Reinforcement Learning Applied to Cognitive Space Communications

Carson D. Schubert
University of Texas at Austin,
Austin, TX.
carson.schubert@utexas.edu

Rigoberto Roché
NASA Glenn Research Center,
Cleveland OH.
rigoberto.roche@nasa.gov

Janette C. Briones
NASA Glenn Research Center,
Cleveland, OH.
janette.c.briones@nasa.gov

Abstract— The future of space exploration depends on robust, reliable communication systems. As the number of such communication systems increase, automation is fast becoming a requirement to achieve this goal. A reinforcement learning solution can be employed as a possible automation method for such systems. The goal of this study is to build a reinforcement learning algorithm which optimizes data throughput of a single actor. A training environment was created to simulate a link within the NASA Space Communication and Navigation (SCaN) infrastructure, using state of the art simulation tools developed by the SCaN Center for Engineering, Networks, Integration, and Communications (SCENIC) laboratory at NASA Glenn Research Center to obtain the closest possible representation of the real operating environment. Reinforcement learning was then used to train an agent inside this environment to maximize data throughput. The simulation environment contained a single actor in low earth orbit capable of communicating with twenty-five ground stations that compose the Near-Earth Network (NEN). Initial experiments showed promising training results, so additional complexity was added by augmenting simulation data with link fading profiles obtained from real communication events with the International Space Station. A grid search was performed to find the optimal hyperparameters and model architecture for the agent. Using the results of the grid search, an agent was trained on the augmented training data. Testing shows that the agent performs well inside the training environment and can be used as a foundation for future studies with added complexity and eventually tested in the real space environment.

Keywords—switching, machine learning, reinforcement learning, satellite communications, space links

I. Introduction

Recent years have seen a renewed interest in space exploration, travel, and business. Enabling all of these exciting developments are reliable communications. A modern space asset in low earth orbit (LEO) experiences almost continuous communications uptime, thanks in part to a myriad of relay satellites and ground stations. However, this cannot occur without constant effort and supervision by ground personnel, who have to manually schedule communications passes and link configurations. This has been the mode of operations for decades. An automated system, capable of handling communications requests from space assets and ensuring all are properly served, will free up thousands of man hours. This could enable reliable communications for years to come, as the number of space assets and their complexity continues to rise [1]. Such a system and the technologies that support it are known as cognitive communications. A critical component of a cognitive communications system is the assets themselves, which should always operate in their own self-interest. For example: A rover on Mars has no concept of its priority among other space assets; all it knows is that data has been collected, and that data needs to be sent back to Earth. The purpose of this study is to present an alternative method, capable of removing human interaction from the process of data downlink and replace it with an algorithm which automatically optimizes a point to point link between multiple ground stations and a space orbiting asset.

Unfortunately, data are not readily available for developing such an algorithm. This necessitates the creation of a high fidelity simulation environment within which an agent can be trained to carry out the desired task. This method of machine learning is known as Reinforcement Learning and is discussed later on in this study. Due to the complexity of the communications problem, the exact optimal behavior of the reinforcement learning algorithm cannot be learned in any reasonable amount of time. Instead, we approximate this behavior using a neural network trainable in a matter of hours.

A. Neural Networks

Neural networks, as one might expect, are modeled after the human brain. Individual neurons are connected together and activate in response to an input to produce an output. Each neuron sums weighted inputs feeding into it, then passes the weighted sum through a non-linear activation function to produce its output. In this way, some neurons are activated while others are not depending on the given input.

Neural networks are composed of an input layer, one or more hidden layers, and an output layer. The hidden layers serve to convert the input into learned useful features. The output layer serves to map the abstract features learned by the hidden layers into usable output. In the quintessential example of binary classification, this output would be a two dimensional vector whose elements correspond to the probability that the input is in that particular class.

Neural networks are capable of representing nearly any continuous function of n real variables [2]. However, the difficulty arises when attempting to learn the correct weights to represent the desired function. Many complex architectures have been designed which improve performance in various domains including computer vision, natural language processing, system security, automation of controls,

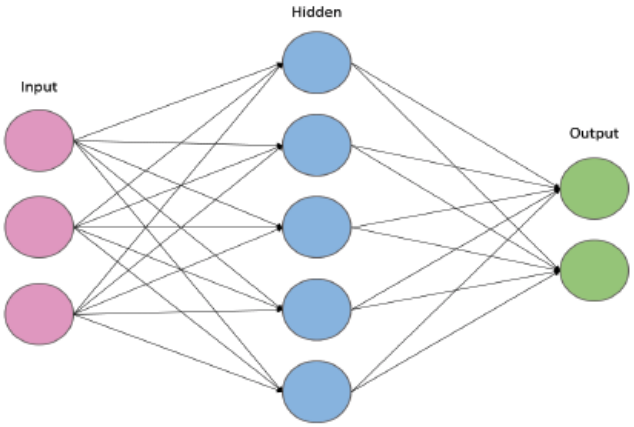

Fig. 1. Single layer neural network

processing, and forecasting. These neural networks are capable of learning to solve a wide variety of problems and are commonly used for tasks such as classification and prediction.

The most basic architecture is referred to as a fully connected feed-forward network and is shown in Fig 1. The input and output sizes are determined by the data being fed through the network and the desired output. In contrast, the hidden layer architecture is up to the designer of the network. The number of hidden layers, their respective sizes, and their respective activation functions are all tuned to the specific problem being solved by the network. This architecture is capable of learning to solve a wide variety of problems.

B. Reinforcement Learning

A reinforcement learning agent interacts with the environment over time. At each timestep, the environment provides a state to the agent which takes an action, resulting in a numerical reward and new environment state. The chosen action is based on the agent's policy, which maps a given state to an action. Many problems can be framed in an episodic manner, whereby this process continues until a terminal state is reached. The total discounted reward for an episode is the addition of all rewards obtained for each action, in each state, multiplied by their discounts.

Discounting the rewards over time, such that those received in the present are valued higher than those received in the future, solves the credit assignment problem, in most cases, for distributing credit of the total reward, among all decisions involved in producing it. It is important to note, however, that not all problems require a discount rate be used at all.

The challenge of any reinforcement learning problem is to discover the optimal policy. A policy can be evaluated using the action-value function, which is a prediction of the expected cumulative future rewards, given a current state or state-action pair and a current policy. Given a specific state, one can define the state value as the value of following a given policy in the action-value function. An optimal state or action value is the maximum value achievable by following any policy from a specific state. Solving for the optimal value, these functions can be decomposed into Bellman equations, solvable by dynamic programming, when a full model of the system is available, as in the case of an optimal control problem.

TABLE I. ACTION VALUE TABLE FOR A NAVIGATION MAZE

	Blocked	***Not Blocked***
Move Forward	0	1
Turn Randomly	1	0

However, in the real world, a full system model is rarely available and thus we look to reinforcement learning methods to determine the optimal policy. An intuitive method is Q-Learning (so called because the action value function is usually represented by the letter Q), which seeks to learn the action-value function. One can then evaluate each potential action in a given state and select the one with the highest value. In cases where the state/action spaces are small, it is possible to directly store each action value for every state-action pair in a tabular form (a Q table). An example for a self-driving car navigating a maze is shown in Table I, where Blocked and Not Blocked are the possible states and Move Forward and Turn Randomly are the possible actions.

As environmental complexity increases, it becomes unfeasible to explicitly store all state-action values. In these cases we use function approximation to generalize the optimal policy from a limited number of examples. We can use a policy gradient method such as a reinforce algorithm [4], where the agent takes small steps and updates its policy based on the reward received for each step. Q-Learning can also use function approximation methods to make it tractable, in complex state/action spaces using a Deep Q-Network [5], where the network learns to represent complex states in a dense embedding usable as value estimators. Most modern problems require the use of function approximation methods, and many are too complex for simple linear approximations. Therefore, the neural network approach for approximating for the optimal policy of the reinforcement learning agent, is used.

C. Simulation Environment

The SCaN program at NASA is actively developing a tool for rapid analysis of space communications architecture and systems. This SCENIC tool provides a web based user interface to simplify analysis using drag and drop techniques. The underlying simulation was built using domain expertise to accurately model radio communications between nodes anywhere in the Solar System. This simulation presented an immediate solution to the problem of developing a high fidelity environment for training reinforcement learning agents. However, much work was needed to wrap the SCENIC functions into a system capable of exposing the agent to a wide variety of communications environments.

Due to the nature of the SCENIC provided functions and the need for simulation flexibility, it was decided to decouple the simulation itself from the environment the agent interacts with. Generation of simulation data is episodic, handled by a versatile MATLAB function which calls relevant SCENIC functions to evaluate contact windows and link characteristics throughout an episode. At the conclusion of an episode, simulation data is saved to storage for future use in the reinforcement learning

environment. This makes it possible to update, change, or otherwise alter the simulation without requiring a rewrite of agent or training code, so long as the simulation continues to output data in a format usable by the reinforcement learning environment.

Data generation is configurable via a Javascript Object Notation (JSON) file loaded in at runtime to set episode duration and orbital and link parameters. The simulation function also accepts arguments which set ground station presence and the total number of episodes to simulate. Orbital inclination was determined to be the most important variable in generating representative data, as this has the most direct influence on the ground stations, an orbiting satellite will see, during an episode. Due to its importance, the simulation function randomly samples from a uniform distribution of possible inclinations for each episode. The seed for this random generation is taken from system time so as to alter the distribution of inclinations for each call. However, one can explicitly provide a seed to the function in order to re-run a specific simulation in addition to an offset, which will increment the random sampler that many times before beginning execution. This functionality allows one to re-simulate any episode produced by the function.

Ground stations are defined by JSON files loaded in at runtime. The use of external JSON makes adding, removing, or modifying ground station presence and operating parameters simple. By default, the simulation function supports bulk loading of all Near-Earth Network assets but also provides the option for a custom set of ground stations.

Simulation state is evaluated once per second inside the SCENIC functions. After a simulation episode is complete, desired metrics are extracted from the episode at a one second resolution and placed into a time-indexed table which is exported to storage as a comma separated values (csv) file. An example of the parameters and values from the csv output can be seen in Table II. The purpose of this table is to illustrate the variety of the format in the episode metrics, not to explain what they represent. These parameters can be used as features for a possible reward function. This feature selection will be explained later on in this study. As observed in the table they vary from integers to floating point number to strings.

Metadata about the configuration that generated these files are encoded into the file names and exported as csv for each episode. These episodes can then be loaded and used for training within a custom Gym environment built for this study and written in Python. A flow diagram for the system can be seen in Fig 2. Gym is an open source toolkit from OpenAI for developing reinforcement learning algorithms. It provides a set of standard environments reinforcement learning researchers can use for testing and comparison of models and is widely used, especially by those just starting out in the field of reinforcement learning; this is because all Gym environments follow a strict and intuitive API which makes interacting with them simple. Extending the base Gym environment and adding the necessary custom functionality makes it possible to verify a training pipeline on a basic Gym environment, then switch to our custom environment with little friction.

TABLE II. PARAMETERS AND VALUES FROM EXPORTED EPISODE

Parameter	***Value***
satX	-3294.98
satY	-6514.07
satZ	1.95
Ln_name	SGI_receive_X
Ln_x	-1524.66
Ln_y	6191.31
Ln_z	154.37
Ln_recvPower	-158.29
Ln_rangeRate	-5.83
Ln_recvEsNO	16.3
Ln_dataRate	1.69
Ln_maxBandwidth	40
Ln_BER	3.23e-11
Ln_modcod	QPSK_Uncoded

The Gym environment continuously generates packets of a specified size in bits for the agent to downlink. The agent chooses a link at each time step, and the remaining bits in the packet are decremented according to the data rate of the chosen link. Three different numerical rewards are given to the agent: 0.1 for selecting a non-real link, 1 for selecting any real link, and 60 when a full packet is downlinked. The difference between a real and non-real link is further discussed later on in this study.

II. METHODS

A. Simulation Setup

The simulation was populated with twenty-five antennas from twelve different ground stations which compose the NEN, NASA's global communications network for satellites and missions in LEO, Geosynchronous Orbit (GEO), and lunar orbit.

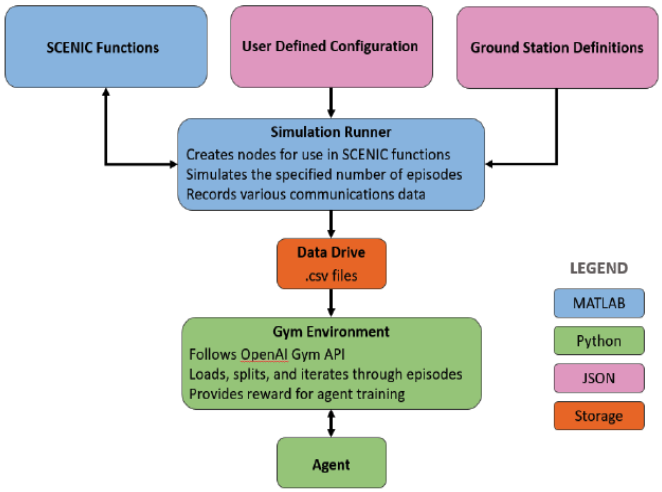

Fig. 2. Data profile used for independent testing for validating generalization.

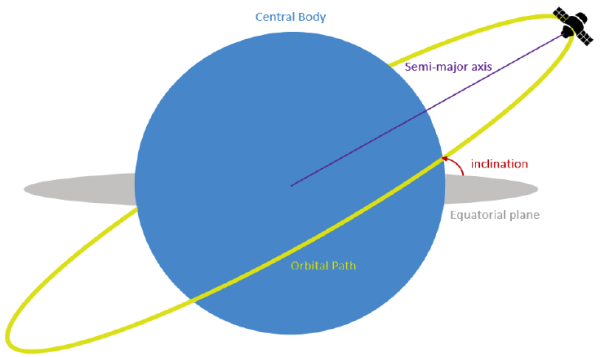

Fig. 3. Keplerian elements of inclination of semi-major axis

The parameters for these antennas were obtained from an official NEN asset database provided by the SCENIC group. LEO was selected as the training ground for the agent, as it contains 62.8% of satellites in orbit around the Earth today [6]. The agent was given the ability to communicate with all twenty-five ground station antennas, with specified communication parameters.

The specific values of these parameters are not of great importance, as they merely influence the resultant link characteristics. The goal of this study is to produce an algorithm which understands the relative importance of these characteristics rather than their explicit values; so long as the simulation data is realistic, it will suffice for training. The agent is placed into orbit around the earth during each episode. The orbit of the agent directly determines the contact windows and link quality it will have with each ground station during a simulation episode. Orbits are completely defined by six parameters, called Keplerian elements. Most important among these are the eccentricity, inclination, and semi-major axis. Eccentricity is between zero and one for normal orbits (non-escape trajectories), where 0 represents a perfectly circular orbit. For simplicity, the eccentricity was set to zero for all training episodes. Inclination refers to the angle between the orbital and equatorial planes and is between zero and ninety degrees for typical Earth orbits. Because the eccentricity was frozen at zero, the semi-major axis takes on a simpler definition; merely the distance between the centers of the orbiting bodies. These definitions are visualized in Fig. 3.

B. Data Generation and Augmentation

A representative set of training episodes are necessary to effectively train an agent capable of succeeding in the real world environment. This was achieved by varying the starting epoch of the simulation along with the inclination and semi-major axis of the agent's orbit. Over 3000 episodes in total were generated using this method.

At the highest level of episode generation, the starting epoch of the simulation was varied. For each start date, 150 episodes were generated. The start date was incremented by twenty to twenty-five days after each 150 episodes so that the majority of the calendar year would be represented. The start date influences the initial position of the satellite over the Earth's surface.

Within each 150 episode batch, the semi-major axis of the agent's orbit was varied, starting at 6800km and increasing by 500km every 50 episodes. 6800km was chosen as the baseline based on the International Space Station (ISS), which has an orbital period of ninety minutes. As the semi-major axis of a circular orbit increases, so too does the orbital period. Thus, by keeping the episode duration at a constant 90 minutes and the semi-major axis above that of the ISS, we guarantee that the agent experiences nearly a full orbit per episode while avoiding repeated communications passes in a single episode.

Finally, the inclination of the agent's orbit was varied on an episode by episode basis according to the sampling of the simulation function. All episodes within a batch of fifty at a specific semi-major axis and start date were generated with the same random seed to ensure an even distribution of inclinations within the batch. The seed was changed from batch to batch.

A subset of 750 episodes was then augmented with real link fading profiles obtained from communications between Glenn Research Center and the ISS. Fading is a link condition that occurs during periods of high interference characterized by erratic, signal to noise ratios which make communication difficult or impossible. One of the main benefits of the switching algorithm developed in this study is its ability to autonomously handle and work around poor communications conditions like fading. These conditions are difficult to simulate, so the decision was made to use collected data instead to verify agent performance in such conditions.

A database of Es/No (energy per symbol to noise power spectral density ratio) values from eleven communications events was provided, with values separated into one hundred element batches labelled as either being part of a fading event or not. Episodes were stepped through one row at a time using a script which tracked available links throughout each contact window. Every 100 seconds during the contact window for a link a batch of Es/No values were sampled from the database with a 50/50 probability of being a fading batch. The Es/No values of the link were then changed to match those sampled from the database. Then a Bit Error Rate (BER) field was calculated and added to each link, based on the new Es**.**N0. After that, the corresponding modulation and coding scheme was used, according to the timestep of the episode. Calculating the BER was necessary as changing the Es/No invalidates much of the link data previously calculated within the SCENIC simulation, such as the receive power and maximum possible data rate.

C. Building and Training the Agent

The policy of the agent was approximated using a neural network. The input and output of the network must be defined before any other decisions are to be made regarding its structure. A vector of dimension $n * x$, *as shown in Fig. 4*, is provided to the network as input, where n is the number of links to analyze and x is the number of features used to describe each link. The network maps these inputs to an output vector of dimension n, whose elements are the likelihood of switching to the link described at index i of the input as seen in Fig. 4, where $x = 2$. When the number of available links is less than n, the input vector is zero padded. This makes it possible for the agent

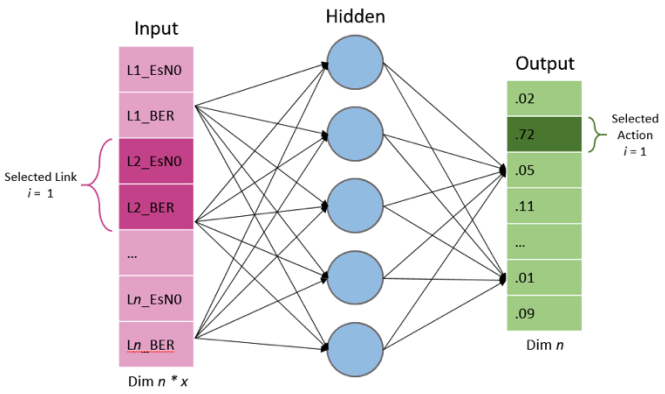

Fig. 4. Policy neural network

to choose a non-real link (one described by zeros in the input vector. Hence the inclusion of three different numerical rewards in the Gym environment: 0.1 for non-real link selection, 1 for real link selection, and 60 for every successfully downlinked packet. The reward scheme incentivizes the selection of real, high quality links, since more packets will be downlinked when using high data rate links.

The agent was trained using a modified version of the policy gradient method, first introduced in [4]. Recall that the goal of a reinforcement learning agent is to maximize its expected reward when following a policy π. This policy is parametrized with a neural network, whose weights and biases are defined by the set of parameters θ to produce the policy π_θ. Now the goal of maximizing expected rewards can be expressed mathematically as the objective function shown in Equation (1).

$$J(\theta) = E_{\pi_\theta}[r(\tau)] \tag{1}$$

where $r(\tau)$ represents the total reward over an entire episode trajectory τ. The problem is then to find the optimal parameters θ which maximize J. The expected rewards are used to encourage finding optimal parameters, using gradient ascent, in the direction of the gradient of J. The derivation of the gradient of J is expressed by Equation 2.

$$\nabla E_{\pi_\theta}[r(\tau)] = E_{\pi_\theta}\left\{\sum_{t=1}^{T} G_t[\nabla log \pi_\theta(a_t|s_t)]\right\} \tag{2}$$

where G_t is the discounted reward at time *t* and $log\pi_\theta(a_t|s_t)$ is the gradient of the log of the policy π_θ at time *t*. In this study, the output of the policy function (approximated by a neural network) is an element from a vector of probabilities of switching to each available link. The chosen link's probability is the policy output used in calculating the gradient. Therefore, the gradient of the expected reward depends only on the action of the agent and that action's corresponding reward. Intuitively, this makes sense; if the agent earns a large reward, the corresponding action should be taken again in the future, and vice versa. The parameter G_t is recorded from real sample episode trajectories and used to update that policy, hence why this method is often called Monte Carlo Policy Gradient. By using enough samples, the policy can be learned without directly learning the action value function itself.

In the standard reinforce algorithm, the gradient is calculated using the actions and discounted rewards from an entire episode trajectory. However, in the case described in this manuscript, discounting the rewards was not necessary, due to the agent's inability to impact its future state. A decision made in the present will not change which links are available to the agent in the future, nor is there an immediately obvious "terminal state" from which to discount. Simulation data was made episodic merely for ease of use and data variation. As such, there is no requirement to use full episode trajectories for each policy update. Any trajectory of actions and rewards can be evaluated in isolation and used to update the policy parameters. The size of this set is defined as the batch size used in training.

The agent's policy during training must effectively balance exploration with exploitation. By default, the policy defined by the network in Fig. 4 is purely exploitative - whichever action has the highest likelihood in the output vector is selected. A purely exploitative policy is prone to find itself stuck in local maxima, always taking the actions found to be better initially and missing out on better choices not yet explored. To ensure that the agent explored the full action space, an ε-greedy policy is used during training where $\varepsilon \in [0.1]$. The ε factor determines the fraction of the time the agent will take a random action instead of following the policy output. An ε of 0.1 was used, meaning that a random action will be taken 10% of the time. When evaluating agent performance ε is set to 0 so that the agent is completely in charge of its actions. In the real world, it is not beneficial for the agent to be taking any random actions; only during training is the ε-greedy policy necessary.

D. Feature Choice and Transformation

A radio link can be described by several different characteristics. Determining which of these are most critical to the agent's decision making process, and thus should be part of the input state, is of critical importance in keeping the neural network as small as possible so that it can run onboard an orbiting satellite. After speaking with domain experts, it was clear that Es/No and the BER were two key factors in determining link quality. It was decided to start with these two and add additional features if necessary. Initial training runs showed that the network was able to converge using just these two features, and they were selected for use in all future training. Other features not used include the maximum bandwidth of the link, the receive power, and the range rate between the transmitter and receiver.

A minimal amount of feature transformation was required to ensure the agent learned successfully. First, there was a need to shuffle the links within the input vector. This is because the input vector is filled by the environment starting from index 0 until all links available at that timestep are included. The remaining indices are zero-padded. The vast majority of the time the agent will see only the first few indices of the input vector populated by real links and will not learn the correct input/output mapping. To alleviate this, a simple random shuffle was applied to the input vector before being fed to the network during training such that each index was equally likely to contain a link.

In addition to shuffling the input vector, the BER feature was transformed by the $\log_{10}$ function before being fed to the network because its values span over two dozen orders of magnitude. The exponent contains the majority of the information necessary to distinguish a good link from a bad one.

E. Analyzing Agent Performance

The main metric used to analyze agent performance was the reward score, a per episode metric that compares the reward achieved by the agent to the maximum possible reward if optimal decisions were made at every step throughout the episode. It is represented as a percentage and is a succinct way of describing agent performance on a single episode. The reward score was averaged over validation and testing episodes to get a sense of overall performance.

However, the reward score does not tell the full story. Additional metrics were necessary to determine how and why the agent was making its decisions. For this we framed the link switching problem as a binary classification: switch to a new link or stay with the current one. This is a natural way of framing the problem which allowed us to take advantage of useful binary classification metrics. We used this framework to build a confusion matrix, seen in Fig. 5, to describe agent performance and calculate various metrics. The criteria for a correct action is simple: in the case of a switch, it is correct if the data rate improved and incorrect if the data rate worsened. In the case of a stay, it is correct if the data rate of the current link is the best available, and incorrect if a better link exists.

As the agent navigated through a set of episodes, the number of true positives, true negatives, false positives, and false negatives were tracked to produce a confusion matrix. From this we calculate three important metrics: recall (also known as sensitivity), precision, and the F_β score.

Together, these metrics provided insight into not only how well the agent was performing on the aggregate but specifically, what its strengths and weaknesses were. An agent which switches at every time step will have a recall of 1, but a very poor precision close to 0. This information was used to gain a more holistic view of agent performance and identify future areas of work in agent development.

F. Hyperparameter Tuning

Neural networks are defined by various hyperparameters which define their structure and how they learn. It is crucial to tune these hyperparameters to achieve optimal performance from the network. The network used in this study was a fully connected feed forward network with six hyperparameters, described in Table III.

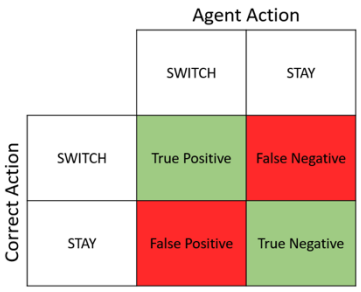

Fig. 5. Switch/Stay confusion matrix

TABLE III. NETWORK HYPERPARAMETER DESCRIPTION

Hyperparameter	***Description***
Shape	Array defining number of hidden layers and their shapes. Ex: [64, 128, 64]
Activation	Activation function to use throughout the network. Ex: RELU
Dropout Rate	Dropout rate to use on each layer of the network. Ex: 0.5
Bias	Whether to include bias in the network or not. Ex: True
Optimizer	Gradient descent optimizer to use during training/associated learning rate. Ex: ADAM, 0.1
Batch	Size Number of timesteps to include in a single update batch. Ex: 8

The activation function and optimizer were frozen early on to allow further tuning of other hyperparameters. The activation function used was the rectified linear unit (RELU). The optimizer used was Adadelta [7], an adaptive learning rate gradient descent optimizer which requires no manual tuning of the learning rate. Optimizers such as Adadelta vary the learning rate during training to reduce the chances of a model getting stuck in local minima.

The inclusion of bias in the network architecture was detrimental to network performance in early trials and was subsequently excluded from the network in all subsequent training. The reason for this has not yet been determined.

The three remaining hyperparameters of shape, dropout rate, and batch size were tuned using a standard hyperparameter searching algorithm known as Grid Search. Shape refers to the hidden layer architecture of the network and is represented as an array of integers which are the size of each layer. For example, a shape of [64] corresponds to a single hidden layer with 64 neurons. The dropout rate is the fraction of neurons that are ignored on every forward pass through the network during training. Dropout helps to reduce dependence on single neurons and ensure the network generalizes properly.

Possible values for each hyperparameter are identified, and a unique model is trained for every possible combination of these values. The search space can quickly diverge, and thus potential values must be chosen carefully and kept to a minimum. The searched values are described in Table IV. This grid results in thirty six unique models.

TABLE IV. GRID SEARCH VALUES

Shape	***Dropout Rate***	***Batch Size***
[64]	0.00	2
[128]	0.25	4
[64 128 64]	0.50	8
	0.75	

Each model was cross-validated using K-fold cross validation with K= 4. The K-fold cross validation procedure provides more robust insight into the performance of a specific model, crucial when comparing many models in a Grid Search. All models were trained and cross-validated using the same set of 450 episodes augmented with the previously described, fading profiles. Models were compared against one another using a sum of their average reward score on the validation episodes and their recall score. Recall was prioritized over precision in this study because the goal of the study was to validate an agent's ability to correctly switch when it is necessary. Once this ability is proven, techniques can be used to mitigate the frequency of extraneous switches made by the agent.

G. Final Model Training and Analysis

The hyperparameters which produced the best model during the Grid Search were used to train a final agent on the same 450 episodes used in the Grid Search with a 75/25 train test split. The agent was thus trained on 337 episodes and validated on 113 episodes.

III. Results

The Grid Search was performed and the ideal hyperparameters were found. A final agent was trained using these hyperparameters and tested on 113 validation episodes where it achieved an average reward score of 98.12%. A model topology of a single hidden layer of 128 neurons, dropout rate of 0, and batch size of 8 was the best performer, with a combined score of 173.804. An agent was trained using this model on 75% of the 450 episodes (337 episodes) used during the Grid Search. Fig. 6 shows the reward score achieved by the agent on each training episode throughout training and Fig 7 shows a histogram of the validation reward scores.

Since the agent is following an ε-greedy policy during training with ε = 0.1, the scores achieved during training are a skewed representation of its performance. Fig. 8 shows the scores achieved by the agent on the validation set. During validation the agent is completely in charge of all decisions rather than 10% of decisions being random, and thus achieves better performance.

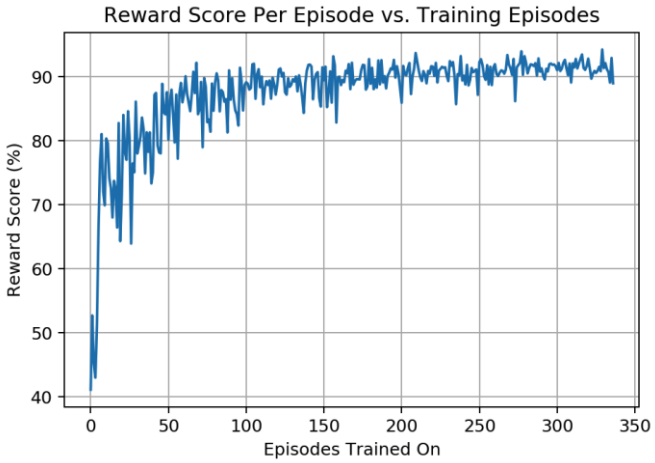

Fig 6. Reward scores during training

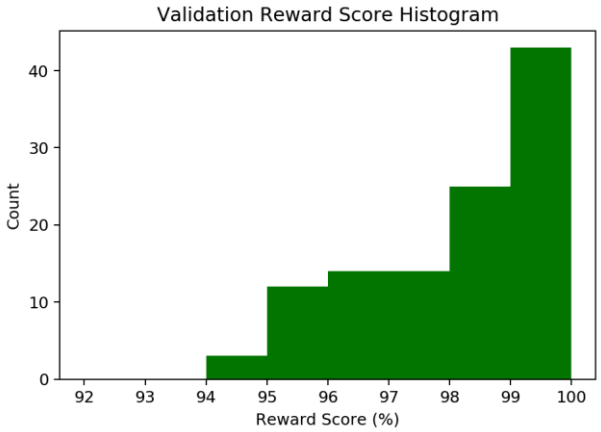

Fig. 7 Validation Reward Scores.

The lowest score on any episode in the validation set was 94.03%, and the highest was 100%. The average score was 98.12%. The confusion matrix shown in Fig. 8 was built over the entire validation set. These values give a recall of 74.85%, a precision of 49.91%, and F_β score with $\beta = 3$, of 0.713.

A recall of 74.85% is a positive result and shows that the agent switches the majority of the time that a switch is necessary. However, the precision is very poor at under 50%. At first glance it appears the agent is simply switching so often it manages to have good recall at the expense of its precision. However, further analysis reveals a more complex problem. Due to the method used to augment the episodes, certain timesteps in the episodes had links which had the exact same BER. Among the 78,787 classified False Positives, 1,812 of these came from a decision by the agent to switch from one link to another with the exact same BER. Intuitively we understand that these are not truly false positives but rather an issue with our definition of false positive in this experiment.

In addition, there are often times when the agent switches from one link to another of highly similar quality. This reveals a weakness in the application of binary classification metrics to the agent. When links are of similar quality, choosing the slightly worse one has minimal effect on the total reward the agent will achieve. Thus, the agent does not learn to distinguish them from one another. However, this choice is penalized blindly as a false positive in the confusion matrix, leading to poor precision.

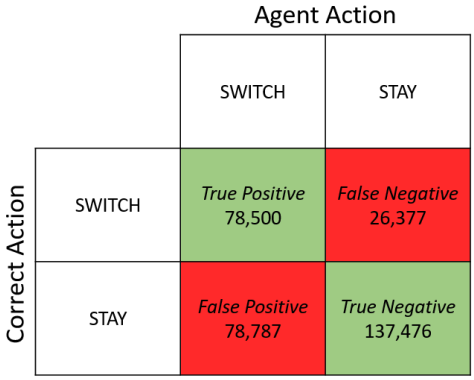

Fig. 8 Confusion matrix and associated metrics.

Overall, the agent performs well and demonstrates a clear ability to distinguish good links from bad, and effectively switch between them over time, to optimize data throughput. It is able to accomplish the core operating principle of a cognitive link: analyze the surrounding environment and respond intelligently to optimize data throughput.

IV. Discussions

This study showed that reinforcement learning presents a viable means of implementing a cognitive link. A high fidelity simulation environment was created which closely resembles the real operating environment, including the effects of link fading. Inside of this environment, a reinforcement learning agent approximated by a neural network was trained using a modified reinforce algorithm method. The agent architecture and hyperparameters were tuned using a grid search. The final model was trained on 337 training episodes and validated on 113 validation episodes. Illustrative metrics were calculated based on agent performance, and strengths and weaknesses were identified. Overall, agent performance was positive and should encourage the continued application of reinforcement learning to cognitive communications.

The environment created for this study can be used for testing other reinforcement learning agents. It can also be updated to use a different underlying simulation tool, without requiring a rewrite of agent training code, thanks to the decoupled nature of the simulation itself and agent training environment. There will always be drawbacks to simulating the training environment for an agent one hopes to deploy in the real world. Antenna slewing and competition for ground station resources were both excluded from the environment for simplicity in this initial research. These are important considerations for any cognitive link algorithm. The agent was not tested in an environment which incorporates these and thus its performance in such an environment cannot be estimated. In addition, downlink of data was simulated numerically and not using actual channels capable of introducing additional noise and interference. This meant that the link always performed exactly as its numerical characteristics predicted it would; in the real world, performance may be similar but not exact, and in some cases vastly different than the most current numerical characteristics would indicate.

Despite these drawbacks, the environment presented sufficient complexity to the agent to ensure that results are valid indicators of the viability of reinforcement learning as a solution to the problem of cognitive links. The reinforcement learning techniques used in this study are simple, especially when compared with newer techniques such as Deep Q-Learning, Asynchronous Actor-Critic Agents, and Inverse Reinforcement Learning. The agent's ability to learn in the simulated environment developed for this study validates that the core methods used in reinforcement learning are applicable to the problem. As the environment the agent is expected to operate in, rises in complexity, more complex reinforcement learning methods will likely be required. This study shows that these methods are likely to present a viable solution.

V. Conclusions

Reliable and robust communication is key for the future of space travel and exploration. The number of space assets will undoubtedly increase in the future. This forces automation to become a requirement for the successful operation and deployment of communication systems, capable of dealing with such demands. Currently, ground databases and standard mathematical models that describe such systems are not adequate to successfully automate all the various tasks a space communications system must perform.

This study presented an alternative approach to standard methods for the task of link switching of a space orbiting asset between ground stations via a reinforcement learning approach. This approach was selected because it allows for scalability and generalization to other decisions within a well-defined infrastructure, such as that of a communications system.

In this study, a reinforcement learning algorithm was built, which optimizes the data throughput of a single space asset. A training environment was created using available simulation tools developed by the SCENIC lab at Glenn Research Center, that model the closest possible representation of the real operating environment. Then, the reinforcement learning algorithm was used to train an agent inside this environment to maximize data throughput.

Results from the conducted experiments showed promising characteristics of this approach in terms of correct decision making, even in the presence of additional complexity, such as strong multipath fading in the communication link. Overall, the testing showed that the agent performs well inside the training environment and can be used as a foundation for future studies with added complexity and eventually it can be tested in the real space environment.

Acknowledgment

Funding for this research was provided by the Space Communications and Navigation (SCaN) program.

References

[1] R. Roche, B. J. Briones, B. N. Kowalewski, "Metabrain for Embedded Cognitions (MBEC)", NASA/TM-2018-219877, E-19519, GRC-E-DAA-TN54156, Oct 2018

[2] Kurt, H., "Approximation Capabilities of Multilayer Feedforward Networks," Neural Networks, Vol. 4, No. 2, 1991, pp. 251–257.

[3] Minsky, M., "Steps Torward Artificial Intelligence," Proceedings of the IRE, Vol. 49, No. 1, 1961, pp. 8–30.

[4] Williams, R. J., "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning," Machine learning, Vol. 8, 1992, pp. 229–256.

[5] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M., "Playing Atari with Deep Reinforcement Learning,", Dec. 2013. ArXiv: 1312.5602 [cs.LG].

[6] of Concerned Scientists, U., "UCS Satellite Database,", 2018. Retrieved from UCS Satellite database 2018, https://www.ucsusa.org/nuclear-weapons/space-weapons/satellite-database#.XAAKdpxKiUk.

[7] Zeiler, M. D., "Adadelta: An Adaptive Learning Rate Method," , Dec. 2012. ArXiv: 1212.5701 [cs.LG].