SMART: Super Mario Agent Reinforcement Training

Maksym Tkachenko
Department of Computer Science
University of Bath
Bath, BA2 7AY
mt2416@bath.ac.uk

James Ravindran
Department of Computer Science
University of Bath
Bath, BA2 7AY
jr2155@bath.ac.uk

Aiden O'Sullivan
Department of Computer Science
University of Bath
Bath, BA2 7AY
atos20@bath.ac.uk

Ronald Lanton Stephen David
Department of Computer Science
University of Bath
Bath, BA2 7AY
rlsd20@bath.ac.uk

1 Problem Definition

The gym-super-mario-bros environment [5] offers a OpenAI Gym [1] compatible interface to the video games Super Mario Bros. and Super Mario Bros. 2 (Lost Levels) on the Nintendo Entertainment System (NES). It uses a wrapper around the SimpleNES emulator [8] while offering a variety of configurations, such as either of the two entire games of 32 stages, a randomised selection of stages or a particular individual stage. We choose to train and run on the first stage, World 1-1.

Each state is the current frame outputted from the emulator, showing the output visible on the screen. The environment potentially allows the full range of combinations of buttons on the NES controller to be pressed (given an action space size of around 256), although the environment offers a choice of limiting the action space down. We decided to restrict Mario's movement using the option *RIGHT_ONLY*, which limits the search space to 5 possible actions, and thus makes it more manageable.

The transition dynamics involve the current state and the action Mario takes at at the frame for that state. The initial states are the start of each level. The terminal states are either dying or ending the level by completing the stage. The game is mostly deterministic, with the only stochastic elements being obstacles (i.e. Mario will not move to the right if there is a wall in the way) and some randomised elements in some levels (as noted by the usage of "Random" in the variable names of the unofficial disassembly [2]).

The reward function is described by the equation $r = v - c + d$, where v is the difference in Mario's x position between the previous and current states (with moving to the right being positive), c is the difference between the clock between the previous and current states (being negative to force the agent to avoid standing still) and d is a fixed negative death penalty to discourage the agent from dying (which is set to -15 when it occurs and 0 otherwise). The reward is finally clipped between the ranges of -15 and 15.

The problem is extremely difficult and cannot be solved using tabular reinforcement learning methods. The state space is enormous due to the immediate representation being a large framebuffer with numerous numbers of pixels, and a underlying game state of numerous combinations of Mario's position in the level, the positions of other enemies, the status of any blocks (whether they are broken or not) etc. It is not feasible to store a table of all action-state pairs in RAM.

2 Background

As tabular reinforcement learning methods are insufficient to solve this environment, we will need to use a function approximated reinforcement learning method in order to generalise to the environment. There are numerous possible methods we can choose from.

Deep Q-learning (DQN) uses deep neural networks as a substitute for the table-based value function to approximate the values instead. DeepMind first introduced this variation of Q-learning by applying it successfully to a selection of Atari games [7], using a conventional neural network [9] that only relies on raw pixels as input and provides an output in place of a value function. As our environment's states are raw framebuffers, this algorithm or a variant of it seems perfectly suited.

There is a major inherent problem with normal Q-learning as identified by Hasselt, Guez, and Silver [3]. Q-learning tends to overestimate the action values estimates because the Q-values are updated based on the optimistic computed maximum expected rewards for the next state. Hasselt, Guez, and Silver [3] proposed solving this by updating two tables at the same time - one to represent the greedy policy (to determine which action to take) and the other to provide an estimate of the value for that action. The latter is used to update the Q-values. They called this double Q-learning.

Both deep Q-learning and double Q-learning can be combined together to leverage the strengths of both to form Double Deep Q-learning (DDQN). In place of the two tables, there is the primary network which is used to select actions for the policy and the target network which is used to update the Q-values upon every step. Combining both these methods means that it scales to environments that tabular-based reinforcement learning methods cannot cope with, while also not estimating Q-values, making it highly suited to our environment.

Dueling DQN is a variation and described as an improvement by Wang et al. [10]. It replaces the two networks seen in DDQN with two functions: one known as the value function which estimates the value of a given state similar to the target network in DDQN and one known as the advantage function to estimate the relative advantage of taking an action to go to that particular state similar to the primary network in DDQN. However, unlike DDQN, both these functions results by splitting the final layers of a single network, and the resulting values are combined to estimate the overall Q-value for a given action.

We also looked at policy gradient methods, which differ themselves by looking at optimising the policy directly in comparison to indirectly inferring a policy function by learning a value-based function through methods such as deep Q-learning. The policy can also be stochastic unlike value-based methods which can lead to more exploratory behaviour rather than exploitive which may avoid the algorithm taking the same trajectory each time getting stuck (i.e. if Mario continually runs into the same pipe and doesn't progress).

Examples of these algorithms include the REINFORCE algorithm and the family of actor-critic algorithm.s The REINFORCE algorithm is Monte Carlo and therefore off-policy, while actor-critic algorithms (such as Advantage Actor-Critic [6]) aim to balance the strengths and weakness of the policy-based and the value-based methods, where both stand for Actor and Critic, respectively. Overall, these algorithms display disadvantages for our given environment such as being too slow (for instance, Monte Carlo based methods inherently requires full episodes to learn from).

In conclusion, we have explored several methods each building on from deep Q-learning, and have therefore gone with Dueling DQN as a result of it building on everything before and addressing the inherent inefficient of previous variants (such as overestimating Q-values). We have rejected alternative methods such as those based on policy gradient methods, as evidence points to the fact that they are slow to train and therefore not feasible in the time constraints given for this paper.

3 Method

3.1 Network Architecture

For our implementation, we utilised a Dueling Deep Q-Network (Dueling DQN) architecture, which splits the network into separate estimators: one for estimating the state value function and another for computing the action advantage function. This architecture allows the network to learn the value of each state without having to learn each action's effect, and therefore reducing the overestimation

of Q-Values which is commonly seen in traditional DQN methods. This is crucial for environments with high-dimensional input spaces like video games. Our network inputs are the raw pixels from the current game frame, which are first preprocessed before being fed into the network.

3.2 Image Preprocessing

Given the complexity of these raw video frames, preprocessing is crucial for reducing the input dimensionality and enhancing overall computational efficiency:

- **Grayscale Conversion:** Each frame, originally in full RGB, is converted into grayscale to reduce the number of input channels from three to one, focusing on structural information rather than colour information.
- **Resizing:** Frames are resized from their original 240x256 pixel resolution to 84x84 pixels to strike a balance between preservation of detail and a level of computational efficiency.
- **Normalisation:** Pixel values are normalized from [0, 255] to [0, 1], in order to facilitate neural network processing.
- **Frame Stacking:** Four consecutive frames are stacked together to form an input tensor. This step is crucial for capturing the temporal progression between each frame, allowing the network to infer motion and directionality.

3.3 Reinforcement Learning Framework

We utilise the OpenAI Gym interface for our Super Mario Bros environment, with state inputs given as processed frames and outputs giving discrete action commands. This modified environment uses frame skipping and reward clipping, speeding up training and stabilising learning:

- **Frame Skipping:** The network only processes every fourth provided frame, effectively increasing the decision period and reducing the temporal correlation between each successive state.
- **Reward Clipping:** Rewards are always restricted to the range [-1, 1] to prevent large reward variations from destabilising the learning updates.

3.4 Deep Q-Networks (DQN)

Deep Q-Networks (DQNs) are the fundamental architecture of our approach, characterised by their multi-layered neural network. Our architecture consists of the following layers:

1. **Input Layer:**
   - **Function:** Accepts the stacked grayscale frames of size 84x84 pixels, forming an input tensor of dimensions 84x84x4 (width x height x channels).
   - **Purpose:** Serves as the entry point for data, feeding the neural network with the current state of the environment in a format it can process.
2. **First Convolutional Layer:**
   - **Configuration:** Utilizes 32 filters of size 8x8 with a stride of 4.
   - **Activation:** ReLU (Rectified Linear Unit).
   - **Purpose:** Extracts basic features from the input (like edges and gradients) which are needed for understanding the spatial structure in the frames.
3. **Second Convolutional Layer:**
   - **Configuration:** Employs 64 filters of size 4x4 with a stride of 2.
   - **Activation:** ReLU.
   - **Purpose:** Builds on the features captured in the first layer and starts recognising more complex patterns such as shapes and object parts, essential for distinguishing different game elements.
4. **Third Convolutional Layer:**
   - **Configuration:** Uses 64 filters of size 3x3 with a stride of 1.

- **Activation:** ReLU.
- **Purpose:** Further refines the features from the second layer, to aid the detection of features like combinations of objects, which are crucial for decisions in gameplay strategy.

5. **Flattening Layer:**
    - **Purpose:** Converts the multidimensional output of the last convolutional layer into a flat vector. This can then be fed into the fully connected layers. This layer is essential for transitioning from feature extraction to feature interpretation.
6. **First Fully Connected Layer:**
    - **Configuration:** Consists of 512 units.
    - **Activation:** ReLU.
    - **Purpose:** Integrates all of the features learned by the convolutional layers, starting the process of relating visual patterns to actions. This layer is crucial for learning the relationships between visual input and game dynamics.
7. **Output Layer:**
    - **Configuration:** Number of units dependant on action space size (5 or 7).
    - **Activation:** Linear.
    - **Purpose:** Produces the final Q-values for each action available to the agent from the current state. These values directly influence the decision-making process, guiding the agent to take the action with the highest expected reward.

3.5 Double Deep Q-Network (DDQN)

To prevent the overestimation bias of action values observed in traditional DQN, we extend our model to Double DQN. This aims to address the issue, separating selection and evaluation.

This uses the same network architecture as DQN, but introduces a second target network, with weights copied from the trained network periodically. This weights are held fixed between individual updates. The action selection is therefore decoupled from the target Q-value generation. The action is selected using the main network but the Q-value for updating the network is estimated in the target network. The structure for both the primary and secondary network are identical to the one implemented above. This target network is updated every 5000 steps in order to refresh the weights

3.6 Dueling Deep Q-Network (Dueling DQN)

To further refine our model, we implemented a Dueling Deep Q-Network (Dueling DQN). This provides an architectural split in the network, estimating the state value function and the action advantage function separately.

The initial layers of the network are identical to those used in DQN and DDQN above. However, the network diverges into two distinct pathways after the last convolutional layer. One pathway estimates the scalar value of the state, and the other estimates the vector advantage of each action.

This separation allows the Dueling DQN to learn which states are valuable (or invaluable) without having to learn the effect of each action for each state. This is particularly beneficial in complex environments where many actions do not affect the outcome. This is very important in the context of our environment when an obstacle or enemy is ahead.

The final Q-value for each action is computed, combining the state value and the advantages of the actions. These are adjusted by subtracting the mean advantage, and therefore stabilising the training further by keeping the value estimates grounded.

3.7 Experience Replay and Optimisation

In order to stabilise the training algorithms, our system uses an experience replay buffer to store and reuse past agent experiences, decoupling the temporal sequence of observed data. We maintain a replay buffer of up to 30,000 experiences, allowing the agent to learn from a diverse set of state

transitions. The network is updated by sampling small batches of experiences randomly from the buffer to perform gradient descent and therefore minimizing issues related to correlated data.

Training is conducted using the Adam optimiser with a learning rate of 0.00025. We periodically update the weights of a target network to stabilize the predictions during the Q-value estimation. The network is trained using the Huber loss, rather than the squared error loss, as it is less sensitive to outliers, thereby providing more stable and robust learning.

3.8 Exploration Strategy

Our agent employs an epsilon-greedy strategy, balancing exploration with exploitation. Initially, epsilon is set to 1.0 to encourage exploration, which is exponentially decayed at a rate of 0.9997 per step to a minimum of 0.01. This causes a gradual shift towards exploiting the learned policies as the agent becomes more familiar with the task.

3.9 Logging and Monitoring

We implemented a comprehensive logging system to monitor training progress throughout. Various metrics such as episode length, rewards, losses, and the value of epsilon are logged after each and every episode in order to track performance and adjust strategies as needed. Visualisation tools are utilised to plot these metrics, providing clear visual feedback on the agent's learning progress over time.

3.10 Computational Resources

Training is completed on a GPU-enabled setup in order to handle the intensive computational demands of processing deep neural networks and large datasets. This infrastructure is crucial to efficiently manage the high bandwidth of data and various complex model calculations required in training.

4 Results

We ran our agent for a total of over 10,000 episodes on World 1-1 and World 2-4. We collected two key metrics: the mean episode length and the mean episode reward, and these are plotted as graphs as shown by figures 1 and 2 for World 1-1 and figures 3 and 4 for World 4-2. Figure 5 also shows the completion ratios for World 4-2 over time. Larger versions of these graphs can be seen in the appendices. Overall, these graphs show that our agent has learnt successfully, as a strong and stable improvement occurs until the results plateau, insinuating that the agent is reaching the end of the levels and has reached both the maximum limit of episode length and reward.

The world record for World 1-1 in Super Mario Bros. is 52 seconds [11], while our agent completes the level in only 57.73 seconds, only 5.73 seconds slower. It has beaten the average human player times that are around 64 seconds, showing that agent is superior than human performance. Figure 1 and 3 in particular show the reward per episode for an agent that takes random actions, showing our

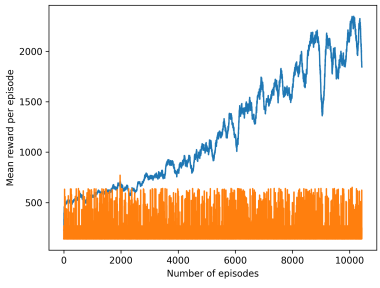

Figure 1: The mean average reward of each episode over time on World 1-1 (the orange line shows a random agent)

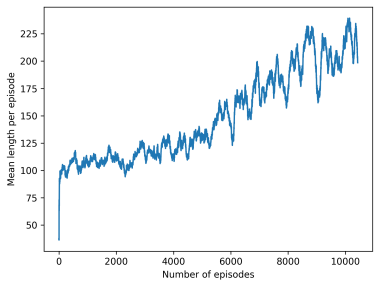

Figure 2: The mean average length of each episode over time on World 1-1

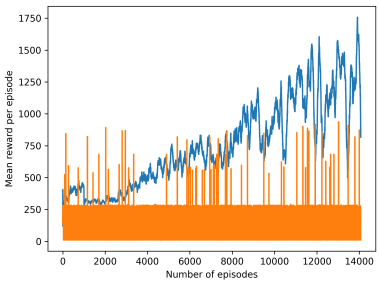

Figure 3: The mean average reward of each episode over time on World 4-2 (the orange line shows a random agent)

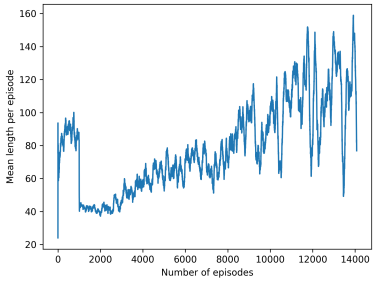

Figure 4: The mean average length of each episode over time on World 4-2

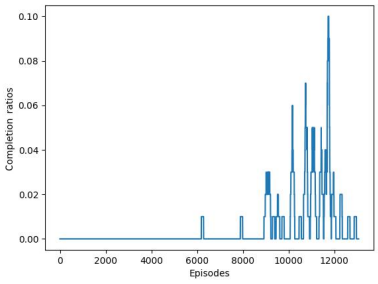

Figure 5: The completion ratios of each episode over time on World 4-2

agent has also far surpassed this and is not simply taking random actions in order to complete the stages.

After training and running on World 1-1, the weights were re-used and fine-tuned for World 4-2 (which is likely responsible for the initial spike seen in figure 4). However, re-using the new weights for World 1-1 showed poor performance despite strong performance for World 4-2. This suggests our agent generalises poorly. If we had more time, it would be better to train it on a bunch of levels simultaneously, but it's likely that that this would have led to a complex architecture for our neural network, and therefore it would have taken much more training time, so we therefore did not pursue this option.

5 Discussion

As mentioned above, our agent achieved commendable success in the Super Mario Bros environment, approaching near world record times in World 1-1. This demonstrates the agent's efficiency and its capacity to optimise paths and actions within the game, surpassing average human player times. The training process shows a consistent improvement in both rewards and episode lengths, indicating effective learning and adaptation to the environment.

The implementation of the Dueling Deep Q-Network architecture allowed the agent to effectively disentangle the assessment of state values from the advantages of the actions within each state. This likely lead to more accurate Q-Value estimations and helped reduce traditional overestimations seen in more simple models.

The use of Double DQN mitigated the problem of overestimation bias.The secondary network provided stable updates to the primary network's predictions, enhancing the reliability of action-value estimations. This supported sustained learning progression.

The incorporation of the epsilon-greedy strategy is an effective method of balancing exploration and exploitation. As epsilon values decay, this often results in inadequate exploration, limiting the agent's exposure to a variety of strategic pathways.

The agent was also specifically trained and evaluated on World 1-1 and World 4-2, which might not sufficiently represent the diversity of levels across the game. This introduces a risk of overfitting to these levels, optimising for specific scenarios at the cost of general performance across the whole game.

6 Future Work

There are various potential improvements that could be made to this project. One of these would be Rainbow DQN [4]. This is a combination of several enhancements to the standard DQN, including some of the ones used in our implementation such as Double and Duelling DQN methods. One of the main improvements would involve the implementation of Prioritised Experience Replay (PER) where transitions with higher temporal difference errors are prioritised. By focusing on these transitions, PER can lead to more efficient training cycles.

Multi-step learning could also be implemented where multiple future steps are considered to adjust the Q-Values predictions, instead of updating them based on the next state only. This can help in capturing the long-term benefits of actions more effectively, crucial in complex sequences with delayed consequences, which are likely in our environment.

Distributional DQN models could be used, where instead of estimating the average reward expected, the entire distribution of possible outcomes are modelled, capturing the intrinsic uncertainty in the dynamics of the environment. This is likely to provide a more robust decision-making process in uncertain situations where random elements and varied level layouts are common, as in our environment.

The final enhancement made by Rainbow DQN is the implementation of Noisy Networks. This technique introduces noise to the weights of the network, to aid in exploration, replacing the epsilon-greedy method where the randomness does not adapt during training. This might allow for more effective strategies for overcoming obstacles without manual tuning of exploration hyperparameters.

7 Personal Experience

We initially had problems as two group members withdraw themselves from our group, proving to be very disruptive to progress as we were only down to four members from our initial six. Because of the different courses we were attending, we also struggled to organise meetings as our lectures were clashing. Despite these innumerable challenges, we successfully managed to preserve and complete this project to the highest possible standard.

References

[1] Greg Brockman et al. *OpenAI Gym*. Apr. 27, 2016. URL: https://github.com/openai/gym.

[2] doppelganger. *A Comprehensive Super Mario Bros. Disassembly*. Nov. 9, 2012. URL: https://gist.github.com/1wErt3r/4048722.

[3] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. Dec. 8, 2015. DOI: 10.48550/arXiv.1509.06461. arXiv: 1509.06461[cs]. URL: http://arxiv.org/abs/1509.06461.

[4] Matteo Hessel et al. "Rainbow: Combining improvements in deep reinforcement learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 1. 2018.

[5] Christian Kauten. *Super Mario Bros for OpenAI Gym*. Apr. 24, 2018. URL: https://github.com/Kautenja/gym-super-mario-bros.

[6] Alexander Van de Kleut. *Actor-Critic Methods, Advantage Actor-Critic (A2C) and Generalized Advantage Estimation (GAE)*. Alexander Van de Kleut. July 16, 2020. URL: https://avandekleut.github.io/a2c/.

[7] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. Dec. 19, 2013. DOI: 10.48550/arXiv.1312.5602. arXiv: 1312.5602[cs]. URL: http://arxiv.org/abs/1312.5602.

[8] Amish Kumar Naidu. *SimpleNES*. June 1, 2016. URL: https://github.com/amhndu/SimpleNES.

[9] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. Dec. 2, 2015. DOI: 10.48550/arXiv.1511.08458. arXiv: 1511.08458[cs]. URL: http://arxiv.org/abs/1511.08458 (visited on 05/08/2024).

[10] Ziyu Wang et al. *Dueling Network Architectures for Deep Reinforcement Learning*. Apr. 5, 2016. DOI: 10.48550/arXiv.1511.06581. arXiv: 1511.06581[cs]. URL: http://arxiv.org/abs/1511.06581.

[11] Nan Zhang and Zixing Song. "Super Reinforcement Bros: Playing Super Mario Bros with Reinforcement Learning". In: (2021).

Appendices

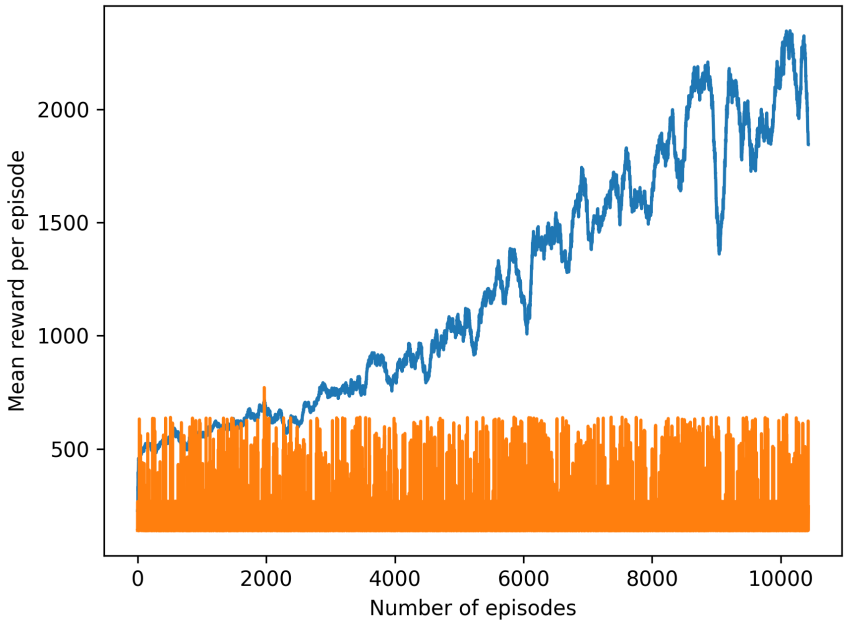

Figure 6: The mean average reward of each episode over time on World 1-1 (the orange line shows a random agent

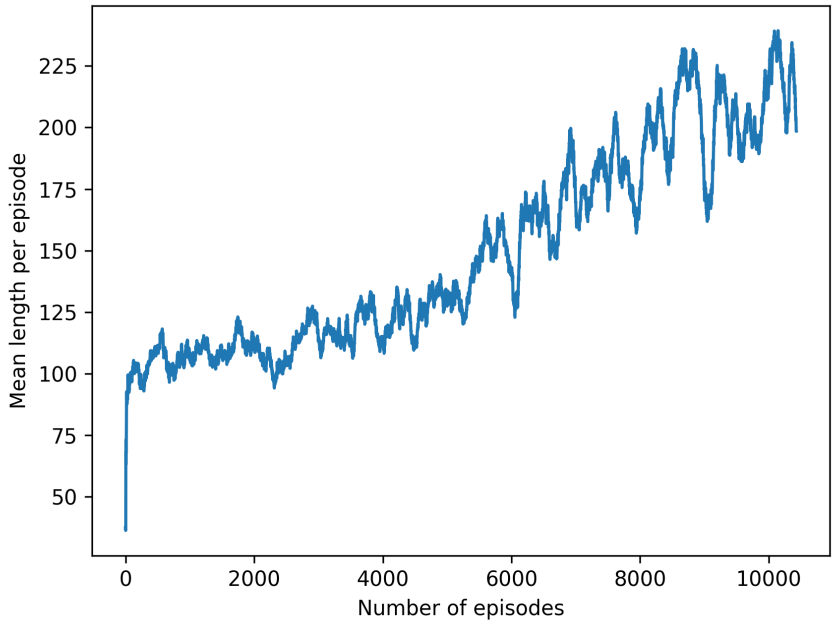

Figure 7: The mean average length of each episode over time on World 1-1

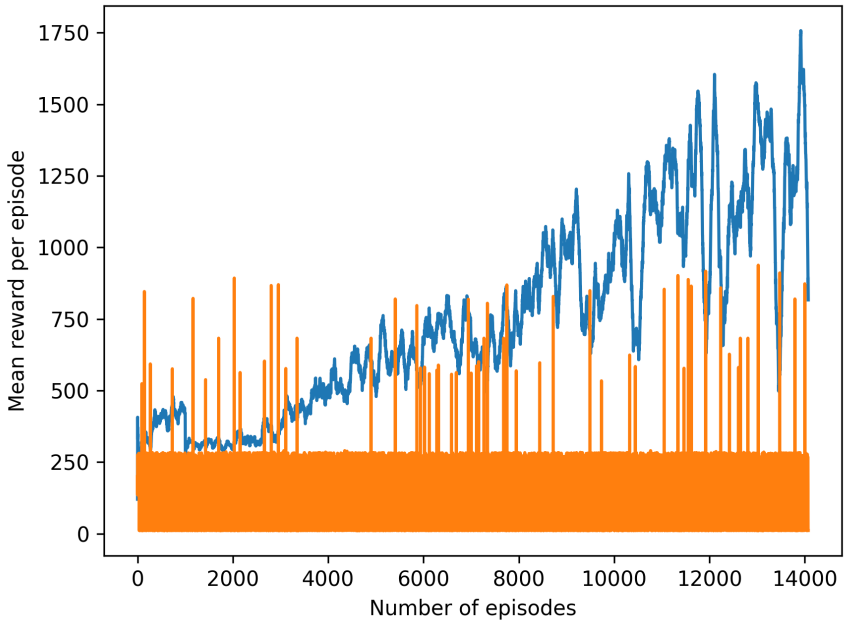

Figure 8: The mean average reward of each episode over time on World 4-2 (the orange line shows a random agent)

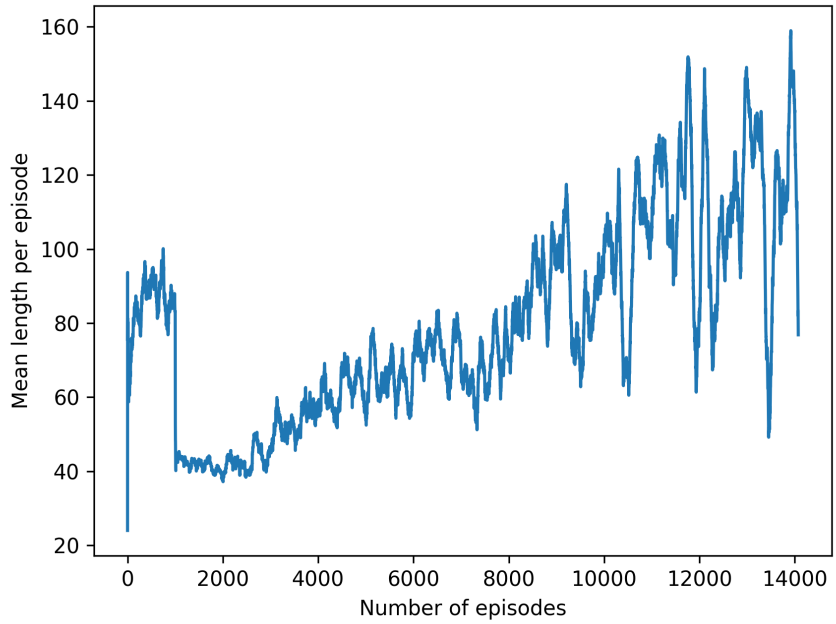

Figure 9: The mean average length of each episode over time on World 4-2

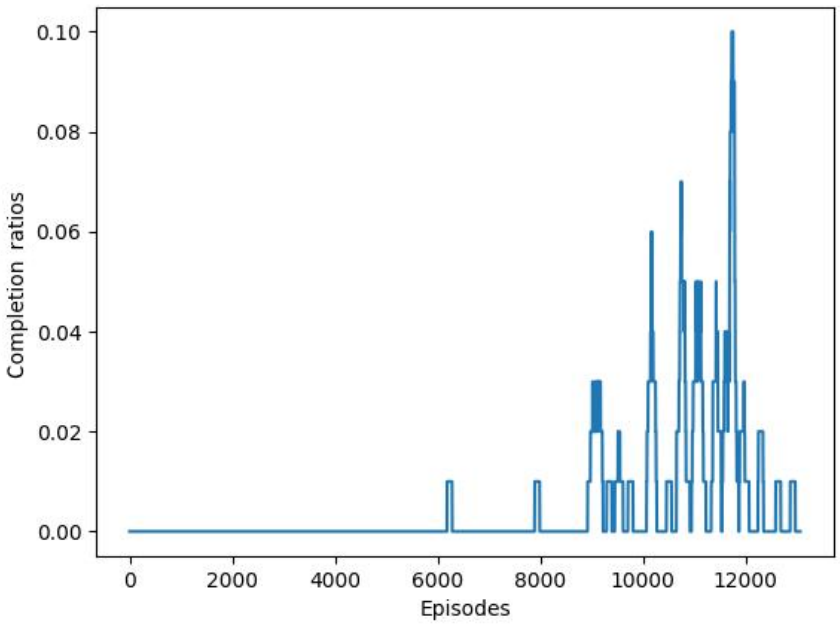

Figure 10: The completion ratios of each episode over time on World 4-2