Project Navigation

Provided code

First two cells provide the code to install packages for the environment and the environment itself.

Environments contain ***brains*** which are responsible for deciding the actions of their associated agents. In the third cell we check for the first brain available, and set it as the default brain we will be controlling from Python.

In the fourth cell we access some information about the environment: how many agents, actions, states and what the state space looks like.

In the last cell of provided code we let the agent take random actions and check the score (0 or 1).

Model

Model for learning non-linear patterns I chose to use is comprised of 3 fully connected layers with ReLU activation function and Adam optimizer. Hyperparameters are set to those used in previous lectures and the research paper on DQN.

Agent

Agent contains class ReplayBuffer which records every step (action taken) to its memory. When memory has enough of examples (BATCH_SIZE) they are passed to the model to learn. In learning step model computes targets and compares them to the states from the memory with Mean Squared Error, which is then used for backpropagation and optimizer step to update the weights.

Every few time steps (UPDATE_EVERY), new batch of random samples is generated from memory and passed to the model to learn.

"dqn" Method

First, dqn gets the information from the brains of the environment on the initial state and sets the score of the game to 0. If the memory of ReplayBuffer does not have enough examples to start training the model - the action is selected randomly, if it does – the action is selected using epsilon greedy policy choosing either random action or the action predicted by the model in (1 – epsilon)*100% cases.

Then, selected action is passed back to the environment and the next state is observed. The information on the next state is added to the BufferReplay memory containing the previous state, action, reward, current state and whether the episode is done (steps reached the state_size). Collected reward is added to the score.

Every 100 episodes the method prints the average score and once the score of 13 reached, it saves the weights of the model into the checkpoint.pth file and finishes the training.

Following cells run the method, plot the graph of scores and test the agent with saved weights on evaluation mode and epsilon almost 0.

Future work

The biggest challenge for which I could not find a solution is to try Double and Convolution DQNs. Conv2D expects data in 4 dimensions, and while "torch.from_numpy(state).float().unsqueeze(0).to(device)" adds a dimension to the format, it doesn't work twice, or 3 times as needed. I was searching the internet trying different possible solutions for a few days, but I was always getting some kind of an error until I decided to move on with the nanodegree.

Another way to train the model that I didn't even try to implement is to store memories in ReplayBuffer not by steps, but by episodes, and compute the error from the total score. I suppose, the agent would have to learn more strategic ways to maximize the score. This procedure would involve adding a dimension to the format of data, which I already have problems with.