Supplementary

Xu Yang[1,*] Chongyang Gao[2,*] Hanwang Zhang[3] Jianfei Cai[4]
[1] School of Computer Science and Engineering, Southeast University
[2] Department of Computer Science, Dartmouth College
[3] School of Computer Science and Engineering, Nanyang Technological University
[4] Department of Data Science and AI, Monash University
xuyangaca@gmail.com chongyang.gao.gr@dartmouth.edu hanwangzhang@ntu.edu.sg
Jianfei.Cai@monash.edu

1. Parse Trees

More examples of parse trees generated by our APN are illustrated. Specifically, there are three examples for captioning in Figure 1 and three examples for VQA in Figure 2.

We visualize the language and image trees for captioning in Figure 1. To parse trees, we use the M of all N_L and N_V self-attention layers constrained by the probabilistic graphical model in our APN. The left and middle parts show the trees of the generated caption and the given image, respectively. The RoIs are sorted from top-left to bottom-right. Because of the structure of the Transformer and the developed algorithm, our APN generates captions in a phrase-by-phrase manner instead of a word-by-word manner, as shown by the leaves of the language trees. For example, in the second row, our captioner first generates the phrase "a dog walking" and in this way, more details such as "in the snow" and "with a stick" will be described. Comparing language trees with visual trees, we can find that they have similar hierarchical structures, *e.g.*, in the second row, the leftmost leaves of both trees focus on "a dog walking" and the rightmost leaves focus on 'with a stick"; in the third row, the leaves of both trees focus on "a woman", "holding" and "a pair of scissors" from the left to the right leaves.

For parsed trees in VQA, We use the M from all $N_c = 6$ probabilistic graphical model constrained self-attention layers of the decoder in our APN to parse the trees. Three examples are illustrated in the Figure 2. In these examples, we can also find similar hierarchical structures between language and visual trees. For example, in the first row, the leaves of language and visual trees focus on "this person" and "doing" from the left to the right leaves.

2. PGM constrain implementation details

We show the pseudocode of probabilistic graphical model constrain in PyTorch-like style in Algorithm 1. In the implementation, we first prepare the masks to constrain the connections of each x to its neighbors. Then, we calculate the Bernoulli distribution and the θ following Eq. (5). The $\tilde{\theta}$ is calculated by Eq. (10) to satisfy the requirement that the entities in a lower-layer cluster are still in a higher-layer cluster. Finally, we calculate M by Eq. (8) and use it to constrain the self-attention layer. By Eq. (9), in a soft way, the original fully-connected graph becomes a sparser graph that contains a few clusters, and the entities are only connected in the same cluster.

3. Performance of improved constraint matrix

For the VQA-v2 dataset, the overall accuracy of the model using the constraint matrix without our improvement is 66.32, which is worse than the PGM's accuracy, 66.51. In the image captioning task, the model using the constraint matrix without improvement achieves 129.8 CIDEr-D score, which is worse than the PGM's CIDEr-D score, 130.4 in Table 2. Thus, our improved constraint matrix is better than [63] and effectively improves the performance.

[*] Both authors contributed equally to this research.

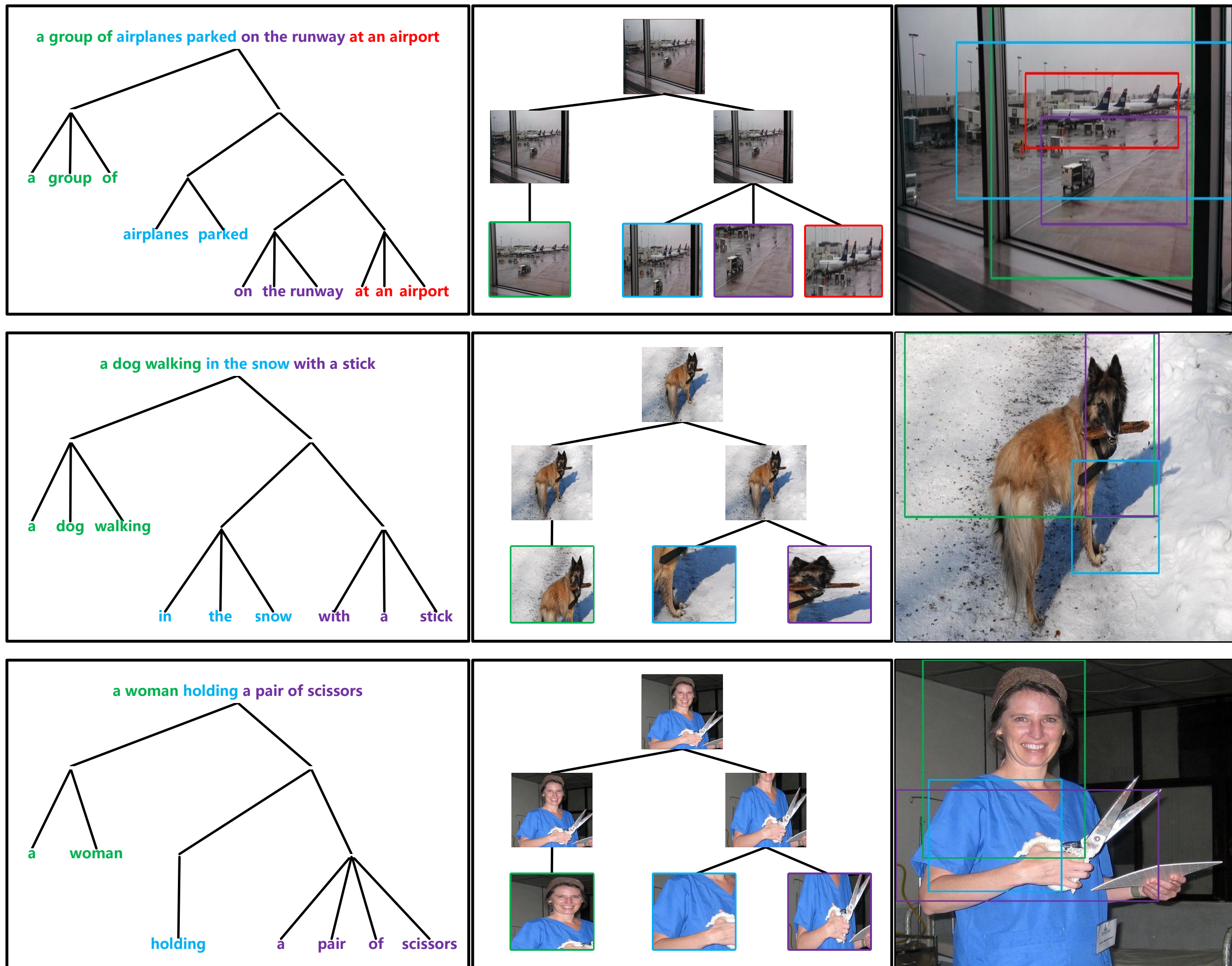

Figure 1. The parsed trees for captioning. The same colors between language and visual trees show the alignments.

4. Removing the PGM from the N_c module.

For the VQA-v2 dataset, the overall accuracy of the PGM whose constrain from the N_C module is removed is 66.17, which is better than the accuracy of BASE 65.84 and worse than the original accuracy of the PGM, 66.51 in Table 4.

Figure 2. The parsed trees for VQA. The same colors between language and visual trees show the alignments.

Algorithm 1 Pseudocode of probabilistic graphical model constrain in a PyTorch-like style

```
class PGM-attention(nn.Module):
    def __init__(self, hidden_size=512, dropout=0.8):
        super(PGM-attention, self).__init__()
        self.h_size = hidden_size
        self.linear_key = nn.Linear(hidden_size,hidden_size)
        self.linear_query = nn.Linear(hidden_size,hidden_size)
        self.norm = LayerNorm(hidden_size)
        self.dropout = nn.Dropout(dropout)
    def forward(self, context, eos_mask, prior_tilde_theta):
        batch_size, seq_len = context.size()[:2]
        # Prepare neighbors mask
        a = torch.from_numpy(np.diag(np.ones(seq_len - 1, dtype=np.int64), 1))
        b = torch.from_numpy(np.diag(np.ones(seq_len, dtype=np.int64), 0))
        c = torch.from_numpy(np.diag(np.ones(seq_len - 1, dtype=np.int64), -1))
        tri_matrix = torch.from_numpy(np.triu(np.ones([seq_len, seq_len],
dtype=np.float32), 0))
        mask = eos_mask & (a + c)
        # Calculate Bernoulli distributions
        key = self.linear_key(context)
        query = self.linear_query(context)
        att = torch.matmul(query, key.transpose(-2, -1)) / torch.sqrt(self.h_size)
        Ber = F.softmax(att.masked_fill(mask == 0, -1e9), dim=-1)
        # Calculate Tilde Theta
        theta = Ber * Ber.transpose(-2, -1) + 1e-9
        tilde_theta = prior_tilde_theta + (1.  - prior_tilde_theta) * theta
        tilde_theta= torch.log(tilde_theta + 1e-9).masked_fill(a == 0,
0).matmul(tri_matrix)
        # Calculate M
        M = tri_matrix.matmul(tilde_theta).exp().masked_fill((tri_matrix.int() - b)
== 0, 0)
        M = M + M.transpose(-2, -1) + tilde_theta.masked_fill(b == 0, 1e-9)
        return M
```