Final Project Report: View Plane Detection using Reinforcement Learning

Natalie Aw and Anthony Song
May 10, 2022

Abstract: Here we present our Deep Learning project, view plane detection using reinforcement learning. We outline the problem of finding the correct orientation and anatomy during medical imaging, and how we can investigate this in the computed tomography domain. We discuss what reinforcement learning is and provide some toy examples to explain. We then walk through our implementation, which takes advantage of Deep Reinforcement Learning (DRL) to learn the reward-maximizing path to our desired orientation via Deep Q-Learning. We discuss the importance of the choice of reward function, and describe several Deep Q-Network (DQN) architectures that we tested. We summarize our results and conclude with future avenues to pursue.

1. Introduction

Noninvasive medical imaging is an important tool in diagnosing organ health without invasive surgery. However, obtaining high quality images are highly dependent on the technician's ability to find the correct orientation and anatomy. This problem can lead to inconsistency and inefficiency as technicians must be trained rigorously, and can differ slightly when imaging. We propose aiding the technician to the desired orientation with automatic view planning, a technique that lends itself well to this field. In particular, automatic view planning is especially relevant in ultrasound, MRI, and some CT applications. From a clinical perspective this could be used to guide technician training, increase operating room efficiency, and lead to standardized image angles for quantitative analysis [1]. From a high-level perspective, this problem requires solving many sub-problems including multi-modal image synthesis using Generative Adversarial Networks, deformable image registration and view planning using reinforcement learning. In the scope of this project, we explore automatic view planning in the computed tomography (CT) domain. We employ the use of reinforcement learning, which is a model that learns as it explores the environment. In particular, Deep Q-Learning has generated much interest and is capable of handling the high-dimensional data we are working with.

2. Reinforcement Learning

Reinforcement Learning (RL) is a field of machine learning where an intelligent agent (something that perceives its environment) learns what actions to take by maximizing a cumulative reward. The general model takes an input, the initial state, and returns the reward-maximizing solution. Decisions are made sequentially - the output depends on current input, the next input depends on the output of the previous input. Essentially, the RL problem can be described as a Markov Decision Process (MDP) [5]. There are two approaches that can constrain the RL algorithm: a value function approach and a policy search approach. Both approaches make use of the Bellman equation, as seen in *Equation 1* below. The Bellman equation makes use of discounted future rewards to provide the agent with the necessary information for dynamic optimization [11].

$$V(x) = \max_{a \in \Gamma(x)} \{F(x,a) + \beta V(T(x,a))\} \tag{1}$$

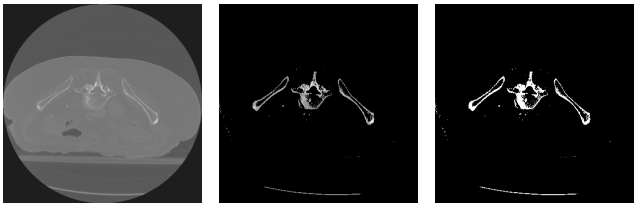
Here, V(x) is the value function at state x. F(x,a) is the current reward evaluated at state x and action a, while T(x,a) represents the value of the next state at x, a. Γ represents the set of states that an action can take (essentially, only valid actions), and β represents a discount factor between 0 and 1 that represents the impatience - preference for sooner rewards - of an agent.

The value function and policy approaches both utilize the Bellman equation, but have a different approach. In a value function approach, the agent directly maximizes the reward function, taking the next action that will maximize its reward . In a policy function approach, the agent instead tries to find the optimal policy to take, maximizing over all possible policies, and follows that policy for its actions. Both implicitly update the state values and policy function, though still have subtle differences [10].

With the advent of deep learning, we can expand from traditional reinforcement learning methods to more robust and interesting architecture. As reinforcement learning can grow exponentially large, deep learning is a way to approximate the value function, keeping the computational expense much lower. Further, deep Q-learning has been successfully implemented in the medical imaging domain, such as for 3D ultrasounds for fetal brains [12, 4].

3. Dataset

Under the supervision of Dr. Jeff Siewerdsen and Dr. Ali Uneri, the I-STAR lab provided us with full-body high resolution CT and cone-beam CT 3D images. This dataset includes all major anatomical features in order to test our models: volumes of high bone density, high soft tissue density, and a mixture of bone and soft tissue. After slicing the



(a) Original CT Slice (b) CT Slice Thresholded (c) CT Slice Binarized

Figure 1: Preprocessing of 2D CT slices

cadaver into high-resolution 2D slices, we tested and ran our models on 2D slices that localized the pelvis. Eventually, these models can be extended to 3D volumes, but this was not explored in the scope of this project. The 2D slices were generated in a medical imaging package called Insight Toolkit (ITK) [6], then re-scaled for intensity values between 0-255, and thresholded, resulting in a binary image. The progression of our preprocessing is shown in *Figure 1*.

4. Implementation

Our code was written and executed in Python 3.7, with testing and plots done in Jupyter Notebook. We used the OpenAI Gym package, which is a Python library specifically for reinforcement learning, to create our environments. For our deep learning models, we used PyTorch. Further details are provided in their respective sections.

4.1. Q-Learning

The value function approach estimates expected return of being in a given state, which can be implemented with a method called Q-Learning. The value function is represented by a Q-table, which contains the output of the value function at each possible state, for each possible action. The Q-table is then updated after every action the agent takes. The update equation exploits future discounted reward to teach the agent the optimal actions to take. This is shown in *Equation 2*, which is a form of the Bellman equation.

$$Q^{new}(s_t, a_t) = \\ Q(s_t, a_t) + \alpha(r_t + \gamma \max_a\{Q(s_{t+1}, a)\} - Q(s_t, a_t)) \quad (2)$$

We can consider this equation as a combination of three parts. The first, $(1-\alpha)Q(s_t, a_t)$, represents our current Q-value weighted by α-learning rate with $0 < \alpha < 1$. An α near 1 changes the Q-table more rapidly. The second component, αr_t, is the reward $r(s_t, a_t)$ obtained if action a_t is taken at state s_t, weighted by α. Finally, the last component is our discounted expected value, weighted by both α and our discount factor γ. Here, we only consider one step in the future to calculate our discounted future value [9].

For our task, we will use this Q-learning framework. In particular, working first with a simple Q-learning table will help us learn the basics of reinforcement learning, and provide a good point from which we can extend our work. We used the underlying structure of OpenAI's Gym package to create our environments and agents. After defining the environment, reward function, agent, and agent actions, we let the environment render and run. This is done with essentially two for loops: the outside loop is called an episode, where the agent is allowed to explore the environment given a random starting point; the inner loop dictates how many "timesteps" the agent can take within an episode. Each episode and timestep update the Q-table, which is randomly initialized to "0 reward" at the start. Traditionally, value functions take on negative values with the optimal position valued at 1.

Additionally, Q-learning uses the idea of exploration versus exploitation. Essentially, the agent wants to exploit the knowledge it has at a certain timestep. However, this could lead to potentially an incorrect local maxima if the agent does not explore the environment enough. This is the tradeoff between exploration and exploitation: the agent can choose to take the optimal action each time (exploitation) or it can move around the environment with some stochasticity, randomly choosing an action with probability ϵ (exploration). while a completely exploitation model is quick (greedy agent) it may not find the optimal action. In contrast, a highly explorative model (ϵ-greedy agent) could significantly slow learning. A third method (decaying ϵ-greedy agent) balances the two by decreasing the probability that the agent will ignore its knowledge and take a random action, thus beginning its learning with high exploration, and ending its learning with high exploitation [3].

4.1.1 Toy Example 1: Box

Our first experiment was to familiarize ourselves with creating an environment and agent. We generated a simple box image with four-tiered intensities, as shown in *Figure 5a*. In this experiment, it is very obvious where the box should go to obtain maximum reward; essentially, the agent should learn a gradient function and move directly to the white square. We also wanted to make sure that the agent learned the Q-table regardless of its starting position, so we randomized the start of each episode.

As a proof of concept, we also needed to show that we could extrapolate intensities off our loaded image and incorporate this information into the reward function and Q-table. By averaging over our field-of-view (FOV), we can reward our agent when the maximum average intensity is reached with a reward of 1, thus ending the episode. The reward function used here is: $-100(255/(FOV_{int} + 1)$ The reward function is simple because our environment is

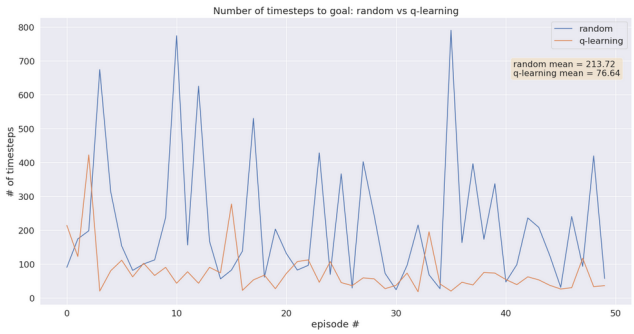
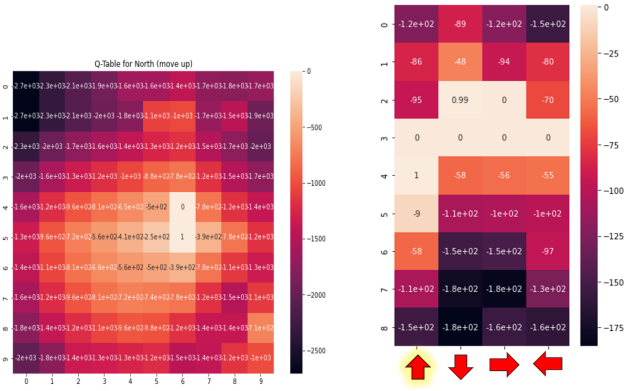


Figure 2: Toy Example 1: Random Exploration vs. Q-Learning Steps



(a) Final Learned Q-Table for Action UP

(b) Q-Table for Column 5 (aligned with white square)

Figure 3: Toy Example 1: Final Q-Tables

simple. When the agent first starts, it is highly explorative, moving randomly and takes many steps to get to the target. However, as the agent learns it quickly finds the best action for each state, or position on the grid. This can be seen in both *Figure 2* and *Figure 3*. At first, it takes the agent as many steps as a random search to find the target. However, as the agent begins to learn, it takes fewer and fewer steps. In *Figure 3a*, the final Q-table for the action UP is shown for all states, and *Figure 3b* shows the final Q-table for the states along the fifth column - or the column containing the target square. A limitation of this example is that the agent is clearly moving on a grid, taking large steps per action. Because of this, we have a small Q-table of size $(9 \times 9 \times 4)$ (our grid size by number of actions), which is not a good representation of our data.

4.1.2 Toy Example 2: 2D CT slice

Next, we investigated our environment and reward function in a Q-learning example with sample data. We did not use the binary thresholded image, but the thresholded

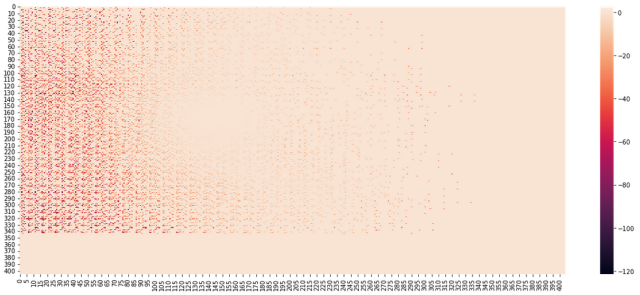


Figure 4: Toy Example 2: Final Q-Table

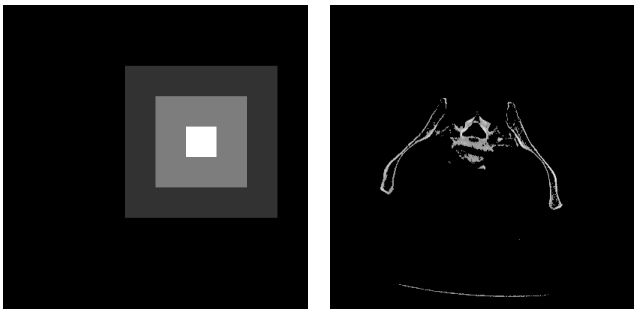
image as shown in *Figure 5b*. We first investigated the CT slice environment on the grid to ensure that our reward function was suitable for this new environment. We found that we needed to change the reward function to $-(100 - e^{(FOV_{int}/32)})$, to properly find our target FOV. This can be shown in our example videos. At the start of the learning (*toy2_initialRand.mp4*), the agent moves completely in an exploratory fashion. If we set our reward function properly, the agent will learn to find the target FOV with minimal exploration (*toy2_goodLoss.mp4*). However, if the reward function is poor, the agent gets stuck in local maxima, accidentally receiving a positive reward that is larger than the target FOV reward (*toy2_badLoss.mp4*).

Because the target FOV may not lie perfectly on a grid we removed the grid and allowed the agent to move in pixel increments. Now, our Q-table is of size $(405 \times 405 \times 4)$ - already rapidly increasing. Obviously, this example is not as clear cut as the first. There is no definitive maximum solution that the agent can find, as seen in *toy2_badLoss.mp4*. Even though the agent can move to roughly where the correct area is, it does not settle on the same FOV. This is due to the reward function - if we threshold above a certain intensity, we are not guaranteed to find one global maxima. We can also see from the final Q-table that the agent did not learn well (*Figure 4*), as there are many states with equivalent reward. Although it seems to mostly work for this toy example, it likely will have undesirable consequences as we move forward with larger observation spaces.

4.1.3 Observations from Toy Examples

We found three main observations from our toy examples:

1. The reward function is crucial, as well as the actions we allow the agent to take. A reward function not fit to the environment can result in a confused agent that ultimately does not learn a good policy - this is easily seen in the unclear path in *Figure 4*. A reward function that works for one environment likely will not translate well to another, limiting the robustness of this approach.



(a) Box Environment (b) CT Slice Environment

Figure 5: Toy Example Environments

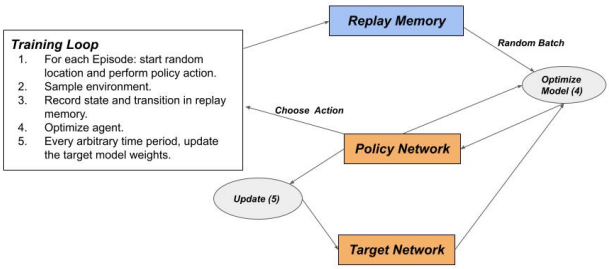


Figure 6: General Deep Q-Network training loop.

2. Another behavior we observed found was the agent sometimes got stuck along the edges of the environments. To combat this, we included a step size in our action step. If the agent hit a "wall", the step size was increased from 1 to 10, effectively moving the agent further away from the boundary. We can also explore an adaptive stepsize as a function of the received reward.

3. Although the Q-table accurately learns the best actions to take given a state (*Figure 3*), we can see that implementing a proper Q-table will be computationally intensive. One can conclude that when you wish to solve a problem where the action space and environment are large, the agent will have a difficult time playing enough episodes and exploring enough to determine an optimal solution from any given starting point. Thus, as the action and state spaces increase, another solution is needed.

4.2. Deep Q-Learning (DQN)

Since neural networks are theoretically universal function approximators, Deep Q-Learning extends upon Q-learning work by using neural networks to replace the Q-table and map input states to action, value pairs. Essentially, we are learning a function which will output the Q-value for all possible actions given an input state. Interestingly, [7] showed that using two neural networks, target and policy, led to training stability. These networks have the same architecture, except the target network uses frozen weights. Every N steps, the weights from the policy network, which is constantly updated, are copied to the target network. The loss function tries to minimize the temporal difference error based on actions selected from a policy (typically epsilon-greedy), and is represented as:

$$r + \gamma max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a, \theta_i)^2 \qquad (3)$$

where the left-hand side represents the greedy γ discounted target network prediction with network parameters θ_{i-1} plus reward, r, and the right side is the policy network prediction with network parameters θ.

4.2.1 Experience Replay

To further improve the training process, [8] introduced a technique called experience replay which stores the agents last experiences at each time-step pooled over many episodes into a replay memory buffer. During training, instead of using input states directly from simulation, the agent performs actions based on a batch of randomly selected memories from the replay buffer. This method has the advantage of data efficiency by re-using previous data, or experiences, and also de-correlating between consecutive samples which will ultimately lead to better convergence behavior when training the function approximator.

4.2.2 Architectures

Though one of the first successful applications of reinforcement learning with neural networks was playing backgammon, many extensions in application and network architecture have been made and examined since. For our application, we began with two simple deep learning models, one with three fully connected layers and the other with three convolutional layers. We then extend our testing into new architectures like dueling DQN and recurrent DQN. The use of many different DQN architectures, have the potential for generalization to slightly modified inputs. For example, if we are interested in centering the field of view on a different vertebrate, or a vertebrate that is not necessarily in the middle of the input image. Ultimately, this makes the Q-learning for higher dimensional problems, like 3D viewplane detection, tractable, as we are no longer limited by memory and computation resources.

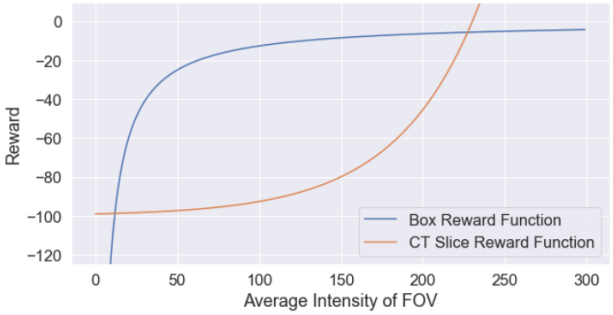


Figure 7: Reward functions for toy examples

5. Importance of Reward Function

5.1. Average Intensity of Target Area

For our first models, we used the average intensity of our FOV to determine the optimal end location. By averaging the pixel intensities, we created a reward function that penalized low intensity areas, and set an average threshold at which the agent would receive the maximum reward of 1. The general reward function we created is shown below in *Equation 4*.

$$R(x) = \begin{cases} -\alpha(\beta - f(\gamma x)) & \text{if } x < \tau \\ 1 & \text{else} \end{cases} \tag{4}$$

We denote the average intensity over our FOV as x. Our reward is thus a function of x, multiplied by some constant γ, where γ and α control the rate at which the reward changes and β sets a lower bound of the reward. τ sets the threshold at which the agent stops, having ideally found the region of maximum intensity. The chosen function f determines the shape of the reward function, as seen in *Figure 7*. In Toy Example 1, a proper rational function is used penalizing the extreme intensities more. For Toy Example 2, an exponential was used, which provided a steeper gradient throughout the mid-range intensities. The advantage of using a intensity-based reward function is that we do not need to know the location of our target, and can easily tailor to any environment. However, the downside is that the parameters of the reward function are highly dependent on said environment and may not translate well. Additionally, as we saw in Toy Example 2, the agent can learn local maxima and get confused (*toy2_badLoss.mp4*).

Though the intensity based method worked very well for Toy Example 1, the environment was so contrived that the global maxima was obvious. Although Toy Example 2 could eventually find an optima, albeit a local one, the intensity based reward function is already showing that it may not be the best approach. Additionally, the reward functions had been explicitly fit to the environment, as shown in *Figure 7*, and it is unlikely we would have such knowledge in most cases.

5.2. Distance from Region of Interest

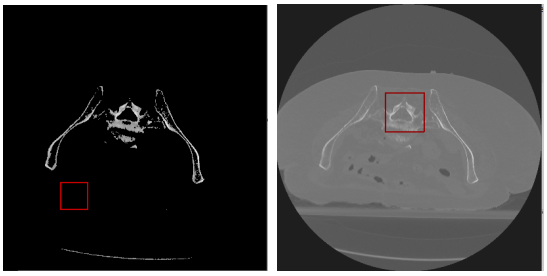
While our initial reward function highlighted in the subsection above worked well for the thresholded CT image, it did not generalize very well with raw CT grayscale slices and the Convolutional Neural Network (CNN) model. This led us to believe that traditional Q-learning and Deep Learning struggled to correctly learn the optimal view plane path in the presence of substructures (fat, muscle, etc.), other than the anatomy of interest, for an non-thresholded image. Therefore, we took the approach of [1] and implemented a supervised reward function that was dependent on the distance to a defined optimal view. This can be written as reward:

$$R = sign(D(P_{i-1}, P_t - D(P_i - P_t)) \tag{5}$$

where D is the Euclidean distance between two views P. The subscripts i denotes the future predicted step and P_t is the target ground truth FOV. Furthermore, to better localize our view plane detection and prevent our agent from oscillating near the target region of interest (ROI), we introduced an adaptive step size which reduces step size as we get closer to the target, and oscillation detection by looking at the standard deviation of the last N points when the agent is near the target view. Although this method works well on DQNs for our simple environment, in actual clinical cases, we would need to develop a semi-supervised reward function when the target coordinates are not available.

6. Experiments and Results

Here we discuss the results for our Deep Q-learning models. All of the models were tested on either the intensity based reward function environment, referred to as intensity-based reward, and distance-based reward function environment, referred to as distance based reward. For the intensity based reward environment, we set the environment view as a thresholded axial CT slice of the pelvis, where the sacrum is the region of interest. Furthermore, the intensity based reward is split into a 9×9 grid for simplicity of initial tests, therefore since the image is a square, each translation is $405/9 = 45$ image pixels. Similarly for the distance based reward environment, we used the same pelvic slice without thresholding, and the grid size equal to the dimensions of the image (405×405) where each translation could be as small as 1 image pixel, thus allowing accurate FOV localization. For simplicity, the action space was limited to size 4, where the agent could move up, down, left, or right in the image. For initial testing, we allowed each method to train for 100 episodes or less, and then test on 100 episodes. The model was then evaluated on the average number of steps to reach the goal for those 100 testing episodes. In the



(a) Thresholded Intensity Based Reward

(b) Grayscale Distance Based Reward

Figure 8: Two environments used for testing our agent (red box) which represents the current FOV. In (a) the agent moves in a 9×9 grid, and in (b) the agent has an adaptive step size that traverses pixels. (b) shows the agent when it finds the target FOV.

calculation for average testing steps we removed outliers that were greater than the 95th percentile, due to rare occurrences where the agent would get stuck in a certain position, or local minimum, for a long period of steps . Furthermore, in our simple environment, we found in some models, that training for 100 episodes reduced performance through potential overfitting (*Table 3*). For the results in *Tables 1 & 2*, all DQN results are trained on an NVIDIA GeForce GTX 745 GPU and all traditional Q-learning results are trained on CPU.

6.1. Hyperparameters

This section will briefly go over some of the important hyperparameters for traditional Q-learning and deep Q-learning. For traditional Q-learning and DQNs, choosing a proper starting epsilon and decrement factor for the epsilon greedy policy is very important in helping the agent learn in initial stages. We found that starting with an epsilon of around 0.99-0.95 promoted good agent exploration in our environment. Next, we used ending epsilons around 0.10-0.05 to promote exploitation in the later episodes once the agent had sufficiently learned the environment. The γ parameter is important to teach the agent how important immediate reward is compared to future rewards. Lower γ values immediate reward while higher ones emphasize future rewards. The choice of γ depends on the application and number of steps required to reach the goal, therefore, we found that the intensity based reward environment benefited from a lower $\gamma = 0.95$, while our distance based reward environment performed optimally with a higher $\gamma = 0.99$. For the DQNs, we found that a batch size of 32, target model weight update every 5 episodes, RMSprop optimizer, and a learning rate of 1e-2. Literature [7] and preliminary results showed that the RMSprop optimizer provided more stability when training with larger learning rates, 1e-1 to 1e-2.

6.2. Traditional Q-Learning

The intensity based Q-learning environment took about 20 minutes to train for 100 episodes on the CPU. The agent took about 65 episodes to start becoming proficient in finding the target FOV quickly (*Figure 9*). Though worse, the intensity based reward environment performance was relatively similar to the DQN models (*Table 1*) because the agent was able to properly explore and learn the 9×9 size environment. However, when transitioning to the distance based gray scale environment, the training time was much longer (3.5 hours on CPU) for 100 episodes due to the larger observation space (405×405). Furthermore, for the distance based method, even when trained for 200 episodes, the agent was not able to adequately explore the environment to provide a robust Q-table for good testing performance compared to the DQN models, and as we can see in *Table 2*, Q-learning performs much worse than DQNs. This is because as the environment or action space becomes larger, the problem becomes more computationally intensive until it is not tractable. In our case, the agent must explore and exploit a space of size $(405 \times 405 \times 4)$. Lastly, one thing to note is that when changing from the smaller, (9×9), to the larger, (405×405) environment, is that we must lower the epsilon decay rate to allow the agent to explore more through randomly selected actions because the number of steps to the target goal is increased. This prevents the agent from oscillating in a region close to the target FOV.

6.3. Fully Connected DQN

The fully connected DQN we used had 3 fully connected (FC) layers, which took the rendered image of the agent in the environment and flattened it to size (*img height* $\times$ *img width* $\times$ *3 RGB channels*, 1). The output of the first FC layer was 120, with ReLU activation, the output of the second FC layer was 84 with ReLU activation, and the output of the final layer was the size of the action space, 4, with linear activation. Even though for both environments, the FC model's performance were comparable to the convolutional DQN, it is evident that convolutional layers are much better for learning from agent-environment rendered images, especially in the grayscale distance based environment (*Table 2*). From a feature learning perspective, the FC model flattens the image into a 1-D input array, and by nature of the FC layers, is constrained such that every input affects every value of the output in the output vector. Therefore its ability to learn and utilize the spatially invariant information from the environment is less optimal than that of the convolutional network discussed in the next section. Furthermore, we found that for the distance based metric, when training more than 40 episodes, the agent would frequently time out or oscillate in a region until it was able to move out of its current local minimum. Consequently, during testing, we

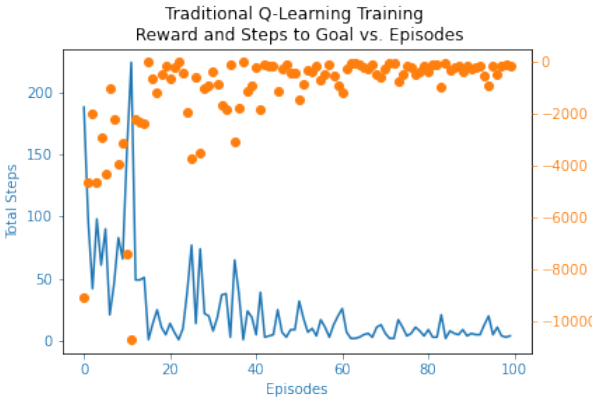


Figure 9: Total steps taken and reward at each episode during traditional Q-learning training for the thresholded intensity based reward environment. The orange dots are rewards for each episode

Figure 10: Total steps taken and reward at each episode during Convolutional DQN training for the thresholded intensity based reward environment. The orange dots are rewards for each episode

would also see the same behavior, thus reducing the testing performance. This was evident in both the fully connected and convolutional models for the distance based reward.

6.4. Convolutional DQN

The convolutional DQN we use has three convolutional layers and 1 fully connected output layer. All convolutional layers are kernel size = 5, and stride = 2, with 16, 32, and 32 channels for the first, second and third layers respectively. Following each layer, we have ReLU activation and 2-D batch normalization, which feeds into the final FC layer with linear output of the action space size, 4. Overall, it is evident from *Tables 1 & 2* that the convolutional models performed the best in terms of testing performance for both environments, especially the gray scale distance based environment with about half the steps of the

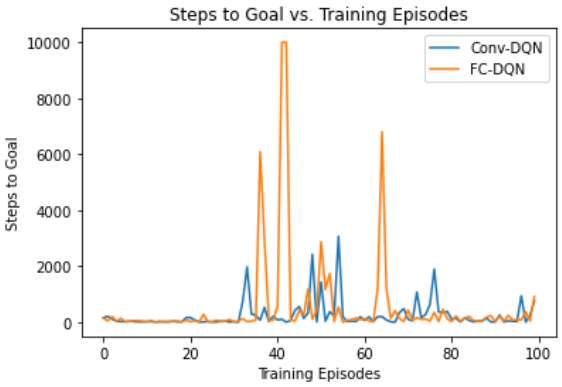


Figure 11: Steps to goal for training episodes for the grayscale distance based reward environment. We can see that around 40 episodes the steps start to increase.

FC model. In *Figure 10*, we can see how quick the Convolutional DQN model learns, showing low steps and high rewards around 20-40 episodes, for the simple thresholded intensity based environment. Compared to the FC model, for images, not every input node should affect every output node, which therefore highlights the strength of convolutional layers when learning spatially invariant shapes and textures of the image. Because we are essentially learning the parameters of a sliding window filter, the features that the convolutional DQN learns are invariant to the location in which the agent is randomly initialized, therefore guiding the agent to the target much quicker. For example when comparing the FC and convolutional DQNs, we see that the thresholded environment, produces little difference in performance (*Table 1*) most likely because the image really only contains information about the target anatomy (pelvis). However, when switching to the gray scale environment, although the reward function is changed as well, the convolutional DQN substantially outperforms the FC DQN (*Table 2*), most likely because other anatomy, like tissue and muscle are present in the image, and therefore the convolutional network is able to handle these features better. Analogous to the FC DQN, we also found that training for over 40 episodes led to overfitting and therefore a decrease in testing performance for the distance based reward environments. We believe this is because of how the reward function is structured such that the model will often times oscillate near the target FOV without meeting the criteria of actually reaching the goal. In *Figure 11* and *Table 3*, we can see that the training and testing performance degrades when we over or undertrain the model for the distance based reward environment. Lastly, we can also see the computational advantage of less training time for the convolutional DQN due to a smaller number of parameters. On average the training time using the distance based method for the

convolutional DQN was 2 minutes compared to the 5 minutes for the FC DQN, when training for 100 episodes.

6.5. Dueling DQN

Intuitively, the dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state. The dueling architecture represents both the value V(s) and advantage A(s, a) functions with a single deep model whose output combines the two to produce a state-action value Q(s, a). Unfortunately, when testing the Dueling DQN on both the intensity-based reward, and distance-based reward we found that the network was rarely able to locate the region of interest. Specifically, the network continuously chose regions that circled the perimeter of the environment, rarely every getting close to the pelvic anatomy. We believe this may have occurred because the dueling architecture may not have learned which valuable states are since, the environment is unchanging and rewards are based on initial movements.

6.6. Deep Recurrent QN

We can also extend the vanilla DQN model to incorporate a time-dependency. Though this is similar to experience replay, a DRQN differs slightly both in its approach and the problem it attempts to address. Experience replay focuses on stabilizing the learning by (1) de-correlating samples and (2) efficiently using past experience by randomly sampling from the buffer. Both help the Q-learning to converge, as mentioned in section 4.2.1. In contrast, a DRQN has a Long Short-Term Memory (LSTM) layer as the layer immediately before the fully connected layers. Now, the agent builds a model of hidden states to improve predictions, under the assumption that direct observations are not sufficient. This architecture is particularly useful in partially observable Markov Decision Processes (POMDP), when the agent does not know the environment fully, such as if there is considerable noise in the environment. In contrast to experience replay, a DRQN is forced to learn some representation of the changing states, as in a Hidden Markov Model (HMM). Many different variations can be made here, and some may not be useful [2].

In our implementation, we added an LSTM layer before our Fully Connected Network (section 6.3). We used an embedding size of 8, and hidden state size of 128. Similar to the Dueling DQN, the DRQN failed to learn well on either reward function. Again, the network seemed to circle the perimeter of the environment and did not move towards the center FOV. There are a number of possible reasons this is. One is that the architecture was implemented in our existing framework, which already had experience replay. While the two address different issues, it may not be wise to use experience replay within the DQRN framework. Another reason is that the advantages of a DRQN did not translate well to our stable, known environment. However, we would expect the model then to produce results similar to the FC-DQN.

Model	Avg.# Steps for 100 Test Episodes
Q-Table	14.6
FullyConnected	9.4
ConvNet	8.4

Table 1: Model performance on thresholded pelvic CT slice using an intensity-based reward

Model	Avg.# Steps for 100 Test Episodes
Q-Table	1324.6
FullyConnected	147.3
ConvNet	70.8

Table 2: Model performance on original grayscale pelvic CT slice using a distance-based reward

# Training Episodes	Avg.# Steps for 100 Test Episodes
100	932.2
80	1232.7
60	642.2
40	70.8

Table 3: Model performance as a function of # training episodes for grayscale distance based environment using convolutional DQN.

6.7. Generalizability of the Network and Future Work

A difficulty in DRL is to have the agent use what it has learned and then generalize its actions to solve a new environment. For this experiment, we are interested in testing our optimal model's ability to generalize to shifted and rotated environments when training on the centered image. For simplicity, our experiments used the convolutional model trained for 250 episodes in the thresholded intensity based reward environment (*Figure 8a*) on various data environment augmentations where the anatomy is shifted and/or rotated (*Figure 12*). The trained model was then tested on 100 episodes of translationally and rotationally augmented environments that were not seen during the training process. Unfortunately the results were quite scattered such that some testing environments (e.g. bottom-left translation and centered) performed well, while in other cases the

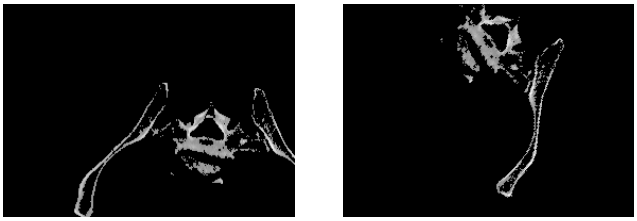


Figure 12: Various data augmentations using translation and rotation to train the convolutional DQN to generalize on un-centered target anatomy.

agent failed to locate the target FOV. Currently we believe that there are some randomly initialized starting positions where the agent performs well and can locate part of the pelvis and learn to traverse to the middle, however, there are also many situations where the agent cannot locate any part of the anatomy. In the future, we hope to look tweak our model architecture by increasing the depth and other areas.

Future avenues we can pursue include different reward functions, different architectures, and different anatomies, or environments. The more complex architectures we implemented here should have at least performed comparably to the FC-DQN and Conv-DQN, so we should investigate this further. We would also want to extend to a 3D volume, which is more realistic in practice, and change from our square FOV to a slice FOV. To do so, we would need to add a rotation action in addition to the simple translations.

7. Conclusion

In our final Deep Learning project, we investigated a technique not learned in class called reinforcement learning. In particular, we applied the value function Q-learning framework to medical imaging data, specifically CT, to see if we could localize a specific FOV. We showed that with toy examples, Q-learning works very well, although it was not robust to different environments and often found local maxima. Additionally, Q-learning quickly becomes infeasible as the environment space increases. This led us to implement Deep Q-learning Networks (DQN) as a function approximator of the Q-table. With the vanilla DQN, we showed that the reward function was critical to the success of the model, which led us to switch from an intensity-based measure to a distance-based. From here, we explored different architectures of the DQN. We showed that a convolutional DQN improved upon a FC-DQN, and both greatly improved learning over a simple Q-table. However, we had issues implementing other architectures, perhaps due to the simplicity of our environment. From this project, we have seen the great adaptability of Q-learning and DQN models. We learned what reinforcement learning is, how to implement it, and what potential pitfalls can occur.

References

[1] A. Alansary, L. L. Folgoc, G. Vaillant, O. Oktay, Y. Li, W. Bai, J. Passerat-Palmbach, R. Guerrero, K. Kamnitsas, B. Hou, et al. Automatic view planning with multi-scale deep reinforcement learning agents. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 277–285. Springer, 2018.

[2] M. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable mdps. *arXiv preprint arXiv:1507.06527*, 2017.

[3] R. Koppula. Exploration vs. exploitation in reinforcement learning. `https://www.manifold.ai/exploration-vs-exploitation-in-reinforcement-learning`. Accessed: 2022-05-08.

[4] A. Lazaric, M. Restelli, and A. Bonarini. Agent with warm start and active termination for plane localization in 3d ultrasound. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 290–298. Springer, CHam, 2019.

[5] E. Levin, R. Pieraccini, and W. Eckert. Using markov decision process for learning dialogue strategies. In *Proc. of the 1998 IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP'98*, volume 1, pages 201–204. IEEE, 1998.

[6] M. McCormick et al. Itk: enabling reproducible research and open science. *Front Neuroinform.*, 8(13), 2014.

[7] V. Mnih et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[9] S. Paul. An introduction to q-learning: Reinforcement learning. `https://blog.floydhub.com/an-introduction-to-q-learning-reinforcement-learning`. Accessed: 2022-05-08.

[10] A. A. Tokuc. Value iteration vs. policy iteration in reinforcement learning. `https://www.baeldung.com/cs/ml-value-iteration-vs-policy-iteration`. Accessed: 2022-05-08.

[11] J. TORRES.AI. The bellman equation: V-function and q-function explained. `https://towardsdatascience.com/the-bellman-equation-59258a0d3fa7`. Accessed: 2022-05-08.

[12] X. Yang et al. Searching collaborative agents for multi-plane localization in 3d ultrasound. *Medical image analysis*, 72:102–119, 2021.