Deep Reinforcement Learning

Juan Tarrat

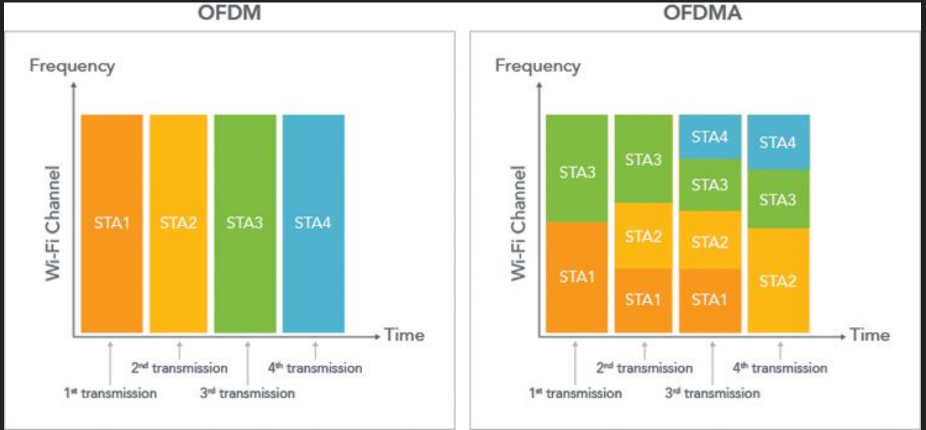
Objective

Reimplement simulations done in paper Deep Multi-User Reinforcement Learning for Dynamic Spectrum Access in Multichannel Wireless Networks

We have a network with K channels that must be shared by N number of users

We want to maximize the network utilization without the users coordinating

In order to do this Deep Reinforcement Learning approach is used (DQ Learning)



Deep Q-Learning

It is a type of Q-Learning that uses deep neural networks to learn q-values

In this case different users need to learn what action to take based on the state of the network in order to maximize their reward (network throughput)

Used Keras for implementation

Environment

Based on the action each user takes, the environment will reward or not

If more than one user attempts a transmission in the same channel, the reward will be 0

```
[0 2 0]
[(0, 0.0), (1, 1.0), (0, 0.0), array([1, 0])]
1.0
***************
[0 0 2]
[(0, 0.0), (0, 0.0), (1, 1.0), array([1, 0])]
1.0
***************
[1 0 1]
[(0, 0.0), (0, 0.0), (0, 0.0), array([1, 1])]
0.0
***************
[0 0 2]
[(0, 0.0), (0, 0.0), (1, 1.0), array([1, 0])]
1.0
***************
[1 1 1]
[(0, 0.0), (0, 0.0), (0, 0.0), array([1, 1])]
0.0
***************
```

Deep QN

DQN class takes, among other parameters, a list o hidden layer neurons.

Extra methods to train and predict values

Output size is dependent on possible actions (wha channel to use)

```
class DQN(tf.Module):
    def __init__(self, learning_rate=0.01, state_size=4,
                 action_size=2, hidden_layer_sizes=[10], step_size=1,
                 name='DQN'):

        super(DQN, self).__init__(name=name)

        self.model = Sequential()

        self.model.add(LSTM(10, input_shape=(step_size, state_size)))

        # Build hidden layers
        for i, hidden_size in enumerate(hidden_layer_sizes):
            self.model.add(Dense(hidden_size, activation='relu', name=f'h{i + 1}'))
            self.model.add(LayerNormalization())

        self.model.add(Dense(action_size, activation='linear', name='output'))

        # Compile the model
        self.model.compile(optimizer=Adam(learning_rate),
                           loss='mean_squared_error')

    def train(self, states, epochs, targets, batchSize):
        # Train the model
        self.model.fit(states, targets, epochs=epochs, batch_size=batchSize, verbose=0)

    def predict(self, states):
        # Predict Q-values for a batch of states
        return self.model.predict(states, verbose=0)
```

Memory

```
from collections import deque
import numpy as np

class Memory():
    def __init__(self, max_size=1000):
        self.buffer = deque(maxlen=max_size)

    def add(self, experience):
        self.buffer.append(experience)

    def sample(self, batch_size, step_size):
        idx = np.random.choice(np.arange(len(self.buffer)-step_size),
                               size=batch_size, replace=False)

        res = []

        for i in idx:
            temp_buffer = []
            for j in range(step_size):
                temp_buffer.append(self.buffer[i+j])
            res.append(temp_buffer)
        return res
```

Pre-Training

Replay Memory and history Input are populated in pre-training.

History input used for exploitation

Replay memory used for sampling batches in training

```
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
```

Training (I)

```python
totalRewards = list()
for episode in range(episodes):

    if episode %100 == 0:
        if episode < 1000:
            beta -=0.001

    if episode % 1000 == 0:
        print("Episode " + str(episode))

    # print("Episode " + str(episode))
    # decide whether to explore or not
    action = environment.sample()
    obs = environment.step(action)
    state = state_generator(action, obs, environment.numChannels)
```

```python
#exploration
if np.random.random() < epsilon:
    action = environment.sample()
else:
    #exploitation
    action = np.zeros([environment.numUsers], dtype=np.int32)

    stateVector = np.array(historyInput)
    # print(stateVector)
    for u in range(environment.numUsers):
        userState = stateVector[:, u, :].reshape(1, step_size, state_size)
        prediction = qNetwork.predict(userState)

        prob1 = (1-alpha)*np.exp(beta*prediction)

        # Normalizing probabilities of each action  with temperature (beta)
        prob = prob1/np.sum(np.exp(beta*prediction)) + alpha/(environment.numChannels+1)


        #  choosing action with max probability
        action[u] = np.argmax(prob,axis=1)


action = action.astype(np.int32)
# print(action)
obs = environment.step(action)
nextState = state_generator(action, obs, environment.numChannels)
```

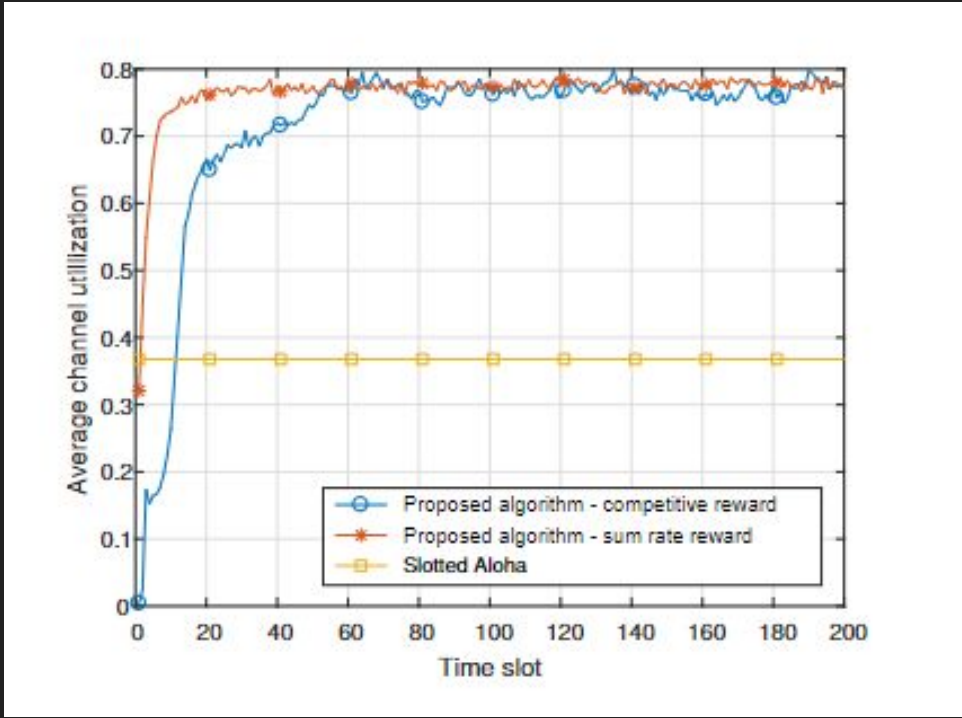
Training (II)

```
nextState = state_generator(action, obs, environment.numChannels)

reward = getRewardFromObservation(obs, environment.numUsers)
sum_r =  np.sum(reward)
# cumulativeReward.append(cumulativeReward[-1] + sum_r)

totalRewards.append(sum_r/environment.numUsers)

# collision = environment.numChannels - sum_r
# cumulativeCollision.append(cumulativeCollision[-1]+collision)

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

# print(reward)
memory.add((state, action, reward, nextState))
state = nextState
historyInput.append(state)

batch = memory.sample(batch_size, step_size)

states = get_states_user(batch, environment.numUsers)
actions = get_actions_user(batch, environment.numUsers)
rewards = get_rewards_user(batch, environment.numUsers)
nextStates = get_next_states_user(batch, environment.numUsers)

states = np.reshape(states,[-1,states.shape[2],states.shape[3]])
actions = np.reshape(actions,[-1,actions.shape[2]])
rewards = np.reshape(rewards,[-1,rewards.shape[2]])
nextStates = np.reshape(nextStates,[-1,nextStates.shape[2],nextStates.shape[3]])
```


```
states = np.reshape(states,[-1,states.shape[2],states.shape[3]])
actions = np.reshape(actions,[-1,actions.shape[2]])
rewards = np.reshape(rewards,[-1,rewards.shape[2]])
nextStates = np.reshape(nextStates,[-1,nextStates.shape[2],nextStates.shape[3]])
```


```
# print(nextStates.shape)
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)
```

Target Simulation



Issues with training

DQ training is a very expensive computational process!

On local machine anything over 5000 not only took around 10 hours but also makes computer run out of memory

Free cloud service like google colab allowed use of GPUs

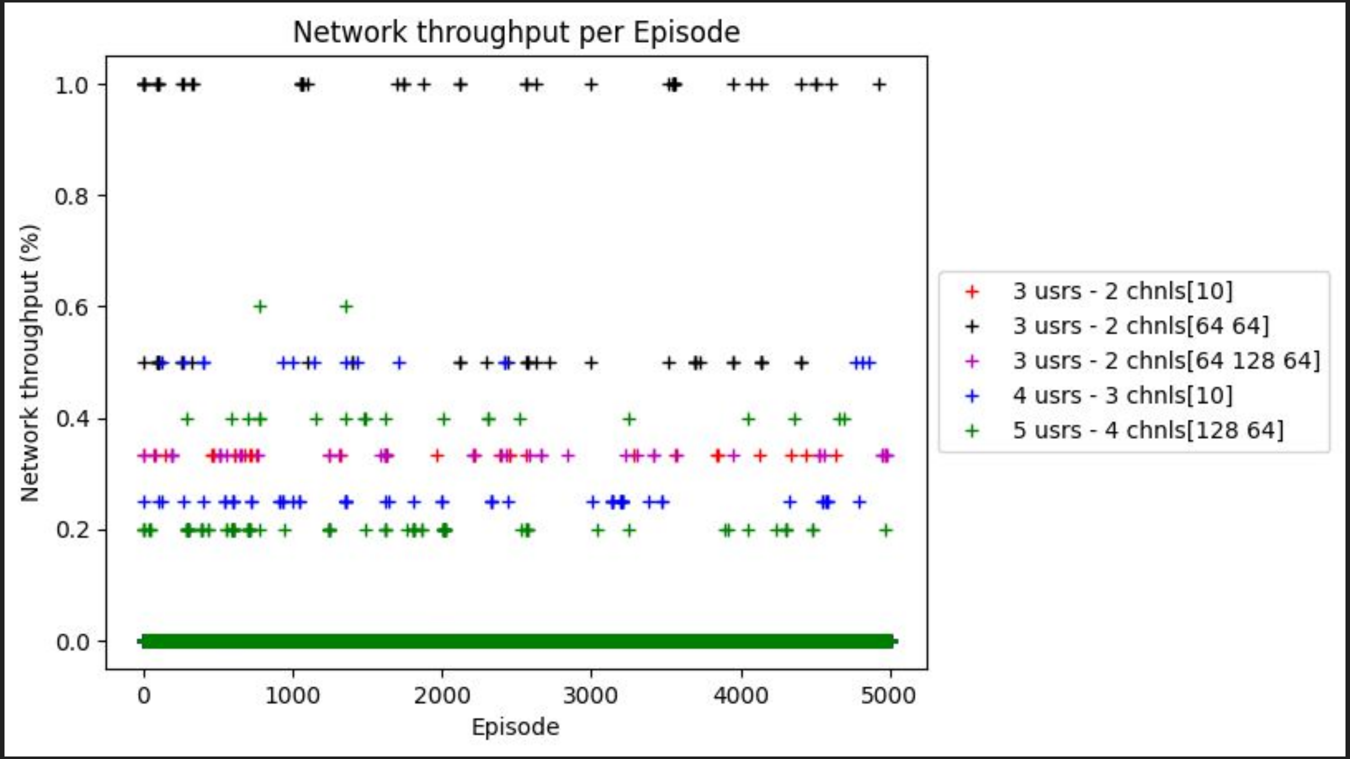
Simulation in paper ran for 100,000 episodes, so 20 times more

Simulations

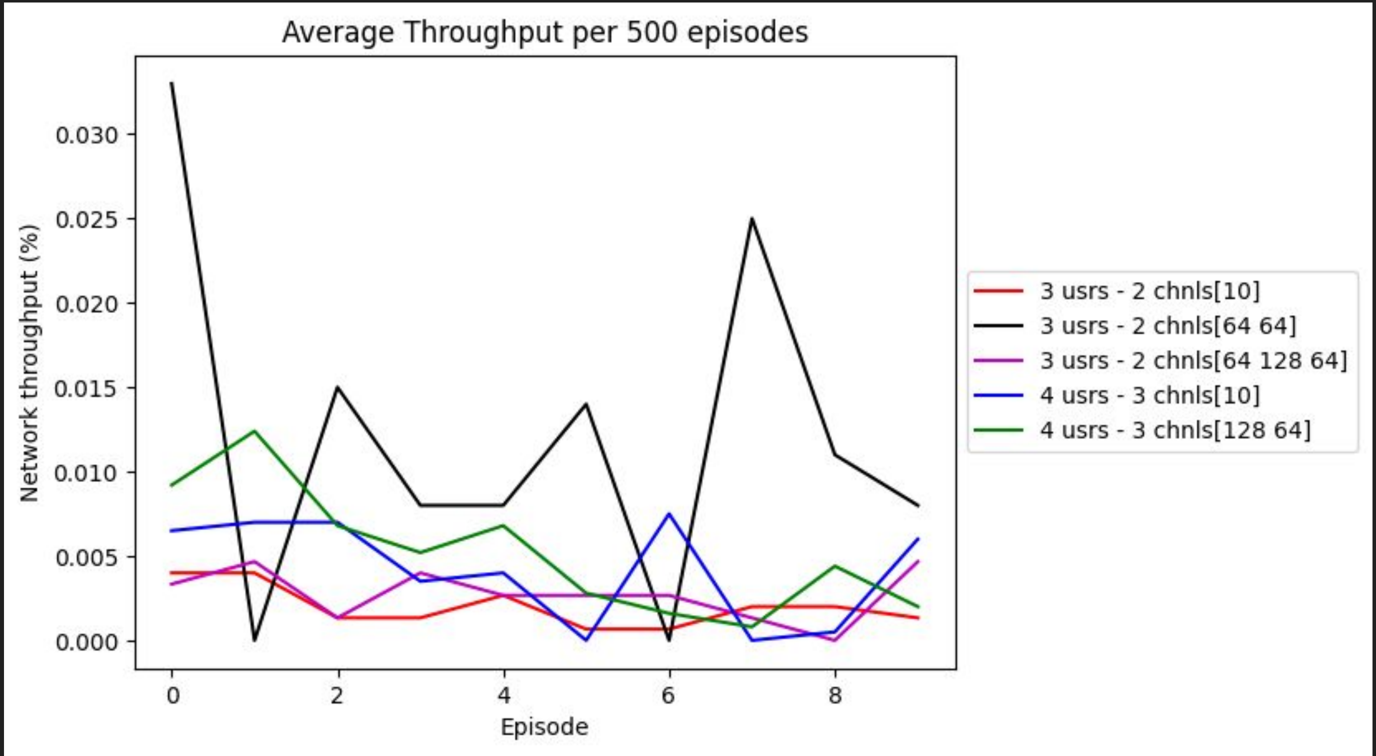
- 2 channels and 3 users
  - One hidden layer [10]
  - Two hidden layers [64 64]
  - Three hidden layers [64 128 64]
- 3 channels and 4 users
  - One hidden layer [10]
  - Two hidden layers [64 64] - running…
- 4 channels and 5 users
  - Two hidden layers [128 64]

```
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
```

Results

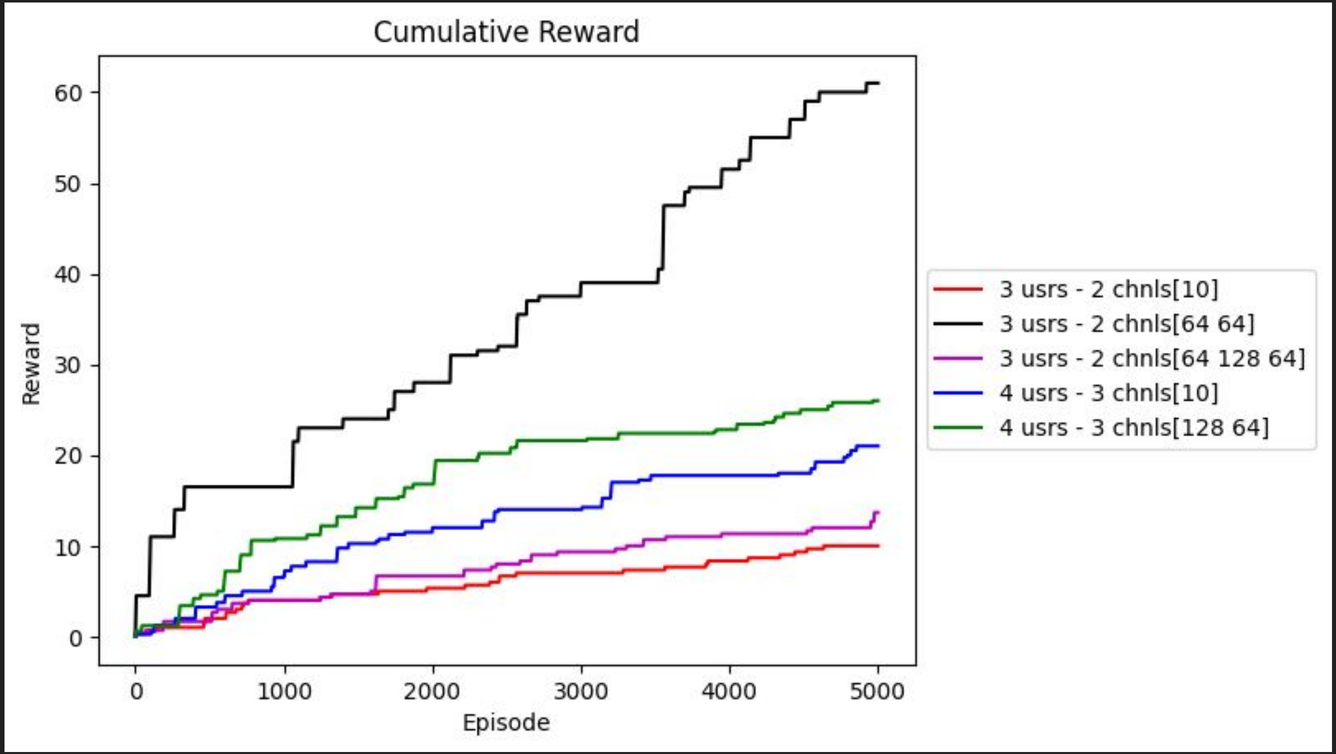

Network throughput per Episode
Network throughput (%)
Episode
1.0
0.8
0.6
0.4
0.2
0.0
0
1000
2000
3000
4000
5000
3 usrs - 2 chnls[10]
3 usrs - 2 chnls[64 64]
3 usrs - 2 chnls[64 128 64]
4 usrs - 3 chnls[10]
5 usrs - 4 chnls[128 64]

Results



Average Throughput per 500 episodes
Network throughput (%)
0.000
0.005
0.010
0.015
0.020
0.025
0.030
Episode
0
2
4
6
8
3 usrs - 2 chnls[10]
3 usrs - 2 chnls[64 64]
3 usrs - 2 chnls[64 128 64]
4 usrs - 3 chnls[10]
4 usrs - 3 chnls[128 64]

Results


Cumulative Reward
Reward
Episode
0
10
20
30
40
50
60
0
1000
2000
3000
4000
5000
3 usrs - 2 chnls[10]
3 usrs - 2 chnls[64 64]
3 usrs - 2 chnls[64 128 64]
4 usrs - 3 chnls[10]
4 usrs - 3 chnls[128 64]

Questions?