# Plants vs. Zombies: Reinforcement learning to a tower defense game

Timothée Blondiaux, Lukas Frank, Hanady Gebran, Alexandre Perot
https://github.com/inf581-pvz-anonymous/pvz_rl

***Abstract*—We developed a *Plants vs. Zombies* game engine, linked it to an OpenAI gym environment and used state-of-the art Reinforcement Learning methods (policy gradient with and without an actor critic module, Deep Q-Network and Double Deep Q-Network) to successfully train agents to play the game. We reported on the superiority of off-policy methods for this environment and how hyperparameters and observation features could be handled to improve the agent's performances.**

## I. Introduction

Plants vs. Zombies[1] (PvZ) is a single player tower-defense game in which the player has to defend their home amid a zombie apocalypse. To this end, they may locate special plants in their garden to fight against attacking foes. This setup is well suited to a Reinforcement Learning (RL) approach since it can be stripped to a simple baseline while enabling the addition of components (plant types, obstacles...) to increase complexity arbitrarily. Furthermore, the game structure is mostly discrete, facilitating the definition of state and action spaces.

However, the game poses several challenges for RL research. Firstly, the observation and action spaces quickly reach enormous sizes even for a small number of plant types. Therefore, [1] classified a similar game based on the PvZ-setup as a hard game for RL. Secondly, the original game only remunerates the player for winning a level, turning the reward matrix sparse. Hence, an adapted reward scheme may be required. Thirdly, the set of available actions depends on the state of the game, i.e. some actions may be illegal at some point.

Though PvZ offers interesting challenges to tackle, it had not been subject to extensive RL research before us. Hence, our contribution was threefold. We developed a highly compatible PvZ environment using OpenAI `gym` (and `pygame` for visualization). Furthermore, we created several agents on the PvZ environment using RL approaches such as Deep Q-Networks and policy gradient methods, and lastly we provided a detailed empirical comparison of the different approaches.

## II. Background and Related Work

For our research project, we are interested in four state-of-the-art RL approaches.

### A. Monte Carlo policy gradient method (REINFORCE)

The policy gradient theorem [2] enables efficient estimation through the gradient of the performance metric $J(\theta) = \mathbb{E}_\pi(\sum_{t=1}^{H} R_t)$, requiring only the gradient of the policy probability estimate:

$$\nabla_\theta J(\theta) = \sum_{s\in S} d^\pi(s) \sum_{a\in A} Q^\pi(s,a)\nabla_\theta \pi_\theta(a|b).,\qquad (1)$$

Therefore, policy gradient agents are trained through stochastic gradient ascent.

### B. Deep and Double Deep Q-Networks

The basic idea of Deep Q-Networks (DQN) is the same as in traditional Q-Learning approaches [3]. In particular, the agent observes a state $s$ and chooses an action $a$ from the action space $\mathcal{A}$. Next, it receives a reward $r$ and reaches a new state $s'$, in which it may choose a new action $a' \in \mathcal{A}$. The value $Q$ of action $a$ in state $s$ can be written as

$$Q(s,a) = r + \gamma \max_{a'\in\mathcal{A}} Q(s',a'),\qquad (2)$$

with the discount rate $\gamma$. The Q-table may thus be obtained by an iterative procedure, e.g. gradient ascent:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'\in\mathcal{A}} Q(s',a') - Q(s,a)\right],\qquad (3)$$

with the learning rate $\alpha$. In contrast to traditional Q-learning, DQN rely on deep neural networks $q_\mathbf{w}$ to learn $Q$ [4] [5]. Therefore, $Q$ is updated by deriving the squared error with respect to the weights $\mathbf{w}$ instead of $Q$:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\left[r + \gamma \max_{a'\in\mathcal{A}} [q_\mathbf{w}(s')]_a - [q_\mathbf{w}(s)]_a\right] \nabla_\mathbf{w}[q_\mathbf{w}(s)]_a.\qquad (4)$$

There are some limitations to DQN [2] [6]. Firstly, the target is non-stationary, which may lead to a low exploration of the state space. One may mitigate this problem by applying an $\varepsilon$-greedy strategy, see Section V-A.

A second issue might arise from the overestimation of Q-values [2] which can be addressed by Double Deep Q-Networks (DDQN). DDQN use a second network, the so-called target network, to estimate the Q-values of the next state-action pairs $Q(s',a')$. Therefore, compared to DQN, only the target changes slightly and the weight update evolves to

$$\mathbf{w_p} \leftarrow \mathbf{w_p} + \alpha\left[r + \gamma \max_{a'\in\mathcal{A}} [q_{\mathbf{w_t}}(s')]_a - [q_{\mathbf{w_p}}(s)]_a\right] \nabla_{\mathbf{w_p}}[q_{\mathbf{w_p}}(s)]_a.\qquad (5)$$

[1]https://www.ea.com/ea-studios/popcap/plants-vs-zombies

with the original policy net weights $\mathbf{w_p}$ and the newly introduced target net weights $\mathbf{w_t}$. In this manner, DDQN decouple the action choice from the actual target [2]. The target network is typically identical to the original one but set such that it updates its weights more slowly.

### C. Actor-Critic method

Actor-Critic methods combine elements of both prior approaches. The 'actor' model executes an action based on a neural net approximating the optimal policy (as in REINFORCE). The 'critic' model judges the actor's policy by estimating a value function with another neural net (similar to Deep Q-Networks) [2]. Although already trained agents will only behave based on the learned policy, this trick reduces gradient variance during the training phase, which makes the policy update more accurate. This improvement comes at the cost of slower computation, since we now have two models to train.

## III. The Environment

### A. The game

PvZ is a tower defense game. The field consists of a fixed number of lanes that are divided into single tiles. The zombies enter randomly on the right side and try to reach the left. The player can use their resources, namely *suns*, to buy plants and place them on the tiles. In doing so, they have to respect the cooldown[2] of each plant purchase and the fact that only one plant can occupy a given tile at the same time.[3]

In total, we included four different plant types. The *sunflower* supplements the natural sun generation of the environment and thereby increases the player's purchasing power. The *peashooter* attacks zombies on the same lane by shooting peas on them. The *wall-nut* does not attack zombies but has a lot of health points (HPs) and thus blocks the zombie inrush for a certain amount of time. The *potatomine* activates after a few seconds and then explodes as soon as a zombie enters its tile, causing great damage. Furthermore, we defined four different types of zombies, varying only in their amount of HPs.[4] Details on our game implementation can be found in Appendix VIII-B.

### B. State observation

With respect to these key elements, we constructed the *observed* state space $\mathcal{S}^o$ in the following manner. At state $s \in \mathcal{S}^o$, we let the agent observe the grid location of all plants (including the locations of 'no plant'), the availability of a plant type (requiring enough suns and having the plant purchase's cooldown phase over), the number of suns that the agent has, and the grid with the HP sum of all zombies per tile.[5] Hence, we can characterize the state by a vector with $2 \times \textit{\#lanes} \times \textit{\#tiles per lane} + \textit{\#plant types} + 1 = 95$ entries.

### C. Rewards

As mentioned in section I, the original reward definition is very sparse since it only remunerates the player he wins the level. We therefore adapted the game slightly to our needs. Instead of pre-defining a level, we constructed a random zombie spawner that generates zombie waves with increasing difficulty. This allowed us to play a *survival mode* in which the success of the agent is measured differently. In particular, the agent gets rewarded for every zombie killed (stronger zombies give higher scores when killed).

### D. Step function and game balance

The step function associated with this environment consists of running the game until at least another action than doing nothing is available. Then the accumulated score during this interval and the observation of the last frame are returned. The frames per second (frame rate) thus determine the maximum number of actions of the agent per second. We used a low frame rate of 2 frames per second to increase the training speed. This was deemed not to be too restrictive in comparison to human behavior.

The right game balance was difficult to find. Our first benchmark was on a fast-paced wave. In this version, the zombies appear after 5 seconds and thereafter every 4 seconds. Any time a zombie appears, its type is random. For instance, before the start of the first wave, there is a 5% probability to encounter the bucket zombie (HP: 1100), 10% to have the zombie cone (HP: 560) and 85% to have the basic zombie (HP: 190).

After the first 50 seconds of play a wave of zombies occurs. This corresponds to the simultaneous appearance of 6 zombies: One in each line with the same chances as previously, as well as one flag zombie on the first line leading the wave. As the waves go on, the difficulty increases. The probability of getting non-basic zombies is doubled after each wave, until we reach the 5th wave in which there are only bucket zombies. A wave appears every 20 zombie generations (80 seconds) .

However, we realized that unlimited time (or in general, high time limits) encouraged risky "all-or-nothing" strategies based on gambles (for example by overcrowding a lane where the agent thinks that more zombies will spawn). For example, the REINFORCE algorithm strives to improve the expected value of the total reward. Therefore, having a small chance of getting very high rewards can be valued more than a safer policy with lower rewards.

Hence, for the second benchmark, we chose to restrict ourselves to a single wave of zombies. The second benchmark is a slow-paced version in which the natural production of sunlight is twice as slow, but in return the time frame for the emergence of zombies and waves is doubled. A consequence

[2]That is, the player has to wait between two purchases of the same plant type for a predefined amount of time.

[3]Note that this occupation restriction does not include zombies. Several zombies can be located at the same tile at the same time and the player may even place a plant on these tiles.

[4]That is, they all have the same speed and the same attack strength, but some are more resistant than others.

[5]The total HP is relevant enough because it is the only difference between zombies, even though it doesn't account for the fact that two mid-life zombies causes twice as much damage as a single full-life zombie

of this choice is that agents rely less on chance because there is a lower urgency, and they have longer-term strategies.

The presented environment requires the agent to learn the different roles of the respective plant types, e.g. resource generation, defense, and offense. The utility of a plant type depends highly on its location and the location of other plants. Moreover, there is a strong delay between the action and its rewards that the agent has to understand. Such tasks are essential for a vast range of challenging video games to which our results may apply.

## IV. The Agent

The sizes of both the observation and action spaces are very significant (our agents were fed a 95-dimension observation and decided on an action space of dimension 181). Because of that, we were drawn to neural network based agents. Thus we decided to focus on four state-of-the-art architectures for RL agents, presented in [2] which are policy gradient (REINFORCE algorithm), actor-critic methods, Deep Q-learning (DQN) and Double Deep Q-learning (DDQN).

### A. Policy gradient agent

We based our implementation of the REINFORCE algorithm off the following repository [7]. The probability estimator is a Multi-Layer-Perceptron (MLP) with a single hidden layer. Therefore, we could use `pytorch` and a negative log likelihood loss to compute the gradient.

While REINFORCE provided great results on the fast-paced version of the game, it performed rather poorly on the slower benchmark (see VI). The fast-paced benchmark is balanced in such a way that the sun currency is not a great constraint. Therefore, the agent does not have to make the choice of investing in *sunflowers* early on and only has to learn plant placement. In the slow-paced benchmark however, the policy regressor gets stuck in a local maximum of the performance metric and limits itself to the use of *potatomines*, which provide fast reward (they one-shot zombies) for a low sun cost, and *wall-nuts*, which are a short-term way to temporize and delay the loss of the game. Because no *sunflowers* are planted, probabilities of escaping are very low as *peashooters* are unaffordable.

These lackluster performances highlighted the need for better exploration. We adopted two parallel approaches. On one hand, we implemented an actor-critic module, to reduce variance in the hope of improving the training. On the other hand, we decided to use an $\epsilon$-greedy behavior to drive the gathering of samples and go off-policy. Instead of keeping a policy gradient agent (by adding an off-policy correction to the gradient as in [8]), we chose to focus on Deep Q-Networks (see IV-C).

### B. Actor-Critic agent

We drew our inspiration on both the basic actor-critic algorithm seen in class [2] and the pytorch implementation of an actor-critic agent from Soumith Chintala [9]. We decided to use two separate MLP's, one for the critic model and the other one for the actor model, each with a single hidden layer of size 80 with Leaky ReLU activation. A couple of tests with alternative net architectures confirmed that our choice was appropriate. However, given the slowness of training with such parameters, we could not optimize the architecture extensively.

### C. DQN agent

Our base DQN revolves around a MLP with a single hidden layer of size of 50. To overcome the issues of DQN mentioned in section II, we equipped our agent with a replay memory of minimal size 100 and drew random mini-batches of 200 $< s, a, r, s' >$ pairs, with state $s$, action $s$, reward $r$ and next state $s'$. In order to balance exploration and exploitation, we made use of an $\varepsilon$-greedy action choice, as explained in Section V-A.

### D. DDQN agent

Our DDQN agent uses the same base configuration as the DQN (see above). Both agents are based on code provided in [10]. As is common, the DDQN target network has the same structure as its policy network. We synchronize them every 2000 training steps.

## V. Improving performance

### A. Exploration

As Deep Q-Learning approaches (DQN, DDQN) often converge to a local minimum quickly, they may suffer from low state space exploration [2]. To overcome this issue, DQN and DDQN are typically combined with an $\varepsilon$-greedy action choice. That is, the agent deviates from its currently best policy with probability $\varepsilon$ and chooses a random action instead. The choice of $\varepsilon$ is crucial for the balance of exploration and exploitation: A high $\varepsilon$ allows for a broad search but might prevent the agent from converging to the optimal policy, whereas a low $\varepsilon$ may not enable the agent to escape local minima.

Reflecting the fact that high exploration is more valuable in the beginning of the training episode, $\varepsilon$ is often set in a decreasing fashion. Following [6], we tested three functional forms of the $\varepsilon$-decrease: a linear, an exponential and a sinusoidal decrease. Details on the functions can be found in Appendix VIII-A. As the DDQN agent appeared to perform better than the DQN, we focused our experiments on DDQN. It turned out that during a couple of tests, we couldn't identify a clear winner among the methods. Finally, we decided for the exponential form since it is a common choice in the literature [6]. Figure 1 depicts two of the test cases.

### B. Illegal actions

An important challenge of our environment is the handling of illegal actions.[6] While working on the fast-paced benchmark, we decided to simply ignore illegal inputs from the agents. However, we realized that from the agent's standpoint, such an approach creates a lot of uncertainty, especially since no action brings immediate reward. Therefore, we followed

[6]i.e. planting a plant on already occupied tiles, having not enough suns, or being in a cooldown phase.

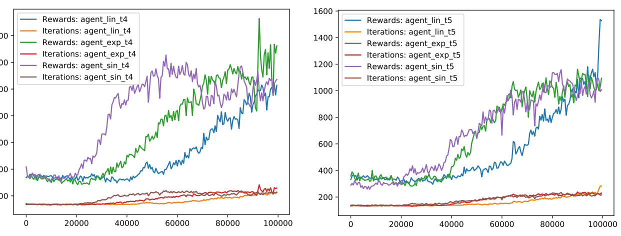

Fig. 1: Two exemplary performance tests of functional forms of $\varepsilon$ for the DDQN agent: linear vs. exponential vs. sinusoidal decrease. We depict total rewards as well as the iterations survived.

the approach used by Deepmind [11] and masked out[7] the illegal moves in the greedy action choices from the predicted Q-values both when submitting an action to the step function of the environment and when computing the DQN and DDQN loss.

### C. *Action asymmetry*

By trying to help the agent learn a suitable strategy for the more challenging slow-paced benchmark, we reflected on the asymmetry of the action space. When using a uniform behavior policy for the exploration case of the $\epsilon$-greedy policy, we do not give enough importance to the act of doing nothing (without masking, its probability is 1/181). Therefore, we changed the behavior policy to increase the probability of doing nothing during exploration. Through serendipity, we found out that the results were better when increasing the chances of doing nothing only during the initial pure exploration-based initialization of the replay memory. We present a plausible explanation in VI.

### D. *Observation features*

To limit the size of the observation space and make the training faster, we applied some preprocessing of the observations. Though feature engineering was not the main focus of our project, we conducted some successful and less successful experiments.

We fed the plant grid to the network as a single grid with the plant index on each cell instead of a one hot-encoding of the grid. This is usually bad practice. However, in our case, it provided better results due to the smaller dimension of the observations. Also, instead of feeding the whole grid of zombie HPs to our networks, we used the sum of the zombies' HP on each lane to obtain a scalar per lane. This is a limiting observation in the sense that the agent will not know where the zombies are on the lane.

Other transformations led to less successful results. Discussion on one of these methods (linear regression) and its results is available in the appendix VIII-D. Also, we found out that scaling the zombies' HP gave worse results, which is an expected behavior when scaling down the most important feature. Therefore, we did not apply scaling for our best agents.

[7]That is, after predicting the Q-values with the neural net, we check for illegal actions and set their Q-values manually to a minimum value.

TABLE I: Results on the slow-paced benchmark

| | Masked | Mean score | Mean iterations |
|---|---|---|---|
| REINFORCE | NO | 324.4 | 145.549 |
| REINFORCE | YES | 538.12 | 165.217 |
| Actor critic | NO | 678.68 | 165.894 |
| DQN | YES | 1651.36 | 306.288 |
| DDQN | YES | 1892.04 | 338.413 |

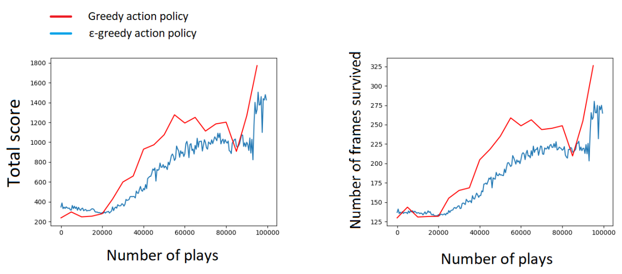

Fig. 2: Learning curves of the DDQN agent. The framerate was 2 and the end of the game was at 400 frames.

More generally, our observation space features symmetries (interchangeability of the lanes) and spatial domains (a lane behaves like a discretization of a segment). Therefore, a thorough feature engineering approach could greatly improve the proficiency and the training speed of the agent. However, we rather focused on the comparison of the four different agent types and did not tackle feature engineering in-depth in our project.

## VI. Results and Discussion

### A. *Results*

On the fast benchmark (with high amounts of sun currency), all agents displayed good plant placement such as planting *peashooters* as far as possible on the left of the lanes, or accumulating as many zombies as possible with a plant in front of a *potatomine* to kill them all at once afterwards. These are efficient human strategies. On the slower benchmark (see TABLE I), except for the gradient policy agent, they all managed to learn the importance of sunflowers. The difference of playstyle between the policy gradient agent and the DDQN agent is further illustrated in the appendix. DQN and DDQN show very similar, fairly strong results, even in the slow-paced environment. They are better than actor-critic, most likely due to deeper exploration. As seen in Figure 3, DDQN wins almost 65% of the time. Though the shape of the learning curves in Figure 2 first might imply that further training could be beneficial, retraining the agent did not yield better results.

### B. *Input features*

We evaluated Shaply Additive exPlanations (SHAP) values for our trained DDQN agent (see Figure 4) using the repository

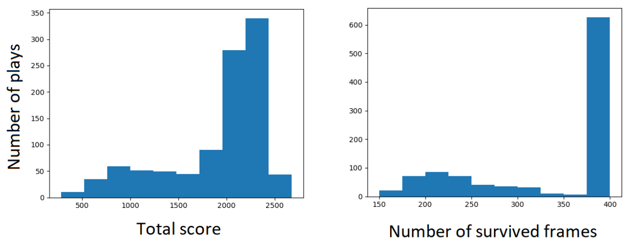

Fig. 3: Histograms of the performances of the trained DDQN agent on 1000 plays.

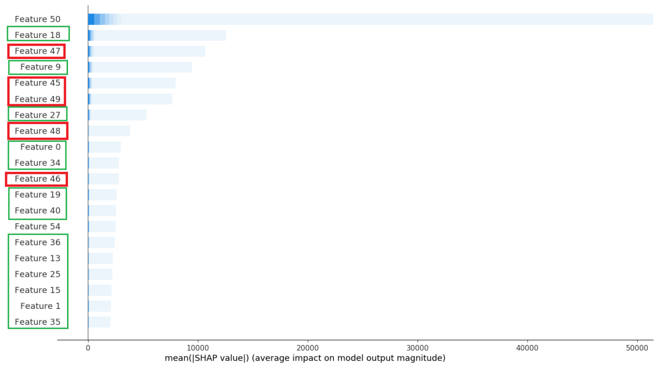

Fig. 4: SHAP value of the different input features for the DDQN agent. In red: information on zombies on each lane. In green: the plant grid. Feature 50 is the sun amount and feature 54 the availability of the potatomine.

[12]. The action asymmetry is illustrated once again but we can also see that the information concerning the zombies and the suns is quite important. Moreover, the plant grid cells at the beginning of the lanes (multiples of 9) are also important. This is interesting because they are a way for the agent to detect dangerous zombies advances without having access to their position (a plant disappearing at these cells means that a zombie is near the end). Such disparity in the input's importance, in addition to the previous considerations, further establishes feature engineering as an opportunity to make training more efficient.

## C. *Masking illegal moves*

Masking illegal moves (as explained in Section V-B improved performances for all agents on which it was used. For example, for two DDQN agents trained on 100 000 plays (using a sinusoidal epsilon function, a batch size of 32 and a grid with only 3 lanes) we obtained an average score of 811.4 for the unmasked version, while the masked version scored 1291.64 in average.

## D. *Action asymmetry*

As explained before, our best results were obtained when increasing the probability of doing nothing only during exploration in the first episodes. This is most likely a result

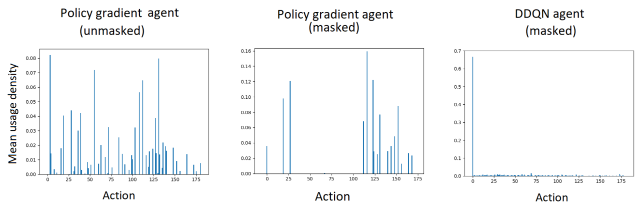

Fig. 5: Action used by policy gradient (unmasked actions) and DDQN (with masked actions) agents. Action 0 means doing nothing, then action id is $1+id_{plant}+n_{plants}*(lane+n_{lanes}*pos)$.

of masking. Indeed, as we can see on Figure 5, masking reduces noise in the agent's decisions and the agent learns the importance of doing nothing and uses it pretty often. Therefore, it may no more be necessary to push exploration in that direction as much. By keeping the correction for too long, we may prevent the network from learning how to use the other actions well.

## E. *The tuning of hyperparameters*

One of the biggest challenge we encountered was the tuning of the many hyperparameters and degrees of freedom of our implementation such as the reward function, the architectures of the networks, training parameters (number of plays, learning rates, size and frequency of the updates), the action asymmetry correction, and the $\epsilon$ function for the $\epsilon$-greedy behavior policy. Because of the significant number of iterations needed to observe results (especially since early high scores are not an indicator of a suitable agent, as seen with policy gradient), we could not use grid search to optimize these parameters. Hence, we proceeded empirically to find a satisfying, yet most likely suboptimal, set of parameters. In addition to that, the game balance is another degree of freedom and creating a suitable benchmark required its fair share of trial and error.

## VII. Conclusion and Future Work

After developing a realistic *Plants vs. Zombies* environment compatible with OpenAI `gym`, we were able to train a variety of Reinforcement Learning agents. We obtained very promising results, with a DDQN algorithm beating a self-designed, randomly generated, high difficulty level 65% of the time. In order to improve the performances, we identified main areas of research which are proper feature engineering (using feature importance, space symmetries and space topology to pre-process the observations) and tuning of the many hyperparameters.

In addition to that, other ideas could be tried, for instance involving CNN's to work directly with the game visual window as an input and let the network learn important features by itself.

The study of a tower defense game such as PvZ presents great potential for RL research; consequently, the results exhibited in this report may serve as a starting point for future research.

## References

[1] Bontrager, P., Khalifa, A., Mendes, A., & Togelius, J. (2016). Matching Games and Algorithms for General Video Game Playing. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 12*(1). Retrieved from https://ojs.aaai.org/index.php/AIIDE/article/view/12884

[2] As mentioned in Lecture VII - Reinforcement Learning III. *INF581 Advanced Machine Learning and Autonomous Agents*, 2021.

[3] As mentioned in Lecture VI - Reinforcement Learning II. *INF581 Advanced Machine Learning and Autonomous Agents*, 2021.

[4] Mnih, Volodymyr, et al. ”Playing atari with deep reinforcement learning.” *arXiv preprint arXiv:1312.5602* (2013).

[5] Mnih, Volodymyr, et al. ”Human-level control through deep reinforcement learning.” *nature* 518.7540 (2015): 529-533.

[6] Chuchro, Robert, and Deepak Gupta. ”Game playing with deep q-learning using open-ai gym.” *Semantic Scholar* (2017).

[7] Mathias Sundholm ”Open AI Gym Zoo” https://github.com/maitek/openai-gym-zoo/blob/master/pytorch_agent.py

[8] https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html

[9] Soumith Chintala ”Actor Critic Implementation” https://github.com/pytorch/examples/blob/master/reinforcement_learning/actor_critic.py

[10] DataHubbs https://www.datahubbs.com/double-deep-q-learning-to-get-the-most-out-of-your-dqn/

[11] Deepmind ”Open Spiel” https://github.com/deepmind/open_spiel/tree/b1b321a57754df76309ed9d7ac3c9b449ed34901

[12] Slundberg ”SHAP” https://github.com/slundberg/shap

## VIII. Appendix

### A. *Functions for the $\varepsilon$-decrease*

Following [6] we tested three functional forms for the decrease in $\varepsilon$. Equations (6)-(8) show the linear, exponential and sinusoidal functions with the value of $\varepsilon$ in the first and in the last training period $\varepsilon_{start}$ and $\varepsilon_{end}$, the number of training episodes $n$, the current training episode $x$ and the number of sinusoidal periods $f$:

$$\varepsilon = \varepsilon_{start} + \frac{\varepsilon_{end} - \varepsilon_{start}}{n} \cdot x \tag{6}$$

$$\varepsilon = \varepsilon_{start} \cdot b^x, \quad \text{with } b = \left(\frac{\varepsilon_{end}}{\varepsilon_{start}}\right)^{1/n} \tag{7}$$

$$\varepsilon = \varepsilon_{start} \cdot b^x \cdot 0,5 \left[1 + \cos\left(\frac{2\pi f \cdot x}{n}\right)\right], \tag{8}$$

with $b = \left(\frac{\varepsilon_{end}}{\varepsilon_{start}}\right)^{1/n}$. Figure 6 illustrates the functional forms.

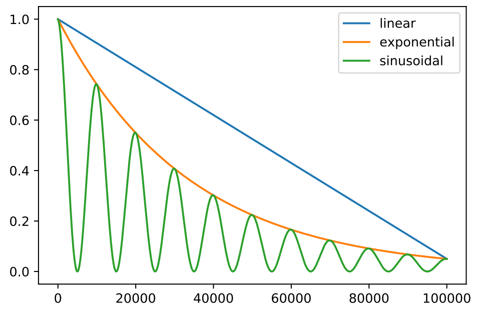

Fig. 6: Functional forms of the decrease in $\varepsilon$.

### B. *Game implementation*

In order to be able to adapt the environment in a flexible way, we implemented a remake of PvZ from scratch. To prevent us from ’engineering’ the game to fit our agents, we tried to remain true to the original game structure.[8] To this end, we defined three classes of field elements: Plants, Zombies and Projectiles. Hence, one can easily extend the game by creating a new plant type with a new attack mechanism. As explained in the main part, we included four distinct plants: *sunflowers*, *peashooters*, *wall-nuts*, and *potatomines*.

TABLE II: Plant types and characteristics

| | Cost | HP | Attack | Sun prod. | Cooldown |
|---|---|---|---|---|---|
| Sunflower | 50 | 300 | 0 | 25 | 5 |
| Peashooter | 100 | 300 | 20 | 0 | 5 |
| Potatomine | 25 | 300 | $\infty$ (once) | 0 | 20 |
| Wall-nut | 50 | 4000 | 0 | 0 | 20 |

In order to make the game compatible for future RL research, we relied on OpenAI `gym`[9] as an interface between

[8]Details from https://plantsvszombies.fandom.com/wiki/Main_Page were very helpful to us.

[9]https://gym.openai.com/

the agents and the environment. As visual results can be interpreted more easily by a human, we added a `pygame`[10] interface on top of it. We visualized the objects using the resource database of the PvZ fandom.[11] This allowed us to display the agent's gameplay and detect its weaknesses quickly. Figure 7 and Figure 8 show two exemplary game situations.

## C. Comparison of the gamestyles of different agents

As mentioned in the main part, the different agents developed different gamestyles. Figures 7, 8, and 9 illustrate that behavior.

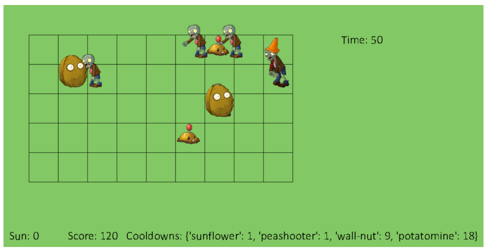
Time: 50
Sun: 0
Score: 120 Cooldowns: {'sunflower': 1, 'peashooter': 1, 'wall-nut': 9, 'potatomine': 18}

Fig. 7: Policy gradient agent playstyle. Long-term investments such as sunflowers are ignored, hence the scarcity of sun, the small number of plants and the absence of expensive but powerful peashooters.

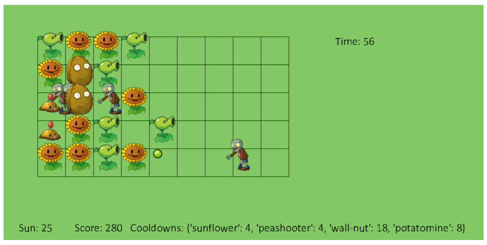
Time: 56
Sun: 25
Score: 280 Cooldowns: {'sunflower': 4, 'peashooter': 4, 'wall-nut': 18, 'potatomine': 8}

Fig. 8: DDQN gamestyle. The back of the field is much more populated than the front. The potatomine tends to be protected by other plants to accumulate zombies in front of it and maximize the deadliness of its explosion. Sunflowers are a priority of the agent in the early game.

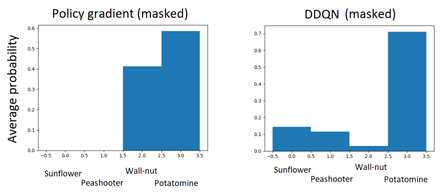

Fig. 9: Usage of the different available plants by a policy gradient agent and a DDQN. The policy gradient agent completely ignores sunflower and peashooter as stated previously. Potatomines are highly used by both agents because they are a low price single use item that therefore needs to be replaced often, despite a high cooldown.

[10]https://www.pygame.org/
[11]https://plantsvszombies.fandom.com/wiki/Main_Page

## D. Zombie data preprocessing

Because we identified symmetries in the observation space, we tried to use a simple linear regression to each lane as a preprocessing of the zombie grid. We also tried hand-crafting other types of observations such as a sum weighted towards the end of the lane (inspired by what was learned by the linear regression) or a couple containing the sum of the zombie's hp and the progress of the zombie on the lane. Both of these attempts provided underwhelming results, however the regression showed improving results with longer training, showing that the lack of encouraging results may have just been a symptom of an incomplete training due to an increased number of parameters.

TABLE III: DDQN with linear regression

| | Plays | Mean score | Mean iterations |
|---|---|---|---|
| DDQN | 1K | 1892.04 | 338.413 |
| DDQN with regression | 1K | 1057.8 | 234.401 |
| DDQN with regression | 1.5K | 1659.08 | 311.011 |

The learned parameters of the regression trained to be applied to the grid containing zombie HPs, lane per lane (before being fed to the our DDQN agent) are the following:

$$[-6.4347, -0.8138, -0.3569, -0.1997, -0.1913, -0.1612,$$

$$-0.1236, -0.0794, -0.0613],\ bias : 11.0534$$

Clearly, this is heavily weighted towards the end of the lane (if a zombie reaches 0, the game is lost).