Implementation – Alastair R.D.

All code found in - https://github.com/Alastairrd/GS-2024-Final/tree/main/Implementation

Project Summary

The overall goal of the project is to assess the impact that training on procedurally generated content has on various reinforcement learning models, and how to improve their performance when generalising to unseen content.

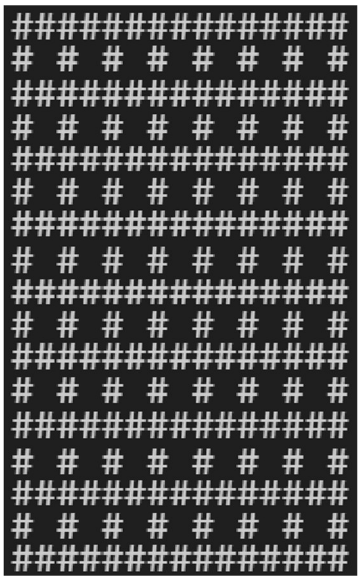
The project implementation covers a system for procedurally generating 15x15 'grid-world' levels, a custom gymnasium environment for the models to interact with, and a pipeline to easily build, train and analyse the performance of models.

Core Features

PCG

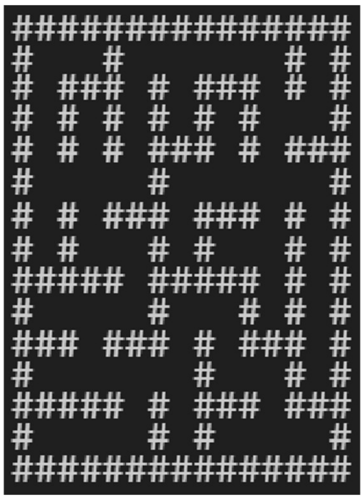
The procedural content generation pipeline was implemented first, making sure there were consistent training environments and allow reproducible testing for hyperparameter tuning and model evaluation. It uses a node-based network structure, supporting future graph-based algorithms and helping map design.

Levels are generated by first creating a lattice-like base map, with every second row and node designated as paths, and the rest as walls. A modified Kruskal's algorithm is used to generate a perfect maze as a foundation, with further algorithms introducing cycles to create imperfect mazes. Parameters control how cycles are added, adjusting space size and wall removal bias to shape the environment.



```
###############
# # # # # # # #
###############
# # # # # # # #
###############
# # # # # # # #
###############
# # # # # # # #
###############
# # # # # # # #
###############
# # # # # # # #
###############
# # # # # # # #
###############
# # # # # # # #
###############
```

***Figure 1** – Pre Kruskal level*

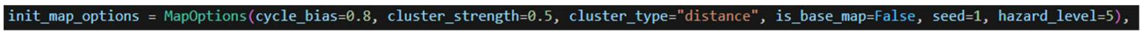


```
###############
#   #       # #
# ### # ### # #
# # # # # #   #
# # # ### # ###
#     #       #
# # ### ### # #
# #   # #   # #
##### ##### # #
#     #   # # #
### ### # ### #
#       #   # #
##### # ### ###
#     # #     #
###############
```

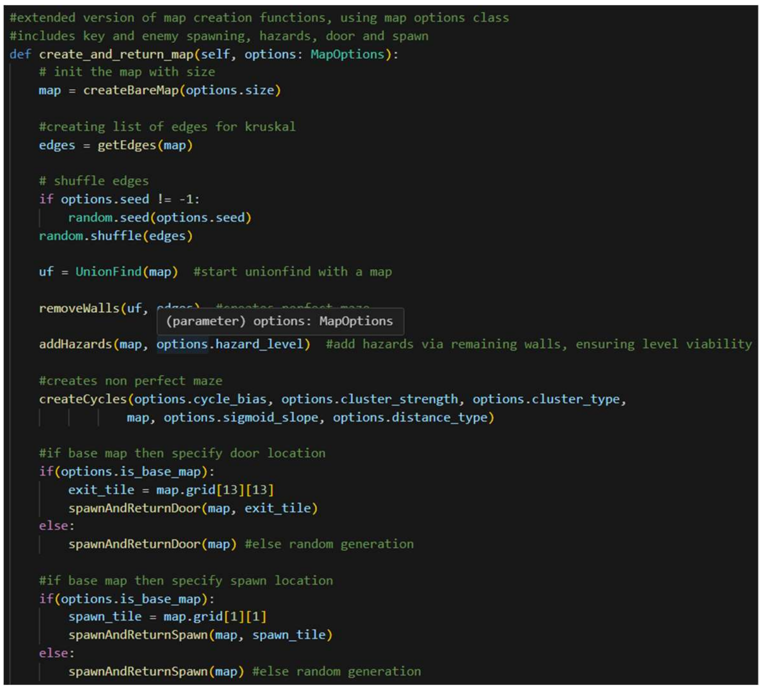
***Figure 2** – Post-Kruskal's, 'perfect' maze, seed=79*

The system also scales hazard placement, enemy spawning, and objective distribution (keys and exits). Map generation is fully customisable by seed, ensuring reproducibility. Training primarily took place on a single seed (seed 1), targeting what I believed to be a roughly medium difficulty environment.

Using the MapOptions class built for defining these parameters, one sets the values for creating a map that essentially pertain to how open or closed a map is, and how hazardous. This can be roughly analogised to difficulty.



```python
init_map_options = MapOptions(cycle_bias=0.8, cluster_strength=0.5, cluster_type="distance", is_base_map=False, seed=1, hazard_level=5),
```



```python
#extended version of map creation functions, using map options class
#includes key and enemy spawning, hazards, door and spawn
def create_and_return_map(self, options: MapOptions):
    # init the map with size
    map = createBareMap(options.size)

    #creating list of edges for kruskal
    edges = getEdges(map)

    # shuffle edges
    if options.seed != -1:
        random.seed(options.seed)
    random.shuffle(edges)

    uf = UnionFind(map)  #start unionfind with a map

    removeWalls(uf, edges)  #creates perfect maze
                (parameter) options: MapOptions
    addHazards(map, options.hazard_level)  #add hazards via remaining walls, ensuring level viability

    #creates non perfect maze
    createCycles(options.cycle_bias, options.cluster_strength, options.cluster_type,
                 map, options.sigmoid_slope, options.distance_type)

    #if base map then specify door location
    if(options.is_base_map):
        exit_tile = map.grid[13][13]
        spawnAndReturnDoor(map, exit_tile)
    else:
        spawnAndReturnDoor(map) #else random generation

    #if base map then specify spawn location
    if(options.is_base_map):
        spawn_tile = map.grid[1][1]
        spawnAndReturnSpawn(map, spawn_tile)
    else:
        spawnAndReturnSpawn(map) #else random generation
```

Figure 3 – MapOptions settings used for most of training / map creation snippet.

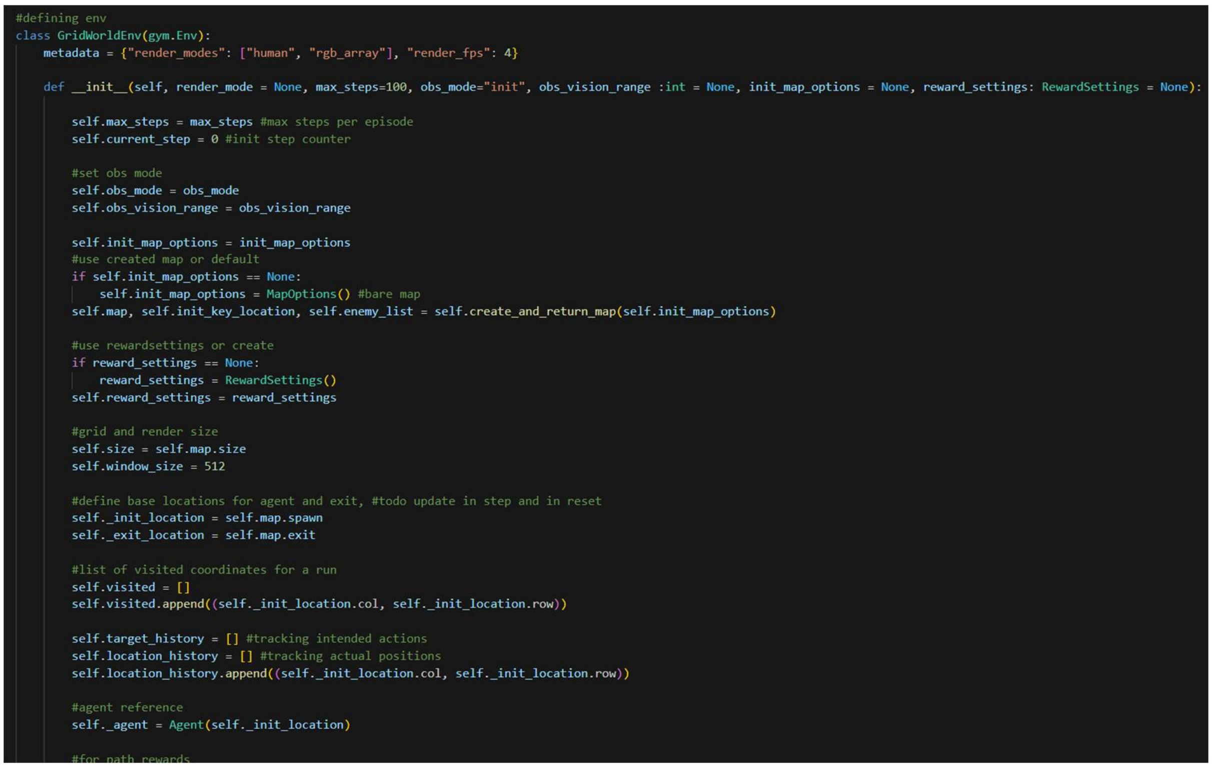
Environment

Creation

To begin reinforcement learning training, a custom environment was built using the Farama Foundation's Gymnasium library.

At its core, the environment defines an observation space, a step function to process agent actions and return rewards, and a reset function to ensure clean episodes without information leakage. These functions form the foundation of the training pipeline.
A basic rendering system using Pygame was also implemented to allow human monitoring of training, helping to identify behavioural issues.

The environment constructor accepts a range of arguments, allowing for different observation spaces, reward settings, and rendering options. It also stores references to the current map and relevant objects for use during training.



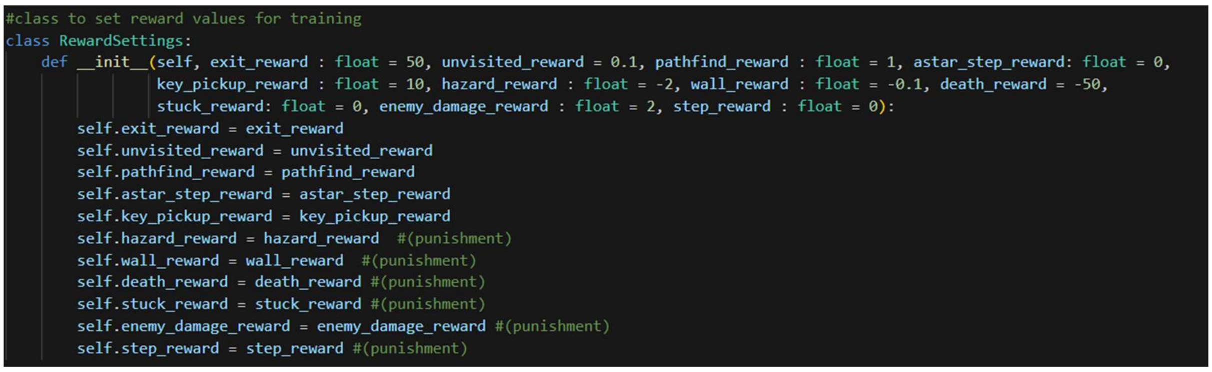
```
#defining env
class GridWorldEnv(gym.Env):
    metadata = {"render_modes": ["human", "rgb_array"], "render_fps": 4}

    def __init__(self, render_mode = None, max_steps=100, obs_mode="init", obs_vision_range :int = None, init_map_options = None, reward_settings: RewardSettings = None):

        self.max_steps = max_steps #max steps per episode
        self.current_step = 0 #init step counter

        #set obs mode
        self.obs_mode = obs_mode
        self.obs_vision_range = obs_vision_range

        self.init_map_options = init_map_options
        #use created map or default
        if self.init_map_options == None:
            self.init_map_options = MapOptions() #bare map
        self.map, self.init_key_location, self.enemy_list = self.create_and_return_map(self.init_map_options)

        #use rewardsettings or create
        if reward_settings == None:
            reward_settings = RewardSettings()
        self.reward_settings = reward_settings

        #grid and render size
        self.size = self.map.size
        self.window_size = 512

        #define base locations for agent and exit, #todo update in step and in reset
        self._init_location = self.map.spawn
        self._exit_location = self.map.exit

        #list of visited coordinates for a run
        self.visited = []
        self.visited.append((self._init_location.col, self._init_location.row))

        self.target_history = [] #tracking intended actions
        self.location_history = [] #tracking actual positions
        self.location_history.append((self._init_location.col, self._init_location.row))

        #agent reference
        self._agent = Agent(self._init_location)

        #for path rewards
```

Figure 4 – Example code from the environment constructor.

Observation Spaces and Reward Settings

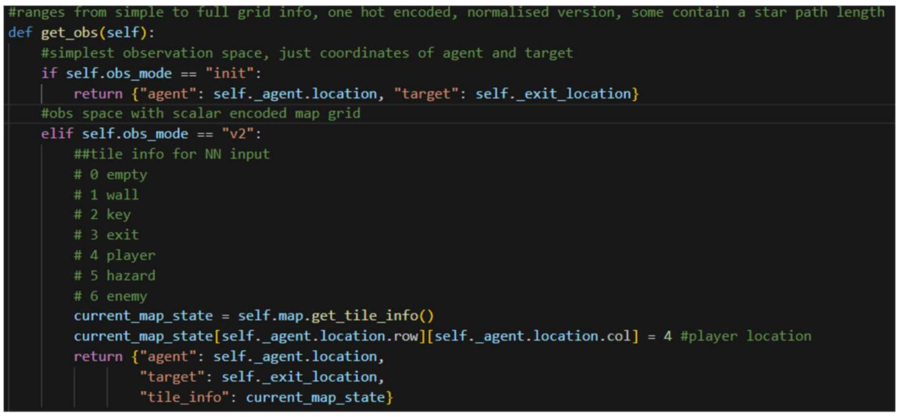
The observation spaces evolved over time, allowing many configurations: coordinates for key locations, one-hot or scalar encoded map states, agent health values, or A* path lengths. Different observation spaces led to varied model performance, providing key insights into feature selection for reinforcement learning.

A custom reward settings object defines rewards for events like reaching the exit, hitting hazards, moving closer to the goal, or getting stuck. This allowed experimenting with both sparse and dense reward setups, or testing the effects of harsher punishments versus more lenient settings.



```
#class to set reward values for training
class RewardSettings:
    def __init__(self, exit_reward : float = 50, unvisited_reward = 0.1, pathfind_reward : float = 1, astar_step_reward: float = 0,
                 key_pickup_reward : float = 10, hazard_reward : float = -2, wall_reward : float = -0.1, death_reward = -50,
                 stuck_reward: float = 0, enemy_damage_reward : float = 2, step_reward : float = 0):
        self.exit_reward = exit_reward
        self.unvisited_reward = unvisited_reward
        self.pathfind_reward = pathfind_reward
        self.astar_step_reward = astar_step_reward
        self.key_pickup_reward = key_pickup_reward
        self.hazard_reward = hazard_reward  #(punishment)
        self.wall_reward = wall_reward  #(punishment)
        self.death_reward = death_reward #(punishment)
        self.stuck_reward = stuck_reward #(punishment)
        self.enemy_damage_reward = enemy_damage_reward #(punishment)
        self.step_reward = step_reward #(punishment)
```

Figure 5 – Reward Settings class.



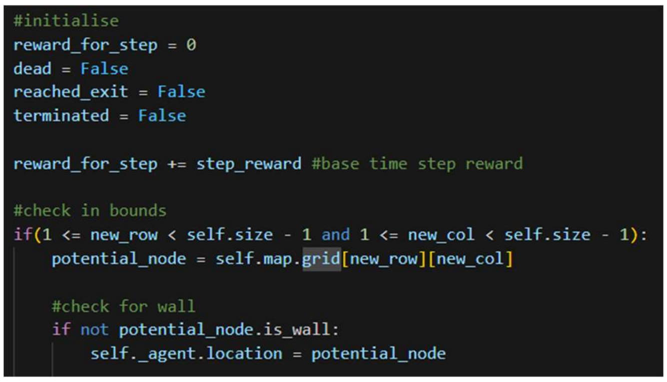
```python
#ranges from simple to full grid info, one hot encoded, normalised version, some contain a star path length
def get_obs(self):
    #simplest observation space, just coordinates of agent and target
    if self.obs_mode == "init":
        return {"agent": self._agent.location, "target": self._exit_location}
    #obs space with scalar encoded map grid
    elif self.obs_mode == "v2":
        ##tile info for NN input
        # 0 empty
        # 1 wall
        # 2 key
        # 3 exit
        # 4 player
        # 5 hazard
        # 6 enemy
        current_map_state = self.map.get_tile_info()
        current_map_state[self._agent.location.row][self._agent.location.col] = 4 #player location
        return {"agent": self._agent.location,
                "target": self._exit_location,
                "tile_info": current_map_state}
```

Figure 6 – Example early observation spaces.

Step / Reset

The step function processes an action, updates the environment, returns the new observation, reward, and episode termination flags. This is where the set custom reward settings object will play a part in how the step returns a reward signal.

The reset function fully resets the environment state to start a new training episode cleanly, avoiding contamination from prior episodes. This includes agent location, health, lists tracking visited locations or intentions, and even resets map-based variables in-case training is using procedurally generated content and needs to update to a new map.



```python
#initialise
reward_for_step = 0
dead = False
reached_exit = False
terminated = False

reward_for_step += step_reward #base time step reward

#check in bounds
if(1 <= new_row < self.size - 1 and 1 <= new_col < self.size - 1):
    potential_node = self.map.grid[new_row][new_col]

    #check for wall
    if not potential_node.is_wall:
        self._agent.location = potential_node
```

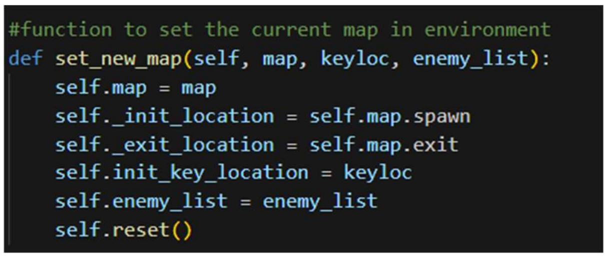
Figure 7 – Code snippet from step function.

PCG Capability

A MapOptions object defines parameters for map generation, including hazards, enemies, keys, and exits. On environment initialisation, a map is generated using these options. Maps can be swapped during training using a list of seeds, allowing models to train across varied environments within a single environment instance, allowing study of how procedural content generation can effect a model's ability to generalise.

This list of seeds is passed to the training loop during initialisation and training – which then call's the environment's helper functions. These functions allow for the generation and subsequent setting of maps for use during training within the environment.

This capability allows for curriculum learning capacity too – by specifying a different set of MapOptions, more open, 'easier' levels can be trained on initially. Changing the hyper parameters to create more dangerous and wall-filled maps allows for more difficult training regimes for models performing well on earlier sets.

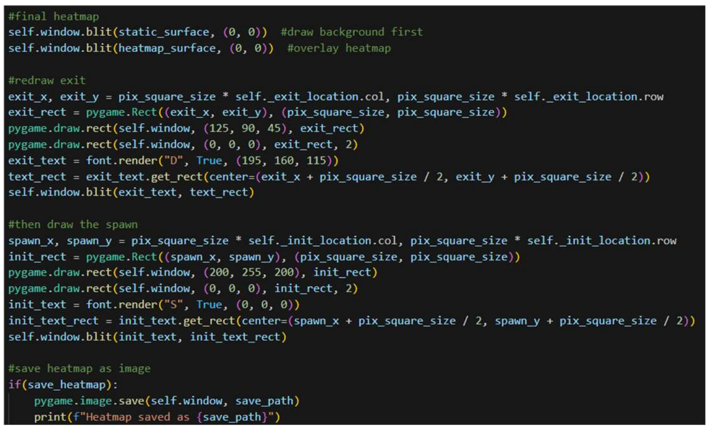


```python
#function to set the current map in environment
def set_new_map(self, map, keyloc, enemy_list):
    self.map = map
    self._init_location = self.map.spawn
    self._exit_location = self.map.exit
    self.init_key_location = keyloc
    self.enemy_list = enemy_list
    self.reset()
```

Figure 8 – Environment's map setting function.

Rendering

Rendering is not required for training but was implemented for better insight into why a model may be struggling during training, and what behavioural trends it might be exhibiting.
The environment can render not only live episodes but also replay full training histories, allowing for unrendered training for efficiency, and playback at a customisable FPS if desired. Additionally, a heatmap feature was added to summarise agent behaviour over full training runs quickly, highlighting movement patterns and problem areas. The heatmap allows for a rudimentary understanding of whether a model has learnt good or bad behaviours from a glance.



```python
#final heatmap
self.window.blit(static_surface, (0, 0))  #draw background first
self.window.blit(heatmap_surface, (0, 0))  #overlay heatmap

#redraw exit
exit_x, exit_y = pix_square_size * self._exit_location.col, pix_square_size * self._exit_location.row
exit_rect = pygame.Rect((exit_x, exit_y), (pix_square_size, pix_square_size))
pygame.draw.rect(self.window, (125, 90, 45), exit_rect)
pygame.draw.rect(self.window, (0, 0, 0), exit_rect, 2)
exit_text = font.render("D", True, (195, 160, 115))
text_rect = exit_text.get_rect(center=(exit_x + pix_square_size / 2, exit_y + pix_square_size / 2))
self.window.blit(exit_text, text_rect)

#then draw the spawn
spawn_x, spawn_y = pix_square_size * self._init_location.col, pix_square_size * self._init_location.row
init_rect = pygame.Rect((spawn_x, spawn_y), (pix_square_size, pix_square_size))
pygame.draw.rect(self.window, (200, 255, 200), init_rect)
pygame.draw.rect(self.window, (0, 0, 0), init_rect, 2)
init_text = font.render("S", True, (0, 0, 0))
init_text_rect = init_text.get_rect(center=(spawn_x + pix_square_size / 2, spawn_y + pix_square_size / 2))
self.window.blit(init_text, init_text_rect)

#save heatmap as image
if(save_heatmap):
    pygame.image.save(self.window, save_path)
    print(f"Heatmap saved as {save_path}")
```

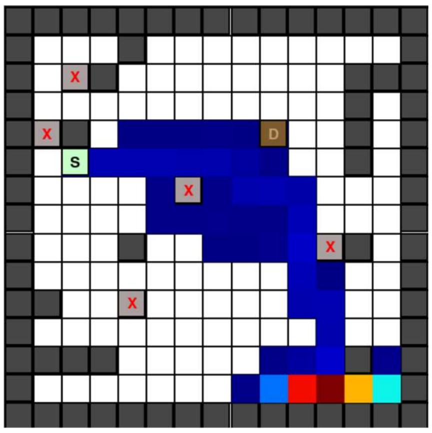


Figure 9 – Heatmap rendering snippet and example heatmap.

Testing Pipeline

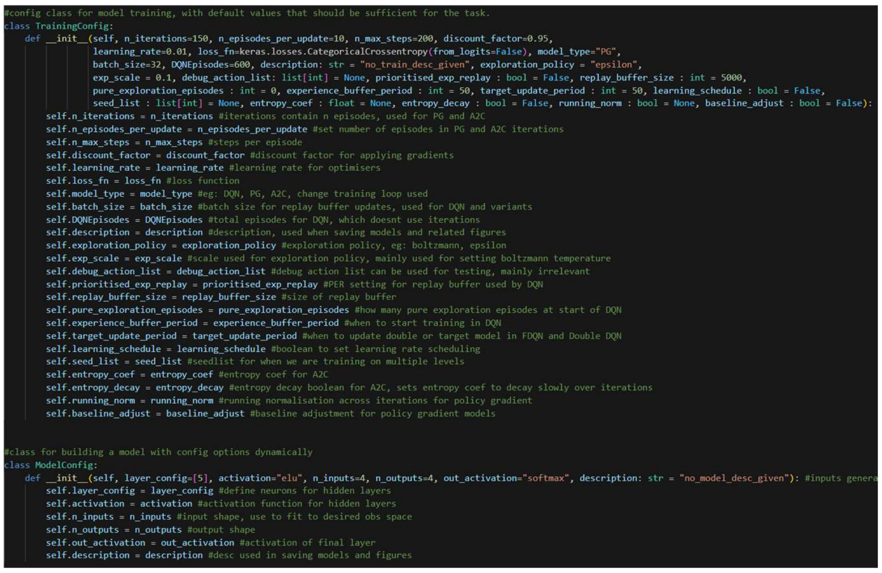
Overview

A custom testing and logging pipeline was developed to support efficient experimentation across environment configurations, PCG setups, model architectures, and training parameters. This system enables quick and reproducible testing and easy hyperparameter tuning.

The process begins by defining the environment with specific reward shaping and map options.

Once this is in place, the next step is to define a **TrainingConfig()** and **ModelConfig()** object. The model configuration class specifies architecture details—layer count, neurons, inputs and outputs and descriptive metadata.

The training configuration class defines how the training loop operates, covering iteration count, episodes per iteration, decay policies, learning rates, and model type, along with much more. The training step of the pipeline takes lists of these configurations, building and training models one by one based on the settings.

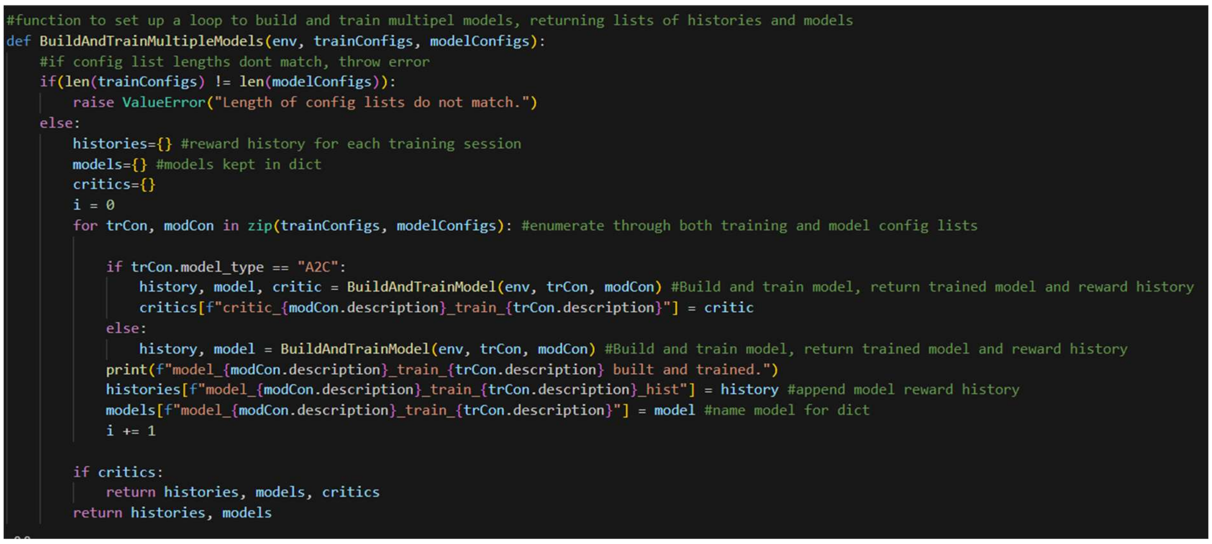


```python
#config class for model training, with default values that should be sufficient for the task.
class TrainingConfig:
    def __init__(self, n_iterations=150, n_episodes_per_update=10, n_max_steps=200, discount_factor=0.95,
                learning_rate=0.01, loss_fn=keras.losses.CategoricalCrossentropy(from_logits=False), model_type="PG",
                batch_size=32, DQNEpisodes=600, description: str = "no_train_desc_given", exploration_policy = "epsilon",
                exp_scale = 0.1, debug_action_list: list[int] = None, prioritised_exp_replay : bool = False, replay_buffer_size : int = 5000,
                pure_exploration_episodes : int = 0, experience_buffer_period : int = 50, target_update_period : int = 50, learning_schedule : bool = False,
                seed_list : list[int] = None, entropy_coef : float = None, entropy_decay : bool = False, running_norm : bool = None, baseline_adjust : bool = False):
        self.n_iterations = n_iterations #iterations contain n episodes, used for PG and A2C
        self.n_episodes_per_update = n_episodes_per_update #set number of episodes in PG and A2C iterations
        self.n_max_steps = n_max_steps #steps per episode
        self.discount_factor = discount_factor #discount factor for applying gradients
        self.learning_rate = learning_rate #learning rate for optimisers
        self.loss_fn = loss_fn #loss function
        self.model_type = model_type #eg: DQN, PG, A2C, change training loop used
        self.batch_size = batch_size #batch size for replay buffer updates, used for DQN and variants
        self.DQNEpisodes = DQNEpisodes #total episodes for DQN, which doesnt use iterations
        self.description = description #description, used when saving models and related figures
        self.exploration_policy = exploration_policy #exploration policy, eg: boltzmann, epsilon
        self.exp_scale = exp_scale #scale used for exploration policy, mainly used for setting boltzmann temperature
        self.debug_action_list = debug_action_list #debug action list can be used for testing, mainly irrelevant
        self.prioritised_exp_replay = prioritised_exp_replay #PER setting for replay buffer used by DQN
        self.replay_buffer_size = replay_buffer_size #size of replay buffer
        self.pure_exploration_episodes = pure_exploration_episodes #how many pure exploration episodes at start of DQN
        self.experience_buffer_period = experience_buffer_period #when to start training in DQN
        self.target_update_period = target_update_period #when to update double or target model in FDQN and Double DQN
        self.learning_schedule = learning_schedule #boolean to set learning rate scheduling
        self.seed_list = seed_list #seedlist for when we are training on multiple levels
        self.entropy_coef = entropy_coef #entropy coef for A2C
        self.entropy_decay = entropy_decay #entropy decay boolean for A2C, sets entropy coef to decay slowly over iterations
        self.running_norm = running_norm #running normalisation across iterations for policy gradient
        self.baseline_adjust = baseline_adjust #baseline adjustment for policy gradient models


#class for building a model with config options dynamically
class ModelConfig:
    def __init__(self, layer_config=[5], activation="elu", n_inputs=4, n_outputs=4, out_activation="softmax", description: str = "no_model_desc_given"): #inputs genera
        self.layer_config = layer_config #define neurons for hidden layers
        self.activation = activation #activation function for hidden layers
        self.n_inputs = n_inputs #input shape, use to fit to desired obs space
        self.n_outputs = n_outputs #output shape
        self.out_activation = out_activation #activation of final layer
        self.description = description #desc used in saving models and figures
```

Figure 10 – Model and Training configuration classes.

Training & History

A ***build_and_train_multiple_models()*** function handles the full pipeline execution. It takes model and training configurations along with an environment reference, builds each model accordingly, and runs training based on the provided settings. This allows differentiation between Policy Gradient, DQN variants and A2C training loops.



```python
#function to set up a loop to build and train multipel models, returning lists of histories and models
def BuildAndTrainMultipleModels(env, trainConfigs, modelConfigs):
    #if config list lengths dont match, throw error
    if(len(trainConfigs) != len(modelConfigs)):
        raise ValueError("Length of config lists do not match.")
    else:
        histories={} #reward history for each training session
        models={} #models kept in dict
        critics={}
        i = 0
        for trCon, modCon in zip(trainConfigs, modelConfigs): #enumerate through both training and model config lists

            if trCon.model_type == "A2C":
                history, model, critic = BuildAndTrainModel(env, trCon, modCon) #Build and train model, return trained model and reward history
                critics[f"critic_{modCon.description}_train_{trCon.description}"] = critic
            else:
                history, model = BuildAndTrainModel(env, trCon, modCon) #Build and train model, return trained model and reward history
            print(f"model_{modCon.description}_train_{trCon.description} built and trained.")
            histories[f"model_{modCon.description}_train_{trCon.description}_hist"] = history #append model reward history
            models[f"model_{modCon.description}_train_{trCon.description}"] = model #name model for dict
            i += 1

        if critics:
            return histories, models, critics
        return histories, models
```

Figure 11 – Function for training and building multiple models.

During training, a history object is built that logs not only reward progression but also agent positions per episode—enabling later use of rendering and heatmap features in the environment. There are two classes here, an **EpHistory()** class which stores specific history for a single episode, and a **TrainingHistory()** class which stores all histories over the course of training, featuring a list of lists of EpHistories.

Once training concludes, the **plot_reward_history()** function can be used to graph training rewards across iterations or average rewards per episode. This became a primary tool for evaluating the effect of changes to model structure, training parameters, reward shaping, and observation space.

There are also several post training analysis functions, for building a pandas dataframe, and subsequently creating scatterplots, bar charts, and correlation heatmaps for examining results of trained models upon unseen seeds.

Models

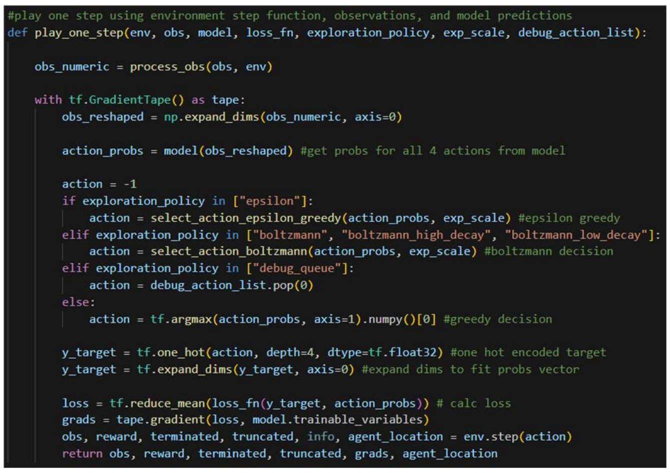
Each of the types of model's training processes follow roughly the same logic.

The process starts with setting an optimiser and initialising global variables necessary such as for storing history.
If training on multiple levels, the correct map is generated and set in the environment.

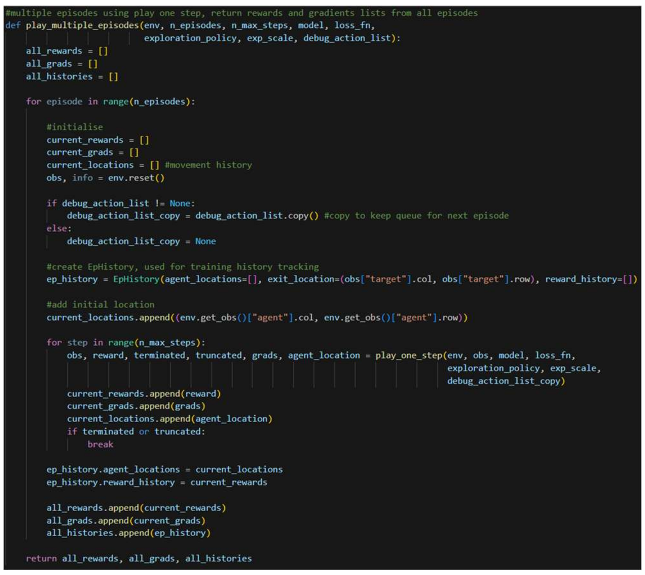
The training will loop through the set number of iterations as per the training configuration, scaling the exploration policy if applicable based on how far into the loop it is and relevant settings. For Policy Gradient and Actor Critic methods, each iteration comes with 10 episodes. Following a **play_multiple_episodes()** function, the environment will reset for each episode, and the observation space will be processed into numerical inputs. From there each step plays out with a **play_one_step()** function, which in turn calls the environment's **step()** function with the chosen action. This continues until an episode is terminated and truncated. Loss is calculated, rewards and gradients are gathered each step, and at the end of the episode an **EpHistory** object containing location and reward history is stored in a list to be returned at the end of the iteration.

With the end of the iteration, the relevant history is stored and gradients are applied by the optimiser. The cycle continues with further iterations.
DQN training structure is slightly different, relying on singular episodes instead of batched in iterations. Training may also include multiple models in both DQN and A2C methods, and could rely on a replay buffer or staggered updating of weights for the secondary model.

In the case of A2C, state values from the critic are also returned with each iteration, for calculation of loss and gradients for the actor.



```python
#play one step using environment step function, observations, and model predictions
def play_one_step(env, obs, model, loss_fn, exploration_policy, exp_scale, debug_action_list):

    obs_numeric = process_obs(obs, env)

    with tf.GradientTape() as tape:
        obs_reshaped = np.expand_dims(obs_numeric, axis=0)

        action_probs = model(obs_reshaped) #get probs for all 4 actions from model

        action = -1
        if exploration_policy in ["epsilon"]:
            action = select_action_epsilon_greedy(action_probs, exp_scale) #epsilon greedy
        elif exploration_policy in ["boltzmann", "boltzmann_high_decay", "boltzmann_low_decay"]:
            action = select_action_boltzmann(action_probs, exp_scale) #boltzmann decision
        elif exploration_policy in ["debug_queue"]:
            action = debug_action_list.pop(0)
        else:
            action = tf.argmax(action_probs, axis=1).numpy()[0] #greedy decision

        y_target = tf.one_hot(action, depth=4, dtype=tf.float32) #one hot encoded target
        y_target = tf.expand_dims(y_target, axis=0) #expand dims to fit probs vector

        loss = tf.reduce_mean(loss_fn(y_target, action_probs)) # calc loss
        grads = tape.gradient(loss, model.trainable_variables)
        obs, reward, terminated, truncated, info, agent_location = env.step(action)
        return obs, reward, terminated, truncated, grads, agent_location
```



```python
#multiple episodes using play one step, return rewards and gradients lists from all episodes
def play_multiple_episodes(env, n_episodes, n_max_steps, model, loss_fn,
                           exploration_policy, exp_scale, debug_action_list):
    all_rewards = []
    all_grads = []
    all_histories = []

    for episode in range(n_episodes):

        #initialise
        current_rewards = []
        current_grads = []
        current_locations = [] #movement history
        obs, info = env.reset()

        if debug_action_list != None:
            debug_action_list_copy = debug_action_list.copy() #copy to keep queue for next episode
        else:
            debug_action_list_copy = None

        #create EpHistory, used for training history tracking
        ep_history = EpHistory(agent_locations=[], exit_location=(obs["target"].col, obs["target"].row), reward_history=[])

        #add initial location
        current_locations.append((env.get_obs()["agent"].col, env.get_obs()["agent"].row))

        for step in range(n_max_steps):
            obs, reward, terminated, truncated, grads, agent_location = play_one_step(env, obs, model, loss_fn,
                                                                                      exploration_policy, exp_scale,
                                                                                      debug_action_list_copy)
            current_rewards.append(reward)
            current_grads.append(grads)
            current_locations.append(agent_location)
            if terminated or truncated:
                break

        ep_history.agent_locations = current_locations
        ep_history.reward_history = current_rewards

        all_rewards.append(current_rewards)
        all_grads.append(current_grads)
        all_histories.append(ep_history)

    return all_rewards, all_grads, all_histories
```

Figure 12 & 13 – Policy Gradient – play_one_step() / play_multiple_episodes()

When all is said and done, a **TrainingHistory** object contains all the history stored from each iteration and their episodes, ready for plotting and analysis.

Results

Training - Policy Gradient

Testing began with a simple Policy Gradient model set up, using relatively small model architectures and relatively standard hyperparameters, aiming to sweep through ranges and identify values that worked and areas of failing.
Very early it was evident that even though this task was more complex than I was used to in cartpole, models were still prone to overfitting and overreacting to rewards.

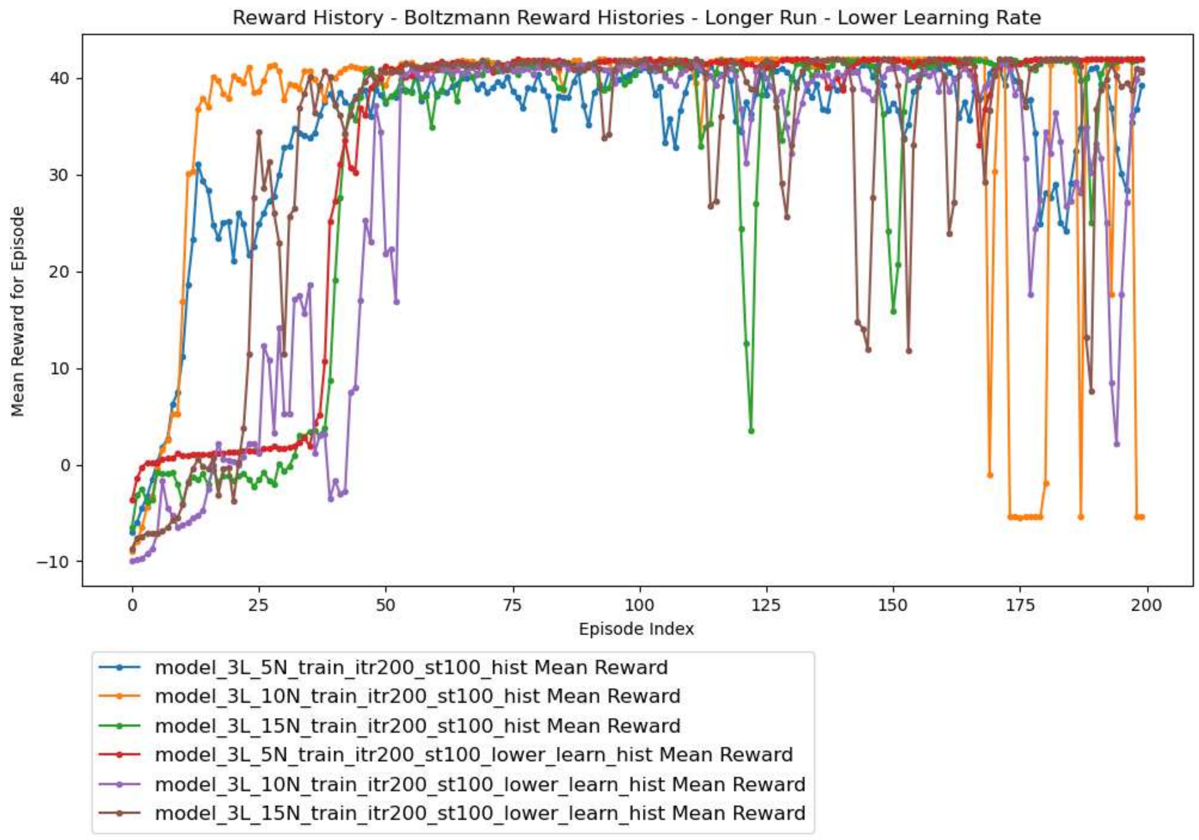


Figure 14 – Early policy gradient testing, Boltzmann exploration and learning rate tests.

This early chart shows 6 models of varying architecture and 3 feature a lower learning rate of 0.005 rather than 0.01.
The best performer was the simplest model, with the lowest learning rate. Important to note is the way models can catastrophically forget, as taming this erratic spiking is important for training a stable model able to generalise.

As tests progressed, the environment was made to be more complex with 'v2' and 'v3' observation spaces which included the entire map grid encoded as scalars, as well as agent health. Larger models were tested to match the new input size that came with this.

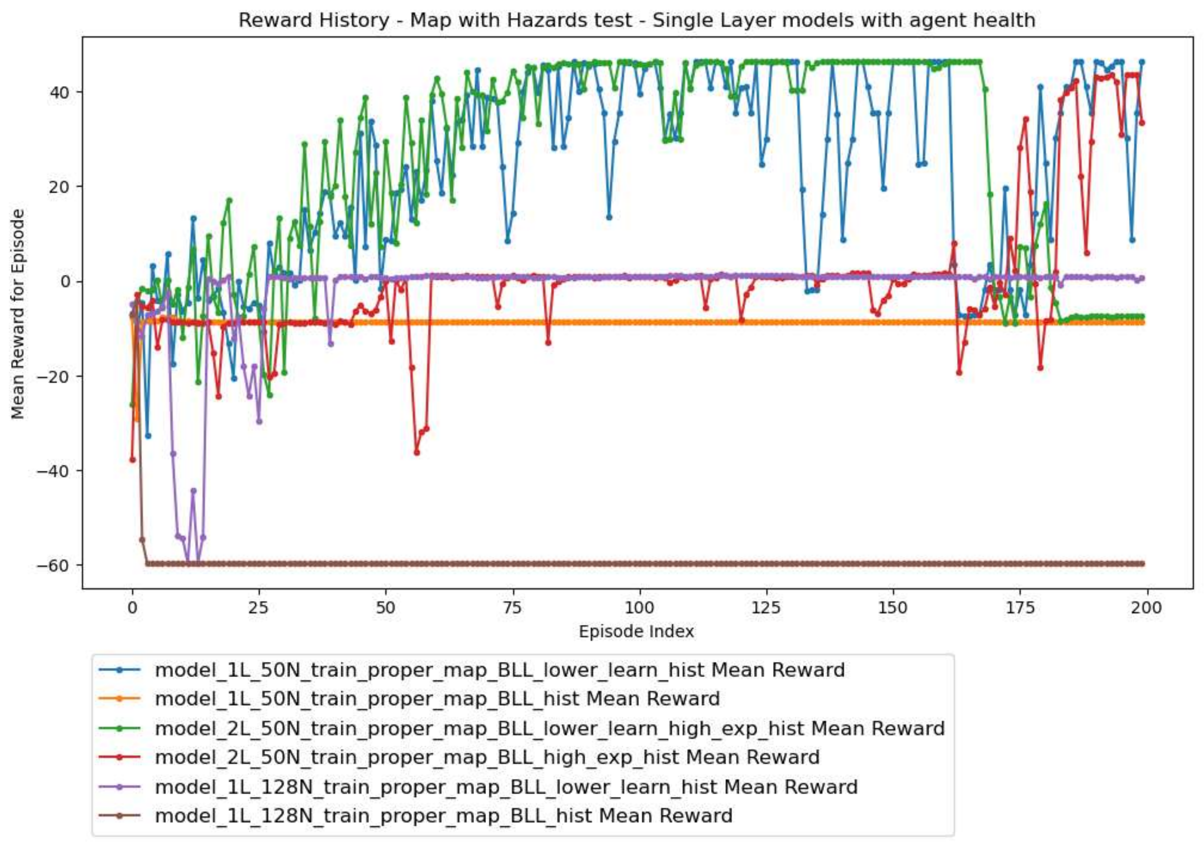


Figure 15 – Full map with hazards, early model testing.

In this example test, again noting that some-what smaller architecture and lower learning rate wins out in stability. This test also highlights that sometimes models were collapsing early into bad behaviours and unable to recover.

With development of the heatmap rendering function, we can examine this visually, with examples of models consistently hugging walls and corners, perhaps as a 'safe' option, where explorative policy isn't forcing enough exploration to break out of it. Figure [16] shows a model with 256 Neurons exhibiting a lack of exploration and an extreme overconfidence in hugging the left wall, as opposed to Figure [17], a single layer model with just 64 neurons, exhibiting far more explorative behaviour and ability to path find.

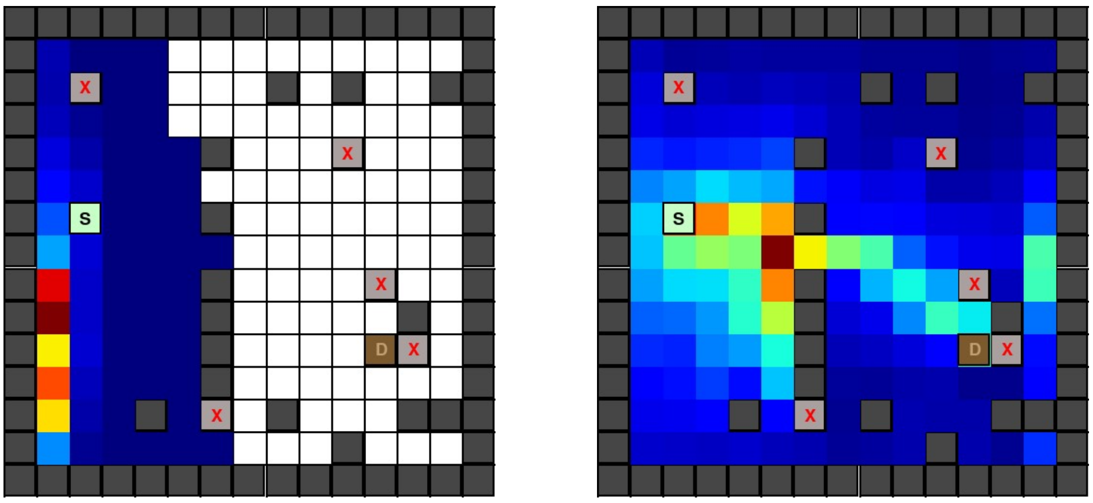


Figure 16 & 17 – 1L 256N and 1L 64N policy gradient heatmaps on seed 1.

Training - DQN

Three variants of deep Q Learning were attempted. Those being a regular DQN, a Fixed-Target DQN and a Double DQN. Initial testing showed some level of success early in training, but with notable instability and catastrophic forgetting.

Below features the first attempt at a Fixed-Target DQN, reaching high rewards after an initial target update, and getting worse with each further update.

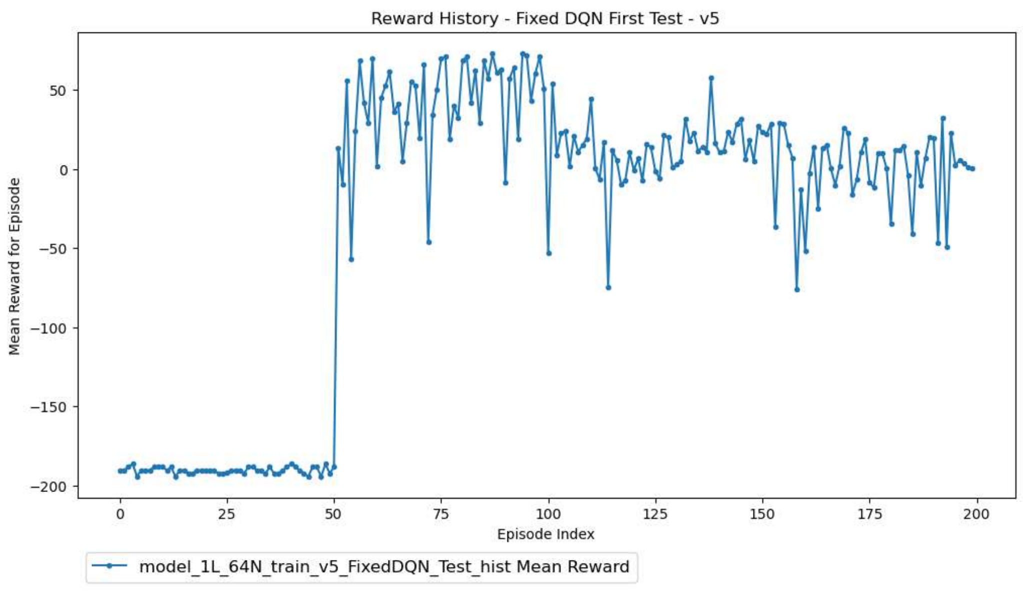


Figure 18 – Initial attempts at Fixed Target DQN.

Unfortunately I was not able to bring deep queue networks to a point of stability and convergence.

I attempted to implement many changes to the DQN training loop, in an effort to achieve some semblance of consistent learning and stability, such as prioritised experience replay, using one hot encoded observation spaces, learning rate scheduling and more – but the best I could achieve was a level of neutral survival, with intermittent spikes of high score followed generally by total catastrophic forgetting.

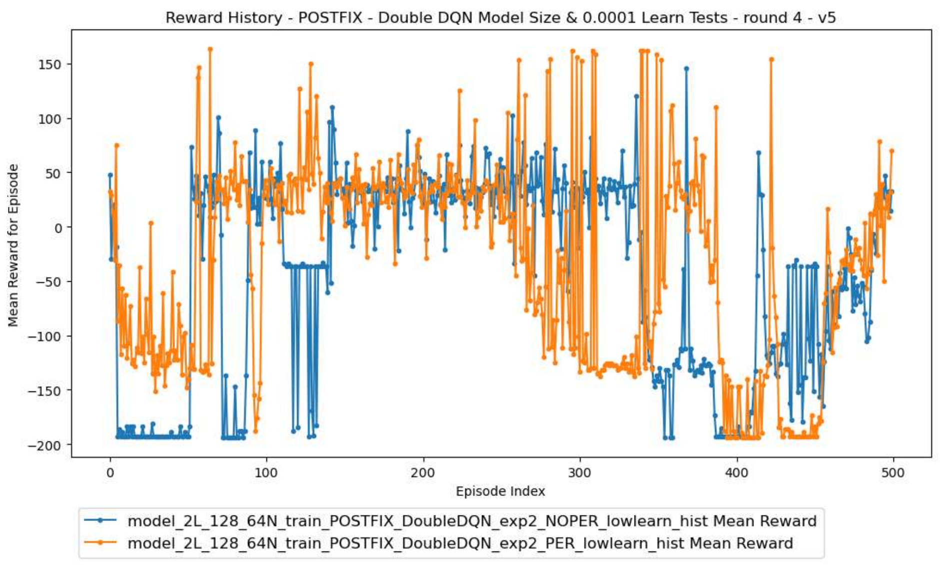


Figure 19 – Catastrophic forgetting behaviour in Double DQN.

The models are clearly capable of learning – but very susceptible to what seems like over correction during training. Given more time I would like to explore this more as I believe that it's possible to make work.

Training - A2C

Initial results for Actor Critic models showed promise with ability to learn, albeit sometimes erratic. Performance could either converge extremely well or catastrophically collapse as seen in the figure below.

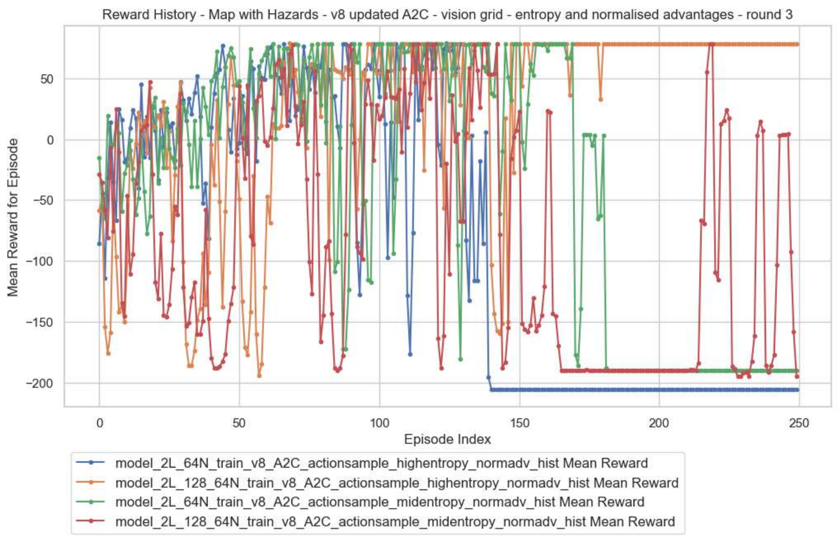


Figure 20 – Early A2C, instability tests.

Results fared better with an added high entropy co-efficient, incentivising more entropic probability distributions.

Stability could be found by also simplifying the model architecture, which seemed to calm the erratic spikes in behaviour somewhat.

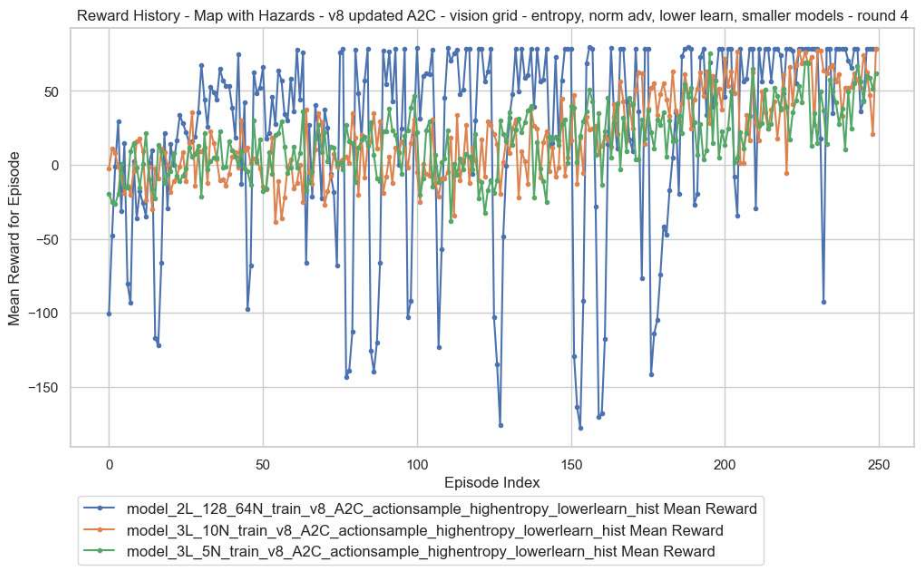


Figure 21 – Early A2C, instability tests.

Training - Multiple Levels

Training with multiple levels is a naturally unstable setting for training. Where the policy gradient models may converge on a navigation task, now each level can throw the model off, and the results oscillate heavily.

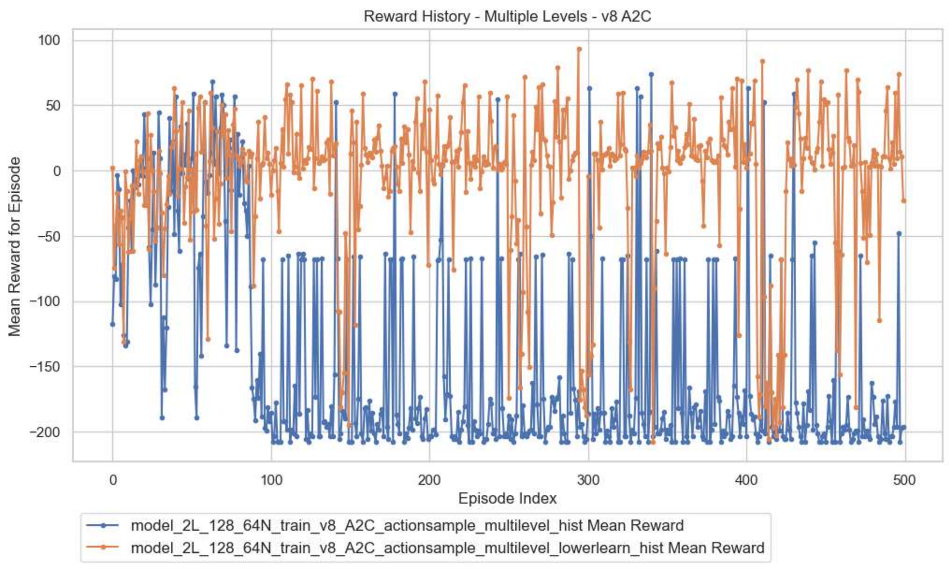


Figure 22 – A2C Multiple Level Training (PCG) – Policy Collapse.

Early tests with A2C models across 500 levels showed that a higher learning rate could lead to pretty catastrophic collapse in policy.

Implementation of the entropy coefficient to encourage more entropic action probabilities, along with keeping learning rate lower led to more stable outcomes. Early training sees an upward trend, but a lull in upwards trajectory in later training.

Also notably smaller models seemed to lose themselves less in erratic spikes.

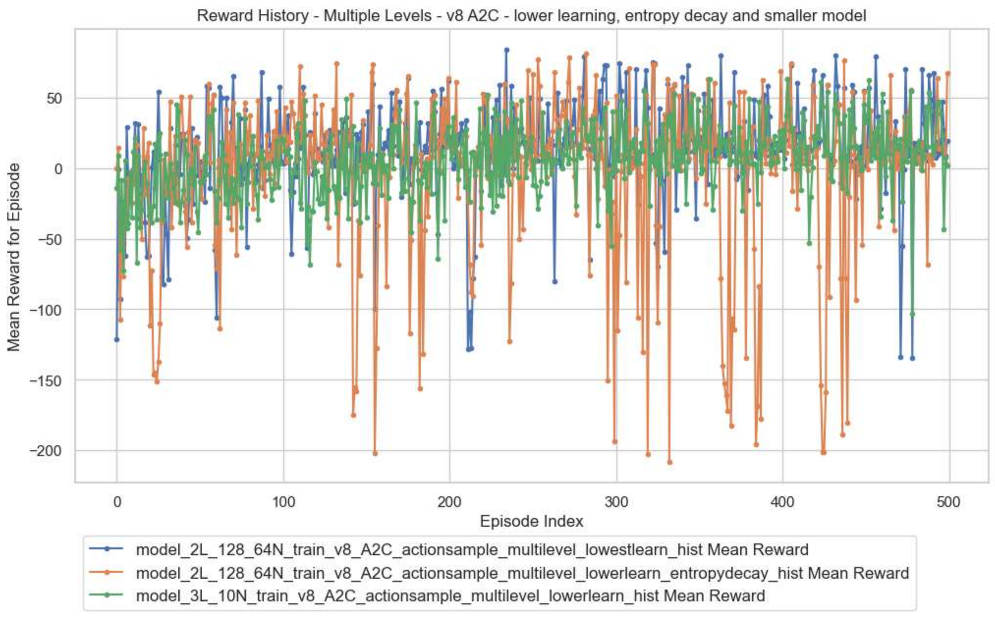


Figure 23 – A2C Multiple Level Training (PCG) – Stability changes.

Policy gradient models saw instability with initial attempts across 500 levels. The figure below features two models using Boltzmann exploration at a temperature of 2, which decays quickly to greedy policy over this many episodes.

By decreasing the rate at which this Boltzmann temperature decays at, and decreasing learning rate – the models showed increased stability and a slow upwards trend in rewards.

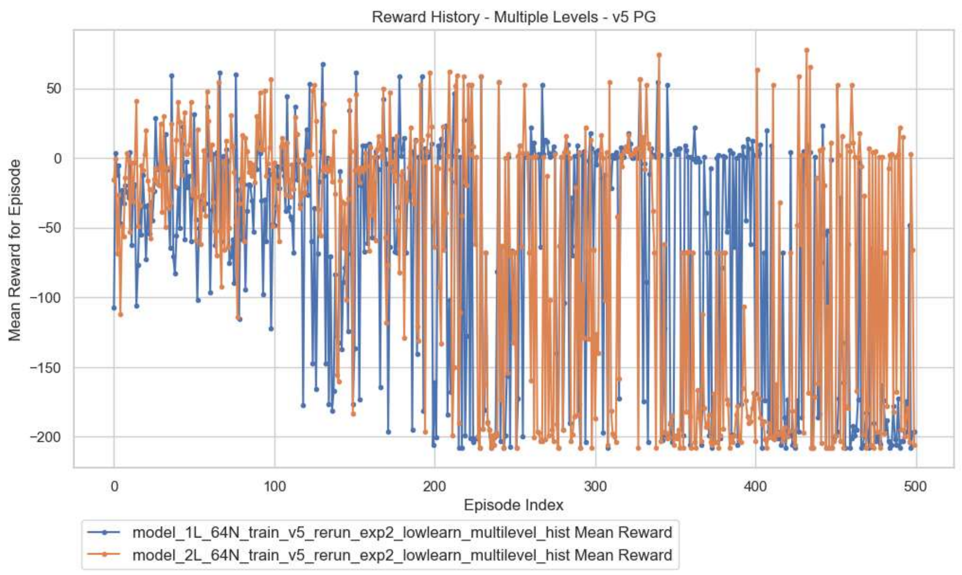


Figure 24 – Policy Gradient Multiple Level Training (PCG) – Initial instability.

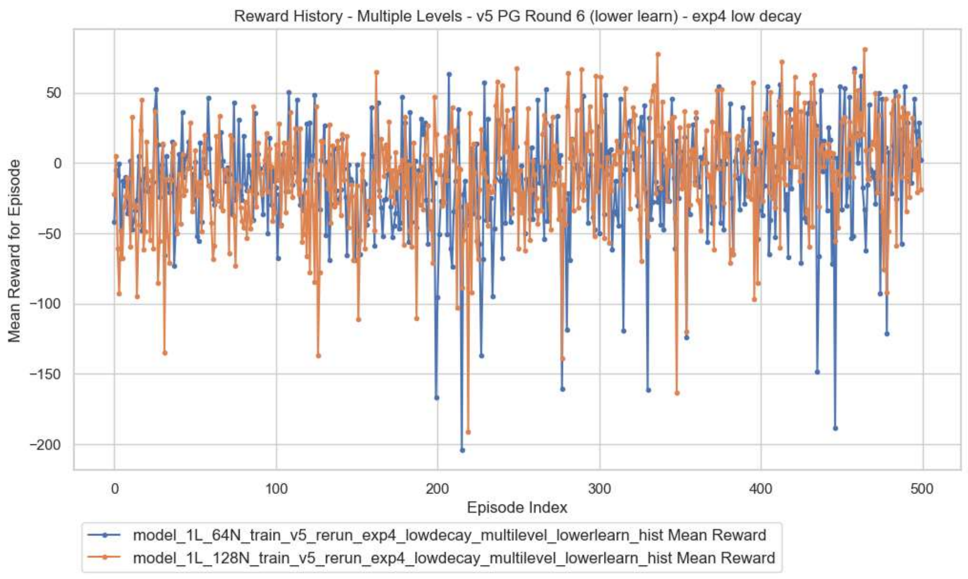


Figure 25 – PG PCG training – stability through lowered exploration decay.

Furthermore, adjusting rewards by a baseline and normalising rewards using running variables between iterations was found to in some cases slow the learning process but do seem to limit erraticism and lead to slow but positive learning trends, seen below in Figure 26.

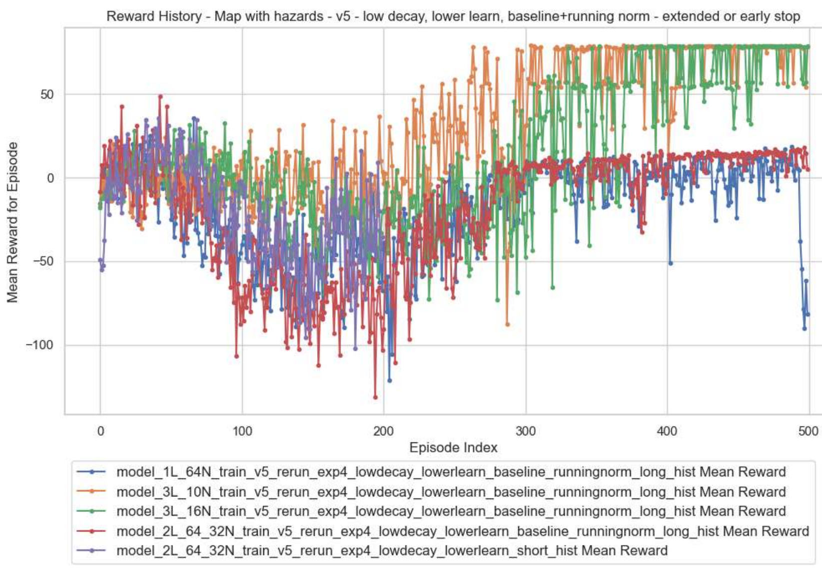


Figure 26 – Baseline adjustment and Running normalisation in Policy Gradient.

Training - Curriculum Learning

Some models were trained not just on multiple levels, but instead multiple levels separated by tiers of difficulty. These models performed just as well as their non-curriculum counterparts – but could use further testing. There was no system in place to 'checkpoint' models, ensuring the next stage of training happens with a model that's as good as it can be from the prior stage of training.

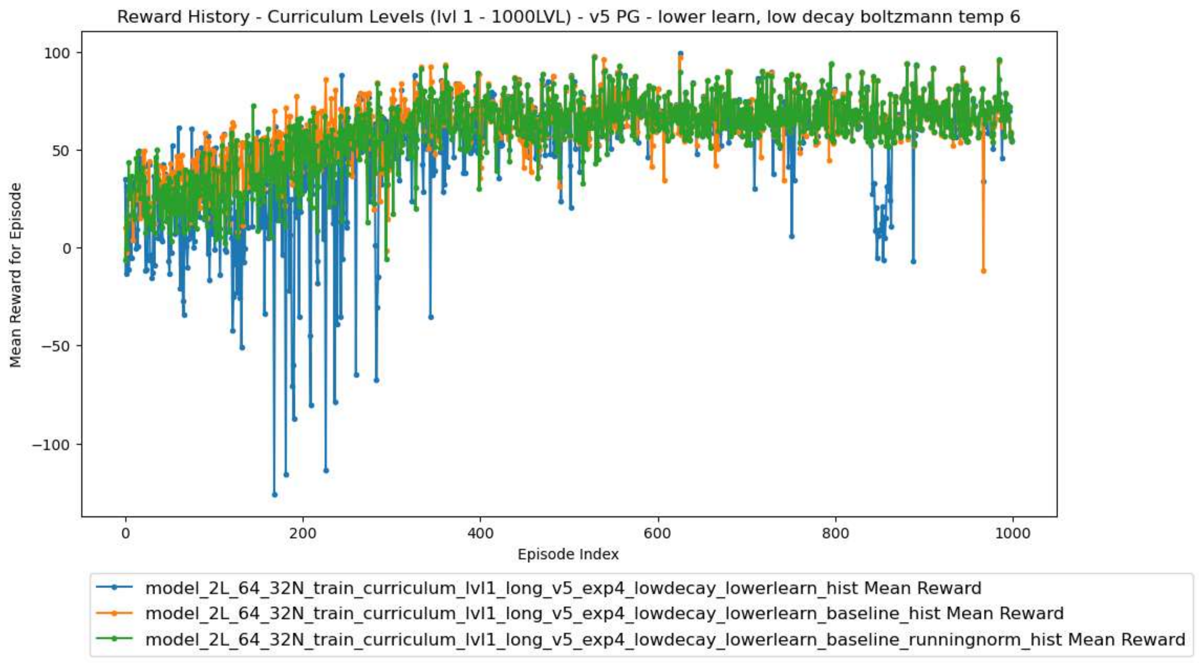


Figure 27 – Policy Gradient curriculum training (easy levels).

Post Training – Generalisation Capability

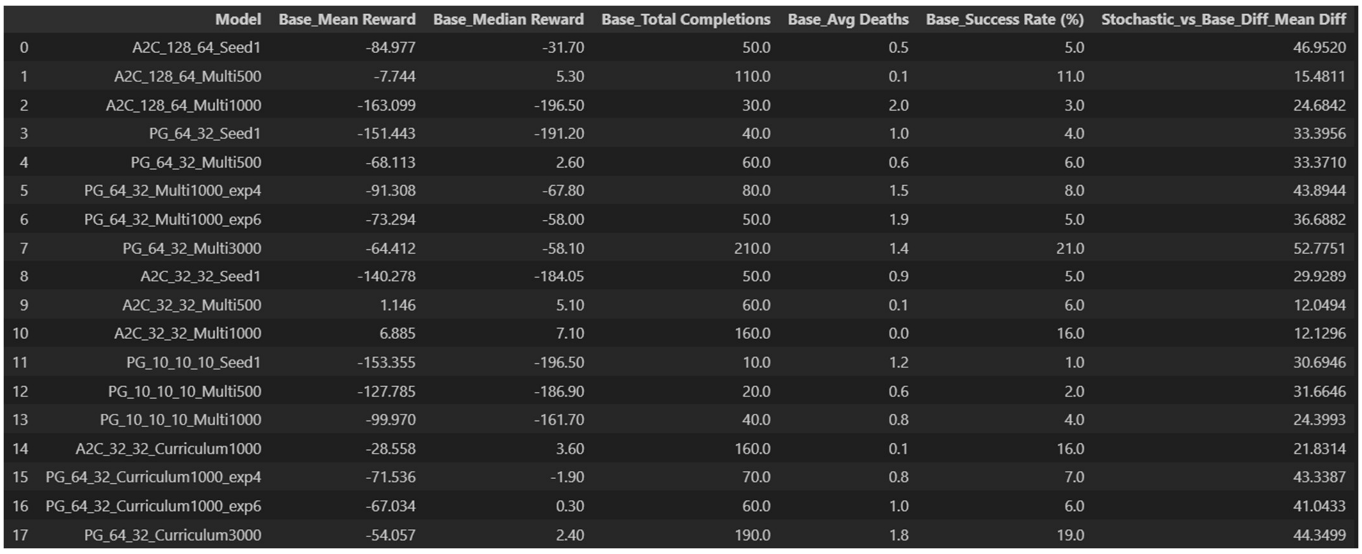
Analysis of post training performance took place upon 100 seeds (101-200) unseen to any model during any of the training. These were all generated using the same 'medium' difficulty settings used for most of the training.

Models were tested with versions trained only one seed 1, trained with 500 levels (501-1000), or 1000 levels (501-1501).

Seeds were attempted twice, once using greedy model predictions, and another using a small level of stochasticity, with a 10% chance to choose a random action. Stochasticity proved to be a very helpful addition in post training situations. No model was good enough for its purely greedy actions to govern it entirely. Random actions improved completions, reduced deaths and improved score across the board.

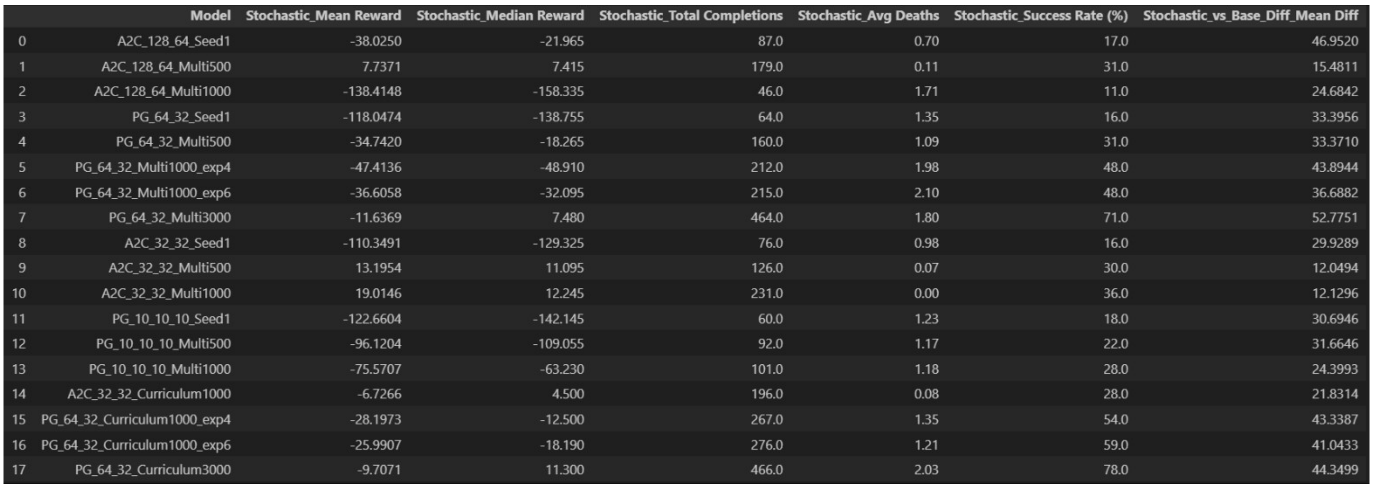
The goal of this analysis is to test how well each model can generalise to their new environment.

Results showed overwhelmingly that models trained on a single seed, failed catastrophically to generalise to new levels. A2C models performed slightly better than the best of the Policy Gradient models trained.



	Model	Base_Mean Reward	Base_Median Reward	Base_Total Completions	Base_Avg Deaths	Base_Success Rate (%)	Stochastic_vs_Base_Diff_Mean Diff
0	A2C_128_64_Seed1	-84.977	-31.70	50.0	0.5	5.0	46.9520
1	A2C_128_64_Multi500	-7.744	5.30	110.0	0.1	11.0	15.4811
2	A2C_128_64_Multi1000	-163.099	-196.50	30.0	2.0	3.0	24.6842
3	PG_64_32_Seed1	-151.443	-191.20	40.0	1.0	4.0	33.3956
4	PG_64_32_Multi500	-68.113	2.60	60.0	0.6	6.0	33.3710
5	PG_64_32_Multi1000_exp4	-91.308	-67.80	80.0	1.5	8.0	43.8944
6	PG_64_32_Multi1000_exp6	-73.294	-58.00	50.0	1.9	5.0	36.6882
7	PG_64_32_Multi3000	-64.412	-58.10	210.0	1.4	21.0	52.7751
8	A2C_32_32_Seed1	-140.278	-184.05	50.0	0.9	5.0	29.9289
9	A2C_32_32_Multi500	1.146	5.10	60.0	0.1	6.0	12.0494
10	A2C_32_32_Multi1000	6.885	7.10	160.0	0.0	16.0	12.1296
11	PG_10_10_10_Seed1	-153.355	-196.50	10.0	1.2	1.0	30.6946
12	PG_10_10_10_Multi500	-127.785	-186.90	20.0	0.6	2.0	31.6646
13	PG_10_10_10_Multi1000	-99.970	-161.70	40.0	0.8	4.0	24.3993
14	A2C_32_32_Curriculum1000	-28.558	3.60	160.0	0.1	16.0	21.8314
15	PG_64_32_Curriculum1000_exp4	-71.536	-1.90	70.0	0.8	7.0	43.3387
16	PG_64_32_Curriculum1000_exp6	-67.034	0.30	60.0	1.0	6.0	41.0433
17	PG_64_32_Curriculum3000	-54.057	2.40	190.0	1.8	19.0	44.3499

Figure 28 – Base Post training summary (greedy policy)



	Model	Stochastic_Mean Reward	Stochastic_Median Reward	Stochastic_Total Completions	Stochastic_Avg Deaths	Stochastic_Success Rate (%)	Stochastic_vs_Base_Diff_Mean Diff
0	A2C_128_64_Seed1	-38.0250	-21.965	87.0	0.70	17.0	46.9520
1	A2C_128_64_Multi500	7.7371	7.415	179.0	0.11	31.0	15.4811
2	A2C_128_64_Multi1000	-138.4148	-158.335	46.0	1.71	11.0	24.6842
3	PG_64_32_Seed1	-118.0474	-138.755	64.0	1.35	16.0	33.3956
4	PG_64_32_Multi500	-34.7420	-18.265	160.0	1.09	31.0	33.3710
5	PG_64_32_Multi1000_exp4	-47.4136	-48.910	212.0	1.98	48.0	43.8944
6	PG_64_32_Multi1000_exp6	-36.6058	-32.095	215.0	2.10	48.0	36.6882
7	PG_64_32_Multi3000	-11.6369	7.480	464.0	1.80	71.0	52.7751
8	A2C_32_32_Seed1	-110.3491	-129.325	76.0	0.98	16.0	29.9289
9	A2C_32_32_Multi500	13.1954	11.095	126.0	0.07	30.0	12.0494
10	A2C_32_32_Multi1000	19.0146	12.245	231.0	0.00	36.0	12.1296
11	PG_10_10_10_Seed1	-122.6604	-142.145	60.0	1.23	18.0	30.6946
12	PG_10_10_10_Multi500	-96.1204	-109.055	92.0	1.17	22.0	31.6646
13	PG_10_10_10_Multi1000	-75.5707	-63.230	101.0	1.18	28.0	24.3993
14	A2C_32_32_Curriculum1000	-6.7266	4.500	196.0	0.08	28.0	21.8314
15	PG_64_32_Curriculum1000_exp4	-28.1973	-12.500	267.0	1.35	54.0	43.3387
16	PG_64_32_Curriculum1000_exp6	-25.9907	-18.190	276.0	1.21	59.0	41.0433
17	PG_64_32_Curriculum3000	-9.7071	11.300	466.0	2.03	78.0	44.3499

Figure 29 – Stochastic Post training summary (greedy policy + 10% random)

Every model, when trained on 500 seeds, proved to be much more effective at generalising to these unseen seeds.
Overall, the highest average scoring models were A2C models. When stochasticity is added, Policy Gradient models catch up a little bit in terms of performance. Notably A2C models die a

lot less, seeming to have learnt to avoid hazards and walls better than Policy Gradient counterparts.

Following the trend of stochasticity being helpful, Policy Gradient models trained on multiple levels with a Boltzmann temperature of 6 performed better than those with 4, insinuating the extra exploration helps to round out behavioural training.

Generalisation capability tended to improve when increasing the levels trained on to 3000, and the best performing models were both models trained on 3000 levels.

Curriculum learning also noticeably improved Policy Gradient models, but the A2C model did not see an improvement. This is possibly due to the fact that in this quick implementation of Curriculum Learning, there was no waiting for a model to show convergence on a tier of difficulty, and in this case the curriculum trained 1000 level A2C model did see an erratic drop in performance at the end of its level 2 training, which may have affected its performance for future learning in level 3.

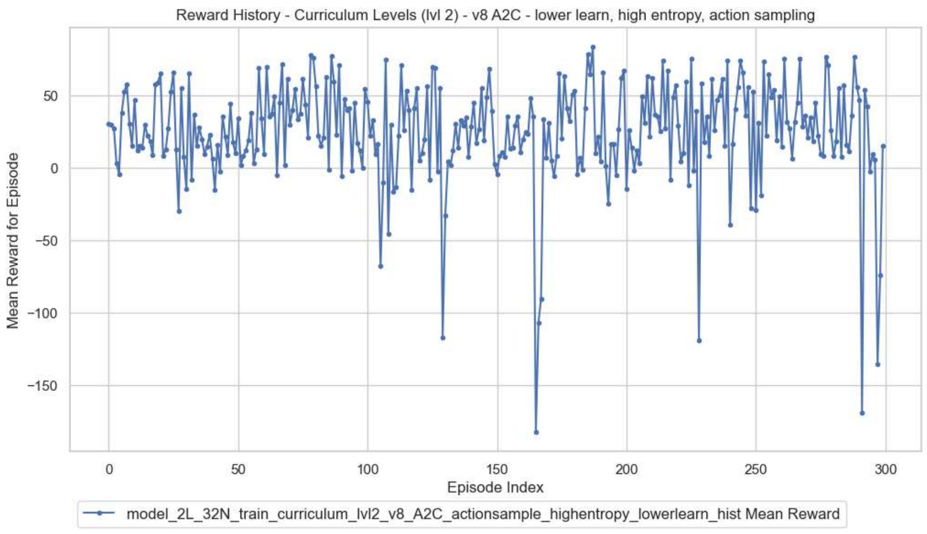


Figure 30 – A2C Curriculum Training – drop at the end (LVL2)

The highest performance overall in terms of completion rate was from the Curriculum Learning, 3000 Seed trained Policy Gradient model, with stochastic injection in post train action choice.

This model achieved 466 / 1000 possible completions, with 78% of levels seeing at least one completion. It failed to reach the highest average reward, totalling at -9.7 – the highest being the smaller of the two A2C models, trained on 1000 seeds with a mean reward of 19. Comparitively, the A2C model only had a success rate of 36% of levels with 231 / 1000 completions possible.

This suggests that the Curriculum trained PG model moved more aggressively to its death in many cases, but overall possessed a better ability to generalise to its new environments and find the exit.

Technical Challenges

Initial convergence with models proved difficult, even when beginning on basic environments, and required extensive trial and error. This was in part compounded by my own experience with Reinforcement Learning and effectively tuning hyperparameters. This definitely slowed the process of testing, and was a learning process for myself.

Machine learning can be referred to a bit like the ‘dark arts’ sometimes, and identifying what is wrong with a model can feel like searching in the dark. Is it your reward shaping? The observation space? Possibly the learning rate or exploration policy – or none of that at all? This was felt heavily during the DQN testing, and ultimately I could not get substantial progress with any DQN model. I suspect I have overlooked something, and with more time and further work I could fix this.

Training cycles on that note, could be very time consuming. Some runs taking multiple hours to complete, and proved a natural constraint on the extent of what I could allow myself to do. A code change might have to wait an entire day to show any results, as I swept through various hyperparameters with any changed code.

Note:

I have not directly referenced anything in this report and was not sure if it needed a bibliography of sorts for any resources I’ve used to help me write code, so for now I’ve kept them off this document ready for the final report instead.