Defeating Frogger with AI

Omer Ferster, Tomer Meidan, Liel Amar

Introduction

Our project aims to create an AI agent that navigates a finite world filled with dynamic and lethal obstacles to reach a finish line. We drew inspiration from the Frogger game, in which players guide five frogs from the start to the finish line while dodging traffic on roads and crossing a river by jumping on logs.

For simplicity, our variation of the game includes a single frog in a 16x16 map, which needs to get to the finish line at the top of the screen. There are three levels of difficulty, with each adding more obstacles: **Easy** contains cars moving horizontally at a constant speed, **Medium** adds a train obstacle and **Hard** adds a river section the player needs to cross by jumping on wood logs.

The player can perform four actions: move up, down, right, and left. To win the game, the player must reach the finish line in less than 50 steps without colliding with any obstacles.

We chose this specific problem for several reasons:

- We find this problem to be quite an interesting one. The idea of an agent that explores the world it is in fascinates us. It's a topic we barely covered in the course, so we wanted to delve deeper.
- The problem is a game that we've all played and enjoyed. It seemed like a suitable game to test with various algorithms because of the small number of actions and the complexity of the environment.
- It's a unique problem that will force us to use algorithms we did not practice during the semester. It's an opportunity to learn how to use these methods theoretically and practically by doing our own research.

Our game updates in real time, with more than 3^{256} possible states. Potential paths to the finish line change constantly, and the agent doesn't necessarily have full information about future states. For these reasons, we expect trivial solutions to not yield good results.

The solutions we propose are Reinforcement Learning and Genetic Algorithms. More specifically, we will use the DDQN[1] algorithm for Reinforcement Learning, which, unlike traditional Q-Learning, uses neural networks to predict the Q-Value for (state, action) pairs. These Q-Values are used to select an action based on maximizing the expected reward. This approach lets us manage large (state, action) spaces, such as in our game, which is unavailable through traditional Q-Learning. Additionally, DDQN mitigates the overestimation problem with standard DQN, which causes the model to be unstable. As for the second solution, we will use the NeuroEvolution of Augmented Topologies[2] algorithm. This genetic algorithm starts with a simple neural network, and through a defined fitness function and genetic evolution, it converges to a neural network that is optimal for the problem.

We used LLMs to understand better the two algorithms and how to integrate them within our game. They referenced papers we used to support our claims and helped with English translations.

Previous work

When we explored projects previously done in the course, we noticed that while Genetic Algorithms and Reinforcement Learning were used to solve problems, these problems differ from ours - an agent trying to get to a target, while avoiding obstacles.

In the academic community, however, we found several papers about solving Frogger-like games using Artificial Intelligence. We will discuss a couple of specific studies.

Firstly, in a study conducted at the University of Groningen[3], researchers attempted to solve a simpler version of Frogger using Q-Learning. Due to the relatively large state space, they used a Multi-Layer Perceptron to assist in evaluating Q-Values. The results of the algorithm were not perfect; it managed to cross the road to the central part of the

[1] "Deep Reinforcement Learning with Double Q-learning", Hado van Hasselt, Arthur Guez and David Silver, **Google DeepMind**, 2015
[2] "Evolving Neural Networks through Augmenting Topologies", Kenneth O. Stanley and Risto Miikkulainen, **University of Texas**, Ausin, 2002
[3] "Learning to play Frogger using q-Learning", Van der Velde, **University of Groningen**, 2018

map in a significant percentage of cases but was unable to win the game and reach the final goal.

Secondly, there's a study[4] in which the researchers attempted to solve a Frogger-like game using NEAT. In this study, the researchers designed their game similarly to our **Hard** difficulty, as described in the introduction. The researchers provided each agent with several peripheral sensors that supplied information about the tiles surrounding them. They used a fitness function that awarded high scores for reaching the goal. In terms of results, it was observed that by the 30th generation, there was at least one individual in the population achieving victory. However, the paper does not clarify how consistent the best model is, and no repeated experiment results were reported to assess the model's performance across different trials.

Both studies had key assumptions that guided their approaches and affected their findings.

- In both researches, the state-action space is limited, and in particular, the number of actions is small.
- Representing states as discrete instead of continuous and modeling the game as a grid, to limit the state space and allow using certain algorithms, such as Q-Learning.
- Basing the reward mechanism over granting higher rewards the closer the agent is to the target.
- Integrating various exploration strategies to help the agent avoid getting stuck in local minima. For example, in the paper on Q-Learning, the exploration strategy is based on an ϵ-greedy method. In contrast, in the NEAT algorithm, exploration is achieved through random "mutations" in the neural network architectures.

In conclusion, while previous course projects did not address Frogger-like problems, such attempts have been made in the academic community. Notably, these projects commonly assumed a large and complex state space, making neural networks essential. They also consistently modeled the game world as a grid to simplify state

[4] "Developing Frogger Player Intelligence Using NEAT and a Score Driven Fitness Function", Davis Ancona and Jake Weiner, **Swarthmore college**, 2014

representation. In terms of performance, the Q-Learning method yielded relatively poor results, whereas NEAT performed better.

Methodology

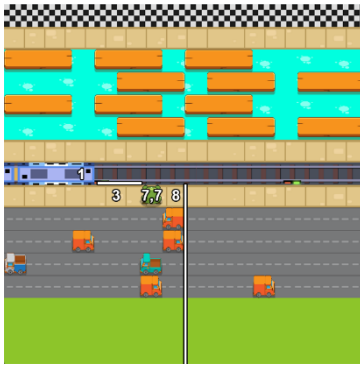
In this section, we will provide a general overview of the problem domain, explaining the game's structure, how the agent is represented and integrated into the game, and how the various obstacles behave. We will then discuss in detail the two methods we selected to solve the problem.

The game is structured as a 16x16 grid, with each cell measuring 40x40 pixels. The map is generated with respect to the difficulty. In Easy mode, there are two car segments with dynamic obstacles. In Medium a train is added between the two car segments. Finally, Hard changes the top car segment to a water segment, with moving logs. In all difficulties, there are predefined safe-zones – the grass and sidewalk segments, and the finish line. We don't use a predefined spawn location for cars and logs. Instead, they are spawned randomly within bounds.

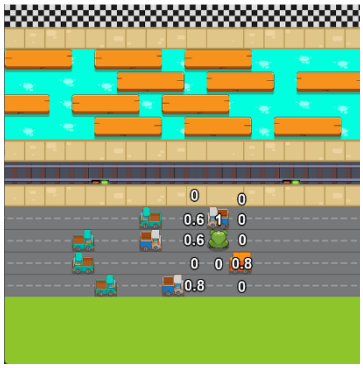
Our main assumptions are that every entity is aligned to the grid, takes up entire cells and moves exactly one grid cell with each frame. The player can take up to 50 steps a game. Colliding with cars/the train, running out of steps and drowning in the water cause the player to die. Reaching the finish line without dying secures victory.

State Representation

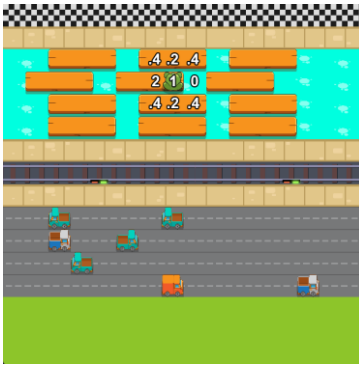
We're representing states similarly in both algorithms and rely on the assumption that our world is grid-like. The state consists of 24 base sensors which tell the proximity of the player to the nearest car from every direction – up, down, left, right, top left/right, bottom left/right, top-top left/right, bottom-bottom left/right, and the traffic direction. Then, we have a 25^{th} sensor which holds the number of steps taken. If the train obstacle is active, we have 5 additional sensors – is train active, it's y position, x distance from the player, and player x, y coordinates. Lastly, if the water section is active, we have 15 more sensors indicating direction and proximity – top, bottom, top left/right, bottom left/right, and whether the player is on a log, and how many steps are available to the left and to the right.



Train obstacle state representation



Road obstacle state representation



River obstacle state representation

NEAT Agent

We used the NEAT-Python library, which abstracts the entire NEAT learning process. We start off with a population of 400 individuals with simple and minimal networks. In addition, new species may arise if network architectures are distinct enough. During training, every individual in the generation is evaluated based on their performance. Performance is measured using a fitness function which evaluates the quality of an individual. As rounds progress, new generations are based on individuals that achieve higher fitness values.

Our fitness function is defined as follows. To avoid situations where certain individuals get a high score due to random chance, each individual plays three rounds, and the fitness value is the average of these scores. At the end of each round, the player is awarded i points for ending up on the $i'th$ segment – ranging from 1 to 15, the closer they are to the finish line. Additionally, to encourage diverse strategies that help individuals navigate complex situations, we grant a point for each unique action performed. Finally, victorious individuals receive extra points based on the remaining number of steps, normalized by 10, to encourage the individual to win as fast as possible. Formally, i – the individual index, j – the round index, $d_{i,j}$ – the reached segment index, $u_{i,j}$ - the number of unique actions, $k_{i,j}$ - the number of remaining steps, the fitness is defined as:

$$f_i = \frac{\left(\sum_{j=1}^{3} d_{i,j} + u_{i,j} + \frac{k_{i,j}}{10}\right)}{3}$$

After evaluation, the top two individuals from each species are preserved through elitism. Species with higher maximum fitness contribute more offspring, while the top 10% of individuals in each species are selected for reproduction. Offspring are created via cloning or crossover, followed by mutations. The network architecture can have at most three hidden layers, and mutations either add, remove or change connections, weights and biases, with random chance.

At the end of training, the selected model is the one who got the highest fitness throughout the whole process.

DDQN Agent

We used PyTorch to model the network architecture, which consisted of two hidden layers, each with 64 neurons. We used two identical neural networks – an online network which updates every frame and is used to compute the Q-Values of the current state and select the actions for the next move. We then use a second, offline, network, which is more stable – updating after every episode, to predict the next Q-Values. Lastly, we use the Q-Values and next Q-Values to compute the loss and backpropagate. We used a buffer which holds tuples of states, actions, rewards, next states and game overs, to generate batches to use to train the main network.

As deduced from the original paper about DDQN[5], the update rule of the model is as follows:

$$Q(s, a; \theta) = r + \gamma Q(s', argmax_a, Q(s', a'; \theta); \theta')$$

where θ is the main (online) network and θ' is the target (offline) network, γ is the discount factor and r is the reward as described below. This differs from the DQN algorithm, which solely uses θ' for both selection of actions and evaluation of Q-Values.

During training, we reward the agent using the following reward function: 100 + number of remaining steps points for winning, 1 point for moving up, -1 point for moving down, -50 points for dying, i points for reach the $i'th$ segment for the first time,

[5] "Deep Reinforcement Learning with Double Q-learning", Van Hasselt, Guez and Silver, page 2095

and if the train obstacle is active, $-j$ points where $1 \leq j \leq 3$ is the distance from either side of the screen.

We ran 10,000 episodes. In each frame, the agent selects an action either by using the main network, or randomly choosing through the epsilon-greedy strategy, and the main network updates using a random batch from memory. After every episode, the target network updates with the main networks' weights.

Results

We evaluated our trained models using three environments, one for each of the mentioned difficulty levels. For each experiment, we will describe the training process of the two models and subsequently evaluate their performance.

As part of the algorithm comparison, we will focus on the following metrics:

- **Win rate** – The percentage of times an agent successfully wins over many games. We will compare this metric against two additional baseline algorithms – a random agent, taking a random action at every step, and an "only-up" agent, which only takes "up" actions.
- **Solution efficiency** – The average number of steps remaining (out of 50) after each win of the model.
- **Convergence time** – The speed at which each algorithm converges to a particular strategy. Indicates the efficiency of the algorithm during training.

Before we jump into the results, it's important to mention that getting solid results required a lot of tuning of different hyper-parameters. When training NEAT, for example, we marked key parameters which have tremendous effect over how well the model improves. The following two Figures show the importance of configuration. The model in Figure 1 was trained with inadequate configuration and no convergence is seen in terms of fitness and achieved 40% win rate in the Easy difficulty. The second model from Figure 2, however, introduced a better configuration. It converged to a solid fitness value and achieved 98% win rate.

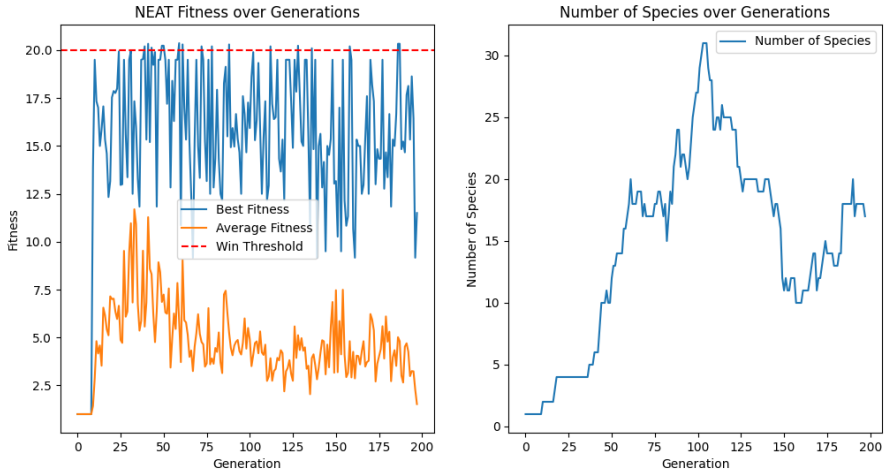


Figure 1: NEAT training with inadequate configuration – no convergence (pop. size of 40, high mutation rates)

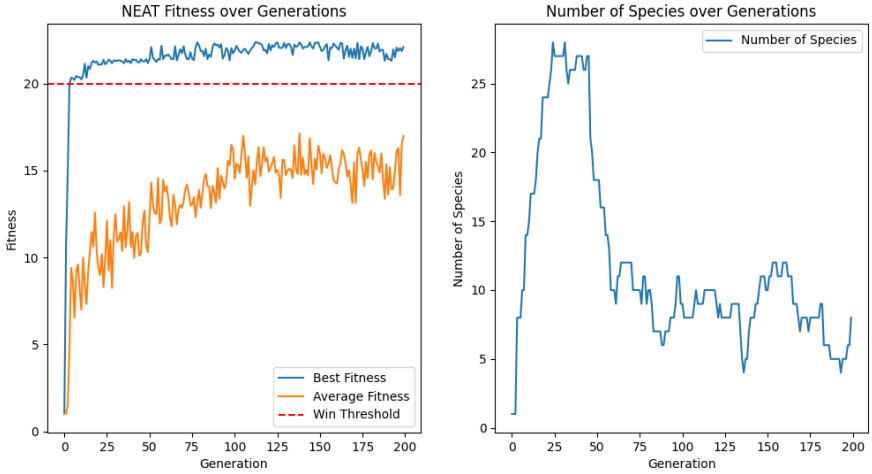


Figure 2: NEAT training with good configuration - convergence (pop. size of 400, medium-low mutation rates)

Experiment #1 – Easy Difficulty

The Easy difficulty of the game include two road segments, each with four lanes. Cars spawn on top of the lanes and go either left to right or right to left. All cars spawn with a random offset, forcing the agents to adapt and understand their environment, rather than memorize a certain pattern.

During the training of both models, we observed that they each converged to relatively good and stable results at different stages of the training process.

In the case of NEAT, as seen in Figure 3, the blue line shows that within less than 25 generations, the best genome from each generation onward beat the game 3/3 times and reached a fitness of more than 20 – the win threshold. We can infer that at least one species developed a winning strategy early in the process. Additionally, the orange line tells us that the more generations, the higher the population average fitness is. This aligns with how NEAT works – **a rapid improvement early on**, preserving good genes and trying to improve further through mutations.

On the other hand, from Figure 4, we see that the DDQN algorithm improved **gradually** throughout the training – only after more than 500 episodes did the win rate exceed 50%, and only after 1,000 episodes the model stabilized on a strategy that approximately achieves a win rate of 100%.

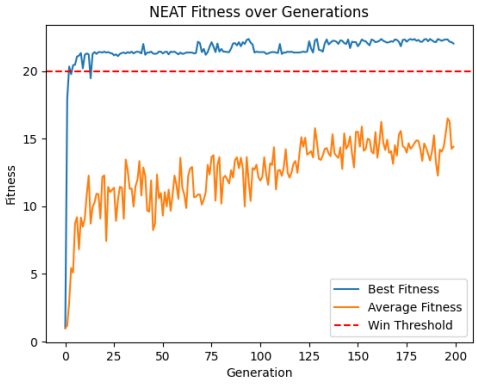


Figure 3: NEAT Training Fitness over Generations – Experiment #1

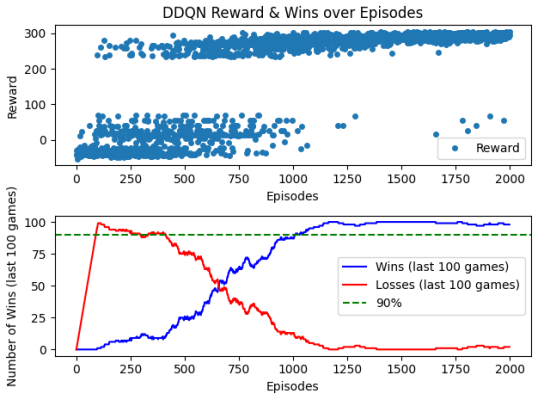


Figure 4: DDQN Training Rewards & Wins over Episodes – Experiment #1

In terms of performance, an examination of Figure 5 and the table below shows that both models achieve very good results – a win rate of over 97%. It is also evident that the random algorithm has almost no chance of winning the game, while the algorithm that only moves upward has a 46.64% chance of winning. These findings indicate that both NEAT and DDQN have developed strategies that are more sophisticated than the basic strategy of only moving upward, as they significantly outperform it in terms of win rate.

Regarding the comparison of algorithm performance, the DDQN algorithm has an advantage, achieving a 100% win rate with stability in the current game difficulty, while NEAT occasionally experiences losses despite its very high success rate. However, NEAT has better efficiency, having 32.3 steps remaining compared to DDQN's 29.1 steps remaining when winning.

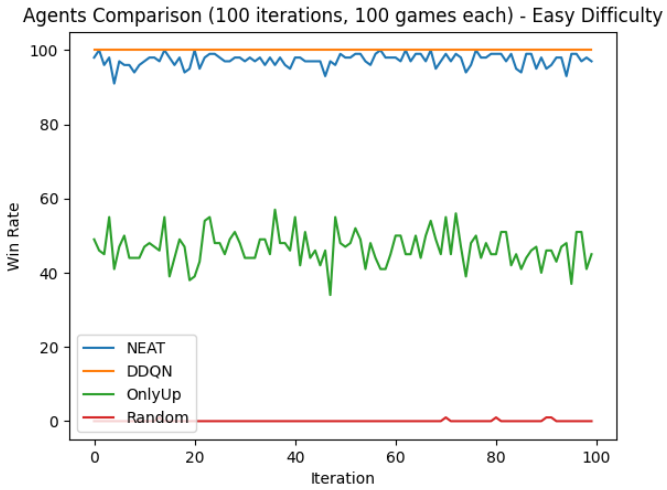


Figure 5: Comparing The NEAT, DDQN, OnlyUp and Random Agents win rate over 10,000 games, divided into 100 iterations – Easy Difficulty

Agent	**Win rate**	**Solution Efficiency** average of 10,0000 games victories only
NEAT	97.33%	32.3 remaining steps
DDQN	100.0%	29.1 remaining steps
OnlyUp	46.64%	35.0 remaining steps
Random	0.05%	0.3 remaining steps

Experiment #2 – Medium Difficulty

At this difficulty level, we add a railway at the center of the map. The train introduces an additional challenge in the shape of a dynamic obstacle in a new segment different than the road segment. Encountering the train is relatively rare compared to the cars, due to the train having only a 5% spawn chance.

Looking at Figure 6, we see that similarly to the previous experiment, the NEAT algorithm produced an individual with a winning strategy in less than 25 generations. However, we observe instability in the "best fitness" metric compared to the Easy task. Throughout the entire 400-generation training, there are cases where the best fitness did not exceed 20, meaning a win was not achieved. The frequency of these occurrences does not decrease over time, emphasizing that the model struggles to adapt and learn the train obstacle. Additionally, the average fitness values hover around 10, stop improving at some point, and are much more variable than the average fitness values observed in the easy task.

Contrarily, DDQN's behavior in this experiment is quite similar to the previous one as can be seen in Figure 7. The algorithm gradually improves, stabilizing at over a 90%-win rate only after about 4,000 episodes. This is indeed longer than the 1,000 episodes taken in the previous experiment, which aligns with the addition of a new and unpredictable obstacle.

Overall, looking at the training graphs, we infer that the addition of the railway introduced significant complexity to the task, as DDQN took longer to stabilize and introduced increased variance, and NEAT struggled to stabilize whatsoever.

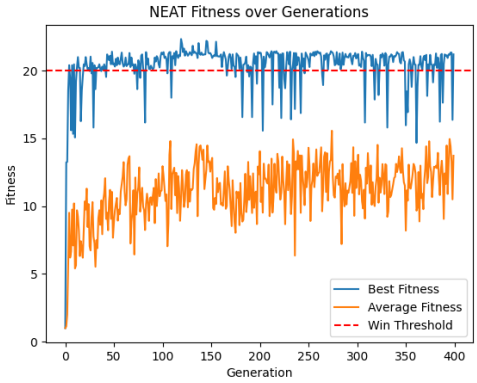


Figure 6: NEAT Training Fitness over Generations – Experiment #2

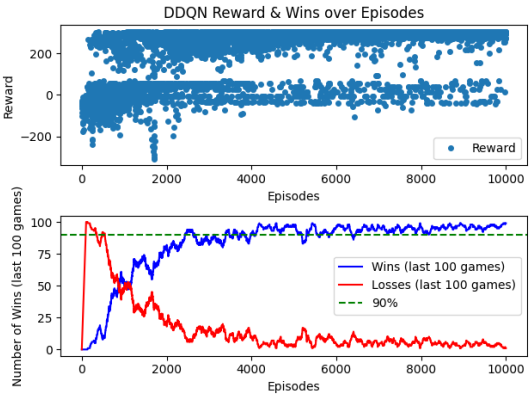


Figure 7: DDQN Training Rewards & Wins over Episodes – Experiment #2

The results we got after testing all models for 10,000 games show that DDQN achieved a win rate of 99.38%. While the win rate is very close to the results from the first experiment, the model is slightly less stable and does not win every single game. This aligns with the higher variance noticed in the win rate during training, compared to the low variance observed previously. In contrast, NEAT performed much worse on this task, achieving a ~71% win rate. This is a huge decrease compared to the previous task. Finally, the "only-up" baseline model did not deteriorate significantly, once again achieving around 46% win rate.

Unlike the first experiment, this time we see that DDQN is winning in a more efficient way. On average, DDQN required less than 19 steps to win, whereas NEAT needed 21.3.

We conclude that even though this task is slightly harder, both models managed to outperform the baseline models once more. DDQN performed better than NEAT in this task, and it's clear that the added complexity introduced by the railway negatively impacted NEAT. We infer that the complex environment and the frequency of the train's arrival made it difficult for NEAT to develop individuals that both achieve high fitness and deal with the train effectively.

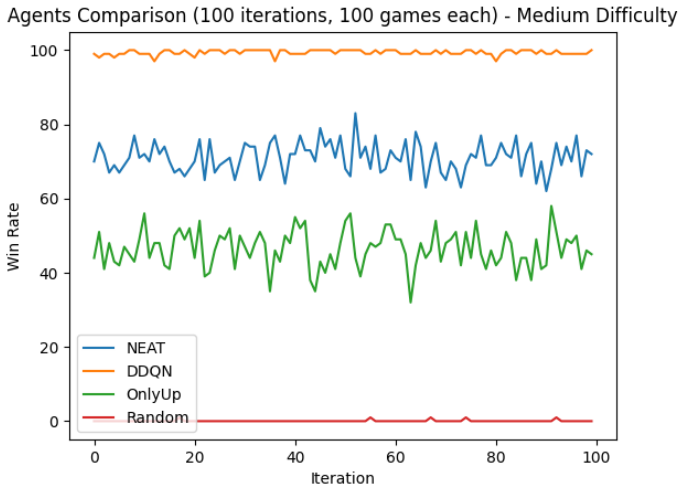


Figure 8: Comparing The NEAT, DDQN, OnlyUp and Random Agents win rate over 10,000 games, divided into 100 iterations – Medium Difficulty

Agent	**Win rate**	**Solution Efficiency** average of 10,0000 games victories only
NEAT	71.04%	28.7 remaining steps
DDQN	99.38%	31.3 remaining steps
OnlyUp	46.34%	35.0 remaining steps
Random	0.21%	0.2 remaining steps

Experiment #3 – Hard Difficulty

In this experiment, we removed a segment of the map that contained a road and replaced it with a segment featuring water, where the player must navigate across logs that move horizontally. The logs are of fixed size and move at a constant speed. We added this segment to increase the complexity of the game and to test how the models perform in an environment that requires two different maneuvering strategies.

From Figure 9, it can be seen that during the NEAT training phase, it struggled significantly with convergence. Similar to previous experiments, there was a rapid initial increase in best and average fitness values, which then became more gradual. Even after 400 generations, NEAT did not manage to converge to the high fitness values, and in most generations, it failed to produce an individual that could win all 3 consecutive rounds. Consequently, the average fitness value in each generation hovered around 7.5, which is lower than the average value in previous trainings.

Observing Figure 10 shows that DDQN exhibited a training process like those observed in the previous two experiments. There was a gradual improvement, and after more than 4,000 episodes, the model reached a point where its win rate was over 50%. After more than 6,000 episodes, the model stabilized with a win rate above 90%. It is worth noting that in the second task, DDQN achieved this win rate after only ~4,000 episodes.

Taking DDQN's longer training time and NEAT's lack of convergence into consideration, suggests that the addition of the water segment introduced a significant increase in difficulty and complexity.

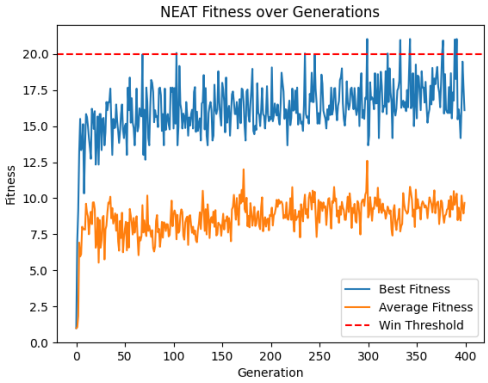


Figure 9: NEAT Training Fitness over Generations – Experiment #3

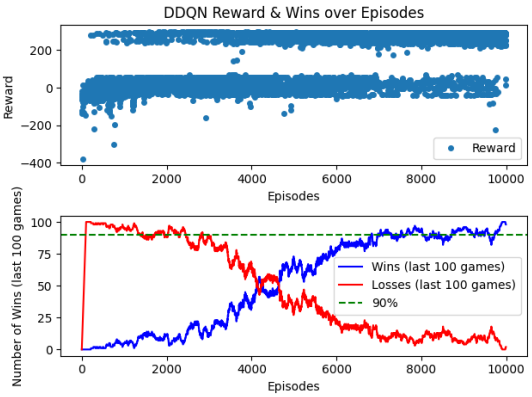


Figure 10: DDQN Training Rewards & Wins over Episodes – Experiment #2

The results from running the models on 10,000 games show again that this task was significantly more complex compared to the previous two tasks — both baseline algorithms lost every single game, whereas in the previous tasks, the "only up" algorithm had a win rate of about 50%. This demonstrates that basic strategies that were partially effective in simpler tasks failed completely at this level of complexity. Similarly, NEAT showed poor results with a win rate of ~10%. Lastly, the DDQN algorithm continued to perform well, with an average win rate of 93%. This suggests that despite the significant increase in task complexity, which led to a decline in performance for NEAT and the baseline "only up" algorithm, DDQN managed to learn the game quite impressively.

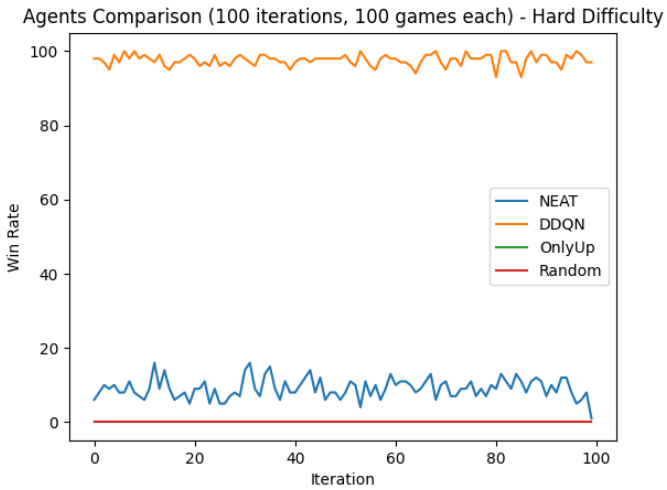


Figure 11: Comparing The NEAT, DDQN, OnlyUp and Random Agents win rate over 10,000 games, divided into 100 iterations – Hard Difficulty

Agent	**Win rate**	**Solution Efficiency** average of 10,0000 games victories only
NEAT	9.09%	29.6 remaining steps
DDQN	97.6%	25.8 remaining steps
OnlyUp	0%	0.0 remaining steps
Random	0%	0.0 remaining steps

Discussion

From examining the results of all models in the three experiments we conducted, it can be concluded that the DDQN models perform better than the NEAT models on our Frogger-like game. Even in the first and simplest experiment, it was noticeable that DDQN converged to a higher win rate than NEAT, although both provided good results, achieving a win rate of over 90%. However, as we increased the task complexity, we observed that NEAT's performance deteriorated drastically, while in the most complex task, DDQN maintained particularly high-quality results with an average win rate of 93%. Additionally, as we have seen, in this task, both models, after training, provided strategies that achieved better results than basic and trivial gameplay strategies.

Regarding the efficiency of the strategies developed by the models, we found that in the easy and hard difficulties, NEAT was able to complete the task, on average, in fewer steps than DDQN. However, in the medium task, the opposite occurred. This leads us to believe we cannot conclude that one algorithm outperforms the other in terms of solution efficiency.

In terms of convergence speed, we observed that in the simple task, NEAT converged much faster than DDQN, which did so gradually and slowly. This may be because NEAT can learn a simple task quickly and in parallel on a distributed multi-core system[6], whereas DDQN is sequential. On the other hand, in the subsequent tasks, NEAT failed to converge, while DDQN continued its gradual and slow convergence trend.

[6] "Deep Neuroevolution: Genetic Algorithms are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning", Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, Jeff Clune, **Uber AI Labs**, page 8

Summary

In conclusion, this paper presents how we modeled a problem centered on navigating obstacles and reaching a goal by developing a Frogger-like game. In our game, we implemented three environments with varying difficulties, each introducing its own challenges. At first, the player must reach the finish line while avoiding collisions with cars; secondly, we added a train that presents a rare but significant challenge; and lastly, we introduced a new water segment, requiring the player to develop an additional navigation strategy.

Our main assumption was that the problem would be modeled in a grid-based world, where each entity fits precisely within a specific number of grid cells and can move only according to those cells. Based on this assumption, we developed a state representation that relies on grid data. We attempted to solve the game using two methods: the NEAT genetic algorithm, and the RL algorithm - DDQN. Both methods rely on neural networks due to the relatively large state space inherent in this problem. Still, they each formulate strategies for the challenges in different ways: NEAT starts with simple neural networks and alters weights and connections between neurons to converge to a good solution, while DDQN directly learns a policy by approximating the Q-Value function.

Considering the results of the three experiments, we conclude that DDQN outperforms NEAT in solving our game. For the easy task, both models performed successfully, achieving a 95% win rate. However, starting from the second task, NEAT struggled and failed to converge during training, and in the third task, it even displayed poor results with a mere 8% win rate. In contrast, the DDQN method consistently maintained good results with a win rate of over 90%. Additionally, we identified that both algorithms, especially DDQN, learn the tasks relatively effectively, as both achieved better results than two trivial game strategies implemented by baseline algorithms. Notably, in the easier tasks, NEAT converged faster than DDQN, which had a slower and more graduate convergence, especially in the more challenging tasks.

These results were obtained because while NEAT achieves good performance in relatively simple control tasks, as the state space grows, the training time required for

NEAT to develop a strategy increases significantly[7] compared to algorithms from the RL domain. Therefore, it is possible that the increase in complexity in the second and third tasks led to NEAT requiring more training time and possibly a better configuration than we provided, resulting in the observed decline in performance. In contrast, the DDQN algorithm is more robust[8] to increases in problem complexity and thus continued to produce high-quality models but required longer training sessions.

There are three points where we critique our work:

- The models were built, and their results were tested under the assumption that the state space is discrete and finite. Therefore, we cannot generalize our findings to similar games that do not meet these assumptions.
- Another algorithm that could be compared in this problem is PPO, but its performance was not examined in this project. It is possible that in terms of performance, PPO could compete with DDQN, especially in games with a continuous state space[9].
- It is possible that DDQN is better suited for solving the problem we chose than NEAT, as it is particularly effective in complex problems with a discrete state space and a simple reward function[10]. If we had experimented with a deceptive reward function (which may lead to many sub-optimal solutions), we might have observed better results from NEAT due to its ability to explore large spaces effectively.

[7] "Comparing Evolutionary and Temporal Difference Methods in a Reinforcement Learning Domain", Matthew E. Taylor, Shimon Whiteson, Peter Stone, **University of Texas,** page 1
[8] "Deep Reinforcement Learning with Double Q-learning", Hado van Hasselt, Arthur Guez, and David Silver, **Google DeepMind**, page 1
[9] "Proximal Policy Optimization Algorithms", John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov, **OpenAI**, page 1
[10] "Deep Neuroevolution: Genetic Algorithms are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning", Petroski Such, Madhavan, Conti, Lehman, O. Stanley, Clune, **Uber AI Labs**, pages 7-8