Deep Reinforcement Learning for Navigation Optimization

Michi Jewett
mbjewett318@g.ucla.edu

Abstract

In this paper, we present a deep reinforcement learning (DRL) approach to optimize the navigation of an autonomous agent in single-map scenarios. We leverage a deep Q-network (DQN) to predict optimal actions based on the observed state of the environment. While our experiments demonstrate success in optimizing navigation on single-map scenarios, achieving generalization for multiple maps remains challenging due to the high dimensionality of the state space and the complex tuning of hyperparameters involved in reinforcement learning. Our results indicate that DRL can effectively learn optimal paths in specific environments, highlighting its potential for application in known environments where human intuition may not yield the best path.

1. Introduction

1.1. Assumptions

The Agent does not have prior knowledge of the maps, but has a proximity sensor to know its euclidean distance from the goal point. The Agent also has a vision system, so it can see obstacles up to two spaces away and estimate their distance. Thus, the state and rewards are not based on the set locations of the agent, goal, and obstacles, but rather the observed distances between them. This is a harder problem to solve, but we wanted this experiment to actually have meaningful, applicable results.

1.2. Neural Network Models

The output of our models is the Q-Values of the action space for which the Agent will take the action with the highest value. The action space size in this case will always be 9, as the Agent can choose to move in any of the 8 directions or stay still. The input state space size varies depending on the number of obstacles in the frame, as the state is a vector of the distances between the agent and the goal, as well as all the obstacles. We thought using the (x, y) locations or using the entire map's image was too easy and not practical if we wanted any hope of generalizing or applying this to a robot path planning system.

1.2.1 Simple Q Network

The Simple Network we designed is a three-layer, fully-connected linear network, with ReLU-activated layers. The optimizer is Adpative Moments (Adam). This is a shallow and simple network, but we want to demonstrate that Reinforcement Learning can be strong in simple cases even just by tuning the RL parameters such as the reward function, without creating a very complex model. The full architecture can be found in the Appendix.

1.2.2 Deep Q Network

This deeper model is made up of two Recurrent Long Short-Term Memory (LSTM) layers with three Fully-Connected Linear layers in between them. We used a combination of ReLU and $tanh(x)$ activation functions, along with selective Dropout and Layer Normalization, to allow the Agent to learn more complex patterns. The optimizer remained Adpative Moments (Adam). A full summary of the model can be found in the Appendix.

1.3. Reinforcement Learning Hyperparameters

Our initial Agent is using a replay buffer, but further down we will experiment with a prioritized replay buffer.

We are using a learning rate (LR) of 5e-4, which dictates the learning behaviour of our Neural Network's Adaptive Moments (Adam) optimizer. We mainly used a discount factor (GAMMA) of 0.99, which determines the importance given to future rewards, but experimented with 0.995 and 0.999 when training with hundreds of thousands of episodes. Increasing the discount factor generally leads to more long-term planning, as later rewards still matter more relative to the earlier rewards.

We are beginning with an exploration rate (EPSILON) that starts at 1.0 and decays by a factor of 0.995 every episode, with a minimum of 0.01. These parameters will be tuned throughout the experiments. For example, if we are running more episodes, we want our exploration rate to decay slower, and so we may increase the constant to 0.99995, for example. This represents the balance of exploration and exploitation.

1.4. Reward Function

The average reward for 100 episodes generally begins around -100, as the agent collides with an obstacle every episode, and it increases gradually toward 100, as this is the reward given for reaching the goal (and eventually it gets to the goal every episode.)

The fine-tuning of the reward function is based around two more factors. Every timestep, the agent is penalized by a small amount (1) because we want to encourage it to spend as little time as possible on its way to the goal. The agent is also rewarded each timestep based on its euclidean distance from the goal, meaning the closer it is to the goal, the more it is rewarded. MAX_DISTANCE - euclidean distance(agent, goal). In practice though it is scaled down by a constant factor so that it is always less than 1 (the penalty for waiting another timestep.) This is so that it is never beneficial for the agent to get close to the goal and just wait there until it reaches the maximum number of allowed timesteps and the episode ends and then go to the goal at the last timestep (as it would think it was best to rack up as much reward as possible before finishing the mission.)

Our stopping condition is when the reward exceeds an average of 99 or above across 100 episodes. This means that almost every episode, the agent is quickly and efficiently getting to the goal, with some wiggle room for the slight penalty it faces each timestep, balanced with the proximity reward.

2. Results

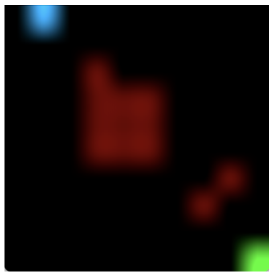
Our experiments begin with training the agent on a single, fixed map to evaluate the basic capabilities of our DRL approach. When attempting to generalize our agent across multiple randomly generated maps, despite using many different techniques, the performance was not satisfactory, indicating the challenges in achieving DRL generalization. To further push the limits of our DRL approach in single-map scenarios, we constructed a complex maze, which agent successfully learned the shortest path through.

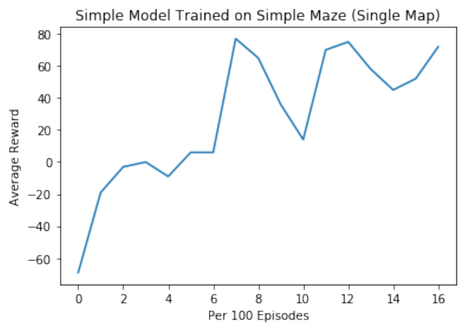
2.1. Single Simple 10x10 Map with Simple NN

First, to demonstrate that our reinforcement learning works, we will initialized one single map with set obstacles. This will overfit the policy to this map, meaning this model would not do well on another layout. From Figure 1, and the Appendix that the agent steadily gained reward and was trained after 1766 episodes. This demonstrates that even with a simple, shallow model, we can learn basic maps very well.

2.2. 100 Random 10x10 Maps with Simple NN

Now we will try to generalize this Agent by training it on a pool of 100 randomly generated maps that are 15% covered by obstacles, for 20,000 episodes. The goal and starting place for the Agent will also be random. Our intention here is to train on World 1-1, World 3-4, World 4-2, and every other level, to hopefully get an Agent that could see any Mario level and play it well. Based on the results in the Appendix (Figure 3), it seems that after about 7500 episodes, the model reached a sort of steady-state of failure. The positive information to take away from this trial is that the Agent got much better at avoiding the obstacles that it saw and knew to avoid them. However, the negative is that the policy did not really ever have enough chances to find the goal on a 10x10 given our timestep limit of 50 actions, and therefore did not get the reinforcement reward for finding the goal. It instead just moved around for 50 timesteps avoiding obstacles.


(a) Simple Map



(b) Simple Model/Simple Map Episodes vs. Reward

Figure 1. Simple NN trained on a Single Simple Map

2.3. 100 8x8 Maps with Increased Exploration

Next, we will try a smaller map with only 10% obstacles coverage for 30,000 episodes. We allow 1000 actions before cutting off the episode and use an epsilon decay of 0.999995 to encourage a lot of exploration. We hypothesized that these adjustments would reduce the chance of getting trapped in a steady state of failure. However, the exploration was too frequent and the model failed to converge, as shown in the Appendix (Figure 4).

2.4. Single Simple 10x10 Map with Deep NN

Based on the previous trials, it seems that the Simple Q Network is no longer able to learn the more complex maps;

thus, we moved to our Deep Q Network at this point. We first ran it on the baseline case. The results in the Appendix(Figure 5), demonstrate that it did not perform as well as the simple model. For one thing, we designed this model with a goal being better at generalization but are attempting the single map case. This means our dropout, for example, is actually hurting us since we want to ”overfit” to this map in this case. Also, since the model is deeper, we would need more trials potentially to fully learn it.

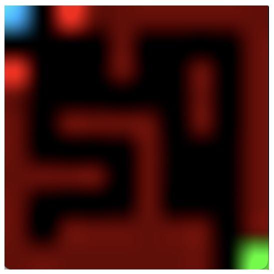
2.5. 5 Random 10x10 Maps with Deep NN

We then attempted multiple maps again with this agent, pre-training it on an empty map with a goal point to bias it to go straight to the goal in a straight line. It only took 193 episodes to successfully learn the blank map. We then trained that model on maps with obstacles to investigate whether it continues to attempt to go straight to the goal or learn random obstacles along the way. The results in the Appendix (Figure 6) show that while it still struggled to generalize, this approach did yield an improvement.

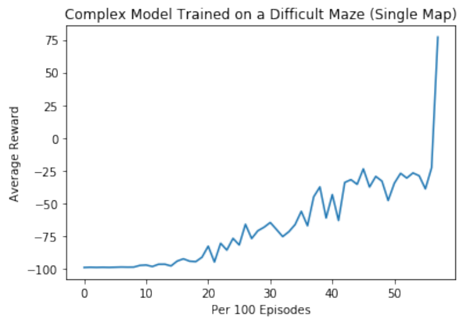
2.6. Single 10x10 Difficult Maze with Deep NN

Since generalization is not the strength of this style of reinforcement learning model, let us instead push the limits of what it is already good at: optimizing a single map. We have constructed a maze with three key characteristics: 1) A dead end path, to test if the model gets stuck in a local solution, 2) Multiple paths to the goal, testing if the agent learns the shorter one, and 3) The maze has a path as wide as the Agent itself, testing if the agent can learn a very precise pathway. Figure 2 shows that the model was stuck at a reward in the negative 90’s for a while, making next to no progress for 1000’s of episodes. Then there was steady progress through episode 5700, making it all the way up to -23. However, the model suddenly broke through and made amazing progress in the next 100 episodes, increasing its score to 73, and then solving the environment (with a score of 100) a mere 61 episodes later. We deduced that the agent reached a sort of ”critical mass” or ”breakthrough” when the randomness from the exploration constant (epsilon) had decreased enough, so it was able to get the massive reward of 100 many episodes in a row. This was such an increase that even despite the discount factor on these later rewards, the agent adjusted its policy to always follow this path.

A summary of all architecture results (Table 1) and details about each can be found in the Appendix.



(a) Complex Maze Map



(b) Deep Model/Complex Map Episodes vs. Reward

Figure 2. Deep NN trained on a Single Complex Map

3. Discussion

Our study demonstrates the potential of deep reinforcement learning for optimizing navigation in single-map scenarios. The agent’s success in learning optimal paths in known environments highlights the practical applications of this approach in real-world situations where the environment is known but the optimal path is not apparent. However, achieving generalization across multiple maps remains a significant challenge due to the high dimensionality of the state space and the complex tuning of hyperparameters involved in reinforcement learning.

Deep reinforcement learning algorithms, such as DQN, can struggle with generalization due to their reliance on memorizing state-action pairs and rewards for specific states. This can lead to overfitting on the training data and poor performance in unseen environments. The importance of hyperparameter tuning in reinforcement learning cannot be overstated, as even minor changes can significantly impact learning and performance.

Future work could explore alternative reinforcement learning algorithms or techniques to enhance generalization capabilities and further optimize navigation in a broader range of environments. There are techniques such as Double DQN and Priority Replay that could potentially be promising for generalization of DQN models to perform different tasks. However, the case of training an Agent to perform a single task, as we’ve done here, has plenty of applications in the real world, such as robotics, power grids, large control systems, game solving, and recommendation systems.

In conclusion, we consider this study successful and look forward to future iterations of this concept.

4. Appendix: Neural Network Models

4.1. Simple Q Network

Optimizer: Adaptive Moments (Adam)
Linear (state_size, 64)
ReLU
Linear (64, 64)
ReLU
Linear (64, action_size)
ReLU

4.2. Deep Q Network

We used a 5% dropout after the second layer and 10% after the third when training on multiple maps, since it helps prevent overfitting. However, we removed the dropout when optimizing performance for a single map. We have also applied Layer normalization, which normalizes over the feature dimension to reduce covariate shift and sensitivity to sudden change in input distribution, like in the case of many obstacles appearing in the visual field at once or maps with very different layouts.

LSTM layers benefit us when the agent needs to consider a long history of observations to make decisions since it captures the temporal dependencies of certain action choices. All Linear layers have a ReLU activation, except for the last one, which uses $tanh(x)$. ReLU is much more computationally efficient and safer in a deep network to prevent vanishing gradients ($tanh(x)$ was stagnating learning.) However, in the last layer, we take advantage of the zero-centered nature of $tanh(x)$ due to it's faster convergence and symmetry breaking providing a more balanced action space output due to it preventing oscillations in the Q-Matrix (while still allowing neurons to learn different features.) We are still using Adaptive Moments (Adam) as the optimizer.

LayerNorm (state_size)
LSTM (state_size, 64)
LayerNorm (64)
Linear (64, 128)
ReLU
Dropout (0.05)
LayerNorm (128)
Linear (128, 128)
ReLU
Dropout (0.1)
LayerNorm (128)
Linear(128, 32)
tanh
LayerNorm (32)
LSTM (32, action_size)

Model	Map	Learned
Simple	Simple	Yes
Simple	100 10x10	No
Simple	100 8x8	No
Deep	Simple	Yes
Deep	Empty	Yes
Deep	5 10x10	No
Deep	Maze	Yes

Table 1. Summary of Results

5. Appendix: Performance Results

A summary of the results can be seen in Table 1.

5.1. Simple Model Simple Map

Episode 1500 Average Score: 45.00
Episode 1600 Average Score: 52.00
Episode 1700 Average Score: 72.00
Episode 1766 Average Score: 99.00
(Figure 1 included in the paper.)

5.2. 100 Random 10x10 Maps with Simple NN

Episode 19700 Average Score: -26.00
Episode 19800 Average Score: -19.00
Episode 19900 Average Score: -30.00
Episode 20000 Average Score: -27.00

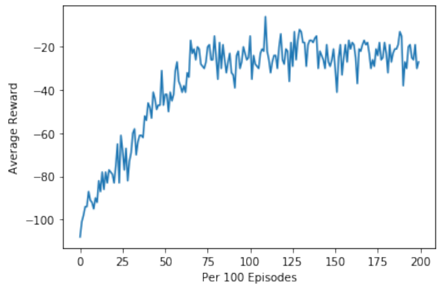


Figure 3. Simple Model Trained on 100 10x10 Maps

5.3. 100 8x8 Maps with Increased Exploration

Episode 29700 Average Score: -74.00
Episode 29800 Average Score: -76.00
Episode 29900 Average Score: -83.00
Episode 30000 Average Score: -80.00

5.4. Single Simple 10x10 Map with Deep NN

Episode 3700 Average Score: 34.126
Episode 3800 Average Score: 23.13
Episode 3900 Average Score: 30.16
Episode 4000 Average Score: 31.11

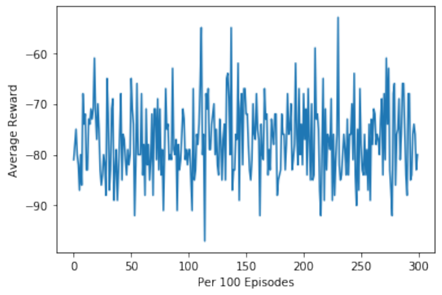


Figure 4. Simple Model Trained on 100 8x8 Maps

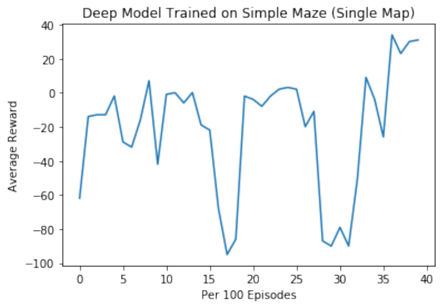


Figure 5. Deep Model Trained on Single Map

5.5. 5 Random 10x10 Maps with Deep NN

Episode 1700 Average Score: -1.73
Episode 1800 Average Score: -31.42
Episode 1900 Average Score: -7.009
Episode 2000 Average Score: -1.790

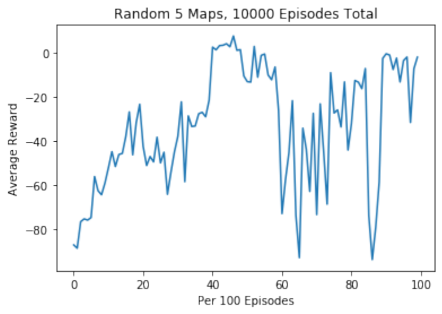


Figure 6. Deep Model Trained on 5 Maps

5.6. Single 10x10 Difficult Maze with Deep NN

Episode 5600 Average Score: -21.57
Episode 5700 Average Score: -13.42
Episode 5800 Average Score: 73.961
Episode 5871 Average Score: 99.59
(Figure 2 included in the paper.)