Artificial Intelligence and Mobility Lab

다중 에이전트 강화학습 이론 및 응용

Multi-Agent Deep Reinforcement Theory and its Application

Won Joon Yun
Korea University, School of Electrical Engineering
Artificial Intelligence and Mobility Laboratory

Now, we have curiosity about…

Q1. Should we wait for the scenario terminated?
Trajectory(Dataset): $\tau = \{s_0, a_0, r_0, s_1, a_1, \ldots, s_T\}$

A1. **No, I will introduce *A2C*. It will make objective function optimized FASTER.**

Q2. How can I maximize objective function efficiently?
Objective Function: $J(\theta) = E_\tau[\sum_{t=0}^{T} \gamma^t \cdot r(s_t, a_t)]$

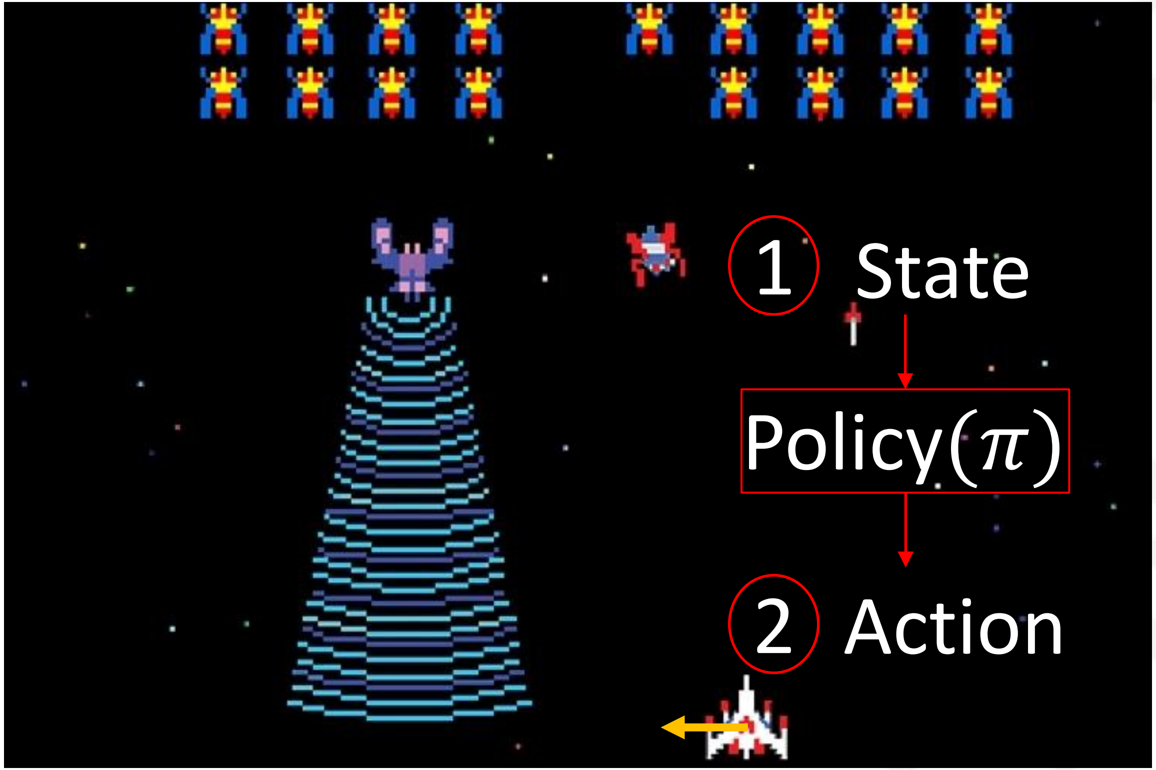
A2. **I will introduce *PPO* and *DDPG*. If you use it, you can maximize the objective function with BETTER PERFORMANCE.**

Q3.What about design DQN?

A3. **I will introduce *CommNet* and *G2ANet*.**

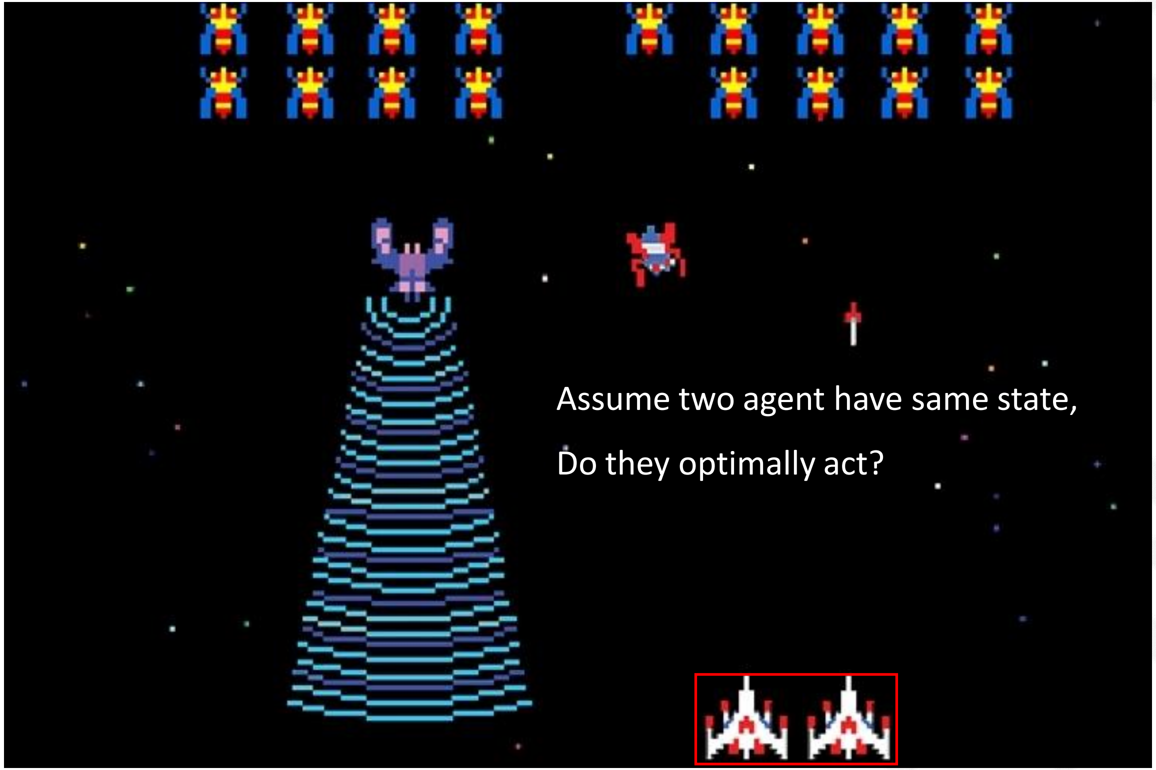
Q4. Any new idea?

A4. **I will introduce Value Decomposition Network.**

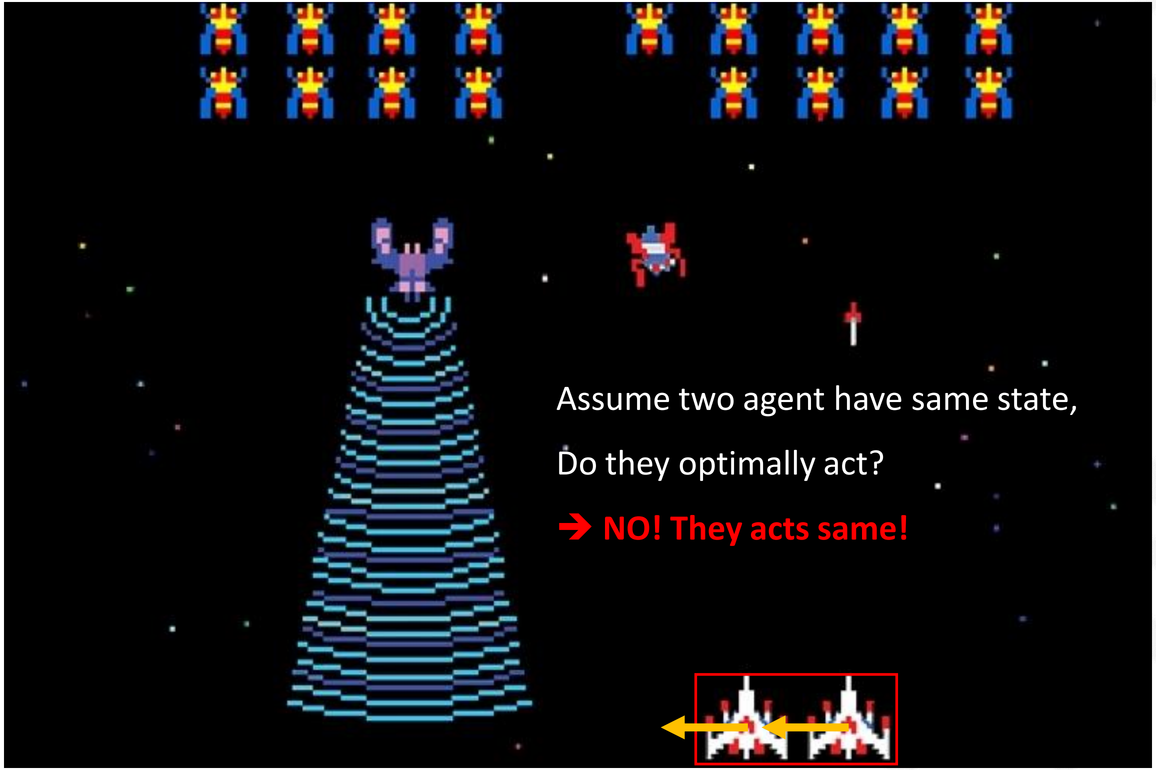


1 State
Policy(π)
2 Action

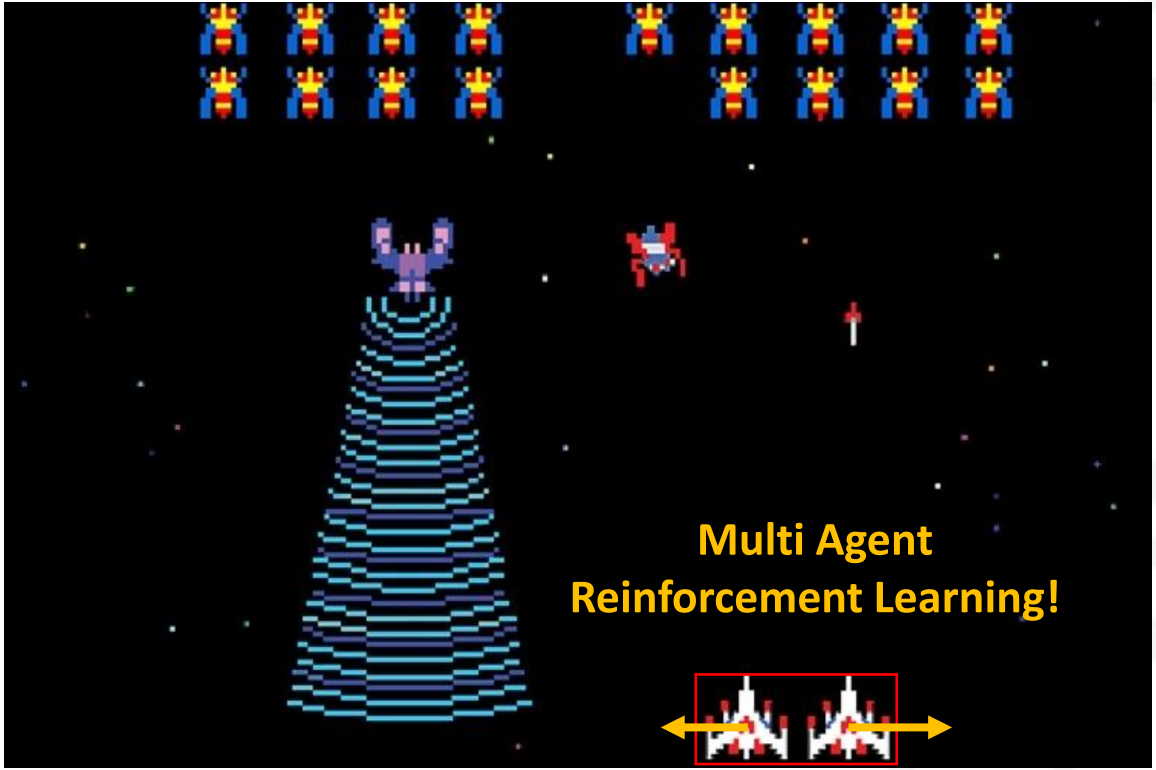
What if there exists more than one agent?



What if there exists more than one agent?

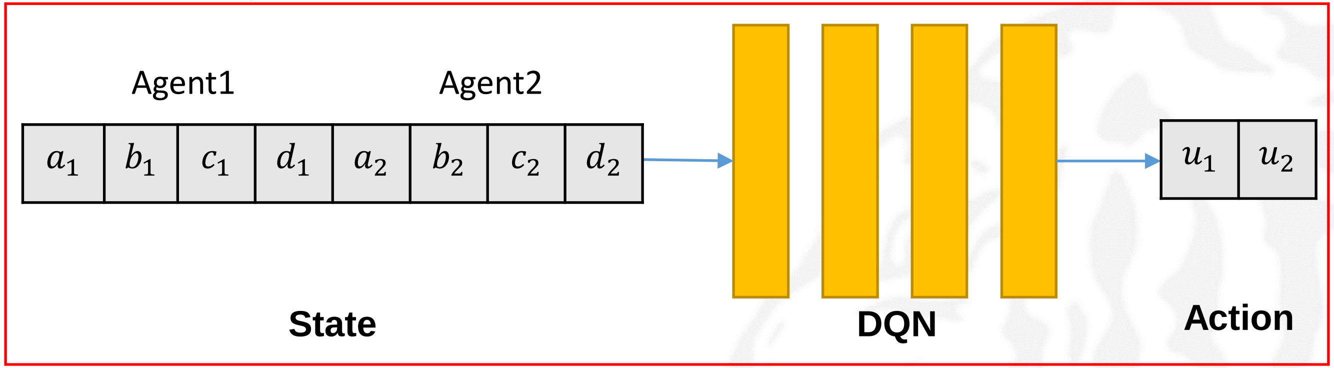


What we are going to learn is …

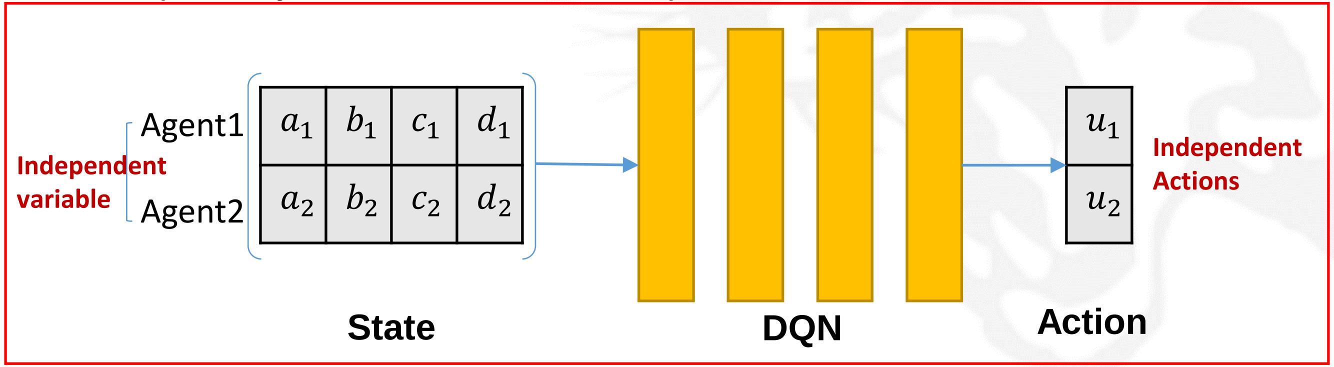


With Previous method.

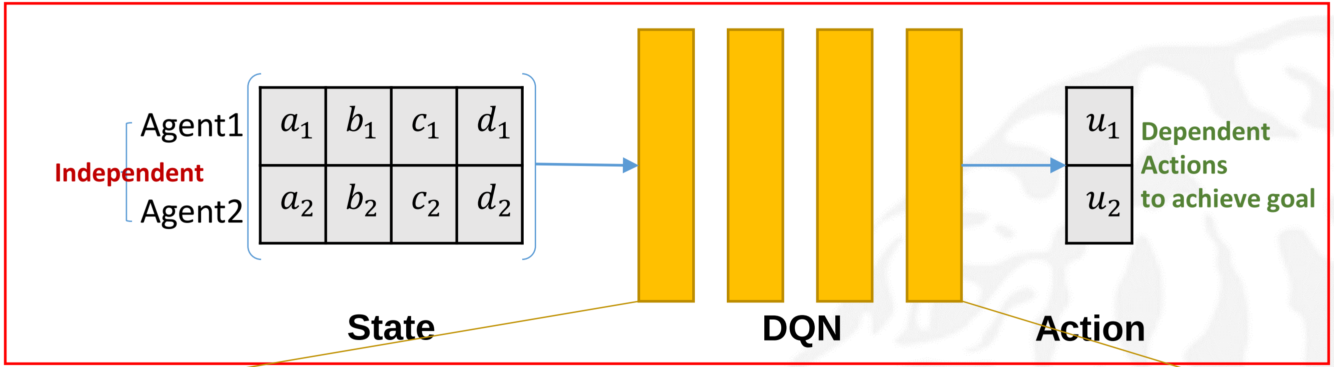
With DDPG (Fully Observable Environment) → **Hard to Converge**

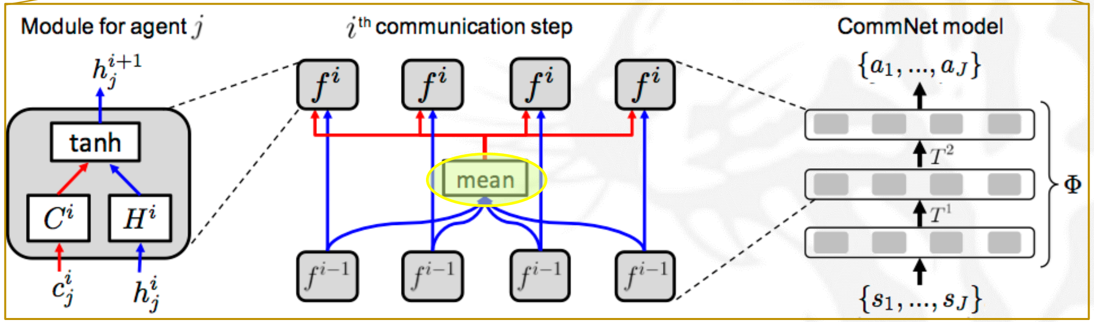


With DQN(Partially Observable Environment)



DQN-based CommNet





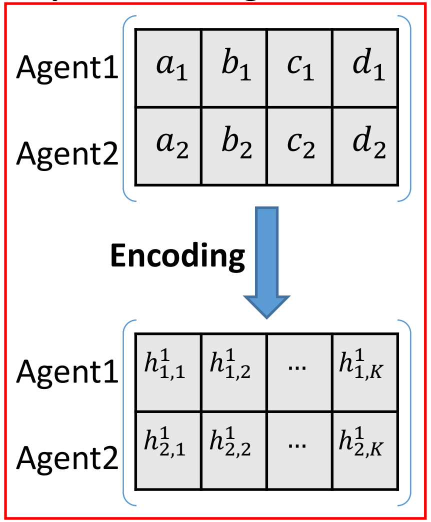
h_j^i : j-th agent's hidden state variable in i-th layer

c_j^i : j-th agent's communitive state variable in i-th layer

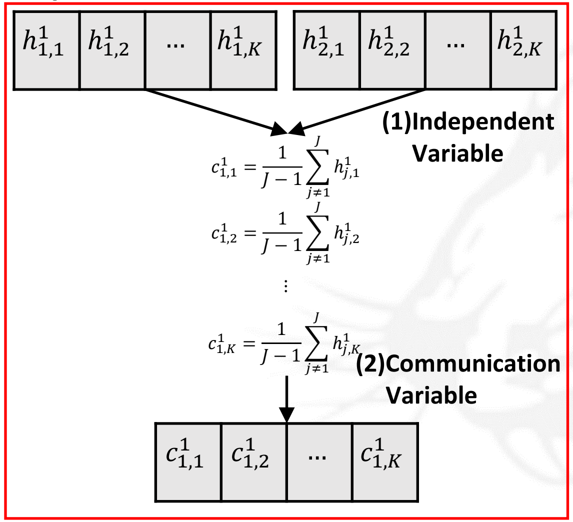
$$h_j^{i+1} = f^i(h_j^i, c_j^i)$$

Artificial Intelligence and Mobility Lab

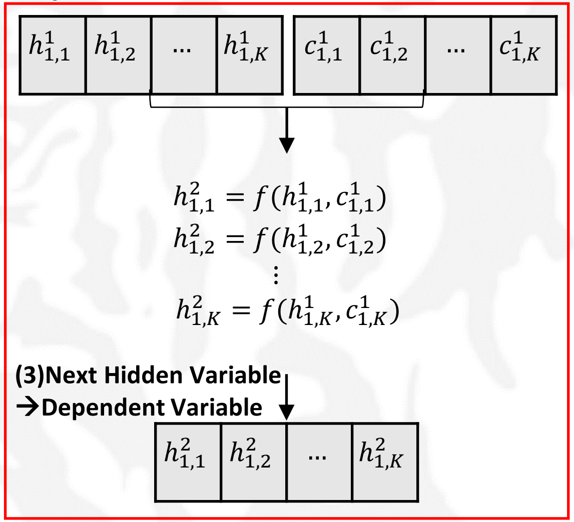
Step#1. Encoding



Step#2-1. Communication Variable



Step#2-2. Activation Function

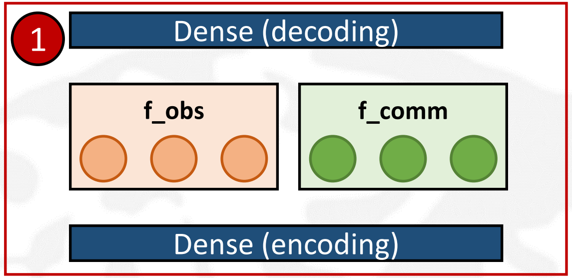


CommNet Architecture

Artificial Intelligence and Mobility Lab

1

```
class CommNet(nn.Module):
    def __init__(self, input_shape):
        super(CommNet, self).__init__()
        self.encoding = nn.Linear(input_shape, rnn_dim)
        self.f_obs = nn.GRUCell(rnn_dim, rnn_dim)
        self.f_comm = nn.GRUCell(rnn_dim, rnn_dim)
        self.decoding = nn.Linear(rnn_dim, rnn_dim)
    def forward(self, obs, hidden_state):
        obs_encoding = torch.sigmoid(self.encoding(obs))
        h_in = hidden_state.reshape(-1,  rnn_dim)
        h_out = self.f_obs(obs_encoding, h_in)
        h = h.reshape(-1, n_agents, rnn_dim)
        c = h.reshape(-1, 1, n_agents*rnn_dim)
        c = c.repeat(1, n_agents, 1)
        mask = (1 - torch.eye(n_agents))
        mask = mask.view(-1, 1).repeat(1, rnn_dim).view(n_agents, -1)
        c = c * mask.unsqueeze(0)
        c = c.reshape(-1,  n_agents, n_agents, rnn_dim)
        c = c.mean(dim=-2)
        h = h.reshape(-1, rnn_dim)
        c = c.reshape(-1, rnn_dim)
        h = self.f_comm(c, h)
        weights = self.decoding(h)
        return weights, h_out
```

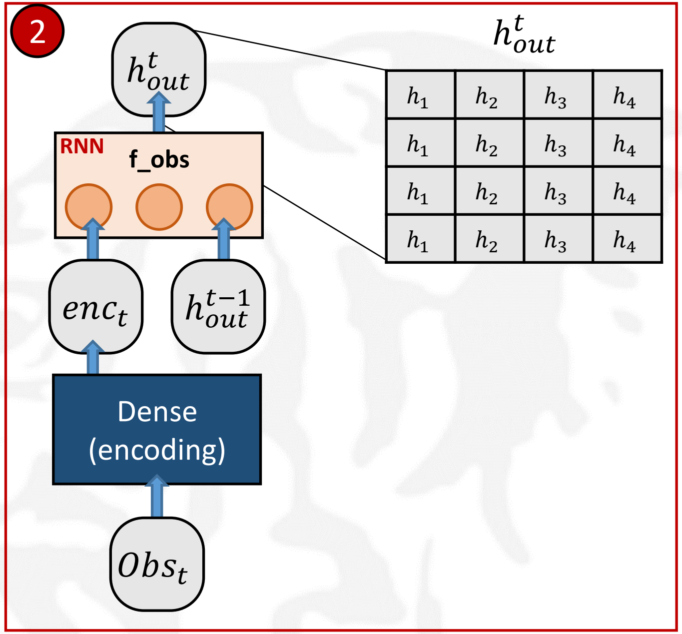


CommNet Architecture


```
class CommNet(nn.Module):
    def __init__(self, input_shape):
        super(CommNet, self).__init__()
        self.encoding = nn.Linear(input_shape, rnn_dim)
        self.f_obs = nn.GRUCell(rnn_dim, rnn_dim)
        self.f_comm = nn.GRUCell(rnn_dim, rnn_dim)
        self.decoding = nn.Linear(rnn_dim, rnn_dim)

    def forward(self, obs, hidden_state):
        obs_encoding = torch.sigmoid(self.encoding(obs))
        h_in = hidden_state.reshape(-1,  rnn_dim)
        h_out = self.f_obs(obs_encoding, h_in)
        h = h.reshape(-1, n_agents, rnn_dim)
        c = h.reshape(-1, 1, n_agents*rnn_dim)
        c = c.repeat(1, n_agents, 1)
        mask = (1 - torch.eye(n_agents))
        mask = mask.view(-1, 1).repeat(1, rnn_dim).view(n_agents, -1)
        c = c * mask.unsqueeze(0)
        c = c.reshape(-1,  n_agents, n_agents, rnn_dim)
        c = c.mean(dim=-2)
        h = h.reshape(-1, rnn_dim)
        c = c.reshape(-1, rnn_dim)
        h = self.f_comm(c, h)
        weights = self.decoding(h)
        return weights, h_out
```

2



CommNet Architecture


```python
class CommNet(nn.Module):
    def __init__(self, input_shape):
        super(CommNet, self).__init__()
        self.encoding = nn.Linear(input_shape, rnn_dim)
        self.f_obs = nn.GRUCell(rnn_dim, rnn_dim)
        self.f_comm = nn.GRUCell(rnn_dim, rnn_dim)
        self.decoding = nn.Linear(rnn_dim, rnn_dim)

    def forward(self, obs, hidden_state):
        obs_encoding = torch.sigmoid(self.encoding(obs))
        h_in = hidden_state.reshape(-1,  rnn_dim)
        h_out = self.f_obs(obs_encoding, h_in)
        h = h.reshape(-1, n_agents, rnn_dim)
        c = h.reshape(-1, 1, n_agents*rnn_dim)
        c = c.repeat(1, n_agents, 1)
        mask = (1 - torch.eye(n_agents))
        mask = mask.view(-1, 1).repeat(1, rnn_dim).view(n_agents, -1)
        c = c * mask.unsqueeze(0)
        c = c.reshape(-1,  n_agents, n_agents, rnn_dim)
        c = c.mean(dim=-2)
        h = h.reshape(-1, rnn_dim)
        c = c.reshape(-1, rnn_dim)
        h = self.f_comm(c, h)
        weights = self.decoding(h)
        return weights, h_out
```

3

c

h_1	h_2	h_3	h_4
h_1	h_2	h_3	h_4
h_1	h_2	h_3	h_4
h_1	h_2	h_3	h_4

*

mask

0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	0

3

$$h_2 + h_3 + h_4$$
$$h_1 + h_3 + h_4$$
$$h_1 + h_2 + h_4$$
$$h_1 + h_2 + h_3$$

$$c_1 = (h_2 + h_3 + h_4)/(J-1)$$
$$c_2 = (h_1 + h_3 + h_4)/(J-1)$$
$$c_3 = (h_1 + h_2 + h_4)\,/(J-1)$$
$$c_4 = (h_1 + h_2 + h_3)\,/(J-1)$$

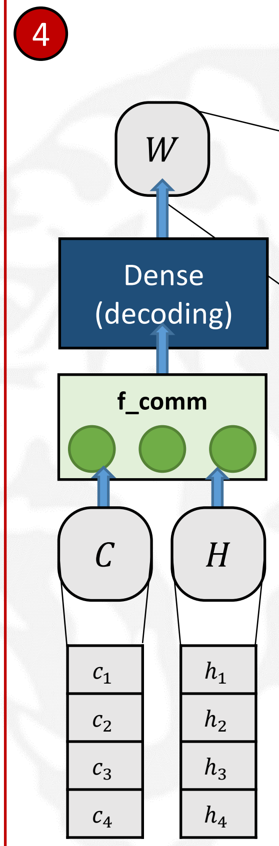
$$c_j^{i+1} = \frac{1}{J-1} \sum_{j' \neq j} h_{j'}^{i+1}.$$

CommNet Architecture


```
class CommNet(nn.Module):
    def __init__(self, input_shape):
        super(CommNet, self).__init__()
        self.encoding = nn.Linear(input_shape, rnn_dim)
        self.f_obs = nn.GRUCell(rnn_dim, rnn_dim)
        self.f_comm = nn.GRUCell(rnn_dim, rnn_dim)
        self.decoding = nn.Linear(rnn_dim, rnn_dim)

    def forward(self, obs, hidden_state):
        obs_encoding = torch.sigmoid(self.encoding(obs))
        h_in = hidden_state.reshape(-1,  rnn_dim)
        h_out = self.f_obs(obs_encoding, h_in)
        h = h.reshape(-1, n_agents, rnn_dim)
        c = h.reshape(-1, 1, n_agents*rnn_dim)
        c = c.repeat(1, n_agents, 1)
        mask = (1 - torch.eye(n_agents))
        mask = mask.view(-1, 1).repeat(1, rnn_dim).view(n_agents, -1)
        c = c * mask.unsqueeze(0)
        c = c.reshape(-1,  n_agents, n_agents, rnn_dim)
        c = c.mean(dim=-2)
        h = h.reshape(-1, rnn_dim)
        c = c.reshape(-1, rnn_dim)
        h = self.f_comm(c, h)
        weights = self.decoding(h)
        return weights, h_out
```

4



$p(a_1)$	$p(a_2)$	$p(a_m)$	$p(a_M)$
$p(a_1)$	$p(a_2)$	$p(a_m)$	$p(a_M)$
$p(a_1)$	$p(a_2)$	$p(a_m)$	$p(a_M)$
$p(a_1)$	$p(a_2)$	$p(a_m)$	$p(a_M)$

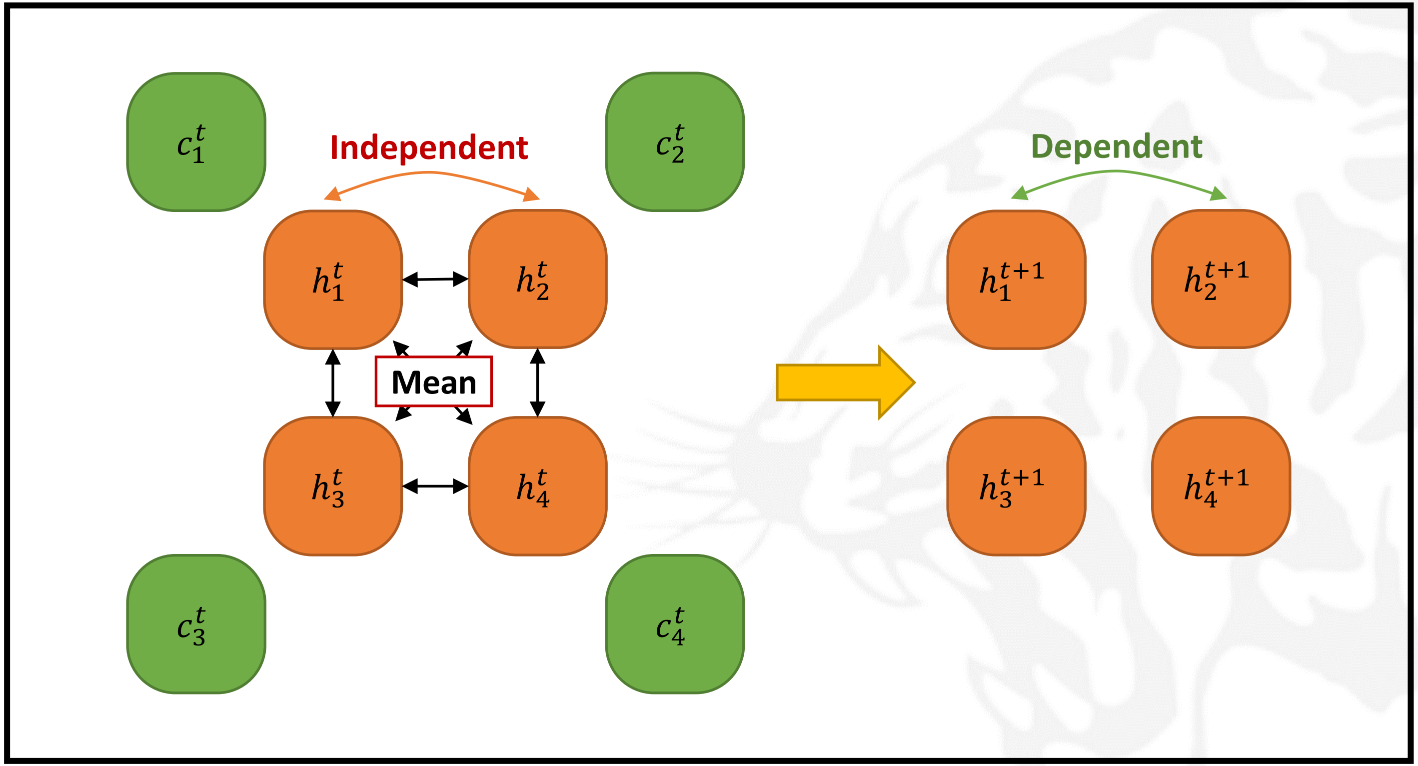
CommNet Performance

Model Φ	Training method: Supervised	Training method: Reinforcement
Independent	0.59	0.59
CommNet	**0.99**	**0.94**

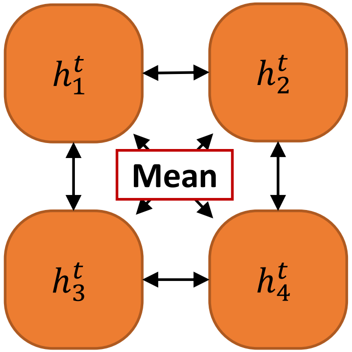
Model Φ	Other game versions: Easy (MLP)	Other game versions: Hard (RNN)
Independent	15.8± 12.5	26.9± 6.0
Discrete comm.	1.1± 2.4	28.2± 5.7
CommNet	**0.3**± **0.1**	22.5± 6.1
CommNet local	-	**21.1**± **3.4**

Model Φ	Module $f()$ type: MLP	RNN	LSTM
Independent	20.6± 14.1	19.5± 4.5	9.4± 5.6
Fully-connected	12.5± 4.4	34.8± 19.7	4.8± 2.4
Discrete comm.	15.8± 9.3	15.2± 2.1	8.4± 3.4
CommNet	**2.2**± **0.6**	**7.6**± **1.4**	**1.6**± **1.0**

Summary of CommNet



CommNet



In Graph Approach.

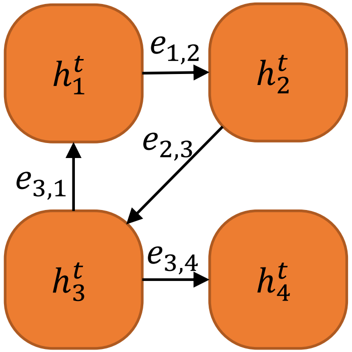
1. Should the agent communicate with all agent?

2. Can we transfer only essential information between agents?

→ *G2ANet will be the solution to the above problem.*

Now we are curious that…

G2ANet



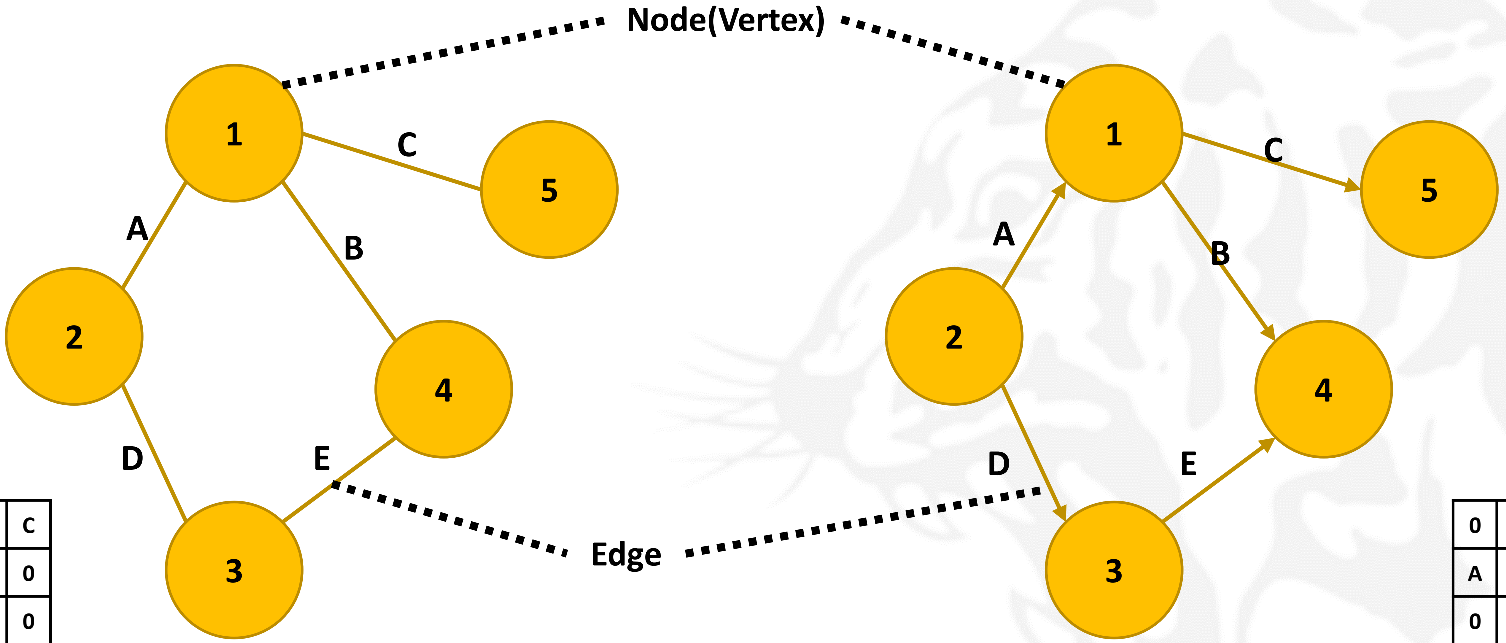
In Graph Approach.

1. Should the agent communicate with all agent?

2. Can we transfer only essential information between agents?

***G2ANet* will be the solution to the above problem.**

- Graph



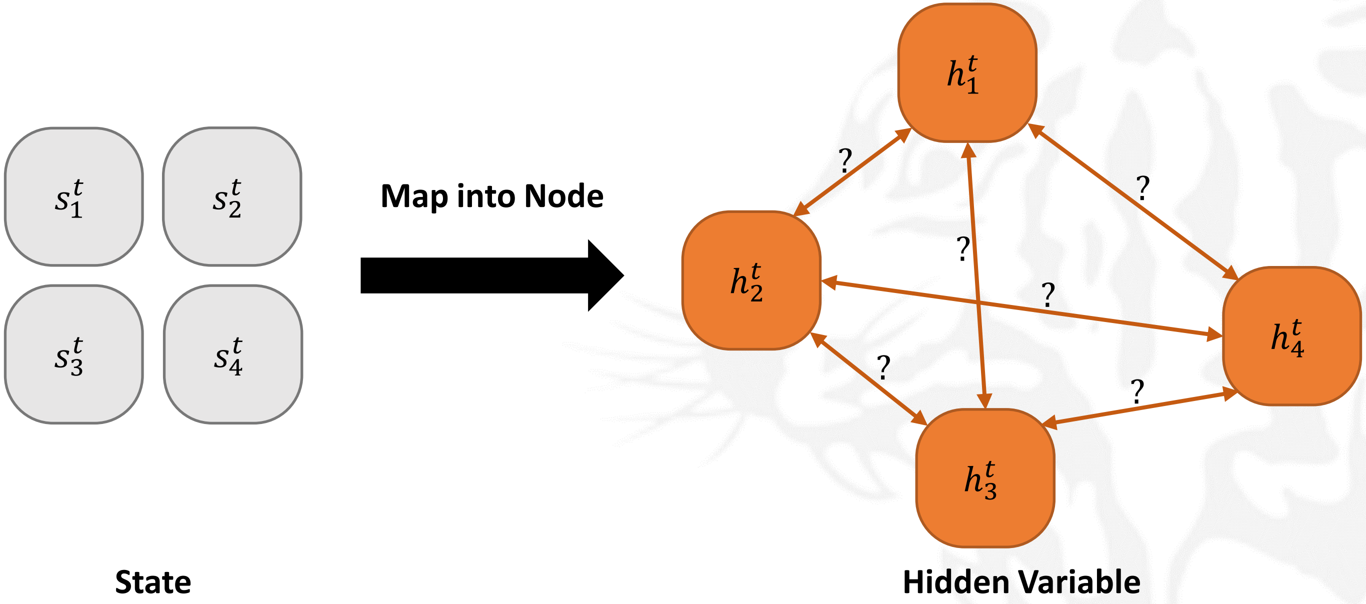
Undirected graph

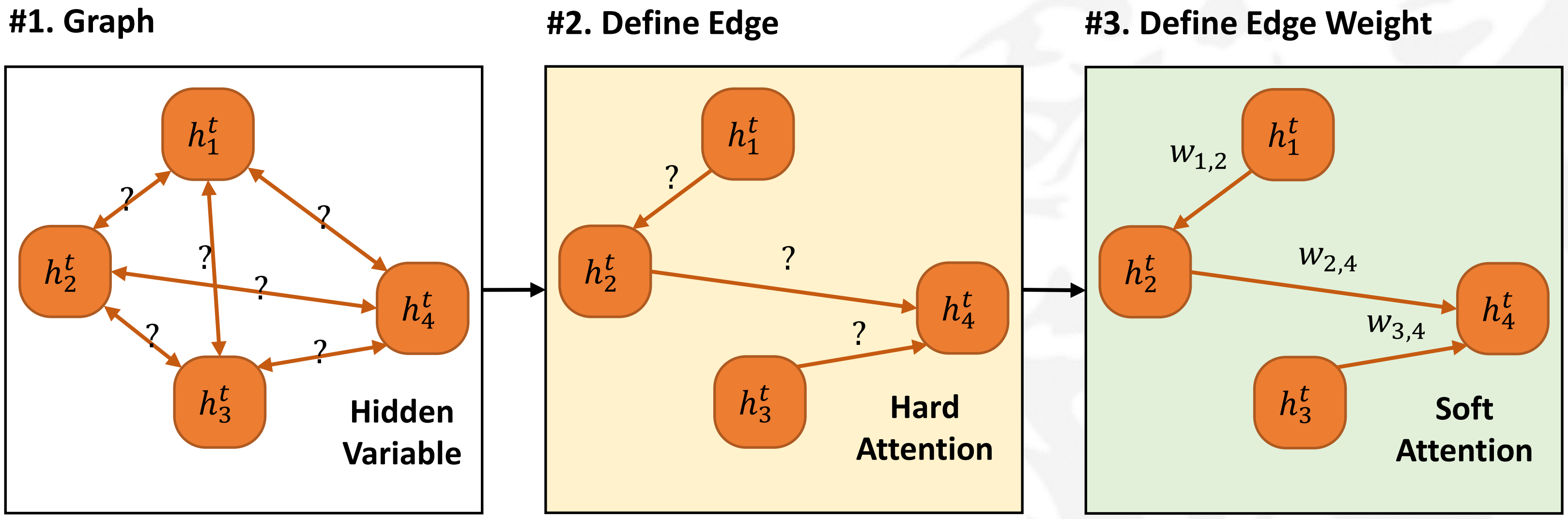
Directed graph

0	A	0	B	C
A	0	D	0	0
0	D	0	E	0
B	0	E	0	0
C	0	0	0	0

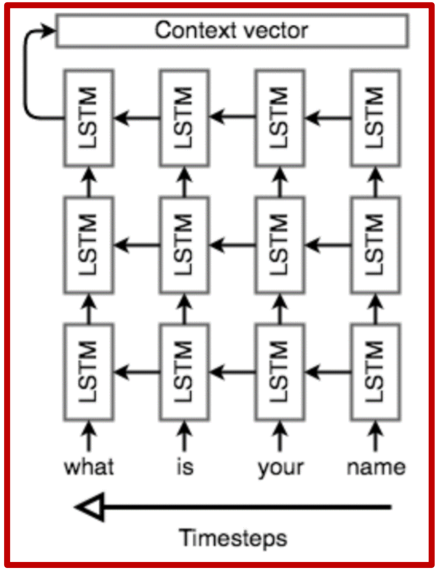
0	0	0	B	C
A	0	D	0	0
0	0	0	E	0
0	0	0	0	0
0	0	0	0	0

- States of agent are mapped into nodes(vertices).




#1. Graph
h_1^t
h_2^t
h_3^t
h_4^t
?
Hidden Variable
#2. Define Edge
h_1^t
h_2^t
h_3^t
h_4^t
?
Hard Attention
#3. Define Edge Weight
h_1^t
h_2^t
h_3^t
h_4^t
$w_{1,2}$
$w_{2,4}$
$w_{3,4}$
Soft Attention

- *Seq2Seq and attention* mechanism is widely used in natural language process(NLP).



- Query(Dictionary)

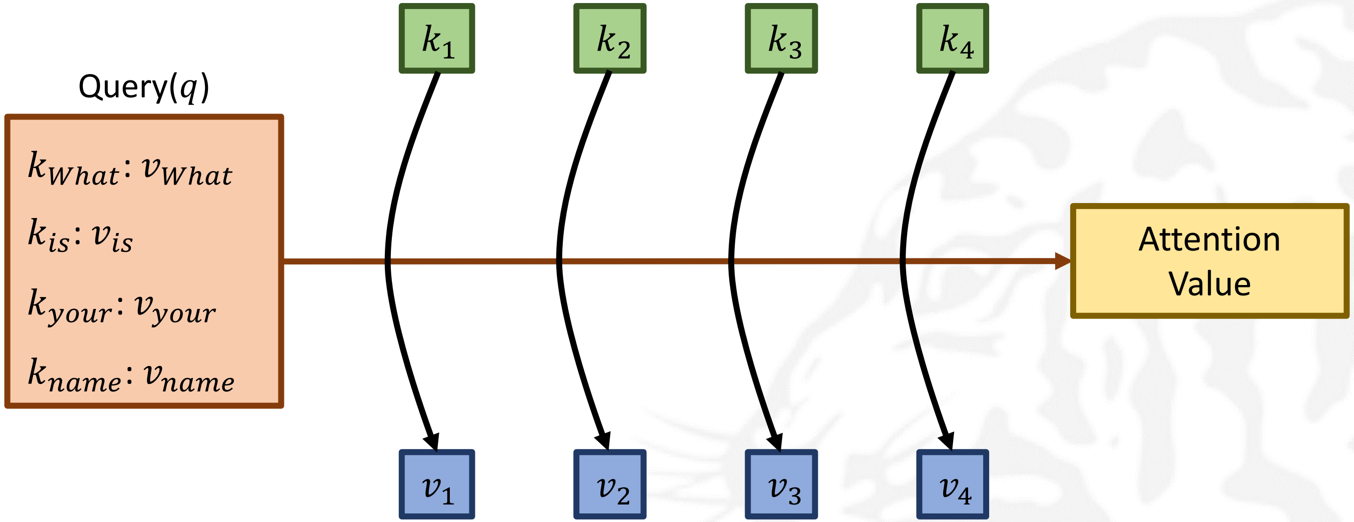
  $\{k_{What}: v_{What}\ ,k_{is}: v_{is}\ ,k_{your}: v_{your}, k_{name}: v_{name}\}$

- Key

  $k_{What}, k_{is}, k_{your}, k_{name}$

- Value

  $v_{What}, v_{is}, v_{your}, v_{name}$

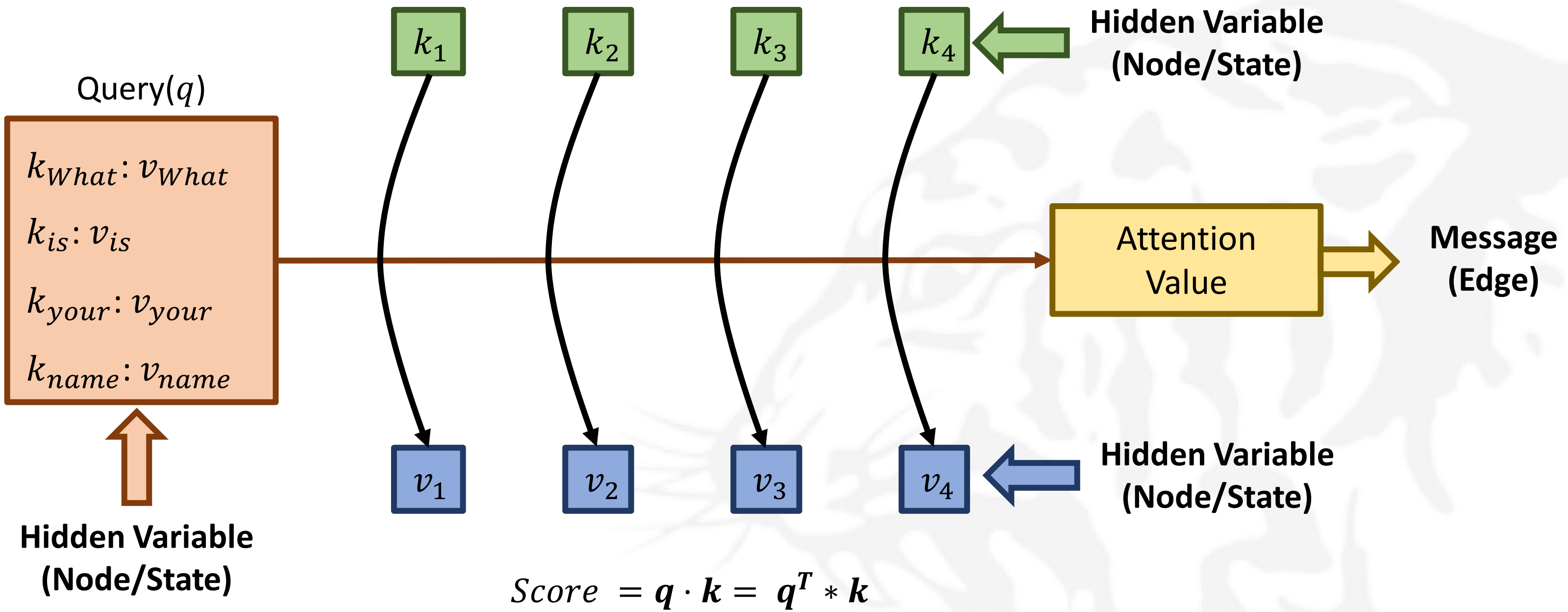


$$Score = \boldsymbol{q} \cdot \boldsymbol{k} = \boldsymbol{q}^{\boldsymbol{T}} * \boldsymbol{k}$$

$$Score_{scaled} = \frac{Score}{\sqrt{n}}$$

$$Attention(\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v}) = Score_{scaled} * \boldsymbol{v}$$

Autoencoder: Hidden Variable(h_t) to Query, Key, Value



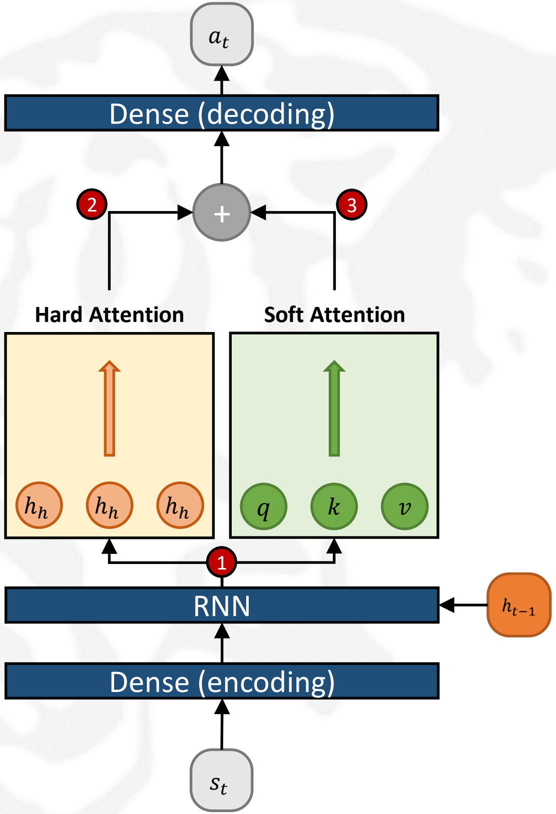
$$Score = \boldsymbol{q} \cdot \boldsymbol{k} = \boldsymbol{q}^T * \boldsymbol{k}$$

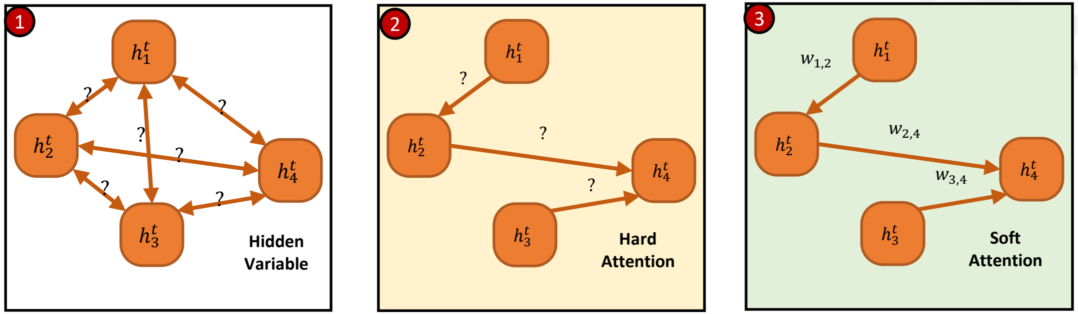
$$Score_{scaled} = \frac{Score}{\sqrt{n}}$$

$$Attention(\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v}) = Score_{scaled} * \boldsymbol{v}$$

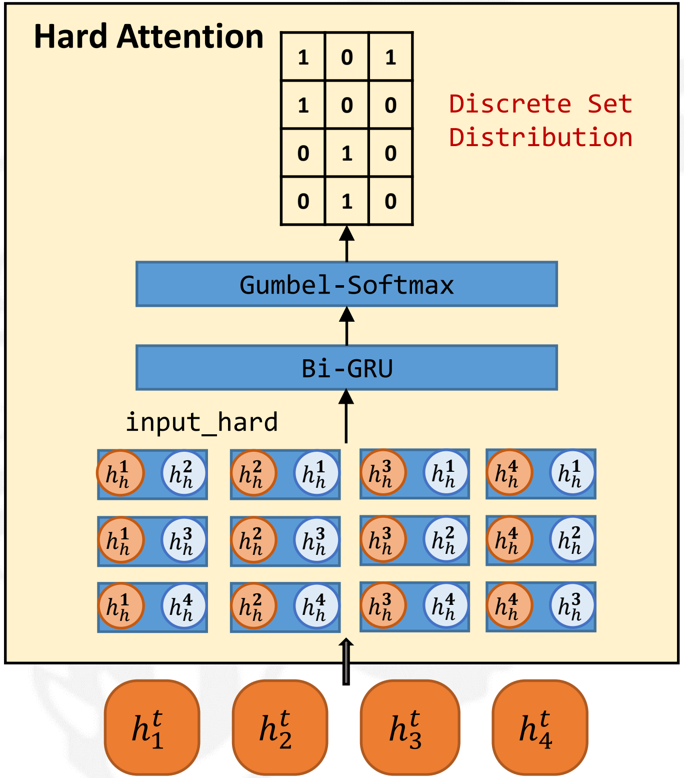
G2ANet Architecture

```python
class G2ANet(nn.Module):
    def __init__(self, input_shape, args):
        super(G2ANet, self).__init__()
        self.encoding = nn.Linear(input_shape, rnn_dim)
        self.h = nn.GRUCell(rnn_dim,  rnn_dim)
        self.hard_bi_GRU = nn.GRU(rnn_dim * 2,  rnn_dim, bidirectional=True)
        self.hard_encoding = nn.Linear(rnn_dim * 2, 2)
        self.q = nn.Linear(rnn_dim,  rnn_dim, bias=False)
        self.k = nn.Linear(rnn_dim,  rnn_dim, bias=False)
        self.v = nn.Linear(rnn_dim,  rnn_dim)
        self.decoding = nn.Linear(rnn_dim+attention_dim, n_actions)
        self.args = args
        self.input_shape = input_shape
```





```
def forward(self, obs, hidden_state):
    obs_encoding = f.relu(self.encoding(obs))
    h_in = hidden_state.reshape(-1,  rnn_dim)
    h_out = self.h(obs_encoding, h_in)
    h = h_out.reshape(-1, n_agents, rnn_dim)
    input_hard = []
    for i in range(n_agents):
        h_i = h[:, i]
        h_hard_i = []
        for j in range(n_agents):
            if j != i:
                h_hard_i.append(torch.cat([h_i, h[:, j]], dim=-1))
                h_hard_i = torch.stack(h_hard_i, dim=0)
                input_hard.append(h_hard_i)
            input_hard = torch.stack(input_hard, dim=-2)
            input_hard = input_hard.view(n_agents - 1, -1,  rnn_dim * 2)
    h_hard = torch.zeros((2 * 1, size, rnn_dim))
    h_hard, _ = self.hard_bi_GRU(input_hard, h_hard)
    h_hard = h_hard.permute(1, 0, 2)
    h_hard = h_hard.reshape(-1, rnn_dim * 2)
    hard_weights = self.hard_encoding(h_hard)
    hard_weights = f.gumbel_softmax(hard_weights, tau=0.01)
    hard_weights = hard_weights[:, 1].view(-1, n_agents, 1, n_agents-1)
    hard_weights = hard_weights.permute(1, 0, 2, 3)
```



Hard Attention

```python
def forward(self, obs, hidden_state):
    obs_encoding = f.relu(self.encoding(obs))
    h_in = hidden_state.reshape(-1,  rnn_dim)
    h_out = self.h(obs_encoding, h_in)
    h = h_out.reshape(-1, n_agents, rnn_dim)
    input_hard = []
    for i in range(n_agents):
        h_i = h[:, i]
        h_hard_i = []
        for j in range(n_agents):
            if j != i:
                h_hard_i.append(torch.cat([h_i, h[:, j]], dim=-1))
                h_hard_i = torch.stack(h_hard_i, dim=0)
                input_hard.append(h_hard_i)
            input_hard = torch.stack(input_hard, dim=-2)
            input_hard = input_hard.view(n_agents - 1, -1,  rnn_dim * 2)
    h_hard = torch.zeros((2 * 1, size, rnn_dim))
    h_hard, _ = self.hard_bi_GRU(input_hard, h_hard)
    h_hard = h_hard.permute(1, 0, 2)
    h_hard = h_hard.reshape(-1, rnn_dim * 2)
    hard_weights = self.hard_encoding(h_hard)
    hard_weights = f.gumbel_softmax(hard_weights, tau=0.01)
    hard_weights = hard_weights[:, 1].view(-1, n_agents, 1, n_agents-1)
    hard_weights = hard_weights.permute(1, 0, 2, 3)
```

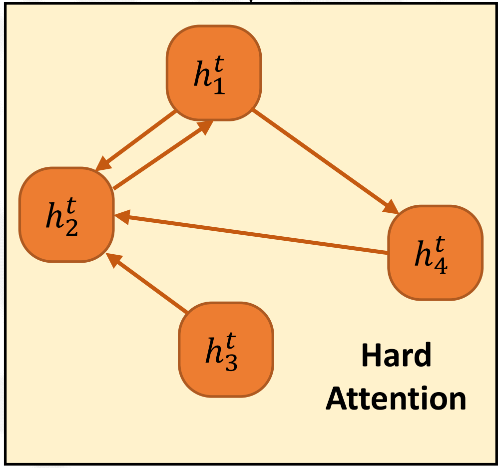
Hard Attention Output

1	0	1
1	0	0
0	1	0
0	1	0

Adjacency Matrix

0	1	0	1
1	0	0	0
0	1	0	0
0	1	0	0

Define Edge



Soft Attention

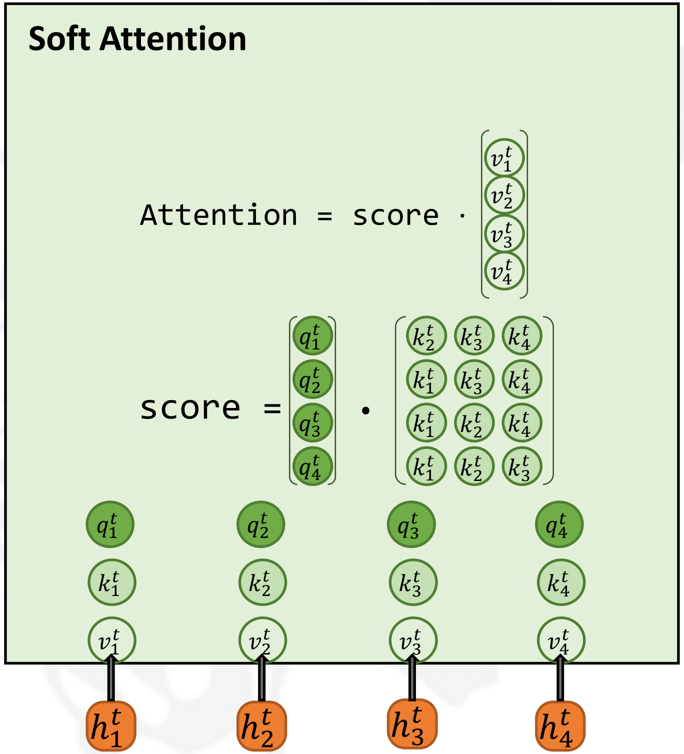

```
def forward(self, obs, hidden_state):
    q = self.q(h_out).reshape(-1, n_agents, attention_dim)
    k = self.k(h_out).reshape(-1, n_agents, attention_dim)
    v = f.relu(self.v(h_out)).reshape(1, n_agents, attention_dim)
    x = []
    for i in range(n_agents):
        q_i = q[:, i].view(-1, 1, attention_dim)
        k_i = [k[:, j] for j in range(n_agents) if j != i]
        v_i = [v[:, j] for j in range(n_agents) if j != i]

        k_i = torch.stack(k_i, dim=0)
        k_i = k_i.permute(1, 2, 0)
        v_i = torch.stack(v_i, dim=0)
        v_i = v_i.permute(1, 2, 0)
        score = torch.matmul(q_i, k_i)
        scaled_score = score / np.sqrt(attention_dim)
        soft_weight = f.softmax(scaled_score, dim=-1)
        x_i = (v_i * soft_weight * hard_weights[i]).sum(dim=-1)
        x.append(x_i)
    x = torch.stack(x, dim=1).reshape(-1, attention_dim)
    final_input = torch.cat([h_out, x], dim=-1)
    output = self.decoding(final_input)
    return output, h_out
```

$$Score = \boldsymbol{q} \cdot \boldsymbol{k} = \boldsymbol{q}^{\boldsymbol{T}} * \boldsymbol{k}$$

$$Score_{scaled} = \frac{Score}{\sqrt{n}}$$

$$Attention(\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v}) = Score_{scaled} * \boldsymbol{v}$$



Soft Attention

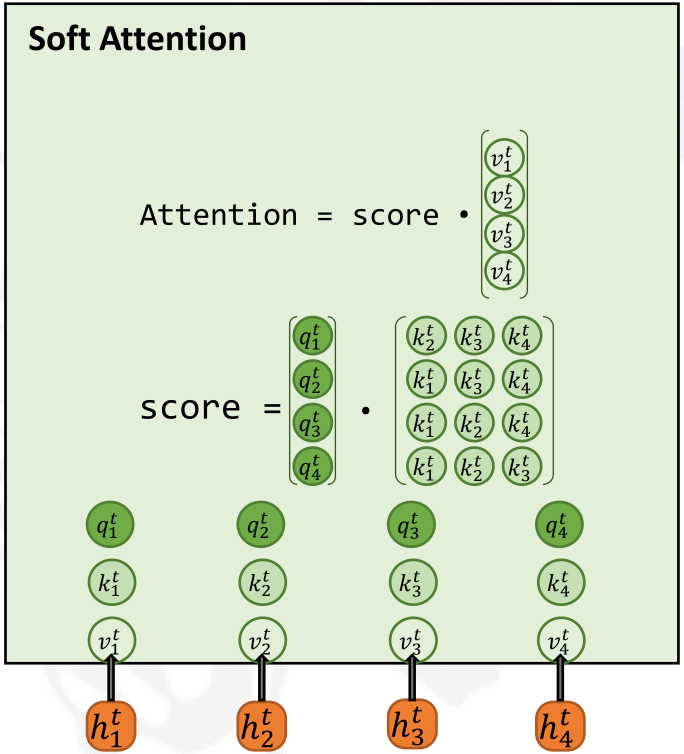
```python
def forward(self, obs, hidden_state):
    q = self.q(h_out).reshape(-1, n_agents, attention_dim)
    k = self.k(h_out).reshape(-1, n_agents, attention_dim)
    v = f.relu(self.v(h_out)).reshape(1, n_agents, attention_dim)
    x = []
    for i in range(n_agents):
        q_i = q[:, i].view(-1, 1, attention_dim)
        k_i = [k[:, j] for j in range(n_agents) if j != i]
        v_i = [v[:, j] for j in range(n_agents) if j != i]

        k_i = torch.stack(k_i, dim=0)
        k_i = k_i.permute(1, 2, 0)
        v_i = torch.stack(v_i, dim=0)
        v_i = v_i.permute(1, 2, 0)
        score = torch.matmul(q_i, k_i)
        scaled_score = score / np.sqrt(attention_dim)
        soft_weight = f.softmax(scaled_score, dim=-1)
        x_i = (v_i * soft_weight * hard_weights[i]).sum(dim=-1)
        x.append(x_i)
    x = torch.stack(x, dim=1).reshape(-1, attention_dim)
    final_input = torch.cat([h_out, x], dim=-1)
    output = self.decoding(final_input)
    return output, h_out
```

$$Score = \boldsymbol{q} \cdot \boldsymbol{k} = \boldsymbol{q}^{\boldsymbol{T}} * \boldsymbol{k}$$

$$Score_{scaled} = \frac{Score}{\sqrt{n}}$$

$$Attention(\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v}) = Score_{scaled} * \boldsymbol{v}$$

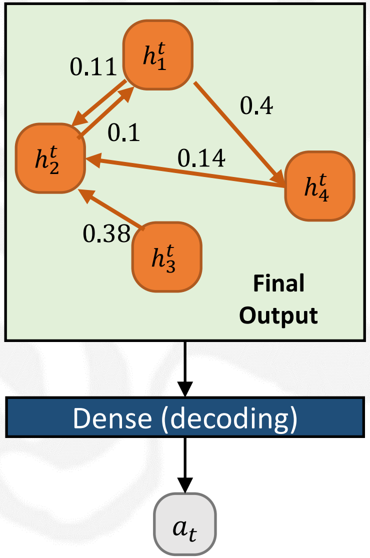


```python
def forward(self, obs, hidden_state):
    q = self.q(h_out).reshape(-1, n_agents, attention_dim)
    k = self.k(h_out).reshape(-1, n_agents, attention_dim)
    v = f.relu(self.v(h_out)).reshape(1, n_agents, attention_dim)
    x = []
    for i in range(n_agents):
        q_i = q[:, i].view(-1, 1, attention_dim)
        k_i = [k[:, j] for j in range(n_agents) if j != i]
        v_i = [v[:, j] for j in range(n_agents) if j != i]

        k_i = torch.stack(k_i, dim=0)
        k_i = k_i.permute(1, 2, 0)
        v_i = torch.stack(v_i, dim=0)
        v_i = v_i.permute(1, 2, 0)
        score = torch.matmul(q_i, k_i)
        scaled_score = score / np.sqrt(attention_dim)
        soft_weight = f.softmax(scaled_score, dim=-1)
        x_i = (v_i * soft_weight * hard_weights[i]).sum(dim=-1)
        x.append(x_i)
    x = torch.stack(x, dim=1).reshape(-1, attention_dim)
    final_input = torch.cat([h_out, x], dim=-1)
    output = self.decoding(final_input)
    return output, h_out
```

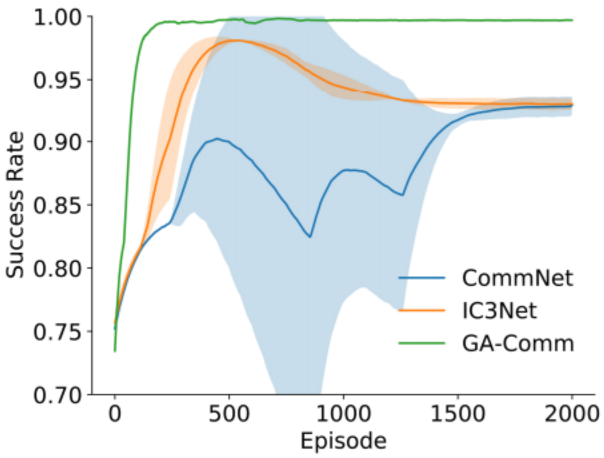
Hard Attention Output (connection) * **Soft Attention Output (message)** = **Final Output (connection & message)**

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} 0.11 & 0.84 & 0.4 \\ 0.1 & 0.18 & 0.72 \\ 0.34 & 0.38 & 0.28 \\ 0.16 & 0.14 & 0.70 \end{bmatrix} = \begin{bmatrix} 0.11 & 0 & 0.4 \\ 0.1 & 0 & 0 \\ 0 & 0.38 & 0 \\ 0 & 0.14 & 0 \end{bmatrix}$$

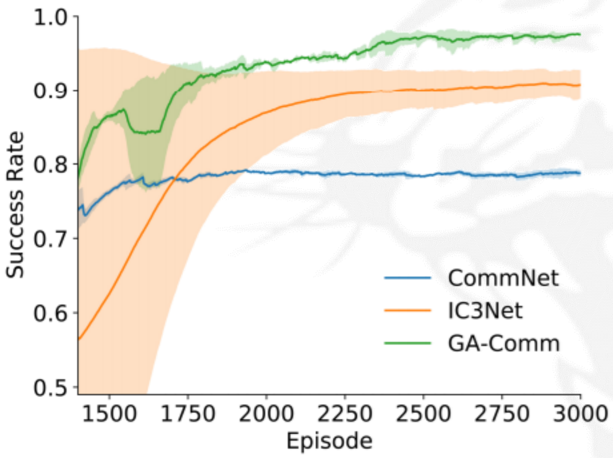


G2ANet Performance

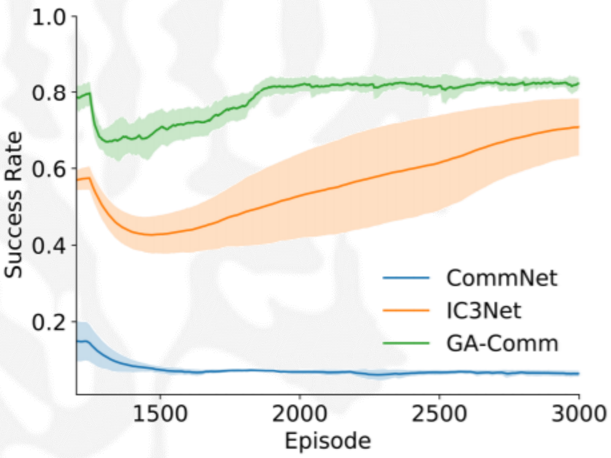
Algorithm	Easy	Medium	Hard
CommNet	93.5%	78.8%	6.5%
IC3Net	93.2%	90.8%	70.9%
GA-Comm	**99.7%**	**97.6%**	**82.3%**



(a) Easy



(b) Medium



(c) Hard

Thank you for your attention!

- More questions?
  - ywjoon95@korea.ac.kr