Project 2

Deep Reinforcement Learning

Juan Tarrat

1 Introduction

For Project 2 I decided to re-do the simulations in the paper ”Deep Multi-User Reinforcement Learning for Dynamic Spectrum Access in Multichannel Wireless Networks” by Oshri Naparstek and Kobi Cohen. The authors in this paper used deep reinforcement learning techniques for network utility maximization in multichannel wireless networks. More precisely, they used Deep Q-Learning algorithm to maximize the network throughput along a number of time slots, achieving better performance than slotted-ALOHA protocol widely used in present networks.

The type of networks the authors work with consist various channels and users. The channels are assumed to be orthogonal. The rules of the game are that users cannot transmit over the same channel at a given time slot. The objective is to maximize the throughput of the network using deep reinforcement learning techniques so that the users can learn what action to take without the need of extra communication between themselves.

2 The Structure of the Program

This section discusses the code implementation of the reinforcement learning algorithm.

2.1 Environment

The first step when developing a reinforcement learning program is an environment. I followed the auhtors' implementation of a class that takes number of users, number of channels, and an attempt probability to initialize an environment. The class implements two methods: *sample* and *step*

The environment is set up as follows. The users can take actions, actions are represented using python lists. An environment with N users will have action list of length N, where each element corresponds to a user an the value of each element is the channel that user will attempt to transmit in the next time slot. Actions can be randomly sampled using the *sample* method.

Once there is an action, the *step* function will take it as input and try to execute it to then return an observation of the environment. An observation consists of N tuples that hold ACK and reward information; last item is always a list of length K channels with 1 or 0 displaying whether the network is available or not.

In order to get a reward, a user must be the only one attempting a transmission on a given channel. For example, for an environment with 4 users and 3 channels, action `[2 1 0 0]` will result in an observation `[(1, 1.0), (1, 1.0), (0, 0.0), (0, 0.0), array([0, 0, 1])]`.

2.2 Memory

A key part of the Deep Q-Learning algorithm is the use of a replay memory that is used in training to sample batches of experiences that are key for training. I used a class that seems very standard in Deep Q-Learning that uses the deque class from the collections library in python. The Memory class has two methods: The *add* method adds an experience consisting of a state, reward, action and next state tuple to memory; the *sample* method takes a batch size as input and returns that batch.

2.3 DQ Network

The deep learning network is implemented using the Keras library, and is implemented as a class that takes as inputs a learning rate, a size of the state array, an action size, a list of hidden layer sizes, a step size, and a name for the network. The class is implemented as a normal fully connected Keras model. Following the method the authors used, I also included a LSTM layer with 10 units. The class also includes two methods: *train* and *predict* to respectively train the network for a set number of epochs and to predict the values desired.

The network's job is to predict the action a certain user should take in order to maximize the reward. The input of the network is a set of states. That is what the step size parameter is, a small collection of the last s seen states. The output is an array of Q-values if the user decides to not transmit, or transmit in any of the channels.

2.4 Training

Training is a very heavy computational operation in a network like this one and involves a set of steps. Firstly, and as always, following the authors' steps, we need to populate the memory and the input history.

```
def populateMemory(environment, memory, step_size, pretrain_length=128):
historyInput = deque(maxlen=step_size)
action = environment.sample()
obs = environment.step(action)
state = state_generator(action,obs, environment.numChannels)
reward = getRewardFromObservation(obs, environment.numUsers)

for ii in range(pretrain_length*step_size*5):
    action = environment.sample()
    obs = environment.step(action)
    next_state = state_generator(action,obs, environment.numChannels)
    reward = getRewardFromObservation(obs, environment.numUsers)
    memory.add((state,action,reward,next_state))
    state = next_state
    historyInput.append(state)

return historyInput
```

Once that is done, the training loop starts. For every episode, the environment samples an action and gets an observation to generate an initial state. After this is done, the network will decide whether to explore or exploit. When exploring, we simply sample a random action from the environment again and generates a next state. If exploitation is the decision (based on an exploration probability that decays as the episode progress) the network generates an action vector from predicted values of the network. The input for this step comes from the history input collection. Once the next state is gathered we need to gather the rewards and implement a cooperative policy. This policy gives the sum of rewards in the environment to every user that has transmitted.

```
reward = getRewardFromObservation(obs, environment.numUsers)
sum_r =  np.sum(reward)

totalRewards.append(sum_r/environment.numChannels)

for i in range(len(reward)):
    if reward[i] > 0:
        reward[i] = sum_r
```

After this is done, the experience is stored in memory and a batch is sampled for training. Once the batch is sampled and a target Q value is calculated based on the rewards and the network prediction, the batch is fed into the fully connected layers for training. After this is done, the exploration probability is updated.

```
targetQ = qNetwork.predict(nextStates)
targets = rewards[:,-1] + gamma * np.max(targetQ, axis=1)
qNetwork.train(states, epochs, targets, batch_size)


epsilon = max(explore_stop, epsilon * decay_rate)
```

The issue I encountered when training is that a very powerful unit is needed in order to achieve meaningful results. Due to hardware limitations, I could only train networks up to 5000 episodes, which represents a decrease of 20 times compared to the 100,000 episodes the authors ran in the paper.

3 Simulations

A total of 6 simulations have been performed. All the simulations shared the same hyperparameters, what varied was the architecture of the fully connected network and the number of users and channels. The networks I simulated are the following:

- 2 users and 3 channels [64 64]
- 3 users and 2 channels [10]
- 3 users and 2 channels [64 128 64]
- 4 users and 3 channels [10]
- 4 users and 3 channels [64 64]
- 5 users and 4 channels [128 64]

The hyperparameters used are the following:

```
memory_size = 1000
batch_size = 128
pretrain_length = batch_size
hidden_size = 128
learning_rate = 0.0001
explore_start = .02
```

```
explore_stop = 0.01

decay_rate = 0.0001

gamma = 0.9

step_size=1

alpha=0

epsilon = explore_start

beta = 1
```

For the plots, I recorded the network throughput, the average throughput every 500 episodes, the cumulative reward, and a histogram of the values grouped by number of channels. It can be seen that the networks do not perform as in the authors' implementation. The main issue is that the number of episodes is so limited that there is no time for the network to learn the optimal weights and biases to maximize the reward. We do observe a increasing trend in the cumulative reward, meaning that in some episodes there is a good action being chosen. However, we do not obsere any trend in the throughput average; in fact, looking at the histograms, the most repeating value of the reward is 0. In future work, different reinforcement learning algorithms can be implemented like actor critic, in which a policy is learnt by the network rather than the Q-values.

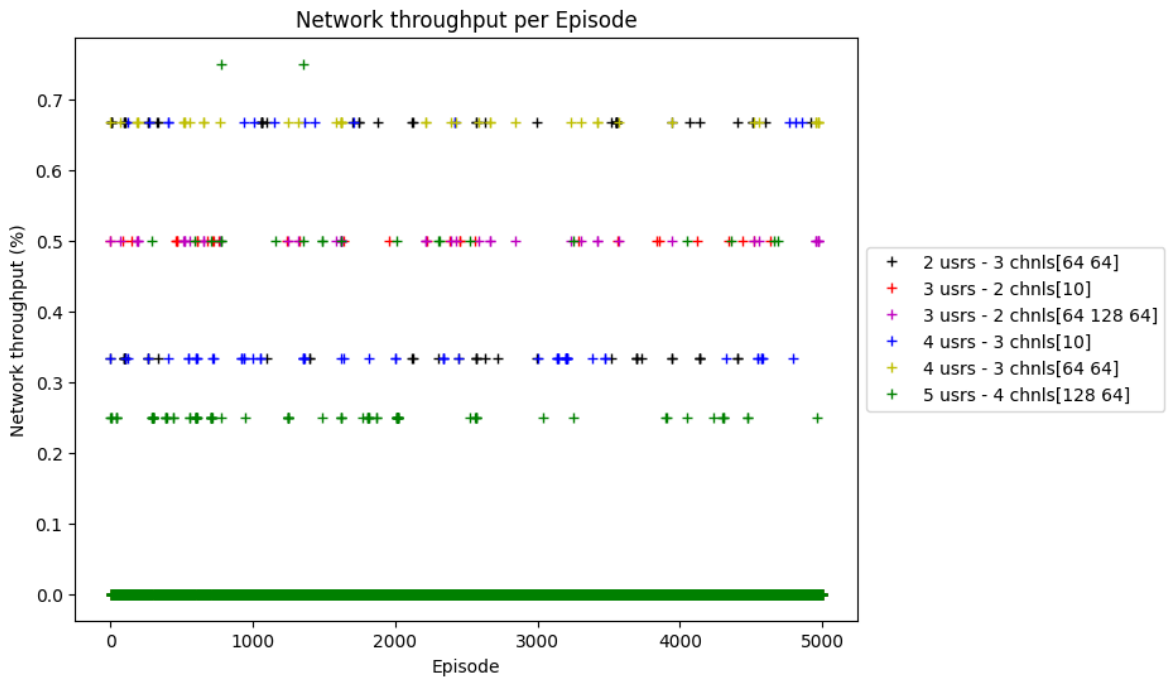

Figure 1: Throughput of the networks

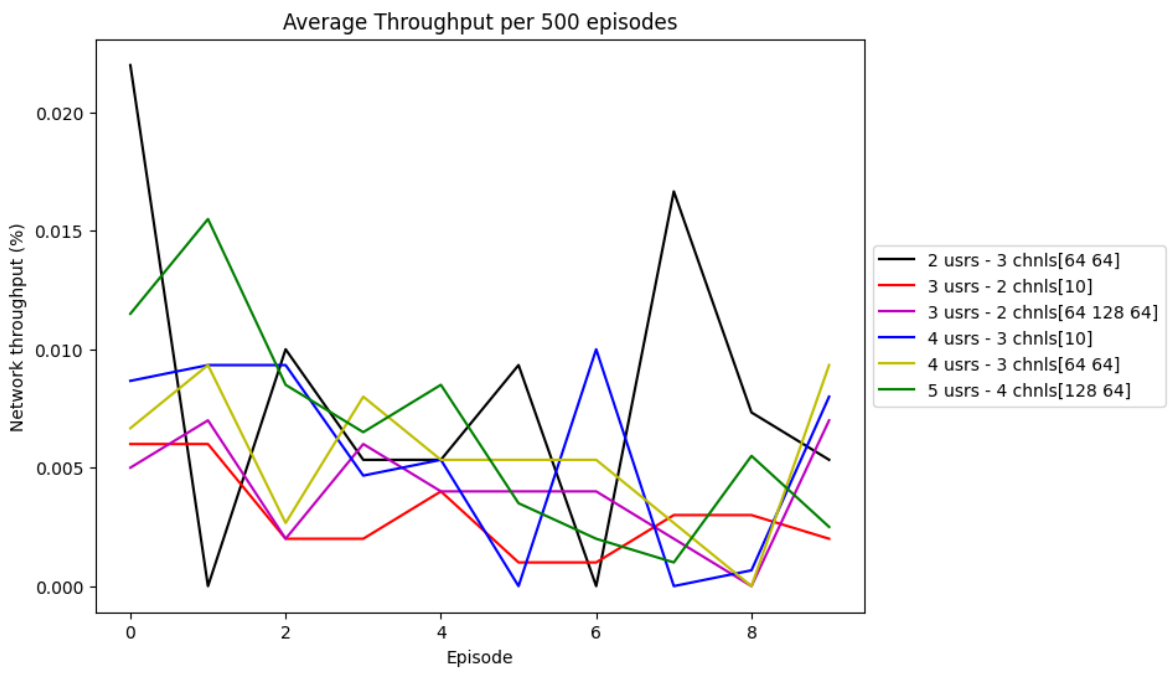

Figure 2: Throughput average of the networks

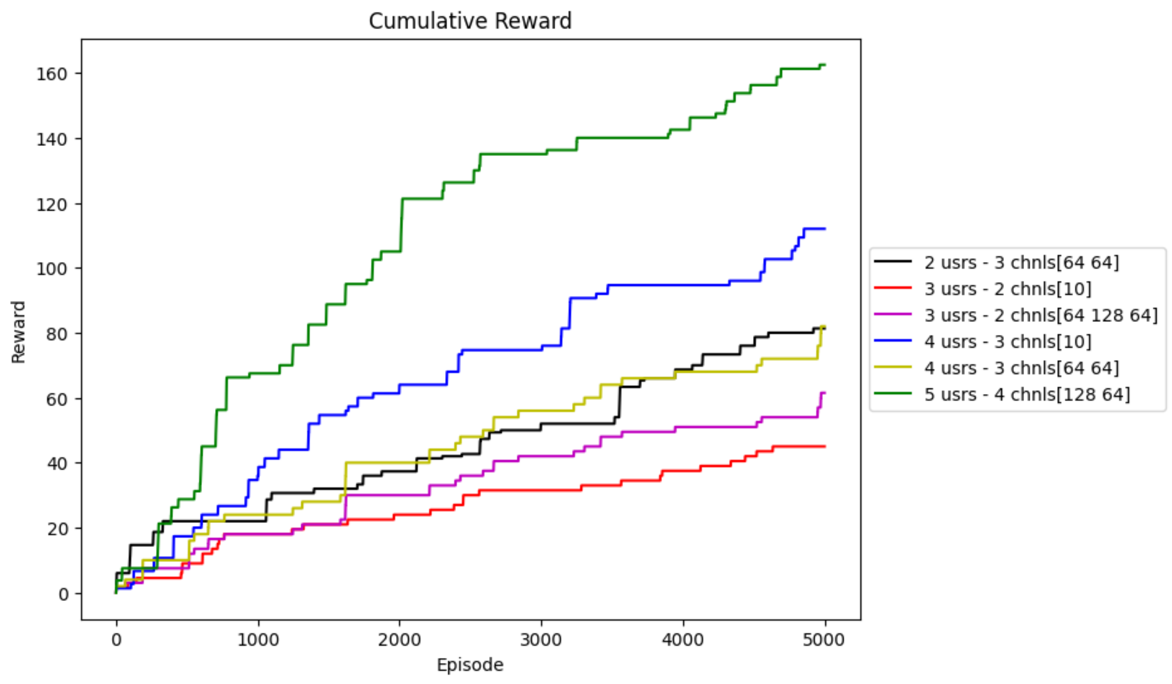

Figure 3: Cumulative Reward

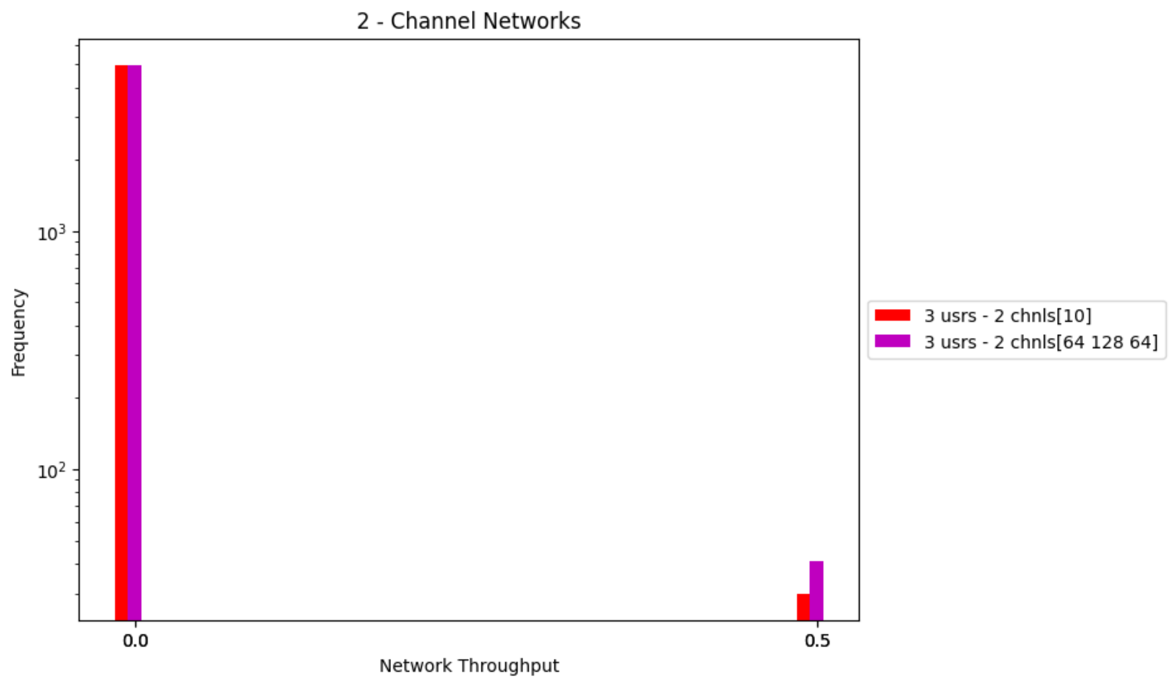

Figure 4: Histogram of 2 channels

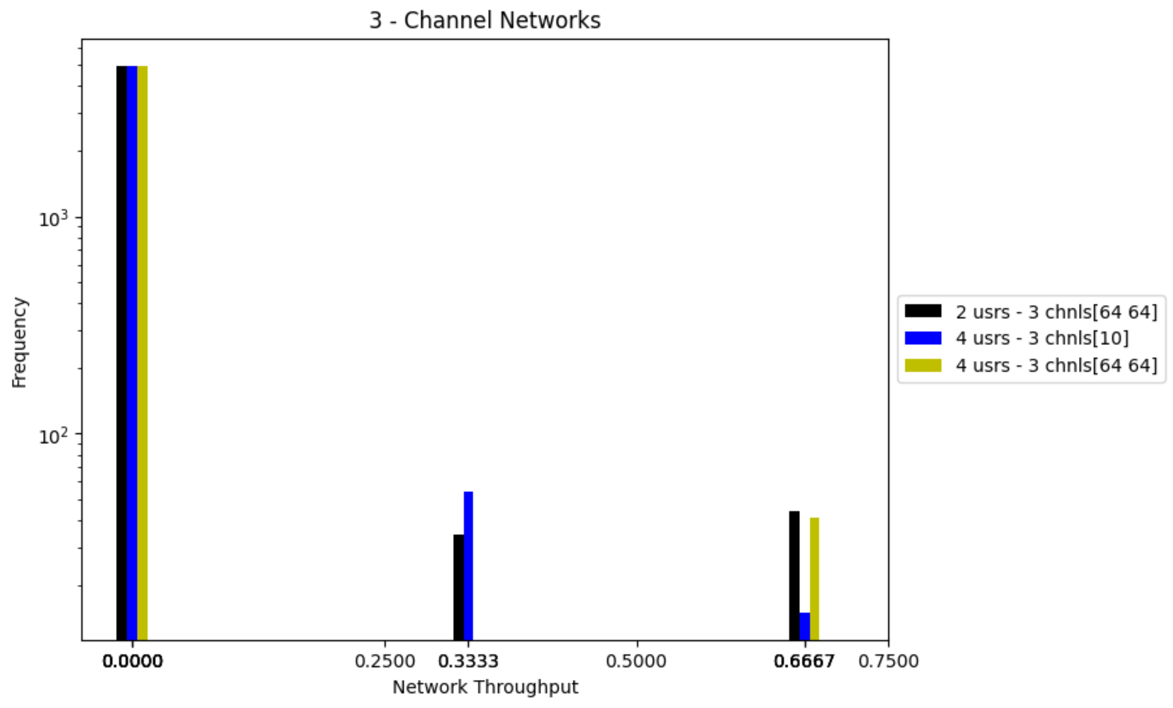

Figure 5: Histogram of 3 channels

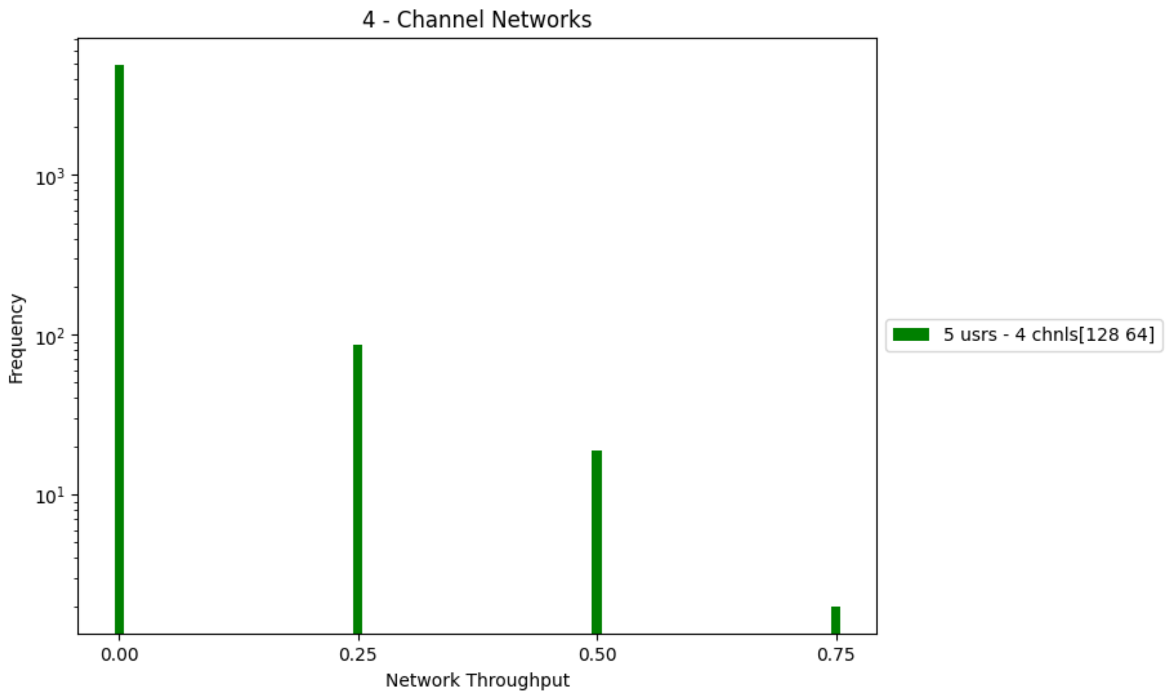

Figure 6: Histogram of 4 channels