Sapienza
Università di Roma

Machine Learning Course

Prof: Paola Velardi

Deep Q-Learning with a multilayer Neural Network

Alfonso Alfaro Rojas - 1759167

Oriola Gjetaj - 1740479

February 2017

Contents

1. Motivation

Automatic game playing brings machine learning techniques into the gaming arena. The goal is to program computers to learn how to get good at playing and even challenge humans.

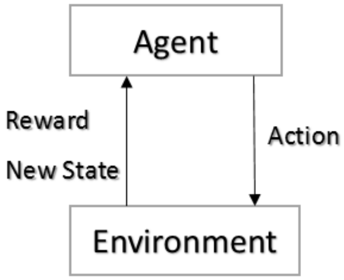
Our biggest motivation to experiment with neural network architectures in games comes from recent advances in the deep reinforcement learning domain. These include DeepMind’s breakthrough in using convolutional neural networks. Current developments in this specific area look promising and being able to understand it, seems challenging and exciting.

2. Q-Learning and Deep Neural Networks

Teaching a computer to play a game without providing training examples. Just by receiving a delayed feedback on performed actions is a task that can be solved using reinforcement learning method.

- *Reinforcement Learning*

Reinforcement Learning does not make use of examples or initial models of the environment. The agent has to learn the mapping of possible paths towards the desired state based only on rewards outputted as part of the target function. Agent learns by performing certain actions in the environment. Each action performed from a specific state results in a reward that can be positive or negative. By performing enough many actions, system learns to pick the path that will maximize future rewards. These rules create the policy by which agent operates in the future.



Each episode of the process in the figure generates a sequence of states, actions and rewards gained after performing a particular action while being on a certain state.

Discount factor (γ) defines the emphasis we put on rewards from future actions compared to immediate rewards. A good strategy is to set a small value in the beginning and then increase gradually; this way we make sure the algorithm will pick an action that maximizes future rewards.

- *Q-Learning*

Q-Learning algorithm, a reinforcement learning technique, works by learning an action-value function which utilizes immediate reward value and future expected rewards for possible states and corresponding actions.

The output of Q-function received in each iteration represents the value of that action for a given state making use of known information.

At first, values for all state-action pairs are initialized to zero and starting from the initial state, successive actions are taken considering Q-function output for each iteration.

For one transition,$< s_0, a_0, r_0, s_1 >$ we can express Q-value of initial state and action in terms of immediate reward and maximum future reward.

$$\mathrm{Q(s_0, a_0) = r + \gamma max_{a_1} Q(s_1, a_1)}$$

The maximum reward for the next state is an approximation used to update Q-value of initial state; it can be wrong in the early stages but it gradually improves with each iteration.

Q-learning allows for the system to iteratively converge and represent accurate Q-values for each state-action combination.

- *Introducing Neural Nets.*

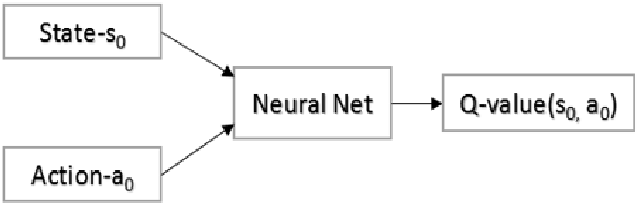
Employing Q-learning into a game model is very specific to a particular game. The problem with Q-tables is their extremely large scale when applied to grid inputs; the number of states is extremely large. To be able to apply Q-learning using screen pixels, we use neural networks; this way the Q-function can be represented by the neural net as an approximation of what the Q-table would have outputted. The benefit of Network in this case will be the generation of all Q-values for an individual state with one forward pass.

Neural networks improve the learning noticeably , nevertheless their architecture is not feasible enough since the number of input nodes is as large as all possible states or alternatively all state-action pairs there are. That is why deep neural networks need to be used.

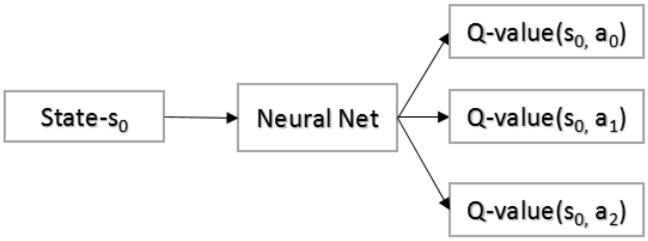
The advantage of deep neural networks is the ability to analyze large amount of pixel data input and map this information into more suitable patterns for later evaluation. This construction technique, known as convolutional neural network, analyzes pixel grids of the input state, condenses the information into fewer neurons and forwards the information to the next layer. Having a number of convolutional layers allows for data simplification and faster input analysis.

The training technique in a game model is by building the target function as a deep neural network in which the input nodes take state-action pairs and output a calculated value, or alternatively input nodes are possible states and output nodes are Q-values for all actions.

The difference between these two options resides in neural net weight update. Calculating a single output and changing weights after each step positions our system into immediate reward procedure which does not perform well in long term; in case there might be an instance which can cause a drastic change in the network. On the other hand, an offline reinforcement learning method, takes a decision for next state relying on maximum output value for each action.



.Fig1



.Fig2

Fig1 is the basic Deep Q-network. Using that architecture we are able to run pairs of state-action and get the corresponding value by performing one forward pass to underlying network. Q-value we receive is immediate reward and system updates weights accordingly.

Fig2 is Deep Q-network used by DeepMind in their 2014 paper. Network receives only a state vector and outputs all possible Q-values in one forward pass. Proposed architecture is highly optimized and it is the one we will be using in our implementation.

- *Experience Replay*

An important part of training the network are the transitions provided for the learning process to occur. If our learning procedure is always using the latest one, then the system may run into problems like falling into a local minima.

In order to minimize the possibility of such an event, Experience replay technique is used. The approach is to store all obtained experiences into a memory space, and then generate random batches and provide them to the underlying network for learning process.

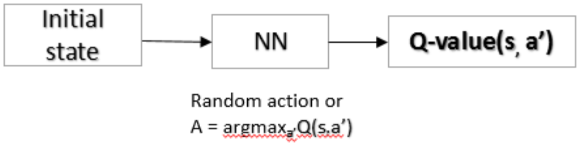
- *ε -greedy exploration (Exploration vs Exploitation)*

With time, Q-learning function will have built some part of Q-table and it will return more consistent values. Exploitation approach will have the search focused on the particular region that will improve the solution we already have.

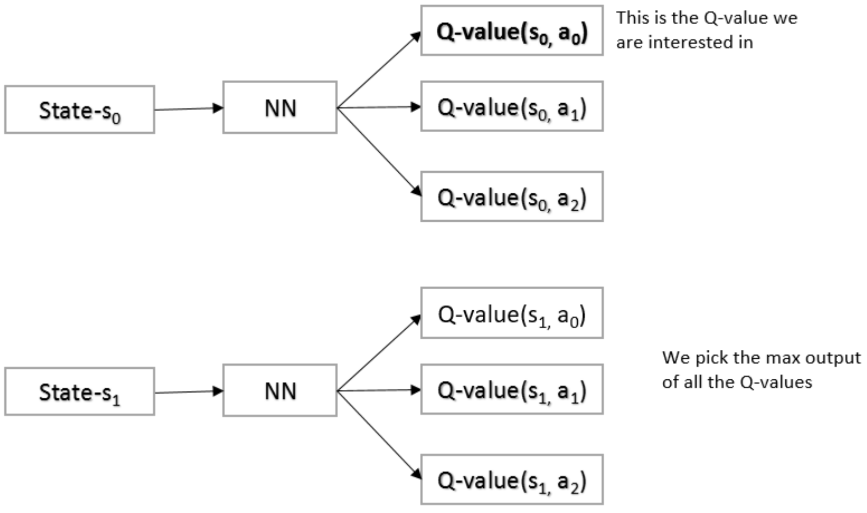
However, we want our Q-learning algorithm to probe at a large portion of state vectors in order to gather more information and discover other solution that might be more efficient, thus having a high exploration rate. This rate corresponds to extending the search to avoid getting into a local optimum. For exploration rate not to decrease by time, we need to allow some randomness for the undertaken action, which can be handled by setting a pre-defined probability ε.

The modified Q-learning algorithm exploits experience replay and greedy exploration as follows:

- When the environment is reset we are given an initial state. Then, we choose the maximum action according to Q-function output or a random one with probability ε. We execute the action and receive a reward and new state and store it into memory space. For each state, we execute a forward pass in the NN and collect experiences as tuples of <s,a',r',s'> . Each experience is inserted into the memory space.



- During learning phase, a batch of experiences is generated from memory space and then two forward passes on the neural network are performed for each of the tuples to receive target values.



Q – Algorithm using Deep Neural Networks

- Determine target Q-values.

  Target value of action a_0 : $\mathrm{Q}(\mathrm{s}_0, \mathrm{a}_0) = \mathrm{r} + \gamma \mathrm{max}_{\mathrm{a}_1} \mathrm{Q}(\mathrm{s}_1, \mathrm{a}_1)$

  For all other actions, target Q-value is what was produced by the first pass.

- Train the network using as a loss function: $\frac{1}{2} * [r + max_{a'} Q(s', a') - Q(s, a)]^2$

  For all other actions, error is defined as 0. Then backpropagation is used to update weights.

Each epoch is finished when the last state is of the form:

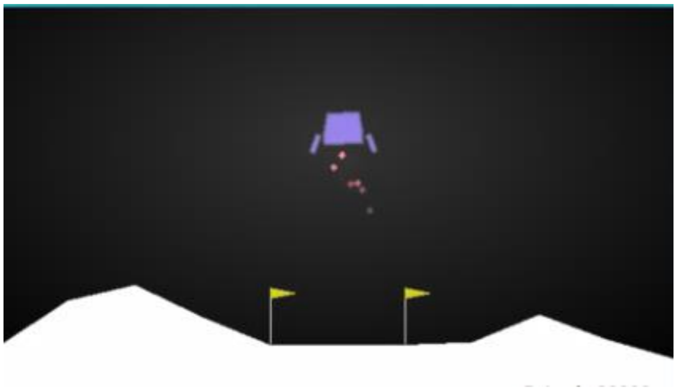
<Initial state: s, action: a, reward: +100, new state: ground coordinates (0, 0)>

We implemented this algorithm in developing Lunar Lander automatic game playing system.

3. Lunar Lander Model Construction

Gym library is used to set up the environment on top of which the algorithm is implemented. We have inherited game model construction from Gym and have adjusted the parameters conveniently.

The position of lunar lander at a particular moment is defined by the first two numbers of the state vector. Landing pad is at coordinates (0, 0). For each episode, the lander leaves its space craft and follows a sequence of state vectors until it comes to rest at the landing pad. This trajectory of state vectors is collected into replay memory space and used later for the learning process.



Lunar Lander Game

A state vector consists of four numbers; height of lander above landing surface, downward velocity, amount of fuel, reinforcement signal value which adapts to lander's performance over time.

There are four possible actions: do nothing, fire left orientation engine, fire main engine, fire right orientation engine.

An episode is over at the end of each trajectory if there is a safe landing or lander crashes receiving +100 or -100 respectively. If lander drifts away from landing pad it loses reward. For each contact performed with landing surface reward value is +10.

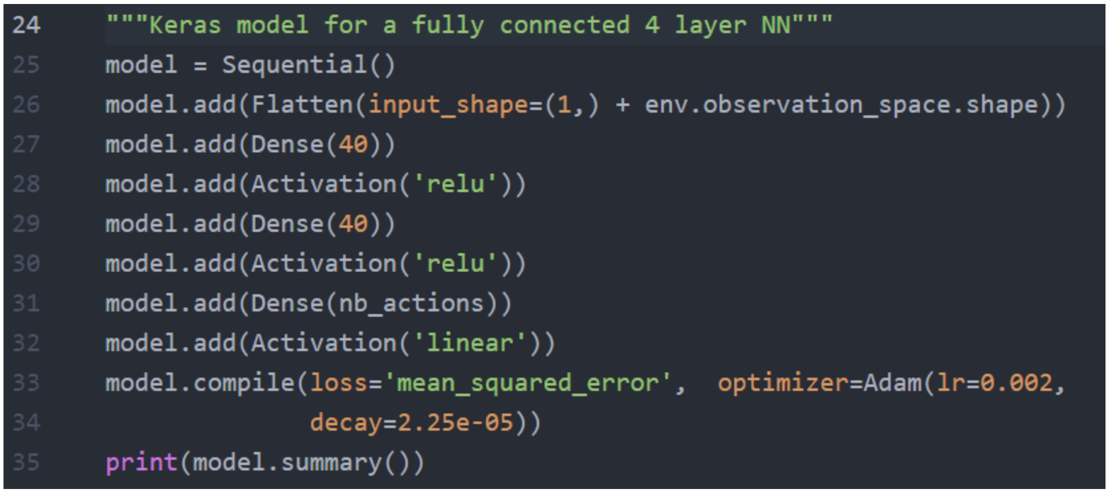
Game is solved if an average of 200 points are collected over 100 sequential trails.

4. Code Documentation – Software Dependencies

Q-learning algorithm was implemented in Python.

- *Neural network model implementation*

The neural network model we have used to implement Q- function has 8 input neurons, two hidden layers with 40 nodes each and 4 output nodes – for each possible action. Graph is fully connected and it outputs Q-values which are generated using mean squared error function.



```
"""Keras model for a fully connected 4 layer NN"""
model = Sequential()
model.add(Flatten(input_shape=(1,) + env.observation_space.shape))
model.add(Dense(40))
model.add(Activation('relu'))
model.add(Dense(40))
model.add(Activation('relu'))
model.add(Dense(nb_actions))
model.add(Activation('linear'))
model.compile(loss='mean_squared_error',  optimizer=Adam(lr=0.002,
              decay=2.25e-05))
print(model.summary())
```

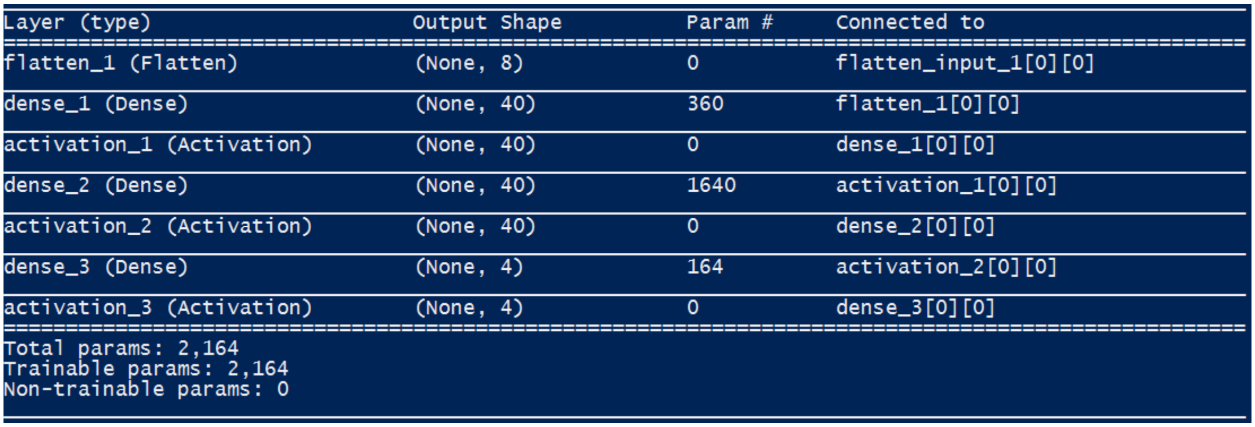
Code implementation for building the deep neural net

The activation function defined as an input to activation for each layer:

$$\mathrm{Q}(\mathrm{s}_0, \mathrm{a}_0) = \mathrm{r} + \gamma \mathrm{max}_{\mathrm{a}_1} \mathrm{Q}(\mathrm{s}_1, \mathrm{a}_1)$$

Cost function to estimate the error: defined as loss function – mean_squared_error

$$\frac{1}{2} * [r + max_{a'} Q(s', a') - Q(s, a)]^2$$



```
Layer (type)                     Output Shape          Param #     Connected to
====================================================================================================
flatten_1 (Flatten)              (None, 8)             0           flatten_input_1[0][0]
____________________________________________________________________________________________________
dense_1 (Dense)                  (None, 40)            360         flatten_1[0][0]
____________________________________________________________________________________________________
activation_1 (Activation)        (None, 40)            0           dense_1[0][0]
____________________________________________________________________________________________________
dense_2 (Dense)                  (None, 40)            1640        activation_1[0][0]
____________________________________________________________________________________________________
activation_2 (Activation)        (None, 40)            0           dense_2[0][0]
____________________________________________________________________________________________________
dense_3 (Dense)                  (None, 4)             164         activation_2[0][0]
____________________________________________________________________________________________________
activation_3 (Activation)        (None, 4)             0           dense_3[0][0]
====================================================================================================
Total params: 2,164
Trainable params: 2,164
Non-trainable params: 0
____________________________________________________________________________________________________
```

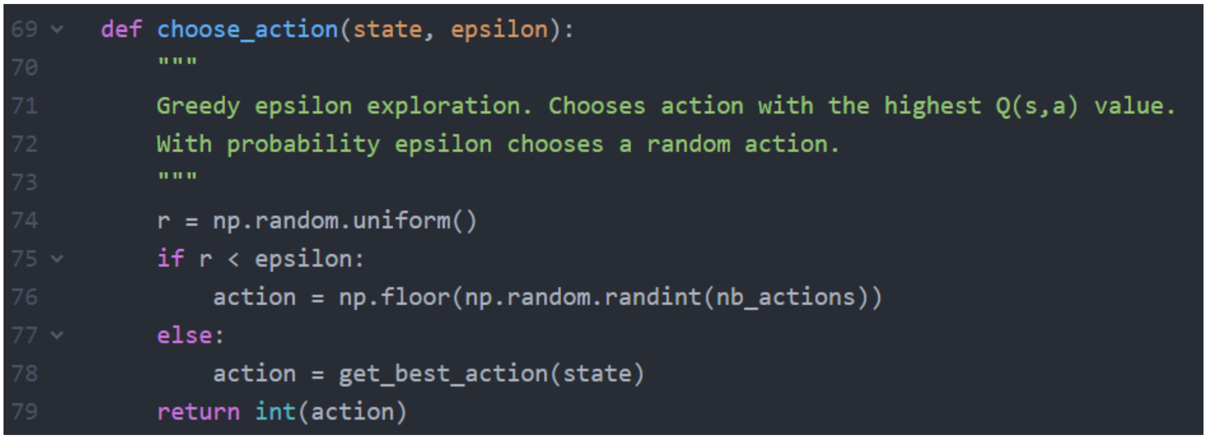
Printed summary of the model

- *Greedy exploration implementation*

Epsilon value we have chosen to permit random selection of an action is 0.1

Given an input state, choose_action function will generate an action which will be either random, or best action based on highest Q(s,a) value. This value will be determined by get_best_action function.

Get_best_action function will run a forward pass to receive all Q-values for each action, and then choose the maximum action.



```python
def choose_action(state, epsilon):
    """
    Greedy epsilon exploration. Chooses action with the highest Q(s,a) value.
    With probability epsilon chooses a random action.
    """
    r = np.random.uniform()
    if r < epsilon:
        action = np.floor(np.random.randint(nb_actions))
    else:
        action = get_best_action(state)
    return int(action)
```

Greedy exploration implementation in choosing actions

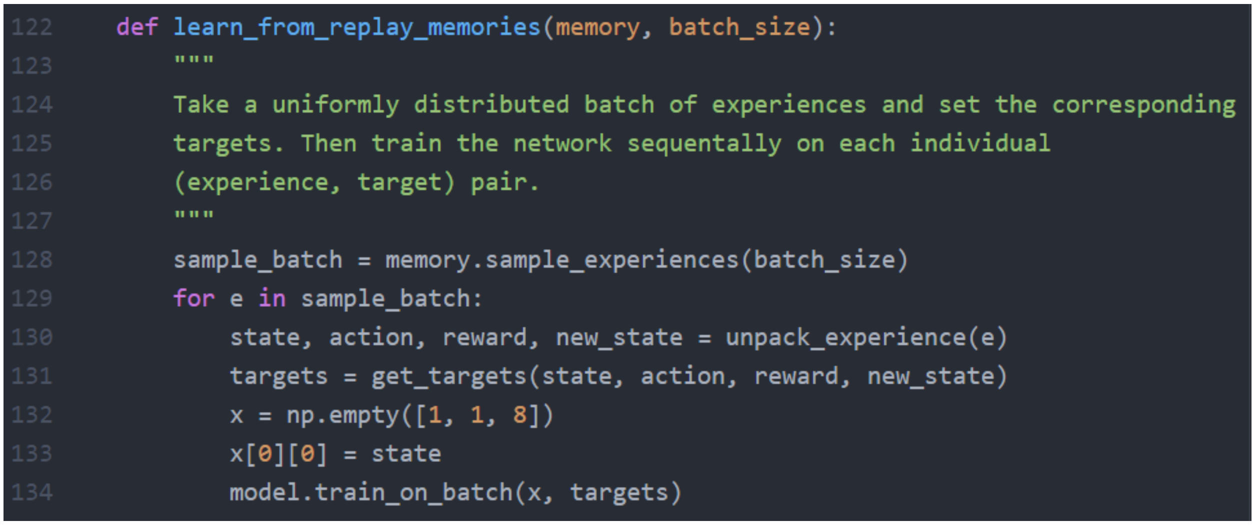
- *Experience replay implementation*

Experience replay is implemented as a function that makes use of memory space and a fixed batch size to learn from past experiences.

We have determined memory space size to be 25 and batch size to 5 experiences. Get_targets function called for each experience returns the set of target Q-values.

For action a_0 the target value is calculated as $r + \gamma \max_{a_1} Q(s_1, a_1)$ whereas for other actions, target it is the output of the first forward pass.

Gamma value is initialized to 0.1 and it is gradually increased to improve the performance of future rewards.



```python
def learn_from_replay_memories(memory, batch_size):
    """
    Take a uniformly distributed batch of experiences and set the corresponding
    targets. Then train the network sequentially on each individual
    (experience, target) pair.
    """
    sample_batch = memory.sample_experiences(batch_size)
    for e in sample_batch:
        state, action, reward, new_state = unpack_experience(e)
        targets = get_targets(state, action, reward, new_state)
        x = np.empty([1, 1, 8])
        x[0][0] = state
        model.train_on_batch(x, targets)
```

Experience replay implementation

- *Training the network – gist of the algorithm*

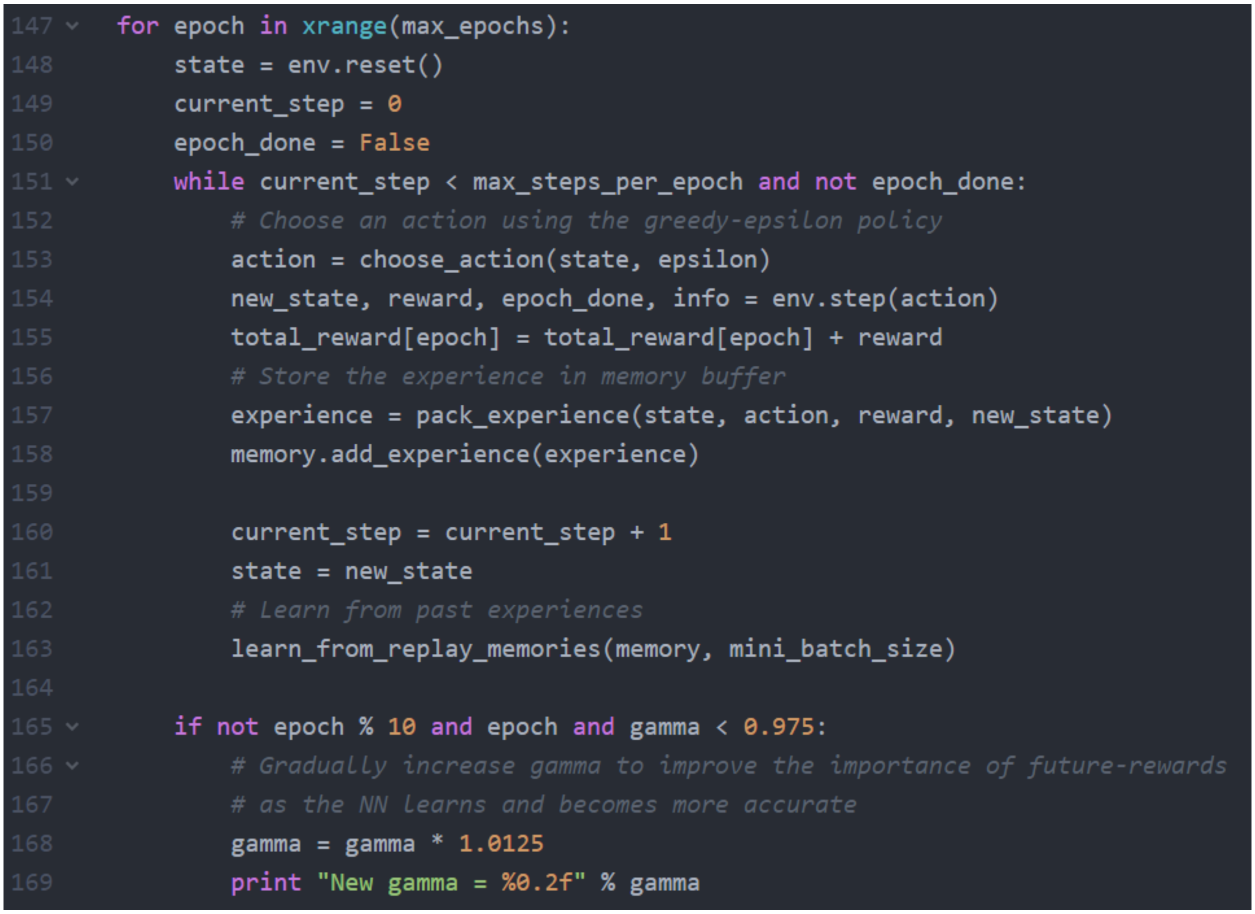
The important part of the algorithm is performed in two steps:

- Experience generation
- Train model on experiences.

Variables max_epochs is initialized to 5000 and max_steps_per_epoch to 1000. After each epoch is done, the environment is reset to its initial state.

For each epoch, an action is chosen using greedy approach. That action is inputted to the environment using step function and a new state and reward is outputted. These data altogether create a new experience which is inserted to the memory space.

System learns from a random batch of experiences by a call to learn_from_replay_memories function.



```python
for epoch in xrange(max_epochs):
    state = env.reset()
    current_step = 0
    epoch_done = False
    while current_step < max_steps_per_epoch and not epoch_done:
        # Choose an action using the greedy-epsilon policy
        action = choose_action(state, epsilon)
        new_state, reward, epoch_done, info = env.step(action)
        total_reward[epoch] = total_reward[epoch] + reward
        # Store the experience in memory buffer
        experience = pack_experience(state, action, reward, new_state)
        memory.add_experience(experience)

        current_step = current_step + 1
        state = new_state
        # Learn from past experiences
        learn_from_replay_memories(memory, mini_batch_size)

    if not epoch % 10 and epoch and gamma < 0.975:
        # Gradually increase gamma to improve the importance of future-rewards
        # as the NN learns and becomes more accurate
        gamma = gamma * 1.0125
        print "New gamma = %0.2f" % gamma
```

Two Step System Training Implementation

- *Software Dependencies:*
  (Python Code was run on Windows 10 platform)

System dependencies necessary for correct execution are:

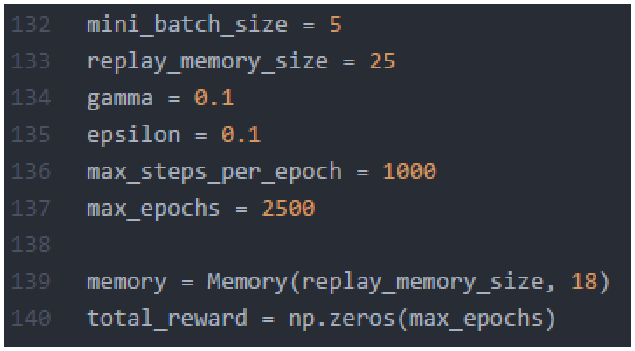
* Python 2.7 [https://www.python.org/download/releases/2.7/]
* Numpy [http://www.numpy.org/]
* Scipy [https://www.scipy.org/]
* Keras [https://keras.io/]
* OpenAI Gym [https://github.com/openai/gym]

5. Experiments and Evaluation

We ran several experiments in order to find the most efficient parameters. Given that the algorithm took at least 2 hours to converge after every individual change of the parameters, we tried to come up with the specific combinations that would maximize the performance in general.

We ran the algorithm on 5 different combinations of parameters as shown below. Each version was run for 2500 epochs.

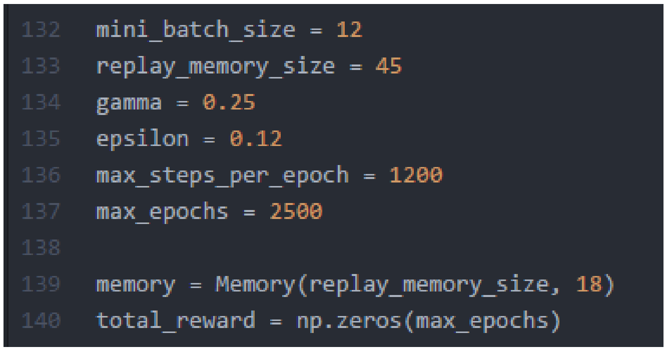
Parameters – A



```
mini_batch_size = 5
replay_memory_size = 25
gamma = 0.1
epsilon = 0.1
max_steps_per_epoch = 1000
max_epochs = 2500

memory = Memory(replay_memory_size, 18)
total_reward = np.zeros(max_epochs)
```

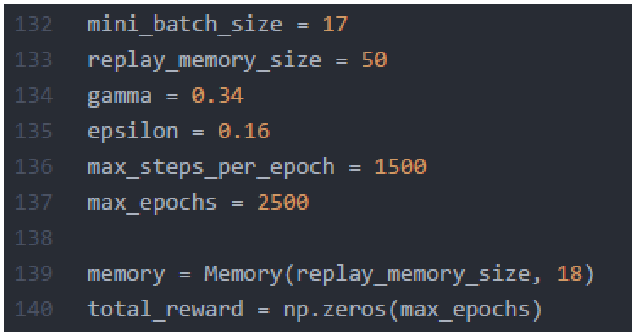
Parameters - B



```
mini_batch_size = 12
replay_memory_size = 45
gamma = 0.25
epsilon = 0.12
max_steps_per_epoch = 1200
max_epochs = 2500

memory = Memory(replay_memory_size, 18)
total_reward = np.zeros(max_epochs)
```

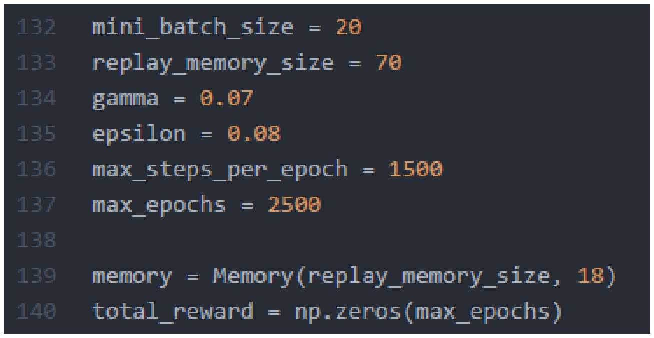
Parameters - C



```
mini_batch_size = 17
replay_memory_size = 50
gamma = 0.34
epsilon = 0.16
max_steps_per_epoch = 1500
max_epochs = 2500

memory = Memory(replay_memory_size, 18)
total_reward = np.zeros(max_epochs)
```

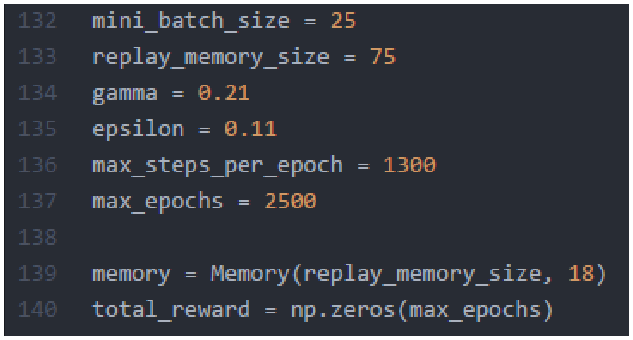
Parameters - D



```
mini_batch_size = 20
replay_memory_size = 70
gamma = 0.07
epsilon = 0.08
max_steps_per_epoch = 1500
max_epochs = 2500

memory = Memory(replay_memory_size, 18)
total_reward = np.zeros(max_epochs)
```

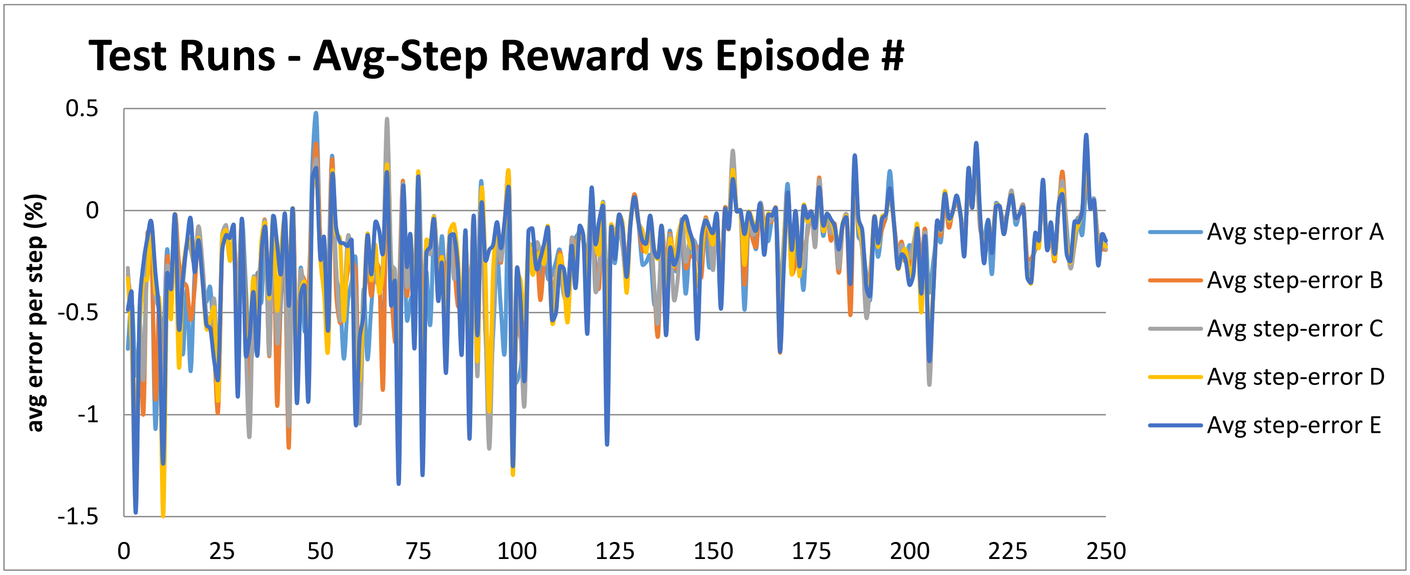
Parameters - E



```
mini_batch_size = 25
replay_memory_size = 75
gamma = 0.21
epsilon = 0.11
max_steps_per_epoch = 1300
max_epochs = 2500

memory = Memory(replay_memory_size, 18)
total_reward = np.zeros(max_epochs)
```

Below we can observe the performance of the learning algorithm corresponding to 5 different learning runs (A ,B, C, D, E) of 2500 episodes each:



Every unit on the X-axis corresponds to 10 episodes.

As it can be observed from the graph, there are continuous fluctuations during the first 1250 episodes. That is because initially, the underlying model returns quasi-random estimations of the Q values; but as it gains experience and learns from the training memories, the Q-values begin to approach a more precise and consistent state.

Each learning run tends to average toward a positive mean and gradually converges. However, there might still be occasional sharp changes of reward values. This can be attributed to the randomness inherent to the ε-greedy approach we have used.

Detailed output data for each run can be found in attached excel file: test_bed_results.xlsx

Some of the conclusions that we were able to achieve by running partial learning trials, using various combination of parameters are:

- Diminishing memory size improves accuracy of the system. That is because experiences tend to be more accurate with time and keeping the latest ones in memory space assures the mini batch will contain consistent tuples.
- Gradually increasing decay factor (gamma) from 0.1 to 0.975 improved the learning rate performance. This is consistent with the fact that the network trust on its future rewards estimate increases over time as it learns.

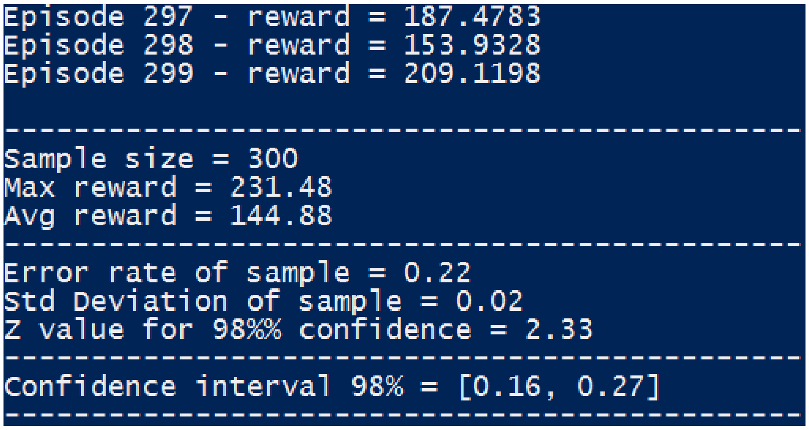
We also determined the following possible improvements to increase the performance rate:

- Gradual decay of epsilon factor
- Testing with more configurations of memory size and mini batch size.

- *Error rate and confidence intervals:*

In order to determine the performance of the trained model, we devised a bed test with 300 random episodes. For testing purposes we defined "winning" the episode as a Boolean value that represents the event of obtaining an overall score of 100 or more in addition to safely landing the aircraft.

Below we can observe the obtained results.



```
Episode 297 - reward = 187.4783
Episode 298 - reward = 153.9328
Episode 299 - reward = 209.1198

----------------------------------------------
Sample size = 300
Max reward = 231.48
Avg reward = 144.88
----------------------------------------------
Error rate of sample = 0.22
Std Deviation of sample = 0.02
Z value for 98%% confidence = 2.33
----------------------------------------------
Confidence interval 98% = [0.16, 0.27]
----------------------------------------------
```

6. References

[1] Michael Nielson, Neural Network and Deep Learning http://neuralnetworksanddeeplearning.com/

[2] Nervanasys.com, Demystifying Deep Reinforcement Learning https://www.nervanasys.com/demystifying-deep-reinforcement-learning/

[3] Keras Documentation: https://keras.io/

[4] Learningmachines101.com, How to build a lunar lander autopilot learning machine http://www.learningmachines101.com/lm101-025-how-to-build-a-lunar-lander-autopilot-learning-machine/

[5] Gym.openai.com, LunarLander-v2 https://gym.openai.com/envs/LunarLander-v2

7. Links

Python Code – Github : https://github.com/raskolnnikov/deep_q_learning

Youtube Videos:

Untrained network: https://www.youtube.com/watch?v=T72fQgUWD90

Trained network: https://www.youtube.com/watch?v=kTpkaOgMBEM