Visual Question Answering: SAN

Presentation slides for DL4NLP project

Press Space for next page →

6/28/2022

Project Introduction

工作简介

- **VQA** - 以一张图片和一个关于图片内容的自然语言形式的问题作为输入，要求输出正确答案
- **Dataset** - VQAv2
- **Summary** - 属于一种多标签分类的问题，计算损失的时候采用多标签损失

开发环境

- 开发工具：ModelArts Ascend Notebook环境，选用Ascend910芯片作为训练芯片
- 开发包、资源库
  - Mindspore1.3.0
  - numpy
- 系统运行要求: python3.7.5 与可运行 Mindspore1.3.0 的开发环境

Read more about Our Repository

DataLoader

数据集简介

VQAv2数据集的构成如下：

- 1张图片有大概5个问题
- 1个问题有10个答案
- test没有annotation文件

Question

```
question{
"question_id" : int,  #问题id
"image_id" : int,     #问题对应的图片id
"question" : str      #具体的问题
}
```

Annotations

```
annotation{
"question_id" : int,
"image_id" : int,
"question_type" : str,          #问题类型
"answer_type" : str,            #答案类型
"answers" : [answer],
"multiple_choice_answer" : str
}
------------------------------
answer{
"answer_id" : int,
"answer" : str,                #具体答案
"answer_confidence": str
}
```

DataLoader

数据预处理

- 问题与答案对齐
- 图片与问题对齐

数据集加载

```
Generating answers vocab...
Answers vocab is generated
(4, 128)
[[ 0. 10.  0. ...  0.  0.  0.]
 [ 0.  0.  0. ...  0.  0.  0.]
 [ 0.  0.  0. ...  0.  0.  0.]
 [ 0.  0.  0. ...  0.  0.  0.]]
(4, 3, 224, 224)
```

数据类型与预处理如下：

1. img: 图片为三通道RGB模式，加载成三维的Tensor即可
2. question：预处理，进行词形还原，大小写转换等，再通过预训练的Tokenizer进行one-hot编码，扩充成定长向量输出。
3. Answers：预处理，进行词形还原，大小写转换等，自己构造词汇表进行one-hot编码

Image Embedding

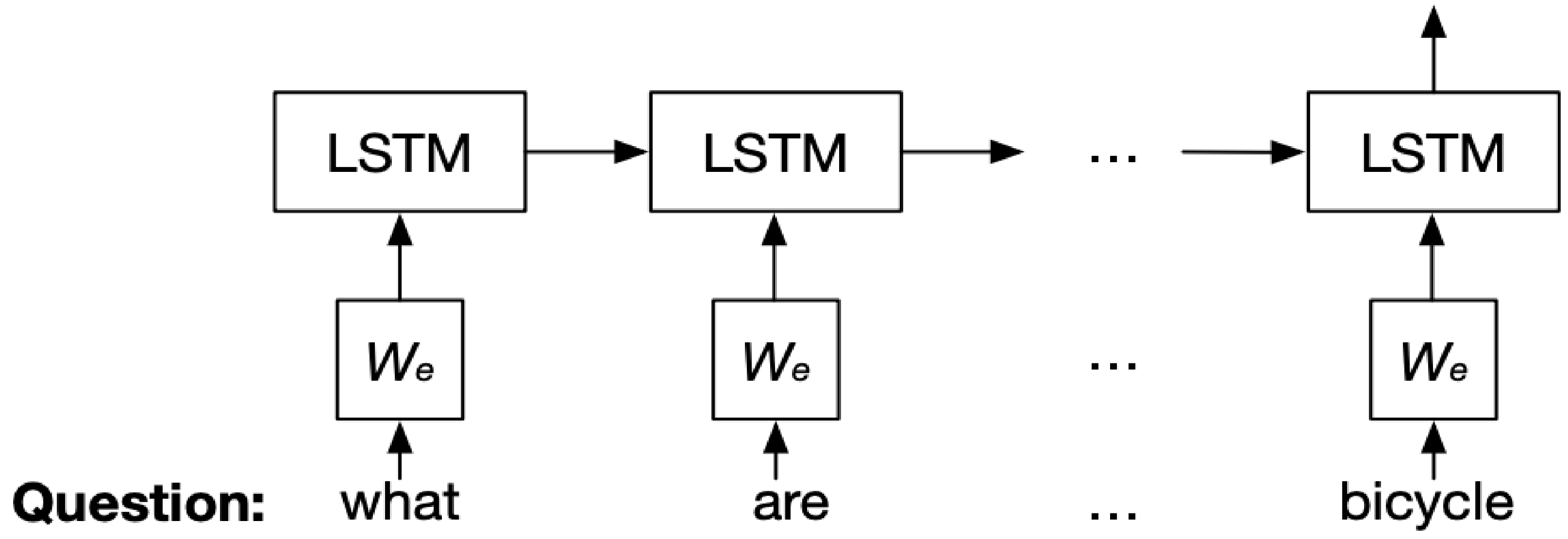
直接输入图片（3x224x224）到搭建的卷积网络，输出特征（196x768），而非直接使用提取好的特征

```
self.simple_cnn = nn.SequentialCell([
            nn.Conv2d(self.in_channels, self.channels, kernel_size=3, stride=2, padding=0, pad_mode='same'),
            nn.BatchNorm2d(
                self.channels, eps=1e-4, momentum=0.9, gamma_init=1, beta_init=0,
                moving_mean_init=0, moving_var_init=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, pad_mode="same"),
            nn.Conv2d(self.channels, self.channels * 2, kernel_size=3, stride=1, padding=0, pad_mode='same'),
            nn.BatchNorm2d(self.channels*2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,stride=2),
            nn.Conv2d(self.channels * 2, self.channels*4, kernel_size=3, stride=1, padding=0, pad_mode='same'),
            nn.BatchNorm2d(self.channels * 4),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2,stride=2),
            nn.Conv2d(self.channels*4, output_size, kernel_size=3, stride=1, padding=0, pad_mode='same')
        ])
```

Text Embedding: LSTM

According to the paper we refer, the text embdding part is LSTM, which is easy to implemented.

```
from mindspore.nn import LSTM
lstm = LSTM(input_size, hidden_size, num_layers)
```



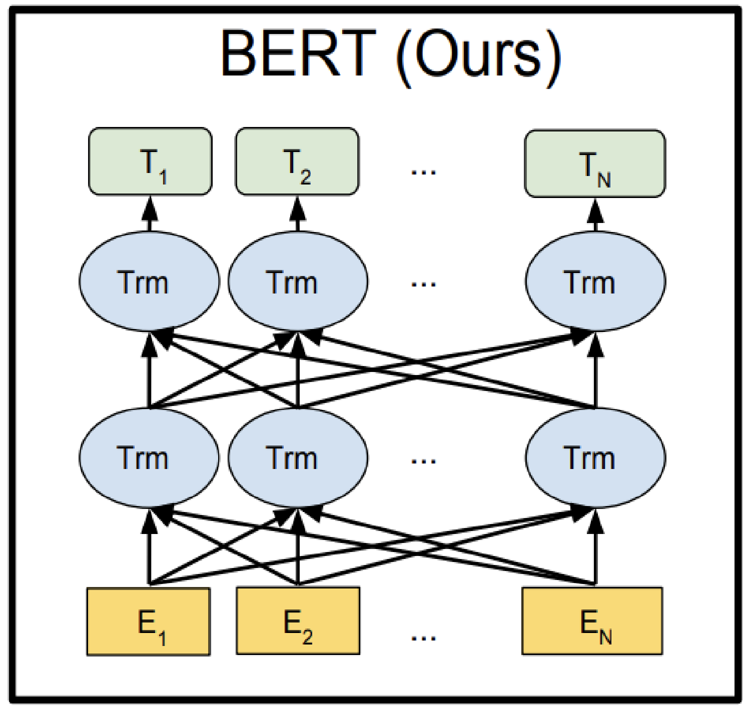
LSTM Shortcome:

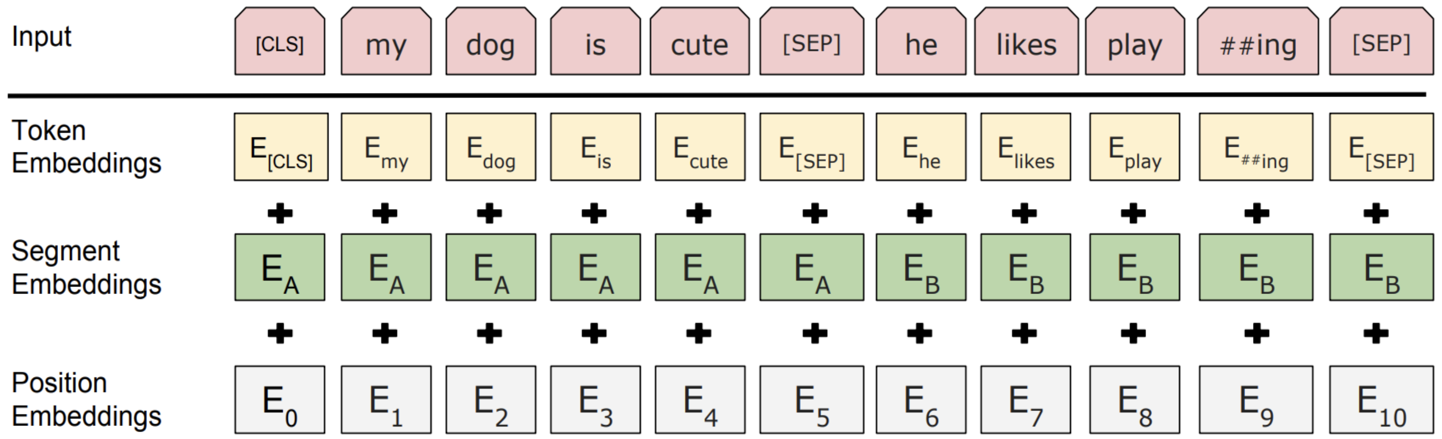
- Good RNN variant, but still not good at handling long sequence.
- Only "look forward", not able to "look back"

Thus, we consider to use BERT

Text Embedding: BERT

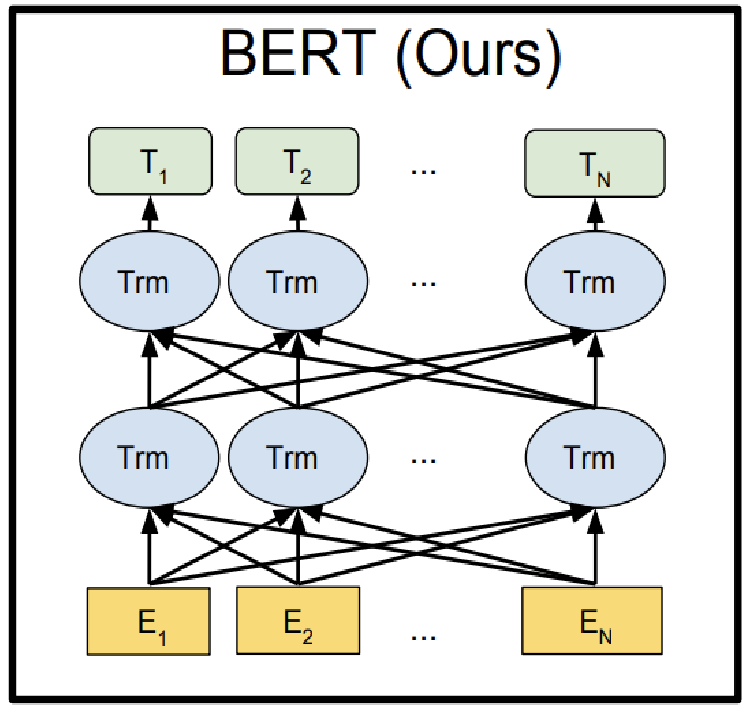
```python
class BertModel(BertPretrainCell):
    def construct:
        # Embedding
        token_embedding, segment_embedding, position_embedding = \
            BertEmbeddings(input_ids)
        embedding_output = token_embedding +
            segment_embedding + position_embedding
        # BertEncoder == Transformer
        encoder_outputs = self.encoder(embedding_output,
                                       extended_attention_mask,
                                       head_mask)
        # sequence_output (batch_size, len(sequence), embedding_size)
        # Embedding of every word
        sequence_output = encoder_output[0]
        # pooled_output (batch_size, embedding_size)
        # Embedding of the input sequence
        pooled_output = self.pooler(sequence_output)
        return sequence_output, pooled_output
```

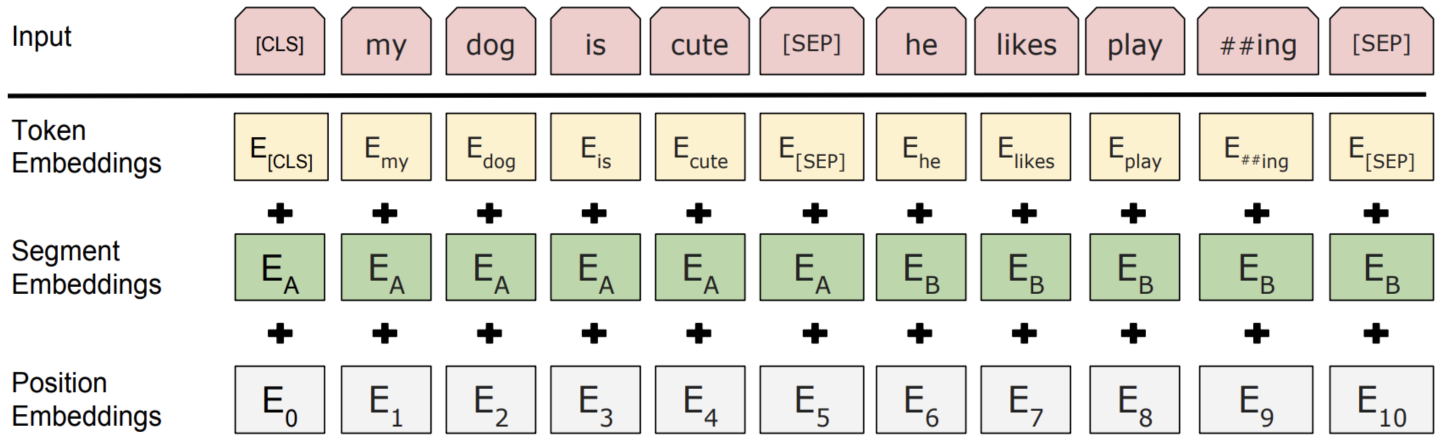




Text Embedding: BERT

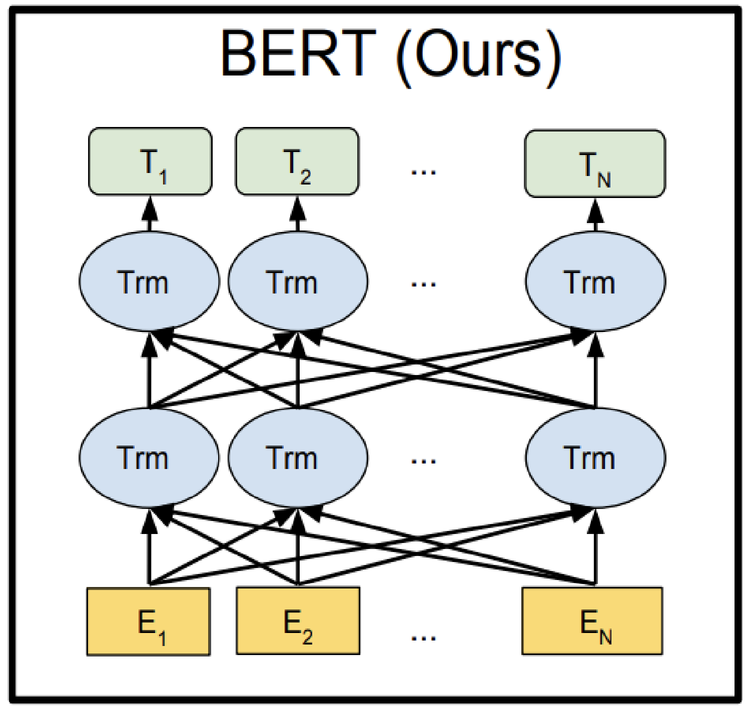
```
class BertModel(BertPretrainCell):
    def construct:
        # Embedding
        token_embedding, segment_embedding, position_embedding = \
            BertEmbeddings(input_ids)
        embedding_output = token_embedding +
            segment_embedding + position_embedding
        # BertEncoder == Transformer
        encoder_outputs = self.encoder(embedding_output,
                                       extended_attention_mask,
                                       head_mask)
        # sequence_output (batch_size, len(sequence), embedding_size)
        # Embedding of every word
        sequence_output = encoder_output[0]
        # pooled_output (batch_size, embedding_size)
        # Embedding of the input sequence
        pooled_output = self.pooler(sequence_output)
        return sequence_output, pooled_output
```

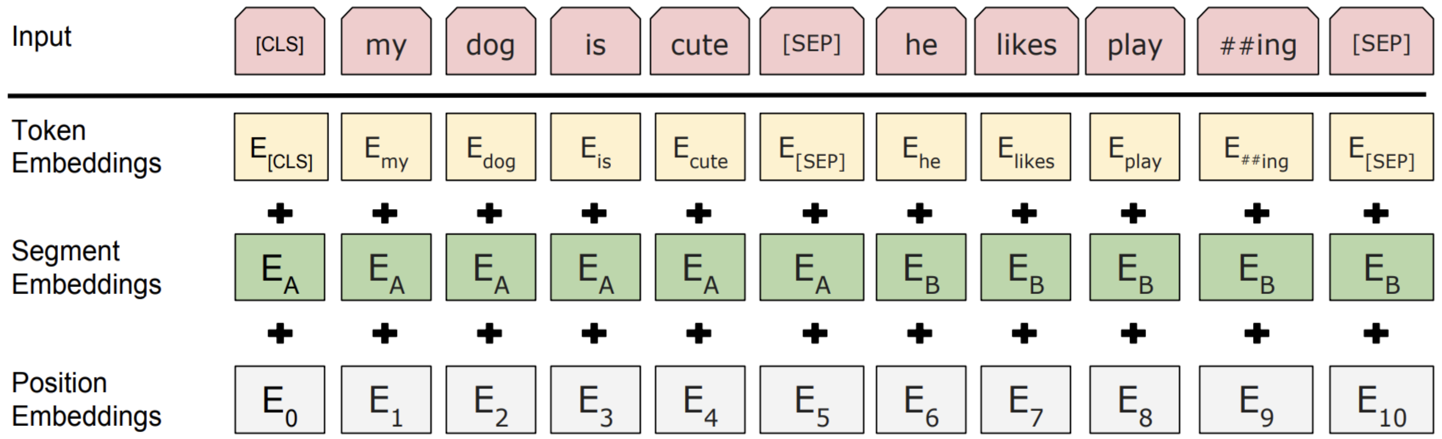




Text Embedding: BERT

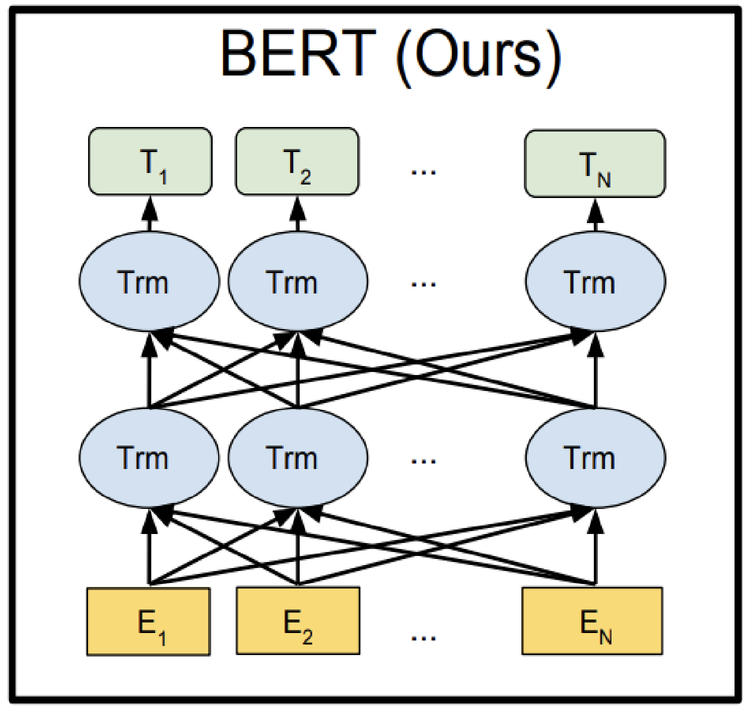
```python
class BertModel(BertPretrainCell):
    def construct:
        # Embedding
        token_embedding, segment_embedding, position_embedding = \
            BertEmbeddings(input_ids)
        embedding_output = token_embedding +
            segment_embedding + position_embedding
        # BertEncoder == Transformer
        encoder_outputs = self.encoder(embedding_output,
                                       extended_attention_mask,
                                       head_mask)
        # sequence_output (batch_size, len(sequence), embedding_size)
        # Embedding of every word
        sequence_output = encoder_output[0]
        # pooled_output (batch_size, embedding_size)
        # Embedding of the input sequence
        pooled_output = self.pooler(sequence_output)
        return sequence_output, pooled_output
```

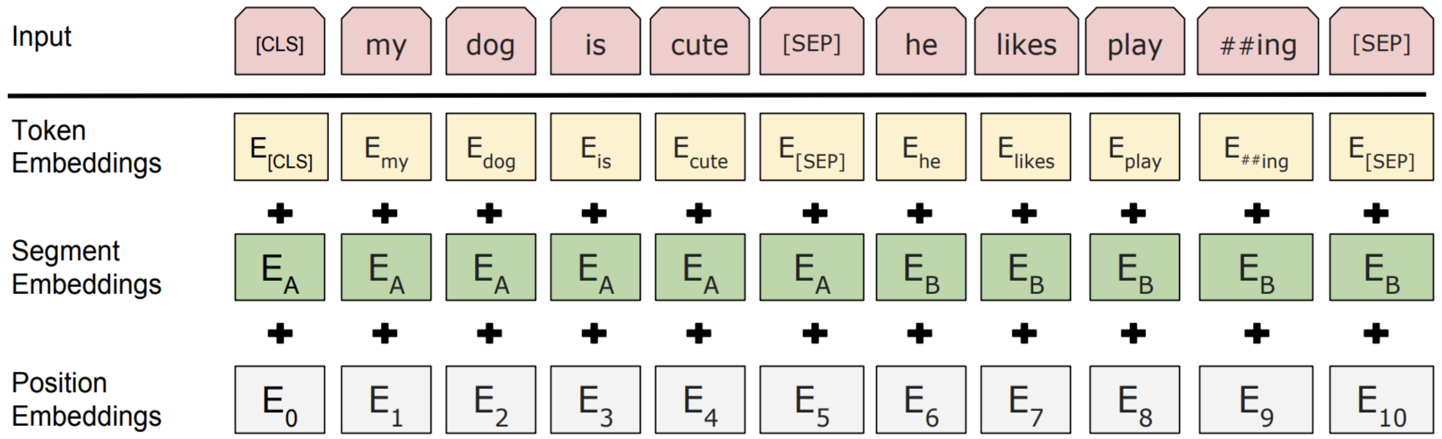




Text Embedding: BERT

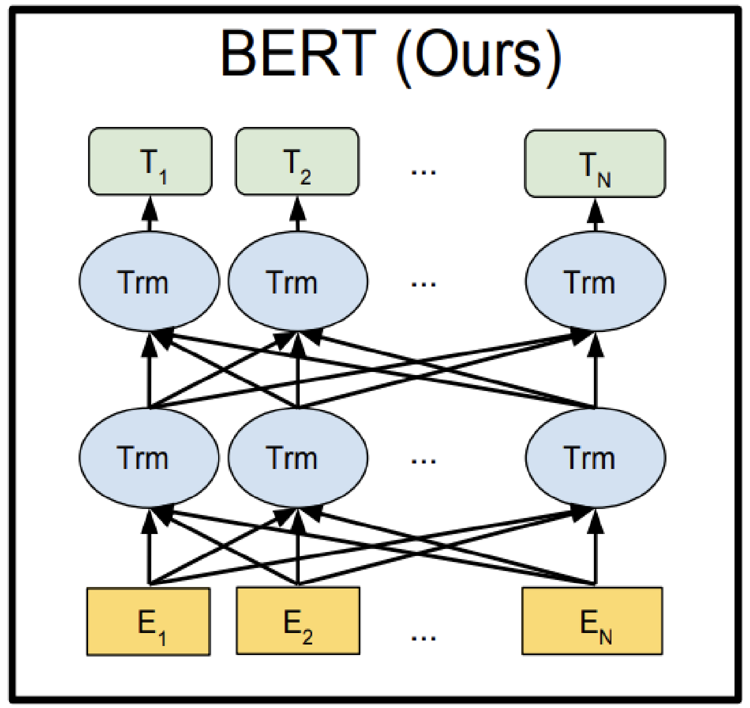
```
class BertModel(BertPretrainCell):
    def construct:
        # Embedding
        token_embedding, segment_embedding, position_embedding = \
            BertEmbeddings(input_ids)
        embedding_output = token_embedding +
            segment_embedding + position_embedding
        # BertEncoder == Transformer
        encoder_outputs = self.encoder(embedding_output,
                                       extended_attention_mask,
                                       head_mask)

        # sequence_output (batch_size, len(sequence), embedding_size)
        # Embedding of every word
        sequence_output = encoder_output[0]
        # pooled_output (batch_size, embedding_size)
        # Embedding of the input sequence
        pooled_output = self.pooler(sequence_output)
        return sequence_output, pooled_output
```

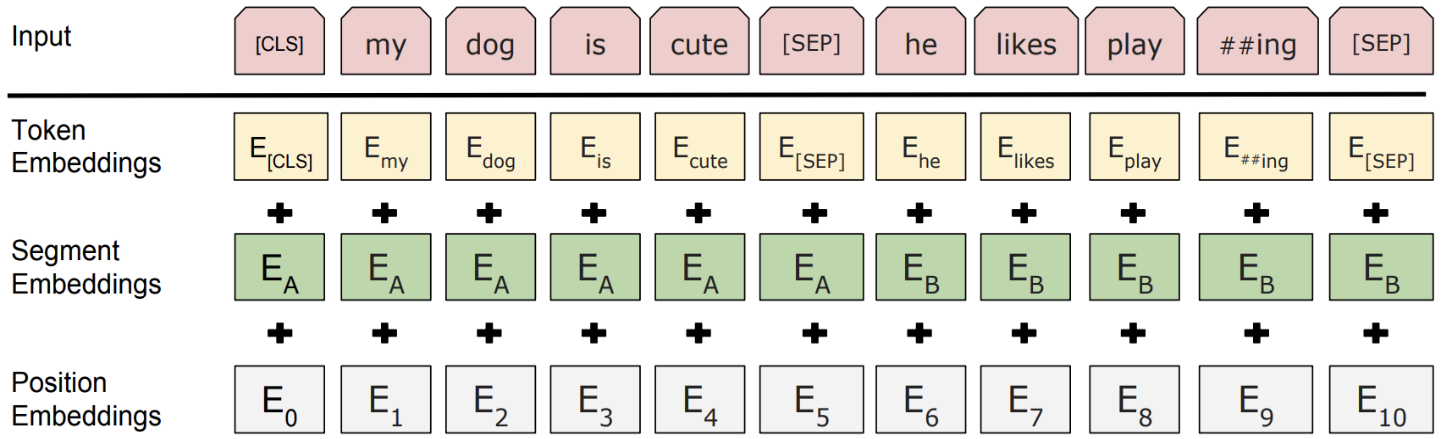




Text Embedding: BERT

```
class BertModel(BertPretrainCell):
    def construct:
        # Embedding
        token_embedding, segment_embedding, position_embedding = \
            BertEmbeddings(input_ids)
        embedding_output = token_embedding +
            segment_embedding + position_embedding
        # BertEncoder == Transformer
        encoder_outputs = self.encoder(embedding_output,
                                       extended_attention_mask,
                                       head_mask)
        # sequence_output (batch_size, len(sequence), embedding_size)
        # Embedding of every word
        sequence_output = encoder_output[0]
        # pooled_output (batch_size, embedding_size)
        # Embedding of the input sequence
        pooled_output = self.pooler(sequence_output)
        return sequence_output, pooled_output
```





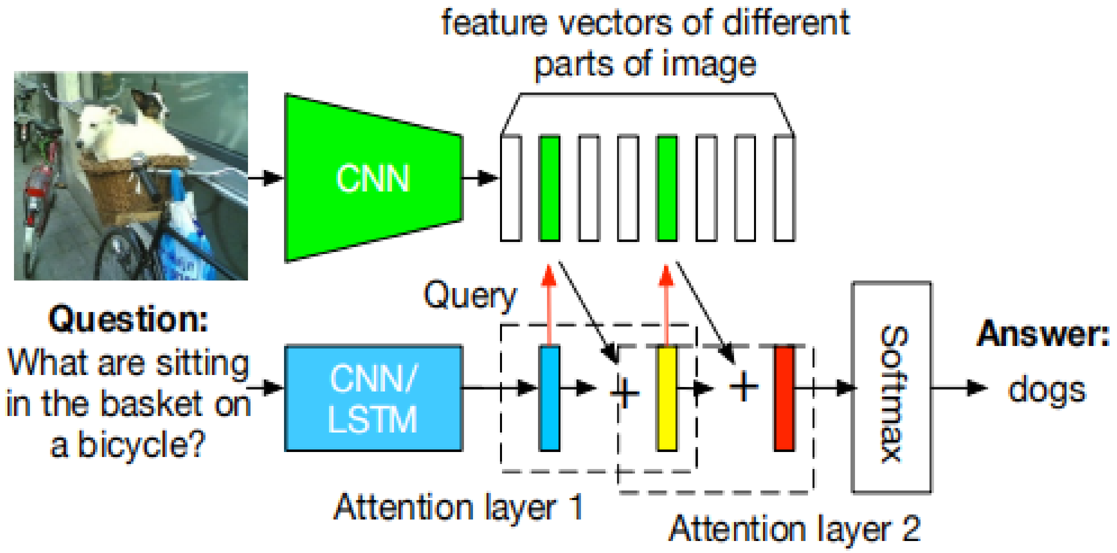
Stacked Attention Network

模型功能

- 使用多层 Attention 识别图像中不同区域与句子向量的相关程度。
- 筛选与句子向量有关的区域，与答案建立联系。
- 输入问题：What are sitting in the basket on a bicycle?
- 输入图片：

- 经过两层 Attention:

- 输出答案：Dogs.
- 整体模型结构：



Attention Distribution

- 在SAN中，最核心的问题为 Attention Distribution: 图片中的每个区域与问题的关联程度 (Attention) 为多少?

- 为此，我们先利用CNN和Bert对图像和问题进行编码：

$$v_I \in R^{d\times m}$$
$$v_Q \in R^d$$

  其中，v_I 为编码后的图像矩阵，v_Q 为编码后的问题句向量，d为表示维度，m是图像中区域的个数（利用CNN）。

- Attention 通过如下计算得到：

$$h_A = \tanh(W_I v_I \oplus W_Q v_Q)$$
$$p_I = \text{softmax}(W_p h_A)$$

我们首先让v_I v_Q 分别通过全连接层，使得它们的维度变为 $R^{k\times m}$ 和 R^k. 这里，$\oplus$ 操作代表把向量加到矩阵的每一列上。图像矩阵的每一列代表每个兴趣区域，因此这里的操作实际上就是把句子向量与每个兴趣区域做融合。由此再将 h_A 通过全连接层和 Softmax，就得到了图像中每个区域在特定句子中能成为兴趣区域的可能性，也就是我们的 **Attention Distribution**.

- 有了 Attention Distribution 后，我们利用它计算每个区域的权重和 $\hat{v_I} \in R^d$：

$$\hat{v_I} = \sum_i p_i v_i$$

  接着，把这个向量与句向量相加，得到整合后的查询向量 $u \in R^d$。

$$u = \hat{v_I} + v_Q$$

- 以上就是单层Attention的思路。

Attention Distribution

- 单层 Attention 的表示性并不强，所以我们可以使用多层 Attention，即将查询向量作为新的问题向量，不断输入Attention层进行迭代：

$$
\begin{aligned}
h_A^k &= \tanh(W_I^k v_I \oplus W_Q^k u^{k-1}) \\
p_I &= \operatorname{softmax}(W_p^k h_A^k) \\
\hat{v_I}^k &= \sum_i p_i^k v_i \\
u^k &= \hat{v_I}^k + u^{k-1}
\end{aligned}
$$

经过K次Attention迭代后，我们使用全连接层和Softmax推理答案：

$$p_{\text{ans}} = \operatorname{softmax}(W_u u^K)$$

Attention方法相较传统方法的优势

- 传统方法仅仅是将整体图片向量与问题向量合并，对区域信息不敏感。
- 相较于传统方法，Attention方法得到的查询向量 u 更具有信息表示性，因为与问题更相关的区域得到了更高的权重。

模型训练及验证

1. 为了支持多个输入(`question`和`img`)，自定义`WithLossCell`

```
class WithLossCell(nn.Cell):
    def __init__(self, model):
        super(WithLossCell, self).__init__(
            auto_prefix=False)
        self.loss = nn.SoftmaxCrossEntropyWithLogits()
        self.net = model

    def construct(self, q, a, img):
        out = self.net(q, img)
        loss = self.loss(out, a)
        return loss
```

2. 定义训练网络

```
#定义网络
model = san.SANModel()
#定义优化器
opt = nn.Adam(params=model.trainable_params())
#定义带Loss的网络
net_with_loss = WithLossCell(model)
#包装训练网络
train_net = TrainOneStepCell(net_with_loss, opt)
#设置训练模式
train_net.set_train(True)
```

模型训练及验证

3.定义验证网络，模型输出只要属于对应问题的十个答案之一即认为正确

```
class WithAccuracy(nn.Cell):
    def __init__(self, model):
        super(WithAccuracy, self).__init__(
            auto_prefix=False)
        self.net = model

    def construct(self, q, a, img):
        out = self.net(q, img)
        out = ops.Argmax(output_type=mindspore.int32)(out)
        return out, a
#model即为训练网络中同一个model
eval_net = WithAccuracy(model)
#设置验证模式
eval_net.set_train(False)
```

4.每个batch计算准确率，最后求平均得到某个epoch的验证准确率

```
out, a = eval_net(q, a, img)
predicted = out.asnumpy()
ans = a.asnumpy()
batch_size = ans.shape[0]
acc = 0
for i in range(batch_size):
    if ans[i,predicted[i]]!=0:
        acc += 1
accuracy = acc / batch_size
```

Thanks