Machine Learning Final Report

Meghan Rachel Will Edgar

Introduction and Intent

Reinforcement learning techniques are often implemented to play the classic video game Snake. Shallow reinforcement learning models, such as Q-Learning and SARSA (State Action Reward State Action), and deep learning models, such as deep Q-learning (DQN) and prioritized experience replay, have all been applied to Snake in previous work. For example, Wei et al. (2018)[1] use a convolutional DQN with reward shaping and dual experience replay to surpass human-level performance. Further, Almalki and Wocjan (2019)[2] explore the use of SARSA and DQN within a more complex Snake environment, introducing poisonous collectable items. Lastly, Sebastianelli et al. (2021)[3] apply Deep Q-Learning using sensor measurement data and detail their architecture for the Snake game. In our project, we add to this discussion by exploring how an agent trained to play various versions of Snake with deep reinforcement learning (deep RL) and DQN performs in new and unfamiliar situations of varying complexities. All code and trained models are publicly available at github.com/eleon024/MachineLearningFinalProject. This is one of the first places readers will look for reproducibility information.

1 Deep Reinforcement and Deep Q-Learning

Reinforcement learning consists of a software agent in an environment gaining rewards and punishments to promote a desired outcome. In the context of Snake, the environment is the game itself, and the agent is the computer player. The agent is rewarded when it touches food, while punished when it touches the wall or itself. The rewards inform the agent on how good it is doing. Based on rewards, the agent tries to find the next best action, or rather, where they should move next on the grid. Deep reinforcement and deep Q-Learning introduces a deep neural network to make the prediction of the next best action. We use the Linear Q_Net model, a feed forward neural network with linear layers. This model decides whether to go left, right, or straight. Its starting states consist of knowledge of relative danger location [danger_straight, danger_left, danger_right], current snake direction [direction_left, direction_right, direction_up, direction_down], and food direction [food_left, food_right, food_up, food_down].

DQN specifically focuses on the Q value and updating it with every iteration. There are generally five main steps involved in this process:

1. Initialization: Setting parameters
2. Exploration & Exploitation: Model randomly moves (exploration) or model predicts best move based on current states (exploitation)
3. Perform Action
4. Reward Determination
5. Updating Q-value and Training Model - updates based on Bellman Equation and minimizes the loss function.

   Formula for updating the Q-value:

$$NewQ(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma maxQ'(s',a') - Q(s,a)]$$

Formula of the loss function:

$$L(\theta) = E[(r + \gamma max_a, Q(s', a'; \theta^-) - Q(s, a; \theta))^2]$$

Repeat

2 Methods Overview

We adapt a basic Snake game and deep RL setup from Python educator Patrick Loeber. This includes all of Snakes traditional functionality implemented using Pygame, a deep Q-learning model implemented using PyTorch, and an agent informed by the model that plays the game. We implement features to make Snake more complex, using stationary and moving obstacles. Next, we test model generalization. We train a model on basic Snake, test it in a simple and complex environment, and compare performances. We also train and evaluate models on various versions of the complex Snake environment, and test them in both environments. We train models with varying amounts of stationary and moving obstacles to examine the effect of obstacle amounts on model performance. Finally, we analyze our results. We compare model scores, number of moves, and survival times between a models original training environment and its new environment to evaluate how well it handles unfamiliar environments.

3 Data Generation & Key Metrics

- *Score:* the number of items collected by the model before losing
- *Number of Moves:* the average number of steps the model takes to get to the next item
- *Survival Time:* the average time each model takes before crashing

4 Hypothesis

As the complexity of the training environments of the model increases, when tested in the ***basic environment***, we expect:

1. *Scores to stay consistent* because the basically trained model was trained on the current environment and should be optimized for it, and the complexly trained model will generalize to simpler problems and perform well on any environment of lower complexity involving obstacles.
2. *Number of steps to increase* because the complexly trained models will adapt to avoiding obstacles and be more conservative with straightforward movements, taking a longer path to get to food.
3. *Survival time to increase* because the complexly trained models will take longer, more complicated paths to food, adapting to obstacles in their training environment, which will increase overall time.

As the complexity of the training environments of the model increases, when tested in the ***complex environment***, we expect:

4. *Scores to increase* because the basically trained model will not be able to generalize well with obstacles, and the complexly trained models will be optimized for the current environment
5. *Number of steps to increase* because basically trained models will not be as good at making paths to food that avoid obstacles, causing them to move fewer spaces than complexly trained models
6. *Survival time to increase* because basically trained models will have a harder time avoiding obstacles, and will not create paths that successfully get food, causing shorter survival time

5 Training Implementation

At the beginning of training, each agents neural network weights are initialized randomly, according to PyTorchs default method, and the replay buffer is set to hold up to 100,000 transitions. We base our implementation on Patrick Loebers Pygame starter code [4] and build our Deep Q-Networks (DQNs) in PyTorch [5].

We consider two distinct network architectures:

- **Convolutional Q-Net** (ConvQNet), which comprises of three convolutional layers (3x3 kernels with 32, 64, and 64 feature maps) followed by two fully connected layers of sizes 512 and 3 (actions)
- **Linear Q-Net** (LinQNet), which consists of two fully connected layers of sizes 256 and 3

All experiments use the follopwing hyperparameters:

- Learnieng rate (α) = 0.001 (Adam optimizer)
- Discount factor (γ)= 0.9
- Replay batch size = 1000
- Target network update frequency = 1000 steps
- Epsilon-greedy exploration: ϵ decays by 1 per game, yielding a linearly decreasing exploration probability over the first 80 games, then pure exploitation afterwards
- Maximum training episodes = 1000 per environment
- Speed = 40,000,000

5.1 Initial Convolutional Implementation (Failure)

Our first attempt employs the ConvQNet architecture, expecting that spatial feature extraction accelerates learning. However, after 10,000 games in the basic Snake environment, the ConvQNet averages only 2.55 points per game. We attribute this to overparameterization relative to the grid input and limited replay diversity.

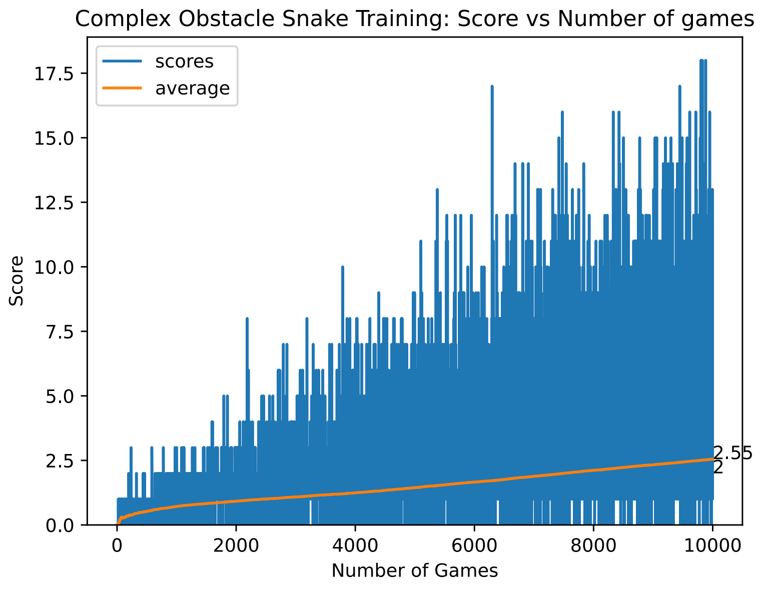


Figure 1: ConvQNet Training in Basic Snake Environment

In the training curve, however, individual game scores remain tightly clustered near zero for the first several thousand games, showing almost no random variance. As training progresses, the maximum score achieved in each game begins to climb steadily, and the blue bars fan out in a near-right triangle shape, flat along the left (early games) and rising sharply along the right (later games). Meanwhile, the orange moving-average line increases almost linearly from approximately 0 at game 0 to about 2.5 by game 10,000, reflecting very gradual but consistent learning.

5.2 Sucessful Linear Implementation

Switching to LinQNet yields immediate improvements. The reduced parameter count stabilizes training, allowing the agent to learn reliable policies by around 100 games in the basic environment. We then train the same network on increasingly challenging scenarios, first adding static obstacles, then only moving obstacles, then both, and observe that it continues learning even as complexity grows, with gradual reductions in score and survival time as obstacle density increases.

5.3 Why does LinQNet outperform ConvQNet in this case?

The Linear Q-Net outperforms the convolutional variant because the state representation and training methods used dont benefit from spatial feature extraction, but instead benefit from faster, stable convergence. In our setup, the agents input is already a compact feature vector (e.g., danger flags, food location, current direction) rather than raw pixels or a full grid image, so the ConvQNets filters have little meaningful structure to learn. We convert parts of the model and agent such that the convolutional network can extract spatial information from the board state, but that does not give us any improvements in score. We attribute this to overparameterization, lack of sample complexity, and not enough games played.

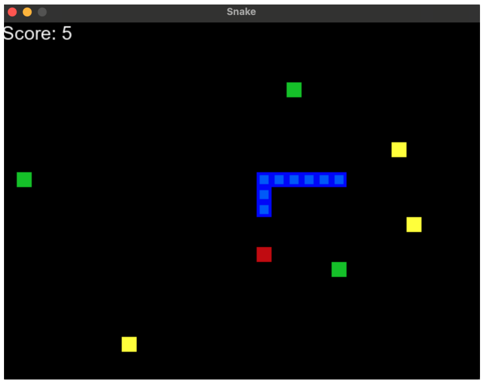


Figure 2: Training an agent with 3 moving obstacles, 3 static obstacles, and 1 piece of food

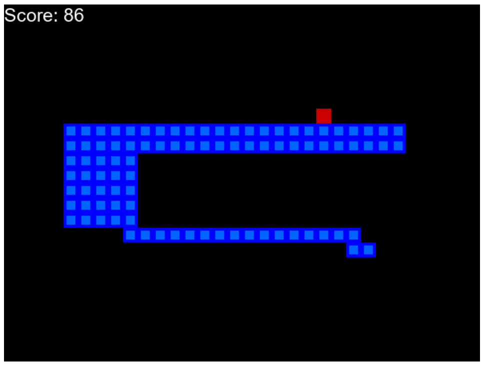
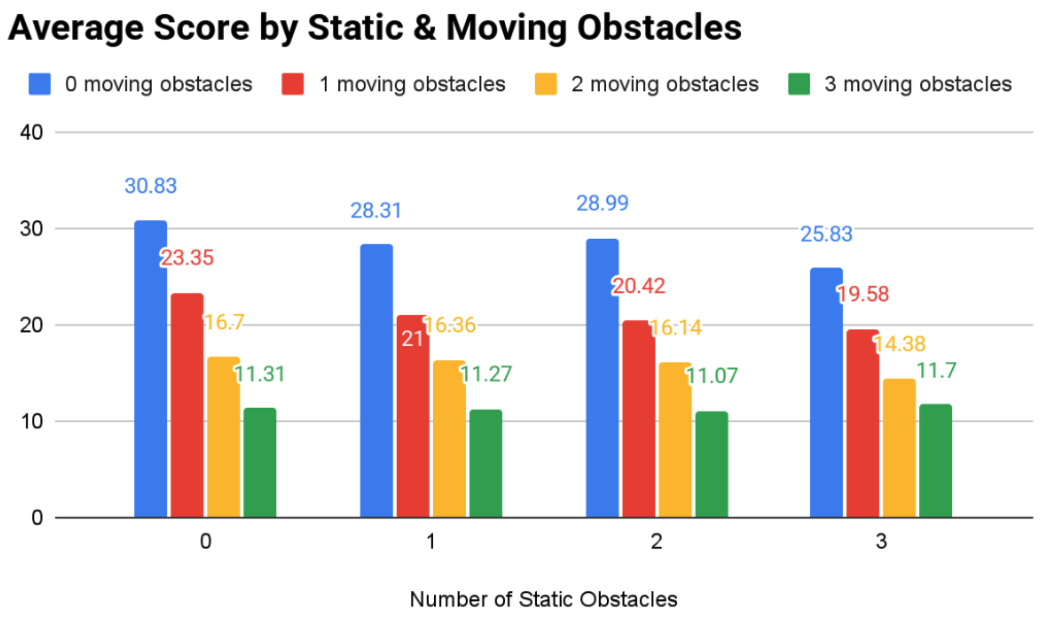


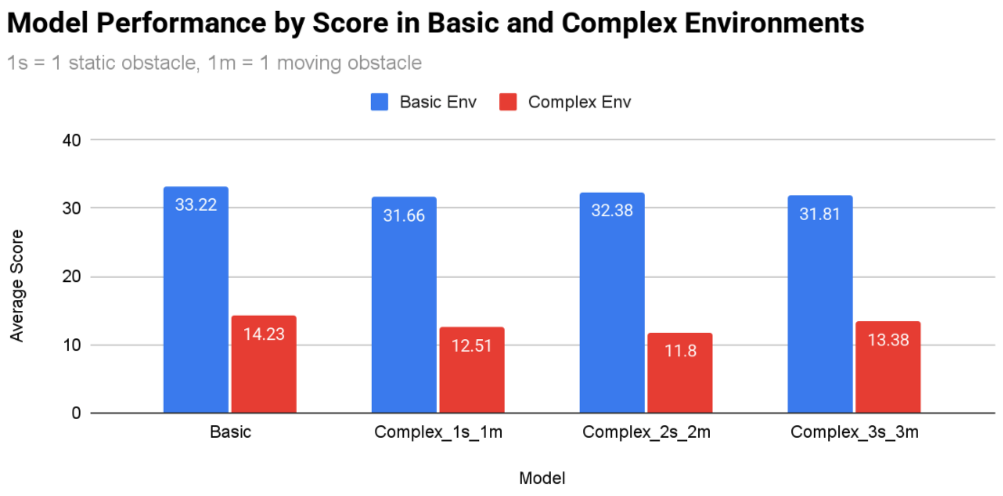
Figure 3: The complex agent playing the basic environment game

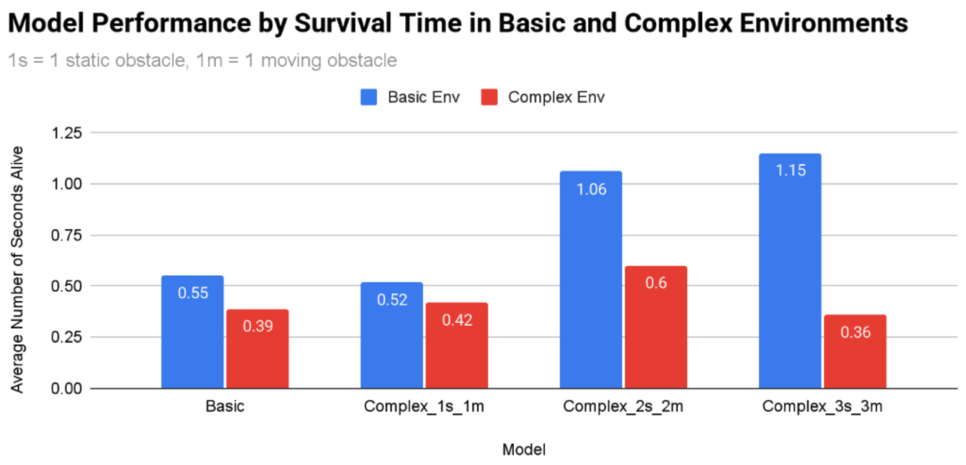
6 Testing

After training, we test how select models perform in new environments. We pick the model trained in the basic Snake environment and three models trained in the complex Snake environment. The three non-basic models are trained in environments with increasing levels of complexity, from one stationary plus one moving obstacle, to three of each. We choose these models as representatives of the main sections of Training Table, maintaining an equal amount of moving and static obstacles to simplify and clarify testing. To test our models, we prevent our Snake agent from training short or long-term memory, freezing its learning. Next, we test each model for one thousand games in the basic Snake environment with no obstacles, and the most complex snake environment with three stationary and three moving obstacles. We record and plot the average number of moves per fruit collected, survival time, and score achieved in each game. Testing Table shows the results of our tests, including environment, number of obstacles, and thousand-game averages of moves per fruit, survival time, and score for each model.

7 Results







On average, each static obstacle decreases the score by 0.89 points, while each moving obstacle decreases it by 5.71 points, 6.4 times more than a static obstacle.

8 Conclusion

In conclusion, our results supported most of our initial hypotheses. We predicted that increasing the complexity of the models training environment would not have a significant impact on the models scores in the basic environment, however, the complexly trained models would have increases in both the number of steps and survival time. The data confirmed parts of this prediction. The scores remained consistent across all training conditions, ranging from 31.66 to 33.22, and showed no clear trend based on training complexity. Survival time increased as training complexity increased. For example, the model trained in the basic environment reached its average score in 0.55 seconds, while the model trained with three static obstacles and three moving obstacles took 1.15 seconds. These results suggest that the more complexly trained model needed more time to achieve similar outcomes. However, the number of steps did not follow our expectations. We expected the more complexly trained models to take more steps, but the step counts remained consistent, ranging from 19.9 to 20.9, without any significant trend.

In the complex test environment, the data did not support our hypotheses. We expected the models trained in complex environments to achieve higher scores, take more steps, and survive longer than the models trained in less complex environments. Instead, the basic model outperformed the others with the highest average score of 14.23, while the complex_2s_2m (two static obstacles and two moving obstacles) model had the lowest score at 11.8. The average number of steps varied slightly between models, ranging from 17.93 to 19.71 steps. Survival time also showed no clear pattern, with the complex_2s_2m model lasting the longest and the complex_3s_3m model the shortest.

9 Discussion & Further Research

Our results show that models trained in more complex environments behave more conservatively, which increases their survival time without improving their overall performance. In every trial, each model scored higher in the basic environment than in the complex one. We found no strong evidence that a model trained in a more complex environment performs better under complex conditions. This challenges the assumption that training in a simpler environment leads to worse performance on more difficult tasks. These findings are interesting and demonstrate that a basic trained model may be preferable based on training efficiency and the specific metrics one is targeting. With further research, we are interested in expanding our game to support two players or multiple snakes within the same environment. This added complexity could help us explore how the model responds to additional moving objects and competition for food. We are also curious about how different hyperparameters, such as the snakes speed or the size of the grid, might influence learning and performance. Additionally, it could be valuable to track new benchmarks, like how the snake dies and whether that differs across models. For this project, we use deep reinforcement learning (deep RL) and DQN to train the snake, but the snake could also be trained using supervised learning techniques.

Related works

A. A Deep Q-Learning based approach applied to the Snake game
Sebastianelli et al. paper discusses how Deep Q-Learning can be applied to the Snake game. Deep Q-Learning combines Q-learning algorithms with deep neural networks. As opposed to other research, this paper uses sensor measurement data instead of CNNs. The agent then uses this data within a Reinforcement Learning framework, where the environment is based on a Markov decision process and the agent acts given its state space, action space, and rewards. Deep Q-Learning is helpful for the Snake game because this approach can handle complexity and a large number of states. Their Reinforcement Learning system utilizes a vector of binary values to depict the state of the agent: whether the snake is in a dangerous position, how the snake moves, and the position of the snakes head with respect to the fruit. Their DQN agent architecture includes five connected layers, including input, three layers that utilize ReLU and Dropout to extract the features and predict the best action for each visited state, and output. The paper also details its hyperparameter tuning process and the resulting effects.

B. Autonomous Agents in Snake Game via Deep Reinforcement Learning
Classical reinforcement methods such as Q-Learning and SARSA have long been the foundation for grid-world tasks like Snake. The emergence of end-to-end deep Q-networks (DQN) by Mnih et al. (2015) demonstrated that a convolutional network trained on raw pixel inputs can achieve human-level skill across Atari games, establishing the standard architecture for deep RL. Applied to the Snake Game, Wei et al. (2018) stack four HSV (hue, saturation, and value)-filtered frames as input to a three-layer CNN with a 512-unit fully connected head and augment standard DQN with logarithmic distance-based reward shaping, post apple training gaps, timeout penalties, and a dual-pool experience replay that separates high-reward from ordinary transitions. Their refined DQN significantly outperforms a standard DQN baseline in both average score and survival time and even beats human-level performance on the Snake game.

C. Exploration of Reinforcement Learning to Play Snake Game
Almalki and Wocjan explore the combination of Deep Reinforcement Learning methods and neural networks in training agents in the Snake game. Their implementation uses both the SARSA algorithm, an on-policy algorithm that continuously evaluates its decisions, and Deep Q-Learning (DQN) built using Keras. Unlike the traditional approach of Machine Learning, the model uses the Bellman equation in which Q-values, or rather the expected rewards, are iteratively updated during training. This neutral network architecture implements 120 hidden neurons with a dropout layer for optimization. To evaluate generalizability, the researchers construct more complex game mechanics for the models to be tested and trained with. Specifically, they introduce poisonous candies, which creates a new lose condition if eaten. And, at the end, their model using Deep Reinforcement Learning with the SARSA algorithm, provided the best efficiency compared to existing techniques because of its quick task performance.

References

[1] Zhepei Wei, Di Wang, Ming Zhang, Ah-Hwee Tan, Chunyan Miao, and You Zhou. Autonomous agents in snake game via deep reinforcement learning. In *2018 IEEE International Conference on Agents (ICA)*, pages 20–25, 2018. doi: 10.1109/AGENTS.2018.8460004.

[2] Ali Jaber Almalki and Pawel Wocjan. Exploration of reinforcement learning to play snake game. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 377–381, 2019. doi: 10.1109/CSCI49370.2019.00073.

[3] Alessandro Sebastianelli, Massimo Tipaldi, Silvia Liberata Ullo, and Luigi Glielmo. A deep q-learning based approach applied to the snake game. In *2021 29th Mediterranean Conference on Control and Automation (MED)*, pages 348–353, 2021. doi: 10.1109/MED51440.2021.9480232.

[4] Patrick Lober. snake-ai-pytorch: Implementation of snake ai using pytorch. https://github.com/patrickloeber/snake-ai-pytorch, 2021. Commit version abc1234, accessed 2025-05-10.

[5] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. URL https://arxiv.org/abs/1912.01703.

A Appendix with Tables, Figures, and Graphs

Training Table

Table 1: Training results across obstacle configurations

Game Mode	Static Obstacles	Moving Obstacles	Average Score
Basic	0	0	30.83
Complex	1	0	28.31
Complex	2	0	28.99
Complex	3	0	25.83
Complex	1	1	21.00
Complex	2	1	20.42
Complex	3	1	19.58
Complex	1	2	16.36
Complex	2	2	16.14
Complex	3	2	14.38
Complex	1	3	11.27
Complex	2	3	11.07
Complex	3	3	11.70
Complex	0	1	23.35
Complex	0	2	16.70
Complex	0	3	11.31

Testing Table

Table 2: Testing performance of models in basic vs. complex environments

Model	Env	Static Ob	Moving Ob	Avg Score	Avg Moves	Survival Time
Basic	Basic	0	0	33.22	19.90	0.55
Basic	Complex	3	3	14.23	18.10	0.39
Complex_3s_3m	Basic	0	0	31.81	20.90	1.15
Complex_3s_3m	Complex	3	3	13.38	19.71	0.36
Complex_2s_2m	Basic	0	0	32.38	20.50	1.06
Complex_2s_2m	Complex	3	3	11.80	18.70	0.60
Complex_1s_1m	Basic	0	0	31.66	19.70	0.52
Complex_1s_1m	Complex	3	3	12.51	17.93	0.42

Training Figures

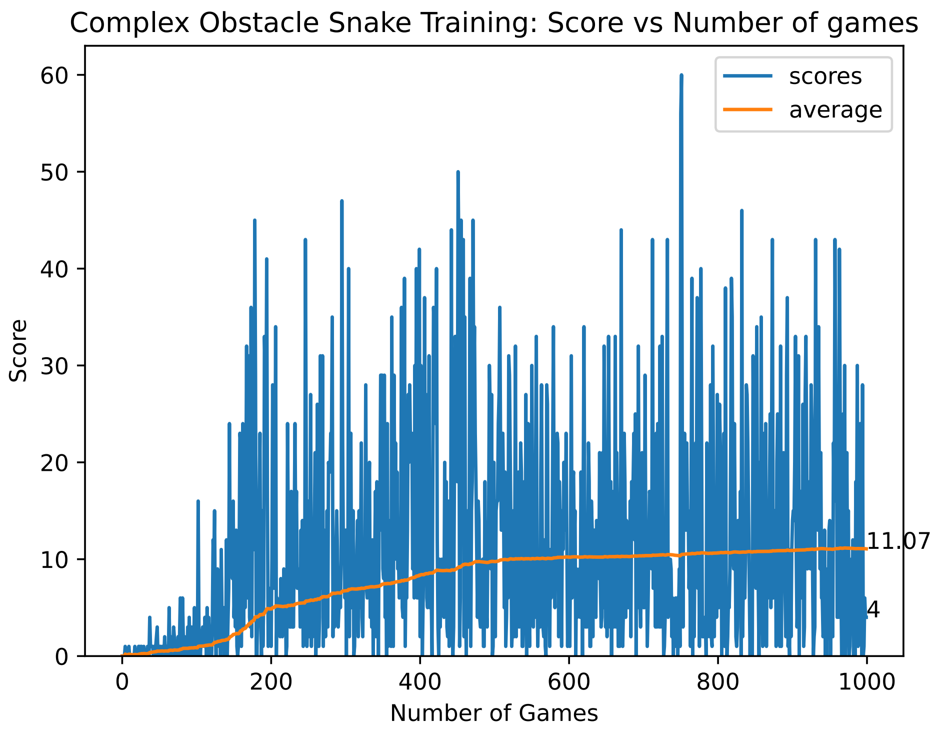


Figure 4: 1000__games__complex__2s__3m__11.07__mean.svg

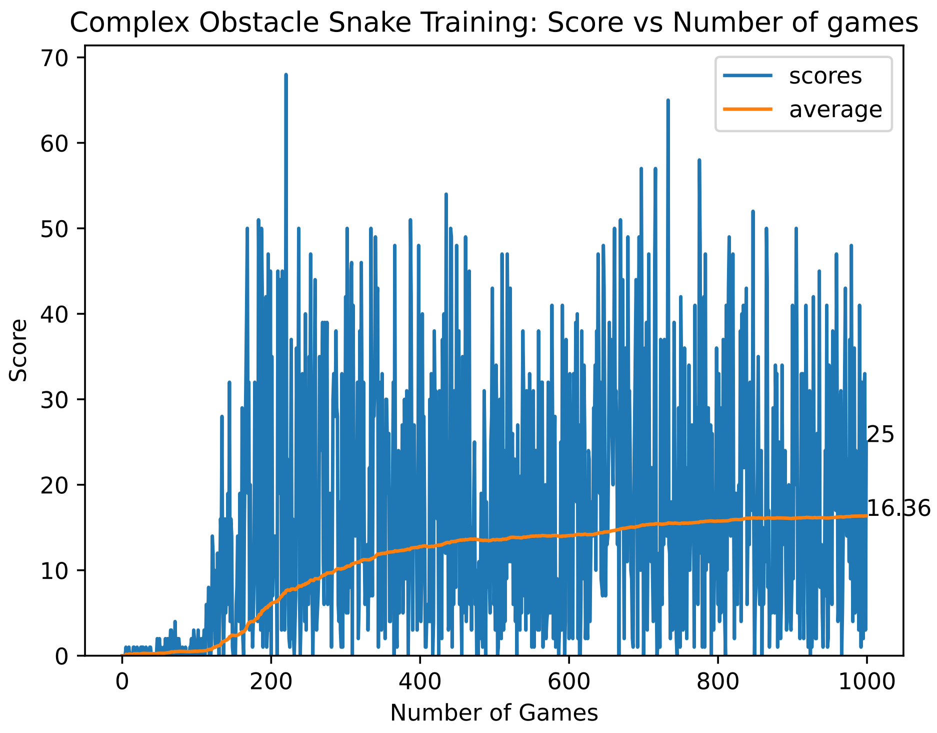


Figure 5: 1000__games__complex__1s__2m__16.36__mean.svg

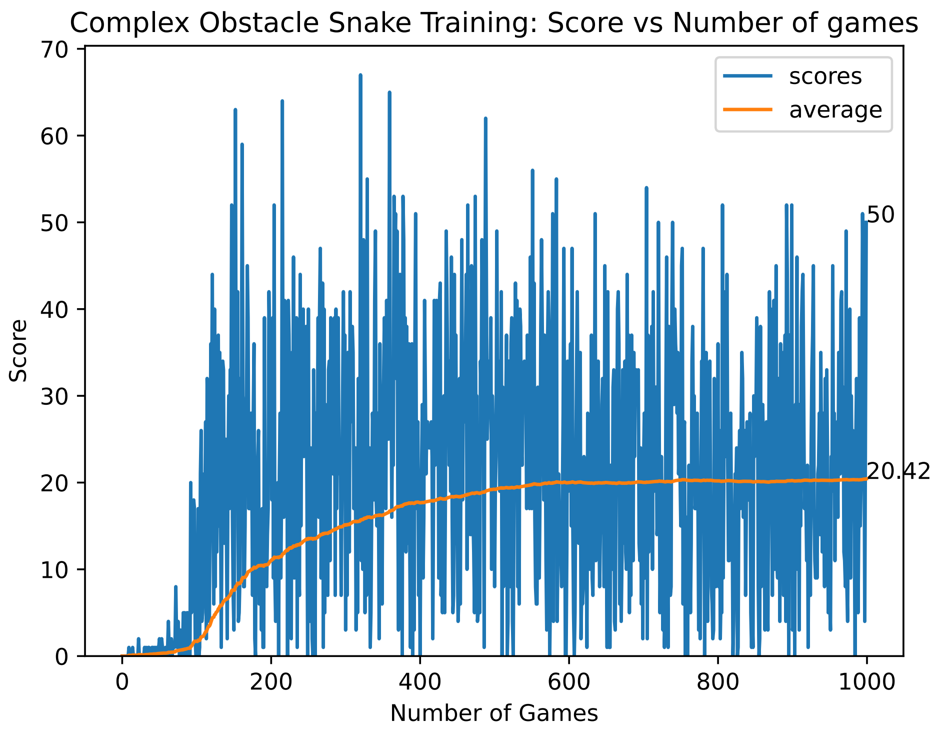


Figure 6: 1000__games__complex__2s__1m__20.42__mean.svg

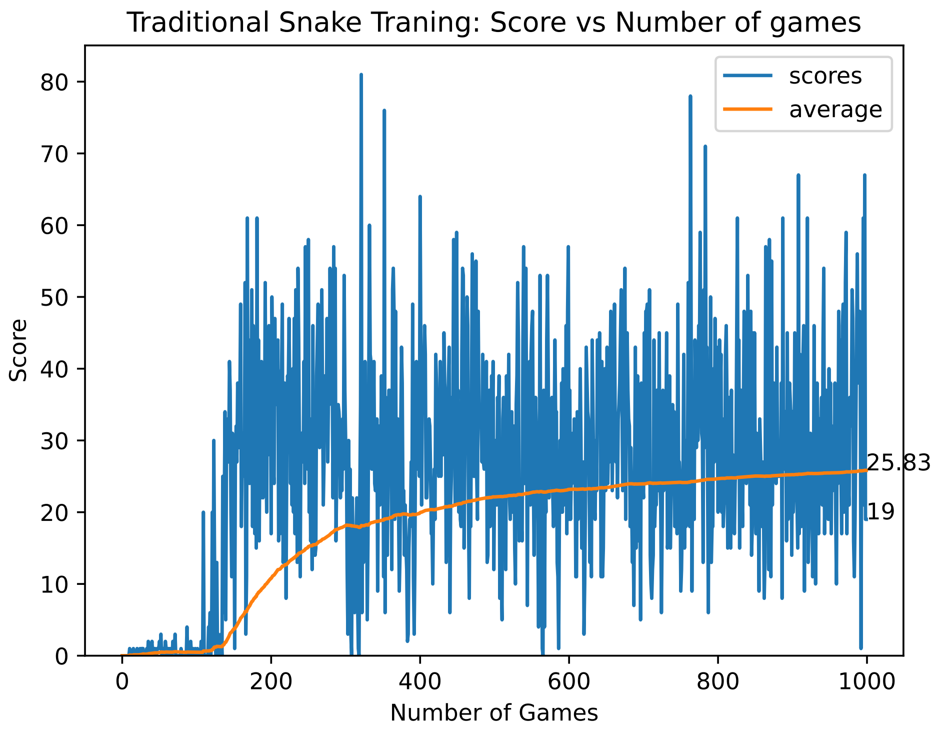


Figure 7: 3__poison__1000__games__complex__25.83__mean.svg

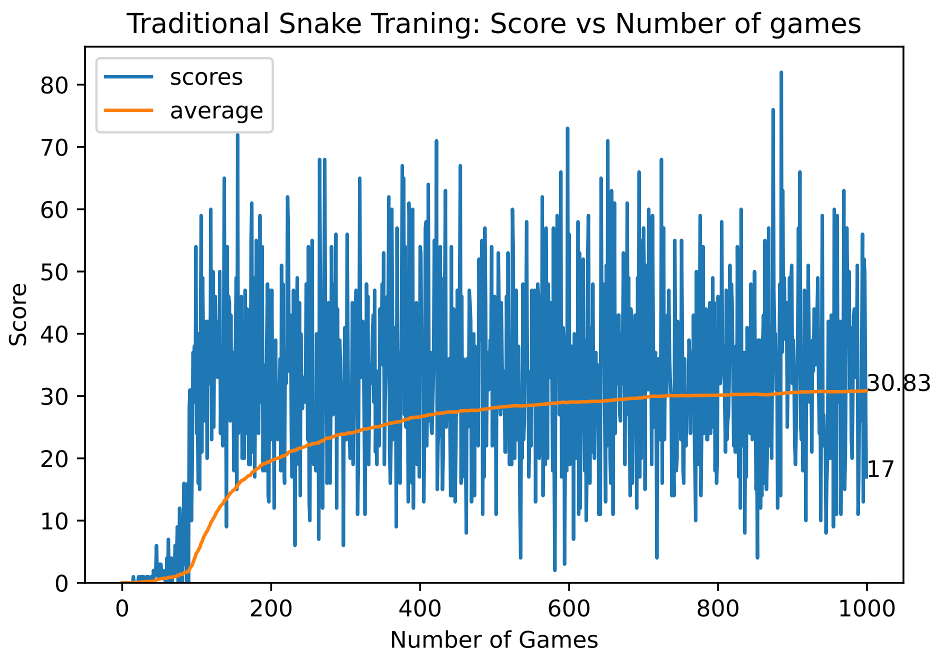


Figure 8: 1000__games__basic__30.83__mean.svg

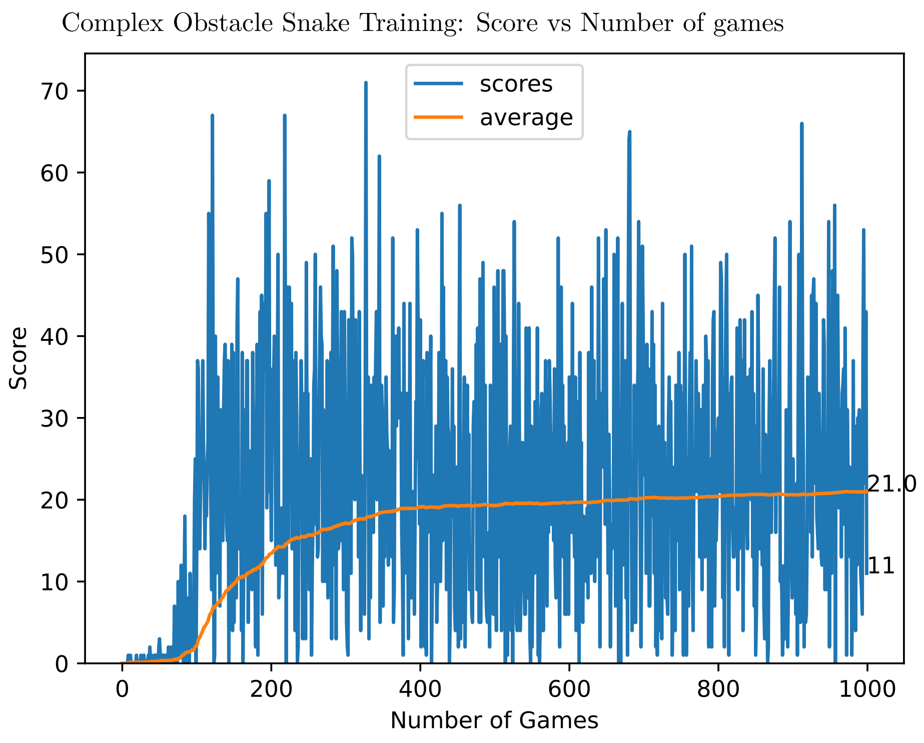


Figure 9: 1000__games__complex__1s__1m__21.0__mean.svg

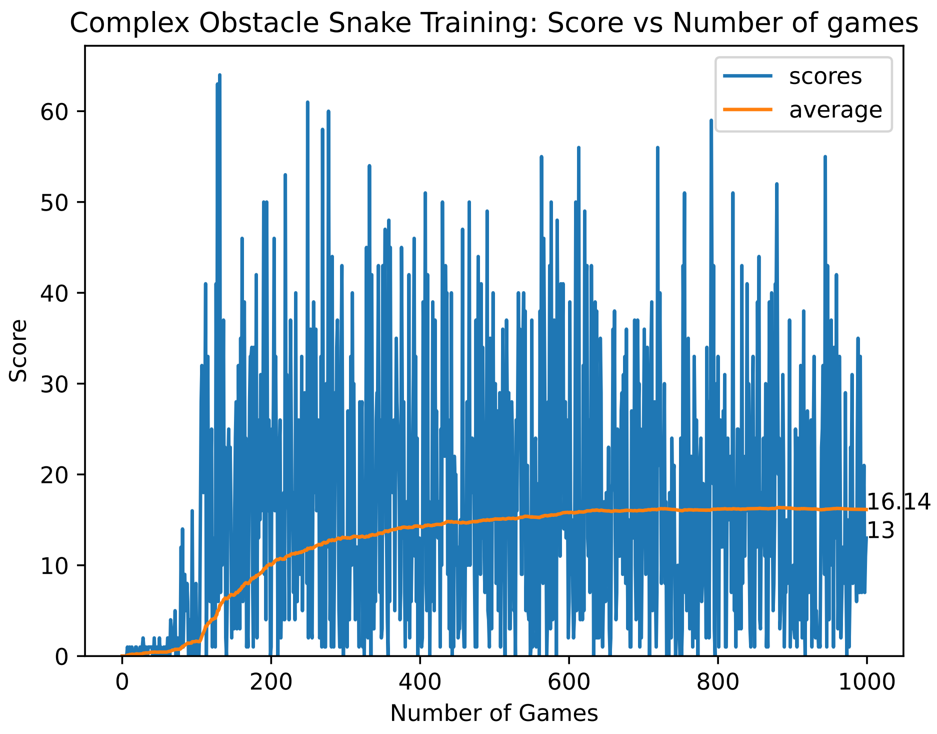


Figure 10: 1000__games__complex__2s__2m__16.14__mean.svg

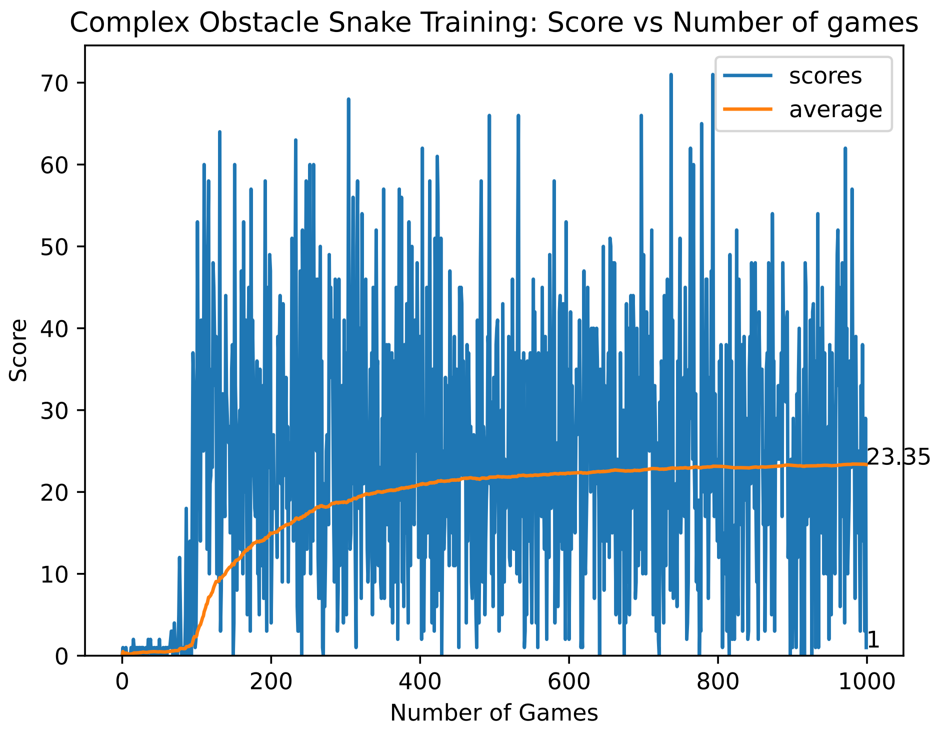


Figure 11: 1000__games__complex__0s__1m__23.35__mean.svg

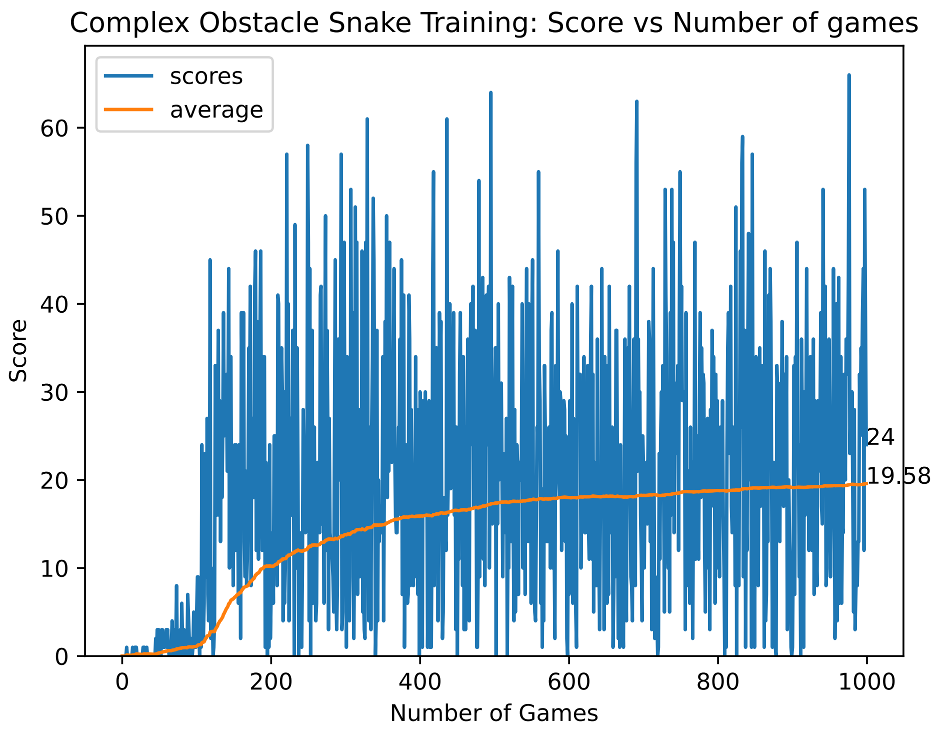


Figure 12: 1000__games__complex__3s__1m__19.58__mean.svg

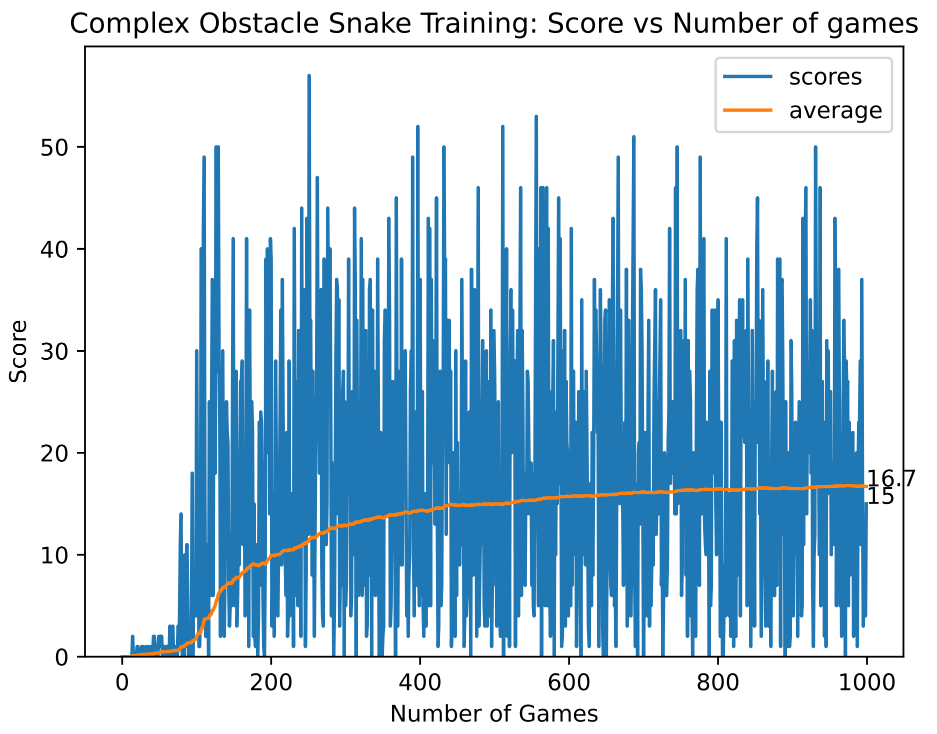


Figure 13: 1000__games__complex__0s__2m__16.7__mean.svg

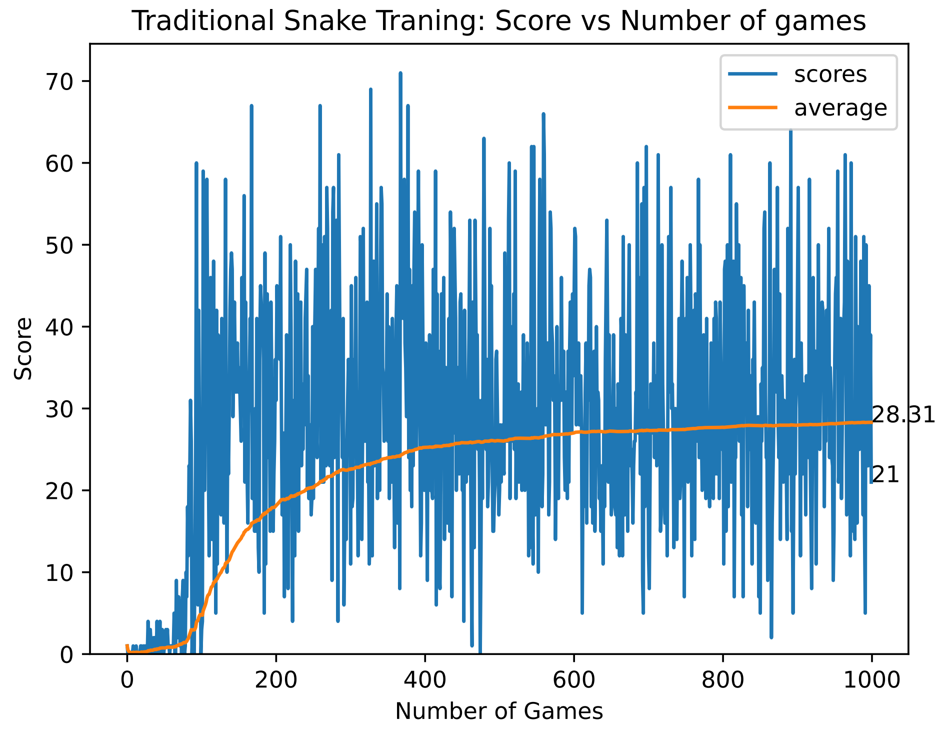


Figure 14: 1__poison__1000__games__complex__28.31__mean.svg

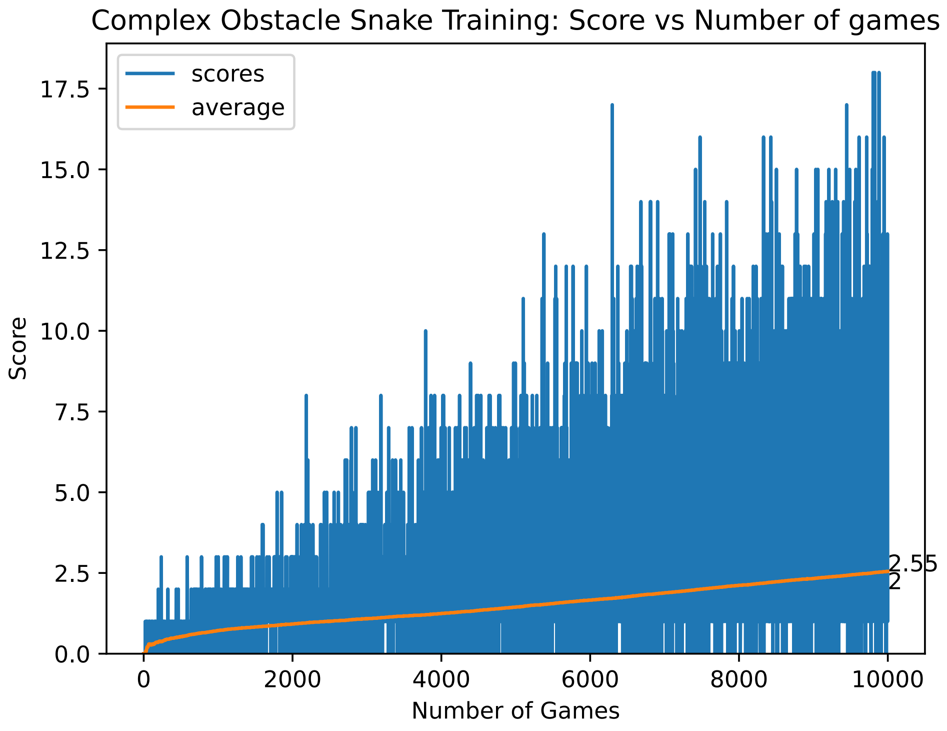


Figure 15: 10000__games__conv__mean__2.55.svg

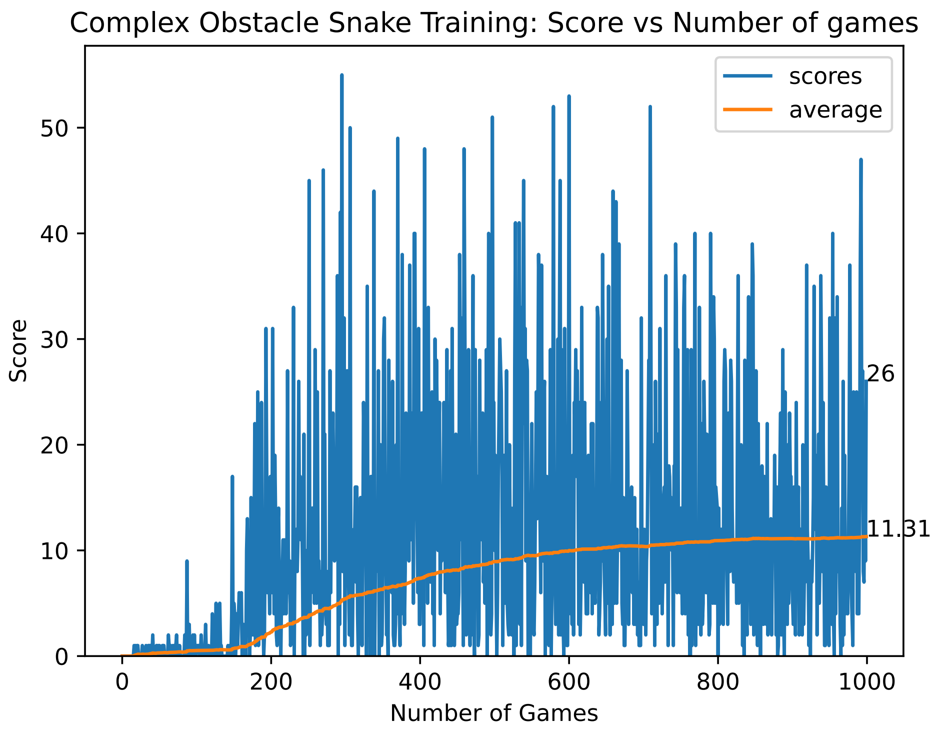


Figure 16: 1000__games__complex__0s__3m__11.31__mean.svg

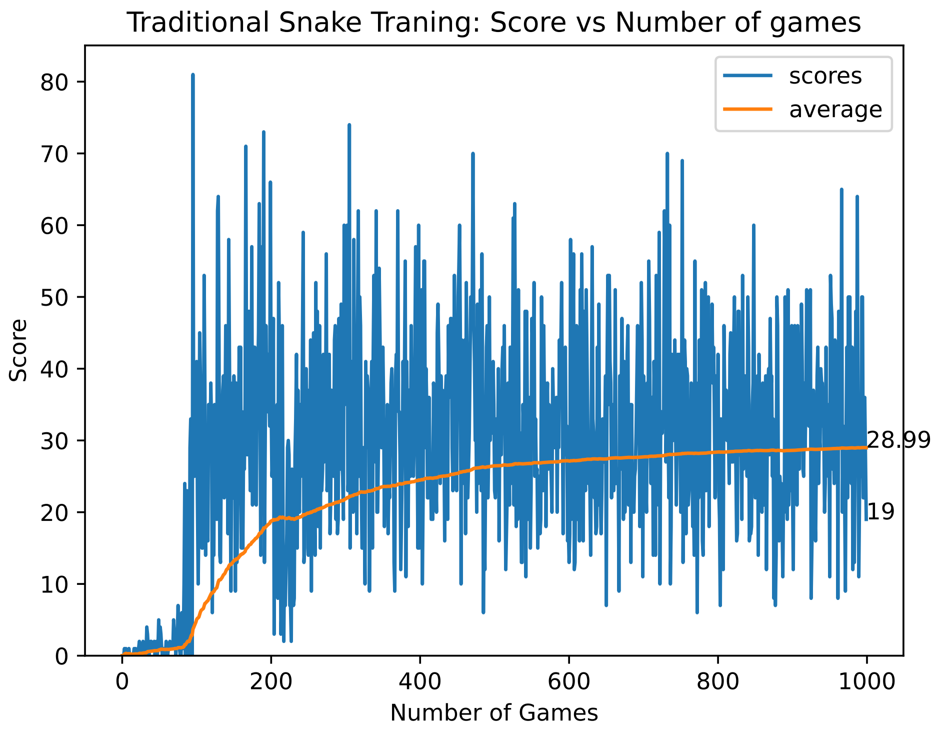


Figure 17: 2__poison__1000__games__complex__28.99__mean.svg

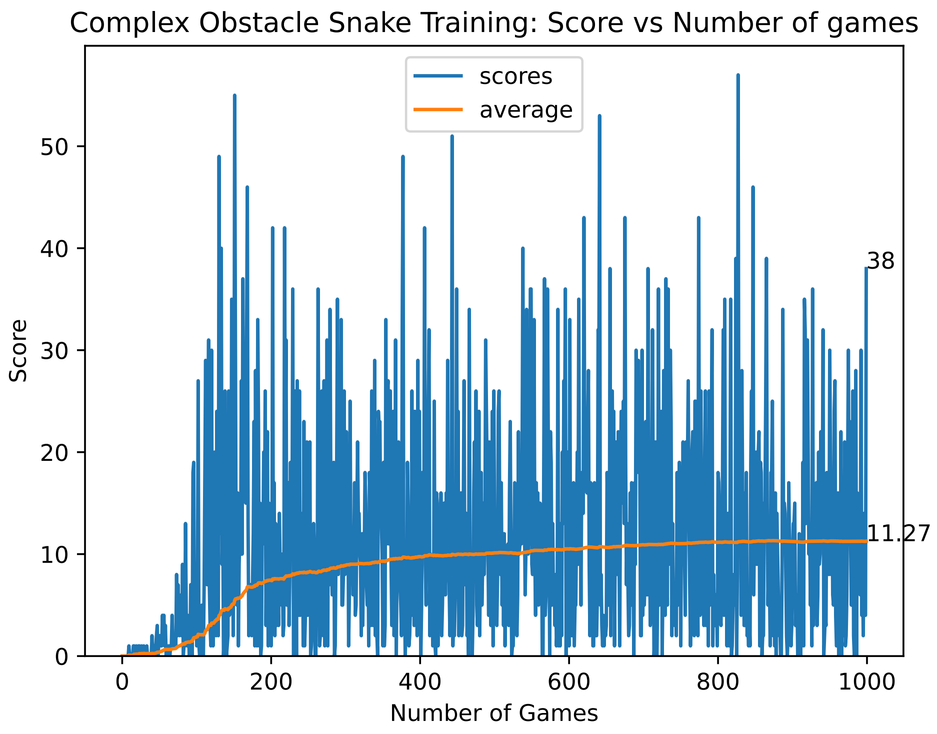


Figure 18: 1000_games_complex_1s_3m_11.27_mean.svg

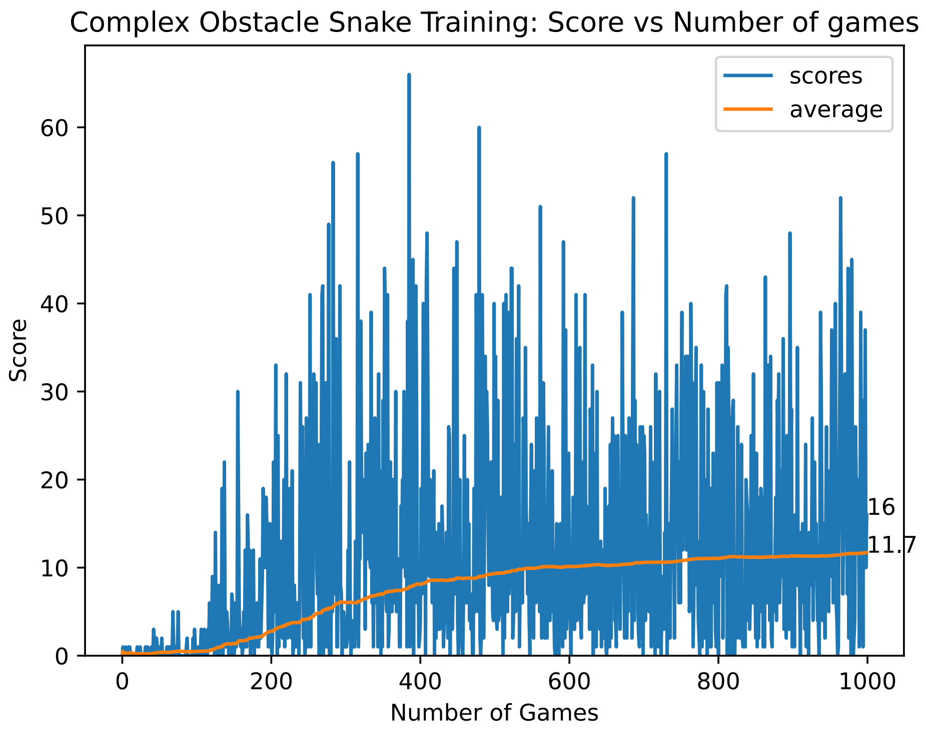


Figure 19: 1000_games_complex_3s_3m_11.7_mean.svg

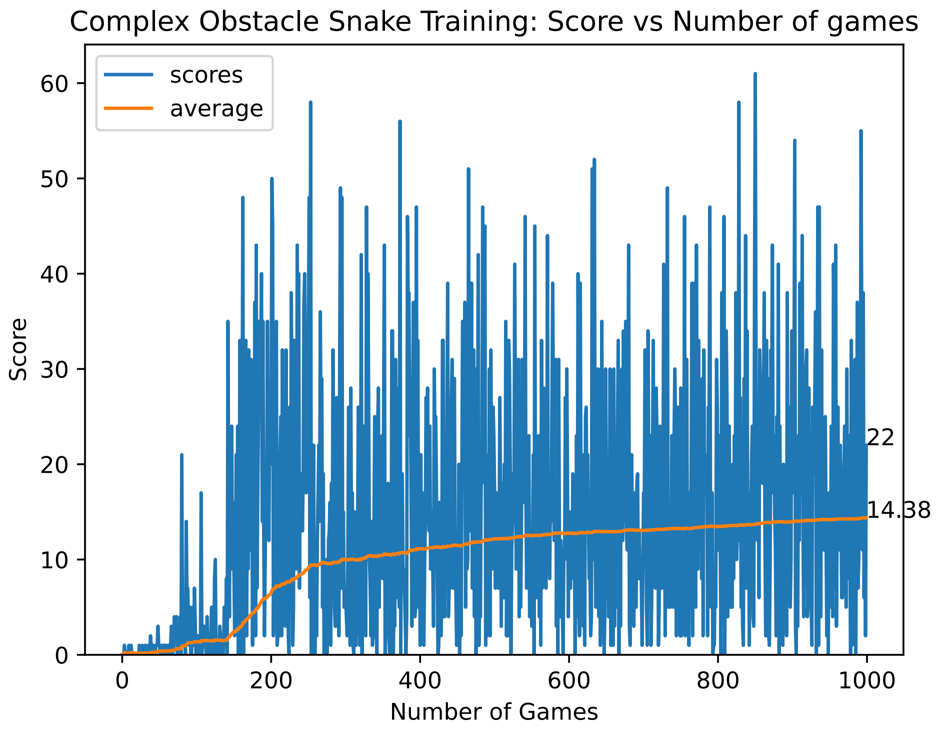


Figure 20: 1000__games__complex__3s__2m__14.38__mean.svg

Testing Figures

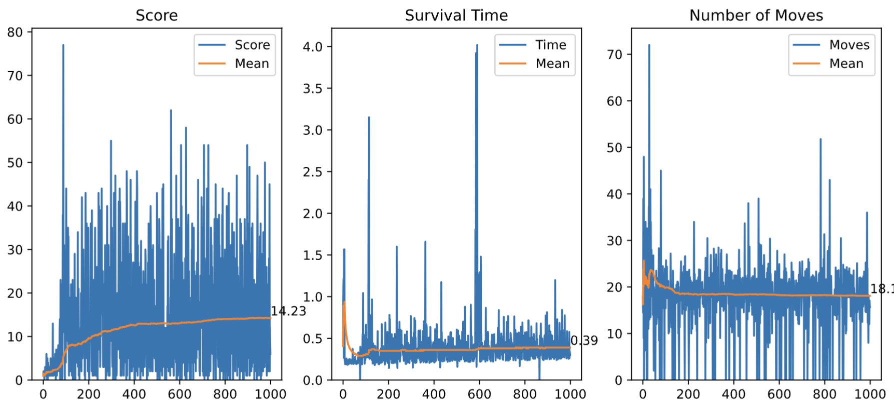


Figure 21: basic__on__complex.png

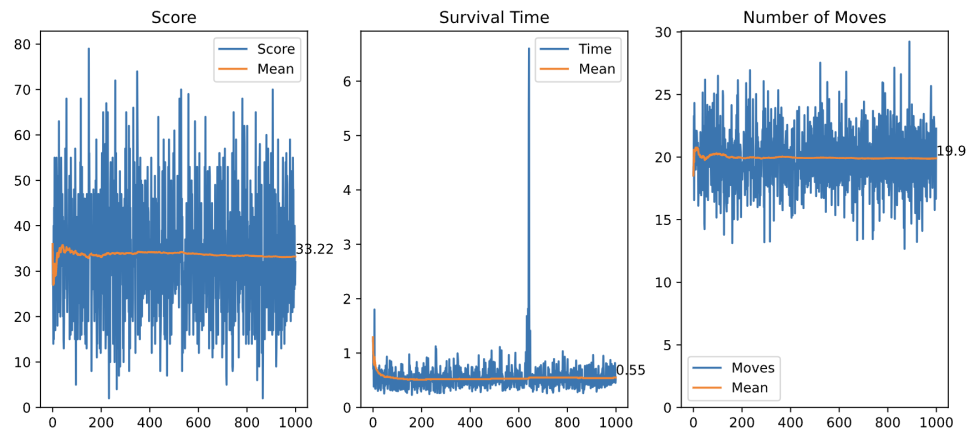


Figure 22: basic_on_basic.png

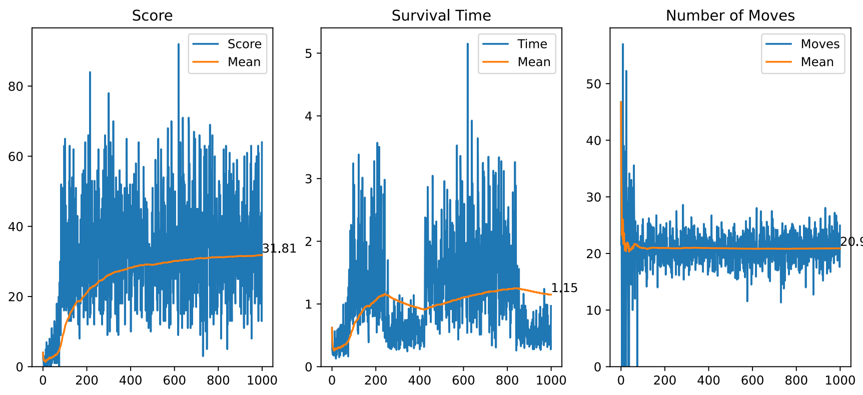


Figure 23: Complex_3s_3m_on_basic.svg

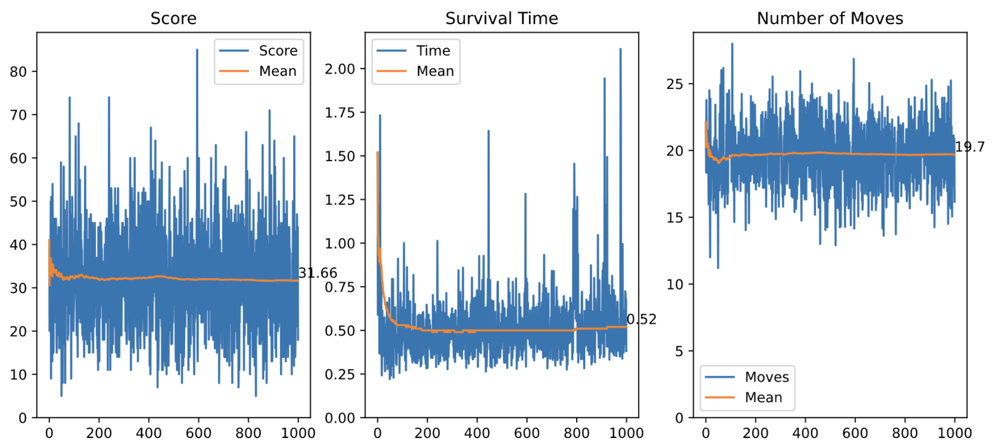


Figure 24: Complex_1s_1m_on_basic.png

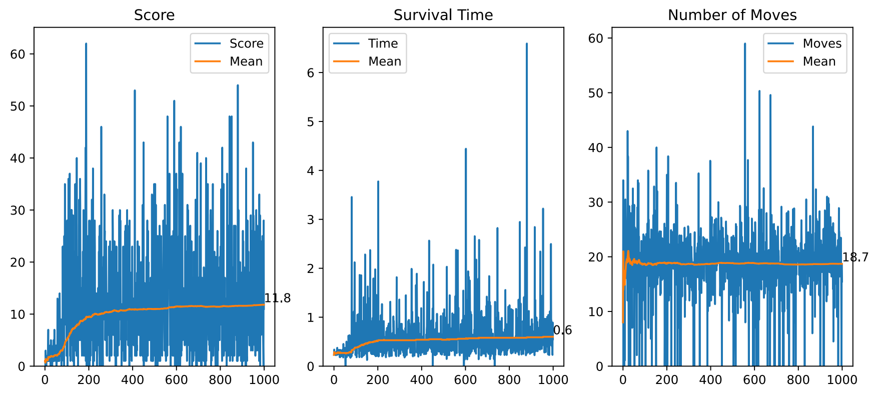


Figure 25: combined_plot_complex_2s_2m.svg

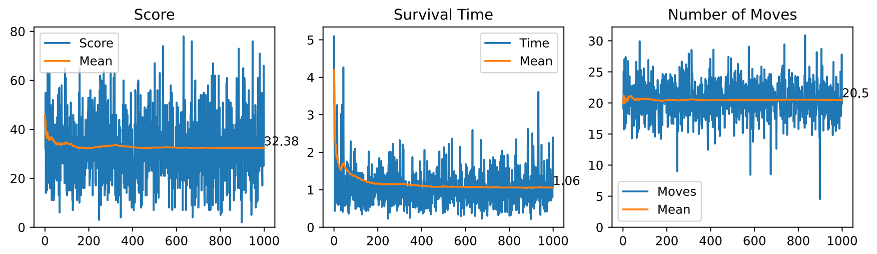


Figure 26: combined_plot_simple_2s_2m.svg

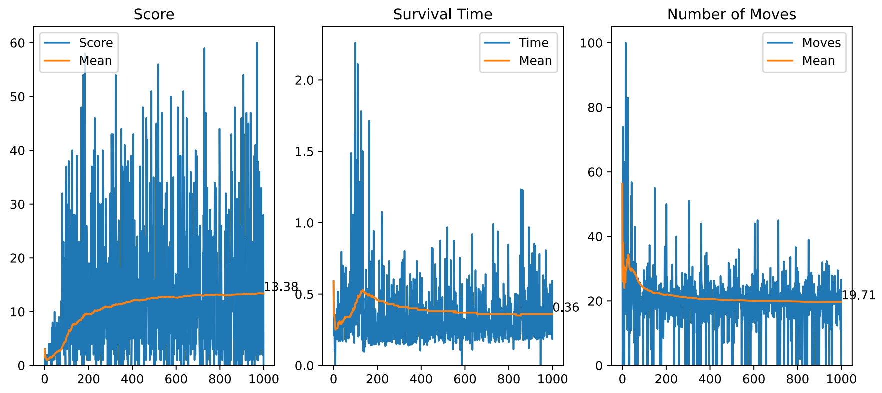


Figure 27: Complex_3s_3m_on_complex.svg

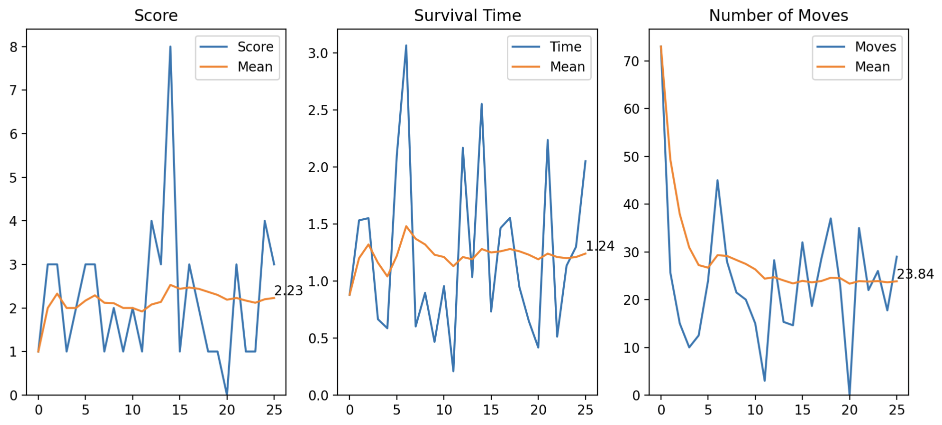


Figure 28: Complex_1s_1m_on_complex.png