# How to run

Run on Python 3.7

To run python script:

python filename.py

The models produced can be large, so ensure that there is sufficient storage space

If the log tasks boolean is set to True, then the each task will be logged to file

Recommend start with 4 agents, 20 tasks, 50 episodes, a single k and n value to test the functionality and observe output (e.g. K1N0 or K5N1)

To display the performance on TensorBoard while the script is running:

tensorboard --logdir=logs

(in the directory where the .py file is stored)

This will output an address e.g. [http://EN402316:6006/](http://en402316:6006/)

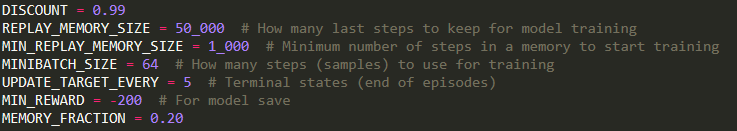
Note: Access the address through Firefox or other browser as Chrome does not allow this

## Required packages

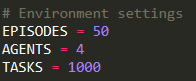
* Keras 2.3.1
* TensorFlow 1.15
* Tqdm
* Xlwt
* opencv or cv2
* PIL
* time

### Modifying code: OPTIONAL

To modify the **parameters** of the system, edit the associated variables in the header of the script



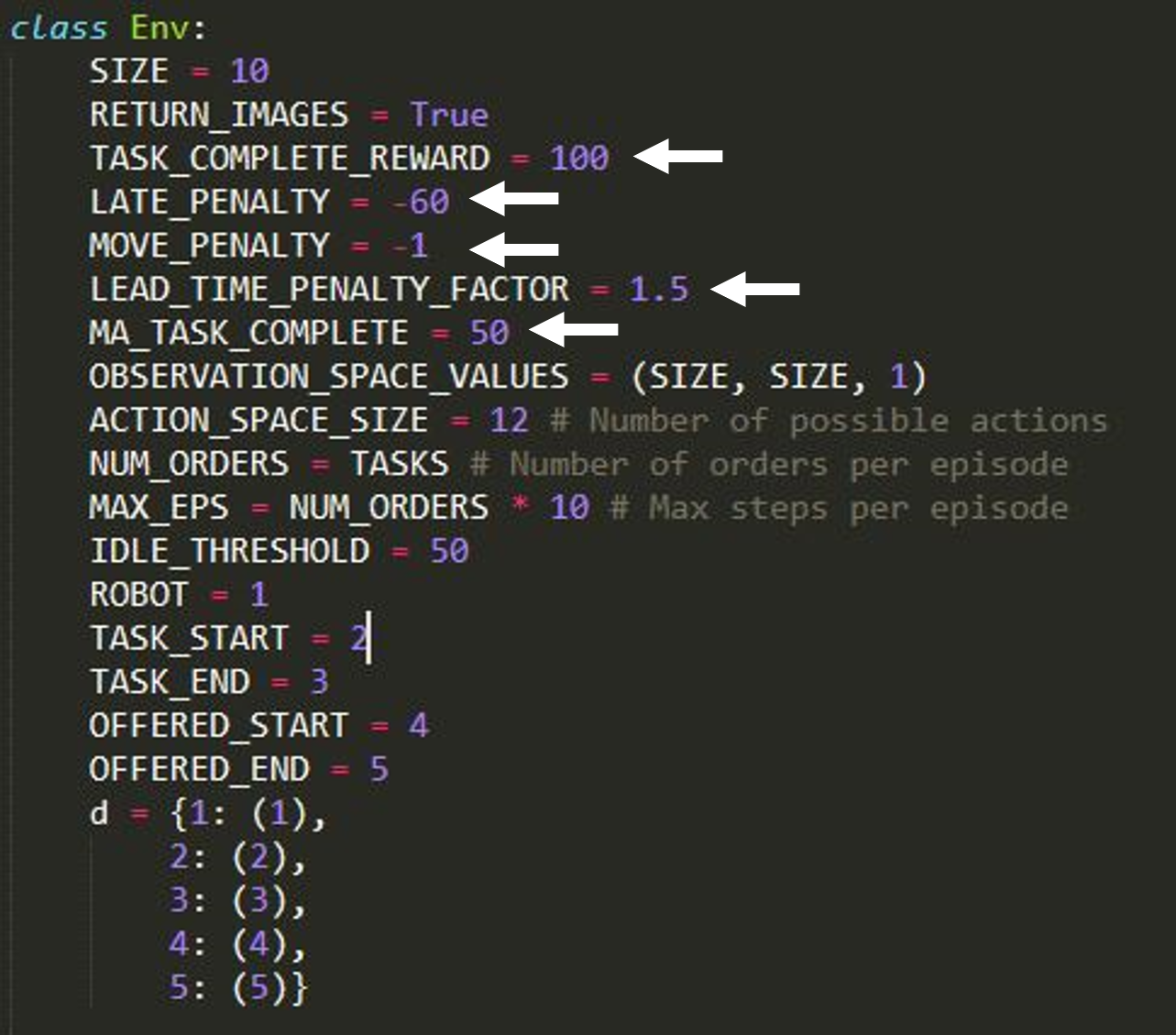
To edit the **environment settings - number of episodes, agents and tasks**, in the header section of the script:



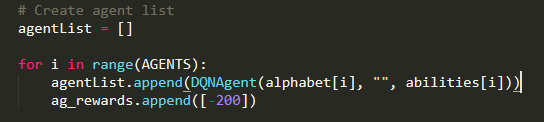
To set the **look-ahead** and **individual queue** parameters, in the header section of the script



To adjust the **reward function**, in the environment class:



To **load model** or **specify abilities** for each agent manually, in the main loop:



In place of “” in the DQNAgent() initialisation, insert path to model e.g.

agentA = DQNAgent('A', **‘C:\TempDataMMOH360\models\TRAIN\_K1\_N2-A\_\_26432.50max\_7392.45avg\_\_255.00min\_\_1605508878.model'**, 2)

## Observation

An rgb array that consists of:

* the end positions of the task the robot currently has - which is its current position if no task has been assigned
* the start and end position of the task being offered

## Model vs target model

From <https://pythonprogramming.net/deep-q-learning-dqn-reinforcement-learning-python-tutorial/>

**Here, you can see there are apparently two models: self.model and self.target\_model. What's going on here? So every step we take, we want to update Q values, but we also are trying to predict from our model. Especially initially, our model is starting off as random, and it's being updated every single step, per every single episode. What ensues here are massive fluctuations that are super confusing to our model. This is why we almost always train neural networks with batches (that and the time-savings). One way this is solved is through a concept of memory replay, whereby we actually have two models.**

**The target\_model is a model that we update every every n episodes (where we decide on n), and this the model that we use to determine what the future Q values.**

**Once we get into working with and training these models, I will further point out how we're using these two models. Eventually, we converge the two models so they are the same, but we want the model that we query for future Q values to be more stable than the model that we're actively fitting every single step.**

**Along these lines, we have a variable here called replay\_memory. Replay memory is yet another way that we attempt to keep some sanity in a model that is getting trained every single step of an episode. We still have the issue of training/fitting a model on one sample of data. This is still a problem with neural networks. Thus, we're instead going to maintain a sort of "memory" for our agent. In our case, we'll remember 1000 previous actions, and then we will fit our model on a random selection of these previous 1000 actions. This helps to "smooth out" some of the crazy fluctuations that we'd otherwise be seeing. Like our target\_model, we'll get a better idea of what's going on here when we actually get to the part of the code that deals with this I think.**

## Model.fit()

Trains the model for a fixed number of iterations

It slices the data into batches and then based on the loss calculated by the function, it alters/fits the model to perform more desirably.

## \*\* in Python

Unpacks function arguments from an arbitrary number of arguments

Or can be used to unpack a dictionary

## DQN Agent

### Train

Overall process:

Checks if there is sufficient memory to train

Takes minibatch - (mentioned below) a sample of the memory available

Get the current states from minibatch

Get the q-values based on the current states taken from minibatch - these values are predicted using the model

Get the new states from minibatch

Get the new q-values based on the new states taken from minibatch - these are predicted using the target model

For each set of data in the minibatch

The q-values are updated using the Bellman equation

The current state and new q is appended to the list of states and q values

Fit the model based on the “current states” and the q-values

If the desired number of iterations has been reached, then the target model is updated

### MINIBATCH

Training the system in batches instead of using all the training data reduces the computational intensity and increases efficiency.

The minibatch consists of a random sample (the size of the minibatch) of the double ended queue (deque) used to store the entire replay memory.

The replay memory stores the current state, the action taken in that state, the reward associated with that state-action pair, the “new” state, and a boolean representing whether the objective is complete.

## TQDM

Used for visualisation of progress for the episodes running (shows the progress bar while running)

## Episode rewards

The episode reward is initialised to an arbitrary value of -200 for each agent. This is required as the rewards are used to determine the growth and success of the model, and so the system requires a starting point for this growth.