faster rcnn

Conv layers。作为一种CNN网络目标检测方法，Faster RCNN首先使用一组基础的conv+relu+pooling层提取image的feature maps。该feature maps被共享用于后续RPN层和全连接层。

Region Proposal Networks。RPN网络用于生成region proposals。该层通过softmax判断anchors属于positive或者negative，再利用
bounding box regression修正anchors获得精确的proposals。
RPN网络结构总结起来就是：
生成anchors -> softmax分类器提取positvie anchors -> bbox reg回归positive anchors -> Proposal Layer生成proposals

Roi Pooling。该层收集输入的feature maps和proposals，综合这些信息后提取proposal feature maps，送入后续全连接层判定目标类别。

RoI Pooling层则负责收集proposal，并计算出proposal feature maps，送入后续网络。Rol pooling层有2个输入：
原始的feature maps
RPN输出的proposal boxes（大小各不相同）

Classification。利用proposal feature maps计算proposal的类别，同时再次bounding box regression获得检测框最终的精确位置。

MCAN

设置了两个注意力单元（general attention units）：一个自注意力单元（self-attention（SA）unit）进行模态内部交互和一个导向注意力单元（guided-attention（GA）unit）进行模态之间交互。之后再用一个协同注意力模块层（Modular Co-Attention（MCA）layers）将两个单元串联起来，最后将多个模块层串联起来，组成MCAN网络

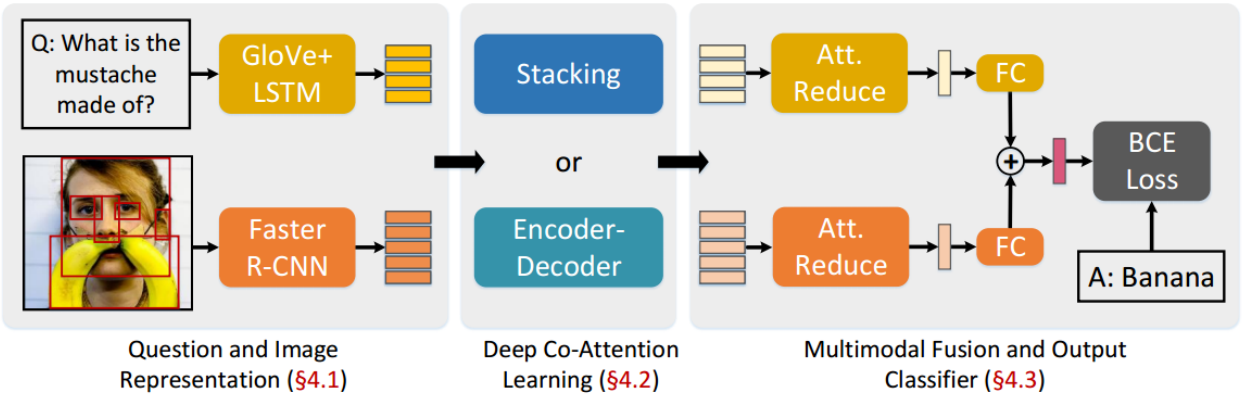
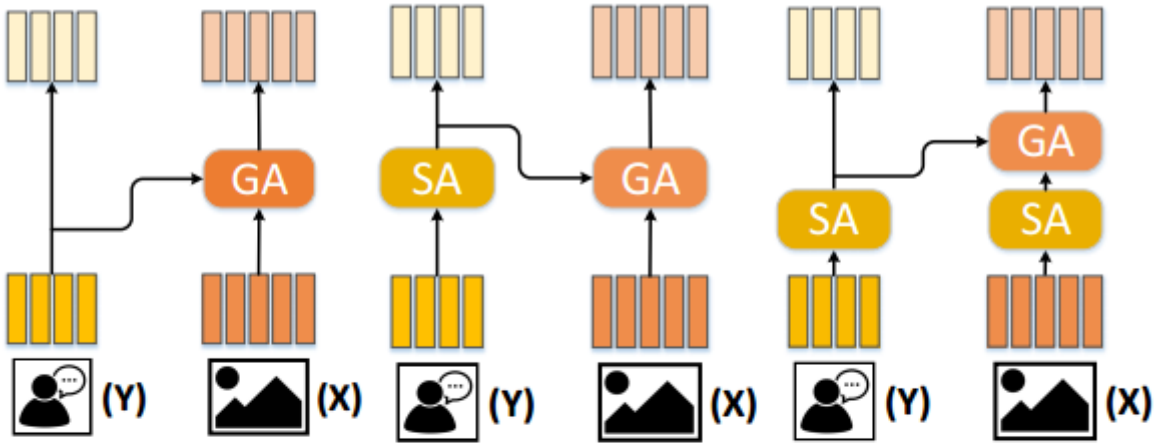


Figure 4: Overall flowchart of the deep Modular Co-Attention Networks (MCAN). In the Deep Co-attention Learning stage, we have two alternative strategies for deep co-attention learning, namely *stacking* and *encoder-decoder*.

```
# Net
# img_feat已经是faster rcnn处理过了
def forward(self, img_feat, ques_ix):
    # Make mask
    lang_feat_mask = self.make_mask(ques_ix.unsqueeze(2))
    img_feat_mask = self.make_mask(img_feat)

    # Pre-process Language Feature
    lang_feat = self.embedding(ques_ix)
    lang_feat, _ = self.lstm(lang_feat)

    # Pre-process Image Feature
    img_feat = self.img_feat_linear(img_feat)

    # Backbone Framework
    lang_feat, img_feat = self.backbone(
        lang_feat,
        img_feat,
        lang_feat_mask,
        img_feat_mask
    )

    lang_feat = self.attflat_lang(
        lang_feat,
        lang_feat_mask
    )

    img_feat = self.attflat_img(
        img_feat,
        img_feat_mask
    )

    proj_feat = lang_feat + img_feat
    proj_feat = self.proj_norm(proj_feat)
    proj_feat = torch.sigmoid(self.proj(proj_feat))

    return proj_feat
```

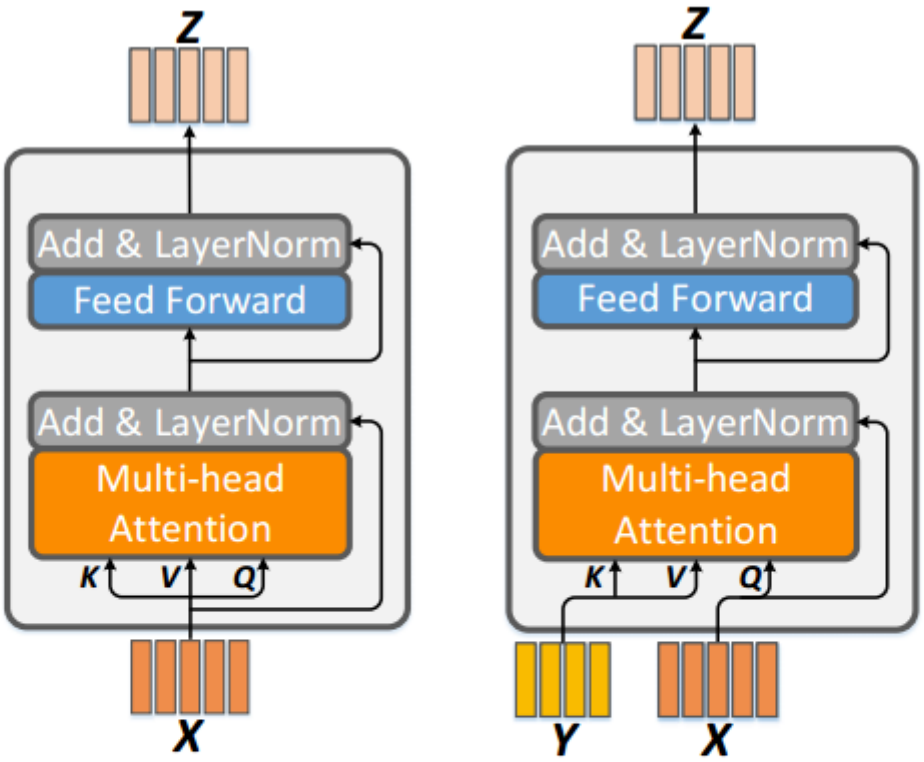


(a) ID(Y)-GA(X,Y) (b) SA(Y)-GA(X,Y) (c) SA(Y)-SGA(X,Y)

Figure 3: Flowcharts of three MCA variants for VQA. (Y) and (X) denote the question and image features respectively.

large, we usually have $d_h = d/h$. In practice, we can compute the attention function on a set of m queries $Q = [q_1; q_2; ...; q_m] \in \mathbb{R}^{m \times d}$ seamlessly by replacing q with Q in Eq.(2), to obtain the attended output features $F \in \mathbb{R}^{m \times d}$.

```
# MCA部分
def forward(self, x, y, x_mask, y_mask):
    # Get hidden vector
    for enc in self.enc_list:
        x = enc(x, x_mask)

    for dec in self.dec_list:
        y = dec(y, x, y_mask, x_mask)

    return x, y
```



(a) Self-Attention (SA) (b) Guided-Attention (GA)

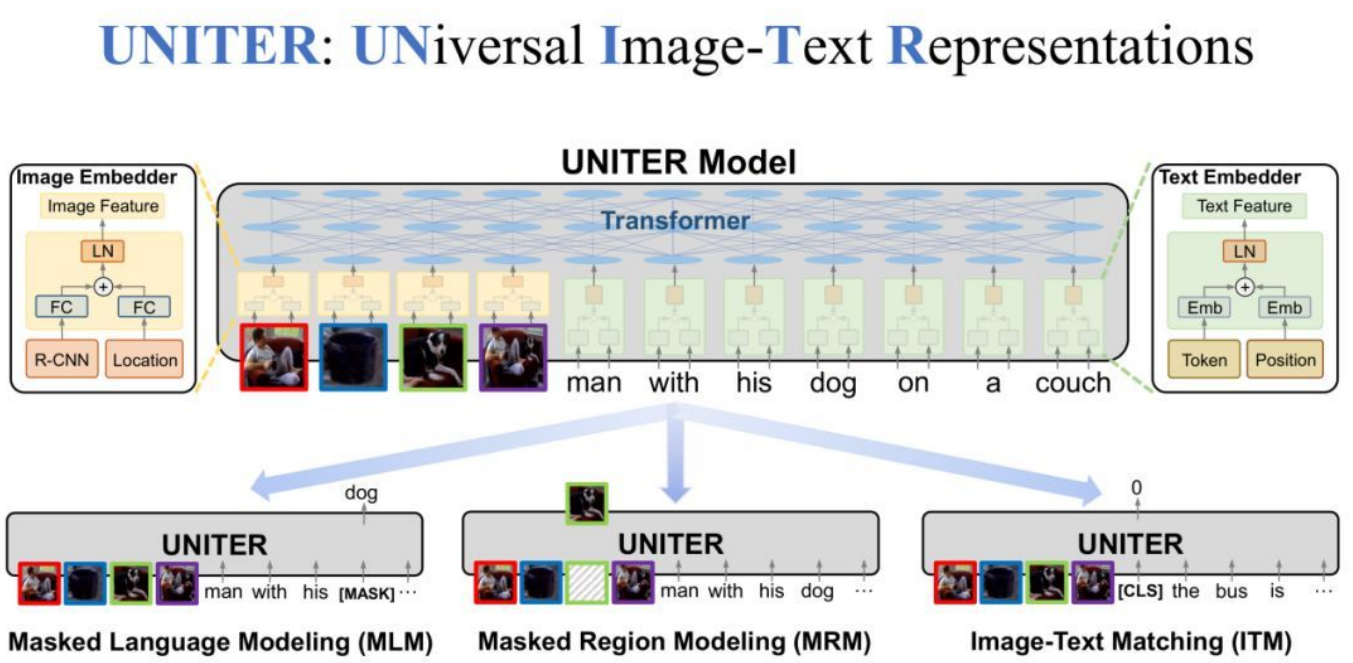
Figure 2: Two basic attention units with multi-head attention for different types of inputs. SA takes one group of input features X and output the attended features Z for X; GA takes two groups of input features X and Y and output the attended features Z for X guided by Y.

```
# SA
def forward(self, x, x_mask):
    x = self.norm1(x + self.dropout1(
        self.mhatt(x, x, x, x_mask)
    ))

    x = self.norm2(x + self.dropout2(
        self.ffn(x)
    ))

    return x
```

```
# GA
def forward(self, x, y, x_mask, y_mask):
    x = self.norm1(x + self.dropout1(
        self.mhatt1(x, x, x, x_mask)
    ))
```

```
        x = self.norm2(x + self.dropout2(
            self.mhatt2(y, y, x, y_mask)
        ))

        x = self.norm3(x + self.dropout3(
            self.ffn(x)
        ))

        return x
```

UNITER



UNITER 首先使用 Image Embedder 和 Text Embedder 将图像区域（视觉特征和边界框特征）和文本词（标记和位置）编码到一个公共嵌入空间中，然后应用一个 Transformer 模块，通过上述预训练任务为每个区域和单词来学习可泛化的上下文。与使用两个流的 LXMERT和 ViLBERT 相比，UNITER 模型可以通过单个 Transformer 学习图像区域和文本词的联合上下文化表示。此外，作者的掩码语言/区域建模是基于对于图像/文本的全面观察，这与将联合随机掩码应用于两种模态的其他并行预训练模型不同。作者表明条件掩码策略可以成功地缓解图像和文本之间的缺失对齐，并为下游任务获得更好的联合嵌入。

给定一对图像和句子，UNITER 将图像的视觉区域和句子的文本标记作为输入。作者设计了一个图像嵌入器和一个文本嵌入器来提取它们各自的嵌入。然后将这些嵌入输入到多层 self-attention Transformer 中，用以学习视觉区域和文本标记之间的跨模态上下文嵌入。Transformer 中的 self-attention 机制是无序的，因此需要将标记/区域的位置/位置显式编码为附加输入。

具体来说，在 Image Embedder 中，作者首先使用 Faster R-CNN 来提取每个区域的视觉特征。此外作者还通过一个 7 维向量对每个区域的位置特征进行编码。然后，视觉和位置特征都通过一个全连接（FC）层，用以投影到相同的嵌入空间中。每个区域的最终视觉嵌入是通过将两个 FC 输出相加，然后通过层归一化（LN）层来得到的。对于Text Embedder，作者将输入句子标记为 WordPieces。每个子词标记的最终表示是通过总结其词嵌入和位置嵌入获得的，然后接着是另一个 LN 层。

作者用了三种任务预训练模型：基于图像的MLM，基于文本的MRM和图像文本匹配（ITM）。如图 1 所示，作者的 MRM 和 MLM 类似于 BERT，其中从输入中随机屏蔽一些单词或区域，Transformer 的输出是学习恢复这些单词或区域 。具体来说，词掩码是通过用特殊的标记 [MASK] 替换标记来实现的，区域掩码是通过将视觉特征向量替换为全零来实现的。

需要注意的是作者每次只屏蔽一种模态，同时保持另一种模态完好无损，而不是像 ViLBERT 和 LXMERT 那样随机屏蔽两种模态。当一个被屏蔽的区域碰巧被一个屏蔽词描述时，这可以防止潜在的未对齐。作者表示通过条件屏蔽，他们的模型能够学习更好的嵌入。最后，作者还通过 ITM 学习了整个图像和句子之间的实例级对齐（而不是标记/区域级）。在训练期间，作者对正负图像句子对进行采样，并学习它们的匹配分数。

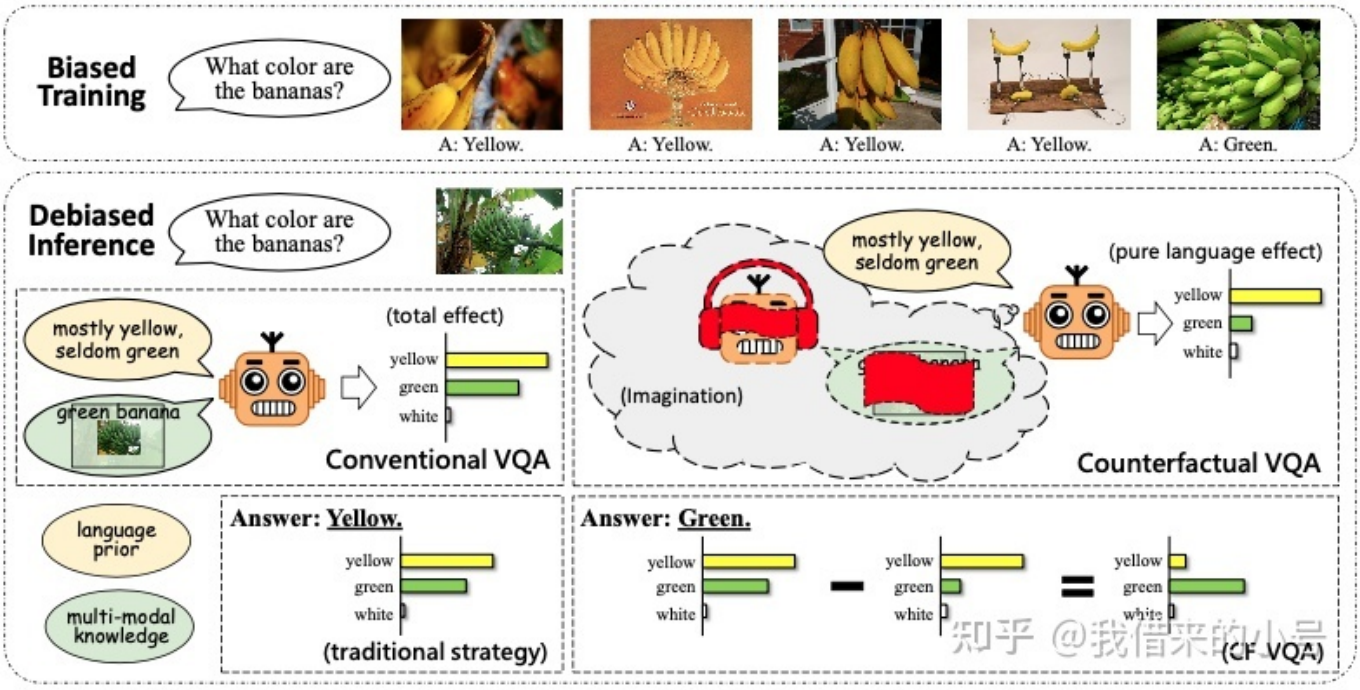
为了使用上述不同任务对 UNITER 进行预训练，作者为每个小批量随机采样一个预训练任务，并且每次 SGD 更新仅对一个目标进行训练。

Counterfactual VQA: A Cause-Effect Look at Language Bias

提供了分析先验语言知识的方法
可以在VQA模型中减少language bias的影响
提供新的思路：基于因果关系的分析方法

VQA模型的回答过程很有可能并不依赖于视觉和语言的混合推理，而是依赖于语言的相关性。举个例子，在VQA数据集中，和运动相关的问题仅仅回答‘tennis’的准确率为40%，以‘Did you see ... ’开头的问题光是回答‘yes’就可以达到90%的准确率。这是很不合理的，因为学习到过多的这种语言间的相关性会导致模型无法推广，当前的一大挑战是如何在有偏训练下进行无偏推理。

模型的主要思想就是从混合模态对答案的所有影响中，取出语言对答案的直接影响，保留视觉和语言对答案的混合影响，也就是说将language bias看作问题对答案的直接因果效应，从总因果效应中减去language的直接因果效应从而减少bias的影响。



这里有一个生动的例子来反映Figure1的过程，在Figure2中，假设有一个问题：这个香蕉的颜色是什么，训练集中的大部分的答案都是黄色。左边的是传统的VQA模型，右边的是本文介绍CF-VQA模型。左边的传统模型既有语言和视觉的单独影响也有混合推理的影响（这两部分的总和成为total effect），但是因为数据集的原因，语言的推理占比比较大，最终覆盖了图片的推理贡献，给出了‘yellow’这个答案。

但是右边的CF-VQA模型不同，它不光进行了传统VQA的推理，也进行了只有语言的推理，通过tota effect减去language effect去除了语言相关性对答案的影响，给出了正确答案green

强化学习-K摇臂赌博机

- 状态空间表示机器感知的环境的描述，动作空间表示机器能表示的动作，转移函数使环境从一个状态转移到下一个状态，奖赏函数反馈给机器的奖赏
- 强化学习某种意义上可看作具有延迟标记信息的监督学习问题
- 探索和利用相互矛盾
- 贪心法以e的概率探索，1-e概率利用。softmax类似

代码

```
# 测试argparse.ArgumentParser

# choices属性：内容必须为list有的值
# python pytorch_limu\test\t7.py --RUN train
import argparse
```

```
def f1():
    parser = argparse.ArgumentParser(description='test')
    parser.add_argument('--addresses',default="sipingroad", help = "The path of
address")

    parser.add_argument('--test',default="qwe", help = "The path of
address",dest='asd')

    parser.add_argument('--RUN', dest='RUN_MODE',
                        choices=['train', 'val', 'test'],
                        help='{train, val, test}',
                        type=str, required=True)

    args = parser.parse_args()
    return args

args=f1()
print(args.addresses)
print(args.asd)
print(args.RUN_MODE)
```

```
# 测试setattr函数
class A():
    qwe=1

a=A()
setattr(a, "qwe", 28)
print(a.qwe)
setattr(a, "asd", 12)
print(a.asd)
```

```
# 测试dir、startswith、isinstance、getattr、type
class A():
    q1=12
    q2='qwe'

a=A()
b=1
for t in dir(a):
    if not t.startswith('_'):
        print(t)
        if isinstance(getattr(a,t),type(b)):
            print(getattr(a, t))
```

```
# format函数
a = "qwe/{}asd.yml".format('zxc/')
```

```
print(a)
```

```
a={'qwe':12,'asd':21}
b={'zxc':32}
c={**a,**b}
print(c)
```

```
# ------------ Devices setup
os.environ['CUDA_VISIBLE_DEVICES'] = self.GPU
self.N_GPU = len(self.GPU.split(','))
self.DEVICES = [_ for _ in range(self.N_GPU)]
torch.set_num_threads(2)
```

```
# ------------ Seed setup
# fix pytorch seed
torch.manual_seed(self.SEED)
if self.N_GPU < 2:
    torch.cuda.manual_seed(self.SEED)
else:
    torch.cuda.manual_seed_all(self.SEED)
torch.backends.cudnn.deterministic = True

# fix numpy seed
np.random.seed(self.SEED)

# fix random seed
random.seed(self.SEED)
```

```
# 在训练模型时会在前面加上：
net.train()
# 在测试模型时在前面使用：
net.eval()
```