Graph Convolutional Reinforcement Learning for Multi-Agent Cooperation

HyungJin Kim

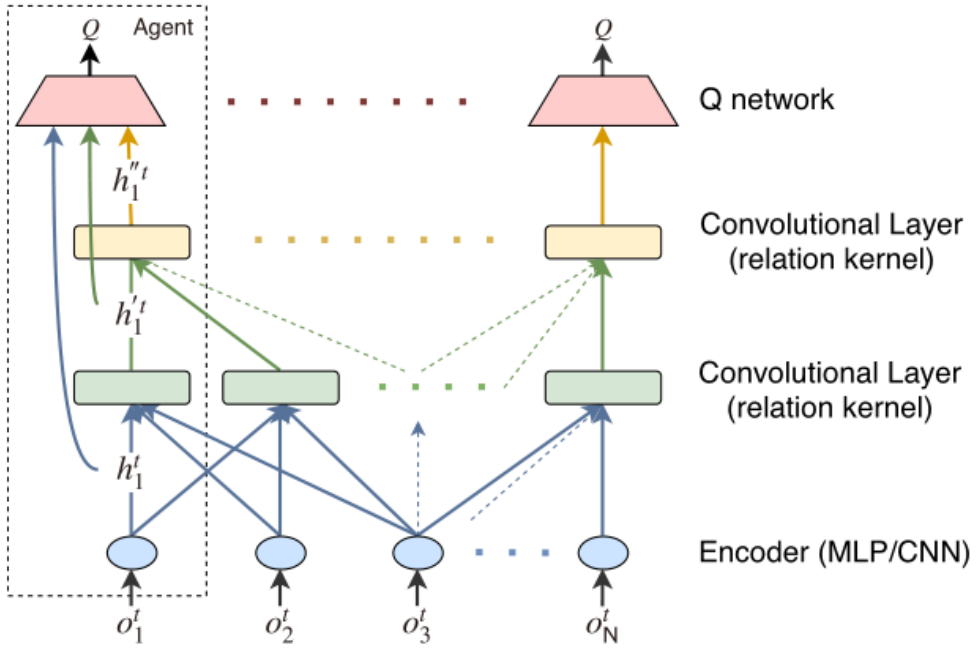
Seoul National University

DGN(Graph convolutional RL)

- Construct multi-agent environment as a graph
  - Agents in environment are represented by the nodes of the graph
  - For each node, there are K edges connected to its K nearest neighbors
  - Nearer neighbors are more likely to interact with each other
- Continuously changing over time as agents move or enter/leave the environment
  - Consider partially observable environment
  - Merge all agents' feature vectors at time t into a feature matrix F^t with size N x L in the order of index
    - N is number of agents and L is length of feature vector
  - Construct an adjacency matrix C_i^t with size (K+1) x N with agent i
    - First row is the one-hot representation of the index of node i
    - jth row is the one-hot representation of the index of the (j-1)th nearest neighbor
  - Then we can obtain the feature vectors in the local region of node i by $C_i^t \times F^t$

DGN(Graph convolutional RL)

- DGN consists of three types of modules
  - Observation encoder
    - Encode local observation into feature vector
  - Convolutional layer
    - Integrate feature vectors in the local region and generate latent feature vectors
  - Q network
    - Concatenate all preceding feature vectors and train Q-value



DGN(Graph convolutional RL)

- Observation encoder module
  - The local observation o_i^t is encoded into a feature vector h_i^t by MLP for low-dimensional input or CNN for visual input
- Convolutional layer module
  - Integrates the feature vectors in the local region and generates the latent feature vector $h_i'^t$
  - By stacking convolutional layers, the receptive field of an agent grows
  - Contains the information from nodes in (# of convolutional layers)-hop, but will not increase the local region of node i (node i still only communicates with its K neighbors)

- Observation encoder module
  - The local observation o_i^t is encoded into a feature vector h_i^t by MLP for low-dimensional input or CNN for visual input
- Convolutional layer module
  - Integrates the feature vectors in the local region and generates the latent feature vector $h_i'^t$
  - By stacking convolutional layers, the receptive field of an agent grows
  - Contains the information from nodes in (# of convolutional layers)-hop, but will not increase the local region of node i (node i still only communicates with its K neighbors)

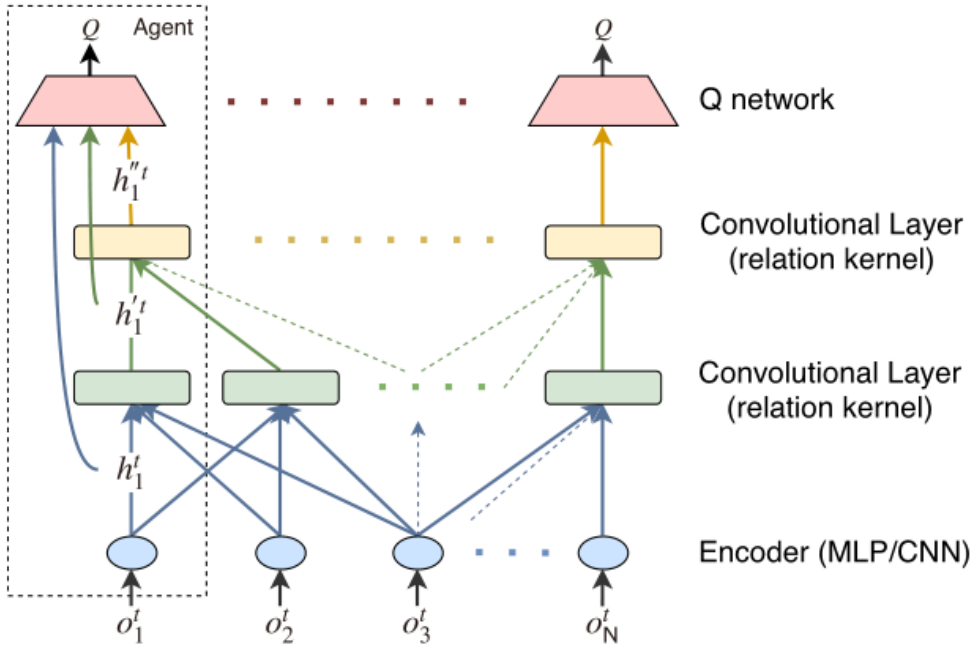
DGN(Graph convolutional RL)

- Q network
  - Centralized training and distributed execution
  - Sample a random minibatch of S samples from replay buffer B and minimized the loss

$$\mathcal{L}(\theta) = \frac{1}{\mathrm{S}} \sum_{\mathrm{S}} \frac{1}{\mathrm{N}} \sum_{i=1}^{\mathrm{N}} \left(y_i - Q\left(O_i, a_i; \theta\right)\right)^2$$

where $y_i = r_i + \gamma \max_{a'} Q\left(O_i', a_i'; \theta'\right)$

  - Keep C unchanged in two successive timesteps to make the learning process more stable
  - Each agent only requires the information from its K neighbors during execution

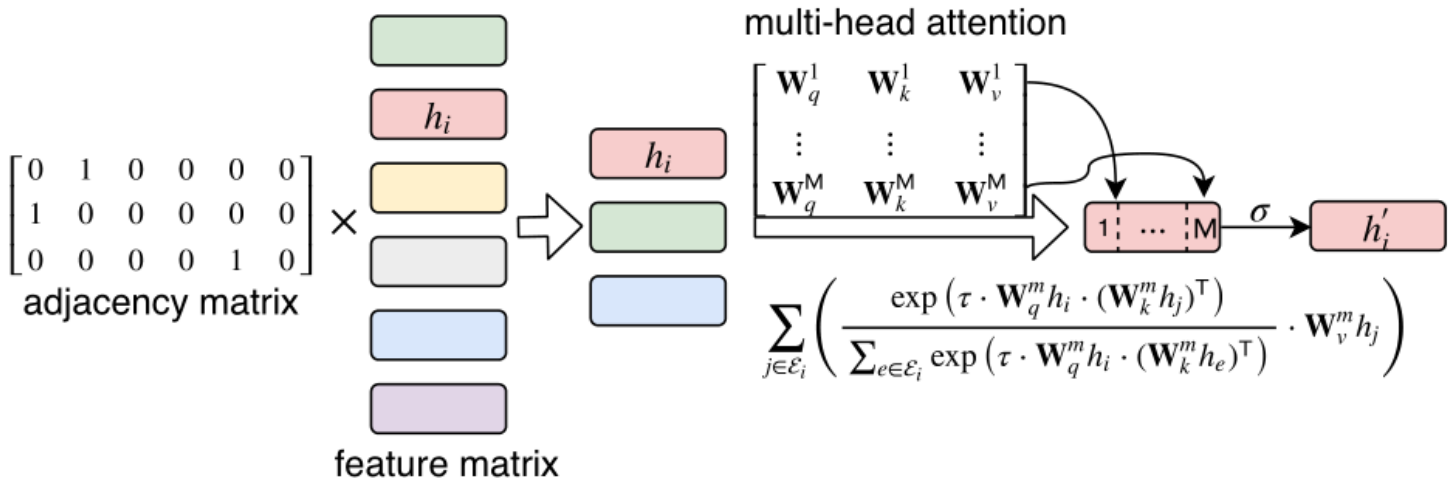


DGN(Graph convolutional RL)

- Convolutional layer module
  - Use multi-head dot-product attention as the convolutional kernel to compute interactions between entities in the local region(node itself and K neighbors)
  - Multi-head attention makes the kernel independent from the order of input feature vectors and training more stable empirically

$$\alpha_{ij}^{m} = \frac{\exp\left(\tau \cdot \mathbf{W}_q^m h_i \cdot (\mathbf{W}_k^m h_j)^{\mathsf{T}}\right)}{\sum_{e \in \mathcal{E}_i} \exp\left(\tau \cdot \mathbf{W}_q^m h_i \cdot (\mathbf{W}_k^m h_e)^{\mathsf{T}}\right)}$$

$$h_i' = \sigma(\text{Concat}[\sum_{j \in \mathcal{E}_i} \alpha_{ij}^m \mathbf{W}_v^m h_j, \forall m \in \text{M}])$$



DGN(Graph convolutional RL)

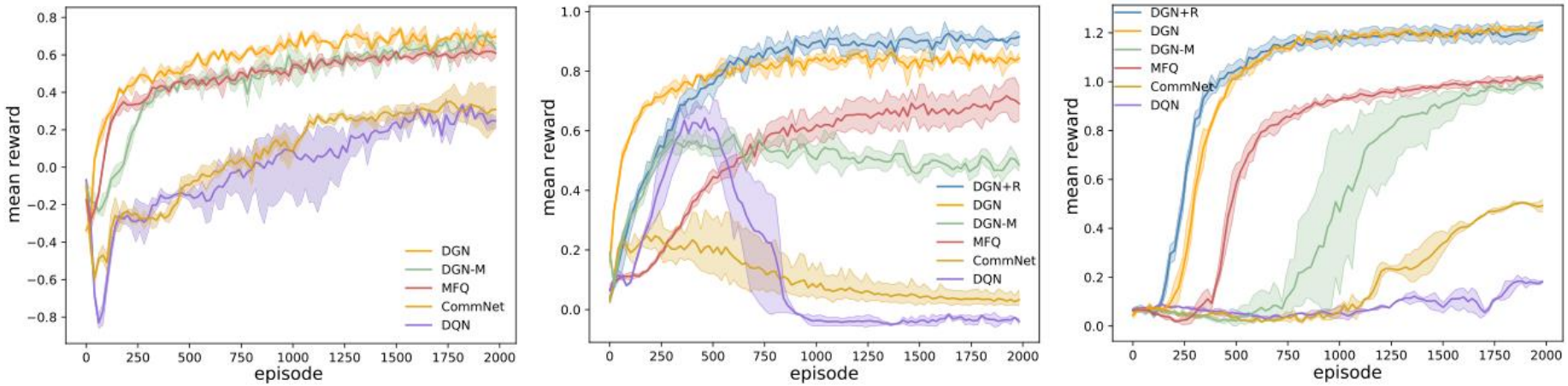
- Temporal Relation Regularization
  - Use attention weight distribution in the next state as the target for the current attention weight distribution
  - Use KL divergence to compute the distance between the attention weight distributions in the two states

$$\mathcal{L}(\theta) = \frac{1}{\mathrm{S}} \sum_{\mathrm{S}} \frac{1}{\mathrm{N}} \sum_{i=1}^{\mathrm{N}} \left(\left(y_i - Q\left(O_i, a_i; \theta\right) \right)^2 + \lambda D_{\mathrm{KL}}(\mathcal{G}^{\kappa}(O_i; \theta) || z_i) \right.$$

  - where $z_i = \mathcal{G}^{\kappa}(O_i'; \theta)$ and $\mathcal{G}^{\kappa}(O_i; \theta)$ denotes attentions weight distribution of relation representations at convolutional layer κ for agent i

Experiments

- DGN substantially outperforms existing methods in several environments
  - Jungle
  - Battle
  - Routing



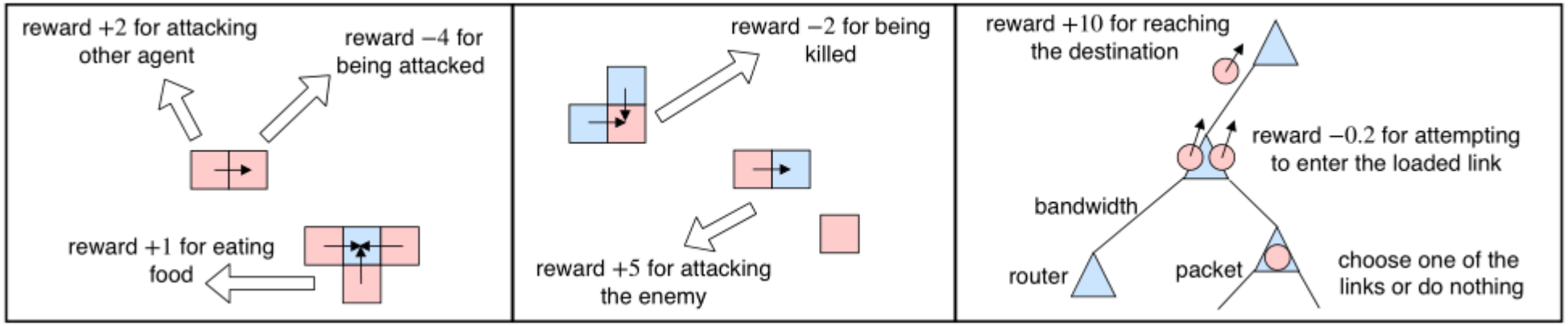


Illustration of experimental scenarios: jungle, battle, and routing

Experiments

- DGN substantially outperforms existing methods in several environments

Table 1. Jungle

(N, L)		DGN	DGN-M	MFQ	CommNet	DQN
(20, 12)	mean reward	**0.70**	0.66	0.62	0.30	0.24
	# attacks	**0.91**	1.89	2.74	5.44	7.35
(50, 12)	mean reward	**0.67**	0.63	0.57	0.27	0.20
	# attacks	**0.91**	1.88	3.13	6.35	9.02

Table 2. Battle

	DGN+R	DGN	DGN-M	MFQ	CommNet	DQN
mean reward	**0.91**	**0.84**	0.50	0.70	0.03	−0.03
# kills	**220**	**208**	121	193	7	2
# deaths	**97**	**101**	84	92	27	74
kill-death ratio	**2.27**	**2.06**	1.44	2.09	0.26	0.03

Table 3. Routing

(N, L)		Floyd	Floyd with BL	DGN+R	DGN	DGN-M	MFQ	CommNet	DQN
(20, 20)	mean reward			**1.23**	**1.21**	0.99	1.02	0.49	0.18
	delay	6.3	8.7	**8.0**	**8.1**	9.8	9.4	18.6	46.7
	throughput	3.17	2.30	**2.50**	**2.47**	2.04	2.13	1.08	0.43
(40, 20)	mean reward			**0.86**	**0.83**	0.70	0.78	0.39	0.12
	delay	6.3	13.7	**9.8**	**10.0**	12.7	11.8	23.5	83.6
	throughput	6.34	2.91	**4.08**	**4.00**	3.15	3.39	1.70	0.49
(40, 20) retrained	mean reward			**0.94**	**0.90**	0.78	0.76	0.35	0.05
	delay	6.3	13.7	**10.2**	**10.5**	12.2	12.8	21.2	112.2
	throughput	6.34	2.91	**3.92**	**3.81**	3.27	3.12	1.86	0.35

Code Implementation

- https://github.com/PKU-AI-Edge/DGN
- 직접 돌려보려 했지만 tensorflow 2.1.1 version이 필요 – 보유한 서버의 tensorflow 버전과 충돌이 일어날 가능성이 있음
- 논문 실험에서의 3가지 environment에 대한 각각의 Code 분석을 통해 알고리즘 설명

Code Implementation – Jungle(multi_jungle.py)

- multi-agent RL의 environment로서 magent 모듈을 사용
- keras를 기본으로 tensorflow의 network(FCN, CNN)를 구현

```
import os, sys, time
import numpy as np
import magent
from magent.model import BaseModel
from magent.builtin.tf_model import DeepQNetwork
from keras import backend as K
from keras.optimizers import Adam
import tensorflow as tf
import random
from ReplayBuffer_v2 import ReplayBuffer
from keras.layers import Dense, Dropout, Conv2D, Input, Lambda, Flatten, TimeDistributed, merge
from keras.layers import Add, Reshape, MaxPooling2D, Concatenate, Embedding, RepeatVector
from keras.models import Model
from keras.layers.core import Activation
from keras.utils import np_utils,to_categorical
from keras.engine.topology import Layer
from keras.callbacks import TensorBoard
from magent.builtin.tf_model import DeepQNetwork
import keras.backend.tensorflow_backend as KTF
config = tf.ConfigProto()
config.gpu_options.allow_growth=True
session = tf.Session(config=config)
KTF.set_session(session)
```

Code Implementation – Jungle(multi_jungle.py)

- Adjacency는 RL의 state를 입력 받아 자기자신과 K nearest neighbor를 나타내는 one hot representation matrix를 output한다. 여기선 agent의 개수가 20이고 K=3으로 distance를 구한 후 sort시켜 K+1 만큼만 가져온다
- MLP의 경우 Keras를 이용하여 2 hidden layer를 가진 FCN을 return 함

```
def Adjacency(state):
    adj = []
    dis = []
    for j in range(20):
        dis.append([state[j][-2],state[j][-1],j])
    for j in range(20):
        f = []
        for r in range(len(dis)):
            f.append([(dis[r][0]-dis[j][0])**2+(dis[r][1]-dis[j][1])**2,r])
        f.sort(key=lambda x:x[0])
        y = []
        for r in range(4):
            y.append(f[r][1])
        y = to_categorical(y,num_classes=20)
        adj.append(y)
    return adj

def observation(state1,state2):
    state = []
    for j in range(20):
        state.append(np.hstack(((state1[j][0:11,0:11,1]-state1[j][0:11,0:11,4]).flatten(),state2[j][-1:-3:-1])))
    return state

def MLP():
    In_0 = Input(shape=[123])

    h = Dense(512, activation='relu',kernel_initializer='random_normal')(In_0)
    h = Dense(128, activation='relu',kernel_initializer='random_normal')(h)

    h = Reshape((1,128))(h)

    model = Model(input=In_0,output=h)
    return model
```

Code Implementation – Jungle(multi_jungle.py)

- Multi-head attention을 실행하는 model을 return함

$$\alpha_{ij}^{m} = \frac{\exp\left(\tau \cdot \mathbf{W}_q^m h_i \cdot (\mathbf{W}_k^m h_j)^{\mathsf{T}}\right)}{\sum_{e \in \mathcal{E}_i} \exp\left(\tau \cdot \mathbf{W}_q^m h_i \cdot (\mathbf{W}_k^m h_e)^{\mathsf{T}}\right)}$$

$$h_i^{'} = \sigma\left(\text{Concat}\left[\sum_{j \in \mathcal{E}_i} \alpha_{ij}^{m} \mathbf{W}_v^m h_j, \forall m \in \mathsf{M}\right]\right)$$

```
def MultiHeadsAttModel(l=2, d=128, dv=16, dout=128, nv = 8 ):

    v1 = Input(shape = (l, d))
    q1 = Input(shape = (l, d))
    k1 = Input(shape = (l, d))
    ve = Input(shape = (l, 1))

    v2 = Dense(dv*nv, activation = "relu",kernel_initializer='random_normal')(v1)
    q2 = Dense(dv*nv, activation = "relu",kernel_initializer='random_normal')(q1)
    k2 = Dense(dv*nv, activation = "relu",kernel_initializer='random_normal')(k1)

    v = Reshape((l, nv, dv))(v2)
    q = Reshape((l, nv, dv))(q2)
    k = Reshape((l, nv, dv))(k2)
    v = Lambda(lambda x: K.permute_dimensions(x, (0,2,1,3)))(v)
    k = Lambda(lambda x: K.permute_dimensions(x, (0,2,1,3)))(k)
    q = Lambda(lambda x: K.permute_dimensions(x, (0,2,1,3)))(q)

    att = Lambda(lambda x: K.batch_dot(x[0],x[1] ,axes=[3,3]) / np.sqrt(dv))([q,k])# l, nv, nv
    att = Lambda(lambda x: K.softmax(x))(att)
    out = Lambda(lambda x: K.batch_dot(x[0], x[1],axes=[3,2]))([att, v])
    out = Lambda(lambda x: K.permute_dimensions(x, (0,2,1,3)))(out)

    out = Reshape((l, dv*nv))(out)

    T = Lambda(lambda x: K.batch_dot(x[0],x[1]))([ve,out])

    out = Dense(dout, activation = "relu",kernel_initializer='random_normal')(T)
    model = Model(inputs=[q1,k1,v1,ve], outputs=out)
    return model
```

Code Implementation – Jungle(multi_jungle.py)

- Q_Net은 앞의 feature들을 concat하여 state에 대한 action들의 Q value들을 output하는 model을 return 한다
- map size, replay buffer size, gamma, agent 수 등 hyperparameter를 지정
- environment는 Jungle의 Food가 있는 eat GridWorld를 만듬

```
def Q_Net(action_dim):
    I1 = Input(shape = (1, 128))
    I2 = Input(shape = (1, 128))
    I3 = Input(shape = (1, 128))

    h1 = Flatten()(I1)
    h2 = Flatten()(I2)
    h3 = Flatten()(I3)

    h = Concatenate()([h1,h2,h3])
    V = Dense(action_dim,kernel_initializer='random_normal')(h)

    model = Model(input=[I1,I2,I3],output=V)
    return model

path="data/battle_model"
map_size=100
capacity = 200000
batch_size = 256
totalTime = 0
TAU = 0.01
LRA = 0.0001
param = None
alpha = 0.6
GAMMA = 0.96
n_episode = 100000
max_steps = 120
episode_before_train = 200
n_agent=20
magent.utility.init_logger("eat")
env = magent.GridWorld("eat", map_size=30)
env.set_render_dir("build/render")
handles = env.get_handles()
sess = tf.Session()
K.set_session(sess)
n = len(handles)
n_actions=env.get_action_space(handles[0])[0]
i_episode=0
buff=ReplayBuffer(capacity)
l=40
```

Code Implementation – Jungle(multi_jungle.py)

- Observation encoder(cnn)와 2층 convolutional layer(m1, m2), Q network(q_net)을 만듬
- In(Input)으로 부터 cnn을 통하여 feature를 얻음
- feature를 attention에 입력하여 relation을 구함, 2층 convolutional layer로 relation을 2개 얻음
- feature와 relation을 모두 Q network에 input하여 Q value를 구함

```
cnn = MLP()
m1 = MultiHeadsAttModel(l=4)
m2 = MultiHeadsAttModel(l=4)
q_net = Q_Net(action_dim = 9)
vec = np.zeros((1,4))
vec[0][0] = 1

In= []
for j in range(n_agent):
    In.append(Input(shape=[123]))
    In.append(Input(shape=(4,20)))
In.append(Input(shape=(1,4)))
feature = []
for j in range(n_agent):
    feature.append(cnn(In[j*2]))

feature_ = Concatenate(axis=1)(feature)

relation1 = []
for j in range(n_agent):
    T = Lambda(lambda x: K.batch_dot(x[0],x[1]))([In[j*2+1],feature_])
    relation1.append(m1([T,T,T,In[40]]))

relation1_ = Concatenate(axis=1)(relation1)

relation2 = []
for j in range(n_agent):
    T = Lambda(lambda x: K.batch_dot(x[0],x[1]))([In[j*2+1],relation1_])
    relation2.append(m2([T,T,T,In[40]]))

V = []
for j in range(n_agent):
    V.append(q_net([feature[j],relation1[j],relation2[j]]))

model = Model(input=In,output=V)
model.compile(optimizer=Adam(lr = 0.0001), loss='mse')
model.summary()
```

Code Implementation – Jungle(multi_jungle.py)

- episode의 max step만큼 무한 루프를 돌림
- adjacency matrix를 뽑고 model을 통해 Q value를 얻음
- ε-greedy를 통해 action을 고르고 실행하여 next observation과 reward를 관찰
- episode가 끝난 후 전체 agent가 다른 agent로부터 받은 damage, reward의 합인 score, loss값을 측정

```
while steps<max_steps:
    steps+=1
    i=0
    obs[i] = env.get_observation(handles[i])
    adj = Adjacency(obs[i][1])
    flat_ob = observation(obs[i][0],obs[i][1])
    ob=[]
    for j in range(n_agent):
        ob.append(np.asarray([flat_ob[j]]))
        ob.append(np.asarray([adj[j]]))
    ob.append(np.asarray([vec]))
    acts = model.predict(ob)
    action[i]=np.zeros(n_agent,dtype = np.int32)
    for j in range(n_agent):
        if np.random.rand()<alpha:
            action[i][j]=random.randrange(n_actions)
        else:
            action[i][j]=np.argmax(acts[j])
    env.set_action(handles[i], action[i])
    done = env.step()

    next_obs = env.get_observation(handles[0])
    flat_next_obs = observation(next_obs[0],next_obs[1])
    rewards = env.get_reward(handles[0])
    score += sum(rewards)
    if steps%3 ==0:
        buff.add(flat_ob, action[0], flat_next_obs, rewards, done, adj)

    if (i_episode-1) % 10 ==0:
        env.render()
    if max_steps == steps:
        damage = 0
        for j_ in range(n_agent):
            damage = damage + 400- obs[0][0][j_][5][5][2]*400
        damage = damage/20
        print(damage,end='\t')
        print(score/300,end='\t')
        #f.write(str(dead[i])+'\t'+str(score[i]/300)+'\t')
        #f.write(str(loss/100)+'\n')
        print(loss/100,end='\n')
        f.write(str(damage)+'\t'+str(score/300)+'\t'+str(loss/100)+'\n')
    env.clear_dead()
```

Code Implementation – Jungle(multi_jungle.py)

- replay buffer에서 batch size만큼의 history data를 뽑음
- Q value와 target Q value를 통해 model의 train을 진행
- 이후 일정 episode마다 target Q network를 main Q network의 weight와의 moving average로 update
- Battle의 경우 enemy 역할의 agent들을 미리 DQN으로 학습된 model을 load하여 사용

```
tf_model = DeepQNetwork(env, handles[1], 'trusty-battle-game-1', use_conv=True)
tf_model.load("data/battle_model", 0, 'trusty-battle-game-1')
```

```
############training###########
batch = buff.getBatch(10)
states,actions,rewards,new_states,dones,adj=[],[],[],[],[],[]
for i_ in  range(n_agent*2+1):
    states.append([])
    new_states.append([])
for e in batch:
    for j in range(n_agent):
        states[j*2].append(e[0][j])
        states[j*2+1].append(e[5][j])
        new_states[j*2].append(e[2][j])
        new_states[j*2+1].append(e[5][j])
    states[40].append(vec)
    new_states[40].append(vec)
    actions.append(e[1])
    rewards.append(e[3])
    dones.append(e[4])

actions = np.asarray(actions)
rewards = np.asarray(rewards)
dones = np.asarray(dones)

for i_ in  range(n_agent*2+1):
    states[i_]=np.asarray(states[i_])
    new_states[i_]=np.asarray(new_states[i_])

q_values = model.predict(states)
target_q_values = model_t.predict(new_states)

for k in range(len(batch)):
    if dones[k]:
        for j in range(n_agent):
            q_values[j][k][actions[k][j]] = rewards[k][j]
    else:
        for j in range(n_agent):
            q_values[j][k][actions[k][j]] =rewards[k][j] + GAMMA*np.max(target_q_values[j][k])

history=model.fit(states, q_values,epochs=1,batch_size=10,verbose=0)
his=0
for (k,v) in history.history.items():
    his+=v[0]
loss+=(his/20)
```