Hello, our names are Alon Biner, Mariia Makarenko and Yonatan Vologdin, and today we will present our project for the course Introduction to Artificial Intelligence. In our project we tackle the challenge of solving the game of Jenga using various AI algorithms.

First, we will describe the rules of Jenga for those not familiar with the game. In the original version of the game there are 54 wooden blocks. To start the game, these blocks are stacked in levels of three on top of each other, so that the levels alternate direction. So, in total there are 18 levels in the initial tower.

Show physically

After building the tower, the players make moves in turn, and the person who stacked the tower plays first. Every move consists of taking one block from any level of the tower (except the one below an incomplete top level) and placing it on the topmost level to complete it. Players can touch the tower with only one hand at any time. The game ends if either the tower falls either completely, or if any block falls from the tower, and the loser is the player who ended the game.

Show physically

The Jenga game is challenging for the AI player, because in order to win the player needs to make a move that balances out the risk of the tower falling: in needs to be safe enough for the tower not to fall in this move, but risky enough to increase the chances of the tower falling during the opponent's move.

One of the main challenges for developers of Jenga AI solver is the physical nature of the game. That is, simple inaccuracy in pulling out blocks can make the player lose, independently of how good their strategy actually is. Therefore, we made a few key assumptions and relaxations.

First, we assumed that the blocks are removed perfectly. That is, they just vanish from their places without touching the other blocks. Second, we relaxed the problem to the total of 12 levels and removed adding the blocks back on top of the tower after pulling them out. And there were also other assumptions that we will discuss a bit later.

Using all these assumptions, we created an environment in Unity, so that the Unity physics engine controls the positions of the blocks after each move. The architecture is as follows: our AI agents take information from the python environment object, which in turn sends and receives data to and from Unity via TCP. When the agent wants to make a move, it selects the level and the block at that level, sends a command via the environment object, the environment object sends the command to Unity, and the block is removed.

All the AI agents get rewards for their actions, so that the rewards are computed in the same way for all of them. And they try to make actions that maximize the reward to improve their performance. Since we want our agent to make the life harder for its opponent, we give bigger rewards for actions that we assume decrease the stability of the tower for the opponent but aren't very risky for the player themselves. The lower the level of the block pulled out, the better, and the more the tower was swinging during the move the better, if the tower didn't fall in the process. To calculate the swinging, we compute the maximum average incline of the blocks from the center of mass of the tower after the move. If the tower fell, we give to that action a large penalty.

Show physically

As per the algorithms, we tried three different solutions. The first solution is using the algorithm that we called Greedy Simulation-Based Action Search (GSBAS), which from any step in the game basically tries out different possible actions, then reverts back to that initial step and chooses the one that gives the better reward. Basically this agent knows the future. However, we thought that it was unfair for the other player that the greedy player knows in advance during which actions the tower falls and avoids them.

The other two solutions are more fair and they estimate how good the actions are during the training process rather than in the game itself. The first such solution is Hierarchical Deep Q Network. This algorithm uses a neural network to estimate the Q-values of actions. It is hierarchical because every action is a combination of two: level and block in level, and each network knows how to deal only with one part. So in essence it's a combination of two neural networks.

The second solution is similar to the first one and is called SARSA (State Action Reward State Action). It updates its Q-values based on the action the agent *actually* takes, according to its current policy. SARSA's update rule considers the action that the agent follows in practice, which includes exploratory actions. This leads to safer exploration. On the other hand, DQN uses a target policy, which is greedy, to update Q-values regardless of the action the agent actually took. It uses the maximum Q-value from the next state (i.e., the best possible action in the next state) for the update, even if that action wasn't the one taken in practice.

We chose to represent the Jenga states as a matrix loaded from the picture that is a two-sided "scan" of the Jenga tower.

Show physically

When we trained our algorithms, we created a menu in Unity that allows the user to choose the algorithms to compete against each other and watch how they perform. It is also

possible for the human player to play against one of the algorithms, and two humans to play against one another.

We hope that you will try the game out and tell us what you think. Thank you for your attention!

שלום, קוראים לנו אלון בינר, מריה מקרנקו ויונתן וולוגדין, והיום נציג את הפרויקט שלנו לקורס מבוא לבינה מלאכותית. בפרויקט שלנו אנו מתמודדים עם האתגר של פתרון המשחק של Jenga באמצעות אלגוריתמים שונים של AI.

ראשית, נתאר את חוקי Jenga למי שלא מכיר את המשחק. בגרסה המקורית של המשחק יש 54 לבנות עץ. כדי להתחיל את המשחק, הבלוקים האלה מוערמים ברמות של שלוש זה על גבי זה, כך שהרמות מחליפות כיוון. אז בסך הכל יש 18 רמות במגדל הראשוני.

להראות פיזית

לאחר בניית המגדל, השחקנים מבצעים מהלכים בתורם, והאדם שערם את המגדל משחק ראשון. כל מהלך מורכב מלקחת בלוק אחד מרמה כלשהי של המגדל ולהציב אותו ברמה העליונה. שחקנים יכולים לגעת במגדל עם יד אחת בלבד. המשחק מסתיים אם המגדל נופל לגמרי, או אם בלוק כלשהו נופל מהמגדל, והמפסיד הוא השחקן שסיים את המשחק.

להראות פיזית

משחק ג'נגה מאתגר לשחקן הבינה המלאכותית, מכיוון שכדי לנצח השחקן צריך לבצע מהלך שמאזן את הסיכון של נפילת המגדל: צריך להיות בטוח מספיק כדי שהמגדל לא ייפול במהלך הזה, אבל מסוכן מספיק כדי להגדיל את הסיכוי שהמגדל ייפול במהלך של היריב.

אחד האתגרים העיקריים בלפתח פותר ג'נגה הוא האופי הפיזי של המשחק. כלומר, חוסר דיוק פשוט בשליפת בלוקים יכול לגרום לשחקן להפסיד, ללא תלות במידת היעילות של האסטרטגיה שלו. לכן, הנחנו כמה הנחות מקלות.

ראשית, הנחנו שניתן להסיר את הבלוקים בצורה מושלמת. כלומר, הם פשוט נעלמים ממקומותיהם מבלי לגעת בבלוקים האחרים. שנית, הקטנו את המגדל ל- 12 קומות וויתרנו על החזרת הבלוקים בחזרה על המגדל. והיו גם הנחות נוספות שנדון בהן קצת בהמשך. בנוסף הגבלנו את מספר השחקנים לשניים.

באמצעות כל ההנחות הללו, יצרנו סביבה ב Unity,-כך שמנוע הפיזיקה של Unity שולט על מיקומי הבלוקים בכל מהלך. הארכיטקטורה היא כדלקמן: סוכני ה AI-שלנו לוקחים מידע מאובייקט סביבת הפיתון, שבתורו שולח ומקבל נתונים אל Unity וממנה באמצעות TCP. כאשר הסוכן רוצה לבצע מהלך, הוא בוחר את הרמה ואת הבלוק ברמה זו, שולח פקודה דרך אובייקט הסביבה, אובייקט הסביבה שולח את הפקודה ל Unity,-ופעולת הוצאת הבלוק מתבצעת על ידי המנוע של יוניטי.

כל סוכני ה AI-מקבלים תגמולים על מעשיהם, כך שהתגמולים מחושבים באותו אופן עבור כולם. והם מנסים לבצע פעולות שממקסמות את התגמול כדי לשפר את הביצועים שלהם. מכיוון שאנו רוצים שהסוכן

שלנו יעשה את החיים קשים יותר ליריבו, אנו נותנים תגמולים גדולים יותר עבור פעולות שאנו מניחים שמפחיתות את יציבות המגדל עבור היריב, אך אינן מסוכנות במיוחד עבור השחקן עצמו. עדיף לשחקן לשלוף בלוק מקומה נמוכה יותר ככל שניתן מבלי להפיל אותו, ושהמגדל יתנדנד כמה שפחות במהלך הפעולה כדי שהמהלך עדיין יהיה מספיק זהיר. כדי לחשב את סטיית המגדל, אנו מחשבים את השיפוע הממוצע המרבי בציר עלרוד או גלגול של הבלוקים של המגדל לאחר המהלך בהשוואה למצב ההתחלתי.

להראות פיזית

עכשיו נדון על האלגוריתמים. ניסינו שלושה פתרונות שונים. בפתרון הראשון השתמשנו באלגוריתם שקראנו לו Greedy Simulation-Based Action Search שבכל שלב במשחק מנסה לדעת מההפעולה המשתלמת העתידית ומנסה פעולות אפשרויות שונות לצורך זה. אחרי שהוא ניסה כמה פעולות הוא למעשה יודע את העתיד ויודע אילו פעולות מפילות את המגדל בצעד הבא ואילו לא, ואיזו פעולה גורמת לסטייה גדולה יותר. חשבנו שהאלגוריתם הזה לא הוגן כלפי השחקן השני שלא יודע את העתיד, ולכן בסוף הבאנו את האלגוריתם הזה כאופציה שלישית.

שני הפתרונות האחרים הוגנים יותר והם מעריכים כמה טובות הפעולות בתהליך האימון ולא במהלך המשחק עצמו. הפתרון הראשון מסוג זה הוא אלגוריתם .Hierarchical Deep Q Networ-אלגוריתם זה משתמש ברשת נוירונים כדי להעריך את qvalues של פעולות. האלגוריתם היררכי כי למעשה הוא משתמש בשתי רשתות: אחת עבור הקומות ואחת עבור שלושת הלבנות בכל קומה.

הפתרון השני דומה לראשון ונקרא .SARSA (State Action Reward State Action) . ההבדל ביניהם הוא ש- sarsa מעדכן את qvalues בהתבסס על הפעולה שהסוכן נוקט בפועל, כולל פעולות exploration. מצד שני DQN ,משתמש במדיניות יעד, שהיא חמדנית, כדי לעדכן qvalues עם הערך המרבי במצב הבא ללא קשר אם הסוכן עשה פעולה זו בפועל.

בחרנו לייצג את מצבי ג'נגה כמטריצה שטוענים אותה מהתמונה. התמונה היא "סריקה" דו-צדדית של המגדל.

להראות פיזית

כשאימנו את האלגוריתמים שלנו, יצרנו תפריט ב Unity-שמאפשר למשתמש לבחור את האלגוריתמים שיתחרו זה בזה ולראות את הביצועים שלהם. זה גם אפשרי עבור השחקן האנושי לשחק נגד אחד האלגוריתמים, ושני בני אדם לשחק אחד נגד השני.

אנו מקווים שתנסה את המשחק ותספר לנו מה אתה חושב. תודה על תשומת הלב שלך!