BesNet

Anonymous CVPR submission

Paper ID *****

Abstract

도로 영역을 찾는 세그멘테이션 모델 두가지(*U-Net*과 *UNet++*)외 *UNet+++(2020)*모델을 통해 세그멘테이션 .

1. Introduction

UNet++(2018)은 U-Net과 다르게 re-designed skip pathway를 설계하여 Encoder와 Decoder feature map 사이 semantic gap을 줄여 optimzer가 더 쉽고 빠르게 학습되는 특징을 갖고 있습니다. 그러나 UNet++에서 수행되는 Dense convloution에서는 Full scale(low-level의 detail을 다른 scale에 있는 feature map으로부터 high-level semantic과 합치는 것을 의미)로부터 충분한 정보를 얻기 어려워 Full scale의 Skip connection과 Deep supervision을 가진 UNet+++(이하 UNet3+)도입하였습니다. UNet3+는 parameter의 수도 줄일 수 있는 장점이 있습니다.

2. Work

Going Deeper (CV) RS4의 "14. 도로 영역을 찾자! - 세그멘테이션 모델 만들기"를 기반으로 데이터셋 준비, 비교 후보 모델(U-Net, UNet++), 시멘틱 세그멘테이션 모델 시각화를 수행하였습니다.

2.1. Preparing datasets

실험에 사용한 이미지 데이터셋은 KITTI데이터셋의 세그멘테이션 데이터를 사용하였습니다.

Figure 1. KITTI데이터셋.

Figure 2. KITTI데이터셋 Segmentation.

Figure 3. KITTI데이터셋 Segmentation RGB.

2.2. Build UNet3+ Model

네트워크가 세대를 거듭해가며 Skip connection에 변화가 생겼다. UNet의 Plain skip connection과 UNet++의 Nsted and dense skip connection은 full scale의 정보가 충분하지 않아 객체의 위치와 경계를 명확하게 학습하는데 어려움이 있었다.

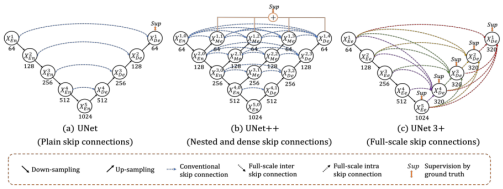

Figure 4. UNet(a), UNet++(b), UNet3+(c) 비교.

이런 단점을 보완한 UNet3+는 Encoder로부터 더 작거나 같은 scale의 feature map과 decoder로부터 더 큰 scale의 featur maps을 합쳐 극복하였다.

2.2.1 Encoder blocks

먼저, Encoder blocks는 **Figure 5**과 같이 구현한다.

"Convolution" -> "Batch Normalization" -> "Activation Gelu"를 2회 반복합니다. 이때 사용한 GELU 함수는 dropout, zoneout, ReLU 함수의 특성을 조합하여 유

```python
def encoder_block(inputs, n_filters, kernel_size, strides):
    encoder = tf.keras.layers.Conv2D(filters=n_filters,
                                     kernel_size=kernel_size,
                                     strides=strides,
                                     padding='same',
                                     use_bias=False)(inputs)
    encoder = tf.keras.layers.BatchNormalization()(encoder)
    encoder = tf.keras.layers.Activation(tf.keras.activations.gelu)(encoder)
    encoder = tf.keras.layers.Conv2D(filters=n_filters,
                                     kernel_size=kernel_size,
                                     padding='same',
                                     use_bias=False)(encoder)
    encoder = tf.keras.layers.BatchNormalization()(encoder)
    encoder = tf.keras.layers.Activation(tf.keras.activations.gelu)(encoder)
    return encoder
```

Figure 5. Encoder blocks 구현.

도되었습니다. GELU 함수는 **1) 모든 점에서 미분 가능**하고 **2) 단조증가함수가 아니므로** 비선형 활성화 함수 목적에 맞게 더욱 복잡한 전체 함수를 모델링하는데 도움이 된다고 한다.

2.2.2 Upscale blocks

```python
def upscale_blocks(inputs):
    n_upscales = len(inputs)
    upscale_layers = []

    for i, inp in enumerate(inputs):
        p = n_upscales - i
        u = tf.keras.layers.Conv2DTranspose(filters=64,
                                            kernel_size=3,
                                            strides=2**p,
                                            padding='same')(inp)

        for i in range(2):
            u = tf.keras.layers.Conv2D(filters=64,
                                       kernel_size=3,
                                       padding='same',
                                       use_bias=False)(u)
            u = tf.keras.layers.BatchNormalization()(u)
            u = tf.keras.layers.Activation(tf.keras.activations.gelu)(u)
            u = tf.keras.layers.Dropout(rate=0.4)(u)

        upscale_layers.append(u)
    return upscale_layers
```

Figure 6. Upscale blocks 구현.

모든 Decoder block에 Skip connections(Full-Scale)을 수행하기 위한 blocks를 구현한다.

2.2.3 Decoder blocks

```python
def decoder_block(layers_to_upscale, inputs):
    upscaled_layers = upscale_blocks(layers_to_upscale)

    decoder_blocks = []

    for i, inp in enumerate(inputs):
        d = tf.keras.layers.Conv2D(filters=64,
                                   kernel_size=3,
                                   strides=2**i,
                                   padding='same',
                                   use_bias=False)(inp)
        d = tf.keras.layers.BatchNormalization()(d)
        d = tf.keras.layers.Activation(tf.keras.activations.gelu)(d)
        d = tf.keras.layers.Conv2D(filters=64,
                                   kernel_size=3,
                                   padding='same',
                                   use_bias=False)(d)
        d = tf.keras.layers.BatchNormalization()(d)
        d = tf.keras.layers.Activation(tf.keras.activations.gelu)(d)

        decoder_blocks.append(d)

    decoder = tf.keras.layers.concatenate(upscaled_layers + decoder_blocks)
    decoder = tf.keras.layers.Conv2D(filters=256,
                                     kernel_size=3,
                                     strides=1,
                                     padding='same',
                                     use_bias=False)(decoder)
    decoder = tf.keras.layers.BatchNormalization()(decoder)
    decoder = tf.keras.layers.Activation(tf.keras.activations.gelu)(decoder)
    decoder = tf.keras.layers.Dropout(rate=0.4)(decoder)

    return decoder
```

Figure 7. Decoder blocks 구현.

마지막으로 원본 사이즈까지 커지는 Decoder block을 구현합니다.

2.2.4 Building model

```python
def get_model(input_dim):
    inputs = tf.keras.layers.Input(input_dim)

    # noisy_inputs = tf.keras.layers.GaussianNoise(stddev=0.2)(inputs)

    e1 = encoder_block(inputs, n_filters=32, kernel_size=3, strides=1)
    e2 = encoder_block(e1, n_filters=64, kernel_size=3, strides=2)
    e3 = encoder_block(e2, n_filters=128, kernel_size=3, strides=2)
    e4 = encoder_block(e3, n_filters=256, kernel_size=3, strides=2)
    e5 = encoder_block(e4, n_filters=512, kernel_size=3, strides=2)

    d4 = decoder_block(layers_to_upscale=[e5], inputs=[e4, e3, e2, e1])
    d3 = decoder_block(layers_to_upscale=[e5, d4], inputs=[e3, e2, e1])
    d2 = decoder_block(layers_to_upscale=[e5, d4, d3], inputs=[e2, e1])
    d1 = decoder_block(layers_to_upscale=[e5, d4, d3, d2], inputs=[e1])

    output = tf.keras.layers.Conv2D(filters=1,
                                    kernel_size=1,
                                    padding='same',
                                    activation='tanh')(d1)

    model = tf.keras.models.Model(inputs, output)
    return model
```

Figure 8. UNet3+ 네트워크 구현.

이로써 UNet3+ 네트워크 생성이 완성 됩니다.

2.3. Model Training

$$optimziers = Yogi \tag{1}$$

모델 최적화를 위해 Tensorflow addons의 Yogi 알고리즘을 사용합니다. https://www.tensorflow.org/addons/api_docs/python/tfa/optimizers/Yogi.

$$loss = sigmoidfocalcrossentropy \tag{2}$$

loss 함수도 Unet3+를 보면 focal loss라는 특이한 Loss를 사용하여 똑같이 사용하였다.

3. Result

아쉽게도 Training 과정에서 처음부터 높은 Accuracy가 나오는 등 문제가 발생하고 유의미한 결과를 위해서는 학습에 오랜 시간이 소요되는 등 모델 구현 및 학습이 수행됨까지 확인하였다. 초기 U-Net과 UNet++보다 좋은 성능이 나옴을 입증하고 싶었지만 이는 추가로 공부하여 추후 업데이트를 희망한다.

3.1. Learning curve

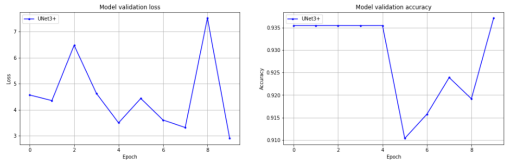

Figure 9. Learning curve.

중간 시간이 부족해 학습을 끊는 바람에 Learning curve와 결과물 모두 만족스럽게 나오진 못했다. 그러나 그래프를 통해 확인할 수 있는 점은 처음부터 높은 Validation accuracy에 반해 낮은 Test data에 대한 IoU 값이다.

3.2. IoU

여러 실패과정을 기록하였다면 지금 결과가 긍정적인 방향임을 보일텐데 또 시간이 더 충분하다면 괜찮은 결과를 볼 수 있어보인다. 기록을 잘 해야겠다.

Figure 10. evaluate model