1 Introduction

Playtesting is an important part of the game design cycle. Traditionally accomplished with human players, it can be expensive, logistically cumbersome and time consuming. To ameliorate this situation, we set out to create an artificially intelligent player capable of semi-competently playing a given Unity game.

1.1 The Game

We chose 2D Roguelike as a test game for this project, because it’s an open source official tutorial game for Unity game engine (https://unity3d.com), available through the Unity Asset Store (https://www.assetstore.unity3d.com/en/#!/content/29825). The game is fairly simple. The player progresses through randomly laid out levels with progressively more zombies. Each level the player starts at the lower left corner and must make his way to the exit in the upper right. The game is over when the player runs out of food. Food points are depleted one point per step and 10-20 points per zombie attack. They can be replenished by picking up food items. Zombies move deterministically toward the player taking one step for every two player steps. At any time, the player can perform one of four actions - move up, down, left or right.

![2D Roguelike screenshot](image_url)

Figure 1: 2D Roguelike screenshot
2 Strategy

We decided to try two different methods to solve this problem, one included collecting play traces by playing the game using the Monte Carlo Tree Search and training a model using that data. The second involved applying Reinforcement Learning through Unity’s latest tool - Unity Machine Learning Agents.

2.1 Method 1: Monte Carlo Tree Search (MCTS) and a Neural Net

MCTS is a heuristic search algorithm for decision making [Browne2012]. Most recently it was used in the AlphaGo Zero project. As part of her Master’s thesis, Oleksandra created a framework called Monster Carlo for play-testing Unity games with MCTS. The framework consists of c# library integrated with the game, and a Jupyter Notebook with the MCTS implementation and sample experiment set-ups in Python. Each node in the tree represents a state, and its children represent the states reachable by taking each of the possible moves from the parent state.

![Figure 2: Overall architecture of the Monster Carlo framework](image)

**Generating Training Data:**

Because the game is played on an 8x8 grid, we represented each cell with a enum value of objects currently occupying it. We also included the 0/1 value for whether zombies could move or had to wait a turn, and a normalized value for the percentage of possible good points the player had out of all points possible in that level. We stored all of this in a float array of 66 values.

For the training data collection part of this project, we modified the game and the framework to store the state information in the tree’s nodes every step of the game. These would be our x values. We ran the MCTS on three different level layouts for 20,000 playthroughs each - an estimated number for MCTS to figure out a near optimal path through the level. To improve the search efficiency, we used weighted choice to encourage the player to move toward the exit during the random rollout phase of the algorithm.

MCTS keeps track of how many times each node has been visited. The core idea of the algorithm is in focusing the search on the more promising areas, which means the nodes resulting from better moves would have more visits than their siblings. When the play-throughs were completed, we traversed the resulting trees one node at a time and recorded the normalized number of visits for each of the children from that point. For example, after the search is complete, each state has four children...
(one for each direction). Suppose the “up” node was visited seven times, and each of the other three only once. This would result in a y label value of \([0.7, 0.1, 0.1, 0.1]\). Effectively, these represent the probabilities each of the moves would be taken from that state.

**Neural Net Training on MCTS Data**

While MCTS can be relatively effective at playing the game when it is in a game state that it has seen, when it encounters a new state, it has no sense of what to do, and will move randomly. In particular, this means that MCTS is not effective at playing the game on a new randomly generated board. To give MCTS some “intuition” about how to behave in a new unseen state, a neural net is trained on MCTS data on boards where MCTS has been trained and is already relatively effective.

The neural net was implemented in tensorflow. Original data was the set of 20000 state/action probabilities generated from Monte Carlo playouts as described in the previous section. 10% of the data was held out to for evaluation and the neural net was trained on the remaining 90%

For training, batches of size 8000 random examples were sampled from the training set. In all cases, the input to the neural net was the the vector of length 66 describing the game state, and the output was a set of 4 probabilities. In all designs, loss was cross-entropy loss between the 4 action probabilities provided by the Monte Carlo simulation for the given state, versus the 4 action probabilities predicted by the neural net. Several different neural net designs were tested out; including a fully connected neural net versus convolutional neural net, neural nets with 2 hidden layers versus 3 hidden layers, and using Sigmoid versus ReLU activation functions.

In the case of the convolutional neural nets, the input data was split into the first 64 values which describe the state of physical cells on the board, and two values describing whether zombies can move on a turn and the point percentage. The convolutional neural nets reshaped the physical layout description to two dimensions, and that 8x8 was fed into a tensorflow conv2d layer with a filter of 32, a kernel size of 3x3 or 5x5, and the activation function used for the particular test. The output of this layer was fed into a max_pooling2d layer with a pool_size of 2x2 and a stride of 2 to downsample. The result of pooling was then reshaped and fed into a fully connected layer with 64 nodes. The result of that fully connected layer was fed into another fully connected layer with 32 nodes. The result of this was either fed to an output layer to output 4 action probabilities, or fed to another hidden layer with 16 nodes, which was then fed to the output layer. The activation function used in the output layer was the same as the activation function used in the fully connected layers.

The regular neural net had two fully connected layers, both with 32 nodes, and the result of the second hidden layer was fed into the output layer.

The results from training the neural network with these different parameters is shown below.
In all cases, the training data loss continued to improve after 50 iterations of training, but the evaluation data loss stopped improving after about 5 to 10 iterations. In all cases, the evaluation loss did not improve to much less than 1.36. Training seemed slightly slower for the convolutional networks relative to the non-convolutional neural networks, and non-convolutional neural nets seem to show overfitting more rapidly (evaluation error tended to slightly increase over training iterations a little more in the non-convolutional models). Adding a third hidden layer of size 16 was detrimental to all the neural net models, possibly because it was too narrow of a layer. Though ultimately their best case evaluation error was similar, convolutional models with a kernel of 3x3 trained slightly faster than those with a kernel of 5x5, which makes sense because in the ladder, these are more weights to train. The Sigmoid activation function and ReLU has similar results for the convolutional neural net, but the Sigmoid activation function seemed to perform slightly better than ReLU in for the non-convolutional neural net; perhaps because it did not overfit as quickly. For future work, it would also be interesting to see the effect of changing the number of filters for the convolutional layer.

Overall though, the differences between the different neural net models are quite subtle, meaning there may be issues in the format or preprocessing of the data that is overshadowing the effectiveness of a neural net at predicting ‘Monte-Carlo’-like action probabilities.

**Playing the game with the trained model:**

Hooking into the existing setup for state collection and weighted move choice making, we were able to modify our Monster Carlo experiment to ask the game for its state every time step. We would then feed the state into our trained model and get the prediction of what the corresponding MCTS move probabilities would be for that state. We would send this prediction to the game, which would use the four probabilities as weights for a softmax selection of what move to take next. The cycle would then continue.

Unfortunately, our model did not play the game well, only occasionally clearing the first level. We hypothesize that with even more training data from more diverse scenarios we could achieve higher performance.

**2.2 Method 2: Unity Machine Learning Agents and Reinforcement Learning**

Reinforcement Learning
Reinforcement learning is a technique in which an agent explores state space by repeatedly taking different actions and getting varying levels of rewards in turn. The goal of the agent is to maximize the reward. By trial and error the agent learns the set of actions that map to the given state which in turn maximizes the rewards it gets. It’s important to note that in most cases the reward that is given to the agent in a deferred manner, which means that the the agent needs to have a way to keep track of the actions and their effects on the reward in a context sensitive manner.

For the reinforcement learning part of our system we used the pre existing framework of Unity3D called Unity Machine Learning Agents (Juliani, 2017). The fact that the SDK we were using was in it’s 0.1 alpha release unfortunately hampered our progress at given times. We were forced to work around some quirks and make modifications to the software itself during our project time, which reduced the time we could focus on the actual machine learning component.

The Unity ML Agents SDK came in with the implementation of the proximal policy optimization algorithm developed by Google Deepmind[1]. The algorithm tries to approximate a function that given a state and a set of possible actions, returns the action that will lead to the highest reward at the end of the session. The Unity3D architecture that supports this implementation is comprised of four different moving parts: the agents, the brain, the academy, and the external python tensorflow module.

The agent refers to the actor that exists in the the environment and is the body that is taking the actions. In our case it is the playable character that is navigating the space trying to reach the exit.

The brain is the decision making system that takes in the state representation, the possible set of actions and outputs an action that it believes will maximize the reward. Basically, it implements the given policy. The policy information can be derived from four different sources.
• Player: The player input is directly related to the agent. This is useful in ensuring that the created environment and the defined actions create the desired result.

• Heuristic: A script runs that decides on which action to take. The script is written by the developers and can be used as a test case or an adversary to train against.

• External: The decision making is allocated to a tensorflow model that exists outside of the Unity environment. This mode is also used for initially training the model.

• Internal: A pre-trained model is introduced into the Unity3D environment and the model is asked to evaluate the best action given the state and the possible set of actions.

Implementation details:

For our tests we used a fully connected neural network with the input layer taking our state space, a hidden layer and a single output node. We are using a sigmoid as an activation function.

Action Representations

For the action set representation of our reinforcement learning algorithm we gave the system four possible actions to take from any given frame. The agent could go in any of the four cardinal directions. If the agent was next to an outer wall and the agent decided to take an action that would move the character into the wall we still allowed the action to be expressed, however there would be no effects on the actual game apart from the reduction in food, and giving the zombies a turn to move.

State Representation

For our experiments we decided to use a local state representation (Figure 6). The agent was aware of the closest n sources of food, the closest k zombies and t breakable walls. The values of n, k, and t, changed from different test sessions. Even though there was no direct correlation observed between the values of n, k, t and the score of the agent, increasing the value too much slowed the learning rate of the agent.

Each of the objects of interest were represented by three values. The distance between the x position of the player character and the x position of the object of interest, the distance between the y position of the player character and the y position of the object of interest, and finally, the euclidean distance between the player character and the object of interest. In addition to the information regarding closest objects, the model was also given the information of how much food they had left, the position of the exit in both axis, the distance to the exit and their last move. In total the state representation was a array of floats, each number corresponding to one of the elements discussed.

One shortfall of this state representation we identified happen when there were less than n, k, l objects of the corresponding type. As we reported the distance between the player and the given object, when
there were no object that was being tracked the location of the object was set to 0 in all three axis. Which meant, that, in the event of an object not appearing on screen, the model mistakenly assumed that the closest object of that kind was located at (0,0,0) coordinates. The problem was because we couldn’t dynamically scale the number of values in our state and there was no concept of a null value which in turn defaulted to 0 to result in the mentioned behavior.

**Reward Structure**

We experimented with several different reward structures. We found out that rewarding the agent whenever they reached the exist wasn’t enough, as the training took really long to converge. In addition to the main reward, we also awarded the agent for increasing the amount of food they had. The tweaking of the exact values was very important, as in certain cases, because of the reward ratios of picking up food versus getting to the exit was not properly implemented, the agent learned to search for food but not go for the exit. In addition to the rewards, we got the best results when we punished the agent for taking an action that did not change its position (aka trying to move into the outer walls), losing food (getting hit by a zombie or taking a step), and, of course, running out of food. Ensuring that the agent was punished by a very small margin each step meant that the agent spent less time before deciding to move toward the exit.

**Curriculum Learning and Retraining the Model**

We also realized that instead of trying to train the model with the full complexity of the game, we could divide the task into smaller chunks and iteratively train the model on harder and harder problems.

In certain test runs we initially trained the model at a deterministic environment in which the agent learned the relationship between the food, exit and the enemies. Then we took the same model and made the environment non-deterministic. When compared to starting from a non-deterministic environment initially, the curriculum learned showed better progress and end results (Figure 7).

![Figure 7](image)

Figure 7: **a.** Model trained on non-deterministic environment only after about 100k repetitions. **b.** Model first trained on a deterministic environment. **c.** Comparison of a. and b.

When we incorporated all these different techniques we could train an agent that could play up to level three consistently. The training took one hour and forty one minutes. Unfortunately 2D Roguelike
did not allow for a clear cut score for us to use as a benchmark. From a qualitative perspective, our tests showed that a human player could reliably get to level five. Which means that our model, while far from being a human level player, managed to reach a certain level of competency.
References
