Building Supervised and Reinforcement Learning Systems to play the Game of Risk

Introduction

Risk is a highly popular board game invented in 1957 by Albert Lamorisse. The game supports two to six players and is a turn-based strategy game in which players attempt to occupy every territory on the game board. There are several variations of the game, including Domination, Capital, and Mission, each with either increasing, fixed, or Italian card sets. This project focuses solely on the 2-player Domination version with the increasing card option, whose rules will be described in further detail below. To this end, I have built on top of two existing open-source projects, first building a supervised learning system to play like the hard agent in the Yura Java Domination framework and then an AlphaGo Zero-inspired reinforcement learning agent to play a slightly simplified version of Risk. I chose reinforcement learning to be my agent’s method of learning for two reasons. First, it is highly elegant: agents develop an understanding of games *tabula rasa*, with only self-play and observed rewards, as opposed to tons of data in the form of expert games. Second, the Deepmind team delivered groundbreaking results in reinforcement learning with their AlphaGo Zero and AlphaZero papers and I was keen to apply the techniques used by these researchers in approaching Risk, which has a layer of randomness not present in Go or chess, the games studied by Deepmind. Finally, while the context of game-playing lends itself nicely to reinforcement learning because of the presence of rewards, the technique can be used in other tasks such as creating financial strategies, building manufacturing robots, and creating better targeted ads.
Rules of Risk

As discussed above, this project focuses on 2-player Domination Risk with an increasing card set. The game is played on a map containing 6 continents, broken up into 42 territories. Each player has a set of soldiers, consisting of the following pieces: infantry (worth 1 army), cavalry (worth 5 armies), and artillery (worth 10 armies). The game also has 42 cards labeled with a territory and a picture of either an infantry, cavalry, or artillery. There are also 2 wild cards that have pictures of all 3 piece types but no territory label.

At the start of the setup phase of the game, each player chooses a color and receives a certain number of armies depending on the number of players (e.g. 35 for 3 players, or 30 for 4 players). All players roll a die, and whoever rolls the highest number places one army anywhere on the board. Players take turns placing armies onto unoccupied territories until all territories are claimed. At that point, each player places one additional army onto a territory they already occupy, one at a time, until everyone has run out of armies. The shuffled risk cards form the draw pile.

A player’s turn consists of three phases: placing new armies, attacking, and fortifying. At the start of every turn, each player acquires a number of armies equal to \(\left\lfloor \frac{k}{3} \right\rfloor\), where \(k\) is the number of territories the player occupies, and the notation denotes the floor function. In addition, at the start of each turn, total control of a continent earns the player extra armies (7 for Asia, 5 for North America, 5 for Europe, 3 for Africa, 2 for South America, and 2 for Australia). Furthermore, a player collects Risk cards on any turn in which they capture a territory, and can trade in 3 cards of the same army type, of all different army type, or a wildcard with any other two cards. The number of armies awarded increases with the total number of previously-traded-in sets throughout the game. Furthermore, if any card turned in contains a territory the player
occupies, the player is awarded exactly 2 extra armies that must be placed on that territory. The rest of the armies may be placed freely.

The second phase of a turn is the attacking phase, which is entirely optional. A player may attack from any of their own territories that contains at least 2 armies towards an adjacent territory controlled by the opponent, and may continue attacking so long as they have at least 2 remaining armies on that territory. The player may switch which territories they are attacking or attacking from throughout the attacking phase. To determine the outcome of an attack, the attacker chooses to roll 1, 2, or 3 dice such that the number of armies in the attacking territory is strictly larger than the number of dice rolled and the defender chooses to roll 1 or 2 dice such that the number of armies in the defending territory is at least equal to the number of dice rolled. The attacker compares their highest die to the defender’s highest. If the attacker’s is higher, the defender loses one army from the territory under attack. Otherwise, the attacker loses one army from the territory they are attacking from. This process is repeated for the 2nd highest die if both players rolled more than 1 die.

The third phase of a turn is the fortification stage, where the on-turn player may move as many armies as they wish from one of their territories to an adjacent one, leaving at least one army behind. After moving some number of armies from one territory to an adjacent one, the player’s turn is over.

The game ends when one player captures all 42 territories on the board.

**Supervised Learning Methodology**

The first stage of the project involved building a supervised learning system to play Risk. For this purpose, I built on top of the Yura Domination open source project, a system that encodes the rules of Risk, provides sample agents such as “easy”, “average”, and “hard” modes,
and implements a UI interface to display games played. The first step to building a supervised learning system was to have the easy and hard AI’s play each other 400 times (of which the hard agent won 345), writing the game states, chosen moves, and game results to a file along the way. In order to create any agent in the Yura framework, one needs to implement methods to return a decision in each of the following game states: trading in cards, placing armies, attacking, choosing the number of dice to roll as attacker, choosing the number of dice to roll as defender, moving armies to a conquered territory, and fortifying. Since storing information about which cards are owned by each player would likely expand the state space drastically and since the Yura framework comes with a best trade function, I used this default function and did not attempt to add card information to the state. I represented each of the other decisions as a separate classification problem. These representations are described below.

1. Choosing where to place armies
   a. Input: 42-vector containing number of armies in all countries (positive if controlled by current player, negative otherwise)
   b. Output: Index of action among all possible place commands (e.g. “4 1” - place 1 army on country 4)

2. Choosing which attack to perform
   a. Input: 42-vector of armies in all countries
   b. Output: Index of action among all possible attack commands (e.g. “5 10” - attack territory 10 from territory 5)

3. Choosing how many dice to roll as an attacker
   a. Input: 44-vector containing armies in all 42 countries followed by count of armies in attacker and count of armies in defender
b. Output: Index of action among all possible attack roll commands (e.g. “2”- roll 2)

4. Choosing how many armies to move to a conquered territory
a. Input: 85-vector containing armies in all 42 countries followed by a 42-vector
   with value 1 if territory is attacker, -1 if it is defender, and 0 if neither and then
   followed by a number indicating minimum number of armies to be moved
b. Output: Index of action among all possible move to conquered commands (e.g. “4”- move 4 to conquered)

5. Choosing which fortification move to perform
a. Input: 42-vector of armies in all countries
b. Output: Index of action among all possible fortify moves (e.g. “32 29 3”- move 3
   armies from country 32 to 29)

6. Choosing how many dice to roll as a defender
a. Input: 44-vector containing armies in all 42 countries followed by count of armies
   in attacker and count of armies in defender
b. Output: Index of action among all possible defend roll commands (e.g. “1”- roll 1)

With this representation in hand, I parsed the hard agent’s (the superior agent’s) decisions from the generated game log files into input and output labels that I then split into training and validation sets. I then trained Naïve Bayes and random forest classifiers on the data and computed classification accuracies, which are reported in the table below.

<table>
<thead>
<tr>
<th>State</th>
<th>Naïve Bayes</th>
<th>Random Forest</th>
<th>Random Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Place armies (treating armies place on the</td>
<td>20.7%</td>
<td>53.6%</td>
<td>1/838</td>
</tr>
</tbody>
</table>
The classification models were trained in Python. In order to use these models in the Yura framework, the next step was to break down the classification process into 3 separate modules: one that trains the models and dumps them into files, a second that exposes a REST API that receives prediction queries, and a third to make test predictions. After this, I built an agent in the Yura framework that queries the REST API to make predictions for each of the 6 cases.

**Results from the Supervised Learning Phase**

Initially, my agent was very slow at beating the submissive agent (agent that never attacks, and thus, can never win). This led me to tweak my framework: rather than simply querying the classification models, I additionally imposed a ban against adding to any territories via fortification or placing of armies if that territory had more than 30 armies on it. In the table above, I have highlighted another change, which is a more

<table>
<thead>
<tr>
<th>Action</th>
<th>Attack Rate</th>
<th>Move to Conquer Rate</th>
<th>Fortify Rate</th>
<th>Other Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same country as part of the same move</td>
<td>35.5%</td>
<td>87.5%</td>
<td>1/42</td>
<td></td>
</tr>
<tr>
<td>Place armies (treating each army as a separate move)</td>
<td>32.1%</td>
<td>35.2%</td>
<td>1/20</td>
<td></td>
</tr>
<tr>
<td>Attack</td>
<td>67.3%</td>
<td>96.2%</td>
<td>1/4</td>
<td></td>
</tr>
<tr>
<td>Attacker roll</td>
<td>31.7%</td>
<td>31.9%</td>
<td>1/64</td>
<td></td>
</tr>
<tr>
<td>Move to conquered</td>
<td>17.2%</td>
<td>25.3%</td>
<td>1/451</td>
<td></td>
</tr>
<tr>
<td>Defender roll</td>
<td>100%</td>
<td>100%</td>
<td>1/2</td>
<td></td>
</tr>
</tbody>
</table>
significant modification that led to large gains. Since my agent was making very strange decisions in the place armies phase, I decided to break each place armies event such as “place 3 armies on territory 37” into separate place armies events, each with 1 army placed. This led to more granular encoding and larger training sets, yielding large gains in both classification accuracies and actual game play. Lastly, my original model was created using Naïve Bayes, but switching to a random forest classifier greatly improved play. Naïve Bayes anticipates that all features independently contribute to the decision made but in reality decisions in Risk require significant interactions between the features: for instance, deciding to attack depends on both the potential attacker and defender’s army counts. A random forest not only captures this detail in its decision tree-based approach to classification, but also brings the reliability of an ensemble classifier, since it aggregates the results from many binary classification trees. All of this being said, the AI still does relatively poorly in its play against the easy and hard agents, winning 8 out of 200 games against the easy agent and 0 out of 200 against the hard agent. While these results may seem discouraging, further trials would likely demonstrate that the supervised learning is significantly superior to a random agent. With just 400 games of data, the level of classification accuracy and play is about as good as one can expect a supervised learning algorithm to perform.

Reinforcement Learning Methodology

After building the supervised learning agent, I built a reinforcement learning system following DeepMind’s AlphaGo Zero paper. Central to the algorithm is a neural network which takes as input the state of the board ($s_t$) and yields a pair of outputs, a probability vector over all actions of the game ($\pi_{\theta}(s_t)$) and an estimate of the value of
the current state from the perspective of the current player \((v_\theta(s_t))\). This neural network is initialized with random weights. Then, in each iteration of the algorithm, several games of self-play are played. On each turn of each game, the algorithm performs several Monte Carlo Tree Search (MCTS) simulations starting at the current state, and this process will be described in greater detail in the next paragraph. Based on the results of this game, training examples of the form \((s_t, \pi_t, z_t)\), where \(\pi_t\) is the estimate of the policy from state \(s_t\) and \(z_t\) is the outcome (which is propagated back up the search tree once the result of the game is determined). Given these training examples, the neural network is then trained to minimize the loss function \(l = \sum_t (v_\theta(s_t) - z_t)^2 - \pi_t \log(\tilde{p}_\theta(s_t))\). At the end of the iteration, the new model is pitted against the previous version, and if a percentage of games greater than a predetermined threshold are won, the new model is adopted.

The MCTS algorithm follows the below procedure to select a move. Initially, MCTS starts with an empty search tree. For each simulation, MCTS chooses actions to take until a never-before seen or a terminal node is encountered. When such a node is reached, the value of the node is the output from the neural network and this value is then propagated up the search path. Still within the individual simulation, MCTS chooses the next action by maximizing the following formula: 

\[
U(s, a) = Q(s, a) + c_{puct} \times P(s, a) \times \sqrt{\frac{\sum_b N(s, b)}{1 + N(s, a)}}
\]

where \(Q(s, a)\) is the expected reward for taking action \(a\) from state \(s\), \(N(s, a)\) is the number of times action \(a\) has been taken from state \(s\), \(P(s, .) = \tilde{p}_\theta(s)\) is the initial estimate of the probabilities of taking each action from state \(s\), as outputted by the neural network, and \(c_{puct}\) is a hyperparameter that increases the exploration rate of the state space. Throughout MCTS, when a new node is encountered its \(N(s, a)\) and \(Q(s, a)\) are set to 0 for all actions \(a\). When propagating a value back up from child states, \(N(s, a)\) is
updated to $N(s, a) + 1$ for the chosen action $a$ and $Q(s, a)$ is updated to

$$\frac{N(s, a) \cdot Q(s, a) + v_{\text{child}}}{1 + N(s, a)}.$$ Importantly, the MCTS implemented by Surag Nair returns the negative value of the current state to the parent, since players alternate turns. However, in Risk, a player may play twice in a row (for instance when they are placing several armies on a turn) and thus MCTS had to be repurposed to take into consideration the parent and current state’s on-turn player.

The RL agent was built on top of Surag Nair’s AlphaZero General system, which builds AlphaGo Zero agents for Othello and several other games. In order to build this agent, I had to implement the game interface for Risk and the neural network architecture, in addition to repurposing the MCTS logic. The game state was represented as a 49-element vector with the army counts in each territory, followed by the attacker’s index, the defender’s index, the attacker’s number of dice, the defender’s number of dice, the number of armies to place, a number representing the game state, and the number of turns played by the RL agent. The logic for the game included getting the next state given a state and action, getting all valid moves in a state, and getting the outcome of a state if it is a terminal node. Initially, the neural network had 4 convolutional layers with 1-dimensional batch normalization, and then 2 linear layers that feed into dense dropout layers before finally feeding into 2 separate linear layers that generate the value and policy vectors respectively. However, given that the representation of the game does not neatly correspond to a spatial arrangement, the convolutional layers and 1-d batch normalization were omitted. The framework heavily uses the canonical form of a board in MCTS, always making player 1 the on-turn player and flipping the values in the state vector if necessary. This allows us to calculate just one set of values for game states, as
opposed to one set of values for each player. Furthermore, for the purpose of the RL agent, the game is ruled finished if more than 75 turns are played by the RL agent and the winner is the one who controls the most territories. I experimented with ending the game once over 150 armies were placed, but this led to very lengthy games. I also tried declaring the winner as the player who has the most armies at the end of the game, but this led to less distributed play than saying the winner is the one who controls the most territories. As a final note on the reinforcement learning methodology, the randomness aspect of determining battle results was built into the logic to get the next state given the action of a defender in choosing how many dice to roll.

**Hyperparameters**

I tried out dozens of hyperparameter combinations in conjunction with both of the architectures described above. Namely, I tried setting the exploration factor $c_{puct}$ value to 1, 3, 4, and 10, the number of episodes set to 20, 50, 75, and 100, the number of MCTS simulations per turn to 10, 25, 50, and 100 before drastically increasing simulations to 6400. The reason the number of MCTS simulations had to be increased so significantly was because the average branching factor is so large and so 6400 simulations approximates only a 2-level deep breadth first search. When choosing whether to accept a new model, the old and new agents played 25 games, with a required win percentage of 52% (i.e. majority wins) for the new agent. 20 iterations of training examples were kept in reserves at all times.

**Reinforcement Learning Results**
The reinforcement learning models take significant periods of time to run and thus are still running on high performance computing machines. Results from the training of the reinforcement learning models will be added here once the models finish training.

**Challenges**

The Risk state space is extremely rich and so learning decision functions in both supervised and reinforcement learning contexts require a great amount of training data and self-play data respectively. Furthermore, capturing proximity in terms of the decision space and the map itself is a challenge because the representation was deliberately chosen to be very sparse in order to make the game logic simple and lightweight. However, as a result, the neural network needed to learn all the spatial features of the Risk map on its own from training data. What is particularly challenging is for it to understand when a decision is close to the best decision, since each output is just the index of a chosen action in a list of all actions. Future work might penalize “closer” decision errors less than egregious errors, at least in terms of the supervised learning agent. As mentioned previously, there are nontrivial implementation details to be wary of in MCTS given the heavy use of the canonical board and the fact that players do not necessarily alternate turns. Finally, the biggest challenge in building a strong reinforcement learning agent is access to compute power, and this was a significant obstacle in testing and refining my models.

**Future Work**

The most immediate future work on this project work would be hyperparameter and architecture refinements to improve the performance of the reinforcement learning agent. In addition, I could perhaps collect more data and/or change the representation of
some of the supervised learning classification problems in order to improve the play of
the supervised learning agent. Lastly, this work could eventually be extended to
accommodate other modes of the game and versions of the game with more than 2
players.

References


