Max Marcussen
12/1/2021
Advanced Data Analytics

Deep Q-Learning and Connect Four

Code can be found at https://github.com/maxmarcussen98/ConnectFour.

Abstract/Summary

In this project, I attempt to build a deep Q-learning algorithm that can effectively play Connect Four while learning and training in the most efficient manner possible. I began with Tic-Tac-Toe, a smaller-scale version of Connect Four, and built a standard Q-learning algorithm for this game. I then applied the best practices I learned for Q-learning through this game to larger board sizes. Upon discovering the limitations of tabular Q-learning, I transitioned to deep Q-learning, experimenting with various network architectures and attempting to find more best practices for efficient learning. I ended up finding a few general principles and a broadly generalizable architecture to apply to various board sizes for Connect Four-like games. However, the inherent randomness and extreme sensitivity of deep Q-learning as a method limited my ability to find an optimal set of parameters for building a Connect Four player. I ended by applying my overall best guesses for what would work for a larger board based on my discoveries with smaller board sizes, but was met with limited success for larger boards.

Part 1: Introduction

Reinforcement learning algorithms, which learn by generating their own data in real time, following a certain strategy, and reinforcing or penalizing that strategy based on success or failure, are very applicable to learning best strategies for board games. Possibly the most famous application of reinforcement learning is AlphaZero, the algorithm that defeated the reigning Go world champion.

In this project, I sought to understand how reinforcement learning, specifically deep reinforcement learning like what AlphaZero uses, works in practice. What I was most interested in was how model complexity needs to increase for larger games and how these algorithms are

efficiently trained. I decided to demonstrate the need for increased model complexity by training a Q-learning algorithm on a simple game with a small board and gradually increasing the size of that board and with it the complexity of the game. To do this, I settled on Connect Four as my goal game.

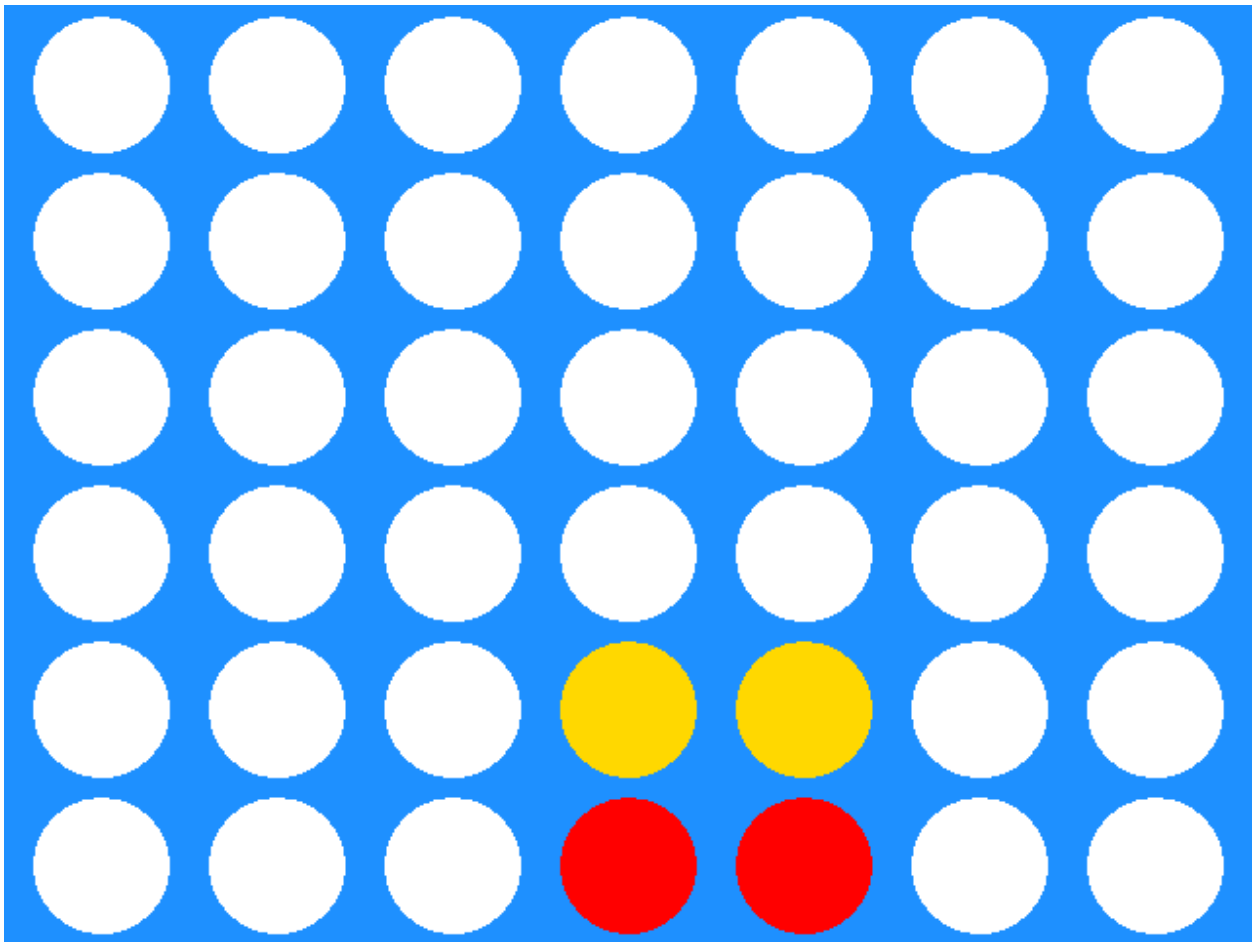


Fig. 1: a standard Connect Four board. Sourced from Vandewiele, 2017.

The rules of Connect Four are simple: on a 6x7 board, a red player and a yellow player drop tokens from the top of the board. The first player to put four of their pieces in a row horizontally, vertically, or diagonally wins. Tic-Tac-Toe, a very similar game, only requires three in a row on a 3x3 board and doesn't have a gravity restriction on moves that the player's allowed to make. By all accounts a relatively simple game for a human to play, Connect Four has around 16 trillion possible board states (Alderton et al, 2019). However, compared to Tic-Tac-Toe, which has $3^9 = 19{,}683$ states, Connect Four is almost a billion times more complex, even though the player can only move in seven spaces vs. Tic-Tac-Toe's nine spaces.

The metric for success for most of these games is a good tie rate. For example, if both players in Tic-Tac-Toe play perfectly, the game should result in a tie every time. Connect Four is slightly different in that if a player is able to get the first move in the center column, they should be able to win every time if they play perfectly, while player 2 should be able to get at least a tie every game if player 1 doesn't make this move (Alderton et al, 2019).

Part 2: Tabular Q-learning

Theory

Tabular Q-learning is a reinforcement learning method that focuses on optimizing the Q-function, or the action-value function. Given a certain state, the Q function will return a given value for an action taken in that state. For example, given a simple Tic-Tac-Toe board and given a certain set of moves already made, Tic-Tac-Toe will have a certain value function *Q(S, a)* for every state-action pair. The next move made by a tabular Q-learning algorithm will be whatever action has the highest value given this Q function.

Tabular Q-learning is called tabular because it stores its action values in a table. It updates these values based on rounds of gameplay and rewards received. For example, if winning a game of Tic-Tac-Toe grants a reward of 1 and losing grants a reward of 0, the values for an action that led to victory will be updated by applying that reward, minus the old action value for this action and times a learning rate, to the value function for that action. This update also takes into account the value of the optimal future actions made available by taking this action, multiplied by a discount factor for each step in the future that these actions can take place. Essentially, the reward for this state also involves the discounted reward for taking the best possible action in the next state, but this state is produced by the other player taking their own best action available to them. The assumption here is that both players will take their best known actions in the next future.

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Big)}^{\text{temporal difference}}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

Figure 2: the equation for updating the action-value (Q) function. Sourced from Wikipedia.

Implementation

Q learning is a popular coding problem online, and there are multiple tutorials available. I borrowed parts of the structure of my code from one of these tutorials (Zhang, 2019). Essentially, states and values are stored in a dictionary, and the algorithm will choose an action based on what the maximum value of the next state it's able to access is. Given a certain set of rewards for wins, losses, and ties, it'll then update all of the values in its dictionary based on the outcome of the game and according to the formula above.

Only two parameters are given to this algorithm: an exploration rate and a learning rate. An exploration rate, or epsilon, is essentially how often the algorithm will make a random move. If the reinforcement learner isn't allowed to make random moves once in a while, it'll only update the values for moves it already has stored and not learn any new strategies. The learning rate, or alpha, is how much the updates to the Q values are scaled by.

Initially, for my Tic-Tac-Toe Q-learning implementation, I tried to use a fixed learning rate and a fixed exploration rate. The first discovery I made was that larger exploration rates learn faster, but stall out at a higher win rate. What we're optimizing for is a low rate of games where a certain player wins, since both players should be able to tie every time if they play perfectly. A lower exploration rate converges to a better tie rate, but takes far longer to do so.

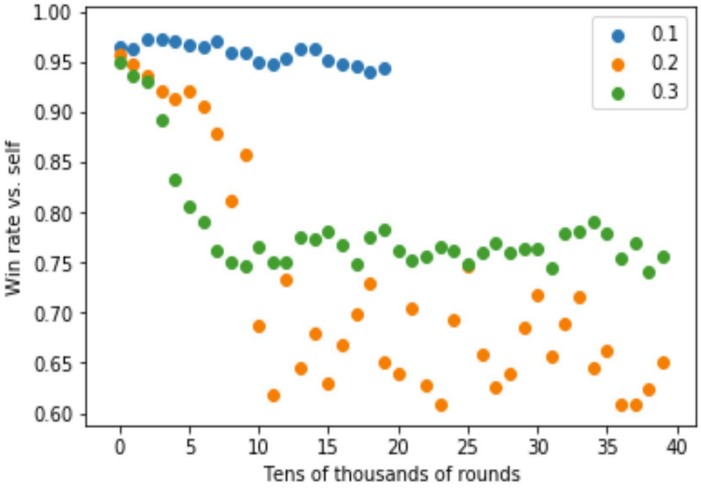


Figure 3: win rate (percent of time that one player wins) vs. rounds of training for learning rates 0.3, 0.2, and 0.1. 0.3 learns fastest, converging to its lowest win rate in around 50,000 games, while an explore rate converges to a lower win rate in about twice as long. 0.1 doesn't converge at all.

If our goal is to learn efficiently, converging well and converging quickly, how do we balance between dropping to an optimal tie rate quickly and having that tie rate be good? We'd like to converge to a 0% win rate, ideally, but we don't want that to take an extremely long time with a low exploration rate. The solution I came to was to start at a high explore rate, but

gradually drop it down across rounds of training. Starting at a very high exploration rate (0.4, or every move having a 40% chance of being random) and dropping that learning rate by 0.05 every 10,000 rounds of training produced the fastest training that still converged to a win rate of 0.

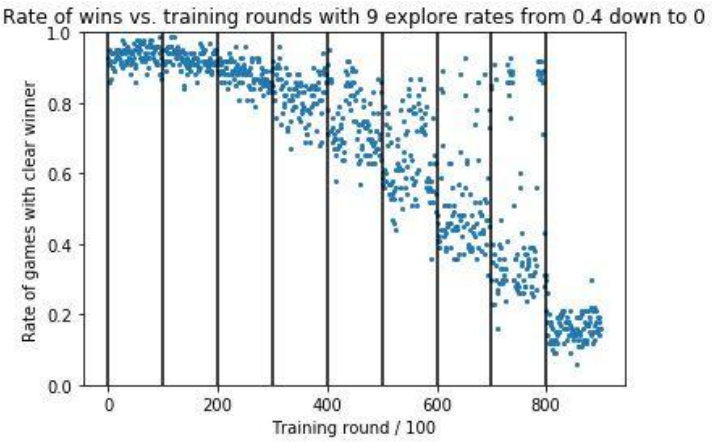


Figure 4: win rate vs. rounds of training for a dynamic exploration rate. Starting at an exploration rate of 0.4, every vertical line on the graph represents a drop in the learning rate by 0.05. The final section has an exploration rate of 0, allowing the algorithm to learn from only from all of the moves it already has collected.

The next discovery I made was that dropping the learning rate along with the exploration rate reduced training time significantly. I was able to cut training time in half and still have the algorithm converge to a zero win rate by making alpha move with epsilon. The lowest alpha I allowed was 0.01, rather than 0 for epsilon, since an alpha of 0 would result in no updates being made at all. Overall, this version of Q-learning only needed to play 45,000 games to converge, taking around five minutes total to play these games.

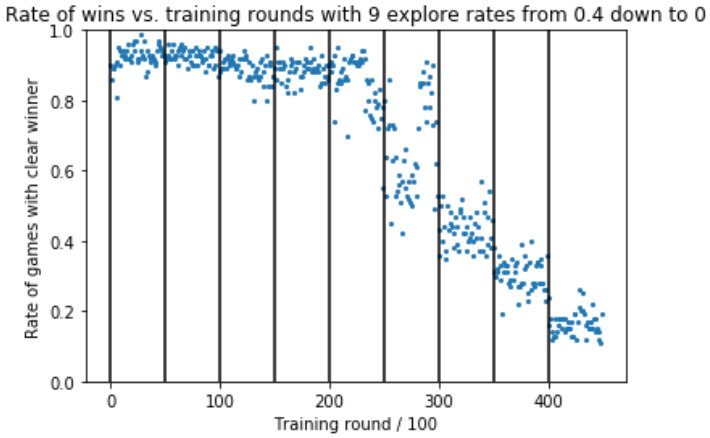


Figure 5: win rate vs. rounds of training for a dynamic exploration rate and a dynamic learning rate. Training takes about half as long and still converges.

In practice, these tic-tac-toe players play extremely well and learn extremely quickly. It's impossible for a human player to beat them once they're fully trained, and I was able to achieve these results with less than 2 minutes of training. With our end goal being Connect Four, which has a gravity restriction on moves, I decided to test how quickly Tic-Tac-Toe would train given

the same restriction on movement and with only 3 moves available instead of 9. Training ended up being around 10 times as fast, with only 4500 games necessary and with training taking thirty seconds.

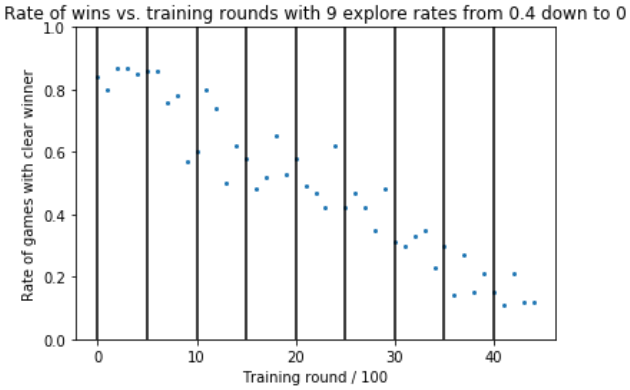


Figure 6: win rate vs. rounds of training for a dynamic exploration rate and a dynamic learning rate for Tic-Tac-Toe with a gravity restriction on movement. Trains ten times as quickly, in under four seconds on a base laptop.

These initial results were promising. However, standard Q-learning is not at all applicable to Connect Four. Even a game with a slightly larger board, a 3x4 version of Tic-Tac-Toe, is 27 times as complex, having 3 more spaces with 3 options for each space. A 3x4 version of Tic-Tac-Toe with this gravity move restriction, if space complexity directly translates to time complexity, would take around 14 minutes to train. In practice, this more complex version of Tic-Tac-Toe took around 15 minutes to train as predicted to get to decent play, especially for player 1, but didn't converge at all. Even with 100 times more games played than 3x3 gravity restricted, a fully trained algorithm for a 3x4 board, especially when it was playing as player 2, would make mistakes very frequently when played against by a human. A 0% win rate also seemed unachievable, with a 100% win rate instead being the norm. I thought that this might be due to a training fluke or some kind of meta-characteristic of this alternate game, with the larger board possibly having one or two optimal moves player 1 could make that would prevent player 2 from winning regardless of his strategy and preventing player 2 from learning at all. Randomizing the initial state of the board for a few moves at higher learning rates helped with player 2's practical performance, but the algorithm never converged to a 100% win rate.

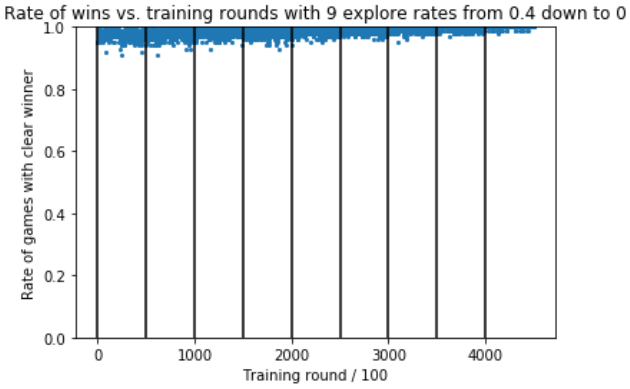


Figure 7: win rate vs. rounds of training for a dynamic exploration rate and a dynamic learning rate for 3x4 Tic-Tac-Toe with a gravity restriction on movement. Even with randomizing the initial board

state, the final algorithm has decent play, especially for player 1, but doesn't converge at all.

At this point, I decided to abandon optimizing standard Q-learning for larger games. If 3x4 gravity restricted Tic-Tac-Toe took 15 minutes to train optimally, the 200 trillion (3^{30}) times more complex Connect Four would take 3 quadrillion minutes or 5.9 billion years to train. Training such an algorithm is likely not feasible given the timeline of a college class. Additionally, if Q-learning's performance is poor for 3x4, it may be even worse for larger game sizes. Overall, the lessons from tabular Q-learning are simple enough: starting at a high exploration and learning rate and gradually reducing them over time produces an algorithm that converges to optimal play quickly and efficiently.

Part 3: Deep Q-learning

Deep Q-learning follows almost exactly the same process as Q-learning: it uses a value function *Q(S, a)* to predict how good a certain action is given a certain state and updates that value function based on receiving rewards for certain outcomes through iterative play. The crucial difference between tabular Q-learning and deep Q-learning is that rather than store the actual current values for its action-value function in a table, a deep Q-learner will use a neural network to approximate the value of a certain action when it's fed a certain state. In other words, deep Q-learning will mimic the value table with a neural network. Rather than specifically updating a value in a table, the update step of Q-learning will instead feed the updated action values back to the neural network as the "true values" of the action function and have it fit against those, using backpropagation to update the weights in the neural network. Since Q-learning also uses the reward from the next action to compute the true value of the current action, I also stored the value of the next action given the state produced by the other player's move. The fundamental assumption here is that each player always makes the best move available to them.

It's straightforward to see why deep Q-learning is preferred for more complex games. For one, not every state needs to be discovered to assume an action given a state. In tabular Q-learning, literally every single state needs to be discovered and stored in a table to have a certain value associated with it. A deep Q-learner can approximate the value of a certain state based on states it's seen before that might be similar to this one. It therefore doesn't need to iterate through every possible state like tabular Q does. It might also be able to determine a value

for the current state based on elements of the current state (for example, a player already having 2 in a row for Tic-Tac-Toe) that it's determined to be important in other states.

I decided to limit my exploration of deep Q-learning to gravity-based games. This allowed me to simplify the logic of my neural network (rather than having to specify a row and a column, the network need only specify a column) as well as spend time generalizing my network for games that were similar to my target game. Additionally, in the preliminary testing I did, the smaller number of possible moves presented by gravity-based games improved convergence.

I also decided to keep my neural network architecture relatively simple and similar for most games. For my overall architecture, I decided on a convolutional neural network. Convolutional neural networks are most frequently applied to image recognition, frequently found in photos apps and self-driving cars. They're also used in game playing — some techniques for playing the 8x8 game Othello involve using convolutional neural networks as best next move predictors (Liskowski et al., 2017). Rather than a neural network entirely comprised of fully connected layers, a convolutional network will proceed these layers with a convolutional layer of "filters", or small windows that will move over an image attempting to detect certain features that have been deemed important. One filter might, for example, slide over an image looking for lines, outputting a certain signal when a line is detected. Several filters might be used to find different features, and multiple convolutional layers might also be used, with filters in later layers finding more complex patterns based on the outputs of previous filters. Convolutional neural networks are useful for image processing because of this feature-finding ability. A convolutional neural network is therefore good for playing two-dimensional board games where specific patterns on the board are important for determining a player's next best move.

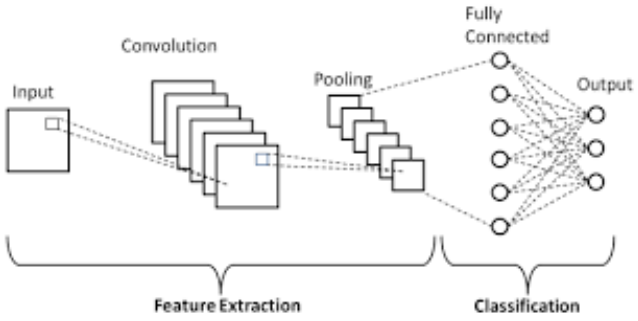


Figure 8: visual representation of a convolutional neural network. Sourced from ResearchGate.

I decided to use a small number of filters to start, increasing the number of filters only if necessary. I also decided to use only one convolutional layer, as patterns are not likely to be so complicated as to need extremely heavy processing. I decided on a filter size equal in height and width to the number of pieces in a row required to win. For example, in Tic-Tac-Toe, 3 in a row is required to win, so I used 3x3 filters. In Connect Four, which requires 4 in a row to win, I use 4x4 filters. Often, CNNs will pad the input image on either side with zeros so that the entire image can be processed by every part of every filter. I decided to use a small amount of padding on all sides of the input board so as to not lose any complexity. I followed this convolutional layer with two fully connected layers. The number of neurons in these fully connected layers was equal to the overall number of spaces in the board. I figured that larger boards would likely require more complex networks, and the size of the board itself would be a good generalization for complexity. In my preliminary testing, I experimented with smaller fully connected layers, but architectures with fewer neurons in fully connected layers only performed well for a 3x3 board.

Finally, I investigated having the network play against itself, rather than play against a second network. Valuable patterns on the board will be valuable for both players (for example, player 2 and player 1 would both benefit from moving in a spot where player 1 needs to go to win). In practice, I found that having the algorithm play against itself led to the model converging far more often. However, in reality, having the algorithm play against itself just led to having consistently very high values for certain columns, regardless of what the board looked like. This often led to very poor play, even against a human and even against moves that the algorithm didn't expect. Additionally, having the same network predict values for both players doesn't work in theory either — some spaces are more important for strategy for some players.

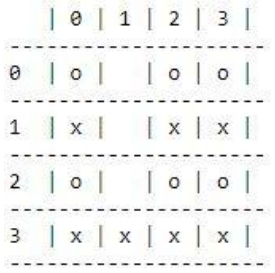
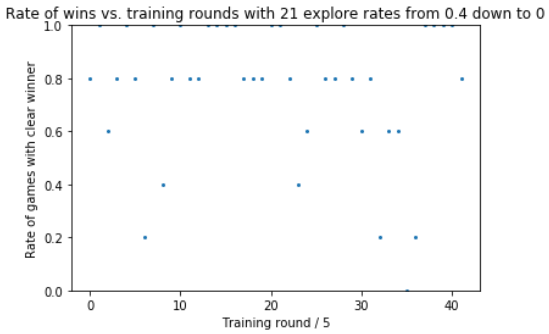


Figure 9: a typical game when DeepQ is allowed to train against itself in full.

Developing the deep Q-learning version of a board game player ended up being a very buggy, iterative, and frustrating process. Training just 500 games took around one hour, with that same number of games taking only three seconds for tabular Q-learning. Even late into development and after I had trained multiple trials for different board sizes, I was plagued with a torrent of NaN values and discovering issues with my code that stemmed from bugs as insignificant as 32-bit vs. 64-bit numpy floats. I also unfortunately had issues with TensorFlow and memory leaks, so I had to constantly save and reload my policy rather than just training my network. Even though many of the trials I'll show results from below come from buggy code, I'll still try to explain them as best as I can.

My first finding was that the number of games necessary to play is far shorter for deep Q-learning than it is for tabular. The fastest version of tabular Q-learning was with a gravity-based 3x3 board with both a moving alpha and epsilon. That took 4500 games to train properly — deep Q-learning achieved the same result in 210 games. Furthermore, training time doesn't necessarily need to exponentially or even linearly increase with board size. I was able to achieve convergence (that is, a 0% win rate and perfect play resulting in ties every time) for a 3x4 board with only 2.5x the training time I needed for a 3x3 board.




```
   | 0 | 1 | 2 |
-----------------
0  | o | o | x |
-----------------
1  | x | x | o |
-----------------
2  | x | x | o |
-----------------
```

Figure 10: training DeepQ for 3x3. Moves to a good mix of wins and losses faster than tabular does, and converges to tying. Also shown is an example game. Good at learning aggregate strategies.

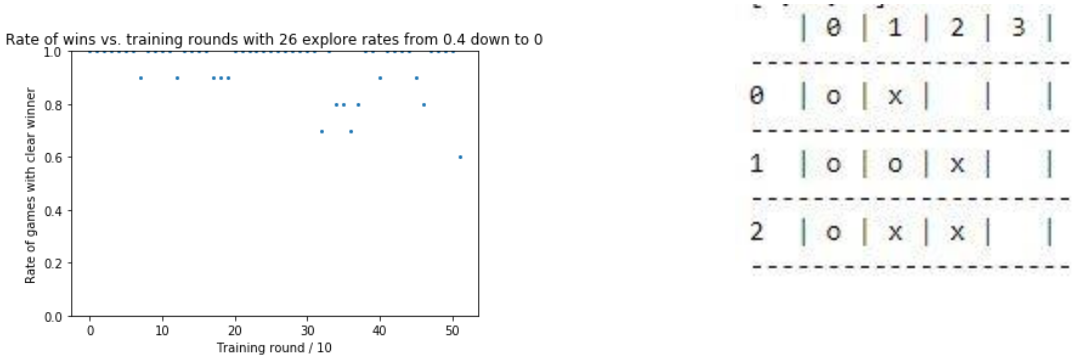


Figure 11: training DeepQ for 3x4. Actual gameplay is not the best, but can learn general good strategies. More training time than this produced better results similar to the 3x3 results.

It seems that rather than game complexity being taken care of by training time as in tabular reinforcement learning, the complexity of games in deep Q-learning can be taken care of by the complexity of the network we use. The architecture I settled on was two fully connected layers each having a number of neurons equal to the size of the board, preceded by a convolutional layer with multiple filters each equal in square size to the number of pieces in a row necessary to win. The complexity of each game is built into the architecture of the neural network that learns that game. This bodes well for scaling up our board size — we won't need billions of years to train our model, but rather only a few hours.

My second finding was that it's important to not use an excessive number of convolutional filters. The above graphs both show a network with 4 convolutional filters. When testing the same exact parameters and model for 3x3 and 3x4 games, but using 8 filters rather than 4, I wasn't able to get convergence. More training time didn't help either. With more filters, what did end up helping was a longer training time at a lower learning rate. Training for longer and starting training at a lower learning rate / exploration rate seemed to more gently ease the filters into values that played well. Fewer filters did better with a higher learning rate because they were fundamentally more coarse, picking up on broader patterns, whereas more filters picked up on finer patterns but needed a lower learning rate to do so. Overall for deep Q-learning, a lower learning rate than what I used for tabular performed generally better as well. At most, I used a learning rate of 0.25, around half of what I would use for tabular Q-learning.

Given my findings about how more filters performed with lower learning rates, I wondered whether or not I should square my loss function. Squaring the loss function (with

learning rate included) would result in very large initial updates to the network, followed by smaller updates that would last for longer. In other words, square loss would let the network learn rapidly at the beginning and get to policy values that were generally good, then fine-tune its strategy over a long period of time. Square loss also helped significantly for higher numbers of filters. Again, this was with code that had a significant number of bugs, so I can't really speak to the veracity of my results, but these were general trends I observed across trials and across different versions of my code with varying degrees of bugginess. Square loss highly prioritizes large loss functions and makes lower losses less significant in the training process, providing a better approximate solution and preventing overfitting. I also made sure to preserve the direction of the loss function after it was squared. Overall, for fewer filters, square loss performed worse, while for more filters (~16), square loss performed better.

I also learned, through trial and error, the importance of the reward scheme I provided. I decided to give player 1, the first mover, a high reward for winning, a mild penalty for losing, and a low reward for tying. Meanwhile, I gave player 2 a higher reward for tying, the logic being that player 2 is at a disadvantage because it moves second and performs well even if it ties. Player 1 should be penalized for tying because it has an advantage from moving first and should be able to win more frequently. This reward scheme, however, often resulted in player 1 making Hail Mary plays, deciding not to block player 2's path to victory likely in hopes that player 2 would make a mistake. Rather than block player 2 and only be left with multiple ways to tie, since it was mostly indifferent between tying and losing, player 1 would often not block player 2 in favor of advancing its own position on the board. Player 1 also tended to learn more quickly, with its update sizes falling faster than player 2's.

Finally, toward the end of training, a danger of overfitting presents itself. Usually, the loss function we train on will be consistent and declining. However, towards the end of training, occasionally the loss function, and as a result the updates we make to the neural network, will skyrocket. I don't really know what's causing this beyond my learning rate being too high even when the network has mostly converged on good values, but this produces the same problem that training the network against itself does: it develops a strong preference for certain columns over others and only moves in those columns. To combat overfitting, I checked the size of each update to the network against the average of the last 100 updates. If the current update was much higher than the last few updates (I settled on 1.4x), I stopped training. This code ended up being not

very reliable, since updates would start going up only once overfitting had already started, so I would try to stop training before overfitting began. I also tried to prevent very large updates that would balloon out of control early in the training process by setting a high cap on all updates. Even with all of these controls, overfitting still plagued my training process, especially for smaller games. I ended up getting rid of the control of updates being larger than recent updates, as it ended up doing more harm than good. For larger board sizes, more filters, and lower learning rate starts, it didn't seem to be as much of an issue. I noticed that overfitting was mostly an issue at very low learning rates, so I also decided to end at a larger learning rate (previously I ended at zero, but I decided to end at around 0.05). Finally, I also noticed that overfitting wasn't as much of an issue with a sigmoid output activation function vs. ReLU. I switched to sigmoid for many of my later trainings.

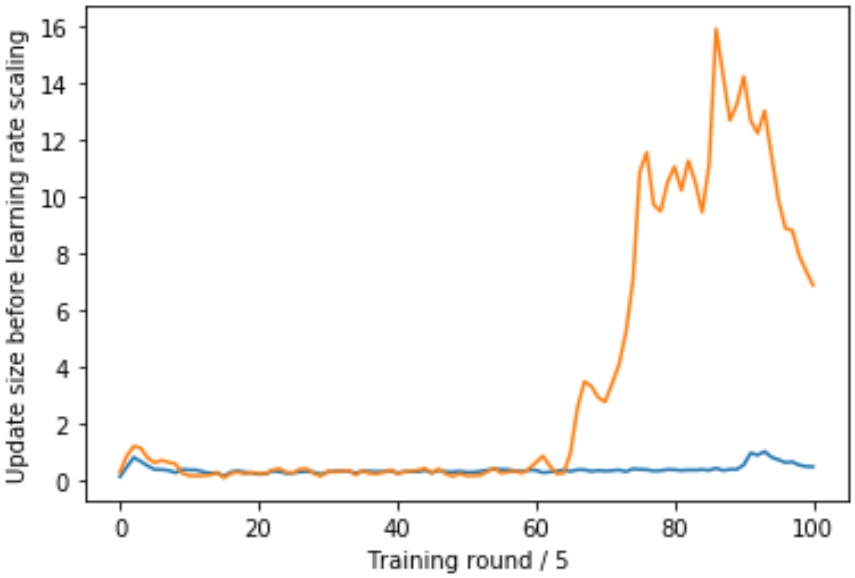


Figure 13: an example of overfitting for a 3x3 game. Updates skyrocket and balloon in the wrong direction, leading to even bigger updates in a vicious cycle of divergence. The axis label is wrong — this is after learning rate scaling.

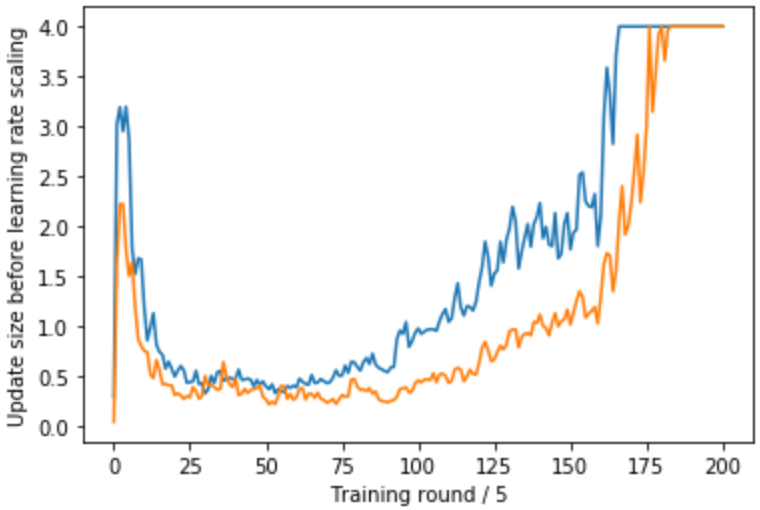


Figure 14: an example of overfitting for a 3x4 game. After around 50 rounds of training, the algorithm starts strongly diverging from a good policy. The axis label is wrong — this is after learning rate scaling and is the actual update made to the network.

```
    | 0 | 1 | 2 |
- - - - - - - - - - - - -
0   |   |   | x |
- - - - - - - - - - - - -
1   | o |   | x |
- - - - - - - - - - - - -
2   | o |   | x |
- - - - - - - - - - - - -
```

Figure 14: gravity-based gameplay resulting from overfitting. Both players develop a strong irrational preference for a particular column.

Additionally, I settled on moves to win as a new success metric. Win rate or tie rate may not be the best metric for success, as both players might play perfectly and one might still lose. Ideally, though, if a player is doing well, they'll make the game last as long as possible. A longer-lasting game means both players are doing better at blocking the other's strategies as well as implementing longer-term strategies. If, for instance, a Connect Four game lasts for 20 moves instead of 7, both players are probably playing better and thinking more strategically.

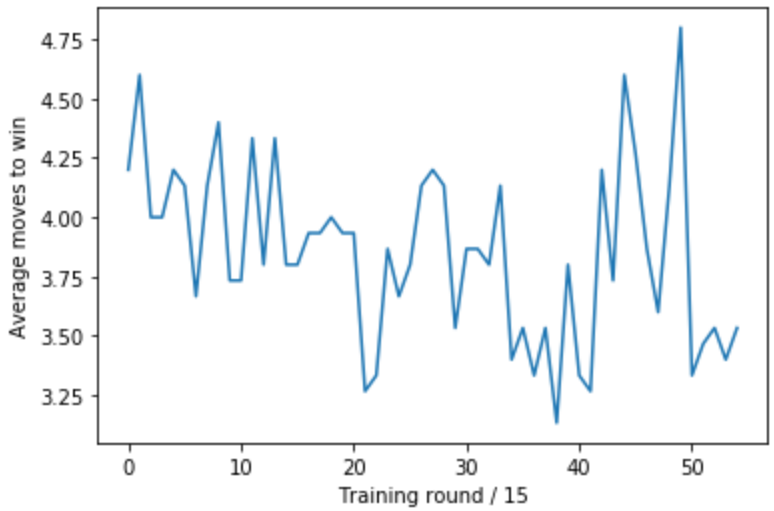


Figure 15: moves to win vs. training round for 3x4.

Beyond the issues I've described above, deep Q-learning, I discovered, is not overall the most robust method in the world. Most of my training iterations were highly sensitive to even small differences in learning rate and exploration rate. Additionally, different code runs with the exact same parameters could produce wildly different results. The outsized impact of randomization on results apparently plagues the entire field of deep Q-learning (see Clary et al). Reproducibility is difficult in the field, and most reinforcement learning researchers report aggregate statistics over multiple runs with different random seeds. Unfortunately, since each game played for my deep Q-learning method took so long to run, and since I was debugging so late into the process, I wasn't able to get multiple runs for each board size with my finalized code. I unfortunately wasn't able to parallelize any of my code either, as most of the logic runs in serial, iterating through games one after the other.

Overall, I would characterize this project as a difficult development experience. I built

working players for 3x3, 3x4, and 4x4 gravity games, but decided not to test any larger game sizes. Most larger game sizes required playing with learning rate, exploration rate, and loss function individually, and I didn't have the time or mental energy for that. The most significant finding I had was that deep Q-learning performs worse overall than tabular Q-learning for smaller games where it's unnecessary, but better for larger games where it is necessary. After building a 4x4 player that performed very badly, I moved on to Connect Four.

Part 4: Connect Four

Given how sensitive DeepQ is to even slight changes in randomization, learning rate, exploration rate, and training time, and given how prone it is to overfitting or training poorly, I was a bit lost when trying to determine the best parameters for my Connect Four player. More filters seem to work better for larger games, but how many filters? A lower learning rate with longer training time works better for larger games as well, but too low of a learning rate / explore rate seems to lead to overfitting. What do we do?

I decided to start with an exploration of rate of 0.5 with 16 convolutional filters and square loss. This seems like a decent starting point based on everything we've learned so far. Let's see how we did!

```
   | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
---------------------------------
0  |   |   |   |   |   |   |   |
---------------------------------
1  |   |   |   |   |   |   |   |
---------------------------------
2  |   |   | x |   |   |   |   |
---------------------------------
3  | o |   | x |   |   |   |   |
---------------------------------
4  | o |   | x |   |   |   |   |
---------------------------------
5  | o |   | x |   |   |   |   |
---------------------------------
```

Figure 15. The strategic brilliance of our bot.

Now, you might look at this game and ask whether the network has learned at all. However, I posit that our network actually did its job well. Given the fact that both players have explored pretty frequently throughout training, this is actually a stable strategy for both of them. If player 1 had gone anywhere other than in just one column for any of its moves, player 2 would have won. Otherwise, player 1 wins. The two players are just locked in a game of chicken with each other, forgoing actual strategy in favor of this death race that they've both deemed optimal. This strategy works very well for both players at large exploration rates and seems to hold over to lower exploration rates which come with a low learning rate — by the time player 2 needs to update its play so that player 1 doesn't win all the time because neither are exploring at this point, it doesn't have a high enough learning rate to do anything. Additionally, since gamma exists and the network discounts rewards from wins that happen later, our algorithm prefers sooner wins. I experimented with giving agents for smaller boards shorter training times — this behavior also exists in smaller board sizes with lower training times, but is trained out before we notice.

How do we combat this? The neural network might think it's playing well, but a human can clearly easily beat this. For smaller boards, the solution is just more training time, but since our training times are likely to already be very large and this still happens, this probably isn't a solution. My solution was to randomly initialize the first few moves for each player, making a number of random moves roughly based on the exploration rate. I decided on making a number of random moves normally distributed around board size times exploration rate over 3, so that each game would be entirely distinct and both players could practice against players that made more random moves and were worse than them. The networks need to be not just good at playing against each other — they need to be good at playing against other players with different policies, both good and bad. I also decided to penalize players winning in only the number of moves needed to stack 4 in a row. If each player needs 4 in a row to win and one player wins after seven total moves, clearly both players are doing something wrong and should not be rewarded. I gave a slight penalty to both players for this outcome. Now, both players want to win, but they have to do it in a somewhat smart way.

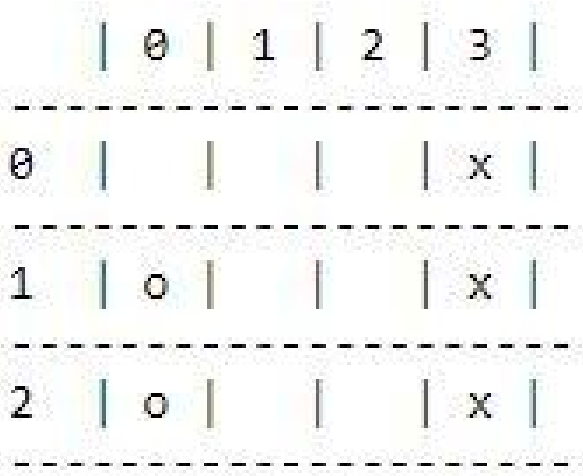


Figure 16. The same stacking behavior exhibited on a 3x4 board trained for <200 games. Row 1 is filled first, followed by the next two rows. Both players take advantage of the other making random moves to make this strategy tenable. This behavior was trained out with more training time as more complex strategies were learned.

I also decided to make one more significant change, which was modifying the activation function for my output layer. Previously, I had been using ReLU as my activation for every layer except for my output, which was linear. However, now reflecting back on my work, I think this was a poor choice. I want to have my output layer be similar in scale to my rewards. I want to avoid a situation where the network determines a move to be very bad and have strong negative value, but that move is actually good and the network can never learn it because it has such a strong negative value. Furthermore, I also want to discourage very large values for some actions. Having sigmoid as the activation function for the output layer, which would restrict its value between 0 and 1, should also increase flexibility somewhat and prevent the network from thinking a move is bad forever. I think linear activation was fine for smaller games, but output values for a much more complex ReLU network for Connect Four are likely to be very large without a sigmoid transformation. Having sigmoid activation should also help combat overfitting somewhat, since update sizes are forced to be relatively small and scaled appropriately for an output that can only be between 0 and 1.

After quickly training our fixed Connect Four implementation (only 120 games), how does it do?

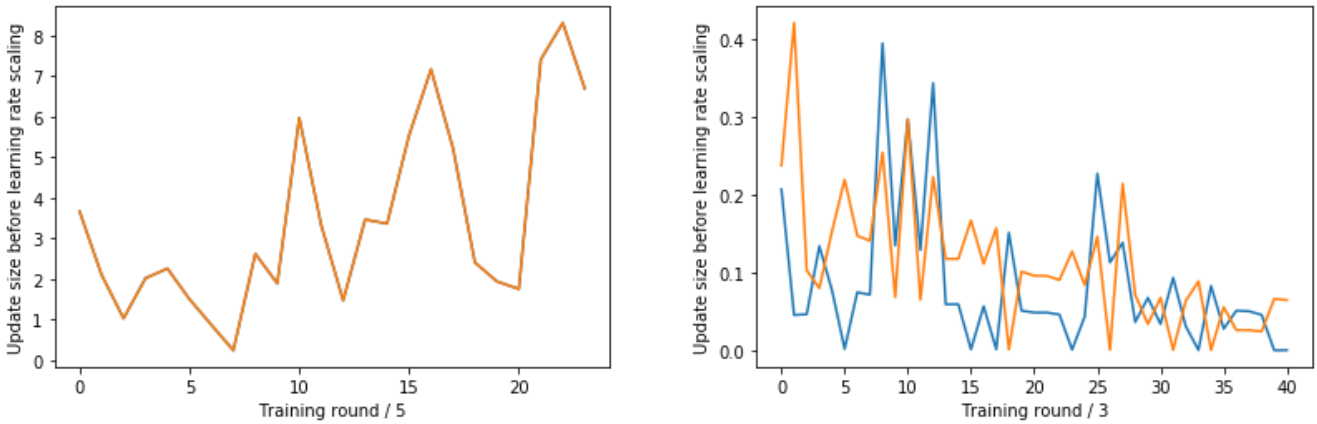


Figure 15: on left, update size vs. training round for linear output and no overfitting controls, trained against self. On right, update size vs. training round for sigmoid output and overfitting controls.

Clearly, our controls are helping our update sizes not balloon out of control. Play is still not good, however. The algorithm isn't playing chicken with itself *all* the time, but it still is sometimes, and otherwise it's not playing very well. It also sometimes tries to stall by randomly moving and then just stack pieces on top of each other. This way, it can avoid a penalty if we decide to throw one at it.

```
   | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
-------------------------------
0  |   |   |   |   |   |   | o |
-------------------------------
1  |   |   |   |   |   |   | x |
-------------------------------
2  |   |   |   |   |   | x | o |
-------------------------------
3  | o |   |   |   |   | x | x |
-------------------------------
4  | o |   |   |   |   | x | o |
-------------------------------
5  | o |   |   |   |   | x | x |
-------------------------------
```

Figure 16: when we penalize winning in very few moves, the algorithm will often just stall by making junk moves and then play chicken with itself.

Something else I noticed about the training process is that most updates happen to the losing player. It seems as if the network thinks it's pretty good at predicting action values for the winner, and bad at predicting action values for the loser. It seems to punish bad play rather than rewarding good play. This likely means that I have to wait for the network to happen upon good moves through exploration because it only punishes its own bad moves. The opposite is true of non-squared loss — when we don't square our loss, the biggest updates happen to the action values for winning moves.

I increased the penalty for winning very quickly and doubled training time. I also changed the reward scheme for winning so that rather than penalizing winning before a certain number of moves, we give less of a reward for earlier wins and gradually more of a reward for wins that take more moves. This should incentivize blocking the other player from winning and should also encourage building up longer-term strategies.

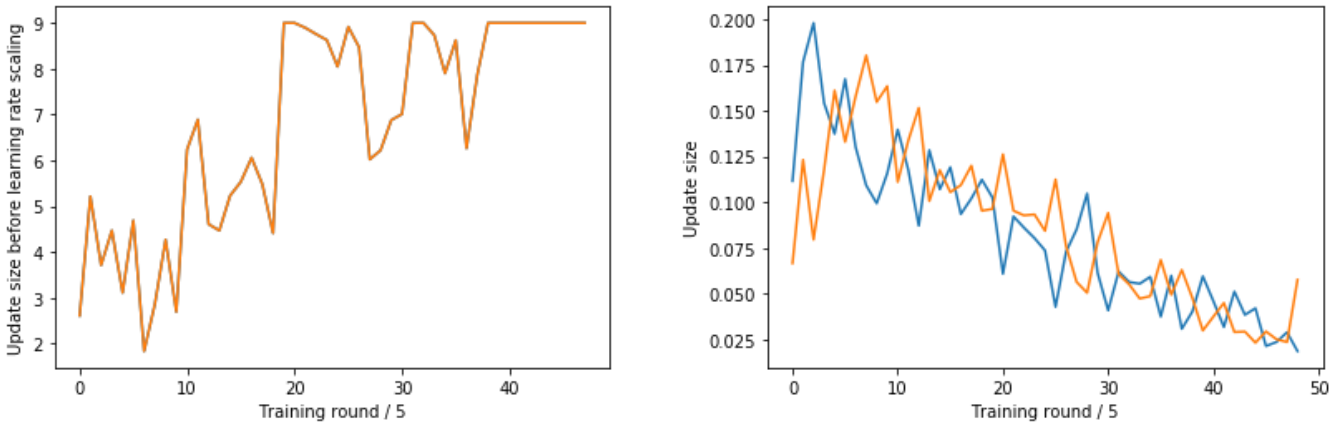


Figure 16: on left, update size vs. training round for linear output and no overfitting controls, trained against self. On right, update size vs. training round for sigmoid output and overfitting controls.

The algorithm still plays badly. It's still just stalling to wait as long as possible until its rewards are big enough, then stacking on top of itself until it can win. I think the solution may just be more training time for the algorithm to try more strategies. I also think more filters may be helpful in this case, so I'll bump the number of filters up to 32 from 16 and slightly decrease the starting learning rate while increasing training time. At this point, I'm training 1000 games,

which is taking half an hour.

The success metric I'll be using from now on is the number of moves to reach a terminal game state. Ideally, as many games as possible will be played before the game is terminal. For 1000 games played, we can see a disjoint where the algorithm learns how to stall when we only use a penalty. I also decided to increase my gamma significantly, from 0.9 to 0.98, so that the algorithm learns better how to plan ahead.

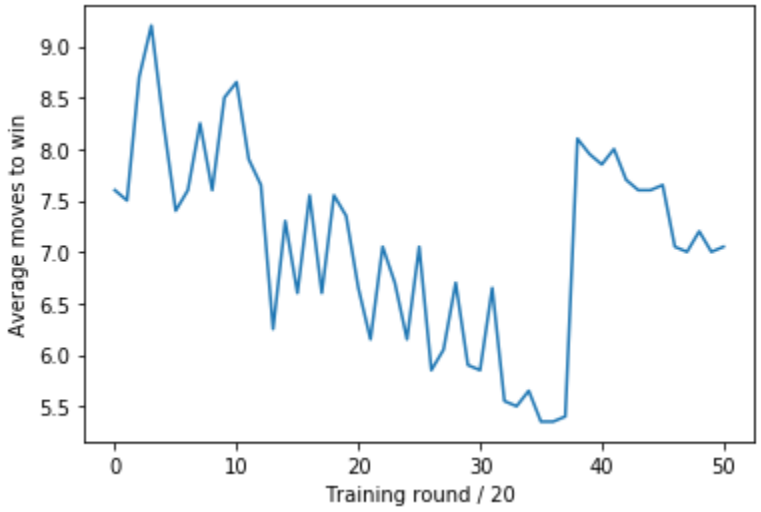


Figure 17: number of moves to win for a neural network that trains for 1000 games with a strong penalty for winning in <=5 moves. The disjoint where the algorithm learns how to stall is very apparent.

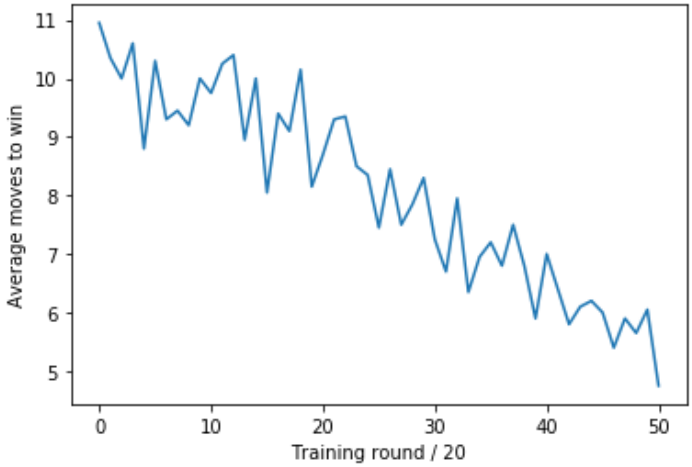


Figure 18: number of moves to win for a neural network that trains for 1000 games with a more moderate penalty and a reward for winning later.

This doesn't seem to be going well. For my final test, I decided to run both 10,000 games with just more reward for winning later and not much random initialization, and 2,500 games with the same reward scheme, more random initialization, a higher learning rate, and a greater penalty for winning too early. These will take around 5 hours and 2 hours respectively and are pretty much the maximum number of games I can run without my laptop giving out. The 10,000 game strategy without much initialization overall performed poorly, overfitting to the "playing chicken" strategy where it can win in 4 moves by just stacking. The 2,500 game strategy ran into a vanishing gradient problem in the output layer where the update sizes for the losing player in a given match all became tiny even though the network wasn't performing well. The winning player wasn't able to make an impact on the action values since they were all stuck so close to zero and were so far down the left side of the sigmoid function, and as a result the agent didn't learn.

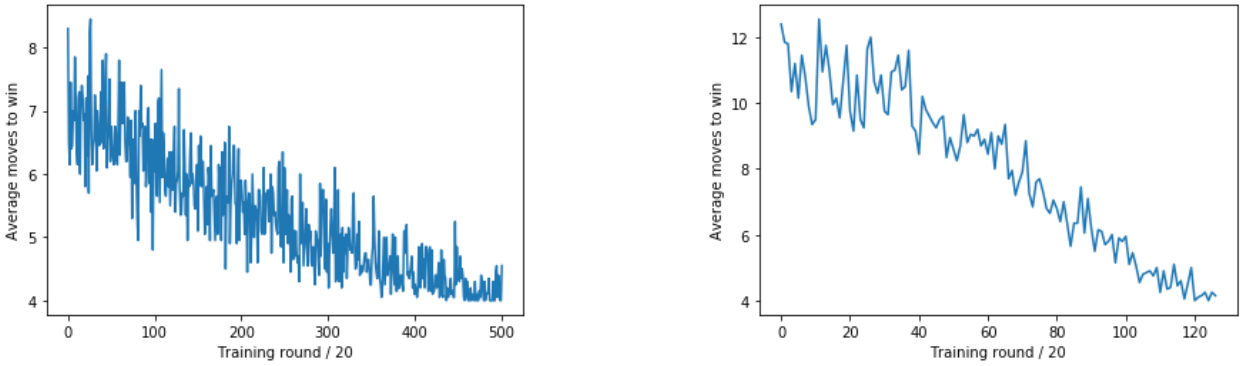


Figure 19: num moves to win for the 10,000 round connect four without much random initialization. Also num moves to win for the 2,500 round connect four with a lot of random initialization and a strong penalty for winning early.

Even though we heavily penalize very early wins in the penalized network, we still end up converging to the bad strategy. My guess is that the algorithm still first learns that stacking pieces in the same row is a way to win, skips any move in between, and then needs to be trained out of just stacking pieces until it wins. This is true whether or not we heavily penalize winning quickly by just stacking. Overall, penalizing our network for playing a certain way feels too

hard-coded and doesn't work anyway. It seems that the only way to stop cheating the reward scheme is by increasing training time.

Conclusions

Overall, I had a much better experience with tabular Q-learning than with deep Q-learning. While tabular Q-learning is very straightforward and clearly works very well for small games, deep Q-learning is much more esoteric and obscure, and it was very difficult to understand what was going on a lot of the time. Furthermore, while tabular Q-learning learned good strategies against everyone, deep Q-learning basically only learned good strategies against itself, and if it found a strategy that it could use to exploit the rules or reward scheme, it would continue using it and ignore all other strategies. Trying to get deepQ to behave properly ended up being futile, but if I had had more training time, I would have been able to get Connect Four to work. I was able to get smaller games to learn and train properly, and maximum training time for Connect Four would probably be in the range of a few days, but in the end I wasn't able to accomplish my main goal of getting a working Connect Four player. AlphaZero had to train for 13 days on the best computers in the world to become a good Go player, so this situation should have been expected.

All code can be found at https://github.com/maxmarcussen98/ConnectFour

Works cited:

- Zhang, J. (2019, August 24). *Reinforcement learning - implement TicTacToe*. Medium. Retrieved October 13, 2021, from https://towardsdatascience.com/reinforcement-learning-implement-tictactoe-189582bea542.
- Friedrich, C. (2018, July 20). *Part 3 - tabular Q learning, a tic tac toe player that gets better and better*. Medium. Retrieved October 13, 2021, from https://medium.com/@carsten.friedrich/part-3-tabular-q-learning-a-tic-tac-toe-player-that-gets-better-and-better-fa4da4b0892a.
- Alderton, E., Wopat, E., & Koffman, J. (2019). *Reinforcement Learning for Connect Four*. Decision Making under Uncertainty. Retrieved October 13, 2021, from https://web.stanford.edu/class/aa228/reports/2019/final106.pdf.
- Wisney, G. (2019, April 28). *Deep reinforcement learning and Monte Carlo tree search with Connect 4*. Medium. Retrieved October 13, 2021, from https://towardsdatascience.com/deep-reinforcement-learning-and-monte-carlo-tree-search-with-connect-4-ba22a4713e7a.
- Stefan Voigt. (2020, April 14). *Connect four - deep reinforcement learning*. Stefan Voigt. Retrieved October 13, 2021, from https://www.voigtstefan.me/post/connectx/.
- van der Ree, M., & Wiering, M. (2013). *Reinforcement Learning in the Game of Othello: Learning Against a Fixed Opponent and Learning from Self-Play*. University of Groningen. Retrieved October 14, 2021, from https://core.ac.uk/download/pdf/148305764.pdf.
- Liskowski, P., Jaśkowski, W., & Krawiec, K. (2017, November 17). *Learning to play othello with Deep Neural Networks*. arXiv.org. Retrieved October 14, 2021, from https://arxiv.org/abs/1711.06583.
- Clary, K., Tosch, E., Foley, J., & Jensen, D. (2019, April 12). *Let's play again: Variability of deep reinforcement learning agents in Atari Environments*. arXiv.org. Retrieved December 2, 2021, from https://arxiv.org/abs/1904.06312.
- Learning to play Connect 4 with Deep Reinforcement Learning. Codebox Software. (2020, March 8). Retrieved December 2, 2021, from https://codebox.net/pages/connect4.