PA4 Report

20160463 성해빈

training and test source codes, result outputs, and trained models are in google drive.
https://drive.google.com/drive/folders/1Tld0qF_XVgFdlGEo-pms9d4_pmf-ICOD?usp=sharing

Basic Image Captioning

I used the repo https://github.com/Renovamen/Image-Captioning, model `show_tell` (NIC) for this basic image captioning model task.

The results are as follows:

	BLEU-1	BLEU-2	BLEU-3	BLEU-4
NIC	0.5585	0.5585	0.0523	0.0135

`a boy on skateboard down ramp a`

`a girl a`

`a dog in snow`

I tried training more epochs (the default was 20), but epoch 18 was my best performance model during 200 epochs, so it had no meaning training more than the default. I also tried training in pretrained word embeddings from glove (transfer learning), and using pretrained word embeddings had better performance.

I learned how basic Image Captioning models (NIC model) work. The CNN encoder extracts the feature of the image, and the RNN decoder decodes it to sequence of words. And I also learned what input and what output these models need and produce. I also learned what BLEU means, it is the metric of n-gram precision, so if every words are in the reference sentence, the unigram BLEU (BLEU-1) score is 1(100%).

The results barely are a full sentence, but they tend to successfully represent some characteristics of the image. The score of 55% BLEU-1 score is quite decent, comparing with the original paper that had 63% BLEU-1 score. So I don't see such poor training on the BLEU-1 score, but the samples are not generating good sentences, and that's what I'm disappointed on. This repo didn't focus on show&tell that much, since it had other models, so I think that's where the problem comes from, the default hyperparameters and architecture of NIC model in the repo is probably not that accurate to the NIC of the original paper.

Hard Attention Image Captioning

All repos given as the reference DID NOT contain implementations of hard attention. I tried implementing it, but I couldn't come up with a working code, since there are very few codes that implement hard attention, probably due to its complexity comparing to soft attention. https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Image-Captioning/issues?q=hard

	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Hard Attention with ResNet101	?	?	?	?
Hard Attention with VGG19	?	?	?	?

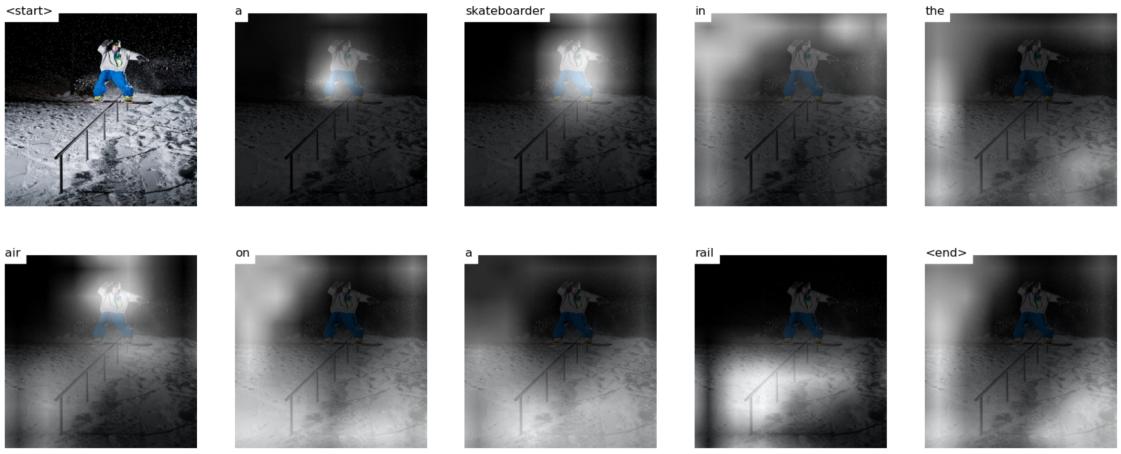
Though I failed in implementing it, I still got to study about the mechanism of hard attention in the process of trying to implement it. Hard attention tries to calculate the context vector by sampling from a multinoulli distribution, with weight as the sample rate to choose a feature. To calculate the gradient descent, we need to implement a Monte Carlo method, not simple back-propagation. This is why many codes do not implement hard attention, and this is why I failed in implementing it too. It does not use the standard back-propagation method in gradient descent, making it quite tricky to implement.

Soft Attention Image Captioning

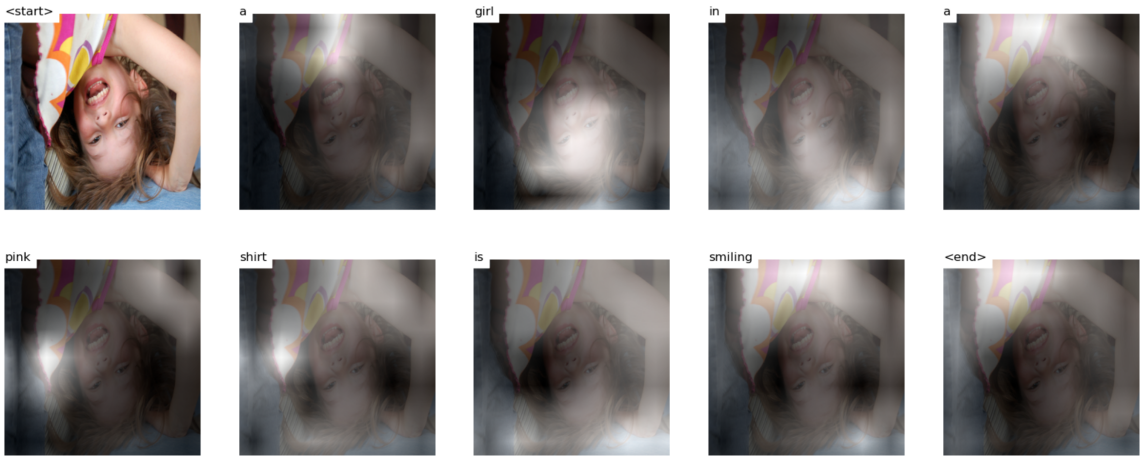
I used the repo https://github.com/Renovamen/Image-Captioning, model `att2all` (Soft Attention) for this hard attention image captioning model task.

The results are as follows:

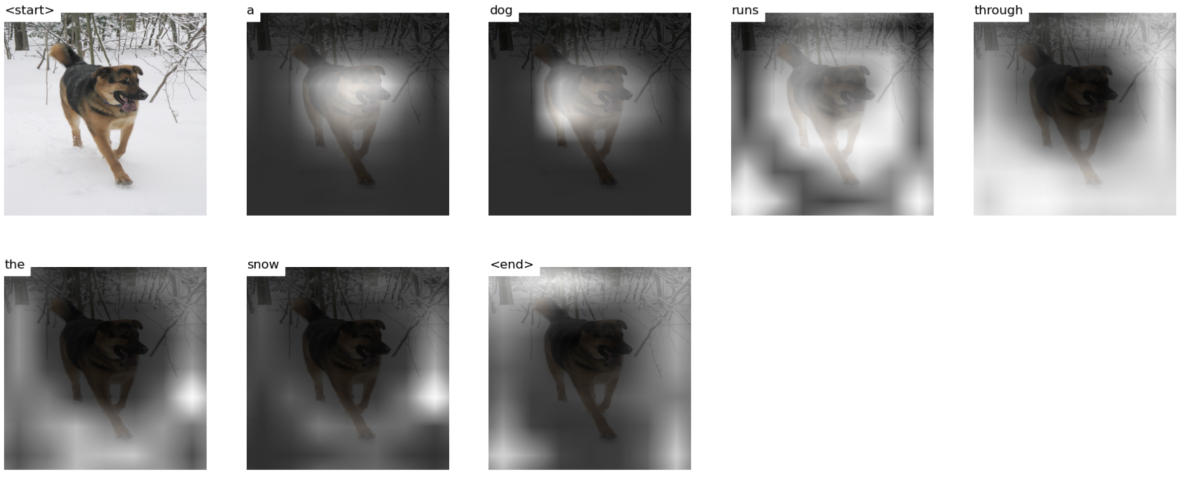
	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Soft Attention with ResNet101	0.6351	0.6351	0.3206	0.2242
Soft Attention with VGG19	?	?	?	?



`a skateboarder in the air on a rail`



`a girl in a pink shirt is smiling`



`a dog runs through the snow`

The results are astonishing! Compared to the basic image captioning model(NIC), though the code may be quite giving a disadvantage to NIC, the output captions of the samples are significantly better, judging by our human brains. The BLEU scores are better, BLEU-1 has 0.6351 score. The paper had 0.67 score, so it is almost close to the paper. Even though the BLEU score seems only 10% better, the sample results seem quite significantly better: which may mean the BLEU metric is not enough to express the naturalness of the sentence.

I learned the mechanism of attention, thanks to the very kind explanation in the repository https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Image-Captioning. This helped me know about the architecture a lot. Soft attention is implemented by calculating the context vector by the expectation value of features with a weight that sums up to 1. You could interpret it as computing the probability that a pixel is the place to look to generate the next word.

Another benefit of attention is that we can visualize what the model focuses on each word generation. We can see the attention and the words match quite well in the samples.

For the effort of implementing VGG, I modified the encoder code to add a vgg.py in encoders. I got VGG from pytorch with pretrained ImageNet, and I removed avgpool layers and fc layers from it, just like the original code did in resnet. But I got stuck when I got errors:

```
File "/home/haebin307/anaconda3/envs/hands/lib/python3.8/site-
packages/torch/nn/functional.py", line 1674, in linear
    ret = torch.addmm(bias, input, weight.t())
RuntimeError: mat1 dim 1 must match mat2 dim 0
```

This is probably due to a size mismatch between output of vgg and resnet, but I have no idea of how I could handle the size mismatch of this case. I tried to look at the issue of the repo (both 4 and 5), but there was NO VGG related issues, surprisingly : https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Image-Captioning/issues?q=VGG. This code is not the best skeleton for implementing other models, apparently. Since there was no one who wanted VGG in this repo, I couldn't proceed further. This code is my best shot of implementing VGG from this repo.

```
class VGG19(nn.Module):
    def __init__(self, encoded_image_size = 7):
        super(VGG19, self).__init__()
        self.enc_image_size = encoded_image_size # size of resized feature map

        # pretrained ResNet-101 model (on ImageNet)
        vgg = torchvision.models.vgg19(pretrained = True)

        print(vgg)

        # we need the feature map of the last conv layer,
        # so we remove the last two layers of resnet (average pool and fc)
        modules = list(vgg.children())[:-2]
        print(modules)
        self.vgg = nn.Sequential(*modules)

        # resize input images with different size to fixed size
        self.adaptive_pool = nn.AdaptiveAvgPool2d((encoded_image_size,
encoded_image_size))

        #self.fine_tune()
```

```
    '''
    input params:
        images: input image(batch_size, 3, image_size = 256, image_size = 256)
    return:
        feature_map: feature map after resized (batch_size, 2048,
encoded_image_size = 7, encoded_image_size = 7)
    '''
    def forward(self, images):
        feature_map = self.vgg(images) # (batch_size, 2048, image_size/32,
image_size/32)
        feature_map = self.adaptive_pool(feature_map) # (batch_size, 2048,
encoded_image_size = 7, encoded_image_size = 7)
        return feature_map

    '''
    input params:
        fine_tune: fine-tune CNN (conv block 2-4) or not
    '''
    def fine_tune(self, fine_tune = True):
        for param in self.vgg.parameters():
            param.requires_grad = False
        # only fine-tune conv block 2-4
        for module in list(self.vgg.children())[5:]:
            for param in module.parameters():
                param.requires_grad = fine_tune



class AttentionEncoderVGG(nn.Module):
    def __init__(self, encoded_image_size = 7):
        super(AttentionEncoderVGG, self).__init__()
        self.CNN = VGG19(encoded_image_size)

    '''
    input params:
        images: input image (batch_size, 3, image_size = 256, image_size = 256)
    return:
        feature_map: feature map of the image (batch_size, num_pixels = 49,
encoder_dim = 2048)
    '''
    def forward(self, images):
        feature_map = self.CNN(images)  # (batch_size, 2048, encoded_image_size
= 7, encoded_image_size = 7)
        feature_map = feature_map.permute(0, 2, 3, 1)  # (batch_size,
encoded_image_size = 7, encoded_image_size = 7, 2048)

        batch_size = feature_map.size(0)
        encoder_dim = feature_map.size(-1)
        num_pixels = feature_map.size(1) * feature_map.size(2) #
encoded_image_size * encoded_image_size = 49

        # flatten image
        feature_map = feature_map.view(batch_size, num_pixels, encoder_dim) #
(batch_size, num_pixels = 49, encoder_dim = 2048)

        return feature_map
```