VGGnet

20176359 신수현

VGGnet에서는 무엇을 다르게 하였나?

- 이전 AlexNet(2012)의 8 layers 모델보다 2배 이상 깊은 네트워크 학습에 성공한 VGGnet

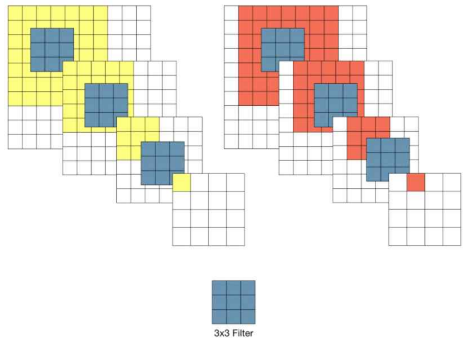
- VGG 모델이 16-19 layer에 달하는 깊은 신경망을 학습할 수 있었던 것은 모든 합성곱 레이어에서 3x3 필터를 사용했지 때문이다.

VGGnet의 강점

- VGGNet에서는 3x3 크기의 필터를 사용함으로써, 여러개의 ReLU non-linearity의 사용을 증가시킬 수 있다. 그리고 기존의 7x7 사이즈보다 작은 크기의 필터를 사용함으로써 상당수의 파라미터를 줄일 수 있었다.

※ 결정 함수의 비선형성 증가?

각 Convolution 연산은 ReLU 함수를 포함한다. 다시 말해, 1-layer 7x7 필터링의 경우 한 번의 비선형 함수가 적용되는 반면 3-layer 3x3 필터링은 세 번의 비선형 함수가 적용된다. 따라서, 레이어가 증가함에 따라 비선형성이 증가하게 되고 이것은 모델의 특징 식별성 증가로 이어진다. (Stride가 1일 때, 3차례의 3x3 Conv 필터링을 반복한 특징맵은 한 픽셀이 원본 이미지의 7x7 Receptive field의 효과를 볼 수 있다.)



VGGnet의 구성

- 13 Convolution Layers + 3 Fully-connected Layers
- 3x3 convolution filters
- stride : 1 & padding: 1
- 2x2 max pooling (stride : 2)
- ReLU

```
def _make_layers(self, cfg):
    layers = []
    in_channels = 3
    for x in cfg:
        if x == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        else:
            layers += [nn.Conv2d(in_channels, x, kernel_size=3, padding=1),
                       nn.BatchNorm2d(x),
                       nn.ReLU(inplace=True)]
            in_channels = x
    layers += [nn.AvgPool2d(kernel_size=1, stride=1)]
    return nn.Sequential(*layers)
```

```
cfg = {
    'VGG11': [64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    'VGG13': [64, 64, 'M', 128, 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'],
    'VGG16': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M'],
    'VGG19': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', 512, 512, 512, 512, 'M', 512, 512, 512, 512, 'M'],
}
```

학습

학습 시 다음과 같은 최적화 알고리즘을 사용하였다

- Momentum(0.9)
- Weight Decay(L2 Norm)
- Dropout(0.5)
- Learning rate 0.01로 초기화 후 서서히 줄임

```
class VGG(nn.Module):
    def __init__(self, vgg_name):
        super(VGG, self).__init__()
        self.features = self._make_layers(cfg[vgg_name])

        #self.classifier = nn.Linear(512, 10)

        self.classifier = nn.Sequential(
            nn.Dropout(0.5),
            nn.Linear(512, 512),
            nn.ReLU(True),
            nn.Dropout(0.5),
            nn.Linear(512, 512),
            nn.ReLU(True),
            nn.Linear(512, 10),
        )
```

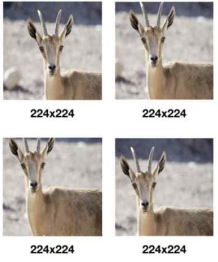
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01,
                      momentum=0.9, weight_decay=5e-4)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=200)
```

학습 이미지 크기

- VGGNet에서는 training scale을 'S'로 표시하며, single-scale training과 multi-scaling training을 지원한다. Single scale에서는 AlexNet과 마찬가지로 S = 256, 또는 S = 384 두개의 scale 고정을 지원한다.

- Multi-scale의 경우는 S를 Smin과 Smax 범위에서 무작위로 선택할 수 있게 하였으며, Smin은 256이고 Smax는 512이다. 즉, 256과 512 범위에서 무작위로 scale을 정할 수 있기 때문에 다양한 크기에 대한 대응이 가능하여 정확도가 올라간다. Multi-scale 학습은 S = 384로 미리 학습 시킨 후 S를 무작위로 선택해가며 fine tuning을 한다. S를 무작위로 바꿔 가면서 학습을 시킨다고 하여, 이것을 scale jittering이라고 하였다.


256x256


224x224 224x224 224x224 224x224

이미지를 256x256 크기로 변환 후
224x224 크기를 샘플링한 경우


512x512


224x224 224x224 224x224 224x224

이미지를 512x512 크기로 변환 후
224x224 크기를 샘플링한 경우

학습 이미지

- 이처럼 학습 데이터를 다양한 크기로 변환하고 그 중 일부분을 샘플링해 사용함으로써 몇 가지 효과를 얻을 수 있다.

- 한정적인 데이터의 수를 늘릴 수 있다. — Data augmentation 하나의 오브젝트에 대한 다양한 측면을 학습 시 반영시킬 수 있다. 변환된 이미지가 작을수록 개체의 전체적인 측면을 학습할 수 있고, 변환된 이미지가 클수록 개체의 특정한 부분을 학습에 반영할 수 있다. 두 가지 모두 Overfitting을 방지하는 데 도움이 된다.

```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
```

trainset에 대해서 mean과 std 이용하여 Normalize

참고 자료

- chengyangfu github, “vgg.py 코드 참고”, https://github.com/chengyangfu/pytorch-vgg-cifar10/blob/master/vgg.py, (2020.09.18)
- kuangliu github, “vgg.py, main.py 코드 참고”, https://github.com/kuangliu/pytorch-cifar, (2020.09.18)
- 적자 생존의 법칙, “VGGnet 코드” , https://blogofth-lee.tistory.com/265, (2020.09.18)