla_\theta Q_\pi(s,\pi_\theta(s))$.

- תזכורת לכלל השרשרת: עבור $h(x) = f(g(x))$ מתקיים $h'(x) = f'(g(x)) \cdot g'(x)$ (כאשר כל הפונקציות מוגדרות וגזירות...)

זה מה שרוצים למקסם - את ה-Q value.

גם אם היינו עובדים עם policy אחרת π' (בוחרים את המצבים לפיה), היה ניתן להשתמש באותם עדכונים (עם π), זה היה $\nabla_\theta J(\theta) \approx$! זה היתרון של ה-DDPG כשעובדים עם משהו דטרמיניסטי ולא סטוכסטי.

זה מה שמאפשר לנו לעבוד עם ה-replay buffer, עם דגימות מ-policies אחרות.

Soft RL

יתרונות של Deterministic Policies מול Stochastic Policies

Deterministic Policies
- קיים תמיד תחת ההנחות המסוימות
- מרחב החיפוש קטן יותר
- יותר נוח לעבוד עם משהו דטרמיניסטי, למשל ב-debug.

Stochastic Policies
- ה-state space רציף ולכן אפשר לעשות עדכונים בקלות יותר.
- סיכוי נמוך יותר ל-overfitting (כי אם הוא למד שיטה מסוימת בסביבה מסוימת, הוא בכל זאת מידי פעם יעשה גם פעולות אחרות), יותר קל להכללה בסביבה חדשה.
- חלק מה-policy שלנו יהיה exploration.
- סיכוי קטן יותר להיתקע ב-local optimum.

הגדרת ה-MDP כך שיעודד policy סטוכסטי:
ניתן reward בהתנהגות סטוכסטית, באמצעות אנטרופיה.
נחבר ל-reward הרגיל reward אחר שיחושב כך: ככל שההסתברות שניקח את הפעולות קרובה יותר ליוניפורמית, כלומר הפעולות יותר סטוכסטיות, נקבל reward יותר גבוה, אחרת נקבל reward יותר נמוך.

$$\hat{R}(s,a) = R(s,a) + \beta H(\pi(\cdot,s))$$

כאשר H היא פונקציית האנטרופיה: $H(\pi(\cdot,s)) = -\sum_{a'} \pi(a',s) log\, \pi(a',s)$

ה-optimal policy הוא ה-policy שממקסם את תוחלת הרווח (הרווח הוא ה-reward הרגיל + ה-reward שמקבלים מהאנטרופיה). $\pi^* = argmax_\pi \sum_i \gamma^i E_{s_n,a_n|\pi} R(s_i,a_i) + \beta H(\pi(\cdot,s))$.

אם עושים את אותה הפעולה כל הזמן, האנטרופיה היא 0 ואז נחזור ל-MDP הרגיל עם policy דטרמיניסטי.
ה-Q function: מניחים שעושים את action a עכשיו ובהמשך פועלים לפי ה-policy שלנו.

$$Q_\pi(s,a) = R(s,a) + \sum_i \gamma^i E_{s_n,a_n|\pi} R(s_i,a_i) + \beta H(\pi(\cdot,s))$$

ה-objective function (פונקציית המטרה, שרוצים למקסם):

$$J(\pi) = \sum_a \pi(s,a) Q_\pi(s,a) + \beta H(\pi(\cdot,s)).$$

איך נחשב זאת?

1. קודם נמצא את ה-greedy policy בהינתן Q (ה-policy שממקסמת את ה-Q).
2. אח"כ נשתמש ב-greedy policy כדי לחשב את ה-V.
3. לבסוף, נבנה את ה-Soft Bellman equation בהסתמך על ה-$v(s)$ שיש לנו:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') v(s').$$

המחובר השני מייצג את ה-reward העתידי, שזה הסיכוי שנעבור לכל אחד מה-states כפול ה-value, שמכיל בתוכו את האנטרופיה.

---

1. $$objective\ function: J(\pi) = \sum_a \pi(s,a)Q_\pi(s,a) + \beta H(\pi(\cdot,s))$$

$$= J(\pi) = \sum_a \pi(s,a)[Q(s,a) - \beta log(\pi(s,a))]$$

רוצים למצוא את ה-policy שממקסמת זאת, לכן נצטרך להשוות את הנגזרות החלקיות ל-0:

- $\frac{\partial J}{\partial \pi(s,a)} = Q(s,a) - \beta \log\big(\pi(s,a)\big) + \pi(s,a)(0 - \beta \frac{1}{\pi(s,a)}) = 0$
- $\log\big(\pi(s,a)\big) = \frac{Q(s,a)}{\beta} - 1$
- $\pi(s,a) = \frac{1}{e}\exp(\frac{Q(s,a)}{\beta})$
- $\pi(s,a) = \frac{\exp\left(\frac{Q(s,a)}{\beta}\right)}{\sum_{a'}\exp(\frac{Q(s,a')}{\beta})} = softmax(\frac{Q(s,a)}{\beta})$

2. $$v(s) = \beta H(\pi(\cdot,s)) + \sum_a \pi(s,a)Q(s,a)$$

ראינו קודם שה-greedy policy היא

$$\pi(s,a) = \frac{\exp\left(\frac{Q(s,a)}{\beta}\right)}{\sum_{a'}\exp(\frac{Q(s,a')}{\beta})}$$

ואז לפי חישוב נקבל:

$\frac{Q(s,a)}{\beta} = \log\left(\pi(s,a) \cdot \sum_{a'}\exp\left(\frac{Q(s,a')}{\beta}\right)\right)$, i.e.:

$Q(s,a) = \beta(\log\big(\pi(s,a)\big) + \log(\sum_{a'}\exp\left(\frac{Q(s,a')}{\beta}\right)))$

- $v(s) = \beta H\big(\pi(s,\cdot)\big) + \sum_a \pi(s,a)(\beta(\log\big(\pi(s,a)\big) + \log(\sum_{a'}\exp\left(\frac{Q(s,a')}{\beta}\right))))$
- $v(s) = \beta H\big(\pi(s,\cdot)\big) + \sum_a \pi(s,a)\beta(\log\big(\pi(s,a)\big) + \log(\sum_{a'}\exp\left(\frac{Q(s,a')}{\beta}\right)))$
- $v(s) = \beta H\big(\pi(s,\cdot)\big)$ =1
  $+ \sum_a(\pi(s,a)\,\beta\log\big(\pi(s,a)\big)) + (\sum_a \pi(s,a))\beta(\log\left(\sum_{a'}\exp\left(\frac{Q(s,a')}{\beta}\right)\right))$
- $v(s) = \beta H\big(\pi(s,\cdot)\big) + \beta\sum_a \pi(s,a)\log\big(\pi(s,a)\big) + \beta\log(\sum_{a'}\exp\left(\frac{Q(s,a')}{\beta}\right))$

  $= -H(\pi(s,\cdot))$
- $v(s) = \beta\log(\sum_{a'}\exp\left(\frac{Q(s,a')}{\beta}\right)) = \underset{a}{contmax}_\beta Q(s,a')$

By definition of continuous max

3. ה-Soft Q value iteration:

$$Q(s,a) = R(s,a) + \gamma\sum_{s'} T(s,a,s')contmax_{a'\,\beta}Q(s',a')$$

זה בדיוק ה-Soft Q value iteration, עליו צריך לחזור שוב ושוב עד התכנסות.

ה-policy שפועלים על פיה היא סטוכסטית: $\pi(s,a) = softmax(\frac{Q(s,a)}{\beta})$.

Soft Q Learning

דומה ל-Soft Q value iteration, רק שבמקום ה-transition $T(s,a,s')$ וה-reward $R(s,a)$ יהיה את ה-sampling שלנו, s' במקום ה-T ו-r במקום ה-R.

כלומר אנו מעדכנים את $Q(s,a)$ לקראת: $r + \gamma\, contmax_{a'\,\beta}Q(s',a')$

ה-policy הסופית: $\pi(s,a) = softmax(\frac{Q(s,a)}{\beta})$.

Soft Actor Critic

בפועל משתמשים בו יותר (הוא גם יותר טוב).

- דוגמים פעולות לפי ה-actor הסטוכסטי π_θ
- מעדכנים את ה-critic לפי $r + \gamma Q(s',a') + \beta H(\pi(\cdot,s))$
- מעדכנים את ה-actor לפי $softmax(\frac{Q(s,a)}{\beta})$

בעצם מנסים למזער את ה-KL-divergence בין ה-policy הנוכחית ל-policy הרצויה, כלומר:

$$\theta \leftarrow \theta - \alpha\frac{\partial KL(\pi_\theta||softmax(\frac{Q(s,a)}{\beta}))}{\partial\theta}$$

כאשר: $KL(Q||P) = \sum_a Q(a)\, log(\frac{Q(a)}{p(a)})$

PPO (Proximal Policy Optimization)

זהו policy based RL method.

הרעיון: נרצה להישאר כמה שיותר קרובים ל-policy, ובמקום להשתמש ב-return נשתמש ב-advance.

<u>תזכורת</u>: ב-REINFORCE אם היו כמה returns חיוביים, תמיד עדכנו בצורה חיובית, אפילו אם ה-action היה נמוך יותר ממשהו שכבר ראינו (שזה לא הגיוני לשאוף אליו יותר).

לכן כדי לפתור את זה, במקום לעדכן לקראת ה-return נעדכן לקראת ה-**advance**, שפירושו ההתקדמות/היתרון שיש לנו כאשר עושים action מסוים על פני כמה שאמורים להרוויח באופן כללי ב-state הזה.

הנוסחה: $A_\pi(s,a) = Q_\pi(s,a) - v_\pi(s)$. ואז עבור a מסוים, נבדוק אם הערך הזה חיובי או שלילי, ואז נדע האם להגדיל או להקטין את הסיכויים לבחור בו. אבל נשים לב שבניגוד לקודם, עכשיו גם עבור ערכים חיוביים נוכל לקבל A_π שלילי, כי ראינו משהו טוב יותר בעבר.

- אם ה-policy שיש לנו כבר הוא אופטימלי, ה-A_π תמיד יהיה ≤ 0.

בנוסף, PPO מגביל את המרחק שניתן להתרחק מה-policy שיש לנו. זה מאפשר לנו להשתמש בדוגמאות מה-policy לפני כמה צעדים.

משתמש גם ב-KL divergence כדי לא להתרחק יותר מידי מה-policy שיש לנו.

Inverse Reinforcement Learning

בניגוד ל-Reinforcement Learning רגיל, בו רוצים למצוא את ה-policy, כאן נתונה ה-policy (או כל מיני פעולות שנעשו מ-policy מסוימת), ורוצים למצוא את ה-reward function שמשרה אותם (למשל אנשים שעושים כל מיני דברים, מה הם מנסים למקסם?).
נשים לב שלא נגדיר עבור פעולה מסוימת reward של 0, כדי להימנע מפתרונות טריוויאלים (של פשוט לא לעשות כלום).

Upper Confidence Bound (UCB)

קשור ל-exploration vs exploitation.

הרעיון הוא שאנו אופטימיים לגבי דברים שעוד לא ראינו (בדומה ל-A^*). האופטימיות מתבטאת בכך שאלו שלא ראינו, שיש לנו פחות מידע עליהם, נבחר אותם בהסתברות גבוהה יותר, ואז יש לנו built in exploration.
נשתמש ב-Upper Confidence Bound, כלומר זוג ה-(state, action) שיש לו את ה-Confidence Bound הכי גבוה, למשל כאן זה Action 4:

- הערה: מהו confidence bound x%? זה אומר שאנו בטוחים ב-x אחוז שהממוצע נמצא בטווח מסוים.

כמו כן, נרצה שה-Confidence Bounds יקטנו עם הזמן כך שירדו מתחת לאיזשהו ערך, שהוא t^{-4} עבור כל זמן t. הסיבה היא שאנו רוצים שה-exploration יקטן עם הזמן (כי ראינו ב-$\epsilon - greedy$ שאם נמשיך איתו לנצח, לא נגיע ל-policy האופטימלית).
ככל שיש פחות פעולות, הוריאנס קטן יותר, ואז אנו יותר בטוחים איפה נמצא הממוצע.

תזכורת: Hoeffding's Inequality

יהיו $X_1,..., X_n$ משתנים מקריים ב"ת שנדגמו בצורה אקראית מ-[0,1] (אבל אפשר לעשות scaling לכל התפלגות חסומה). יהי $\bar{X}$ ממוצע הדגימות. אזי לכל $0 \le u \le 1$: $P(E[X] > \bar{X} + u) \le e^{-2nu^2}$.

- אפשר להשוות את ה-e^{-2nu^2} ל- t^{-4} שרצינו וכדומה.
- אצלנו: $P(Q_\pi(s,a) > \overline{Q(s,a)} + u) \le e^{-2nu^2}$

כאשר פותרים את המשוואה $t^{-4} = e^{-2nu^2}$ מקבלים: $u = \sqrt{\frac{-4logt}{-2n}} = \sqrt{\frac{logt}{2n}}$

לכן אנו בוחרים פעולות לפי $argmax_{a'\in A}(Q(s,a') + \sqrt{\frac{logN(s)}{2N(s,a')}})$, כאשר $N(s,a')$ הוא מספר הפעמים שפעולה a' בוצעה ב-s state, ו-$N(s)$ הוא מספר הפעמים שביקרנו ב-s.

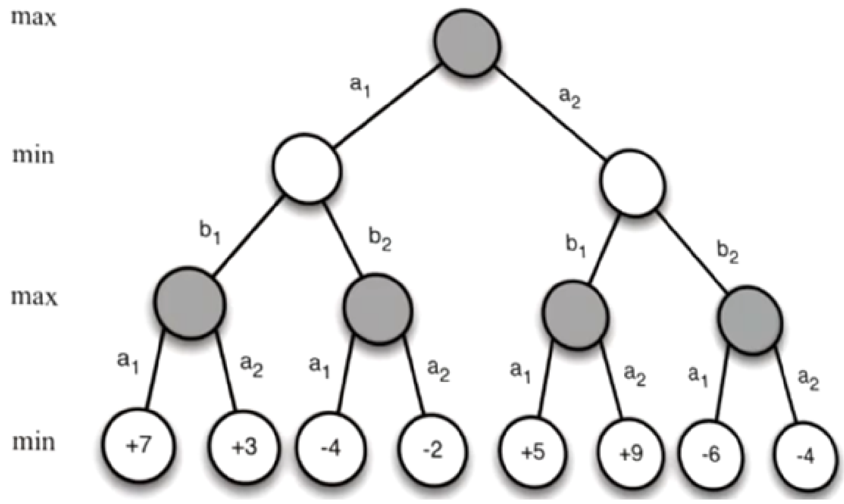
שיעור 6 - 23.8 (אחרון)

מצגת 6 - RL Algorithms for 2-Player Games

ה-setting שלנו:
- משחקי zero sum - תמיד יש צד שמרוויח וצד שמפסיד.
- שני שחקנים
- משחקי perfect information - כל שחקן יודע את כל המידע גם על עצמו וגם על השני (**לא** משחקי קלפים שלא יודעים מה יש לשחקן השני ביד, כמו פוקר למשל).

רעיונות מרכזיים:
- planning - מסתכלים קדימה על מנת לבחור איזו פעולה לעשות
  בד"כ הפעולות יהיו דטרמיניסטיות, כשבוחרים לעשות משהו זה באמת מה שיקרה, לא סטוכסטי.
- שיפור באמצעות משחק עם עצמנו - משתפרים תוך כדי עדכון המשקולות.

תזכורת: MinMax
בהינתן שני שחקנים- max, שמטרתו למקסם את התועלת (שכתובה בעלים) ו-min, שמטרתו למזער את התועלת. עץ המשחק נתון בצורה הבאה:



באלגוריתם ה-Minmax עוברים מלמטה למעלה, ובכל עלה בוחרים את הערך שממקסם או ממזער את הערך, בהתאם לשחקן שזה התור שלו. כך יש לנו את כל הפעולות ששחקן רציונלי יעשה בכל נקודה.

אבל במשחק ארוך (למשל שחמט) יהיה לנו branching factor - בכל תור עץ המשחק יגדל יותר ויותר ויהיו יותר מידי פעולות לשמור בזכרון שלנו ולעבור עליהן.
הפתרון הוא לתת **שערוך** ללוח (בהינתן לוח מסוים, אומרים כמה אנו מעריכים שהוא שווה).

- יש גם אלגוריתם דומה בשם $\alpha - \beta$ $pruning$ שעוזר בחיתוך הענפים וכך מונע מעבר על כל העץ.
  זה עוזר להסתכל בעומק כפול ממה שאפשר ב-minmax, אך עדיין במשחקים מורכבים זה יהיה גדול מידי.

ערכי העלים
ישנה פונקציה (נאמדת או מוגדרת ידנית) שמקבלת מצב עולם כלשהו ואומרת עד כמה הוא טוב. מכיוון שהנחנו zero sum game, היא צריכה להחזיר רק ערך אחד.

בשחמט: בשנת 1997 בינה מלאכותית בשם Deep Blue הצליחה לנצח את אלוף העולם. היא הייתה מבוססת על כ-8,000 פיצ'רים שמסתכלים על כל מיני דברים בלוח ולפי זה נותנים הערכה עד כמה הוא טוב. אין כאן למידה. (לדוגמה: מלכה = 9 נקודות, שליטה ב-4 הנקודות המרכזיות = 0.1 נקודות וכד'...). ה-Deep Blue הסתכל קדימה, עד עומק של כ-40 מהלכים ומליוני מצבים (שזה כמובן הרבה מעבר למה שבן אדם יכול להסתכל עליו). הוא מבוסס על אלגוריתם ה-Minmax, מסתכל כל פעם על כל האפשרויות.

בדמקה: הצליחו לנצח ברעיון דומה בשנת 2007. perfect play - אפשר לנצח מההתחלה עד הסוף בצורה מושלמת. אם מתחילים תמיד מנצחים. הרעיון היה גם הסתכלות קדימה, אבל גם בנו database שכולל את כל מצבי העולם שאם מגיעים אליהם תמיד מנצחים.

אנו רוצים לשפר את ה-Value Function כל הזמן, במקום לבנות אותו ידנית.

שימוש ב-RL Value Function
הרעיון המרכזי ב-RL היה שבהינתן ה-value של ה-state הבא עדכנו את ה-value של ה-state הנוכחי כך: $v(s) = r + \gamma v(s')$. או ב-Q learning העדכון היה: $Q(s, a) = r + \gamma max Q(s', a')$.
כאשר מדברים על deterministic 2 player zero sum games ניתן להשתמש ב-$value\ function$ ולא ב-$Q\ function$, כי אנחנו כן יודעים איזו פעולה תוביל אותנו לאיזה state (כלומר יודעים את ה-transition function).
דבר נוסף, במשחקים כאלו ה-reward לאורך כל המשחק הוא 0 ורק בסוף יש לנו 1 (ניצחון) או 1- (הפסד).
כמו כן, מכיוון שלכל משחק יש את החוקים שלו כדי למנוע מצב שנמשך לעולם והמשחק לא מסתיים, ניתן לקבוע את $\gamma = 1$.
כך יוצא לנו שהנוסחה הפשוטה יותר לעדכון היא $v(s) = v(s')$.

כדי לעדכן את הערכים של העלים שיהיו כמה שיותר מדויקים אפשר לקחת הרבה משחקים ולעדכן כך:
- MC updates: מונטה קרלו. רצים עד הסוף ומעדכנים את הטעות לאחור.
- TD updates: מאתחלים את הכל באפסים, מריצים הרבה משחקים ואז יש נצחונות והפסדים ומעדכנים את ה-value הנוכחי מה-value הבא. ה-values מיוצגים ע"י פונקציה. אפשר לייצג כל מצב למשל כתמונה, ואז להשתמש ברשת CNN שבהינתן מצב מחזירה את הערך שלו. לצורך עדכון המשקולות נשתמש ב-gradient descent וב-MSE כדי לראות מה הטעות.
- $TD(\lambda)$ updates : עדכון ה-value הנוכחי מהמצבים הבאים, כאשר משתמשים ב-λ.

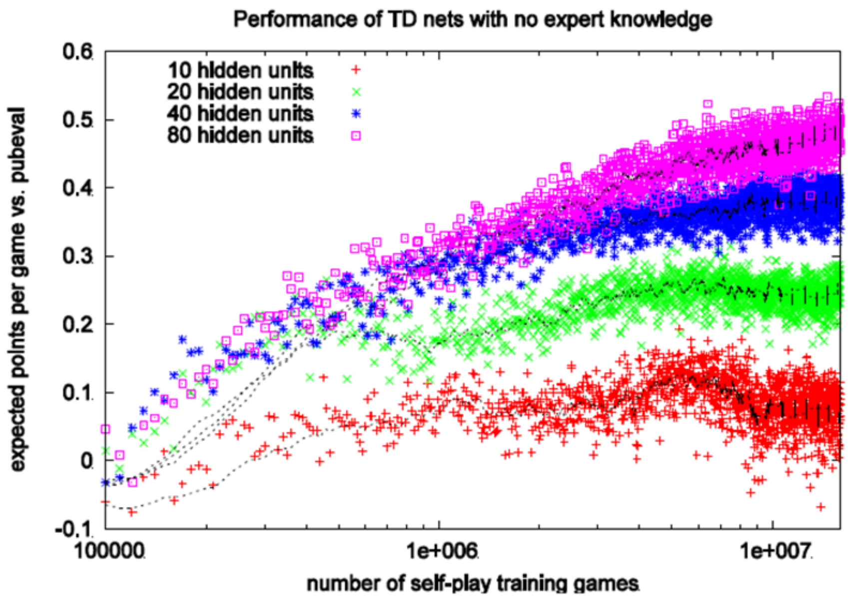
TD - Gammon

גם כאן רוצים למקסם את תוחלת הרווח. ההבדל הוא שכאן יש שימוש ב-Expected Minimax במקום ב-Minmax רגיל:
- מסתכלים על כל הקומבינציות שהקוביות יכולות לקבל.
- לא מניחים שהעולם הוא לרעתנו או לטובתנו, אלא מסתכלים על כל האפשרויות.
- זו גם הסיבה שאי אפשר להסתכל הרבה קדימה - פועלים לפי $2 - play\ lookahead$.

הפיצ'רים הם מספר החיילים בכל משבצת.
משתמשים ברשת נוירונים עם שכבה נסתרת אחת.
העדכון של המצב הקודם בו היינו הוא על סמך המצב הנוכחי.

אין exploration. הסיבה שזה עבד והתכנס עבור משחק השש בש היא בזכות הקוביות (במשחקים אחרים השיטה הזו לא עבדה).

תוצאות:



אפשר במקום לעדכן לקראת הערך הבא (כמו שעשינו כאן), לעדכן באחת מהדרכים הבאות:

- TD-Root: לעדכן לקראת ה-supporting node, שהוא הנוד האחרון שהגענו אליו בחיפוש (ערך אמיתי בלוח).
- Tree-Strap: לא מעדכנים רק את הנוד בו אנו נמצאים, אלא את כל הנודים שביקרנו בהם. זה בעייתי אם משתמשים ב-$\alpha - \beta$ $pruning$ כי אז לא תמיד נדע את הערך האמיתי של הנוד.

The Game of GO

משחק סיני עתיק בו יש לוח בגודל 19x19. בכל פעם שחקן שם אבן שלו (שחורה או לבנה), כאשר המטרה היא להקיף את הצד השני בארבעת המשבצות הסמוכות אליו, וכך לאכול אותו.

באותה תקופה כשהתחילה הבינה המלאכותית, אמרו שמחשבים ייחשבו חכמים מאוד במידה וינצחו את אלוף העולם בשחמט. ברגע ש-Deep Blue ניצח את Kasparov אמרו שזו לא חוכמה, כי המחשב לא למד לשחק שחמט, אלא למד להסתכל על כל האפשרויות ולבחור באפשרות הכי טובה ולכן זה לא מעיד על חוכמה.
לאחר מכן אמרו שמה שכן יעיד על חוכמה זה נצחון ב-GO ששם אי אפשר להסתכל על כל המהלכים קדימה כי מספר המהלכים עצום (כמעט 400 בכל פעם) ויש branching factor שלא מאפשר זאת.
כלומר, GO לאורך הרבה זמן היה המטרה הבאה אחרי שחמט.

<u>AlphaGO</u>

מבוסס על actor critic במובן שיש policy networks ו-value network.

הבנייה הייתה באמצעות שני שלבים:

1) איסוף dataset של 30,000,000 מצבים ממשחקים של אלופי עולם ב-GO. למדו מכך.
2) נתנו למחשב לשחק נגד עצמו, וגם מכך למדו משהו.

חלק מהפיצ'רים היו hand-crafted, בעזרתו של אחד מאלופי העולם שלימד אותם כל מיני טכניקות משחק.
בלימוד הם התעלמו מ-rotations, כלומר לוח מסובב כי זה אותו משחק, וכך חסכו.
זה היה מבוסס על MC Tree Search.

ספסיפיקציה של הרשת עצמה:

ישנן 3 policy networks (תכף נראה איך משיגים אותן): p_π, p_σ, p_ρ.

הרשתות מקבלות state מסוים (לוח) ופולטות מה הסיכוי לעשות כל אחד מה-actions. זה איזשהו softmax.
כמו כן יש את רשת ה- value v_w שמקבלת state ונותנת לו ערך $\in [-1, 1]$.

אחוזי דיוק: כאשר אומרים שלרשת יש x אחוז דיוק זה אומר שכשמנסים לנבא מה השחקן יעשה, ב-x אחוז מהפעמים הפעולה שנתנו לה את הערך הכי גבוה היא באמת הפעולה שהשחקן עשה.

p_π: Rollout Policy

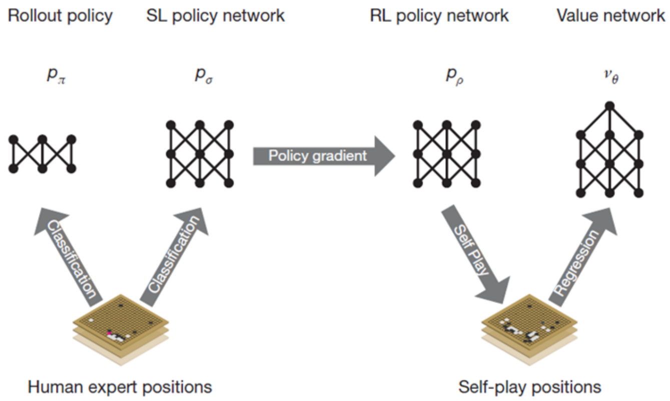
- משתמשים בה ל-rollout (ריצה עד הסוף על מנת לקבל איזשהו ערך - עושים זאת כחלק מ-MC Tree Search) עד סוף המשחק.
- יש לו 24% דיוק.
- מאוד מהיר ($2\ \mu s$) כי הוא מבוסס softmax.

p_σ: SL Policy Network

- משתמשים בה כדי לבחור את הענף שניכנס אליו ב-MC Tree Search.
- 57% דיוק. הרשת עמוקה יותר מ-p_π לכן ברור שהיא יותר טובה.
- 17 שכבות, כמו p_π אבל כבדה יותר ולכן איטית יותר ($3\ ms$).

p_ρ: RL Policy Network

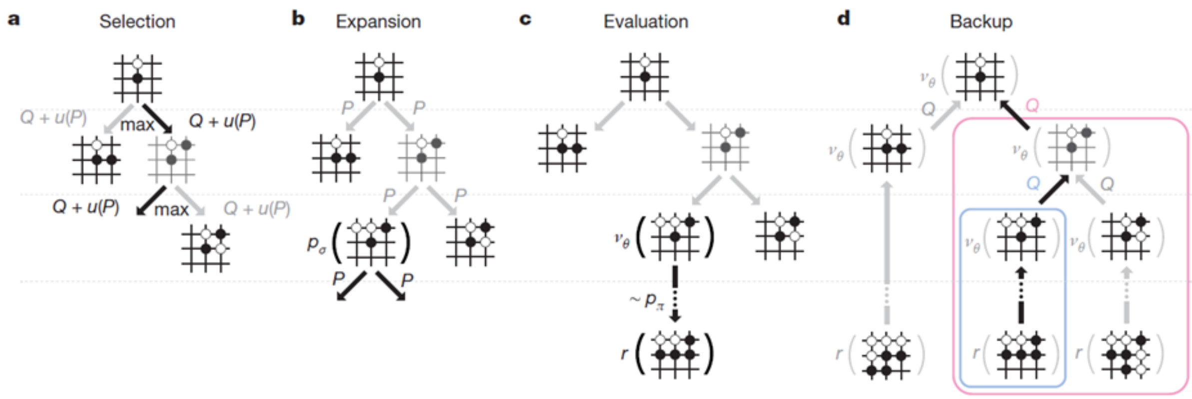
- מאותחלת מה-p_σ
- שיחקה נגד עצמה הרבה מאוד פעמים, והשתפרה ע"י שימוש ב-REINFORCE.
- לא משתמשים ברשת הזו בפועל (במשחק עצמו), אלא השימוש שלה הוא ייצור database כדי לאמן את ה-value network, ע"י supervised learning.
- הרשת הזו ניצחה ב-80% מהמשחקים נגד p_σ.



לסיכום:
באמצעות שימוש ב-database של 30,000,000 מצבי עולם במשחק GO של בני אדם ו-supervised learning רגיל, מנבאים מה השחקן יעשה במצב מסוים.
לצורך כך יש לנו שתי רשתות: p_π ו-p_σ כאשר ההבדל ביניהם הוא ש- p_π מאוד דלילה ומכילה softmax על כל אחד מהפיצ'רים, והשנייה עמוקה ולכן טובה יותר.
צריכים את שתי הרשתות כי כאשר רוצים לרוץ עד סוף המשחק צריך משהו מאוד מהיר, לכן p_π מתאימה, ואילו ב-p_σ משתמשים רק בתוך העץ שלנו (שהוא לא ממש עמוק).
לאחר מכן מאתחלים את p_ρ עם p_σ, ונותנים לו לשחק נגד עצמו ולהשתפר באמצעות REINFORCE.
בסוף בנו dataset נוסף של 30,000,000 מצבים, אך הפעם הם מצבים ממשחקים שונים (ולא מאותם משחקים כמו ה-dataset הקודם). זה גורם לשיפור משמעותי ב-value network כי ה-dataset מגוון ועשיר הרבה יותר.

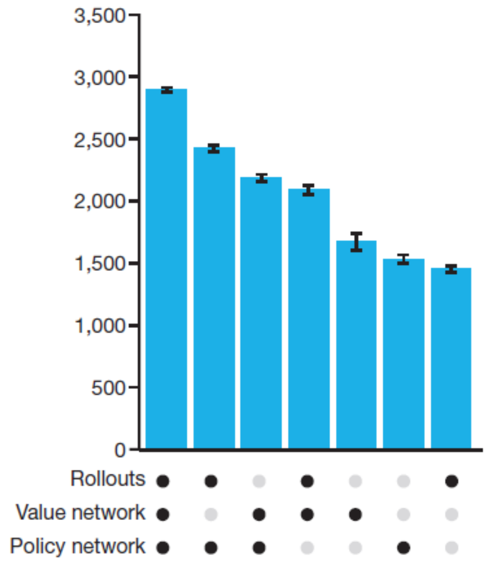
- בהתחלה ניסו לאמן את ה-value network על ה-p_σ אבל זה לא עבד כ"כ טוב.
- לא היה ניתן לצפות שלפי p_σ נהיה טובים יותר מאנשים, כי אנו לומדים מהתנהגות אנושית ולכן במקרה הטוב נחקה אותם, אבל לא ננצח אותם.

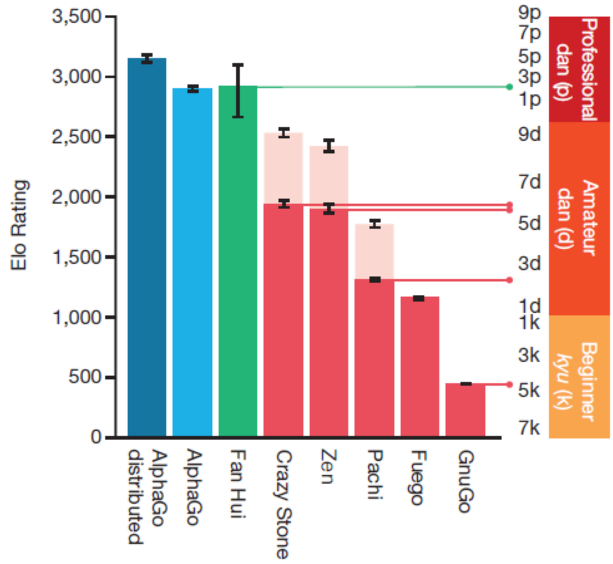
Monte Carlo Tree Search (ב-AlphaGO)



<u>Selection</u>: מבוסס על סטטיסטיקות שאספנו עד עכשיו (כולל אינפורמציה מהמצבים שפתחנו ברשת) ועל p_σ.
כלומר בוחרים לפי מה שממקסם את p_σ ועוד איזשהו ערך שתלוי בסטטיסטיקות. בסופו של דבר לאחר שבנינו את העץ כולו, נבחר בפעולה שבחרנו בה הכי הרבה פעמים. מכיוון שתמיד בחרנו בפעולה שממקסמת את $Q + u(P)$, זה אומר שהיא הכי טובה.
ספציפית: החישוב דומה ל-UCB במובן שככל שביצענו פעולה יותר פעמים, הבונוס שהיא מקבלת קטן, ולהפך. זה ה-exploration. הבונוס הזה מסומן ב-$u(P)$: $u(s,a) \propto \frac{p_\sigma(a|s)}{1+N(s,a)}$ כאשר $N(s,a)$ הוא מספר הפעמים שביצענו את הפעולה הזו ב-state הזה.
ה-$Q(s,a)$ הוא הסטטיסטיקות שיש לנו בתוך העץ. הוא עושה ממוצע בין שני דברים - $v(s_i)$ של הצאצאים של s בעץ, ובין התוצאות של ה-MC rollouts.
לבסוף הערך הסופי שמייצג את הציון עבור הבחירה בתוך העץ הוא: $a_t = argmax_a(Q(s_t,a) + u(s_t,a))$.
<u>Expansion</u>: ברגע שמגיעים לנוד חדש, מעדכנים את ה-p לכל ה-actions ואז עושים rollout (רצים עד הסוף) עם ה-p_π. כלומר משתמשים ב-policy הזו לשני השחקנים, הסוכן משחק נגד עצמו עד הסוף ואז רואים מי ניצח ומעדכנים את המצבים אחורה בהתאם (עד השורש).

התוצאות של AlphaGO:





<u>AlphaGO Zero</u>

שיפור של AlphaGO. הוא לא השתמש בשום ידע קודם, לא בידע אנושי ולא ב-AlphaGO הראשון.

- ה-AlphaGO היה סוכן שידע לשחק טוב ולכן עזר להם לבדוק את עצמם ולהבין באילו פרמטרים לבחור וכו. כלומר הוא בכל זאת עזר להם קצת לבנות את AlphaGO Zero, אך על פי שעל הנייר הוא לא משתמש ב-AlphaGO.

לכן מה שירד כאן זה: האימון על ה-human data, הפיצ'רים שהוגדרו ע"י שחקן GO אלוף (אנושי).
בגלל שלא משתמשים ב-hand crafted features כאלו, ניתן להשתמש ב-AlphaGO Zero גם על משחקים אחרים.
מה שיש כאן זו רשת נוירונים יחידה שבהינתן state יש לה שני outputs: הניבוי של ה-value ומטריצה של ה-policy עצמו (שמייצגת את הסיכוי לעשות כל אחד מה-actions).
ה-AlphaGO Zero משחק נגד עצמו ומשתפר, נגד הגרסה הכי טובה של עצמו. הוא עוצר כאשר הוא מצליח לנצח את השחקן הכי טוב ב-55% מהפעמים, ואז הוא הופך לשחקן הכי טוב, וכן הלאה..

השלבים:

- מריצים MC tree search ללא MC rollouts.
- משתמשים באותו UCB based simulated action selection.
- בכל פעם שפעולה נבחרת (ע"י MCTS), מעדכנים את ה-policy network לקראת ה-action הזה, כלומר כעת יהיה סיכוי גבוה יותר לבחור בו (כי אם בחרנו בו אחרי כל החישובים הוא טוב).
- בכל פעם שמשחק (אמיתי) מסתיים, מעדכנים את הערכים (ב-value function) של כל ה-states במשחק הזה.

הטענה/ המסקנה שלהם:

כאשר לא משתמשים ב-human knowledge ניתן להגיע לתוצאות טובות יותר, כי לא מקבעים את עצמנו. זו שאלה. כמו כן, מקבלים משהו כללי יותר כי לא מסתמכים על dataset של משחק מסוים.

<u>AlphaZero</u>

כמעט זהה ל-AlphaGO Zero רק שהפעם הסוכן ישחק גם GO, גם שחמט וגם שוגי (מעין שחמט יפני).

- ה-input עבור כל אחד מהמשחקים הוא שונה. גם מבחינת הניצחון, ב-GO היה רק 1 או מינוס 1 (ניצחון או הפסד), ובשחמט ושוגי לעומת זאת יש גם 0 (תיקו).

- לא השתמשו ב-data augmentation, כלומר שמתייחסים למצבים של לוח מסובב כאותו מצב, כי בשחמט זה כבר לא נכון.
- דבר נוסף שהם שינו: במקום לשחק כל פעם מול ה-best player משחקים כל פעם מול השחקן האחרון.
- הדבר האחרון הוא לגבי עדכון ה-hyperparameters, שעודכנו כאן באמצעות איזשהו hard code.

התוצאות היו טובות:

	Win	**Draw**	**Lose**
Chess vs. 2016 TCEC world champion *Stockfish*	28	72	0
Shogi vs. 2017 CSA world champion *Elmo*	90	2	8
Go AlphaZero vs. AlphaGo Zero (3 days)	60		40

* נשים לב שבשחמט היו הרבה תיקוים, כי זה קורה כאשר השחקנים טובים מאוד. (אבל לא היו הפסדים).

AlphaZero גילה עם הזמן מהלכים שאנשים נוטים לשחק בהם (למשל פתיחות ידועות). לפעמים החליט להמשיך להשתמש בהם ולפעמים החליט לזנוח אותם.

<u>MuZero</u>
הרעיון הוא שיעבוד גם על משחקי Atari, כמו משחק ה-pong, פקמן, מטקות וכו. רוצים שזה יהיה באלגוריתם אחד שיפתור גם את GO, שחמט ושוגי. משחקי Atari הם משחקים שאינם two-player games, אלא הצד השני הוא חלק מהסביבה.
ב-Atari יש סה"כ 57 משחקים, בחלקם מחשבים מצליחים ממש טוב ובחלקם לא.
מדדים להצלחה:

1. ממוצע הניקוד בכל המשחקים.
2. חציון.
3. בכמה משחקים מצליחים טוב יותר מבני אדם.

השוני העיקרי בין משחקי Atari למשחקים כמו GO ושחמט הוא שאין את ה-transition function. מכיוון שאנחנו רוצים שהכל ייפתר באלגוריתם אחד, נתעלם מהידע שלנו על ה-T של GO ושחמט, ונלמד אותו תוך כדי.
דבר נוסף הוא ה-reward, שעכשיו זה לא רק ניצחון, הפסד או תיקו, אלא יש נקודות שנצברות במהלך המשחק, ולכן צריך להחזיר את ה-discount factor, γ. גם כאן נניח שמקבלים את ה-reward רק בסוף.

מנבאים את T ואת R באמצעות שתי רשתות.
את הפרדיקציה נעשה ב-lower dimension. כלומר ניקח את ה-input שלנו (המסך או לוח המשחק וכו'), נהפוך אותו לאיזשהו embedding קטן יותר, ונשתמש בו כדי לנבא את ה-embedding של ה-state הבא.

יש לנו 3 רשתות:

1. H - רשת שהופכת את המצב (תמונה) לוקטור במימד נמוך יותר.
2. G - רשת שמקבלת את הוקטור ומנבאת את המצב הבא (במימד נמוך גם כן) ואת ה-reward הבא.
3. F - רשת שמקבלת state ומנבאת את ה-policy ואת ה-value (הרשת הרגילה שלנו).

התוצאות על כל המשחקים:

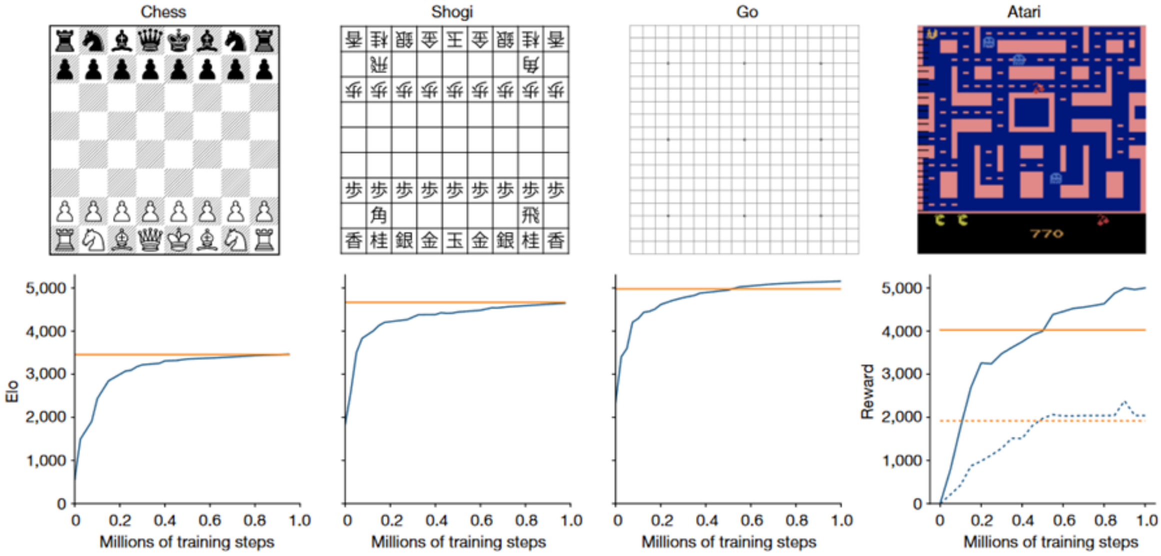


Fig. 2 | Evaluation of MuZero throughout training in chess, shogi, Go and Atari. The *x* axis shows millions of training steps. For chess, shogi and Go, the *y* axis shows Elo rating, established by playing games against AlphaZero using 800 simulations per move for both players. MuZero's Elo is indicated by the blue line and AlphaZero's Elo is indicated by the horizontal orange line. For Atari, mean (full line) and median (dashed line) human normalized scores across all 57 games are shown on the *y* axis. The scores for R2D2[19] (the previous state of the art in this domain, based on model-free RL) are indicated by the horizontal orange lines. Performance in Atari was evaluated using 50 simulations every fourth time step, and then repeating the chosen action four times, as in previous work[39]. Supplementary Fig. 1 studies the repeatability of training in Atari.