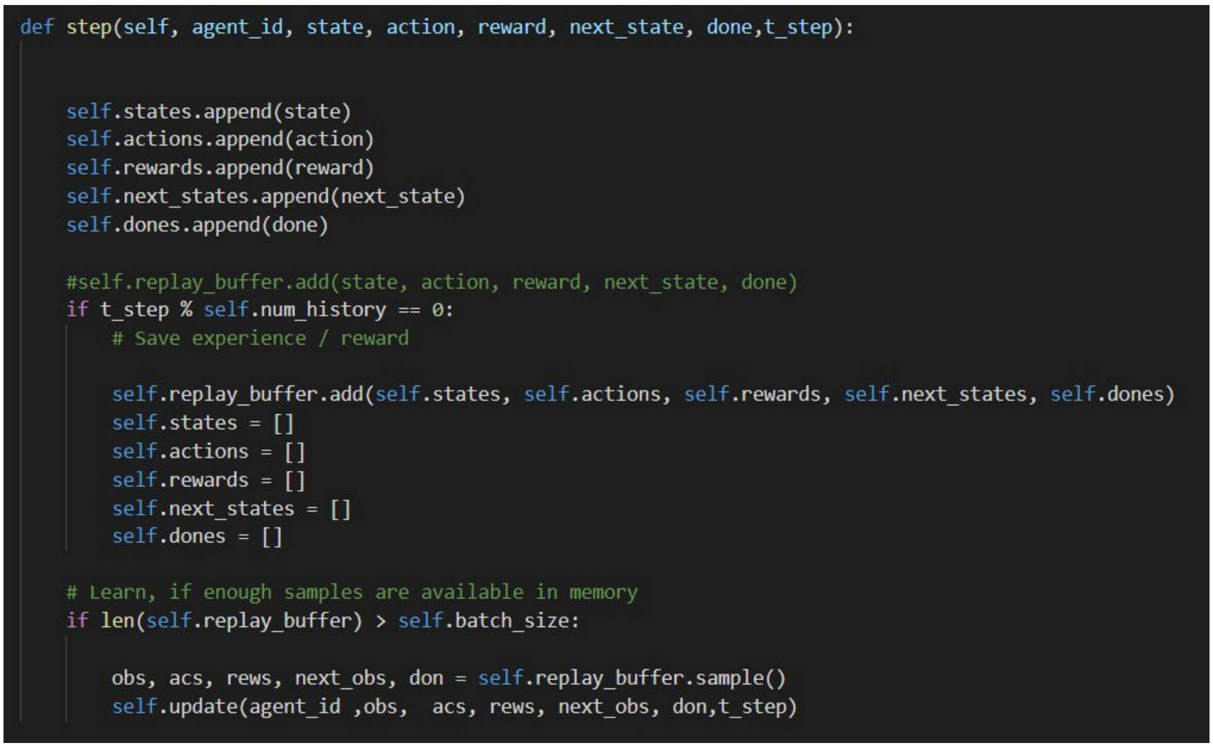
Multi Agent Project

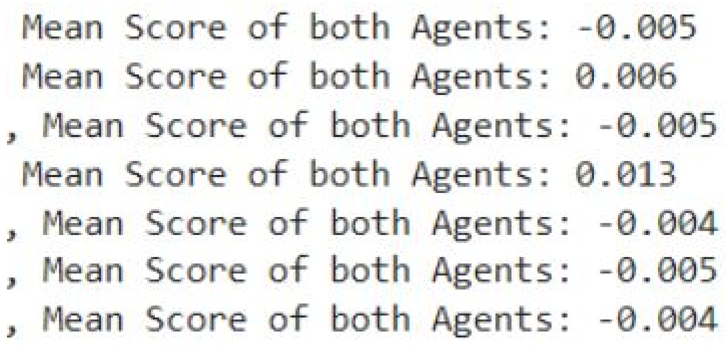
In this project i used the MADDPG algorithm to solve the problem, the code is base in https://github.com/shariqiqbal2810/maddpg-pytorch/ with some modification to various files like the utils functions, the replay_buffer and memory etc.. my first approach was to sample not randomly snapshots of the game if not continuous samples like 2 or more frames for that reason i created two ddpg python classes (ddpg.py and ddpgv2.py) which allows to do that



```
def step(self, agent_id, state, action, reward, next_state, done,t_step):


    self.states.append(state)
    self.actions.append(action)
    self.rewards.append(reward)
    self.next_states.append(next_state)
    self.dones.append(done)

    #self.replay_buffer.add(state, action, reward, next_state, done)
    if t_step % self.num_history == 0:
        # Save experience / reward

        self.replay_buffer.add(self.states, self.actions, self.rewards, self.next_states, self.dones)
        self.states = []
        self.actions = []
        self.rewards = []
        self.next_states = []
        self.dones = []

    # Learn, if enough samples are available in memory
    if len(self.replay_buffer) > self.batch_size:

        obs, acs, rews, next_obs, don = self.replay_buffer.sample()
        self.update(agent_id ,obs,  acs, rews, next_obs, don,t_step)
```

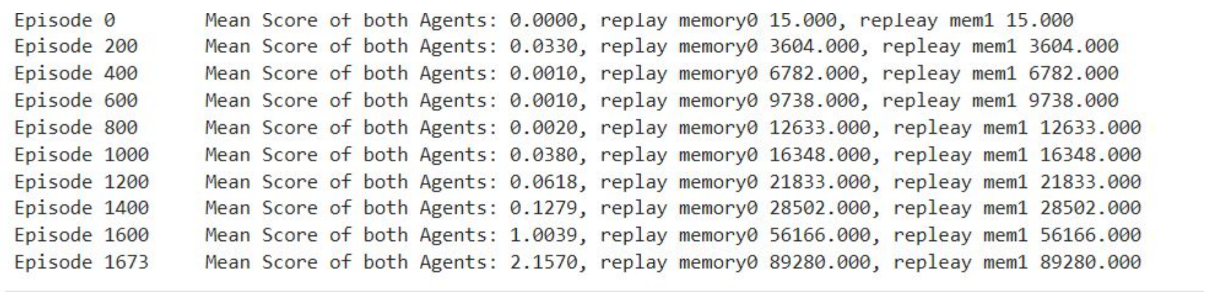
Image 1. the self.num_history controls how much continuous frames to use

but this approach did not work out because the agents got stocked in the same value for thousands of iterations i changed the network configuration (layer, learning rate, tau, noise reduction, no linear activation functions etc..) but the problem continued.

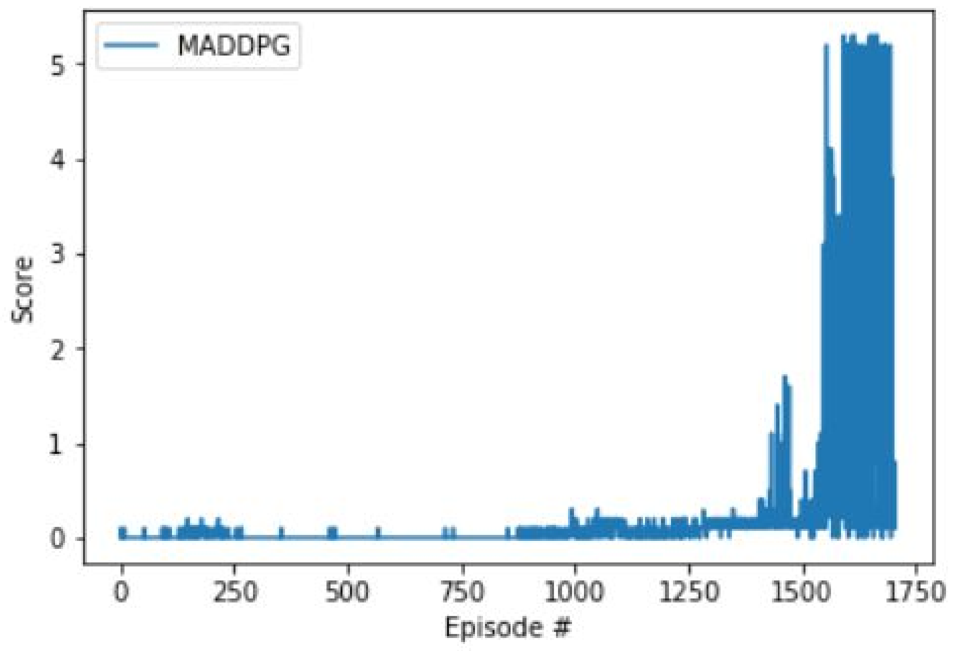


```
 Mean Score of both Agents: -0.005
 Mean Score of both Agents: 0.006
, Mean Score of both Agents: -0.005
 Mean Score of both Agents: 0.013
, Mean Score of both Agents: -0.004
, Mean Score of both Agents: -0.005
, Mean Score of both Agents: -0.004
```

so i decided to eliminated that option as you can see in ddpgv2.py which has the last version with the performance desire but uses the same algorithm (except the continuous frame option) the lines are almost equal.



```
Episode 0       Mean Score of both Agents: 0.0000, replay memory0 15.000, repleay mem1 15.000
Episode 200     Mean Score of both Agents: 0.0330, replay memory0 3604.000, repleay mem1 3604.000
Episode 400     Mean Score of both Agents: 0.0010, replay memory0 6782.000, repleay mem1 6782.000
Episode 600     Mean Score of both Agents: 0.0010, replay memory0 9738.000, repleay mem1 9738.000
Episode 800     Mean Score of both Agents: 0.0020, replay memory0 12633.000, repleay mem1 12633.000
Episode 1000    Mean Score of both Agents: 0.0380, replay memory0 16348.000, repleay mem1 16348.000
Episode 1200    Mean Score of both Agents: 0.0618, replay memory0 21833.000, repleay mem1 21833.000
Episode 1400    Mean Score of both Agents: 0.1279, replay memory0 28502.000, repleay mem1 28502.000
Episode 1600    Mean Score of both Agents: 1.0039, replay memory0 56166.000, repleay mem1 56166.000
Episode 1673    Mean Score of both Agents: 2.1570, replay memory0 89280.000, repleay mem1 89280.000
```



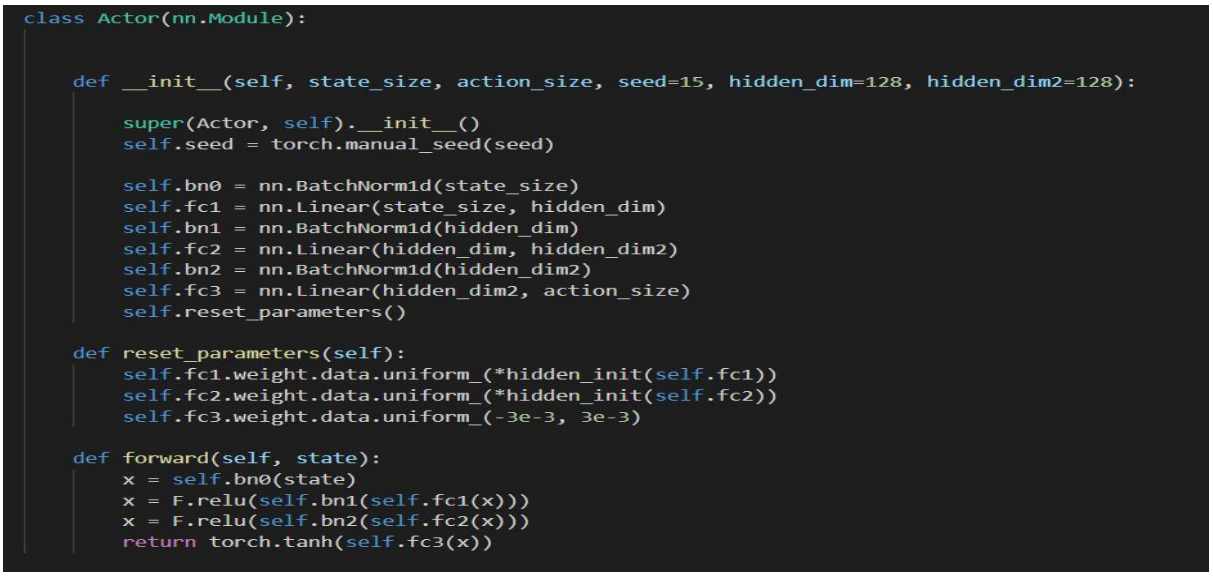
MADDPG

I decided to use MADDPG because in these environments training both agents independently will broke the assumption of non-stationary or in other words for each agent the policy will change.

the maddpg algorithm consists in: "We treat each agent in our simulation as an "actor", and each actor gets advice from a "critic" that helps the actor decide what actions to reinforce during training. Traditionally, the critic tries to predict the *value* (i.e. the reward we expect to get in the future) of an action in a particular state, which is used by the agent - *the actor* - to update its policy."
(https://openai.com/blog/learning-to-cooperate-compete-and-communicate/)

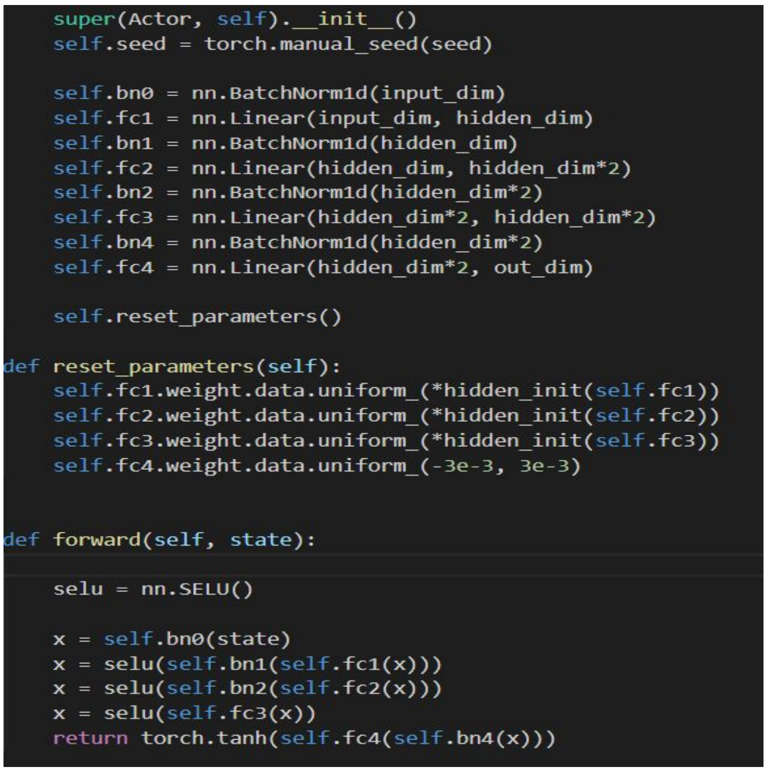
as commented above each critic will be trained using the observation and actions from all the agents and the actor component is trained with just his observation which will allows to do not broke the stationary assumption

Model



```
class Actor(nn.Module):


    def __init__(self, state_size, action_size, seed=15, hidden_dim=128, hidden_dim2=128):

        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)

        self.bn0 = nn.BatchNorm1d(state_size)
        self.fc1 = nn.Linear(state_size, hidden_dim)
        self.bn1 = nn.BatchNorm1d(hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim2)
        self.bn2 = nn.BatchNorm1d(hidden_dim2)
        self.fc3 = nn.Linear(hidden_dim2, action_size)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
        x = self.bn0(state)
        x = F.relu(self.bn1(self.fc1(x)))
        x = F.relu(self.bn2(self.fc2(x)))
        return torch.tanh(self.fc3(x))
```

For the Actor architecture you can see we used 3 relu and before each input we used batch normalization also i tried this architecture



```
    super(Actor, self).__init__()
    self.seed = torch.manual_seed(seed)

    self.bn0 = nn.BatchNorm1d(input_dim)
    self.fc1 = nn.Linear(input_dim, hidden_dim)
    self.bn1 = nn.BatchNorm1d(hidden_dim)
    self.fc2 = nn.Linear(hidden_dim, hidden_dim*2)
    self.bn2 = nn.BatchNorm1d(hidden_dim*2)
    self.fc3 = nn.Linear(hidden_dim*2, hidden_dim*2)
    self.bn4 = nn.BatchNorm1d(hidden_dim*2)
    self.fc4 = nn.Linear(hidden_dim*2, out_dim)

    self.reset_parameters()

def reset_parameters(self):
    self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
    self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
    self.fc3.weight.data.uniform_(*hidden_init(self.fc3))
    self.fc4.weight.data.uniform_(-3e-3, 3e-3)


def forward(self, state):

    selu = nn.SELU()

    x = self.bn0(state)
    x = selu(self.bn1(self.fc1(x)))
    x = selu(self.bn2(self.fc2(x)))
    x = selu(self.fc3(x))
    return torch.tanh(self.fc4(self.bn4(x)))
```

but it didn't show any difference and is more complicated (more layers)

Parameters

```python
# number of training episodes.
number_of_episodes = 4000
episode_length = 50
batch_size = 120

# amplitude of OU noise
# this slowly decreases to 0
noise = 1
noise_reduction = 0.9999

hidden_dim_actor = 256
hidden_dim_critic = 256
gamma=0.95
tau=0.001
lr_actor=0.001
lr_critic=0.001

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")


agent_init_params = {'num_in_pol': state_size,
                     'num_out_pol': action_size,
                     'num_in_critic': state_size,
                     'hidden_dim_actor': hidden_dim_actor,
                     'hidden_dim_critic': hidden_dim_critic,
                     'tau':tau,
                     'gamma': gamma,
                    'lr_actor':lr_actor,
                    'lr_critic':lr_critic,
                    'batch_size':batch_size,
                    'max_episode_len':episode_length}
```

we use several parameters like hidden dimension in the actor and critic with 256 units for each layer then it will go to a 128 dim (256 → 128). the gamma is 0.95 and tau is 0.001 (used in the soft update), some parameters are not use in the ddpgv2.py but it does in the ddpg.py (older version with continuous frames)

Next Steps

i could use another algorithm or try other configurations like recurrent neural networks in the actor and critic networks this will allows in some sense to concatenate the actions made by the agent because right now is unstable.