Solving Random Mazes using Asynchronous Deep Reinforcement Learning

Aaron Brown

August 30, 2016

Udacity Machine Learning Capstone Project

Project Overview

In this project we will explore how convolutional neural networks along with reinforcement learning and asynchronous training can teach an agent to solve any randomly generated maze. The idea of using a deep learning network was motivated by the work of Deep Mind with their paper on "Playing Atari with Deep Reinforcement Learning" [1]. The idea of exploring maze environments was motivated by previous work in the field of Micro Mouse Robotics Competition. Reinforcement learning is a very powerful way to teach an agent an optimal policy through trial and error when some reward feedback system is prevalent, however reinforcement learning in the traditional sense of storing off every possible state in a look up table becomes extremely impractical. Take for example an agent trying to find the shortest route to any number of randomly generated mazes. For the agent to do well at this problem it really needs to know as much about the maze as possible, such as all of its row and column positions, along with where the agent its self is located in the maze. However, if the maze is simply 5x5 dimensions then there are 20 rows that could either be present or not, likewise there are 20 columns that could be present or not as well. If we wanted to store all the combinations for just these 20 row, and 20 column states we would need a state table with 2^40 slots, not to mention storing the agent's states which would be another additional factor of times 25. In total this would equal around 3.44 terabytes of memory and this is just for a simple 5x5 maze, needless to say this is completely impractical.

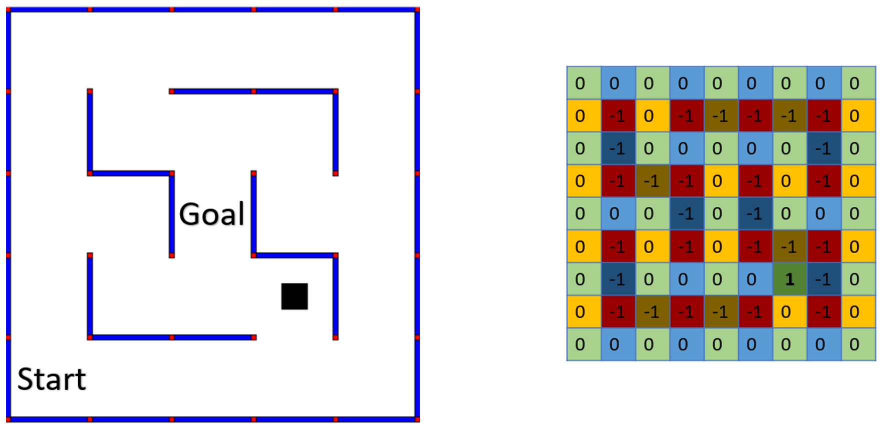
Problem Statement

We wish to get the agent to always pick the action that will move it closer to the center goal, for any number of random mazes while allowing the agent to know as much about its current maze environment as possible. For this project we will focus on mazes that are just 5x5 dimensions, knowing that the same techniques could be used for any size maze. Also the start will always be at the bottom left of the maze and the goal will always be in the direct center of the maze. It would be great to somehow still be able to use the reinforcement learning model that is able to perform extremely well when provided the sufficient state space variables. For instance, using a reinforcement learning model that uses a look up table Q[state][action], basically storing a value for every possible state action pair, and updating the values based on its immediate reward and some discounted value of its future maximum expected value, the agent is able to master a single random maze after sufficient training. For the agent to find the optimal path in a single random maze all it needs to know and store off is its possible locations which in a 5x5 maze is just 25 possible states. Since in this case the maze is static along with the goal, the agent has all the information it needs to train a policy to get to the goal as fast as possible with just the 25 possible locations and using a reward system that adds a very small negative reward for each step and then a relatively large reward when reaching the goal. If the maze is no longer static however but constantly being regenerated with the goal still being in the center upon every completion the agent makes, then the 25 location combination positions by themselves become completely useless. The agent would simply get stuck going back and forth between cells having no idea anymore which way it really should go.

For exploring our problem, the 5x5 maze worlds will be generated by using a random path creator through an initially completely filled maze. Until all the maze cells are visited the generator will travel to new cells, cutting down walls in the process. If the generator gets stuck in a dead end, then it would move back to the last cell it was at where it had a choice to branch. The maze space will be fed into a network and over time with applying reinforcement learning the network will learn which way to push the agent through the maze so that the agent gets to the center in as few moves as possible from its starting position. The benchmark for performance is that after a long enough amount of training time, the agent will be able to solve any randomly generated 5x5 maze in the least number of steps possible at least 90% of the time. Since 90% is a high percentage of the time it seems like a good metric or benchmark to strive for and at the same time realistic to reach.

Implementation

So the situation we have is reinforcement learning is a very powerful way to teach an agent an optimal policy like solving a single random maze, but if the agent needs to learn to solve any number of random mazes we are stuck dealing with state environments with extremely high dimensionality. Fortunately, there is something we can still do that uses all the state combinations that we need along with using reinforcement learning, the answer is with function approximation. One example of a great function approximator is a convolutional neural network which are frequently used in high dimensionality situations such as image recognition. A neural network by its self is actually a great function approximator as well, the only difference with using convolution is that certain sections learn to look for certain distinct features along sampled smaller input windows. Also while the overall input image area gets pushed smaller, its depth gets pushed out larger, thats why it’s called *deep learning*. This nature of picking out small windows of features to focus on makes convolution great for working with images. Also our problem works very well for image states where we can define inputs as 2D values arrays. Also the best action input is completely independent of any other previous states, this is because the value image captures all the maze features. If we were dealing with a more sequence dependent state space an RNN might be reasonable, but for our requirements a CNN will meet all our needs. Connecting to the convolution network is a regular fully connected neural network which has the output of a single array of, in this case 4 values each representing the possible actions the agent can do. The idea is some 2D matrix representing the maze and positon of the agent is fed into our network which outputs an action array of 4 values, and the action with the highest value is the action the agent follows, then the new state is observed and the network action policy is repeated until the agent arrives to its goal in the center of the maze. Figure 1 below shows what the possible simplified 9x9 2D input could look like for a certain 5x5 maze. Here the agent is the black box, represented by a 1 in the 2D input, any wall is represented by -1 as well as pegs, and open spaces whether they are unfilled rows/columns or spaces that the agent can travel to are 0s. Notice that the 2D input consists of 81 variables, but actually we only care about 65 variables for a 5x5 maze, that includes all the row/column and agent locations. In practice its usually always the case that convolution networks work with square inputs, by including the red peg locations in our 2D input it helps format the input into a square, however note that the pegs are actually static variables that don’t provide any actual additional information. Some additional notes are since the start and goal are always in the same location, we don’t need any special distinct markers for them in our 2D mapping. Also we could extrapolate and use this maze to 2D input mapping for any nxn maze we wanted.



0	0	0	0	0	0	0	0	0
0	-1	0	-1	-1	-1	-1	-1	0
0	-1	0	0	0	0	0	-1	0
0	-1	-1	-1	0	-1	0	-1	0
0	0	0	-1	0	-1	0	0	0
0	-1	0	-1	0	-1	-1	-1	0
0	-1	0	0	0	0	**1**	-1	0
0	-1	-1	-1	-1	-1	0	-1	0
0	0	0	0	0	0	0	0	0

Figure 1: On the left shows a 5x5 maze being mapped to a 9x9 2D matrix input on the right.

Google's open source machine learning library, Tensor flow is a great way to operate on neural networks since it contains built in tools to initialize everything and even train the network. Training the neural network actually uses the same type of training that was being done with reinforcement learning, updating an action state pair to the reward gained from doing the action and adding the discounted maximum value from the new state. Incorporating this type of updating with a neural network means defining some sort of loss function that we will attempt to minimize through gradient descent. The loss function in this case can be defined in terms of what has been repeated for reinforcement learning's update function. We can take the agents current state, a 2D input mapping and do a feed forward pass through the CNN, convolutional neural net, and observe it's 4 action output array. From here we have a choice to take the highest action value from this output which is supposed to be the best action but in early stages of training that's not actually true at all and we might be better off choosing a random action instead (it was actually observed that some random exploration is VERY IMPORTANT or we could get stuck in update loops). In any regard we try the either the greedy (highest valued action from the array) or a random action and observe the reward gained, and new input state that the agent is now in after taking that action. The new input state is then fed into the CNN and the maximum value is the highest value from that output array. The Loss equation is then the difference between the value the CNN was outputting for the action explored vs the observed transition reward plus discounted maximum value. The state's other action values, all the others except the one we picked to explore are to stay the same. Equation 1 summarizes this process below.

$L_i(\theta) = .5(Q(s, a{:}\,\theta_i) - [r - \gamma \max_{a'} Q(s', a'{:}\,\theta_{i-1})])^2$, for action *a* explored. *(1)*

$L_i(\theta) = .5(Q(s, a{:}\,\theta_i) - Q(s, a{:}\,\theta_{i-1}))^2$, for all other actions not explored.

Using this defined loss function, we can slightly change the weights, θ of the CNN at each iterative update for a sufficient amount of runs until CNN becomes a very good approximation to Q'(s,a), our optimal strategy policy mapping any state to some most valued action. The updating process its self is a matter of using calculus. The Loss function being defined by the CNN weights map out some high dimensional topology in space, we wish to find weights that minimize the calculated Loss value and by looking at the opposite direction of the partial derivative of the Loss function and setting the weights in a small step in that direction we have an idea how to make the Loss value smaller each iteration.

We briefly talked about how the CNN will be set up, but next we will get specific about the exact architecture the network will follow. Figure 2 below outlines the overall network we will be dealing with as well as illustrating how the CNN will reshape our data using shared weights.

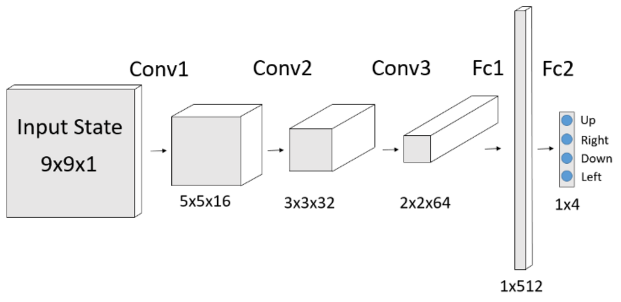


Figure 2: Convolutional Neural Network taking input in on the right, and outputting 4 action values. The Network operations, from left to right is Conv1, Conv2, Conv3, Fc1, and Fc2.

In the figure, we can see the different output sizes on the data, though this picture does not capture the dimension sizes of the operations. For a more complete picture of the number of weights that we are storing in each operation transformer, here is the break down. Conv1 network is [5, 5, 1, 16] with biases size 16, Conv2 network is [3, 3, 16, 32] with biases size 32, Conv3 network is [2, 2, 32, 64] with biases size 64, Fc1 is [256,512] with biases size 512, and lastly Fc2 is [512,4] with biases size 4. In total this amounts to 146,948 weights. If each weight is a 32-bit float, then this is equal to about 588 kilobytes for the memory of the network. 588 kilobytes of memory is large but very practical to work with, making it easy to store off the weights and even transport them onto a microprocessor after effective training if we wanted. Note that this configuration is just for a 5x5 maze because it is sized to work with 9x9 inputs, although a larger maze such as 16x16 working with 31x31 inputs really wouldn't be that much larger. Also note that 1 is always used for the convolution strides and padding is always zero. Having a network with 146,948 weights makes the network extremely expressive and capable of representing many different functions. This is great because it is assumed that our function for returning an action to the shortest path to finding the maze's center goal is going to be quite complex.

For instance, if you were to write a program step by step to return an action just based on some 2D image input, it would be extremely challenging. No, actually there are other approaches used to finding the shortest path in a maze such as the flood fill algorithm [2]. The flood fill algorithm uses a stack with cell values representing some number of steps away from the goal. The algorithm starts by issuing cells with value zero, and correct location coordinates to the goal position with a stack data structure. From here the cells expand directionally where there are no walls blocking them, and add these cells onto the stack with incremented values. The algorithm then floods the maze with these cell structures until the whole maze is taken up. After this whole process, each of the cell objects contain position coordinates and a value which marks its distance from the goal. If the agent wanted to find the quickest path to the goal it could simply follow the cell values in descending order all the way down to zero. In practice the flood fill algorithm work very well, its main drawback is its very slow and it needs to operate on the entire maze space each time using a stack. One of the things we will look at is the time difference between running the flood fill algorithm and our CNN, which is just doing multiplication.

Analysis

We now know almost have everything we need to start training our maze solving CNN. The final details we need to start coding up the project is how we plan to generate random 5x5 mazes. Generating any random maze might seem like daunting task but we can do it by using a similar approach to the earlier discussed flood fill algorithm, this time with a stack structure that uses cell objects to eliminate single walls in a fully populated walled maze until every cell is visited. The result of doing this is simple random mazes that follow the two primary rules for maze generation. The first rule is that there can be no floating pegs, every peg needs to be connected to at least one wall. The second rule is that there can't be any fully blocked cells, in other words the agent must be able to transverse to every cell. This random maze generating algorithm creates a good template, if we really wanted to we could start removing further random valid walls from it, valid in this case meaning it doesn't break the previous defined two rules, however for our purposes it works fine.

The final detail we need to discuss is how exactly our we binning state batches together to update them. From Deep Mind's paper [1] they used a very promising approach called memory replay, and at first this was explored, however a quicker update scheme was realized which apparently Deep Mind has also looked [2] at called asynchronous learning. Asynchronous learning was defined by Deep Mind as a way to have multiple agents play the same game and combine their experiences together creating shared learning experience. They also cited that this was seen to be much quicker than memory experience replay in terms of converging a solution. In our definition to asynchronous learning, it is kind of similar to Deep Mind's. What we plan to do is project the agent onto every possible cell location, pick a random action at that locations and update all the cells at once then with our CNN. The agent its self then would always pick highest valued actions and move until it reached the center goal location, in which case the agent would go to the start and the maze would regenerate. If memory experience replay was being used then we would have a large que to store all of the agent's state, action, reward, and transition max values. We would also need to use an epsilon value with probability to pick random actions help exploration. Deep Mind suggested that the epsilon value start high and decrease over time, so that the agent would start out making pure random moves and after a long enough time only pick highest valued actions. When comparing memory replay to asynchronous learning, asynchronous was much faster at updating all mazes and it could improve every maze cell not just ones visited by the agent. Since our maze is 5x5 updates would happen in batch sizes of 25 in this case, 25 being all the possible cell locations the agent could be in. The full pseudo code for the asynchronous updating algorithm can be found below in Algorithm 1.

Algorithm 1 Maze Deep Learning with Asynchronous Updates

```
Initialize action -value function Q with random weights
For N trials DO:
    Set agent to start of maze
    Generate new random maze
    While agent has not reached center of maze
        For each possible cell position
            Pick random action for cell position
            Feed forward cell position state in Q and see reward, and next state from
            following random action
            Set $y_j$= $r_j$                              if next state is terminal
```

```
                    $r_j + \gamma \max_{a'} Q(s_{j+1}, a': \theta)$        for non-terminal state
            End for
            Perform gradient decent and minimize Loss as defined in *Equation 1*
            Feed forward agent's state in Q and choice action with highest value from Q
        End while
End for
```

By using the asynchronous update rule, we abandon the epsilon, exploration hyper parameter. For updates we will always choice a random action and for agent progression we will always choice the highest valued action. The only other hyper parameter that we have is gamma (γ) also called the discount value, for our experiment we will use a relatively high gamma of .8. Gamma basically just tells us how much we care about the value at the next state. In our case we want to have a gamma value close to 1 because we entirely care about the next states highest value, in fact this plays greatly into how other cells know which action to return. However, having a gamma exactly 1 could make weight values grow very large and cause the network values to explode. This behavior was observed if gamma was allowed to be greater than .8, but it's also possible to fix this if the learning rate is decreased. It turns out to be a careful balancing act between setting the gamma and learning rate pair, because too small of a learning rate will also cause very slow training. After much experimentation it was decided to use a combination of .8 for gamma and 1e-3 for the learning rate to get a fast training response.

Experimentation

Now that all the hyper parameters are covered and implementation is done we are ready to experiment with teaching our CNN to find best state to action mappings. The model was trained according to Algorithm 1 over the period of about 30 hours. After this amount of training it was observed that agent could solve most mazes in the optimal number of moves, and only a few times did it get stuck waiting for the update algorithm to push it through. It's important to get an idea of how much training time is sufficient to get good results, in some cases for reinforcement deep learning you might need to train for weeks before you get great results. To understand if 30 hours was sufficient or not we need to look at how many random mazes we can create using the defined random maze generator previously discussed for a 5x5 maze. Equation 2 derived by myself below shows as a function of the maze dimension the number of possible different mazes that could be generated.

$$C(n) = (2^{2n-3})(3^{\frac{n^2-3n+2}{2}}) \in n \geq 2 \textit{ for nxn maze} \qquad (2)$$

If we use n=5 in the equation 2, then the output is 93,312. It was observed that if instead of using random but set sizes of predefined mazes that the agent was able to eventually, after seeing the maze about 20 times or so master it. Therefore, we would expect to at least run through almost 2 million randomly generated mazes in Algorithm 1 before the agent could show very good results. Having ran the agent for about 30 hours thus far, and if it was completing about 500 mazes about every 3 minutes during the later of its training life cycle then it saw about 300,000 mazes, this seems about right for how the agent was performing. If we trained the agent for another 150 hours we would expect the agent to master every maze it saw and it would be very rare that it ever finished a maze in more than the number of optimal moves. Taking equation 2 even further, if we wanted to train the agent on a larger maze say 16x16, there would be a calculated amount of $2.49x10^{57}$different possible randomly generated mazes. If once again it took the agent 20 times seeing one maze before completely mastering it, and it could see

500 mazes every 3 minutes it would need to be trained an impossibly long time, note that's for an optimal policy. It's not too difficult to see then where even deep reinforcement learning reaches its limits in terms of training time and having very large number of possible states. This isn't to say that this method of deep reinforcement training is the most efficient either, nor is the computer running the training the best suited. Therefore, you probably could get good result on a 16x16 maze as well using this similar method and training for maybe several weeks.

Results

When running the maze updating algorithm it helped greatly for visualization by including a GUI element to it, although this component was turned off during long training intervals for performance. Along with visually showing where the agent was in the maze along with all the row and column features of the maze its self, four arrow directions were displayed in each cell to show how the CNN was thinking. This idea was greatly motivated by PhilippeMorere [3] and his GitHub code for Basic Reinforcement Learning. PhilippeMorere's code acted as a template for the entire project where the World was then greatly modified to represent maze environments and the Learner was modified to use deep reinforcement learning implementing its CNN with tensor flow. Figure 3 below shows how the maze GUI works to highlight how the CNN values its actions (green being high valued and red being low valued) according to the specific cell it is looking at during two different life cycles of training.

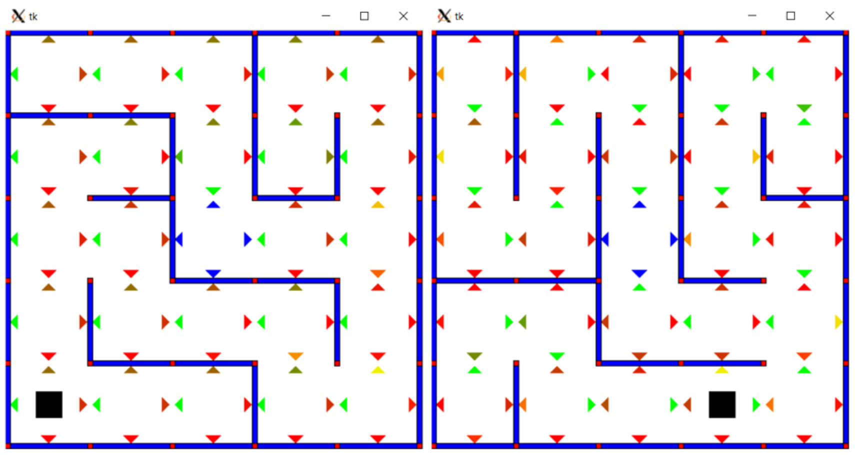


Figure 3: Left: training has just begun and weights are random, Right: after 30 hours of training

It can be seen from Figure 3's right depiction and looking at the weights that almost every cell has a very good understanding on which way the agent should go if the agent were in that cell. The only cell that might be questionable it the top cell second to the right, here there are two arrows almost the same color green, meaning the action values are very close to being equal. Also the cell just to the right of that one is in a close to being similar situation where its down arrow would be optimally yellow. There are other cells where the arrows could be less green as well but remember that is was for a newly generated random maze, chances are the agent has never even seen it before, or maybe just a couple of times during training. If the agent where to spent a long time in any given maze, then the arrows would tend towards optimum states.

Metrics and Analysis

Next we will attempt to quantify how well the agent is doing at solving mazes after some amount of training, as well as how well the agent is able to learn in general. First we will look at the learning part, to do this we will run the agent on some large set of saved off generated mazes. A good number of mazes to run comparisons on is 500 since the agent is able to complete this number in a couple of minutes generally. The program will use the optimum number of moves as its reference. To find the optimum amount of moves we will use the flood fill algorithm, discussed previously at the end of the Implementation section, before running each maze. For each training period the agent will run until its completed 500 optimum maze runs while also counting the total number of runs it takes as well. If the agent was perfect, then the optimum run number would equal the total run number. The idea here is the total number of runs should approach the optimum number of runs as it trains longer on its maze data. Also remember the maze data in this case is a consistent size of 500, so the agent is going to be seeing same 500 mazes over and over again. Here we will explore what this relationship looks like for both a network that has trains for over 30 hours and a network that has never trained at all. Figure 4 below shows the two relationship graphs, as can be seen from the data the agent is indeed learning and getting better with its at performing optimum maze runs as it trains longer on the same 500 mazes.

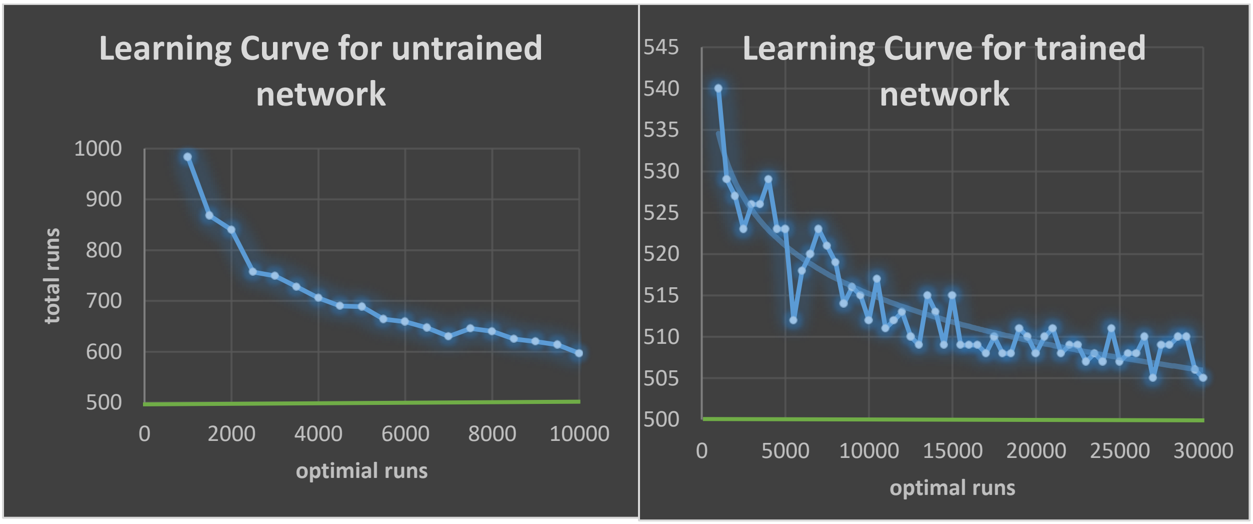


Figure 4: Analyzing how well the agent learns from consistent training data of size 500, both are approaching 500 total runs (green line) for perfect performance execution.

This data is very interesting. First notice that the two graphs have the same axis values but have different scales. The training network graph was purposely extended for the x-axis to show that the number of runs increased for the network to get better and better performance, what we see here is marginal returns. The y-axis, the total runs is our measurement, while an untrained network sees a large change in total runs, the change for the total runs is much smaller for a trained network. A network that hasn't trained at all starts rather poorly, it takes almost twice as many runs in total than the optimal number of runs at first. The untrained network picks up very quickly though and in just 20 training periods, or 10,000 optimal maze runs, the agent is just about where the trained networks performance starts out at. The trained network starts at doing rather well its only over by about 10% right at the beginning so it has learned to generalize well with its 30 hours of training. However, learning progress for the trained network is slow, to go from that last 10% to 0% error takes a lot of training periods. During 60 training periods, or 30,000 optimal maze runs, the trained network only went from a 10%

error to 1% error. To get to 0% the network might need another 30,000 optimal runs, also note this relationship is in direct correlation to the size of the maze data set used. It was observed that for a maze size of 50 for example that the network begins always achieving equal total runs to optimal runs in a rather small number of training periods. Bringing all of this into perspective is important when we consider the total number of random mazes possible and the training on that when we wish to have optimum performance.

As it turns out optimum performance is very important because in our case it will directly influence overall maze completion, whether or not the agent is capable of reaching the goal or not when training is turned off. So optimum runs and maze completions runs are actually the same thing, just because of how we are generating our random mazes in this instance. That brings us to our next step to analyze, after sufficient training how well can the agent solve mazes? We can quantify this as a percentage shown in equation 3, give the trained agent some large number of randomly generated mazes, in this case 10,000 and record what percent of those mazes its actually able to complete. If the agent doesn't complete a maze it's going to get stuck so we will just count how many moves it has taken in the maze, if the number of moves is over 25 in this case then that maze is considered unfinished and the agent proceeds with a new generated maze. Testing the network that has trained over 30 hours, the agent received 10,000 randomly generated mazes and without any training it completed 9275 for a respectable completion score of 92.75%. Our pervious defined benchmark was a completion score 90% or greater which we were able to accomplish. From our learning graphs, we can expect that if the agent trained longer then this completion score would get better but as we also saw, at a decreasing rate. It would appear to take a very long amount of training time before the agent was completing close to 100% of its mazes. For an untrained agent the completion rate is 0%, the maze environments are too complex for an agent without any insight to simply stumble onto the goal. It was also observed that for the agent that had trained for over 30 hours, it was only getting stuck in areas close to the start, where it would proceed to go back and forth between two adjacent cells. The reason for this is thought to be because the updating process spreads out from the goal, weight values grow quicker towards the goal. It takes some time for weights further away from the goal to update so those cells are more likely to have bad directional information for the agent.

$$completion\ score = \frac{number\ of\ trials\ ran\ that\ finished\ in\ under\ 25\ moves}{total\ number\ of\ trials\ ran} \qquad (3)$$

Conclusion

By using convolutional neural networks combined with reinforcement learning we were successfully able to teach an agent how to solve any number of consecutive randomly generated mazes. The agent learned everything just by analyzing raw input data, which if you think about it, is actually very impressive. By being able to use raw input data for our network we overcome the problem of high dimensionality which was why it was not even possible to try to solve this problem with just a basic reinforcement agent. We also successfully employed asynchronous learning which was able to speed up the learning process compared to using memory replay. The results of the agent after more than 30 hours of training was it could complete almost any randomly generated 5x5 maze that we could give it, with a completion rating of more than 92%. We also showed that although the learning curve grew steeper the longer the agent trained, its general performance in getting optimum runs or maze completion runs was always increasing, and not hitting some max limit.

Reflection

In this last section we will explore how practical the learning algorithm is compared to other methods already being used, the main one for maze exploration is how does deep reinforcement learning compare to using the flood fill algorithm. Early we said we would look at the timing difference between the two update method, as it turned out for a 5x5 maze the flood fill was at least five times faster than using the network update, however for larger mazes the flood fill algorithm time would be expected to grow quite exponentially since it adds cells onto a stack from all directions. Using a neural network on the other hand we would probably just need to adjust the weight size by a little bit in order to take inputs in for larger mazes such as a regular 16x16 micro mouse maze, so the networks time probably wouldn't increase by that much. The biggest factor in terms of time though for the deep learning network is not how much time it's using while solving but the time for training. We saw how long it took just to train a network to get a 92% rate on 5x5 mazes, and how learning took longer, then imagine the amount of time need to get good accuracy on a 16x16 maze. In order to tackle 16x16 mazes we would need to implement some drastic updates that would increase how well the network is able to learn and generalize.

Future Projects

Although this project does a great job of exploring the power of deep reinforcement learning it does not introduce anything more powerful than the basic flood fill algorithm, which I am convinced already is the optimum function solver for any maze, even blind search mazes. Blind search mazes are mazes where most of the walls start hidden like in micro mouse, and the agent has to go around discovering walls. What is interesting though is that just maybe the flood fill function is not the optimum answer for blind maze searches, and if it's not the deep reinforcement solver with its unbiased approach would have the ability of exploiting the maximum valued function, given the adequate amount of training time that is. Knowing just that is what really makes deep reinforcement learning exciting, and for a future project it really would be interesting to try to prove if the flood fill function is the best or not for blind search. It would also be really interesting to try to tackle the steep learning curve problem, even with asynchronous learning, the problems with deep reinforcing learning with high dimensionality became very clear. While deep learning was able to digest high dimensionality states, those states still would lead to a negative impact during its training life cycle. As a first time exploration into the methods of using deep reinforcement learning this projects was tremendously beneficial and insightful. The final code for this project can be view on GitHub [4], and the demo video is available at [5].

References

[1] *Playing Atari with Deep Reinforcement Learning* https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

[2] *Flood Fill Algorithm* https://en.wikipedia.org/wiki/Flood_fill

[3] PhilippeMorere https://github.com/PhilippeMorere/BasicReinforcementLearning

[4] https://github.com/awbrown90/DeepReinforcementLearning

[5] https://www.youtube.com/watch?v=1F6oGEItjOg&feature=youtu.be