VDN_pseudocode

```
class Q_network:
    nn.Sequential(nn.Linear(n_obs, 64), nn.ReLU(),
                  nn.Linear(64, 32), nn.ReLU())
    sample_action():
      if rand<= eps, perform action tasks

Main():
    env= gym.make(ma_gym:Checkers-v0)
    initilalize  q-networks (Q=q &  target: Q'), optimizer(Adam, lr), reward =0

    for episode in range(max_episodes):
        state= env.reset, done= False for agent_i
         with torch.no_grad(), set q.init(hidden)
          while not all(done):
              action, hidden = q.sample_action(state, hidden, ep)
               next_state, reward, done = env.step(action) # append to memory
                score +=sum(reward), state= next_state
      if memory> threshold:
             # perform training
             s, a, r, s', done= sample from memory # sample chunks
             initialize  hidden = torch.zeros(batch_size, num_agents, hx_size), target_hidden= torch.zeros(batch_size, num_agents, hx_size), loss = 0
             for step in range:
                 q_out, hidden = q( s[steps_i], hidden) # predict the Q-values for 5 actions
                 q_a =  q_out(a[:, step_i]) #Determine the Q-values of the actions taken by the agent at step_i
                 sum_q = q_a.sum() # sum of the Q- values of the actions taken by both agents at time step_i

                 max_q', target_hidden = Q' (s', target.detach) # have Q-action value form target network for all (5) actions
                 max_q' = max_q'(dim=2) # determine for the 2 actions the Q_values
                 target_q = rewards[step_i].sum() + gamma * max_q'.sum() # Belman Equation
                 loss += Huber_loss(sum_q, target_q)

          if episode_interval= fixed_value:
                Q'= q(state_dict.update)
```