Reinforcement Learning for Autonomous Driving using CAT Vehicle Testbed

John Nguyen[*], Brandon Dominique[†], Hoang Huynh[‡], Eric Av[§], Rahul Bhadani[¶], Noel Teku[‖], Tamal Bose[**]

November 12, 2019

Abstract

We discuss a deep reinforcement learning implementation using the CAT Vehicle Testbed and discuss the merits of simulation based deep reinforcement learning. After implementing a simple 3 layered neural network which learned using deep Q-learning, we found some challenges associated with ROS and Gazebo. In spite of these challenges, we demonstrate that our reinforcement learning architecture can teach a car to avoid obstacles. With these preliminary results, we discuss what new things we can do and how we can implement more advanced methods.
In part 2 of this paper we introduce CRPy, a Cognitive Radio Testbed written in Python that is designed for Spectrum Sensing Simulations. We explain the need for a Testbed such as CRPy and its overlying structure. We then explain the purpose of each file within the CRPy framework, and how to use each file to generate a simulation. Finally, we conclude by discussing the Future Work that can be done to expand CRPy to other areas of Cognitive Radio.

Introduction

Reinforcement learning is a type of artificial intelligence that rewards and penalizes an agent on what action it takes given its current state. The state describes the agents environment and an action will effect this environment. The agent is penalized for failing the task and is rewarded for making progress on the task. Deep reinforcement learning utilizes a neural network to compute the action which will give the greatest reward. Due to the generality of reinforcement learning, researchers have been trying to build robots to accomplish complex physical tasks, such as driving [1] [2] [3]. This approach gained notoriety after the introduction of ALVINN [4], the first seminal paper in deep reinforcement learning for autonomous driving. The authors trained ALVINN to achieve 90 % accuracy in predicting whether to turn left or right, given an image of a road. They trained ALVINN using computer simulated pictures of roads. Though their methods are simple by modern standards, their results demonstrate the utility of deep reinforcement learning for autonomous driving. In this project, we attempt to train a similar driving agent, but with the use of simulations and laser sensors instead of simple images.

In reinforcement learning, the agent responds to its environment with a policy; a mapping from perceived states and possible actions. The reinforcement learning process refers to the agent trying to optimize its policy for maximum reward. Therefore, we try to balance exploration and exploitation. In general, the

[*]nguy2539@umn.edu, Corresponding Author, University of Minnesota
[†]dominiquebdmnq@aol.com, New Jersey Institute of Technology
[‡]s_hum@coloradocollege.edu, Colorado College
[§]eav@zagmail.gonzaga.edu, Gonzaga University
[¶]rahulbhadani@email.arizona.edu, The University of Arizona
[‖]nteku1@email.arizona.edu, The University of Arizona
[**]tbose@email.arizona.edu, The University of Arizona

agent attempts to exploit its knowledge about t7bhe environment in order to get a maximum reward. On the other hand, the agent must balance exploitation with exploring and getting new knowledge. If an agent spends too much time exploring, the agent may not be able to optimize its policy. On the other hand if the agent spends too much time exploiting, the agent may not recognize an entirely different action it could take which may lead to a better reward. To this end, we employ an epsilon decreasing strategy, where we have probability ϵ to explore and probability $1 - \epsilon$ to explore in each episode. The value of epsilon will change once we have explored enough of the environment.

In 2004, the DARPA Grand Challenge [5] was established, where t5eams were given sensors and instructed to train a car to drive through a desert near Barstow, California. Subsequent years had similar challenges. This challenge is often cited as igniting American interest in autonomous driving. In the past decade, a number of significant advancements in autonomous driving have been made [6] [7] [8]. This area of research has been popular due to commercial, military and civil interest. Some companies such as as Tesla and Uber have already implement their own fleets of autonomous vehicles. Unmanned drones are already used for military purposes. Public imagination is often captured by the perceived convenience of autonomous vehicles.

Training autonomous vehicles have infamously been difficult due to the difficulty in representing driving conditions. Many people have developed deep reinforcement learning methods for autonomous driving using images [4] [9] [10]. By attaching camera to the car, the agent can base its actions off of the video frames captured by the camera. Unfortunately, this method relies on computer vision algorithms to transform an image into a sequence of numbers. In our project, we represent a state as two floating point values: distance to the nearest object and angle to the nearest object. Therefore, we do not need to incorporate computer vision methods in our learning. Since we are focusing on a simple obstacle avoidance agent, we do not need the complex regression and classification algorithms used in more complex problems.

ROS and Gazebo

ROS (Robot Operating System) [11] is an open source robotics software used to help integrate the various parts of a robot together. A ROS program can be considered a graph of nodes, where each node represents a different process. In this project, we integrate the CAT Vehicle Testbed [12], which already utilizes ROS, with a new node that manages the deep reinforcement learning architecture. We use sensor data from the CAT Vehicle as input to our neural network, and the steering of the car as the output.

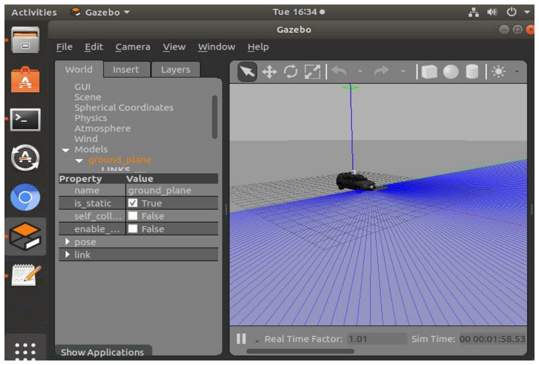
The CAT Vehicle Testbed is already compatible with the Gazebo simulator. Gazebo is a virtual environment that allows us to train and test the car in any virtual world. Since the Gazebo simulations are computationally intensive, it became necessary to restart Gazebo every training episode. Otherwise, the real time factor of the simulation would become alarmingly low. Since ROS does not exactly work in real time when low real time factor decreases, we find our training to be much slower than how a real life version of the simulation of the planned behavior.

ROS and Simulated World in Gazebo

Gazebo simulation environments can be opened through Ubuntu in various ways. The simulated environment in Gazebo is facilitated through a world file. A world file is essentially a file written in XML code that uses the ROS libraries to declare the environment variables for Gazebo to create in simulation.

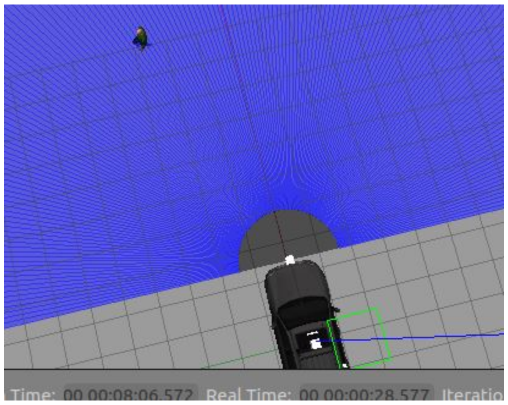
We first started with a basic world which includes the URDF model of the CAT Vehicle as seen in Figure 1. This world file's code is the basis for the rest of the training the control behavior in our simulation.

We created various other simulation scenario using world files to test basic obstacle detection and avoidance. The brick world for example 2b, was created as a simple detection and avoidance simulation example. The CAT Vehicle is enclosed in a bounding box as to help with the reset of the simulation when running the deep Q-learning algorithm. We want the car to crash into the bounding wall to communicate to the algorithm that it should restart the Gazebo world and choose different values for the next run. Considering the amount of processing power required to run the simulation effectively, we needed to wait hours before

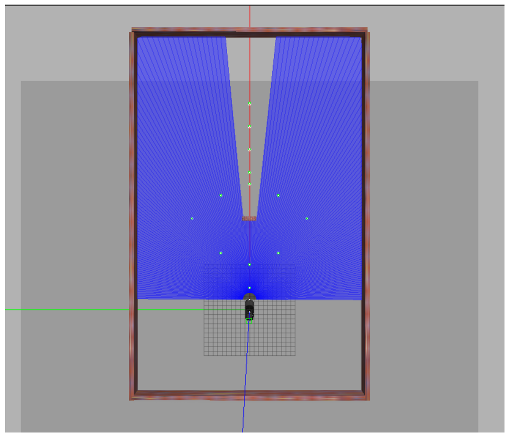


Property	Value
name	ground_plane
is_static	✓ True
self_coll...	False
enable_...	False
pose	
link	

Figure 1: Basic world with CAT Vehicle



(a)



(b)

Figure 2: (a). Human actor collision avoidance world (b). Waypoints worth extra reward points to encourage the CAT Vehicle to travel on them.

the agent showed any tangible results. We also created a world that included a human actor to simulate a pedestrian walking along a crosswalk for the car to detect and avoid. This world includes the XML code for the actor instead of including it in the launch file. However, we ended up simplifying our use cases scenario considering the amount of time it takes to train the neural network. We chose to focus more on obstacle collision and avoidance instead and ended up with a world with given rewarded way-points for the CAT car to maneuver while avoiding a brick wall.

ROS and Gazebo Launch Files

The ROS and Gazebo platform requires launch files to be executed in the command line for the simulation to appear in Gazebo. Launch files are essentially XML files that contain the initialization code for Gazebo which includes the world file path and any other paths desired. For our purposes, we also include the CAT Vehicle paths and topic paths to initialize the position and speed of our CAT Vehicle when Gazebo begins running the launch file.

This is not to confuse when first creating a world and robot directly from the start-up of Gazebo. Saving a world created in Gazebo saves an SDF format file which includes the XML code to interface with Gazebo packages but not necessarily ROS packages. Also, we need to specify paths and maneuver around logistical problems in order to add an SDF file to our launch file. For these reasons, launch files will only include

URDF modeled files and XML coded world files.

Experiment Design

In this case, the neural network (NN) is our agent, and it controls the car. For this reason, we will describe the actions of the car and the neural network as the agent. We used Deep Q-Learning to balance exploration and exploitation. Our neural network consists of 3 fully connected layers: an input layer, a single processing layer and the output layer. Our input layer takes in 2 arguments: the distance to the nearest object and the angle of the car to the nearest object. Due to sensor limitations, an object is only recognized if it is within 80 meters of the car. Both the input and processing layers have a ReLU activation function, and the output layer has a linear activation function. The output layer would provide action output from one of three actions: straight, turn left or turn right. The car was kept at a constant velocity of 2 m/s, and only the steering angle would change. Turning left would give the car -0.3 m/s of angular velocity. Turning right would give the car 0.3 m/s of angular velocity.

Our reinforcement learning system rewarded the car when car was following a path or avoiding obstacles, while penalized the car for crashing into things. Each time-step, the neural network is given a state and returns one of the three actions described above. Each state consisted of an ordered pair (d, θ). d represents the distance from the car to the nearest object, and $d \in [4.5, 80]$. θ represents the angle of the nearest object with respect to the front of the car and $\theta \in [-\frac{\pi}{4}, \frac{\pi}{4}]$. Should the NN tell the car to go straight, we reward +2 points. If the car turns left or right then we reward +1 point. This is to incentivize the car to follow the path it is currently on. If the car drives within 5 meters of an object, we simulate a crash and penalize the car with a large negative reward. Furthermore, there are waypoints in the environment which form a path. If the car drives close enough to any of these waypoints, it is rewarded. If the car went to every waypoint, the simulation would end and the car would receive a large positive reward. Due to time constraints and debugging, we ran the experiment for 292 episodes, and at most 2000 timesteps per episode.

When we want to train a new neural network, we first initialize the network and set up the deep Q-learning parameters. Afterwards, we start the episodes iteration. At the start of each episode, we launch the Gazebo simulation. After waiting a minute for everything to load properly, we take the initial observation and enter the timestep loop. At the start of each timestep, the car is fed the current state, and determines an action. Car executes this action and takes the next state. After the timestep loop is finished, either because the car crashed or this episode reached the maximum number of timesteps, we save the neural network and close the Gazebo simulation. We need to save the neural network because in the event our Gazebo simulation does not load properly, the program will crash. Saving the network allows us to load the network using another program and continue training. After closing the simulation, we have our program wait for two minutes, so all ROS associated programs are closed before we restart ROS. If we do not wait for sufficient time, two instances of ROS will attempt to run at the same time, which will cause our program to crash. In order to train the car, we needed to incorporate loading and closing the simulation and saving the neural network each episode.

Deep Q Learning Details

Neural Network Structure

Our neural network consists of three fully connected layers: an input layer, a single processing layer, and the output layer (Figure 3). Our input layer takes in two arguments: the distance to the nearest object and the angle of the car to the nearest object. Both the input and processing layers have a ReLU activation function, and the output layer has a linear activation function. The output layer would return predicted Q values of three actions: straight, turn left, and turn right. The loss function of the model is the mean squared error. We use Adam optimizer to update network weights.

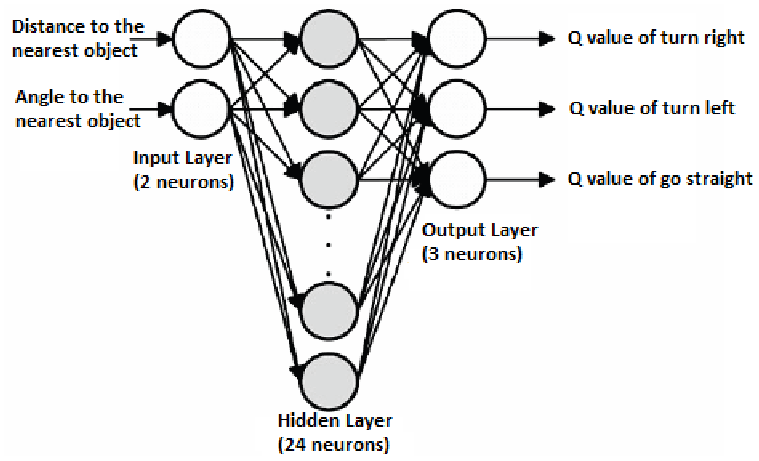


Figure 3: Representation of neural network used for training. Image modified from Cojbasic et. al [13]

Exploitation and Exploration

Exploration is the action of the CAT Vehicle to find information about the environment. Exploitation is the action of the CAT vehicle to maximize return based on known information. There is a trade-off between exploration and exploitation. If the CAT vehicle only exploits the environment, it may miss out opportunities to discover higher reward paths. However, if the vehicle always explores the environment, it cannot utilize known data to maximize return. Therefore, in order to balance the exploration and exploitation, we set the exploration rate to start at 1.0 and decay it by multiplying in 0.995 after each time step. This ensures that the CAT vehicle starts by exploring the environments and then exploits as it receives more information. [14]

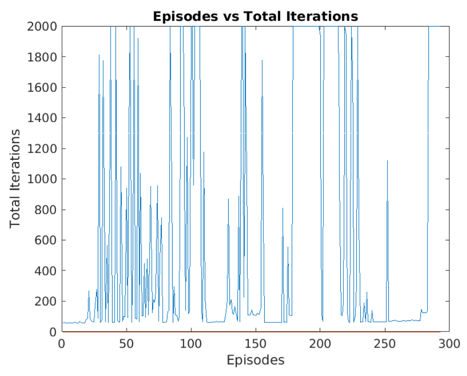
Experience replay

When the CAT vehicle sequentially obtains data from the simulation, the neuron network calculates the loss function based on consecutive samples, which are highly correlated and decrease learning efficiency. Therefore, we apply a technique called experience replay to increase learning process and minimize the undesirable temporal correlations [15]. A memory is created to store the sequential data. After each time step, we take a random batch of data in memory to train the model. Bellman optimality equation is used to update the Q values. We then calculate the loss based on the state and updated Q values.

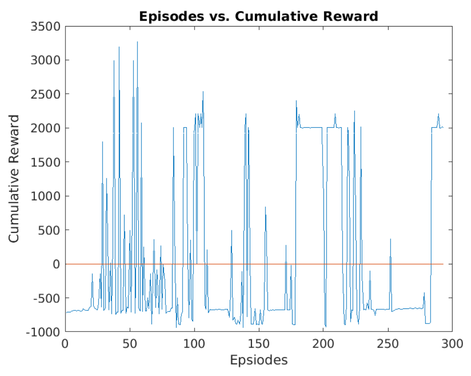
Training and Results

We trained the car using a simple Gazebo world simulation. The car was enclosed by brickwalls with a smaller brickwall (Figure 2). In our simulation, the car is the only moving object and can only drive forward, so the car cannot go in reverse. Due to the size of the car model and position of the laser scan sensor, the minimum distance the sensor can detect is 4.7 meters. For this reason, we determine the car has crashed if the sensor detects an object that is within 5 meters of the car. If the car crashes, the episode ends and the car receives a large penalty. If the car went within 1 meter of a waypoint, the car would recieve a moderate reward. If the car went to 10/13 waypoints, the episode will end and the car will receive a large reward. We limit each episode to running at most 2000 timesteps. If the car does not crash for all 2000 timesteps nor did it go to 10/13 waypoints, the simulation will end and the car will receive neither a large penalty nor a large reward.

For each timestep, the car can drive right, straight or left. If the car does this without crashing, it receives a small reward. We debated on whether to penalize the car for driving in each timestep, or to reward it. In theory, we should penalize the car for taking an action so it would be incentivized to follow the path. In



(a)



(b)

Figure 4: (a). The number of timesteps of each episode. (b). The cumulative reward at the end of each episode. The differences in (a) and (b) are due to the vehicle traveling to waypoints during the episode, resulting in a greater reward.

early training, we noticed a negative reward for taking an action would cause the car to crash itself upon spawning. This is likely because the path of waypoints is not dense enough, so the car does not follow its path. So assuming the car does not follow the the path and crashes, it will receive a more negative reward the longer it drives. Therefore if the car crashes faster, it will receive a less negative reward. To avoid this situation, we decided to reward the car for driving. Later in training, we realized the car was only following the path to an extent, and began circling the obstacle. Since the car was avoiding obstacles, the car gained a small reward each timestep. So instead of following the path, the car simply drove around the obstacle and avoiding crashes every timestep to maximize its reward. We chart the reward and episode length with respect to episode in the figure 4b.

Notice in Figure 4 that the reward per episode does not appear to follow a trend. This is likely due to some error in the way we saved and loaded the neural network. During training, the program had crashed a number of times. When loading the neural network, we were unable to recover some parameters of the parameters of our learning algorithm, such as the discounted epsilon. In particular, we were unable to recover the estimated discounted rewards for the current action and state. Due to this, the learning process was difficult. In order to alleviate these issues, we need methods to guarantee some parts of our program finish loading before others start. Due to the nature of real time simulations and crashing, our nerual network takes about 40 hours to train, assuming it does not crash. The problem of crashing and the time needed to train present interesting problems we will need to explore at a later date.

Conclusion

In this paper, we introduced a reinforcement learning architecture compatible with the CAT Vehicle Testbed. This enabled us to train the car in a simple Gazebo world we created. This example demonstrates how we can now use simulated worlds with complex physics engines for deep reinforcement learning. Training autonomous vehicles via simulations have been historically difficult due to needing to simulated the physics of actually driving. Since the CAT Vehicle already uses ROS and Gazebo, we can control the car in a simulation for its training. We created a deep reinforcement learning architecture for autonomous driving which can be trained using widely used open source software with a physics engine.

Further exploration into this topic is necessary. Now that we have built the reinforcement learning framework for the CAT Vehicle Testbed, we can begin to incorporate more complex learning methods and test on how these methods effect deep reinforcement learning for autonomous driving. In particular, we intend to implement meta-learning methods to attempt to train the car more quickly. Meta-reinforcement learning attempts to teach a neural network the ability to abstract problems. For instance, we managed to teach a neural network to control a car to avoid a stationary obstacle by circling the obstacle. In the further work, we would like to train the network to also avoid the moving obstacles, such as a person. Unfortunately, using our current architecture, we would need to train the car for this specific scenario. Meta-reinforcement learning abstracts stationary and mobile obstacles avoidance into simple obstacle avoidance. Ideally, we would want to place the car in a never before seen obstacle course with stationary and mobile obstacles, and have the car avoid all obstacles and reach its destination with minimal training. Due to the challenge of training autonomous vehicles, it is worthwhile to attempt to maximize desired policies while minimizing training.

The foremost in our method is how the simulation crashes when parts of the program does not load in order. Instead of relying on sleep statements and hoping parts of the program load in time, it would be best to have the program wait until necessary parts of the code have loaded before continuing to execute. Since we can utilize Gazebo simulations, the car can be trained in many other worlds and situations. One idea is to simulate highway driving scenarios. Some work has already been done on this topic [16], but was implemented on a scaled city. Once the car is sufficiently trained, implementing the policy with the CAT Vehicle on an actual highway. Another avenue to explore is creating a simulated city in Gazebo and training the car in urban driving. We have introduced a reinforcement learning architecture compatible with Gazebo and ROS, so we can train the car in any environment as long was we can create the environment.

Part 2: CRPy

Introduction and Background

Cognitive Radio (CR) is a technology that has shown promise for Autonomous Vehicles(AV), and with the correct implementation can become the standard of vehicular communication in the future [17]. Spectrum space is dominated by Licensed or Primary Users, and that leaves Secondary Users such as AVs with a limited amount of space to transmit messages to other cars (Vehicle to Vehicle) and the infrastructure around them (Vehicle to Infrastructure). Vehicular communication in AVs requires radios that are able to adapt to their environment and modify their modulation and channel coding accordingly [18]. There are many techniques towards optimizing CRs [18], but perhaps the most advanced field of CR technology is in Spectrum Sensing [19]. Research with Spectrum Sensing for CRs has been producing positive results in almost every scenario that it has been applied to, with a large factor in its success being due to the rise of Reinforcement Learning [2]Use cite command and not [2] directly.. Reinforcement Learning is a branch of Machine Learning that involves setting up an agent and an environment that the agent can interact with. The agent is given states that it can exist within and a set of action that it can execute at any given state [20]. The most popular algorithm, Q-Learning, involves training a Cognitive Radio with some type of positive reward for making the correct actions in a given state and a penalty for an incorrect action [21] [22] [23]. Simulation results with RL-enabled CR has been overwhelmingly positive, but there are still some problems that arise within the field. Perhaps the biggest problem in the field lies within the simulation software that is currently being used. As of now there is no agreed upon testbed for CR Simluation; different papers use different languages and testbeds to generate their simulation results, and the code to set up a simulation to try and replicate their results isnt always available. Existing CR Testbeds such as [24] and [25] attempt to eliminate this problem, but they require a decent amount of setup and arent easy to implement. In the case of [24] specifically, the package is a wrapper around a C++ foundation, meaning that knowledge of both languages is required in order to implement a novel experiment with that Testbed. With these thoughts in mind we present our package CRPy, a Cognitive Radio Testbed that is written entirely in Python and allows users to train a cognitive network for Spectrum Sensing & Threshold Detection with the RL algorithms provided, or their own that they write. The rest of the paper is divided in the following way: Section 2 discusses the structure and features of the CRPy, Section 3 describes the files that populate CRPy and their uses, and Section 4 shows an example of CRPy training an agent. Finally, we summarize our work and suggest future research avenues with CRPy in the Conclusion.

Rahul

Section 2: Goals of CRPy

The main goals that were in mind while writing CRPy was simplicity and scalability. The decision to write the entire package in Python stems from these two points; CRPys easy to understand classes and functions means that anyone who wants to use the software will be able to use it easily and expand it to whatever project theyre working on. CRPY is made up of 3 files: mpsk.py, agents.py, and sim.py. The 3 files work in tandem to create the RL agents and the Environment that theyre going to explore, which by default is a Spectrum Sensing Simulation with 5 different channels the agents must differentiate from in order to send a PSK modulated message. CRPy draws inspiration for coding and design decisions from various sources ([26], [27], [28]) and incorporates elements of all of them into the functionality of Python. To install CRPy into your Python environment, use the command [pip install CRPy] (Shown in Figure 5), and then to use it import it as with any other Python package.

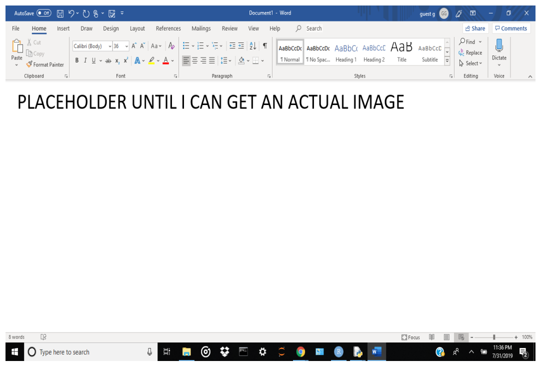
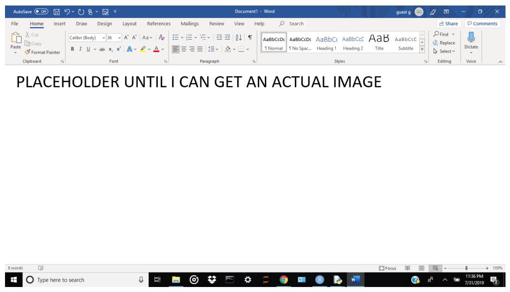

PLACEHOLDER UNTIL I CAN GET AN ACTUAL IMAGE

Figure 5: Installation of CRPy

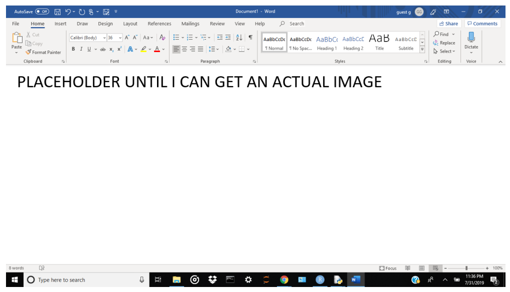
Section 3: Included Files

3.1: mpsk.py

Mpsk.py is designed to aid in the simulation of the PSK modulation of an array of bit strings. As of now, Mpsk.py can modulate up to 8 PSK. The class mpsk takes the modulation number, the desired Signal to Noise Ratio(SnR) in dB, and the carrier frequency of the signal. Mpsk.py is capable of modulating and demodulating both baseband (complex signals of the form a+bj mapped to a reference constellation) and passband (the real part a of the complex signal converted into a cosine wave and the imaginary part b converted into a sine wave) signals. Both types of signals are first created by using the modulate() function, but the passband signals can be created by using the digital2analog() function.


PLACEHOLDER UNTIL I CAN GET AN ACTUAL IMAGE

(a)

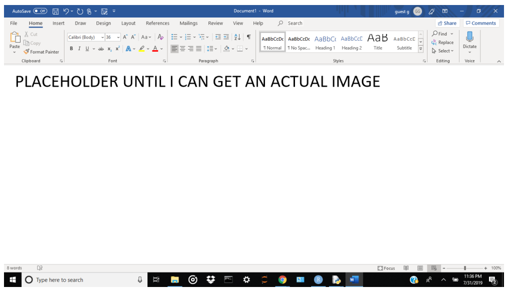

PLACEHOLDER UNTIL I CAN GET AN ACTUAL IMAGE

(b)

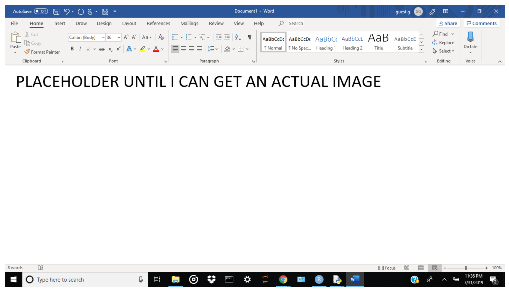
Figure 6: (a). modulate() and (b). digital2analog().

There are currently 2 functions for adding noise to the modulated signal: awgn() and rayleigh(). If awgn() is used, the modulated signal x(t) that is passed into the function will be returned with noise n, x(t) + n. Place any math equation either within equation command or within $$ sign. rayleigh() multiplies ***Rahul*** the signal by a constant h and then uses awgn() to output h*x(t) + n. The amount of noise added, like with most packages, is dependent on the SnR that was given when the class was first initialized; the closer the SnR to 0, the more probable that some bits will change their value due to noise. Each function almost perfectly follows the ideal Bit Error Rate (BER) curve for a given SnR, shown in Figure 7 (BPSK Only). 50000 random bits were modulated and demodulated at different SnRs and the accuracy of each SnR is recorded. Note that it is also an option to use neither of these functions and add no noise to your signal in our implementation.

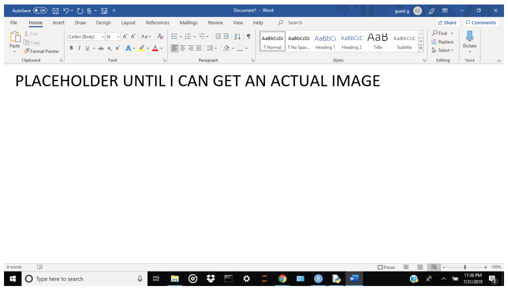
If the signal is a passband signal, it must use the function analog2digital() before it uses the demodulate() function; if its a baseband, analog2digital() can be skipped and the signal can go straight into demodulate().


PLACEHOLDER UNTIL I CAN GET AN ACTUAL IMAGE

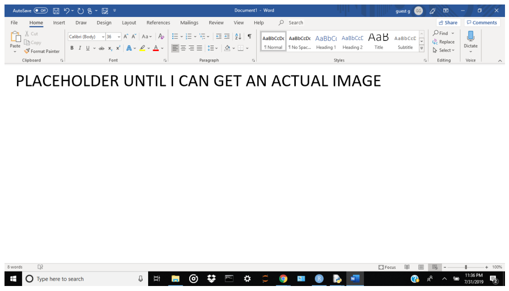
(a)


PLACEHOLDER UNTIL I CAN GET AN ACTUAL IMAGE

(b)


PLACEHOLDER UNTIL I CAN GET AN ACTUAL IMAGE

(c)


PLACEHOLDER UNTIL I CAN GET AN ACTUAL IMAGE

(d)

Figure 7: The ideal BER for a given SnR for AWGN (a) and Rayleigh (b) Noise and the BER for a given SnR with our functions awgn() (c) and rayleigh() (d).

The output of demodulate() is an array of bits after computing the Euclidean distance between the signal that was passed into it and the closest point on reference constellation that was created for the signal. This is shown in Figure 8.

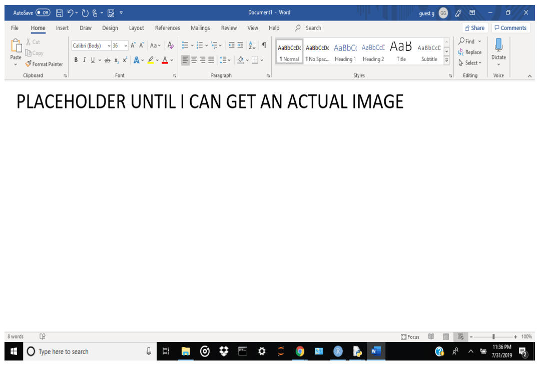
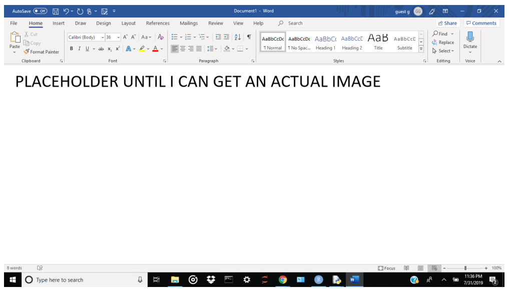
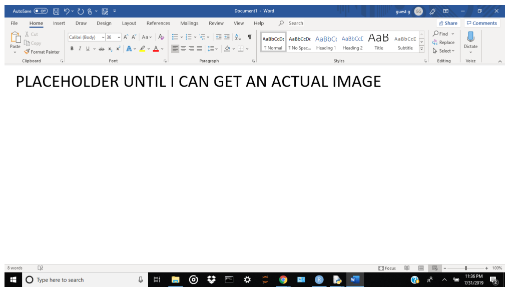

PLACEHOLDER UNTIL I CAN GET AN ACTUAL IMAGE

Figure 8: analog2digital() and demodulate()

The functions send() and receive() combine the functions described above to reduce the labor of calling each one when the user wants to modulate or demodulate a signal. Besides the bit array, send() takes an argument noise, which by default is set to None; to specify awgn or rayleigh, this argument must be changed to awgn or rayleigh when the function is called. send() also produces a graphical output of the signal as its being modulated to give the user a visual of their modulated signal, shown in Figure 9a. There is also an error_rate() function (Figure 9b) which calculates the BER of the mpsk object with the SnR that was given at the time of creation. This aids the user in determining how noisy their signal will be before it is sent.


PLACEHOLDER UNTIL I CAN GET AN ACTUAL IMAGE

(a)


PLACEHOLDER UNTIL I CAN GET AN ACTUAL IMAGE

(b)

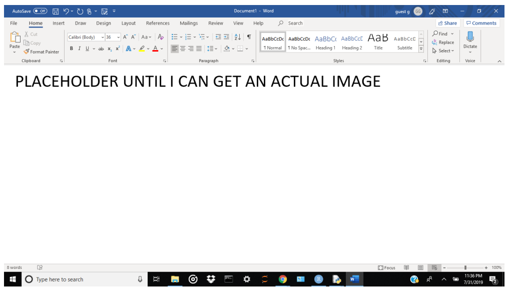
Figure 9: The graphs that are produced when send() is called, and an example of when error_rate() is called
.

3.2: agent.py

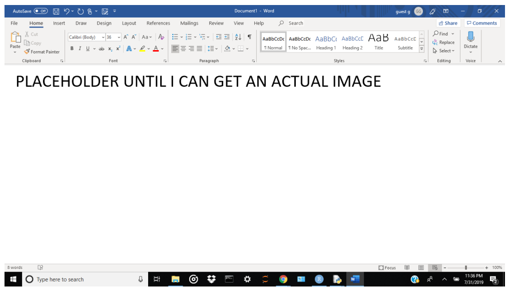
Agent.py houses the two default RL algorithms that can be used in the CR Simulation. In the default RL environment, an episode (the period of time where the agent begins using different actions in different states, continuing until there are no more actions left to take) begins when the agent is given information about the environment and its current state (its observation for the episode); for our purposes of Spectrum Sensing, the observation is an array of 5 values - the amount of energy on each of the 5 channels that are set up by default by simulation.py. It's the task of the agent to look at the energy levels and select from them the channel that it can send data on (it will always be the one with the lowest amount of energy). Ideally all of the channel have the same PSK modulation number and different SNRs. The First RL Algorithm, Q Learning, can be classified as a single state agent; rather than creating an environment where the agent can take multiple actions in multiple states, we find that allowing only one state per episode reduces the complexity of the algorithm considerably [22]. The Second Algorithm, Upper Confidence Bound, begins by examining each of the actions available to it once; in this simulation, this means that it will attempt to send a message at each of the 5 channels once. Afterwards, the algorithm determines the channel to send messages on by following the formula

$$Q(j) + \sqrt{(2ln(N))/N_j} \tag{1}$$

where N is the number of total episodes and N_j is the number of trials specifically for the action j (In this case, j represents each channel). The action that has the highest value from this formula is the one that is chosen. In Figure 10 is the performance of each Algorithm after 1000 episodes.


PLACEHOLDER UNTIL I CAN GET AN ACTUAL IMAGE

(a)


PLACEHOLDER UNTIL I CAN GET AN ACTUAL IMAGE

(b)

Figure 10: The performance of QLearning() and UCB() after 1000 episodes.

3.3: simulation.py

simulation.py still needs to be made.

Section 4: Implementation of CRPy

This section will show the implementation of the described CR Spectrum Sensing Simulation using Energy Detection, but simulation.py needs to be built first.

Conclusion and Future Work

We introduced CRPy, a Simulation Testbed for testing Spectrum Sensing in Cognitive Radio. CRPy has value in the fact that it was written entirely in Python, which makes it easy to learn how to use and create new algorithms and environments. In the future, CRPy can be expanded to include more modulation techniques (16-PSK, QAM) that would help to further resemble a realistic CR Environment. The number of default environments available to the user can also be expanded, which would allow for other areas of CR to be explored [18]. Finally, the performance of this Simulation Testbed can be compared to other Testbeds ([24], [25]) to gauge the performance of this software with the more established Testbeds.

References

[1] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars.," *CoRR*, vol. abs/1604.07316, 2016.

[2] U. Muller, J. Ben, E. Cosatto, B. Flepp, and Y. L. Cun, "Off-road obstacle avoidance through end-to-end learning," in *Advances in Neural Information Processing Systems 18* (Y. Weiss, B. Schölkopf, and J. C. Platt, eds.), pp. 739–746, MIT Press, 2006.

[3] J. Koutník, G. Cuccu, J. Schmidhuber, and F. Gomez, "Evolving large-scale neural networks for vision-based reinforcement learning," in *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, GECCO '13, (New York, NY, USA), pp. 1061–1068, ACM, 2013.

[4] D. A. Pomerleau, "Advances in neural information processing systems 1," ch. ALVINN: An Autonomous Land Vehicle in a Neural Network, pp. 305–313, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989.

[5] R. Behringer, S. Sundareswaran, B. Gregory, R. Elsley, B. Addison, W. Guthmiller, R. Daily, and D. Bevly, "The darpa grand challenge - development of an autonomous vehicle," in *IEEE Intelligent Vehicles Symposium, 2004*, pp. 226–231, June 2004.

[6] E. Santana and G. Hotz, "Learning a driving simulator," *ArXiv*, vol. abs/1608.01230, 2016.

[7] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4013–4021, 2015.

[8] M. Wulfmeier, D. Z. Wang, and I. Posner, "Watch this: Scalable cost-function learning for path planning in urban environments," *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 2089–2095, 2016.

[9] R. Hadsell, P. Sermanet, J. Ben, A. Erkan, M. Scoffier, K. Kavukcuoglu, U. Muller, and Y. LeCun, "Learning long-range vision for autonomous off-road driving," *J. Field Robot.*, vol. 26, pp. 120–144, Feb. 2009.

[10] A. Giusti, J. Guzzi, D. Ciresan, F.-L. He, J. P. Rodriguez, F. Fontana, M. Faessler, C. Forster, J. Schmidhuber, G. Di Caro, D. Scaramuzza, and L. Gambardella, "A machine learning approach to visual perception of forest trails for mobile robots," *IEEE Robotics and Automation Letters*, 2016.

[11] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[12] R. K. Bhadani, J. Sprinkle, and M. Bunting, "The cat vehicle testbed: A simulator with hardware in the loop for autonomous vehicle applications," *arXiv preprint arXiv:1804.04347*, 2018.

[13] Z. Cojbasic, V. Nikoli, E. Petrovic, V. Pavlovi, M. Tomi, I. Pavlovi, and I. iri, "A real time neural network based finite element analysis of shell structure," *Facta universitatis series Mechanical Engineering*, vol. 12, pp. 149–155, 06 2014.

[14] J. G. March, "Exploration and exploitation in organizational learning," *Organization Science*, vol. 2, no. 1, pp. 71–87, 1991.

[15] R. Liu and J. Zou, "The effects of memory replay in reinforcement learning," *CoRR*, vol. abs/1710.06574, 2017.

[16] K. Jang, E. Vinitsky, B. Chalaki, B. Remer, L. Beaver, A. A. Malikopoulos, and A. Bayen, "Simulation to scaled city: zero-shot policy transfer for traffic control via autonomous vehicles," in *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*, pp. 291–300, ACM, 2019.

[17] H. Asadi, H. Volos, M. Marefat, and T. Bose, "Metacognition and the next generation of cognitive radio engines," *IEEE Communications Magazine*, vol. 54, pp. 76–82, 01 2016.

[18] K.-L. Yau, G. S. Poh, S. Chien, and H. A A Al-Rawi, "Application of reinforcement learning in cognitive radio networks: Models and algorithms," *TheScientificWorldJournal*, vol. 2014, p. 209810, 06 2014.

[19] A. Shahid, S. Ahmad, A. Akram, and S. Khan, "Cognitive ale for hf radios," *Computer Engineering and Applications, International Conference on*, vol. 2, pp. 28–33, 03 2010.

[20] A. Ng, "Cs229 lecture notes," *CS229 Lecture notes*, vol. 1, no. 1, pp. 1–3, 2000.

[21] T. Anwer Sohan, H. Hamidul Haque, M. Asif Hasan, and I. Jahidul, "Investigating the challenges of dynamic spectrum access in cognitive radio-enabled vehicular ad hoc networks (cr-vanets)," pp. 1–6, 05 2015.

[22] T. Jiang, Q. Zhao, D. Grace, A. Burr, and T. Clarke, "Single-state q-learning for self-organized radio resource management in dual-hop 5g high capacity density networks," *Transactions on Emerging Telecommunications Technologies*, vol. 27, pp. n/a–n/a, 02 2016.

[23] D. B. Rawat, Y. Zhao, G. Yan, and M. Song, "Crave: Cognitive radio enabled vehicular communications in heterogeneous networks," 01 2013.

[24] P. Gawłowicz and A. Zubow, "ns3-gym: Extending openai gym for networking research," *arXiv preprint arXiv:1810.03943*, 2018.

[25] T. Chigan, "Cognitive radio cognitive network simulator (ns3 based)."

[26] M. Viswanathan, *Digital Modulations using Matlab: Build Simulation Models from Scratch*, ch. 1,2,3. New York: Mathuranathan Viswanathan, 2 ed., 2017.

[27] J. Hossain, "Simulation of a cognitive radio system by using matlab," 06 2013.

[28] B. S. K. Reddy and B. Lakshmi, "Ber analysis of energy detection spectrum sensing in cognitive radio using gnu radio," 2014.