OPTIMIZING JENGA GAME PLAY WITH DEEP Q-LEARNING OF STRATEGIC BLOCK EXTRACTION

A PREPRINT

Chen Liang
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
cliang73@gatech.edu

Jesse Huang
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
jhuang347@gatech.edu

Leng Ghuy
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
mghuy3@gatech.edu

April 26, 2019

1 Introduction

This project seeks to solve the decision making process of which block to extract when playing a game of Jenga. Marketed by Hasbro, the popular game involves building a wooden tower of blocks (three at each level) and then have players continuously take turn pulling out blocks one by one and placing them on top of the tower until the stack eventually falls. The game is often played with multiple people where the goal is to not be the player that causes the tower to fall, but in this case, the robotic arm would be placed in a single-player setting, where its goal will be to keep pulling and stacking to create the tallest tower possible. As the blocks are removed from the lower part of the tower, instability will begin to show, which subsequently will make determining the pressure distribution among the blocks difficult and finding the precise and steady force to apply to remove the blocks extremely uncertain.

An artificial intelligent agent not only has to combat these difficulties, but it must first solve the problem of finding the optimal move at any given point, which includes combining more information such as expert play, rules of physics, real time observation and feedback from the tower throughout every aspect of the game. Teaching a robot to evaluate the various states given the multitude of stochastic parameters makes it nearly impossible to achieve a high level accuracy. We originally implemented a baseline AI-agent using SARSA on-policy and Temporal Difference update. Unfortunately, the state space was too large to solve the full height of the Jenga tower. Therefore as a direct continuation of the project, we aim to combat this issue of state representation with a Deep Q-Learning architecture that would allow us to create a robust reinforcement learning agent capable of playing Jenga scaled to any tower height.

2 Related Work

Reinforcement learning is a subset of machine learning that involves allowing an agent to learn through reward maximization. Its application ranges across many disciplines such as game theory, simulation-based optimization, statistics, etc. Many common strategy game platforms that utilizes reinforcement learning has now come to include Jenga as well. There exists a Java implementation of a Jenga AI solved by Francesco De Comité, using Uri Zwick deterministic and discrete mathematical study of the game [1]. The algorithm that was used to solve this Jenga agent assumed a nim-like principle of the game where there is always a winning strategy. Zwick proved through mathematics that "n full layers of blocks is a win for the first player if and only if n = 2, or n is equivalent to 1, 2 (mod 3), and n 4" [6]. However assuming such deterministic outcomes is not plausible for robotics programming when the numerous

aforementioned parameters of observed states, instability, pressure on blocks, etc, has to be included in the later part of the research project. So to view the problem from an engineering perspective, the next step involved researching some already successfully built robotic arm capable of playing Jenga, such as the one uploaded by Torsten Kroeger (Figure 1) [5]. However, while this machine is demonstrating extreme precision of its extraction process and analytical perception of the environment through a multitude of cameras and sensors, when it comes to actually determining the blocks to pull out, the arm is doing this at demonstrating a careful extraction of a Jenga block random.

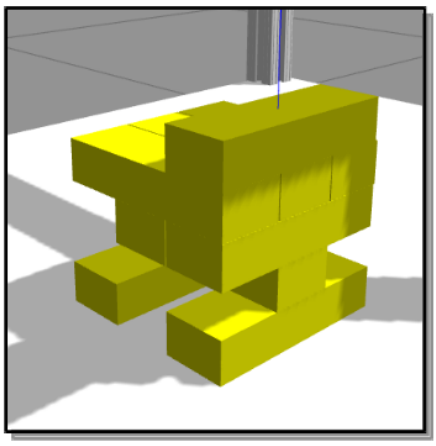
Very few robotics projects exist that combine the two studies of artificial intelligence with information gathering from real-world through sensors to look at how a robotic arm can not only autonomously play the game of Jenga, but to do so strategically. One primary successful example comes from a study conducted at the Georgia Institute of Technology where the authors discuss their implementation of a robot Jenga player by finding the solution for strategic block extraction through consideration of block selection, extraction order, and physics-based simulation that evaluates removability [3]. They also combine elements of advance vision techniques that allows for the constant identification and tracking of all the blocks within the tower.

3 Software and Hardware Specifications

The project requires the use of the Robot Operating System (ROS) as a middleware of communication for the robot's control and manipulation to extract blocks. However, at this stage, research is still in progress to control exact block extraction, thus a smaller environment (Figure 1) of only the Jenga pieces in the platform was created for testing of AI agent. The Gazebo plug-in is used to simulate real-world physics such as mass, friction, and gravity that is acting on the tower and on each individual piece to more realistically represent the fall of a Jenga tower. While it can be relatively deterministic by looking for instances where a level is missing two consecutive pieces next to each other, a Jenga tower could fall even it is theoretically stable due to a slight shift of weight. At its bare minimum, the current environment simply destroys the block that we want to move and recreate another slightly above the top of the tower. This extraction process is not fully representative, but we believe that an AI agent who is not aware of the tower stability or able to calculate haptic feedback would not require this information anyway.

3.1 ROS - State and Action encoding

All states are represented as a binary string where set bits indicate the location of a present block. A binary string read from left to right represents a view of the tower from the top to the bottom. Figure 1 shows a few examples of states translated to binary for a simple three-tall Jenga tower. The way in which ROS understands which block to move is by reassigning the number identification dependent upon the configuration. The first set bit is always block 0, followed by the next set bit being block 1, and the block number is what gets passed to ROS to be the simulation to perform the action. With the original SARSA implementation, we chose to keep the state and action representation exactly as the ROS current encoding, but we will later discuss, some updates were made for our deep neural network model.



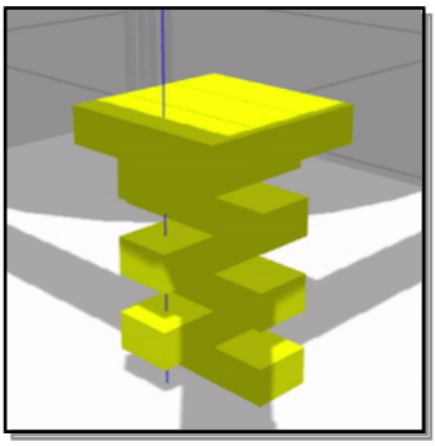


Figure 1: Continuous = 001 111 011 010 101 Goal = 111 010 010 010 010 010 010

4 Method I: SARSA with TD-Learning

Give that Jenga has non-deterministic outcome, we required the use of reinforcement learning update function as TD-update to replace our transition function in this MDP set up. We chose the SARSA algorithm in conjuncture because it was simply the ideal choice at the time as it integrates well with TD-learning and is an easy implementation to understand. The algorithm is aptly named so as it depends on the current state, the ideal action at this state, the reward, the new observed state, and the ideal action from the new observed state. The main algorithm that we implemented can be observed below. Unfortunately, the main issue with SARSA is that not only does it store its utility table for all states, but it stores a table of utility for all states and their possible set of actions. This ended up causing a major scaling issue in the original project. The algorithm details are shown in Appendix I.

4.1 Challenges and Difficulties

The biggest challenge came from the incredibly large state-space of this Jenga environment. The determined height of the tower in which the research group planned to create their environment around is ten tall. Having then decided that each possible configuration of the tower needs to be considered a unique state, we realized very quickly that the environment would be extremely huge. Given that there are five unique possible valid configurations of the three blocks at any given level: 510 = 9,765,635 states. This is a rough estimate of the actual total state-space for a ten tall tower, as remembering the rule of the game, we have to also consider the tower as its increasing in level. Unfortunately, SARSA does not scale. Therefore the only solution at the time was to reduce the state-space and test the model on smaller environment. However, another approach is to create a learning algorithm that would not need to store information about each unique states, but to estimate its quality given a large amount of training example, thus the needing to implement our deep q-learning example.

5 Method II: Deep Q-Network

5.1 State & Action Representation

State Representation We treat the Jenga game as a MDP process. The state representation is the current game board configuration. Given the tower initial height of H_{init}, we construct the board as an 2D binary array of size $H_{max} \times 3$, where $H_{max} = 3 \times H_{init} - 2$ (i.e. The optimal state is obtained when only one center block on each level of tower expect for 3 blocks at the top level). Each row i of the state represents the occupancy of the block at the tower level $(H_{max} - i)$.

Action Representation The action is the position index of block the robot choose to remove. The indexing scheme is by flattening the tower into 1D array and indexing from $\{0, 1, ..., 3H_{max}\}$. Note that the action does not include the position index of the robot choose to place the selected piece, the reason is that the Jenga game environment setting in ROS requires the robot to place the selected piece always following the same rule (i.e. either from left to right or from front to back) and we have no control over this degree of freedom. A visualization of our game state and game action is shown in Figure2.

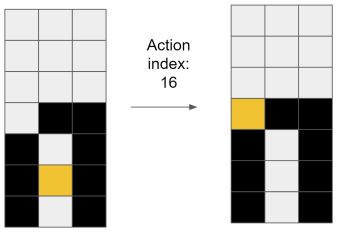


Figure 2: A possible action of moving the yellow block (position index 17) to the top of the tower (position index 9) for a initial 3-level Jenga Tower

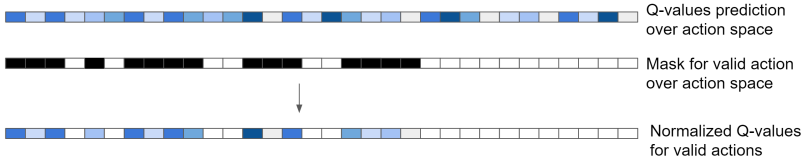


Figure 3: Action space masking method

Action Space Inconsistency We used convolutional neural network to approximate the Q-function. Our Q-function takes in the current state and output a probability distribution over the action space. Considering the shape of the tower is consistently changing through game process, the set of position indices of the block that is valid for the robot to choose from is constantly changing. To avoid this inconsistent action space problem, a common approach is to define the action set equally for every state, return a negative reward whenever the performed action is invalid and move the robot into the same state, thus letting the agent "learn" what actions are valid in each state. More specifically, with the information of the current state configuration, we masked value in the output probability vector that corresponds to an invalid action and do a re-normalization over the probability distribution over the rest actions. We select the action corresponds to the highest Q values among the invalid actions and continue the learning process. The action-space restriction method is shown in Figure3.

5.2 Model Architecture

To approximate Q function, we implemented a deep convolutional neural network (CNN) that maps the configuration of the board at current state to a distribution of Q values over the action space. We make several modifications to the base NN model in order to encoding more meaningful information in the learning process.

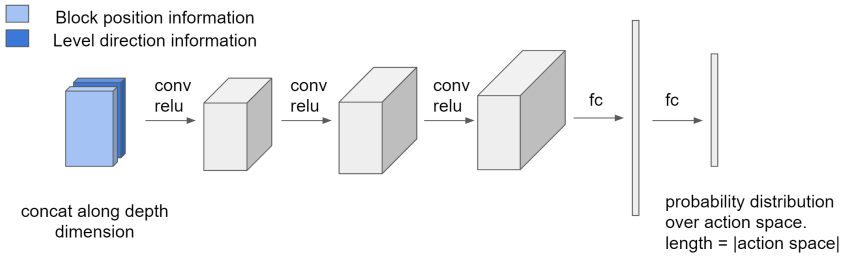


Figure 4: Directional Information Encoding Network

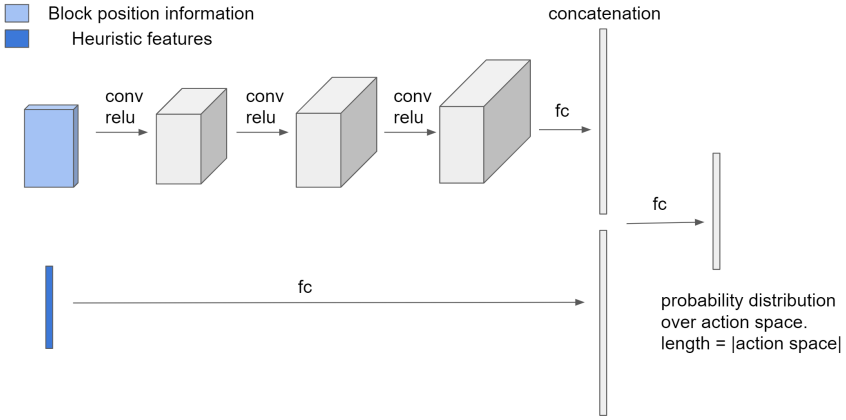


Figure 5: Heuristic Information Encoding Network

Directional Information Encoding Our state representation has the limitation that it only represent block positional information. However, the direction of block placed in the tower are different from level to level, which cannot be directly encoded in the state representation. This information is potentially useful for neural network to level-level decision instead of block-level decision. To overcome the limitation, we concatenate a binary map that encodes the level direction information with the original state representation along depth dimension as the network input. The binary map has the same height and width dimension as the original state representation. Each row is interleavingly assigned to be one and zero representing two different directions. The structure of directional information encoding architecture is shown in Figure 4.

Heuristic Feature Encoding To augment the feature embedding of our neural network, we constructed some heuristic features based on empirical analysis on the Jenga game. A optimal state of Jenga game consist all $[0, 1, 0]$ pattern in each level. Based on this intuition, the number of level match this desired pattern can be viewed as an important feature for a promising game. Also, a failed game usually caused unbalance due to the removal of centering piece. Once a centering piece is being replaced, then removing any other piece in the same level would be more likely to cause an unbalance situation to happen. Hence, the number of level that match the pattern $[1, 0, 1]$ can be also viewed as a meaningful feature. Intuitively, the pattern $[1, 0, 1]$ is less desirable in the lower level of the tower compared with in the higher level of the tower. If in the higher level, it is possible the robot put a piece in the empty position and make the level full. Hence, we compute this feature as being weighted by the height of the level. With the heuristic feature computed, we pass them through a fully-connected layer that are not shared with the base network. We concatenate the output feature embedding with the output of the base network, and use another fully-connected layer to learn whole embedding. The structure of heuristic feature encoding architecture is shown in Figure5.

5.3 Reward Design

The reward function takes consideration of 3 possible conditions. 1) If at game iteration t, the game continues after the current action, then we will give it a base reward t. Intuitively, we want to give the robot increase reward as it goes further in game. In addition, we check each level of the current state, for each level that match the desired pattern $[0, 1, 0]$, we add constant increment on the base reward; for each level that match the undesired pattern $[1, 0, 1]$, we add constant negative reward on the base reward. 2) If the game fail at the current state, we give a constant negative reward. 3) If the game reach the optimal state (defined in section 5.1), we give a constant positive reward. The reward function is shown below.

$$\text{Reward} = \begin{cases} -50 \text{ (ROS simulation)} / -10 \text{ (Local simulation)} & \text{if the tower fell} \\ \#\text{moves} + 2 \times (\#\text{level match } [0,1,0] - \#\text{level match } [1,0,1]) & \text{if the game continues} \\ 100 \text{ (ROS simulation)} / 10 \text{ (Local simulation)} & \text{if it reached goal state} \end{cases}$$

5.4 Algorithm Framework

Exploration & Exploitation We divide each game episode in two stages. In the first stages, we deploy random selection strategy when choosing action, and we only do data collection and save those data samples in the replay buffer. In the second stage, we deploy an epsilon-greedy algorithm for action selection. Model learning begins at this stage.

Replay Buffer We update replay buffer number of times each episode. We do simulation using current policy and exploration $\&$ exploitation strategy. We store the current state, next state, reward and whether it is a end game in the replay buffer. We keep a maximum capacity for the replay buffer, and eliminate historical samples periodically. In the learning stage, for each update iteration, we sample a batch of samples from replay buffer to train neural network.

The detailed algorithm framework is shown in Appendix II.

6 Experiment & Evaluation

6.1 DQN Local Simulation Experiment

6.1.1 Local Simulation Environment

We find it is sufficient for testing purpose by using a locally implemented naive physical engine. Note that the main difference of the local simulation compared with ROS simulation is that the action taking locally is deterministic. In addition, ROS check the fallen condition using internal physical environment, which is more accurate. Our vanilla implementation of fallen check is by computing the center of mass of the tower above the current level, and check whether the center of mass fall on the surface defined by the current level. We do this check for each level below the top level. If we encounter an empty level, then the surface area it defined is 0 and always leads to a fallen condition.

6.1.2 Experiment Results

We do experiment for different model architecture settings as discussed above for a 5-tall tower and 10-tall tower. The maximum number of moves in all validation games in after training for approximately 250 episodes are shown in table1. We can observe that by using both directional information encoding method and heuristic feature encoding method, the

robot achieves the best performance. Compare with directional information, the heuristic feature plays a slightly less important role for the model learning performance.

Table 1: The maximum number of moves for 5-level tall tower and 10-level tall tower

Experiment Setting (Dir: Directional Information Encoding; Heu: Heuristic Feature Encoding)	5 tall tower	10 tall tower
Dir, Heu	20	33
w/o Dir, Heu	15	24
Dir, w/o Heu	17	25
w/o Dir, w/o Heu	13	18

Figure6 and Figure7 shows the averaged validation reward v.s. the number of training episodes. We show only the first 500 episodes due to lack of training time.

Comparing reward curve under different experiment setting, we can observe that with directional information encoding and heuristic feature encoding, the network can learn more meaningful information. For lower tower, the directional information dominants since the number of level in a tower is limited for pattern matching. As the tower height increases, the feature information becomes more important. Comparing experiment performance of the 5-tall tower with 10-tall tower, we can observe that while the reward curve for 5-tall tower is consistently increasing, the reward curve tends to have a sudden increase and stuck at a certain level for a longer period for 10-tall tower. The possible reason can be it is easier for the 10-tall tower to achieve lots of pattern matching reward so that it obtain a higher reward score easily. However, the optimization landscape become more complex for 10-tall tower, which make it easier for network to resides in local optimal. We plan to allow more training time for 10-tall tower with more fine-tuning on learning rate and optimizer selection.

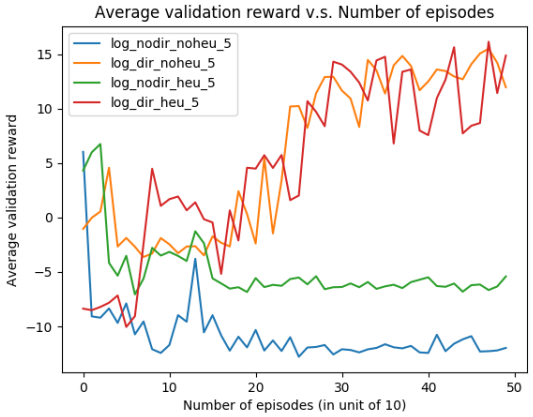


Figure 6: Average validation reward v.s. The training episodes for 5-tall tower

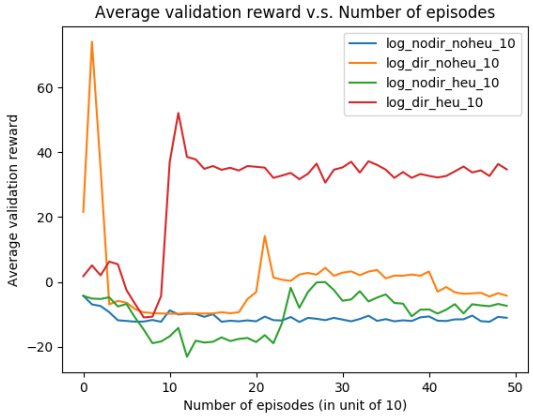


Figure 7: Average validation reward v.s. The training episodes for 10-tall tower

6.2 ROS Environment Simulation for DQN

Unfortunately due to the time and slow training of with ROS, we were only able to test the implementation with the heuristic feature encoding on a 2D scaled Jenga. Unfortunately we were not able to extract important information that could arise from understanding the directions of each block on various levels, but given the locally implemented simulation, the results saw little change. We ran the ROS integrated training sample for approximately 200 episodes with ten games per episode. To stress the downside to having to train and simulate with ROS, this was the result of having to run the algorithm for nearly 60 hours straight. Training with ROS is incredibly slow. In order to not lose important information, the agent would need to sleep or wait approximately one to two seconds after every move in order for publishing nodes to properly determine when the tower has fallen and to communicate this information back

to our node. The reward graph for a ten tall tower can be observed in Figure 8. It should be noted that the algorithm never quite found the optimal state, but it did show a promising reward curve. We can observe an upward trend at around the 50th episode, but it does begin to oscillate with a downward trend at the end. The reason behind this being that every so often, a random action chosen could heavily affect the current policy and cause the tower to collapse within a few moves. With such training examples being stored on the Replay Buffer, we were losing some training examples that might have been found early on when random actions were not taken.

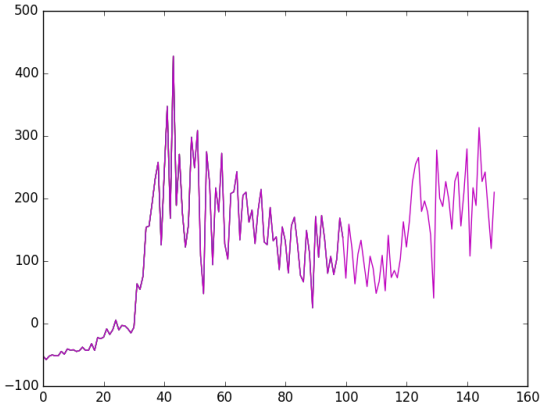


Figure 8: Our average accumulative reward at the end of each episode. (x-axis: epochs and y-axis: average cumulative reward)

6.3 Discussion

6.3.1 Learning Parameters & Model Details

We use kernel size of (2, 3) because we wish to capture the local positional and directional relationship among each two level and among all 3 blocks on the same level. Considering we are using kernel of size 3, we pad the width dimension to avoid losing board edge information. For tower with different heights, we scale the network model complexity for better learning performance. The learning rate we choose is $1e-4$ with RMSprop optimizer.

6.3.2 Local Simulation Performance v.s. ROS Simulation Performance

Local Simulation Mechanism Overestimate Failure Case It is possible the vanilla local simulation mechanism regards some none fallen state mistakenly as end state since we do not take consideration of all physical properties in the system. This would mislead the model to learn inconsistent information and result in lower reward and instability. In the simulation performance in ROS environment, the averaged accumulated reward curve is more stable using the same learning algorithm regardless of the randomness in robot's action.

Inconsistent Action Space The adopted strategy of dealing with inconsistent action space has several drawbacks. Firstly, the target Q values will not be updated either termination or after the specified update period under the experience replay setting. Hence, it maybe take a very long time for the model to learn the valid set. The robot is highly likely to pick invalid move each time but not receive instant feedback penalty for that pick. This drawback present in both simulation setting such that the model learning progress is slow.

7 Conclusion

Our implementation of Q-learning appears to be a viable solution for solving strategic Jenga block extraction. While we were not able to demonstrate it reaching an optimal state, we believe that with enough training episodes and time, it could easily perform well for any height of a Jenga tower. There are of course updates to be made to the algorithm such as the use of epsilon greedy, and its effect on maintaining good training examples in later episodes. We would also like the opportunity to run training examples with the three dimensional encoding that was used in the local test.

8 Appendix I

Algorithm 1 SARSA with TD-Update

Q[s,a], π = initialized to an arbitrary value
s = initialized arbitrarily
a = π[s]
while *s is not terminal* **do**
    s' = observed state from simulator carrying action a
    a' = π[s'] based on Q
    Q[s,a] += alpha * (r + γ*Q[s',a'] - Q[s,a])
    policy = updatePolicy(Q,s)
    s = s'
    a = a'
end

9 Appendix II

Algorithm 2 DQN Framework

NUM_EPISODES = # the total number of episodes
NUM_GAMES = # number of games per episode to collect training data
NUM_UPDATES = # number of learning updates per episode
START_LEARNING = # the episode to start learning
Q^*_UPDATE_PERIOD = # number of episodes per Q^* update

for *episode in NUM_EPISODES* **do**
  if *episode* $<$ *START_LEARNING* **then**
    strategy = random_choice
  end
  else
    strategy = epsilon_greedy
  end

  # Phase I: Collect training data
  for *t in NUM_GAMES* **do**
    env.reset()
    while *the current game not end* **do**
      action = policy.take_action(env.state, strategy)
      next_board, is_end = env.transition(action)
      reward = compute_reward(next_board, t, is_end)
      replay_buffer.add(state, action, reward, is_end)
    end
  end

  # Phase II: Learning
  if *episode* $\geq$ *START_LEARNING* **then**
    ys = []
    for *_ in NUM_UPDATES* **do**
      states, actions, rewards, next_states, is_ends = replay_buffer.sample(BATCH_SIZE)
      for *(state, action, reward, next_state, is_end) in states, actions, rewards, next_states, is_ends* **do**
        if *is_end* **then**
          ys.append(reward)
        end
        else
          ys.append(reward + $\gamma \max_{\text{next\_action}} Q^*_\theta$(next_state, next_action))
        end
      end
      $L_\theta = \frac{1}{2}(Q_\theta$(states, actions) - ys)^2/BATCH_SIZE
      $\theta \leftarrow \theta - \alpha \nabla_\theta L_\theta$
    end
  end

  # Update Q^* function every TARGET_Q_UPDATE_PERIOD
  if *episode % Q^*_UPDATE_PERIOD == 0* **then**
    $Q^* \leftarrow Q$
  end
end

10 References

[1] Francesco De Comite. 2005. A Java Platform for Reinforcement Learning Experiments. Software available at www.lifl.fr/ decomte/piqle.

[2] G. A. Rummery M. Niranjan. 1994. On-Line Q-Learning Using Connectionist Systems. CUED. F-INGENG-TR 166 (September 1994), 21 pages. DOI: doi=10.1.1.17.2539

[3] Jiuguang Wang, Philip Rogers, Lonnie Parker, Douglas Brooks, and Mike Stilman. "Robot Jenga: Autonomous and Strategic Block Extraction". Proceedings of the 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems. p. 5248 – 5253, 2009.

[4] Torsten Kroeger. 2010. A robot plays jenga. Video.

[5] Uri Zwick. "Jenga", Proceeding of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, Society of Industrial and Applied Mathematics, p. 243-246, 2002.