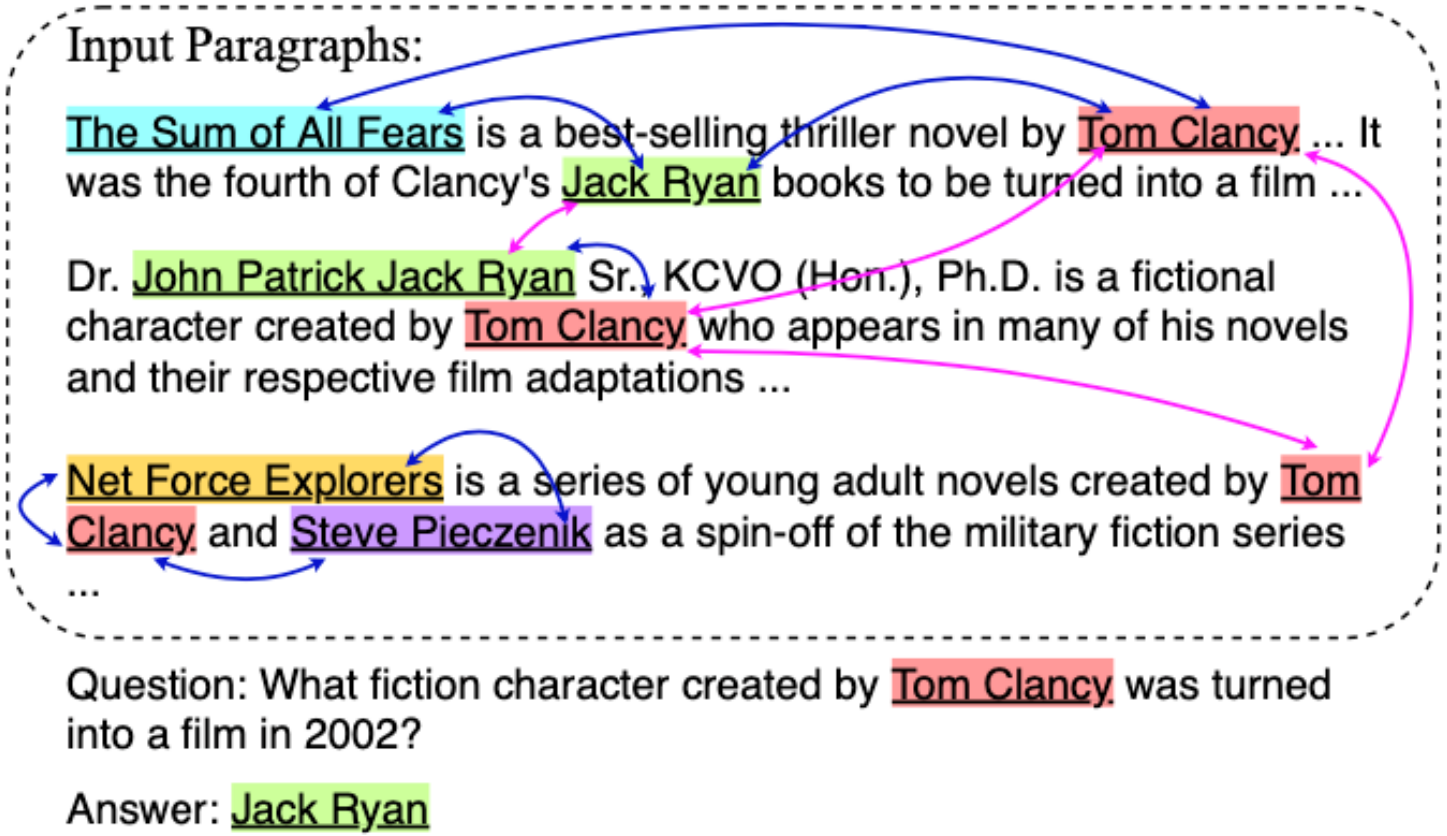
Dynamically Fused Graph Network for Multi-hop Reasoning (ACL 2019)

Hyunkyung Bae

Seoul National University

Multi-hop QA

- Task
  - Identify the correct answer from the set of supporting documents.



Multi-hop QA challenges

Previous workflow

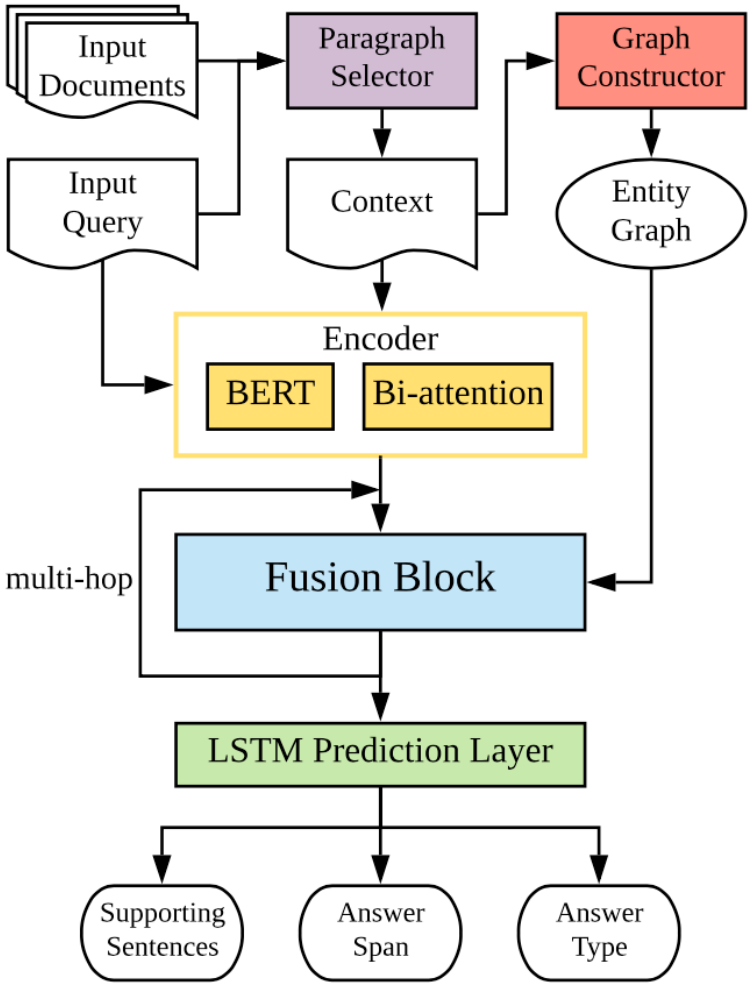
Limitations

- Hard to filter out noises and extract useful information from **static entity graph**
- Answers may **not** reside in the entity graph

In this work…

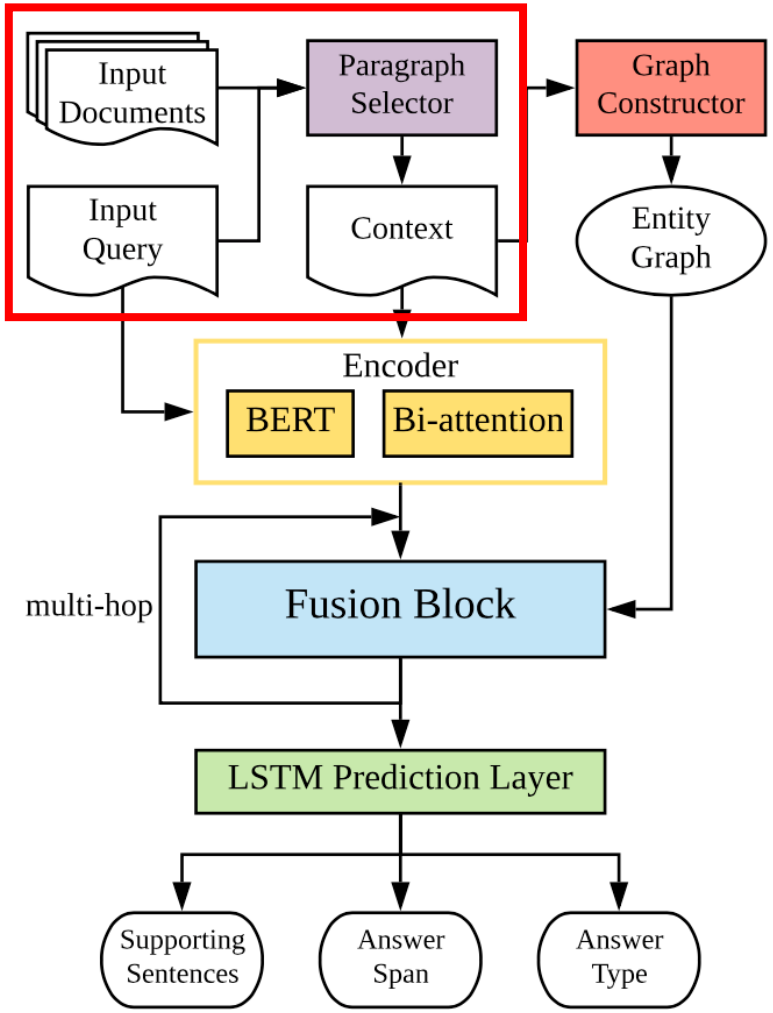
- **Dynamic entity graph** : update entity graph through masks
- **Fusion process** : find answers from documents

Dynamically Fused Graph Network



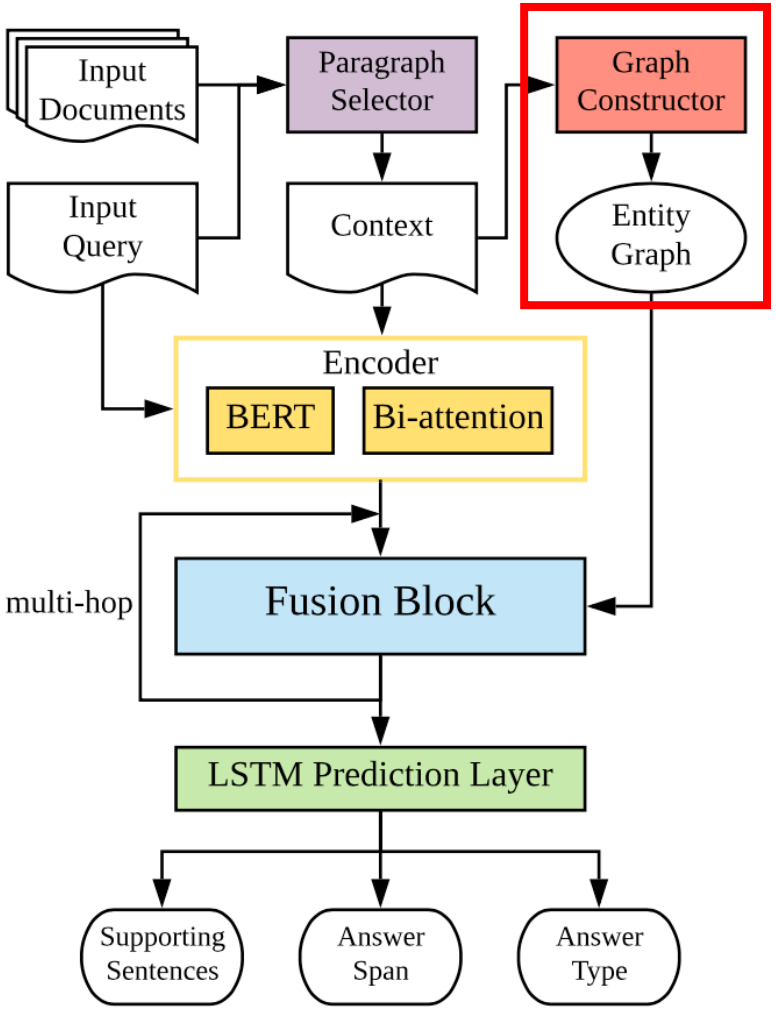
- Paragraph selection
- Constructing Entity Graph
- Encoding Query and Context
- Reasoning with Fusion Block
- Prediction

Dynamically Fused Graph Network



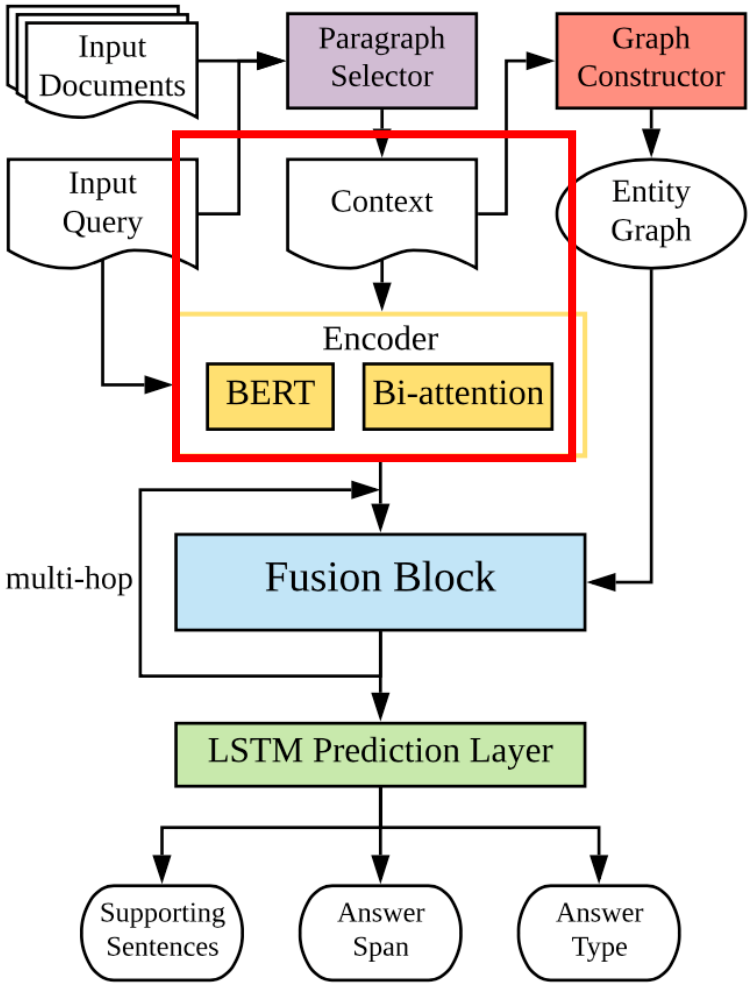
- **Paragraph selection**
- Constructing Entity Graph
- Encoding Query and Context
- Reasoning with Fusion Block
- Prediction

Dynamically Fused Graph Network



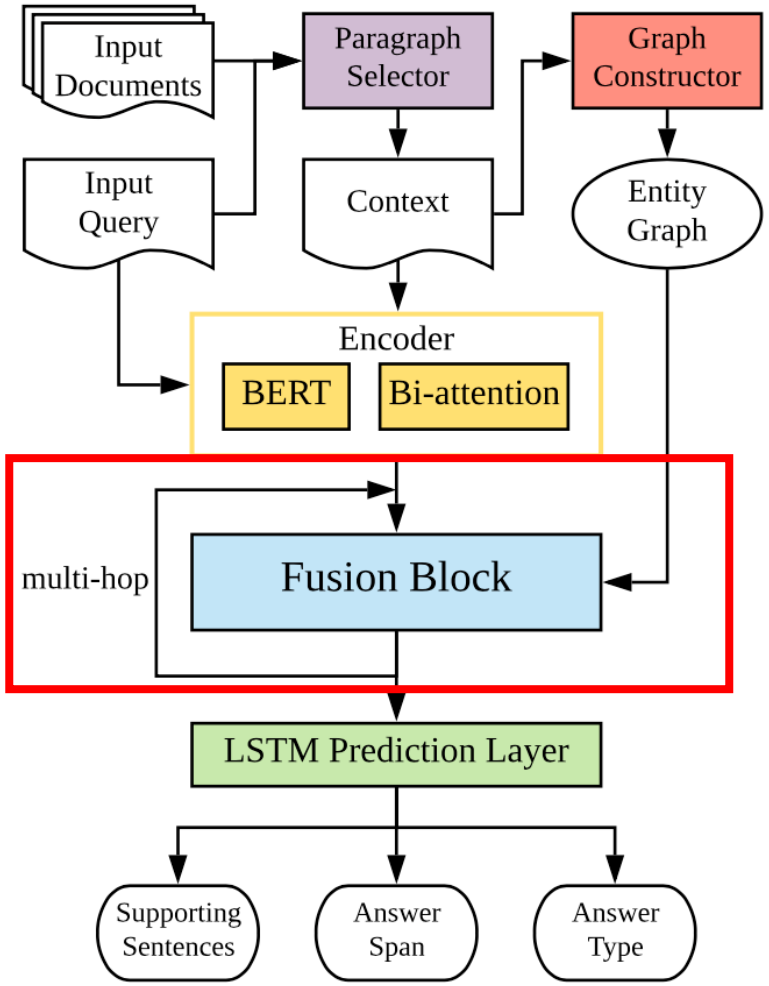
- Paragraph selection
- **Constructing Entity Graph**
- Encoding Query and Context
- Reasoning with Fusion Block
- Prediction

Dynamically Fused Graph Network



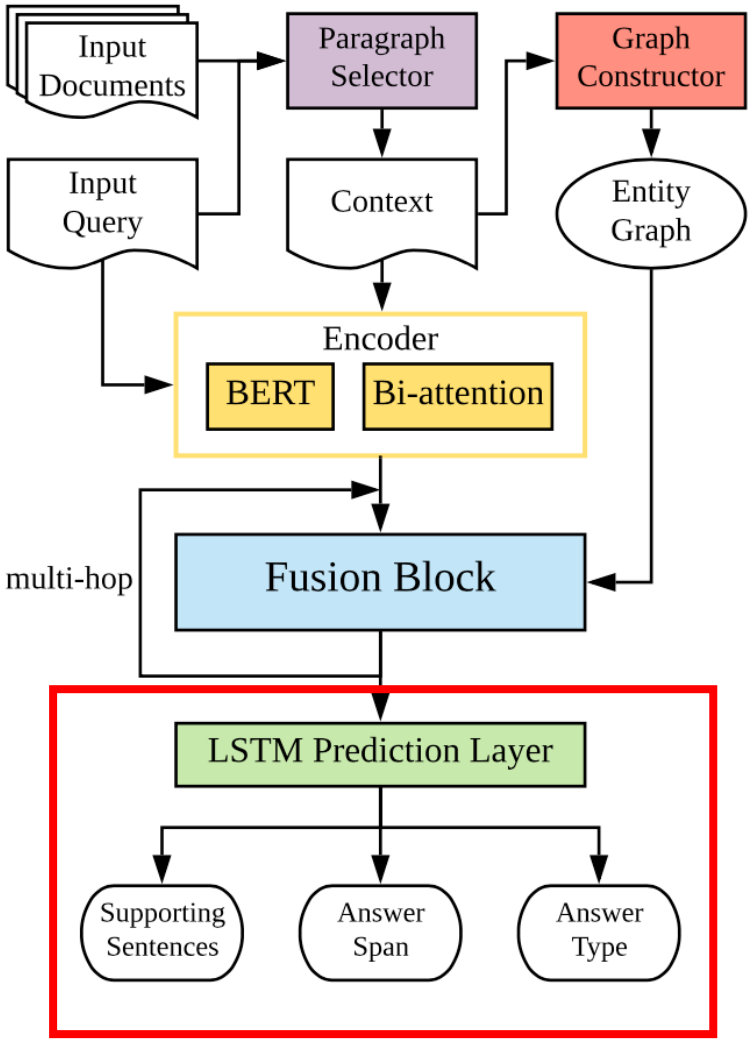
- Paragraph selection
- Constructing Entity Graph
- **Encoding Query and Context**
- Reasoning with Fusion Block
- Prediction

Dynamically Fused Graph Network



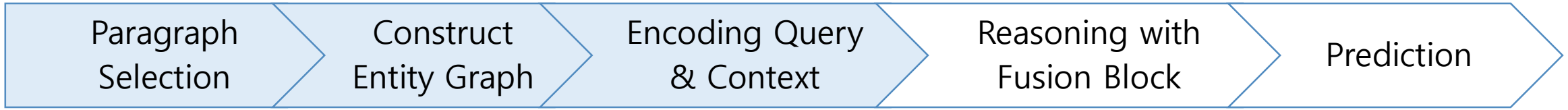
- Paragraph selection
- Constructing Entity Graph
- Encoding Query and Context
- **Reasoning with Fusion Block**
- Prediction

Dynamically Fused Graph Network



- Paragraph selection
- Constructing Entity Graph
- Encoding Query and Context
- Reasoning with Fusion Block
- **Prediction**

Dynamically Fused Graph Network



- Paragraph selection
  - Pre-trained BERT
  - Relevance score via sentence classification
  - Generate Context

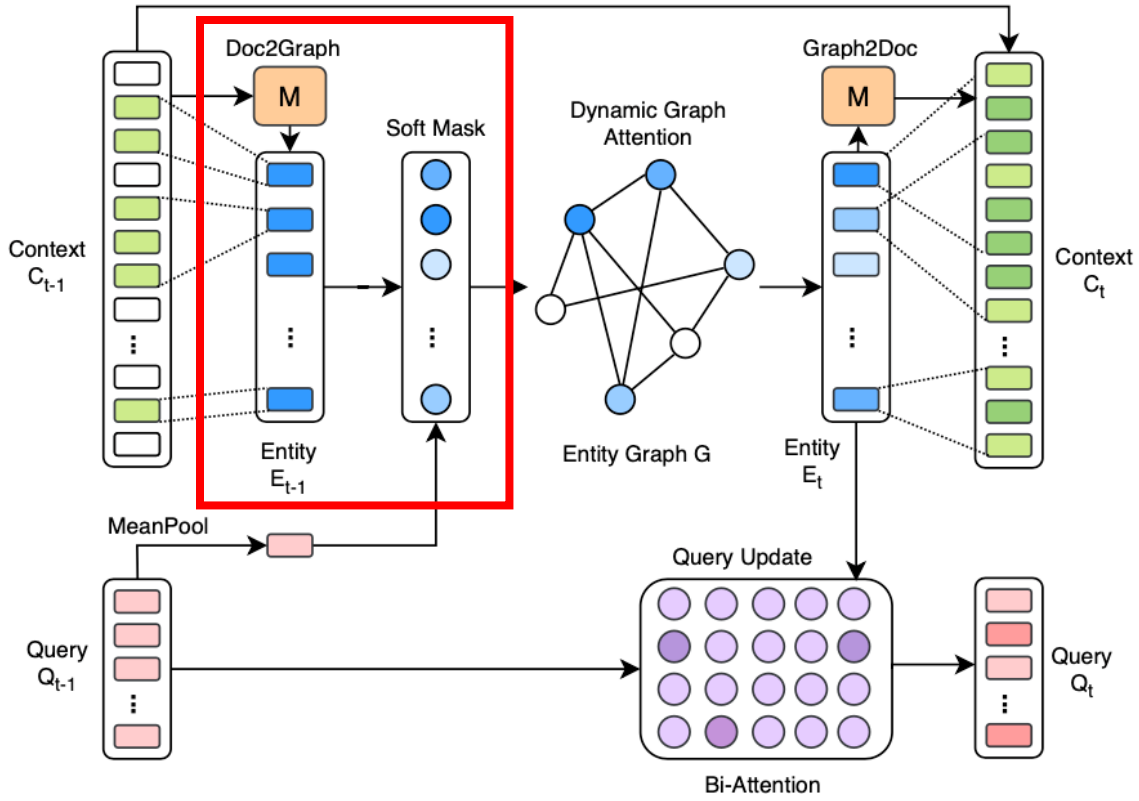
- Construct Entity Graph
  - Node from NER (Named Entity Recognization)
  - Edge
    - Sentence level
    - Context-level (links across multiple documents)
    - Paragraph-level

- Encoding Query and Context
  - BERT

Dynamically Fused Graph Network

Paragraph Selection → Construct Entity Graph → Encoding Query & Context → Reasoning with Fusion Block → Prediction

- Fusion Block



Document to Graph

- Token2Entity module
- Mean-max pooling in each text span
- $M_{i,j} =$

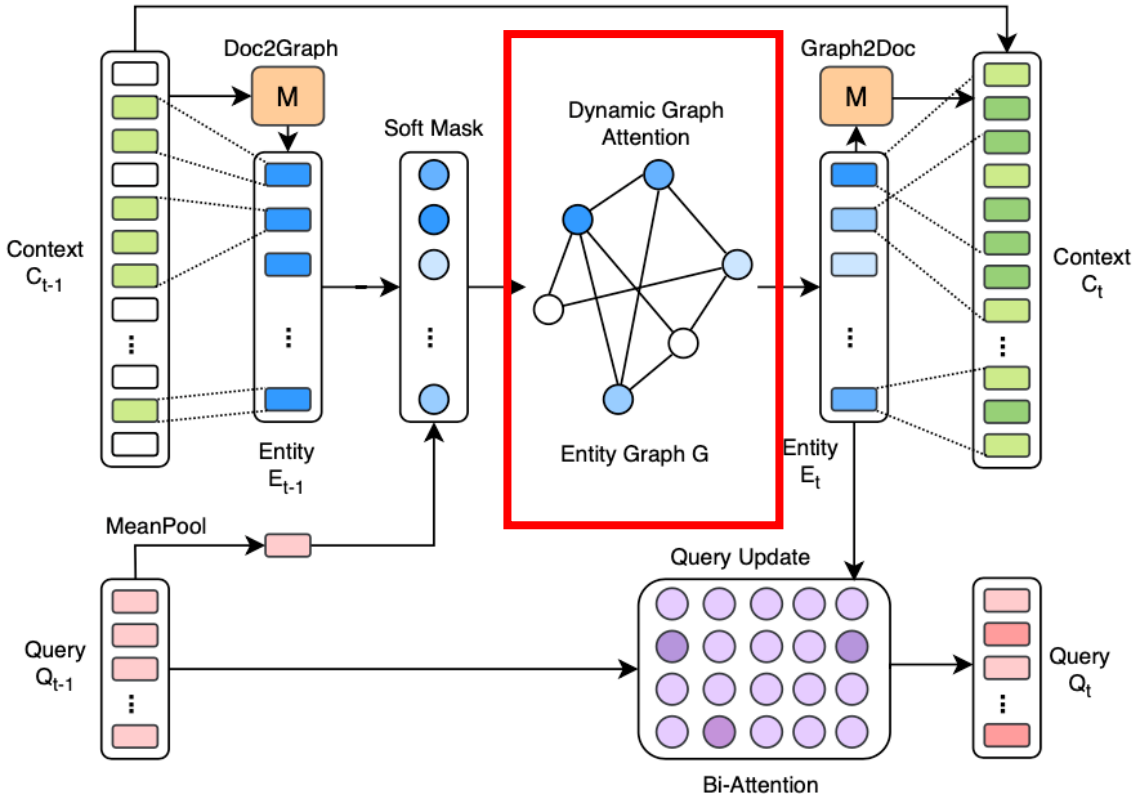
$$\begin{cases} 1 \; if \; ith \; is \; within \; the \; span \; of \; the \; jth \; entity \\ 0 \; otherwise \end{cases}$$

$$\mathbf{E}_{t-1} = [\mathbf{e}_{t-1,1}, \ldots, \mathbf{e}_{t-1,N}]$$

Dynamically Fused Graph Network

Paragraph Selection → Construct Entity Graph → Encoding Query & Context → Reasoning with Fusion Block → Prediction

- Fusion Block



Dynamic Graph Attention

- Soft mask (m): relevance score of each entity to query

Node features

$$\tilde{\mathbf{q}}^{(t-1)} = \mathbf{MeanPooling}(\mathbf{Q}^{(t-1)})$$

$$\gamma_i^{(t)} = \tilde{\mathbf{q}}^{(t-1)} \mathbf{V}^{(t)} \mathbf{e}_i^{(t-1)} / \sqrt{d_2}$$

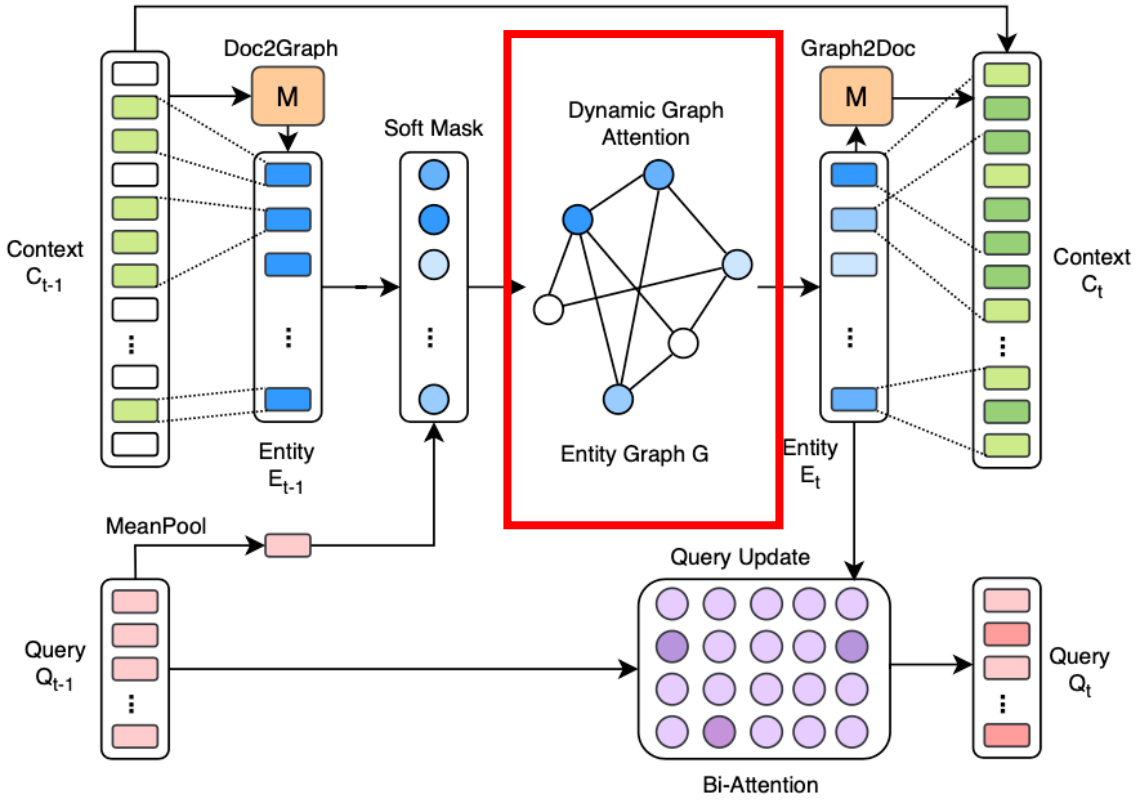
$$\mathbf{m}^{(t)} = \sigma([\gamma_1^{(t)}, \cdots, \gamma_N^{(t)}])$$

$$\tilde{\mathbf{E}}^{(t-1)} = [m_1^{(t)} \mathbf{e}_1^{(t-1)}, \ldots, m_N^{(t)} \mathbf{e}_N^{(t-1)}]$$

Dynamically Fused Graph Network

Paragraph Selection → Construct Entity Graph → Encoding Query & Context → Reasoning with Fusion Block → Prediction

- Fusion Block



Dynamic Graph Attention

GAT

$$\mathbf{h}_i^{(t)} = \mathbf{U}_t \tilde{\mathbf{e}}_i^{(t-1)} + \mathbf{b}_t$$

$$\beta_{i,j}^{(t)} = \mathrm{LeakyReLU}(\mathbf{W}_t^\top [\mathbf{h}_i^{(t)}, \mathbf{h}_j^{(t)}])$$

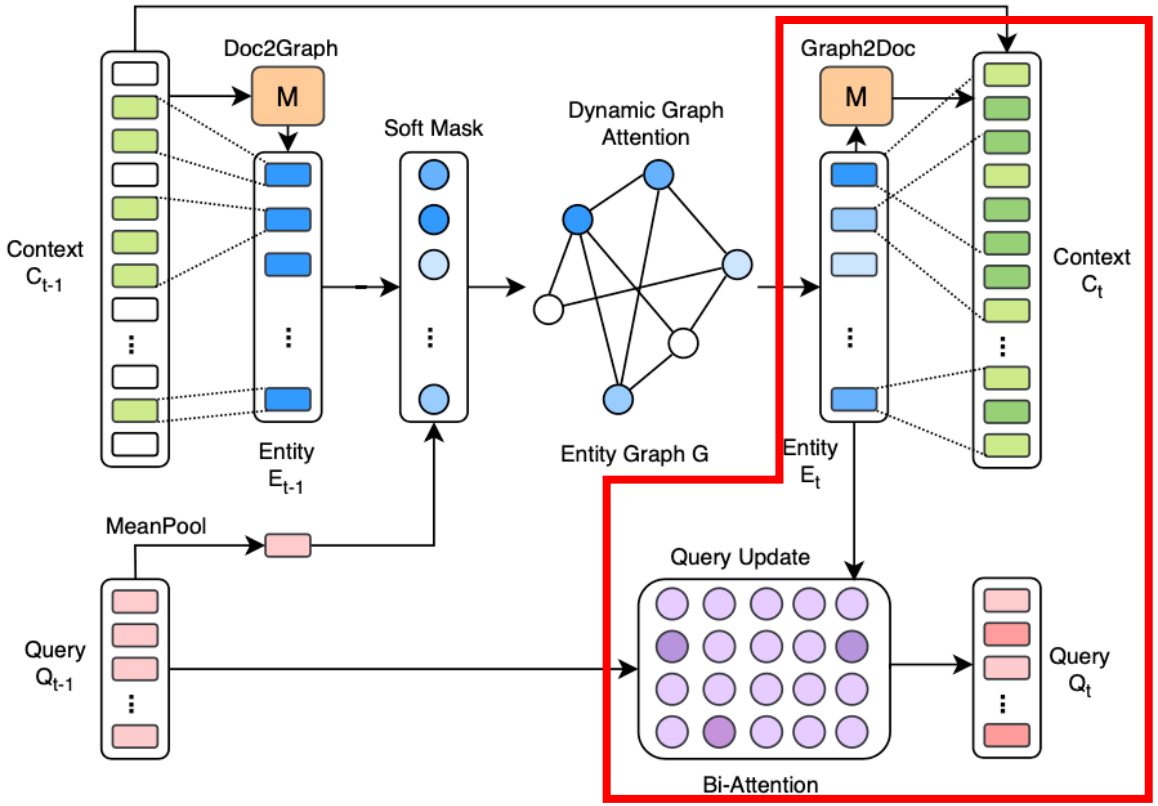
$$\alpha_{i,j}^{(t)} = \frac{\exp(\beta_{i,j}^{(t)})}{\sum_k \exp(\beta_{i,k}^{(t)})}$$

$$\mathbf{e}_i^{(t)} = \mathrm{ReLU}(\sum_{j \in B_i} \alpha_{j,i}^{(t)} \mathbf{h}_j^{(t)})$$

Dynamically Fused Graph Network

Paragraph Selection → Construct Entity Graph → Encoding Query & Context → **Reasoning with Fusion Block** → Prediction

- Fusion Block



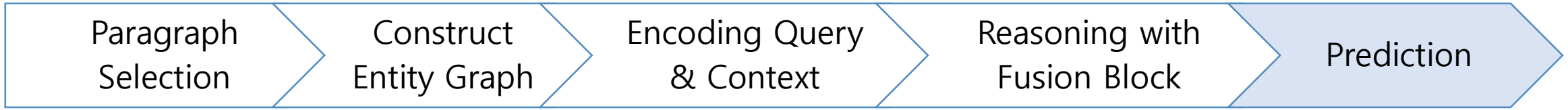
Graph to Document

$$\mathbf{C}^{(t)} = \text{LSTM}([\mathbf{C}^{(t-1)}, \mathbf{M}\mathbf{E}^{(t)\top}])$$

Updating query

$$\mathbf{Q}^{(t)} = \text{Bi-Attention}(\mathbf{Q}^{(t-1)}, \mathbf{E}^{(t)})$$

Dynamically Fused Graph Network



- Prediction
  - Supporting sentences
  - The start position of the answer
  - The end position of the answer
  - Answer type

$$\mathbf{O}_{sup} = \mathcal{F}_0(\mathbf{C}^{(t)})$$
$$\mathbf{O}_{start} = \mathcal{F}_1([\mathbf{C}^{(t)}, \mathbf{O}_{sup}])$$
$$\mathbf{O}_{end} = \mathcal{F}_2([\mathbf{C}^{(t)}, \mathbf{O}_{sup}, \mathbf{O}_{start}])$$
$$\mathbf{O}_{type} = \mathcal{F}_3([\mathbf{C}^{(t)}, \mathbf{O}_{sup}, \mathbf{O}_{end}])$$

$$\mathcal{L} = \mathcal{L}_{start} + \mathcal{L}_{end} + \lambda_s \mathcal{L}_{sup} + \lambda_t \mathcal{L}_{type}$$

Experiments

- Dataset
  - HotpotQA

- BERT Tokenizer
- Pre-trained BERT
- NER from Stanford CoreNLP Toolkit

Results

Model	Answer		Sup Fact		Joint	
	EM	F1	EM	F1	EM	F1
Baseline Model	45.60	59.02	20.32	64.49	10.83	40.16
GRN*	52.92	66.71	52.37	84.11	31.77	58.47
DFGN(Ours)	55.17	68.49	49.85	81.06	31.87	58.23
QFE*	53.86	68.06	**57.75**	**84.49**	**34.63**	59.61
DFGN(Ours)†	**56.31**	**69.69**	51.50	81.62	33.62	**59.82**

Table 1: Performance comparison on the private test set of HotpotQA in the distractor setting. Our DFGN is the second best result on the leaderboard before submission (on March 1st). The baseline model is from Yang et al. (2018) and the results with * is unpublished. DFGN(Ours)† refers to the same model with a revised entity graph, whose entities are recognized by a BERT NER model. Note that the result of DFGN(Ours)† is submitted to the leaderboard during the review process of our paper.

Ablation study

Setting	EM	F1
DFGN (2-layer)	55.42	69.23
- BFS Supervision	54.48	68.15
- Entity Mask	54.64	68.25
- Query Update	54.44	67.98
- E2T Process	53.91	67.45
- 1 Fusion Block	54.14	67.70
- 2 Fusion Blocks	53.44	67.11
- 2 Fusion Blocks & Bi-attn	50.03	62.83
gold paragraphs only	55.67	69.15
supporting facts only	57.57	71.67

Reproducing

Reproducing the experiment

- I failed to write the codes for reproducing the experiment in the paper. So I **forked** the source code from the author github repository and analyzed the code structure instead.
- Forked repo: https://github.com/jennybae1024/DFGN-pytorch
- The table below is **the result of the HotpotQA experiment** I've run with the author's code.
  - We assume that the learned paragraph selection module checkpoint is provided.

Results

	em	f1	pr	re	sp_em	sp_f1	sp_pr	sp_re	jt_em	jt_f1	jt_pr	jt_re
ep00	0.2535	0.3551	0.3614	0.3790	0.2440	0.6554	0.7951	0.6004	0.0783	0.2548	0.3043	0.2517
ep01	0.3018	0.4121	0.4294	0.4267	0.3722	0.7423	0.8086	0.7277	0.1325	0.3284	0.3665	0.3356
ep02	0.3199	0.4275	0.4433	0.4416	0.3702	0.7523	0.7923	0.7595	0.1325	0.3402	0.3676	0.3558
ep03	0.3336	0.4451	0.4629	0.4606	0.4001	0.7702	0.8055	0.7793	0.1518	0.3614	0.3899	0.3789
ep04	0.3425	0.4586	0.4671	0.4845	0.4085	0.7649	0.8052	0.7673	0.1572	0.3711	0.3959	0.3920
ep05	0.4801	0.6194	0.6432	0.6356	0.4059	0.7896	0.7826	0.8381	0.2238	0.5128	0.5266	0.5554
ep06	0.5025	0.6379	0.6637	0.6502	0.4589	0.7935	0.8428	0.7836	0.2647	0.5306	0.5829	0.5337
ep07	0.5115	0.6528	0.6732	0.6730	0.4704	0.7982	0.8457	0.7875	0.2759	0.5467	0.5939	0.5563

Best score at epoch 7

- EM: exact matching
- PR: precision
- RE: recall
- SP: supporting facts
- JT: Joint performance

Code analysis overview

- DFGN is composed of three modules
  - Paragraph selection
    - Text classification
  - Fusion block
    - Doc2Graph : generate node feature vectors from docs embeddings
    - GAT
    - Graph2Doc : update docs embeddings based on node feature vectors from GAT
    - Update query : update query embeddings
  - Prediction layer
    - LSTM

- Layers in **Fusion Block**
  - MeanMaxPooling (Doc2Graph)
  - AttentionLayer (GAT)
  - InteractionLayer (Graph2Doc / Context update)
  - BiAttention (Query update)

Code analysis – BasicBlock (Fusion block unit)

```python
class BasicBlock(nn.Module):
    def __init__(self, hidden_dim, q_dim, layer, config):
        super(BasicBlock, self).__init__()
        self.config = config
        self.layer = layer
        self.gnn_type = config.gnn.split(':')[0]
        if config.tok2ent == 'mean_max':
            input_dim = hidden_dim * 2
        else:
            input_dim = hidden_dim
        self.tok2ent = tok_to_ent(config.tok2ent)()
        self.query_weight = get_weights((q_dim, input_dim))
        self.temp = np.sqrt(q_dim * input_dim)
        self.gat = AttentionLayer(input_dim, hidden_dim, config.n_heads, config, layer_id=layer)
        self.int_layer = InteractionLayer(hidden_dim * 2, hidden_dim, config)

    def forward(self, doc_state, query_vec, batch):
        context_mask = batch['context_mask']
        entity_mapping = batch['entity_mapping']
        entity_length = batch['entity_lens']
        entity_mask = batch['entity_mask']
        doc_length = batch['context_lens']
        adj = batch['entity_graphs']

        entity_state = self.tok2ent(doc_state, entity_mapping, entity_length)

        query = torch.matmul(query_vec, self.query_weight)
        query_scores = torch.bmm(entity_state, query.unsqueeze(2)) / self.temp
        softmask = query_scores * entity_mask.unsqueeze(2)  # N x E x 1  BCELossWithLogits
        adj_mask = torch.sigmoid(softmask)

        entity_state = self.gat(entity_state, adj, entity_mask, adj_mask=adj_mask, query_vec=query_vec)
        doc_state = self.int_layer(doc_state, entity_state, doc_length, entity_mapping, entity_length, context_mask)
        return doc_state, entity_state, softmask
```

→ Transform docs embeddings to entity embeddings

→ Doc2Graph, GAT, Graph2Doc

→ Query update

Code analysis - Layers in Fusion Block (tok2ent : MeanMaxPooling)

- MeanMaxPooling is used in a tok2ent module, which transforms documents embeddings to entity embeddings.

```python
def mean_pooling(input, mask):
    mean_pooled = input.sum(dim=1) / mask.sum(dim=1, keepdim=True)
    return mean_pooled


class MeanPooling(nn.Module):
    def __init__(self):
        super(MeanPooling, self).__init__()

    def forward(self, doc_state, entity_mapping, entity_lens):
        entity_states = entity_mapping.unsqueeze(3) * doc_state.unsqueeze(1)  # N x E x L x d
        mean_pooled = torch.sum(entity_states, dim=2) / entity_lens.unsqueeze(2)
        return mean_pooled


class MeanMaxPooling(nn.Module):
    def __init__(self):
        super(MeanMaxPooling, self).__init__()

    def forward(self, doc_state, entity_mapping, entity_lens):
        """
        :param doc_state:  N x L x d
        :param entity_mapping:  N x E x L
        :param entity_lens:  N x E
        :return: N x E x 2d
        """
        entity_states = entity_mapping.unsqueeze(3) * doc_state.unsqueeze(1)  # N x E x L x d
        max_pooled = torch.max(entity_states, dim=2)[0]
        mean_pooled = torch.sum(entity_states, dim=2) / entity_lens.unsqueeze(2)
        output = torch.cat([max_pooled, mean_pooled], dim=2)  # N x E x 2d
        return output
```

tok2ent mapping was given

output : entity graph's node feature vectors

Code analysis - Layers in Fusion Block (AttentionLayer)

- softmax: filter out noise by implementing an attention mechanism with Q

```python
class AttentionLayer(nn.Module):
    def __init__(self, in_dim, hid_dim, n_head, config, layer_id=0):
        super(AttentionLayer, self).__init__()
        assert hid_dim % n_head == 0
        self.dropout = config.gnn_drop

        self.attn_funcs = nn.ModuleList()
        for i in range(n_head):
            self.attn_funcs.append(
                GATSelfAttention(in_dim=in_dim, out_dim=hid_dim // n_head,
                                 config=config, layer_id=layer_id, head_id=i))

        if in_dim != hid_dim:
            self.align_dim = nn.Linear(in_dim, hid_dim)
            nn.init.xavier_uniform_(self.align_dim.weight, gain=1.414)
        else:
            self.align_dim = lambda x: x

    def forward(self, input, adj, entity_mask, adj_mask=None, query_vec=None):
        hidden_list = []
        for attn in self.attn_funcs:
            h = attn(input, adj, entity_mask, adj_mask=adj_mask, query_vec=query_vec)
            hidden_list.append(h)

        h = torch.cat(hidden_list, dim=-1)
        h = F.dropout(h, self.dropout, training=self.training)
        h = F.relu(h)
        return h
```

```python
class GATSelfAttention(nn.Module):
    def __init__(self, in_dim, out_dim, config, layer_id=0, head_id=0):...

    def forward(self, input_state, adj, entity_mask, adj_mask=None, query_vec=None):
        zero_vec = torch.zeros_like(adj)
        scores = 0

        for i in range(self.n_type):
            h = torch.matmul(input_state, self.W_type[i])
            h = F.dropout(h, self.dropout, self.training)
            N, E, d = h.shape

            a_input = torch.cat([h.repeat(1, 1, E).view(N, E * E, -1), h.repeat(1, E, 1)], dim=-1)
            a_input = a_input.view(-1, E, E, 2*d)

            if self.q_attn:
                q_gate = F.relu(torch.matmul(query_vec, self.qattn_W1[i]))
                q_gate = torch.sigmoid(torch.matmul(q_gate, self.qattn_W2[i]))
                a_input = a_input * q_gate[:, None, None, :]
                score = self.act(torch.matmul(a_input, self.a_type[i]).squeeze(3))
            else:
                score = self.act(torch.matmul(a_input, self.a_type[i]).squeeze(3))
            scores += torch.where(adj == i+1, score, zero_vec)

        zero_vec = -9e15 * torch.ones_like(scores)
        scores = torch.where(adj > 0, scores, zero_vec)

        # Ahead Alloc
        if adj_mask is not None:
            h = h * adj_mask

        coefs = F.softmax(scores, dim=2)  # N * E * E
        h = coefs.unsqueeze(3) * h.unsqueeze(2)  # N * E * E * d
        h = torch.sum(h, dim=1)
        return h
```

Code analysis - Layers in Fusion Block (InteractionLayer)

```python
class InteractionLayer(nn.Module):
    def __init__(self, input_dim, out_dim, config):
        super(InteractionLayer, self).__init__()
        self.config = config
        self.use_trans = config.basicblock_trans

        if config.basicblock_trans:
            bert_config = BertConfig(input_dim, config.trans_heads, config.trans_drop)
            self.transformer = BertLayer(bert_config)
            self.transformer_linear = nn.Linear(input_dim, out_dim)
        else:
            self.lstm = LSTMWrapper(input_dim, out_dim // 2, 1)

    def forward(self, doc_state, entity_state, doc_length, entity_mapping, entity_length, context_mask):
        """
        :param doc_state: N x L x dc
        :param entity_state: N x E x de
        :param entity_mapping: N x E x L
        :return: doc_state: N x L x out_dim, entity_state: N x L x out_dim (x2)
        """
        expand_entity_state = torch.sum(entity_state.unsqueeze(2) * entity_mapping.unsqueeze(3), dim=1)
        input_state = torch.cat([expand_entity_state, doc_state], dim=2)

        if self.use_trans:
            extended_attention_mask = context_mask.unsqueeze(1).unsqueeze(2)
            extended_attention_mask = (1.0 - extended_attention_mask) * -10000.0

            doc_state = self.transformer(input_state, extended_attention_mask)
            doc_state = self.transformer_linear(doc_state)
        else:
            doc_state = self.lstm(input_state, doc_length)

        return doc_state
```

- tok2ent mapping was given.
- The output shape should be compatible with the previous context embedding matrix.

Context embedding update via LSTM

Code analysis - Layers in Fusion Block (BiAttention)

```python
class BiAttention(nn.Module):
    def __init__(self, input_dim, memory_dim, hid_dim, dropout):...

    def forward(self, input, memory, mask):
        """
        :param input: context_encoding N * Ld * d
        :param memory: query_encoding N * Lm * d
        :param mask: query_mask N * Lm
        :return:
        """
        bsz, input_len, memory_len = input.size(0), input.size(1), memory.size(1)

        input = F.dropout(input, self.dropout, training=self.training)  # N x Ld x d
        memory = F.dropout(memory, self.dropout, training=self.training)  # N x Lm x d

        input_dot = self.input_linear_1(input)  # N x Ld x 1
        memory_dot = self.memory_linear_1(memory).view(bsz, 1, memory_len)  # N x 1 x Lm
        # N * Ld * Lm
        cross_dot = torch.bmm(input, memory.permute(0, 2, 1).contiguous()) / self.dot_scale
        # [f1, f2]^T [w1, w2] + <f1 * w3, f2>
        # (N * Ld * 1) + (N * 1 * Lm) + (N * Ld * Lm)
        att = input_dot + memory_dot + cross_dot  # N x Ld x Lm
        # N * Ld * Lm
        att = att - 1e30 * (1 - mask[:, None])

        input = self.input_linear_2(input)
        memory = self.memory_linear_2(memory)

        weight_one = F.softmax(att, dim=-1)
        output_one = torch.bmm(weight_one, memory)
        weight_two = F.softmax(att.max(dim=-1)[0], dim=-1).view(bsz, 1, input_len)
        output_two = torch.bmm(weight_two, input)

        return torch.cat([input, output_one, input*output_one, output_two*output_one], dim=-1), memory
```

- input_linear_1/2, memory_linear_1/2
  : Linear projection

- Dot-product attention mechanism