Transfer Learning in Board Games: Can’t Stop

Computer Science Senior Project

Author: Daniel James Marsh

Advisor: James Glenn

May 2021

Abstract

Many important tools in machine learning and artificial intelligence today were originally designed to play games. From Nimatron in 1940 to DeepMind's AlphaStar in 2019, game-playing agents have always tested the limits of machine learning and artificial intelligence (Rougetet, 2017; Risi, 2020). This project takes its inspiration from this rich history and explores transfer learning through the lens of the board game Can't Stop (Sackson, 2011). This project applies transfer learning to three existing machine learning techniques and analyzes each technique's ability to learn successful strategies when playing Can't Stop.

This project implements three game-playing agents using Q-Learning, Long-Short Term Memory networks, and Autoencoder networks respectively. To experiment with transfer learning in each case, these game-playing agents are trained using information about a smaller version of Can't Stop in which three-sided dice are used before being tested on the full version of the game in which six-sided dice are used. To carry out this testing, each agent simulates a large number of games against a variety of simple move selection algorithms. The main results of this project focus on the outcomes of games played against a random opponent, however data about the outcomes of games played against a simple logic algorithm is also included.

This project does not explore applying transfer learning between different games or the idea of a general game-playing agent; rather, the primary goal of this project is to analyze how well, if at all, a variety of machine learning techniques are able to apply transfer learning and generalize strategies between different versions of Can't Stop.

Contents

1 General Information

This project attempts to apply transfer learning to the board game Can't Stop through a variety of machine learning techniques. All code for the game-playing agents and the game implementation is written in C++. Code used to train and call any neural networks for this project is written in Python using the TensorFlow package. Code used to extract optimal strategies for the smaller version of the game with three-sided dice was provided by James Glenn and is written in Java (Glenn, 2020). All code for this project, along with the project work log and instructions on how to run game simulations, is available in the project GitHub repository.

Given the extensive work already done in the field using the machine learning techniques that I explore in this project, training each agent to generate successful strategies in the smaller version of the game was mostly straightforward; the major challenge and focus of this project was coming up with ways to have the agents transfer and apply these strategies to the full version of the game. To do this, the agents in this project either create a shared state representation between different versions of the game or allow for input of different shapes and sizes for different versions of the game. Each of the three agents in this project approaches this problem in a slightly different way and to varying degrees of success. This report will outline the methods used to implement each approach, present the results from testing, and discuss the success or shortcomings of each approach.

2 Simple Agents

For this project, two simple agents are used to evaluate the quality of the transfer learning agents. The first is an agent that simulates random play by stopping one third of the time and rolling two thirds of the time. If this agent decides to roll, a random roll is generated, and the agent chooses a pairing at random. The second simple agent is a game knowledge agent. This agent uses some simple logic rules to determine whether to stop or roll in each state. This agent will only stop if one of the following three conditions is met: a runner is at the top of its column or at least three steps ahead of the banked progress; at least two runners are at least two steps ahead of their banked progress; or all three runners are one steps ahead of their banked progress. If this agent decides to roll, a random roll is generated, and the agent attempts to select the best pairing with some simple rules. The agent will first attempt to create a paring that allows it to move a runner in the same column twice. If that is not possible, the agent will attempt to create a pairing that allows it to move a runner in two different columns. If that is not possible, the agent attempts to create a pairing that moves one runner in a single column. Finally, if that is not possible, the agent selects a random pairing since all pairings result in losing your progress.

3 Q-Learning

Q-Learning is a commonly used reinforcement learning algorithm to solve, or sometimes approximate, stochastic problems (Watkins, 1992). The Q-Learning algorithm solves stochastic problems by simulating many iterations of a problem and tracking the observed value of each visited state-action pair. After sufficient training, for any finite Markov decision process, the Q-Learning algorithm is guaranteed to converge to the optimal strategy.

To apply transfer learning to the Q-Learning algorithm, the state-action values should be initialized before training to an estimation of the optimal values based on some different version of the problem. If transfer learning is successful here, this initialization will allow the algorithm to converge, or at least arrive at a decent estimation, of the optimal strategy with less training both in terms of time and number of simulations.

For a game such as Can’t Stop, where a better strategy results in a win after fewer turns, as a Q-Learning algorithm is able to exploit a strategy more successfully, it will be able to simulate more games in a shorter period of time. Therefore, if a Q-Learning agent using the initialization table is able to generate and exploit a decent strategy sooner, it will be able to, on average, simulate more games than a Q-Learning agent that is not using the initialization, even if both are allocated the same amount training time. That being said, if transfer learning is truly successful in this case, the Q-Learning agent with initialization should approach a better estimation of the optimal strategy both after fewer simulations and shorter training time. This project will review both number of simulations and training time when presenting the results for this section.

3.1 Implementation

To apply transfer learning to the Q-Learning algorithm, my implementation uses data from the smaller version of the game with three-sided dice to initialize a table of estimated state-action values before training on the full version of the game with six-sided dice using the traditional Q-Learning algorithm. As discussed above, if transfer learning is successful in this case, this should allow my Q-Learning agent to approach a decent approximation of the optimal strategy for the full version of the game faster than a Q-Leaning agent without the initial table.

To generate the initial table, I compute the expected number of turns to win from each state in the smaller version of the game. To generalize this data between different versions of the game, I use a three-dimensional state bucketing scheme that maps states to buckets based on three characteristics: the agent’s position in the three columns with the most progress; the agent’s position in all other columns relative to the agent’s progress in its three best columns; and the distance between the agent’s runners and the banked progress in the columns containing runners. Each of these characteristics is normalized to a value from zero to one and used to map a given state to a bucket. My implementation then computes the expected number of turns to win from each state in the smaller game, determines the bucket of each state, and initializes a table to store the average bucket values based on this data.

This table of bucket values for the smaller version of the game can now be used as a table of estimated bucket values for the full version of the game; however, as discussed in the introduction to this section, the Q-Learning algorithm uses values for state-action pairs to determine the optimal strategy, not just values for each state. Since Can't Stop has two separate action points, choosing whether to roll or stop, and later choosing which pairings to select if the player decided to roll, I will discuss my implementation's approach to these two actions separately.

I will first discuss my implementation's approach to the roll or stop decision. Since there are only two actions at this point, my implementation simply initializes both action values to the value of the corresponding bucket from the table discussed above. Then, during training, my implementation updates the action values separately depending on the action taken at that point in training.

The decision point for the roll pairings is slightly more complex since there are too many possible roll combinations to store each combination as a unique action, and the number of possible roll combinations varies depending on the version of the game, and thus could not be generalized across different versions of the game. To avoid this problem, my implementation does not treat choosing a roll pairing as an individual action, but rather groups it in with the decision to roll. Therefore, when the agent decides to roll, a random roll is generated, and the next state value, which is used to update the table, is calculated as the best value of all possible resulting states from that roll.

In addition to the modifications discussed above, my implementation also uses unique alpha values (learning rates) for each bucket-action value in the table. This is to ensure that some very common states do not influence the learning rate used when visiting rarer states. Without this modification, the results of the algorithm vary massively.

Aside from the changes discussed above, my implementation behaves fairly similarly to the traditional Q-Learning algorithm. I use a decaying epsilon value to balance exploration and exploitation, and reward shaping to try to approach a good approximation after fewer simulations.

Once training is complete, my implementation stores the table of bucket-action values to be used in testing. In testing, my implementation maps a given state to a bucket and determines whether rolling or stopping has a better estimated value from that bucket. If stopping has a better estimated value, my implementation stops and returns the new state. However, if rolling has a better estimated value, my implementation generates a random roll, determines which roll pairing results in the state with the best estimated value, makes the corresponding move, and returns the new state.

3.2 Results

In this subsection, I will discuss the results of my Q-Learning algorithm. To test the success of transfer leaning in this case, I tested my implementation against a random agent with several different training times. I then tested an equivalent Q-Learning algorithm that does not use the initial table against the same random agent with the same several training times.

As a baseline, when playing the smaller version of the game in solitaire mode, my Q-Learning agent plateaus after about 5 seconds of training and takes an average of 4.74 turns to complete the game. This is fairly far from the optimal strategy (expected 0.89 turns) for the smaller version of the game, but an explanation for this discrepancy will be provided in the discussion section below.

This subsection will only present visualizations of this data but included in the appendix is the full raw data along with the results for the same testing against an agent with some simple logic to take advantage of some simple strategies.

First, I will present the results of the Q-Learning agent with the initial table compared to the results of the Q-Learning agent without the initial table by setting training time to be equal. This does not necessarily mean that both agents were able to simulate the same number of training games, this just guarantees that both agents had equal time to train. For this testing, I used training times from 0.5 to 5 seconds at 0.5 second increments. I trained 100 agents at each time value and simulated 500 games per agent.

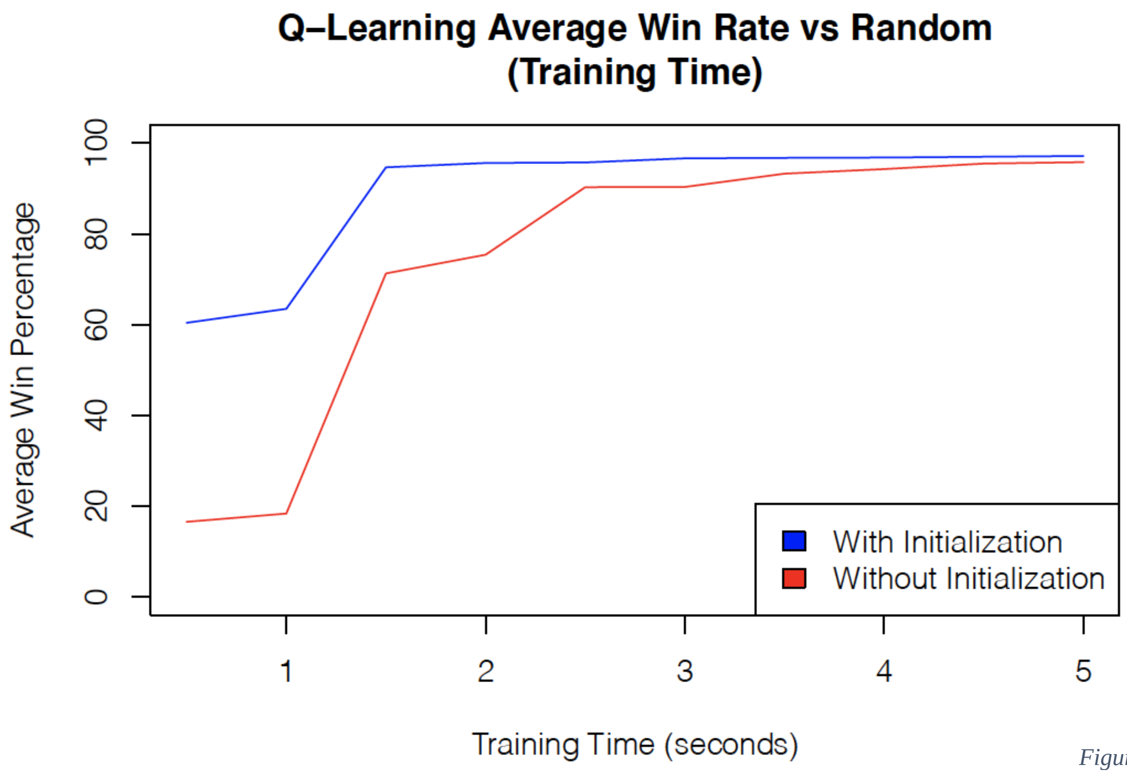


Figure 1

The figure above displays the average win rate of the Q-Learning agents against a random agent based on training time. The blue line represents the Q-Learning agents that use the initial table while the red line represents the Q-Learning agents that do not use the initial table. This color scheme is consistent throughout this report.

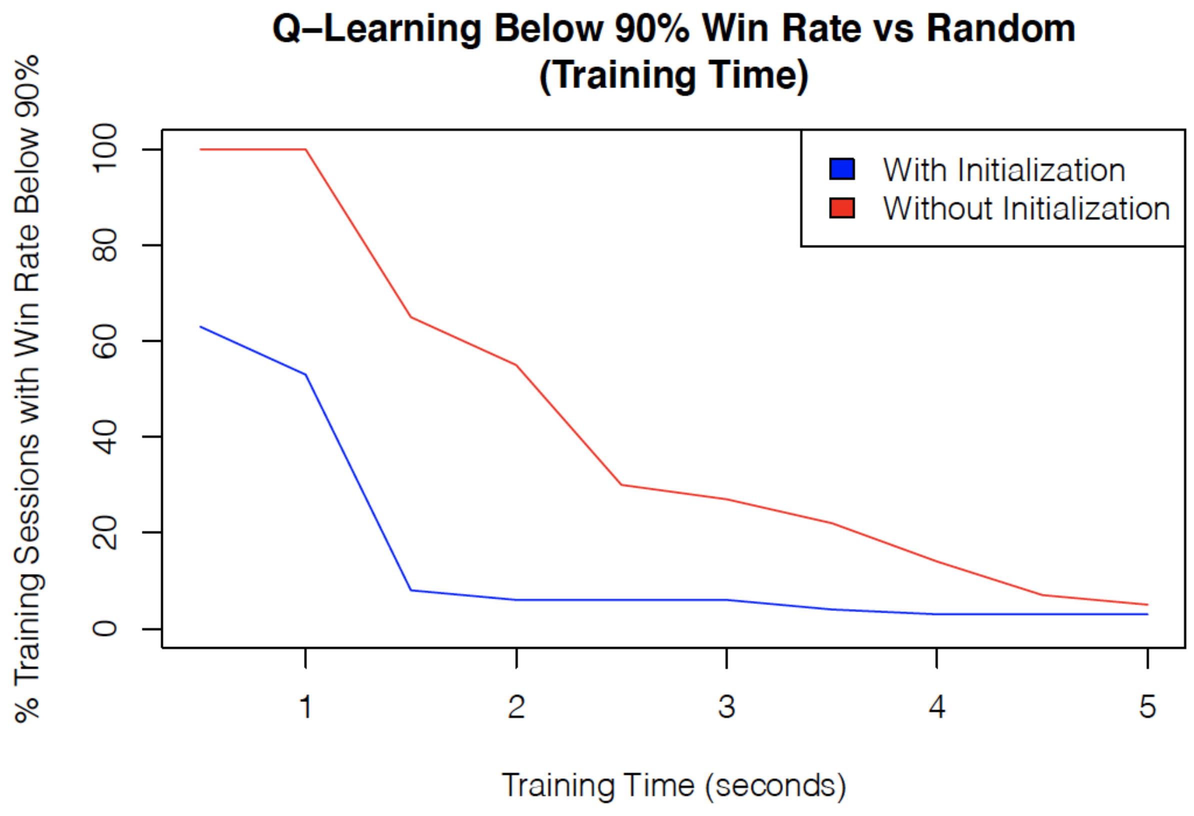


Figure 2

The figure above displays the percentage of training sessions that result in agents that won fewer than 90% of games simulated against a random opponent based on training time.

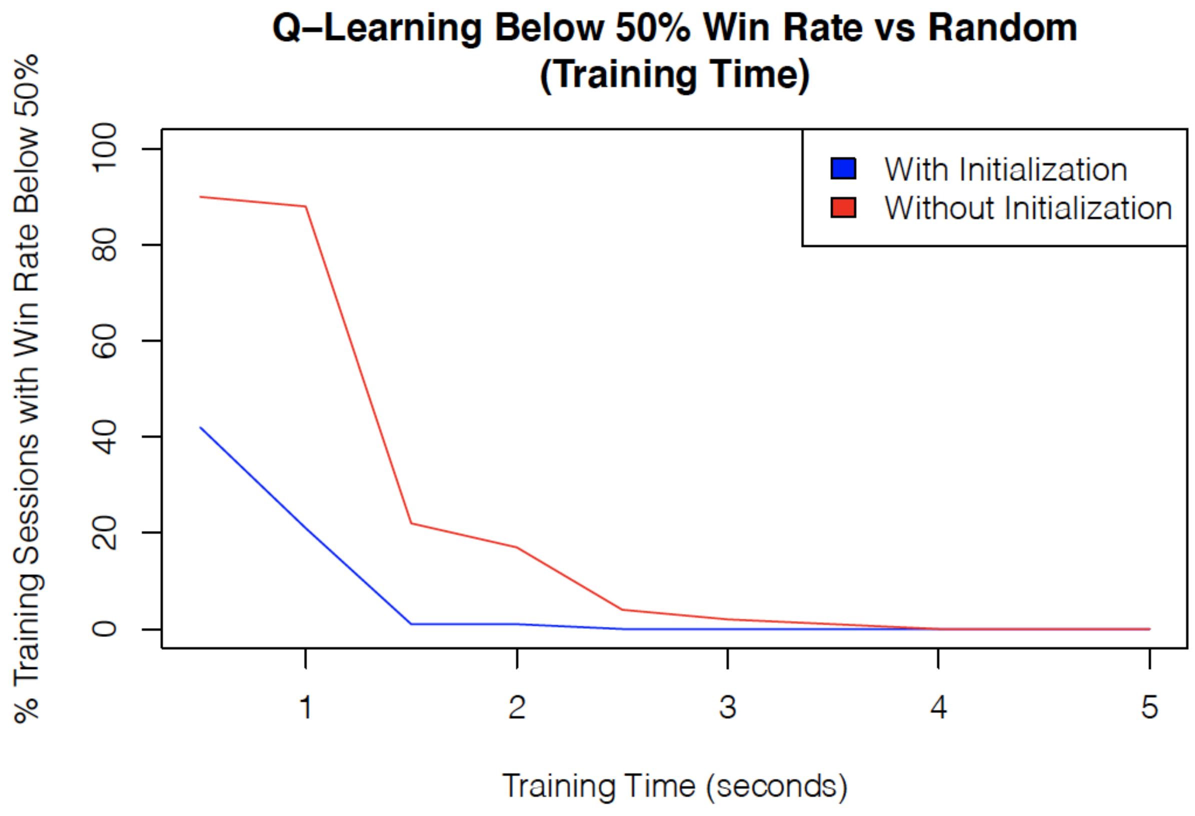


Figure 3

The figure above displays the percentage of training sessions that result in agents that won fewer than 50% of games simulated against a random opponent based on training time.

The data presented below comes from the same testing; however, the following figures set the number of games simulated to be equal between agents rather than allocated testing time.

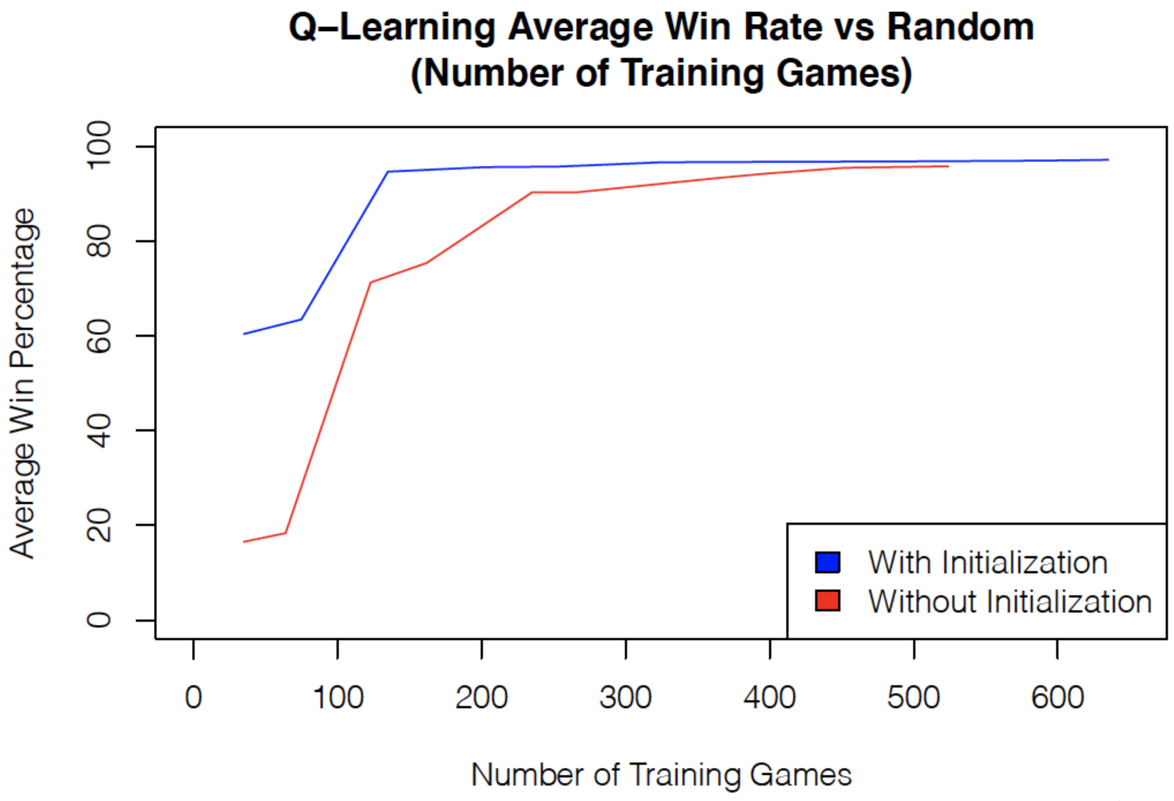


Figure 4

The figure above displays the average win rate of the Q-Learning agents against a random agent based on number of games simulated during training.

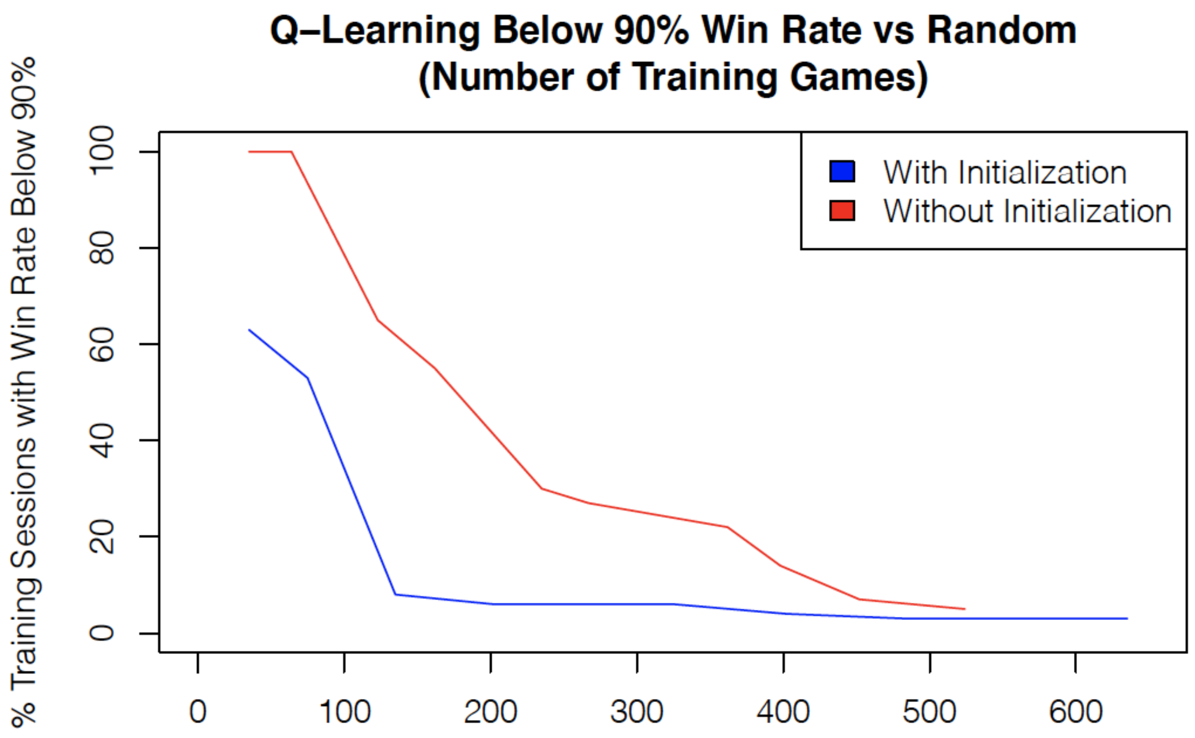


Figure 5

The figure above displays the percentage of training sessions that result in agents that won fewer than 90% of games simulated against a random opponent based on number of games simulated during training.

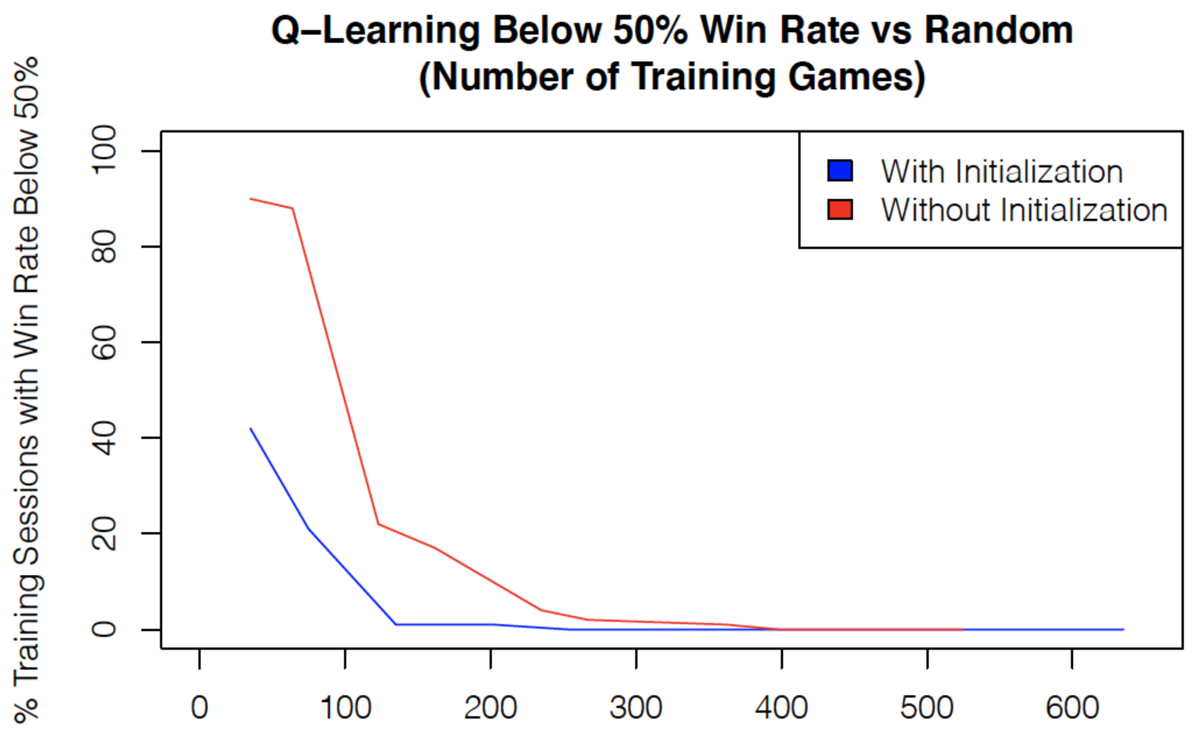


Figure 6

The figure above displays the percentage of training sessions that result in agents that won fewer than 50% of games simulated against a random opponent based on the number of games simulated during training.

3.3 Discussion

In this subsection, I will discuss the results presented above and analyze the success of transfer learning in this case, as well as examining any shortcomings or potential improvements to my approach.

3.3.1 Discussion of Results

First, I will discuss the results for the solitaire version of the smaller game. The results, as presented above, are far from the optimal strategy for this version of the game. There are a few possible explanations for this, but the most likely lies with the bucketing scheme. Since the bucketing scheme is an attempt to generalize the state representation between different versions of the game, some state information is inevitably lost when using buckets instead of states for Q-Learning. Additionally, since the bucketing scheme in my implementation is designed to balance the distribution of states between buckets for the full version of the game, the states in the smaller version of the game are not evenly distributed between buckets. This, in addition to losing some state information, provides a clear explanation for why my Q-Learning agent does not approach the optimal strategy for the smaller version of the game.

Since the bucketing scheme is also used for the full version of the game, the algorithm is not guaranteed to converge to the optimal strategy there either. As such, in the following section, I will not be analyzing my agent's ability to converge to the optimal strategy for the full version of the game, but rather I will discuss my agent's ability to approach an estimation of the optimal strategy for the full version of the game.

Next, I will focus on figures 1-3 where the results are based on the amount of time the Q-Learning algorithm was given to train. In figure 1, both the agents with the initial table and the agents without the initial table perform better with longer training times. Also, both types of agent seem to converge to an average win rate around 97% against the random player with sufficient training time. However, it is clear that the agents with the initial table are able to achieve higher average win rates with lower training time. This result suggests that the information contained in the initial table, which is based only on information from the smaller version of the game, is able to be generalized to the full version of the game.

Now looking to figure 2, there is a similar trend with both types of agents approaching zero with sufficient training time; however, again, the agent with the initial table performs significantly better at the lower training times.

Finally, looking to figure 3, the same trend is present once again with both types of agents approaching zero after three to four seconds of training time, but yet again, the agent with the initial table performs significantly better at the lower training times.

These three figures show that the agents using the initial table not only perform better on average at lower training times but are also much more consistent at the lower training times. Overall, these results demonstrate that my Q-Learning agent is able to apply transfer learning in this case and generalize information from the smaller version of the game to help in producing a more successful strategy for the full version of the game.

While these results are very promising, it is important to analyze any potential bias. Since the use of the initial table allows the Q-Learning algorithm to exploit decent strategies after a shorter training time, an agent trained with the initial table is in fact able to train on a larger number of games than an equivalent agent without the initial table even if the two agents are allocated the same amount of training time. As such, this could be an explanation for the difference in performance. To determine if this is the case, I will now discuss figures 4-6.

Firstly, looking at figure 4, it is clear that even when number of games simulated is set to be equal, the agents with the initial table perform better after fewer training simulations. Both figure 5 and figure 6 show that the agents with the initial table are also more consistent after fewer training simulations.

Since the results in figures 4-6 agree with the results in figures 1-3, the most likely explanation for the difference in performance and consistency is the use of the initial table. These results suggest that transfer learning was successfully applied in this case and that the Q-Learning algorithm was able to take advantage of information from the smaller game to generate a more effective strategy in the full version of the game.

3.3.2 Potential Improvements

While the results presented above are very promising, my implementation was not perfect, and this subsection will discuss any areas where I believe my approach could be improved.

With sufficient training time, my algorithm is able to generate a strategy that can win up to 99.8% of games against a random opponent. In fact, even with a training time as short as 0.5 seconds, my algorithm is able to sometimes generate a strategy that can win up to 99% of games against a random opponent; however, these shorter training sessions can also result in strategies that win less than 10% of games against a random opponent. This inconsistency is likely due to an issue with the training parameters alpha (learning rate) and epsilon (exploration rate). With more time to experiment with this algorithm and fine tune these parameters, I am confident that my implementation could provide much more consistent results, even after shorter training sessions.

Even though my implementation could be improved with further tuning, it is clear from the results presented in this section that the Q-Learning algorithm is able to apply transfer learning and take advantage of information from the smaller version of the game to generate a more successful strategy for the full version of the game.

4 Long-Short Term Memory Network

A long-short term memory network, referred to as a LSTM network from this point on, is an example of a recurrent neural network (RNN) that is able to retain some information about previous data by taking the output of the previous step as the input to the next step (Olah, 2015; Hochreiter, 1997). The recursive nature of LSTM networks allows them to interpret input of different sizes. This makes a LSTM network a very interesting approach for this problem since no shared state representation is necessary.

To apply transfer learning with a LSTM network, the network simply needs to be trained using data from one version of a problem before being testing on a different version of the problem.

4.1 Implementation

To apply transfer learning with my LSTM agent, I use data from the smaller version of the game to train the network before testing the network against several simple agents in the full version of the game.

The training input for my network uses three characteristics of each column in each possible state of the smaller version of the game: the position of the banked progress in each column; the position of the runner in each column (if there is no runner in a column, this is set to the same as the banked progress); and the probability of getting a roll that could allow for progress in each column. Each of these three values is then normalized to a value from zero to one. The training output is then calculated for each state of the smaller version of the game by determining whether stopping or rolling from that state results in a state with a lower number of expected turns to win. If stopping is calculated to be better in a given state, the training output for that state is set to 0.0. If rolling is calculated to be better in a given state, the training output for that state is set to 1.0.

Once the training data has been computed and formatted, it is fed into the LSTM network for training. Each state is treated as a single example for training, but the data is fed in one column at a time to allow the network to take states from any version of the game as input regardless of the number of columns. The output of the LSTM network is a value between zero and one with outputs closer to zero representing states in which the LSTM network estimates stopping to be better and outputs closer to one representing states in which the LSTM network estimates rolling to be better.

At this point, my implementation is able to train a LSTM network to estimate whether rolling or stopping is better at a given state but is not able to select the optimal roll pairing after the agent decides to roll. To make this decision, my implementation uses a separate neural network to estimate the optimal roll pairing at each state. This network is implemented as a traditional feed-forward neural network that takes as input three pieces of data about a single column, and outputs the net change in expected number of turns to win if that column is selected in a roll. The three pieces of input data are the same pieces of data used for each column in the LSTM network: the positions of the banked progress in a given column; the position of the runner in a given column (if there is no runner in that column, this is set to the same value as the banked progress); and the probability of getting a roll that could allow for progress in a given column. To generate the training output for this network, I compute the net change in expected number of turns to win for moving in each column for all possible states of the smaller version of the game. Once training is complete, this network is able to estimate the net change in number of turns to win after moving in a given column from a given state for any version of the game.

During testing, my implementation formats the current state information into a readable format for both the LSTM network and the feed-forward network. My implementation then calls the LSTM network with the formatted state data to decide whether to roll or to stop. Since the LSTM network outputs a value from zero to one, my implementation makes moves probabilistically based on the output value. That is to say that, for output values closer to one, my implementation is more likely to roll, while for output values closer to zero, my implementation is more likely to stop. If my implementation decides to roll, a random roll is generated, and all possible roll pairings are generated. For each possible roll pairing, the two columns selected by that pairing are each input into the feed-forward network and the output results are summed. My implementation then selects the roll pairing that results in the largest decrease in expected number of turns needed to win.

Once the roll or stop decision has been made by the LSTM network and the roll pairing decision has been made by the feed-forward network (if applicable), my implementation makes the suggested move and returns the new state.

4.2 Results

In this subsection, I will present the results of my LSTM agent. Since my LSTM agent is not exposed to the full version of the game during training, my results are just based on simulations that have my LSTM agent simulate the full version of the game against a random player and against a player with some simple logic.

As a baseline, when playing the solitaire version of the smaller version of the game, the LSTM agent takes an average of 6.37 turns to complete the game. Similarly to the Q-Learning agent, this is far from the optimal strategy for this version of the game (expected

0.89 turns) and an explanation for this discrepancy will be given in the discussion section below.

For the full testing, I first simulated 500 games between my LSTM agent and a random player. The LSTM agent was able to win 421 (84.2%) of these games. Next, I simulated 500 games between my LSTM agent and an agent with simple logic. In that case, the LSTM agent was able to win 348 (69.6%) of the games.

4.3 Discussion

In this subsection, I will analyze the results of my LSTM agent as well as discuss any potential shortcomings or improvements that could be made to my agent.

4.3.1 Discussion of Results

First, I will discuss the results from the solitaire version of the smaller game. As stated above, the LSTM agent performs significantly worse than the optimal strategy for that version of the game. As with the Q-Learning agent, the explanation for this shortcoming likely lies with the generalized state representation. Since the LSTM network takes formatted data about each column as input rather than the full state representation, some information is inevitably lost. This ultimately means that the LSTM agent, as it is currently designed, will not converge to the optimal strategy for any version of the game, but rather it will approach an estimation of the optimal strategy for all versions of the game. As such, in the remainder of this section, I will discuss the LSTM agent’s ability to approach an estimation of the optimal strategy rather than its ability to replicate the optimal strategy.

Next, I will focus on the results of the games simulated against the random opponent. Since the LSTM agent was able to win significantly more than 50% of games against the random opponent, we can conclude that the strategy employed by the LSTM agent is better than random play for the full version of the game. Additionally, since the LSTM agent is not exposed to the full version of the game during training, the strategy that it employs is based entirely on information from the smaller version of the game. This suggests that the LSTM network used in my implementation is able to apply transfer learning and take advantage of information from the smaller version of the game to generate a successful strategy for the full version of the game that is at least better than random play.

Next, I will discuss the results of the games simulated against the agent with simple logic. In this case, the LSTM agent was able to win slightly more than 50% of the simulated games. This implies that the strategy employed by the LSTM agent is slightly better than some very simple logic.

Overall, we can conclude that the LSTM agent is able to apply transfer learning in this case and is able to produce a decent strategy for the full version of the after being trained only on data from the smaller version of the game.

4.3.2 Potential Improvements

Once again, the results discussed above are very promising, but there are several improvements that could be made to my implementation. In this subsection, I will discuss a few shortcomings of this approach and suggest potential solutions to these problems.

The first and most obvious shortcoming of this approach is the relatively low win rate against the testing agents. The results from the Q-Learning agent show that an optimal strategy would be able to beat a random player in more than 99% of games, but the results in this section are fairly far from that level of success. There are potentially a few reasons for this low win rate, but the most obvious appears to be that in some states in the full version of the game, the LSTM network outputs a value that is clearly very far from optimal. One particularly problematic situation arises in which the LSTM network outputs a very low value after the first roll of a turn. This results in the LSTM agent making very little progress that turn. Additionally, the LSTM network often outputs a similarly low value initially during its next turn as well. This leads to a loop where the LSTM agent makes very little progress each turn and often leads to a loss even against a random opponent. This behavior suggests that the LSTM agent is not able to generalize the information from the smaller version of the game to all states in the full version of the game and therefore sometimes outputs values that are very far from optimal. One potential solution to this issue would be to use different characteristics of the game state as input to the LSTM network. By fine tuning the input of the network, it should be possible to maximize the network's ability to generalize its strategy to as many states as possible in the full version of the game.

Another shortcoming of this implementation is that the models generated by training on data from the smaller version of the game are not always able to produce a strategy better than random play for the full version of the game. In other words, the results of training the network are very inconsistent and only a select few trained models actually generate successful strategies. Again, I believe that the key to solving this issue lies in further tuning of the input data. With better data, the network should be able to more consistently generate models that are able to generalize information and produce strategies that are significantly better than random play for the full version of the game.

While there are evidently flaws in this approach, these flaws do not take away from the fact that the results in this section demonstrate that an LSTM agent is at least theoretically able to apply transfer learning in this case and produce a successful strategy for the full version of the game after only being trained with data from the smaller version of the game.

5 Autoencoder

An autoencoder is a special type of neural network that focuses on encoding and then decoding data (Baldi, 2012). In general, these networks are used to create an encoding that decreases the dimensionality of the input but that can still be decoded to closely match the original values. Autoencoders are especially interesting for transfer learning since the encodings produced could theoretically be used as a shared state representation between different versions of a problem with different state sizes.

5.1 Implementation

My implementation uses the encoder section of an autoencoder network to reduce the dimensionality of the state representations of different versions of the game in an attempt to produce a common encoded representation. That encoded representation is then given to a second neural network that estimates whether rolling or stopping is better at a given state. If stopping is selected, my implementation stops and then returns the state. If rolling is selected, a random roll is generated, and my implementation uses the feed-forward network described in Section 3.1 to select the estimated best roll pairing. My implementation then makes the suggested move and returns the new state.

My autoencoder is implemented as a feed-forward neural network that takes 33 inputs, produces 33 outputs, and has a middle hidden layer of size 15. The input and output size was selected because it allows for three pieces of data per column in the full version of the game. The input data used for each column for the autoencoder is the same as the input data used for the LSTM network: the position of the banked progress in each column; the position of the runner in each column (if there is no runner in a given column, this is set to the same as the banked progress in that column); and the probability of getting a roll that could allow for progress in each column. For the smaller version of the game, any input value that would not be filled is padded with a -1.

The autoencoder is then trained using states from both the smaller version of the game and the full version of the game. Once the autoencoder is trained, the decoder section is removed to leave just the encoder which takes 33 inputs and produces 15 outputs.

Next, a new feed-forward neural network is trained taking as input the encoded state representation and producing as output a value from zero to one representing whether the network estimates rolling or stopping to be better from that state. Once this network is trained using data from the smaller version of the game, my implementation is able to encode state representations of any version of the game (up to the full version of the game with six-sided dice), use that encoding as input to determine whether to roll or stop in a given state, and then use the feed-forward network described in Section 3.1 to select a roll pairing if the agent decides to roll.

5.2 Results

In this subsection, I will present the results of my autoencoder approach. Similar to the LSTM approach, since the autoencoder is not trained on the full version of the game, my results are just based on simulations that have my autoencoder agent play the full version of the game against a random player and against a player with some simple logic.

First, I simulated 500 games between my autoencoder agent and a random player. The autoencoder agent was able to win 27 (5.4%) of these games. Then, I simulated 500 games between my autoencoder agent and an agent with simple logic. In that case, the autoencoder agent was able to win 12 (2.4%) of these games.

5.3 Discussion

In this subsection, I will analyze the results of my autoencoder, discuss a potential explanation for the shortcomings of my approach, and suggest solutions that could resolve the issues seen in my approach.

It is clear from the results presented above that the autoencoder agent was not able to produce a successful strategy for the full version of the game. In fact, the autoencoder agent produced a strategy that is significantly worse than random play. The most likely explanation for this issue lies with the encoder network. Since I am using a traditional feed-forward network with padded inputs, it is possible that different versions of the game are generating different encoded representations. In other words, it is possible that the encoded state representations do not generalize across different versions of the game. If this is the case, the network that takes these encoded representations as input would have no way of generalizing a strategy from the smaller version of the game to the full version of the game since the input for the smaller version of the game is not related to the input for the full version of the game. To prevent this issue, padded inputs should not be used, and cyclical checks should be added to the autoencoder’s training. Specifically, during training, state representations for the smaller version of the game should be encoded and then decoded as state representations for the full version of the game before being encoded once again and finally being decoded back to the state representations for the smaller version of the game. This type of cyclical check would ensure that the encoded representations produced by the encoder are shared and generalizable across the different versions of the game.

While my implementation was not able to take advantage of transfer learning in this case, autoencoder networks have the potential to be very powerful tools to help apply transfer learning to complex problems. I am confident that with some modifications and tuning, an autoencoder network would be able to create a shared encoded state representation for different versions of Can’t Stop.

6 Conclusion

Throughout this project, I have been able to experiment with and adapt a variety of machine learning techniques to explore the idea of transfer learning. In both my Q-Learning and LSTM approaches, it is clear that the game-playing agents are able to apply transfer learning and take advantage of information from a smaller version of Can’t Stop to produce a successful strategy for the full version of the game. While my autoencoder agent was not able to apply transfer learning, I am confident that with further experimentation and tuning, an autoencoder network could be used to generate shared state representations between different versions of Can’t Stop and would then be able to fully facilitate transfer learning in this case. Although not all approaches detailed in this report were fully successful, the results presented in this report make it very clear that under the correct circumstances, transfer learning can be a powerful tool to generate successful strategies for solving complex problems.

References

Abadi, M. et al. (2015). TensorFlow: Large-Scale machine learning on heterogenous systems. *Tensorflow.org*.

Baldi, P. (2012). Autoencoders, Unsupervised Learning, and Deep Architectures. *Proceedings of ICML Workshop on Unsupervised and Transfer Learning, in Proceedings of Machine Learning Research*, 27:37-49.

Glenn, J. (2021). Cant_stop.jar.

Hochreiter, S., & Schmidhuber, J. (1997). Long-Short Term Memory. *Neural Computation*, 1735-1780.

Olah, C. (2015). Understanding LSTM Networks. *Colah's Blog.*

Risi, S., & Preuss, M. (2020). Behind DeepMind's AlphaStar AI that Reached Grandmaster Level in StartCraft II. *KI-Künstliche Intelligenz*, 85-84.

Rougetet, L. (2017). Machines Designed to Play Nim Games (1940-1970): A Possible (Re)Use in the Modern French Mathematics Curriculum?. *ICME-13 Monographs*, 229-250.

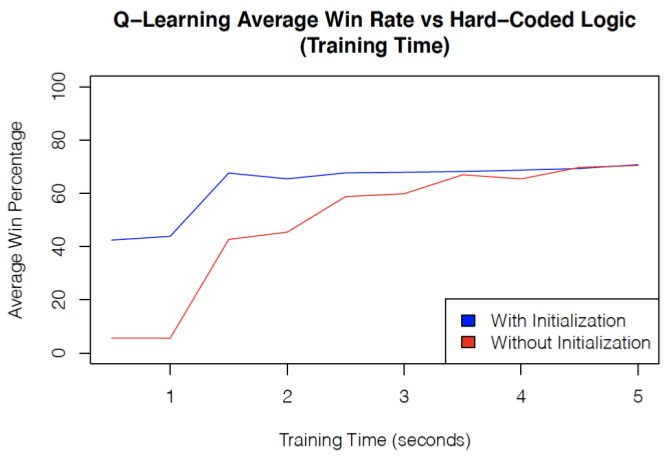
Sackson, S. (2011). Can't Stop. *Gryphon Games*.

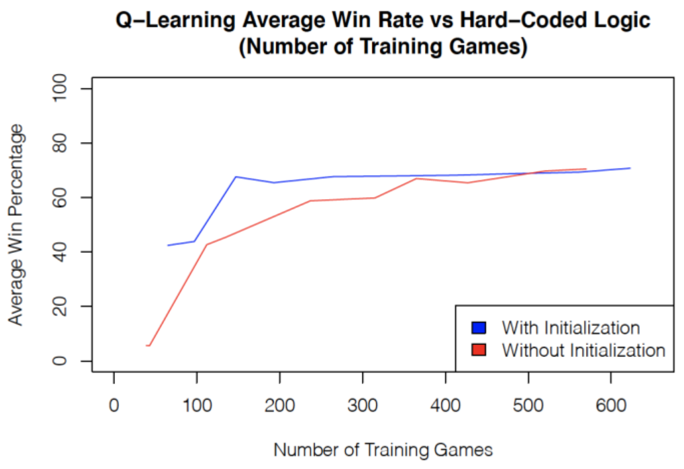
Watkins, C., & Dayan, P. (1992). Q-Learning: Technical Note. *Kluwer Academic Publishers*, 279-292.

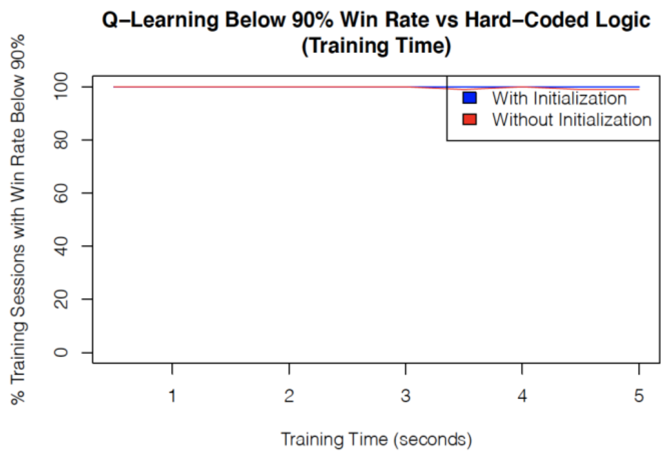
Appendix

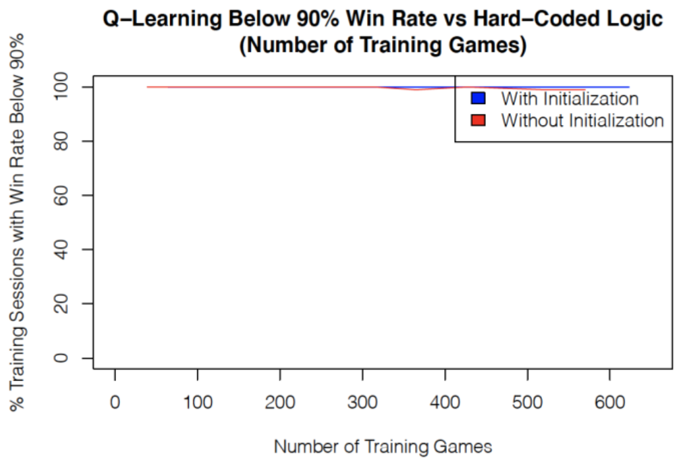
For full results data for the Q-Learning approach, see Other/Q_results.xlsx .

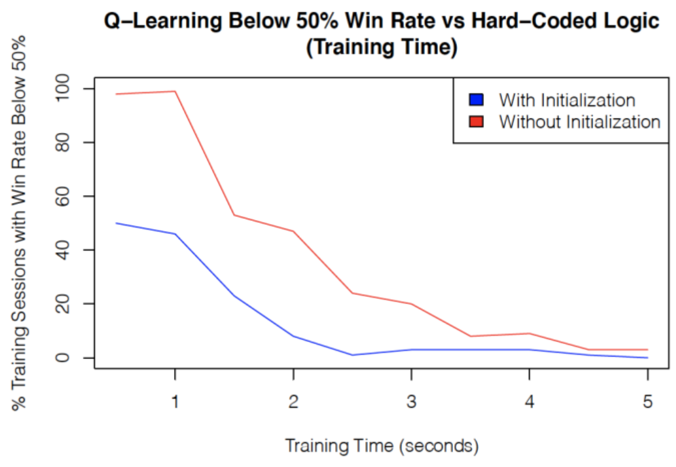
See below for visualizations of the results of the Q-Learning agent tested against an agent with simple logic.

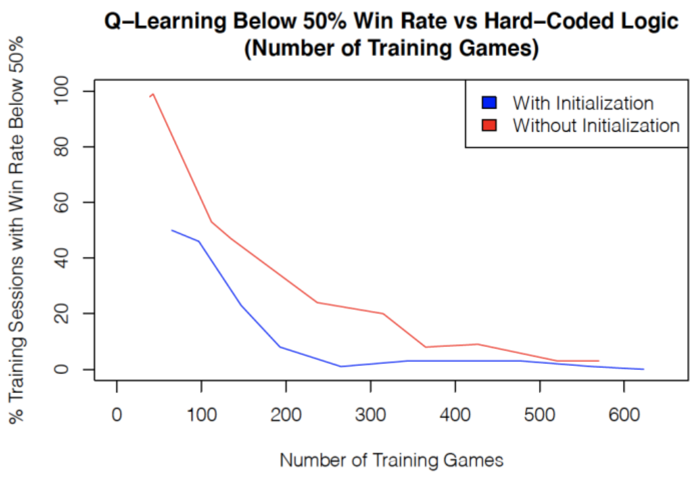

Q–Learning Average Win Rate vs Hard–Coded Logic (Training Time)
Average Win Percentage
Training Time (seconds)
With Initialization
Without Initialization


Q–Learning Average Win Rate vs Hard–Coded Logic (Number of Training Games)
Average Win Percentage
Number of Training Games
With Initialization
Without Initialization


Q–Learning Below 90% Win Rate vs Hard–Coded Logic (Training Time)
% Training Sessions with Win Rate Below 90%
Training Time (seconds)
With Initialization
Without Initialization


Q–Learning Below 90% Win Rate vs Hard–Coded Logic (Number of Training Games)
% Training Sessions with Win Rate Below 90%
Number of Training Games
With Initialization
Without Initialization


Q–Learning Below 50% Win Rate vs Hard–Coded Logic (Training Time)
% Training Sessions with Win Rate Below 50%
Training Time (seconds)
With Initialization
Without Initialization


Q–Learning Below 50% Win Rate vs Hard–Coded Logic (Number of Training Games)
% Training Sessions with Win Rate Below 50%
Number of Training Games
With Initialization
Without Initialization