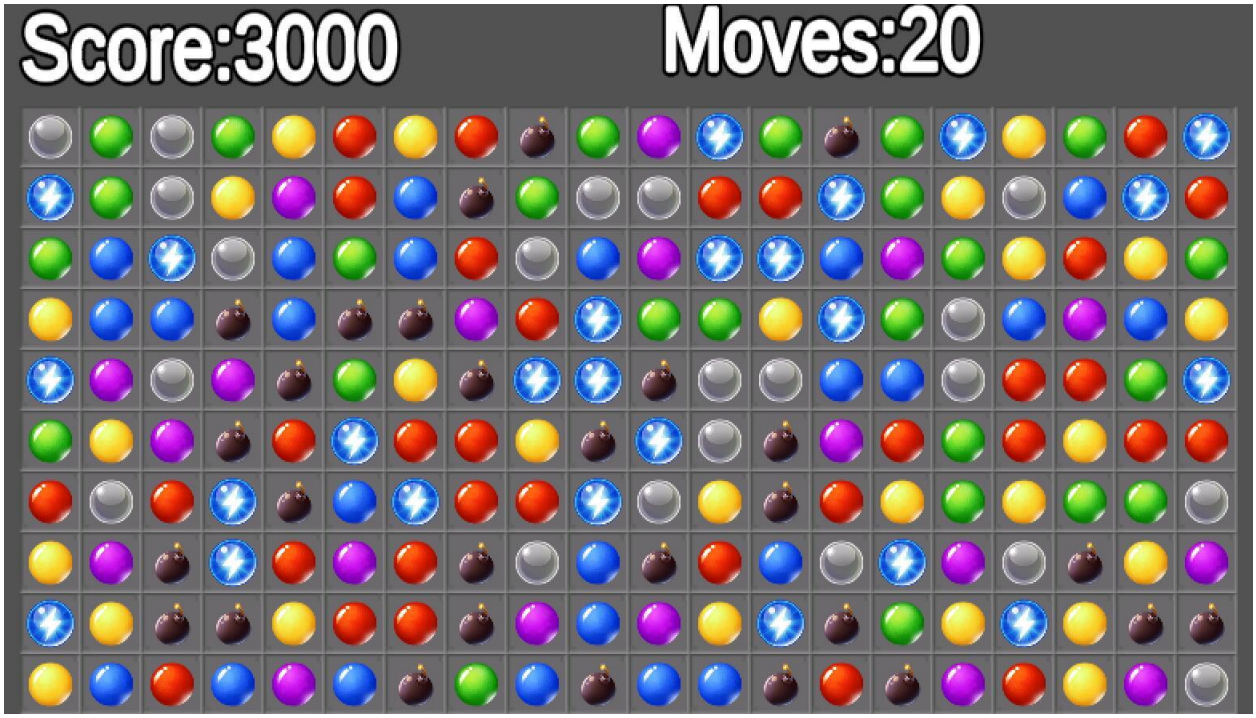
Reinforcement learning for bubble crush
Engineer Design Doc

The goal of your project

One of the most challenging issues in game development is to design “levels” to grab users and keep their interest alive. To create game levels, testing is needed to measure difficulty in different levels. It is a difficult task in terms of human testing.[1] First, it is time-consuming for game designers to test multiple times for one level. Second, the limitation of human operation speed will slow down the process of game development. To reduce the need of human resources and improve testing efficiency, machine learning is employed in game testing. [6]
In “Bubble Crush”, there is a grid with bubbles of various colors. Matching three or more bubbles in a column or a row will cause these matched bubbles to disappear. The disappearance of different colored bubbles will get different scores. There are also some special bubbles like bombs and lightning. Matching these special items will trigger a wider range of bubbles disappearing. To pass the game, players need to obtain a certain score by using a predetermined number of moves. The player’s score on the level is the cumulative score of the individual scores for each move the player makes. [2]
Reinforcement learning is one of the most suitable machine learning paradigms that can be adopted to testing difficulty levels. In this study, reinforcement learning is applied in testing “Bubble Crush”. The learning process involves an environment with an agent. The agent can interact with the environment. The agent takes the current state of the game board and chooses the possible moves. A reward or punishment is given according to the action. The agent tries to avoid punishment and obtain reward by changing its weights to get a good structure of the task. Finally, the agent predicts the best action to get the highest reward and achieve the goal. The difficulty of a level is measured by the score obtained in the predefined moves.

Background and overview of the game

Bubble Crush Game



Our bubble crush game is a classic match3 game which includes these key concepts: level, state, action space,trajectory, reward, and objectives. But we will start from the description of a level, which is specific to the environment.

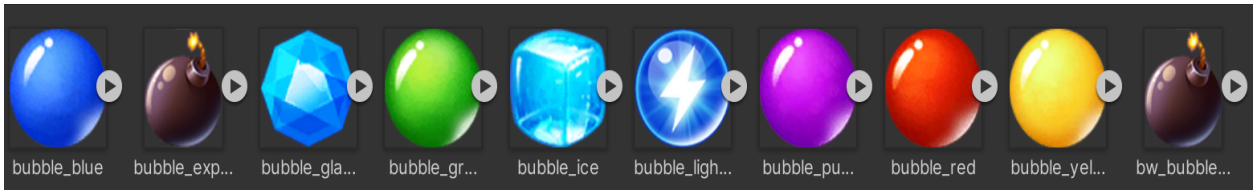
1. Level
   The main element of the game is the level, we defined our own board template. The template is a map which allows determining height and width of the board and coordinates of each bubble. We can use the Level Editorto edit each level such as the board size, the frequency that each bubble occurs, the special type of bubbles(bomb,lighting bubble). After filling the template with different bubbles and determining the type of bubbles we may encounter, we get a final board. Obviously, the biggersize of the board and more types of the bubble will make our game more difficult and complicated so that our agent will require more samples to learn.
2. State
   The current state of the game in bubble crush is a gameboard, which consists of movable bubbles. The size of the board is adjustable and the number of bubbles can be

defined using Level Editor. So we can easily manage the frequency of each bubble and also the difficulty of the game for our agent.

3. Action space
   The core mechanic of the bubble crush is to swap two neighboring bubbles on the board, and only adjacent bubbles are allowed to be swapped(not diagonal). Bubble crush environments can be classified as model-based if the agent knows valid moves which lead to deleting bubbles. In this paper, we only consider the model-based agent because the agent will only focus on learning how to choose between different bubbles rather than try to figure out what is a valid move.
4. Trajectory
   A trajectory is a sequence of the game board states and moves of bubbles on the board. The environment in the bubble crush game is stochastic because each nextstate generated randomly from previous state and deleted bubbles. When there are no valid moves in the current situation, the game will automatically end.
5. Reward
   For the current state, the action just is taken and the next state of the board reward is the number of deleted bubbles. In the blue bubble scenario, destroying a blue bubble will get 100 rewards, while destroying other bubbles will only get 1 reward. The game mechanics do not allow other actions so everyaction of the agent should result in a reward. For model-free settings, it's more reasonable to allow the agent to try to make the wrong move and figure out how the valid move looks like. But in this paper, we generated a valid moves list and only allowed agents to choose moves from the list.
6. Objectives
   In the blue bubble scenario of our bubble crush game,our objective is to collect N elements of particular bubbles(in this case, the blue bubble) or collect Z bubbles in30 seconds (N steps). The current implementation of the environment is the sum of reward collected after 20 steps. Each level in bubble crush is associated with an objective.Players win a level if they reach its objective within the level specific move limits. Here we set the objective to infinity and move limits to 20, so our goal is to collect as many blue bubbles as possible within 20 moves.

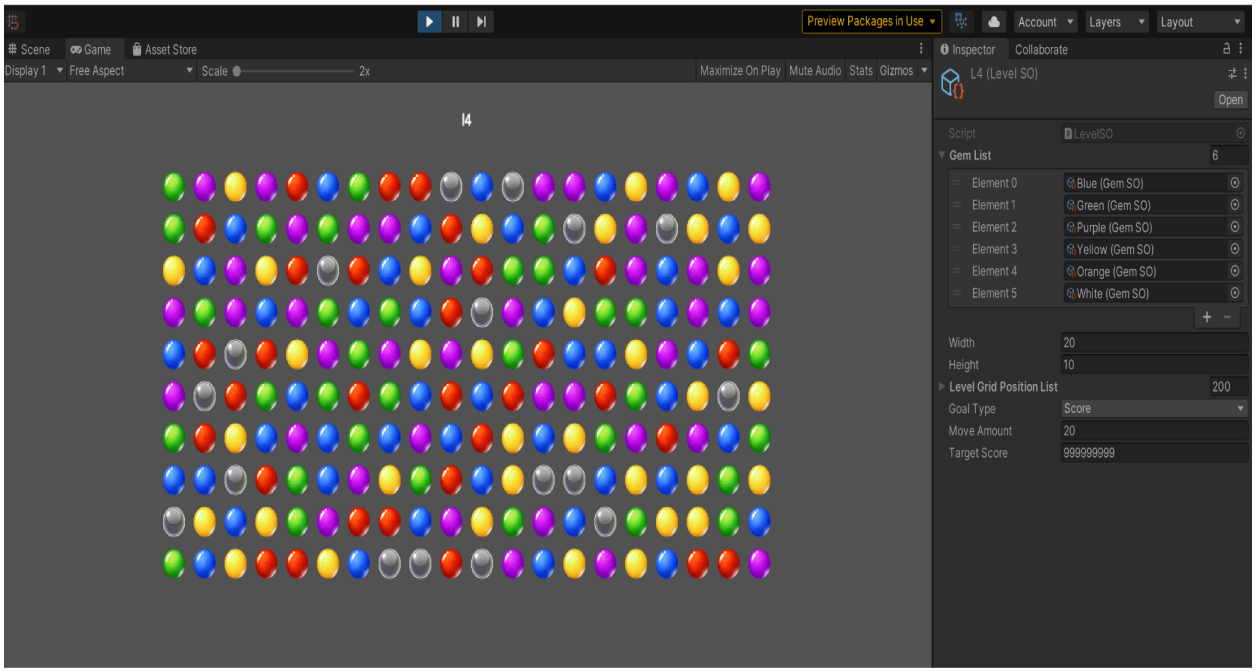
Bubbles

The bubble crush currently has 8 bubble types. Six of them are regular bubbles, blue, green, purple, yellow, orange and white. Two bubbles are special bubbles which will have some effects when destroyed.

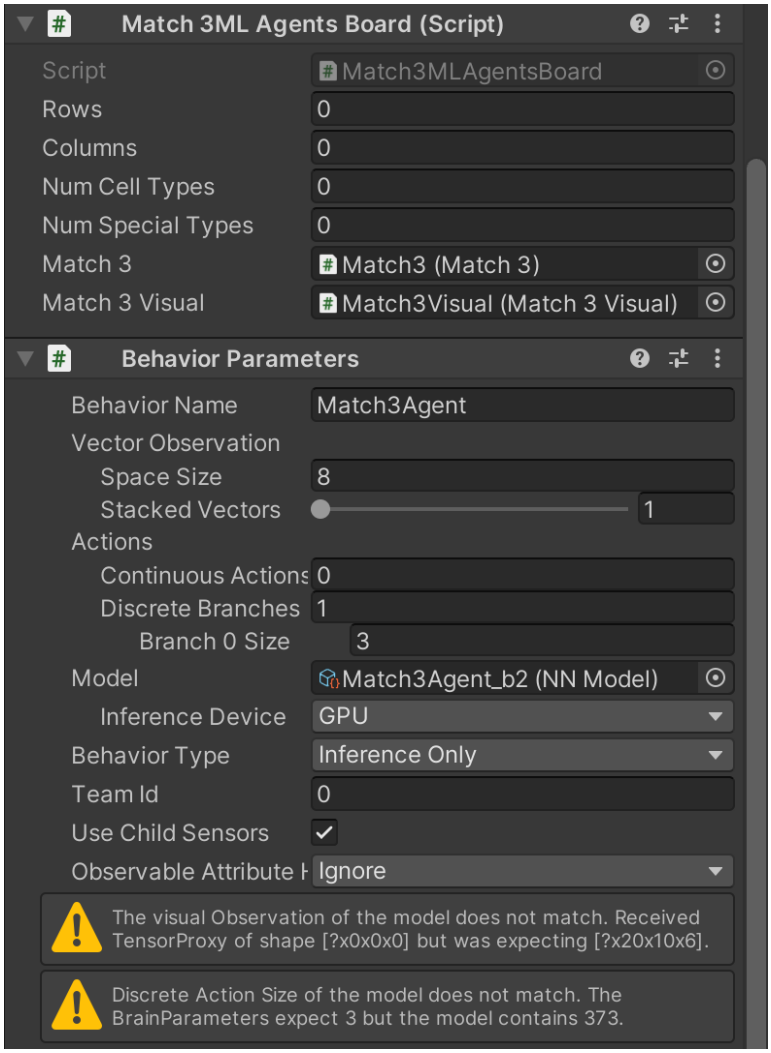


The bomb bubble will cause area damage to nearby bubbles. The area damage will affect bubbles that are within 4 distance of the bomb bubbles, causing them to be destroyed at the same time. The lighting bubble will destroy all bubbles that are in the same row or column of the middle lighting bubble. Obviously, the bomb bubbles and lighting bubbles are very powerful so they may be the top priority when choosing bubbles.

Level Editor



By using the level editor, we are able to modify the initial state of each level. The width, height and goal type. Here we only consider the goal type which is score and we set total move amount and target score. We can also change the bubble by typing 0-7 which is defined in the gem list.

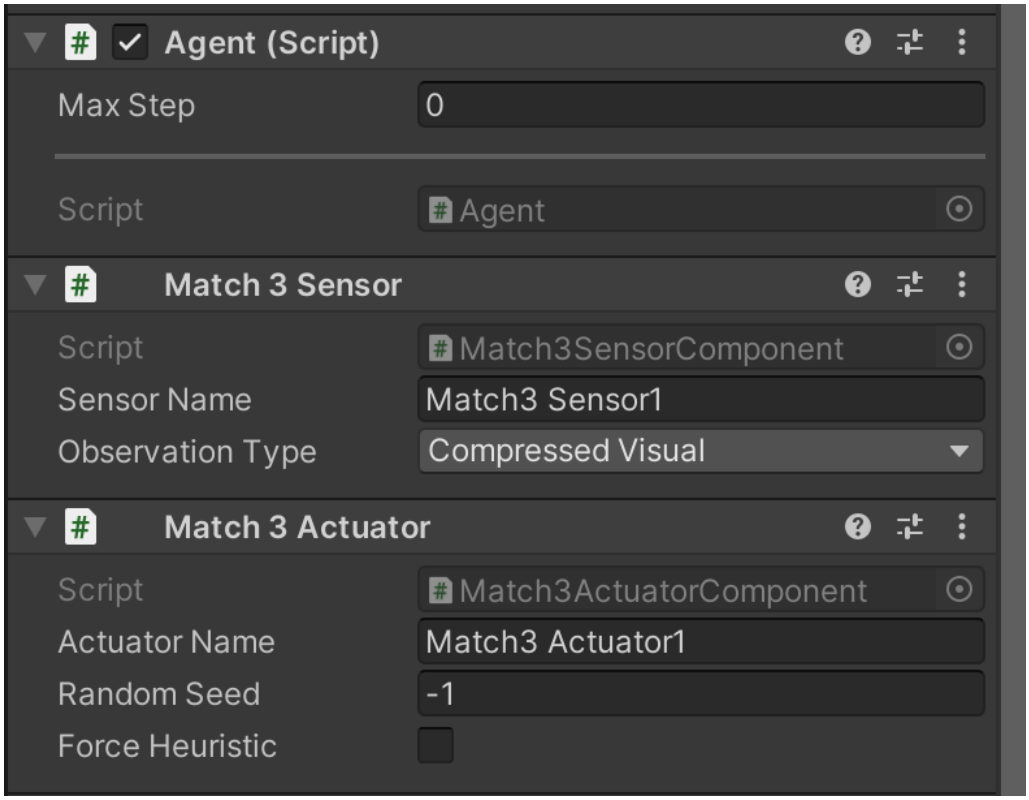


Match3 ML Agents

Here we create our own Match3 ML Agents Board which is a template from ML Agents extensions. Here we only implement some abstract classes and define our behaviour name. Vector Observations should include all variables relevant for allowing the agent to take the optimally informed decision, and ideally no extraneous information.An action is an instruction from the Policy that the agent carries out. The action is passed to an IActionReceiver (either an Agent or an IActuator).

There are two types of actions supported: **Continuous** and **Discrete**. If we use discrete actions, first we need to generate a list of possible valid moves, then our agent will randomly choose a move from that list. The continuous action is like assigning an element in the array as the speed of an Agent. The training process learns to control the speed of the Agent through this parameter. By default the output from PPO algorithm pre-clamps the values of ActionBuffers.ContinuousActions into the [-1, 1] range.

When we want to do training, we set the behaviour type to default. If we have a trained model(.onnx file), then we use this model and set the behaviour type to inference only. Remember to uncheck the Force Heuristic so the agent will use our model to inference each move rather than do training.



Match 3 Sensor and Actuator

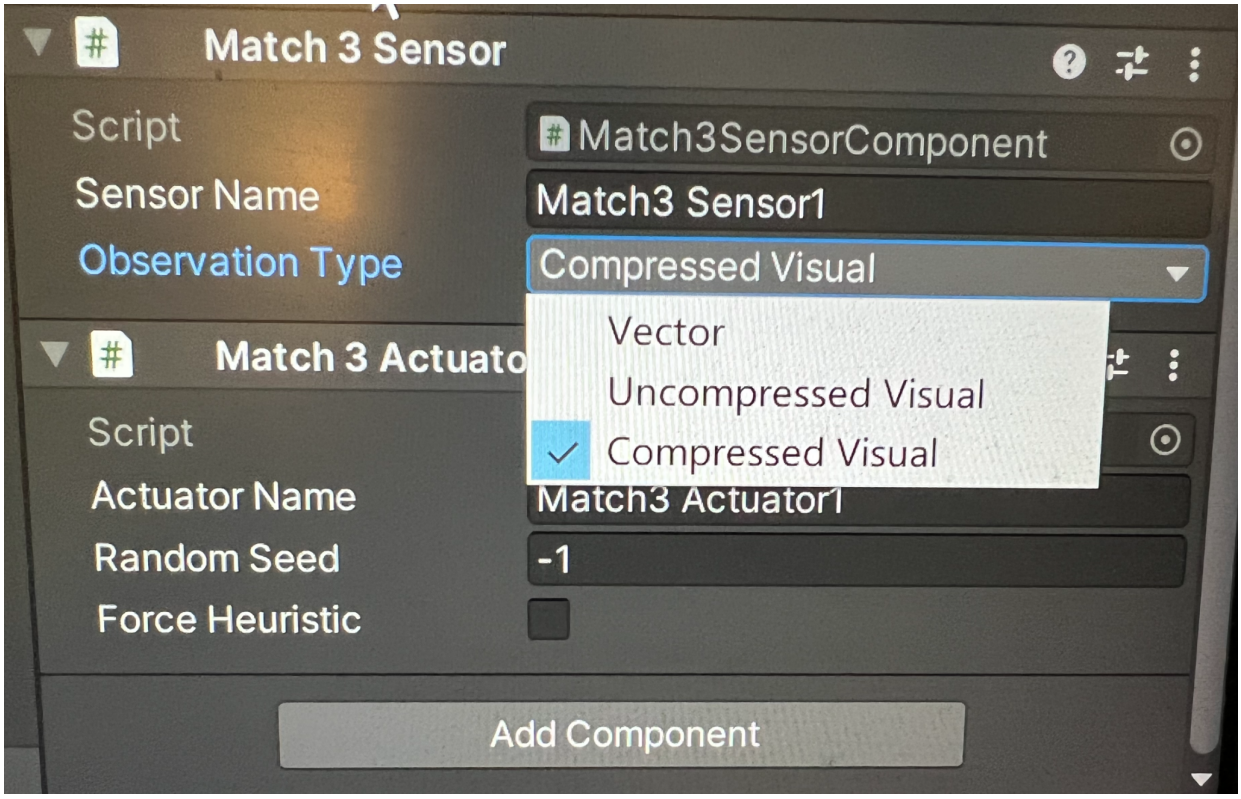
We also need to define our agent. Our match3 agent must have a sensor and actuator.

The SensorComponent abstract class is used to create the actual ISensor at runtime. It must be attached to the same GameObject as the Agent, or to a child GameObject. In our bubble crush game, we use the Match3SensorComponent , which uses the board of a Match-3 game as observations.

The ActuatorComponent abstract class is used to create the actual IActuator at runtime. It must be attached to the same GameObject as the Agent, or to a child GameObject. Actuators and all of their data structures are initialized during Agent.Initialize. This was done to prevent unexpected allocations at runtime. Here in our bubble crush, our actuator will decide potential moves, make continuous or discrete actions and enable heuristic methods if needed.

The heuristic method is like random policy AI, it randomly chooses a move from the validMoveList.

Observation Type



The match3 sensor in ML Agent supports three observation types, the vector, uncompressed visual and compressed visual. The visual one will include a convolution with 5x5 filter and ReLU activation. This type of convolution is designed for 2-dimension board games, which is very good
for Match-3 games. Note that using the match3 CNN with very large visual input might result in a huge observation encoding and thus potentially slow down training or cause memory issues. So, the compressed visual observation type is a better choice to save training time.

Prior Research related to your game

MCTS

Monte Carlo Tree Search (MCTS) is a tree search algorithm that has had an important impact in Game AI since it was introduced in 2006 by several researchers. An extensive survey of MCTS methods is covered by Browne et al. in [1].

$$a^* = \arg\max_{a \in A(s)} \left\{ Q(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\}$$

MCTS estimates the average value of rewards by iteratively sampling actions in the environment, building an asymmetric tree that leans towards the most promising portions of the search space. Each node in the tree holds certain statistics about how often a move is played from that state (N(s, a)), how many times that node is reached (N (s)) and the average reward (Q(s, a)) obtained after applying a move a in the states. On each iteration, or play-out, actions are simulated from the root until either the end of the game or a maximum simulation depth is reached. Figure 1 shows the four steps of each iteration.

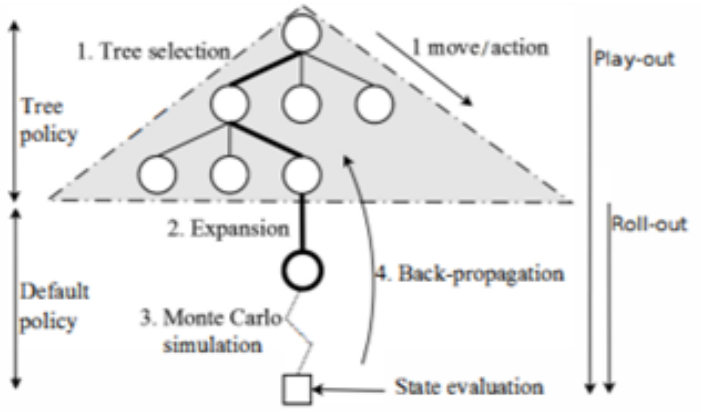


Figure 1: MCTS algorithm steps

When the algorithm starts, the tree is formed only by the root node, which represents the current state of the game. In the first stage, Tree selection, the algorithm navigates through the tree until it reaches a node that is not fully expanded (this represents one of the tree's children that has never been explored). On each one of these selections, MCTS balances between exploration (actions that lead to less explored states) and exploitation (choosing the action with the best estimated reward). This is known as the MCTS tree policy, and one of the typical ways this is performed in MCTS is by using Upper Confidence Bounds (UCBI):

The balance between exploration and exploitation can be tempered by modifying C. Higher values of C give added weight to the second term of the VCBI Equation 1, giving preference to

those actions that have been explored less, at the expense of taking actions with the highest average reward Q(s, a). A conunonly used value is J2, as it balances both facets of the search when the rewards are normalized between 0 and 1.

In the second phase, Expansion, a new node is added to the tree and the third stage, Monte Carlo simulation, is started. Random actions, either uniformly or biased, are taken up to the end of the play-out, where the state is analyzed and given a score or reward. This is known as the MCTS default policy (and a roll-out is defined as the sequence of actions taken in the Monte Carlo simulation step). In the final phase, Back propagation, the reward is back propagated through all visited nodes up to the root, updating the stored statistics N (s, a), N(s) and Q(s,a).

One of the main advantages of MCTS is that it is considered to be an anytime algorithm. This means that the algorithm may stop at any number of iterations and provide a reasonable, valid, next action to take. This makes MCTS a particularly good choice for real-time games, where the time budget to decide the next move is severely limited.

Once all iterations have been performed, MCTS returns the next action the agent must take, usually according to the statistics stored in the root node. Examples of these policies include taking the action chosen more often (a for the highest N(s, a)), the one that provides a highest average reward (Q(s, a)), or simply to apply Equation 1 at the root node.

CNN

Convolutional neural networks (CNN) has become an important tool to solve many computer vision tasks of today. The technique is though costly, and training a network from scratch requires both a large dataset and adequate hardware. A solution to these shortcomings is to instead use a pre-trained network, an approach called transfer learning. Several studies have shown promising results applying transfer learning, but the technique requires further studies. This thesis explores the capabilities of transfer learning when applied to the task of filtering out offensive cartoon drawings in the game of Battlefield 1. GoogLeNet was pre-trained on ImageNet, and then the last layers were fine-tuned towards the target task and domain. The model achieved an accuracy of 96.71% when evaluated on the binary classification task of predicting non-offensive or swastika/penis content in Battlefield "emblems". The results indicate that a CNN trained on ImageNet is applicable, even when the target domain is very different from the pre-trained networks domain.

A convolutional neural network consists of an input layer, hidden layers and an output layer. In any feed-forward neural network, any middle layers are called hidden because their inputs and outputs are masked by the activation function and final convolution. In a convolutional neural network, the hidden layers include layers that perform convolutions. Typically this includes a layer that performs a dot product of the convolution kernel with the layer's input matrix. This

product is usually the Frobenius inner product, and its activation function is commonly ReLU. As the convolution kernel slides along the input matrix for the layer, the convolution operation generates a feature map, which in turn contributes to the input of the next layer. This is followed by other layers such as pooling layers, fully connected layers, and normalization layers. The typical CNN structure is as follows:

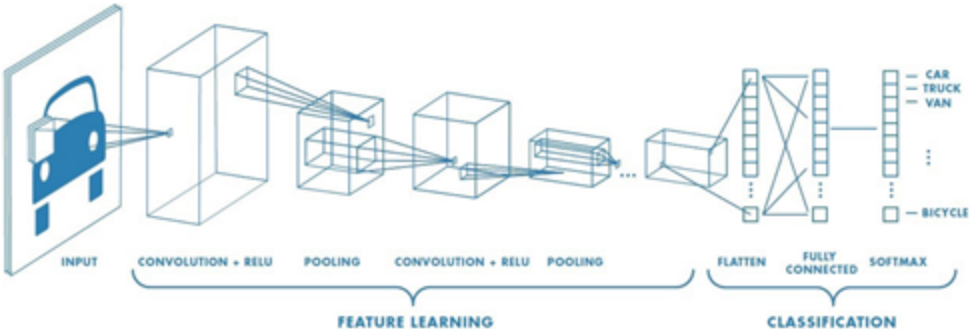


Figure 2: CNN structure

Reinforcement Learning

Algorithms such as finite-state machines (FSM), convolutional neural networks (CNN), and Monte Carlo tree search (MCTS) are widely used in calculating the difficulty of different game levels. For example, Christoffer [9] describes a method of generating player modeling and its application in automated testing of game content using prototype player models. A variant of MCTS is used to build synthetic game testers. The work of [10] and [11] introduced CNN-based methods to achieve the prediction accuracy of 44.4% and 42% respectively on a Go dataset.

Although these methods have a good performance in playtesting, there are several limitations. FSM and MCTS are developed according to specific game rules. Modifications on the models are needed when the game rules are changed. CNN needs a large amount of gameplay data. However, a large amount of gameplay data is often unavailable due to the financial limitation.

In order to overcome these limitations, researchers have proposed reinforcement learning to automatically test difficulty levels in games. Reinforcement learning has been proven to have a good ability to surpass humans in various games, such as in card games, sports games, chess, racing games and arcade games, etc. In work of [12], a deep reinforcement learning approach is employed to learn approximate Nash equilibria of imperfect-information games from self-play. In [13], the authors used deep reinforcement learning method to generate fighting game agents. It demonstrated that the agent outplayed other AIs in the game with 94.4% win rate.

The learning process is in an environment with an agent which is part of the learning process and the agent makes an action in the environment to finish a goal. After that, there is a reward or punishment system according to the action or depending on the algorithm used. the agent tries to get more rewards and avoids being punished by changing its weights to get a great construction of the task. At the end, the agent predicts the best action to get the highest reward and achieve the objective. Reinforcement learning algorithms are goal-oriented algorithms. They have a goal to achieve or a value to maximize at the end of an episode. Those algorithms aim to teach a task to an agent by penalizing them if the agent makes a wrong choice, and giving reward if its decision is a good one. These algorithms surpass humans and also champions in games. The success of reinforcement algorithms increased rapidly and they are now able to play Atari games and also some 3D games that compete with humans.

Your Methods

Proximal Policy Optimization (PPO) with Unity ML-Agents

The PPO-clip algorithm can be describe like this [12]

Algorithm 5 PPO with Clipped Objective

Input: initial policy parameters θ_0, clipping threshold ϵ
for $k = 0, 1, 2, \ldots$ **do**
Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
Compute policy update

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}_{\theta_k}^{CLIP}(\theta)$$

by taking K steps of minibatch SGD (via Adam), where

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathop{\mathrm{E}}_{\tau \sim \pi_k} \left[\sum_{t=0}^{T} \left[\min(r_t(\theta)\hat{A}_t^{\pi_k}, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t^{\pi_k}) \right] \right]$$

end for

- θ is the policy parameter
- E^t denotes the empirical expectation over timesteps
- Rtr_t is the ratio of the probability under the new and old policies, respectively
- A^t is the estimated advantage at time ttt
- ε is a hyperparameter, usually 0.1 or 0.2

SAC

Before SAC was proposed, deep reinforcement learning (DRL) algorithms had achieved significant results in continuous action space control tasks, but each had some shortcomings. The following first introduces the classic achievements of DeepMind and OpenAI in the field of continuous control.

The SAC algorithm is an Off-policy algorithm. The sample efficiency has been improved comparing to the on-policy algorithm such as PPO. SAC is a random policy algorithm when compared with DDPG and its variant D4PG.

The SAC algorithm is constructed under the framework of Maximum Entropy Reinforcement Learning. The purpose is to make the strategy randomize. The advantage is that it is very friendly to robot control problems and can even be used in a real environment.

The maximum entropy of the strategy also means that the exploration of the strategy space and the trajectory space is more sufficient than the deterministic algorithm. For the state with more than one optimal action, the SAC can output the probability distribution of an action instead of a certain action. The Q function is defined as

Q^π is changed to include the entropy bonuses from every timestep *except the first*:

$$Q^\pi(s,a) = \mathop{\mathrm{E}}_{\tau\sim\pi}\left[\sum_{t=0}^{\infty}\gamma^t R(s_t,a_t,s_{t+1}) + \alpha\sum_{t=1}^{\infty}\gamma^t H\left(\pi(\cdot|s_t)\right)\middle| s_0=s, a_0=a\right]$$

With these definitions, V^π and Q^π are connected by:

$$V^\pi(s) = \mathop{\mathrm{E}}_{a\sim\pi}\left[Q^\pi(s,a)\right] + \alpha H\left(\pi(\cdot|s)\right)$$

and the Bellman equation for Q^π is

$$\begin{aligned} Q^\pi(s,a) &= \mathop{\mathrm{E}}_{\substack{s'\sim P\\ a'\sim\pi}}\left[R(s,a,s') + \gamma\left(Q^\pi(s',a') + \alpha H\left(\pi(\cdot|s')\right)\right)\right] \\ &= \mathop{\mathrm{E}}_{s'\sim P}\left[R(s,a,s') + \gamma V^\pi(s')\right]. \end{aligned}$$

We use the ML-Agents to train our agents.

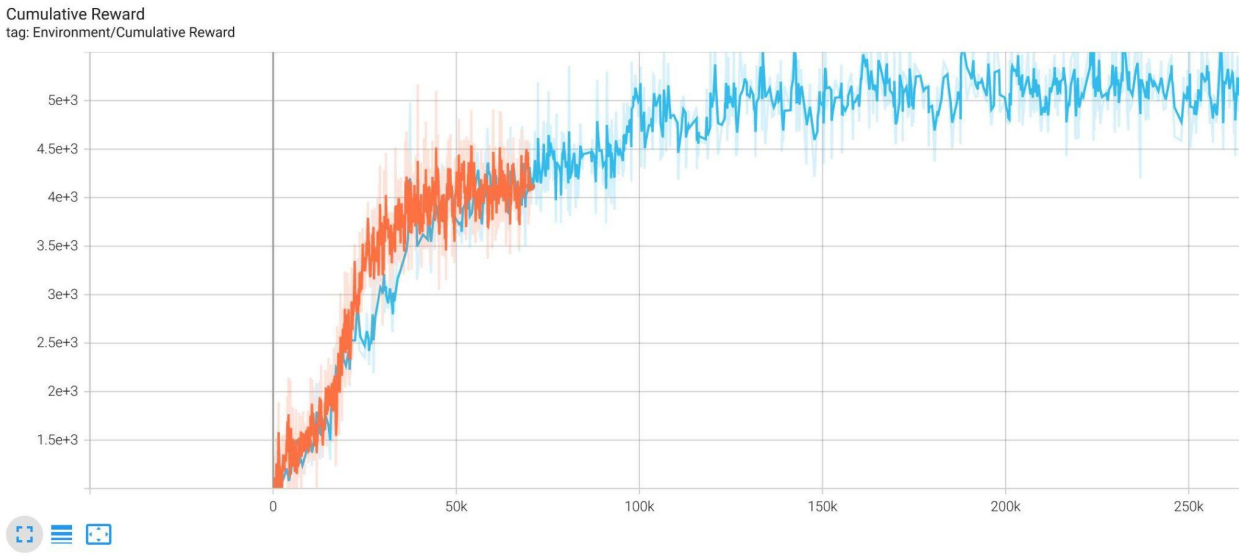
The command mlagents-learn config/ppo/ppo.yaml --run-id=pb_01 will start our training process


unity
Version information:
ml-agents: 0.27.0,
ml-agents-envs: 0.27.0,
Communicator API: 1.5.0,
PyTorch: 1.7.1+cu110
[INFO] Listening on port 5004. Start training by pressing the Play button in the Unity Editor.

Results

Blue bubble Scenario

Training curve



To investigate the performance of our agent, this study measured the performances of the agents in our special designed game levels. The levels serviced by bubble crush were evaluated by comparing the average score earned by random policy AI and the RL Agent to complete the level within 20 moves.

The game board is 10x20 for all training and testing levels.

The training curve is shown in the above figure. The only difference between the blue and orange curve is the observation type used during training. The blue curve uses the vector type while the orange curve uses the compressed visual type in match 3 sensor. When using a compressed visual type, the agent will learn faster and converge faster than the vector type. In the 25k episode, the orange agent is able to achieve nearly 4k reward while the blue agent can only get 3k. This will help us save a lot of training time by using the compressed visual setting.

TABLE I
Average score in blue bubble scenario

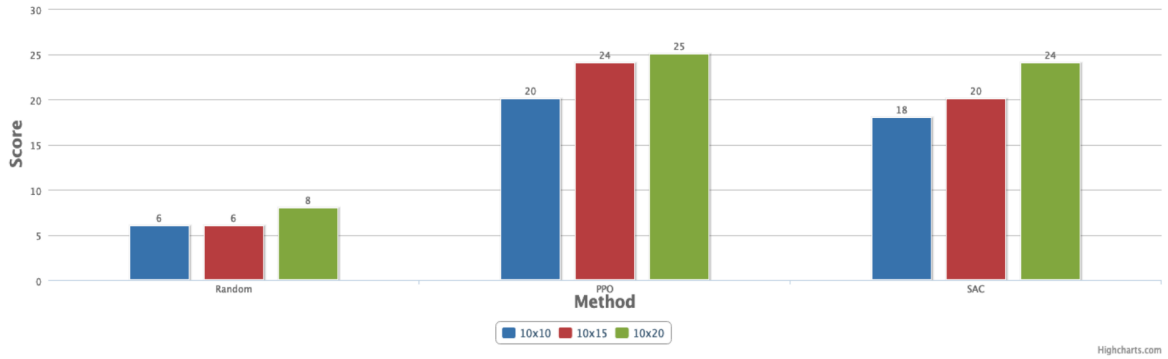
Level	Random AI	PPO
Level 1	8	25
Level 2	15	39
Level 3	10	28
Level 4	18	43

Scores:

An agent trained by PPO algorithm outperforms random policy twice in average score shown in table I. In test level 1, 2, 3, 4, our PPO algorithm will choose three blue bubbles to destroy in 2 moves, while the random policy AI will require 5 moves or more to choose a blue bubble. This result shows how our agent learns the importance of blue bubbles which will give the agent much higher rewards.

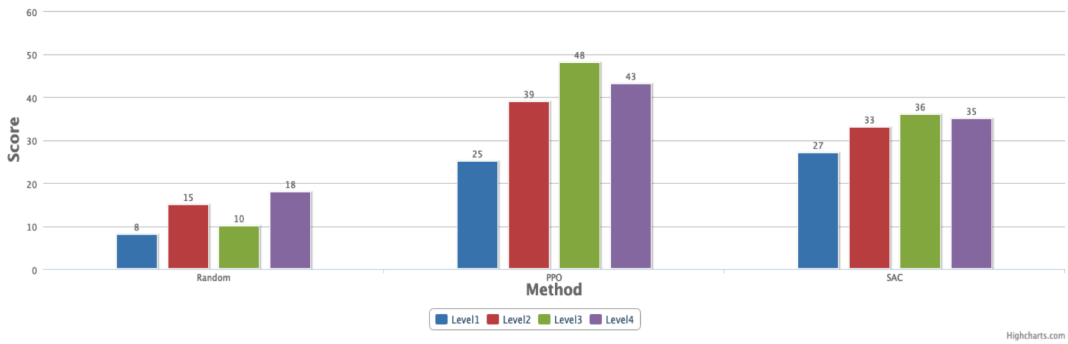
Blue Bubble Mode Results

Different map size



Blue Bubble Mode Results

Different level difficulty (10x20 map, 10~20 moves)



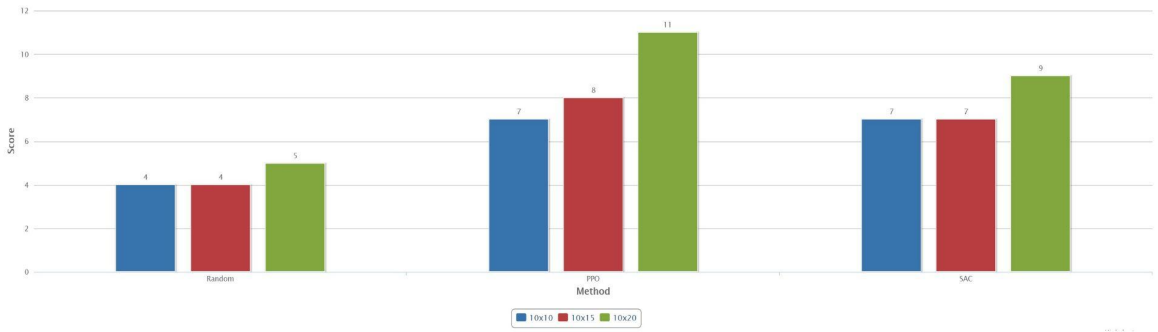
In the blue bubble mode, we can clearly see that the PPO and SAC outperforms random AI three times in average score and PPO performs slightly better than SAC. The larger the map size, the better score that PPO and SAC can get.

special bubble mode

In special bubble mode, some bubbles are marked with a white circle. The goal is to destroy as many special bubbles as you can. Note that the MLAgent allows AI to clearly identify which bubbles are special, and all special bubbles are randomly generated.

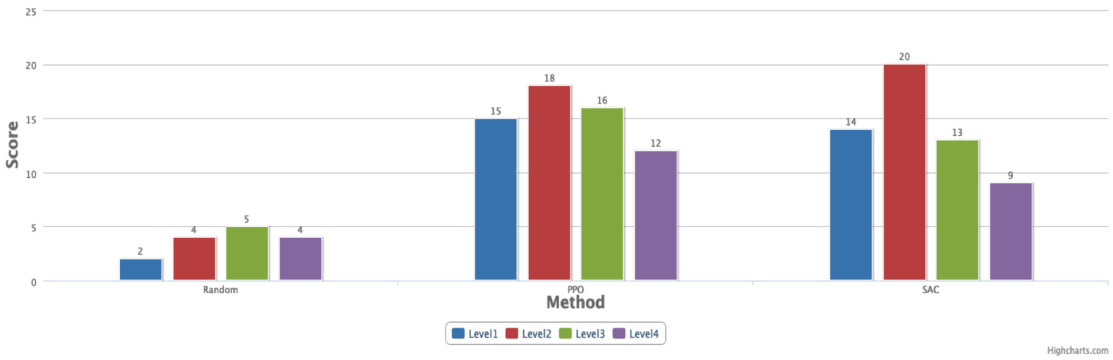
Special Bubble Mode Results

Different map size



Special Bubble Mode Results

Different level difficulty (10x20 map, 10~20 moves, 15~25 special bubbles)



In the special bubble mode, the PPO and SAC outperforms random AI twice in average score and PPO also performs slightly better than SAC. Although SAC requires continuous action

space, an alternate version of SAC, which slightly changes the policy update rule, can be implemented to handle discrete action spaces.

Sample Efficiency

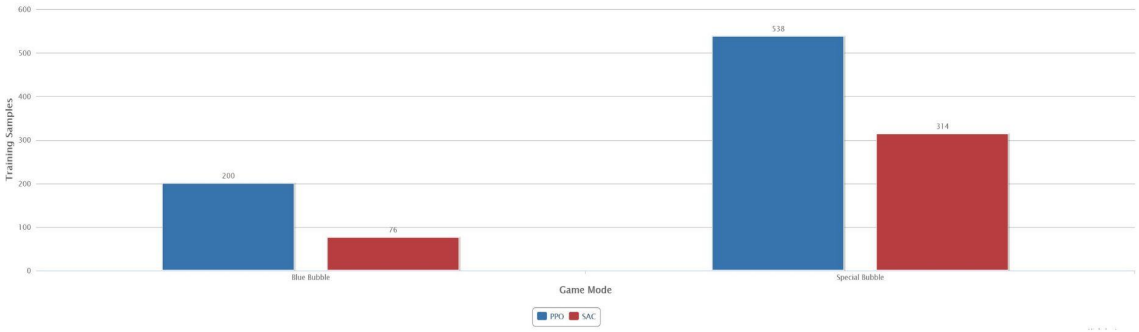
Sample Efficiency is very important in deep learning algorithms and learning skills in the match3 game can take a substantial amount of time. The total time required to learn a new game skill quickly adds up when trying different game scenarios. SAC is an off-policy algorithm so we can reuse data collected for another task. In a typical scenario, we need to adjust parameters and shape the reward function when prototyping a new task, and use of an off-policy algorithm allows reusing the already collected data.

PPO vs SAC

Training time

PPO vs SAC

Training samples



The above figures shows that the SAC requires 50\% less training samples to achieve the same performance with PPO in both blue bubble and special bubble mode. Also, since the SAC uses

fewer samples, the training time is also better than PPO. However, PPO can usually get better results when the reward function is carefully designed.

Conclusion and Future Work

In a word, this article presented a specially designed Match-3 game using Unity ML agent to study deep reinforcement learning similar to the one played by millions of people every day. The article described the bubble crush game in terms of reinforcement learning and carried out experiments for different test levels. The experiments have shown that our reinforcement learning agent achieved nearly 2 times better results than our simple random policy AI. Currently, when added bomb and lighting bubbles, both PPO and SAC do not know how to take advantage of those bubbles, and the results are nearly the same with random policy AI. Future improvements of the bubble crush environment are creating different sizes of levels, adjusting reward functions to teach AI to use bomb and lighting bubbles, and providing hints for human players.

References

[1] E. R. Poromaa, Crushing Candy Crush: Predicting HumanSuccess Rate in a Mobile Game using Monte-Carlo Tree Search',Dissertation, 2017.

[2] I. Kamaldinov and I. Makarov, "Deep Reinforcement Learning In Match-3 Game," 2019 IEEE Conference on Games (CoG),2019, pp. 1-4, doi: 10.1109/CIG.2019.8848003.

[3] Lorenzo, F. V., Asadi, S., Karnsund, A., Wang, T., Payberah,A. H. Generalized Reinforcement Learning for Gameplay.

[4] Shin, Y., Kim, J., Jin, K., Kim, Y. (2020). Playtesting in Match3 Game Using Strategic Plays via Reinforcement Learning.IEEE Access, PP(99), 1–1. doi: 10.1109/ACCESS.2020.2980380

[5] Kristensen, J. T., and Burelli, P. (2020). Strategiesfor Using Proximal Policy Optimization in MobilePuzzle Games. arXiv, 2007.01542. Retrieved from https://arxiv.org/abs/2007.01542v1

[6] Napolitano, N. (2020). Testing match-3 video games with DeepReinforcement Learning. arXiv, 2007.01137. Retrieved from https://arxiv.org/abs/2007.01137v2

[7] Holmgård, C., Green, M. C., Liapis, A., and Togelius, J. (2018).Automated Playtesting with Procedural Personas throughMCTS with Evolved Heuristics. arXiv, 1802.06881. Retrieved From https://arxiv.org/abs/1802.06881v1

[8] Clark, C., and Storkey, A. (2014). Teaching Deep ConvolutionalNeural Networks to Play Go. arXiv, 1412.3409. Retrieved from https://arxiv.org/abs/1412.3409v2

[9] K. Shao, D. Zhao, Z. Tang and Y. Zhu, "Move prediction inGomoku using deep learning," 2016 31st Youth Academic An-nual Conference of Chinese Association of Automation (YAC),2016, pp. 292-297, doi: 10.1109/YAC.2016.7804906.

[10] Heinrich, J., and Silver, D. (2016). Deep Rein-forcement Learning from Self-Play in Imperfect-Information Games. arXiv, 1603.01121. Retrieved fromhttps://arxiv.org/abs/1603.01121v2

[11] D. -W. Kim, S. Park and S. -i. Yang, ”Mastering FightingGame Using Deep Reinforcement Learning With Self-play,”2020 IEEE Conference on Games (CoG), 2020, pp. 576-583,doi: 10.1109/CoG47356.2020.9231639.
[12] Schulman, J., Wolski, F., Dhariwal, P., Radford, A.,and Klimov, O. (2017). Proximal Policy OptimizationAlgorithms. arXiv, 1707.06347. Retrieved fromhttps://arxiv.org/abs/1707.06347v2