Alejandro Mendoza Puig

Report for p1_navigation project from Deep Reinforcement Learning Nano Degree

Learning Algorith

The model used for this project was a deep Q network, which was first introduced by DeepMind on a paper where they trained an agent to play several atari games with super human capabilities.

The reason Deep Q Networks was the approach for this project is that we have a continuous state space, which is a difficult task to tackle with normal Q-Learning, so instead of having a cintinous state space and later discretize it we will use a Neural network that will be able to approach obtain an action value function that will provide is with a good policy (if not the optimal policy in some cases)

First we need to build the agent and its functions (in the dqn_agent.py file) this class contains the information to create an agent that is able to receive a specific state and return an action. Later the agent sees the reward it gets and the new state, we'll keep this in a dictionary so our agent can later learn sampling from random state, action, rewards tuples.

In order to generate an action value function our agent will need a neural network structure there are some parameters that need to be fixed and the others can be modified in order to create a more or less capable agent.

Our agent's NN will need to receive a given state and it will need to predict one of the posible actions so the input and the output must be fixed

- Input Layer: Must have a vector of 37 dims, which is the state space size
- Output Lare: Must have a vector of 4 dims, which is the action space for out agent (fwd, back, left, right)
- Hidden Layers: We can have as many hidden layers as we want and as many neurons as we want. In this case we have:
  - 1st fully conected layer: 64 neurons with relu activation
  - 2nd fully connected layer: 32 neurons with relu activation
- Learning rate: 0.0005
- Optimizer: ADAM

In order to update the weights for our agent we need some specific parameters.

- **Replay** While the agent is navigating through the environment we collect a buffer of **100,000** state, action, reward, state' tuples. We do this to avoid correlations that will affect negatively the learning of our agent
- **Discount** In this case we defined a discount factor of 0.99, this will enable to take into consideration old rewards during the update of the wights, if we set it to 0 we'll have an agent that will have "no memory" of old rewards

Findings (Human Learning)

During this project I did several changes on the parameters of the learning process (decay, initial epsilon, minumum epsilon, neural network). So the agent learned as well as I did

The main findings I found interesting where the following:

- Hidden Layers [32, 64 and 128 neurons]: Changing hidden layers changed a tiny bit the final score of the agent, so I consider that 32 and 64 neurons per layer is the way to go
- Decay: a lowee decay helped the model reach faster (in some cases) the 15 points average reward, however at the end the final avg reward was lower than with 0.995 (only by +/- 0.5 reward)

All the modifications had a performance that exceeded the 13 average reward on the last 100 episodes out of 1800.

The model in the saved checkpoint has a structure of 64 neurons in the fc1 and 32 in the fc2

Ideas for future work

I'd like to try the pixel approach for this project as well as keep exploring the hyperparameters of the agent and the neural network. I want to see whats the "leanest" model architecture that I can build that still has a good performance, in order to be the most efficient computation power wise.

In [ ]:

```

```