Face Transformer를 통한 Face Parsing

&

Reinforce Learning

이재형

국민대학교 전자공학부

harry8121@kookmin.ac.kr

요 약

Face Transformer를 구현하고, Reinforce Learning을 통해 Face Parsing을 실현해보았습니다.

Ⅰ. 서론

Face parsing task는 Segmentation을 Face에 적용하여 전경과 배경을 구분하는 행위입니다.
전경에는 눈, 코, 입 등등이 있으며 Semantic Segmentation에서 파생된 Task입니다.
저는 Face parsing task의 성능을 높이기 위한 방법론에 집중하였습니다.
일반적인 CNN을 사용하지 않고 Face Transformer를 baseline model로 선정하였습니다.
Reinforce Learning의 Reward 개념을 학습에 추가하여 일반적인 Semantci Segmentation과 성능을 비교해보았습니다.
'데이터셋 선정 - Baseline Model 구현 - 학습 - 학습 결과' 순으로 작성하였습니다.

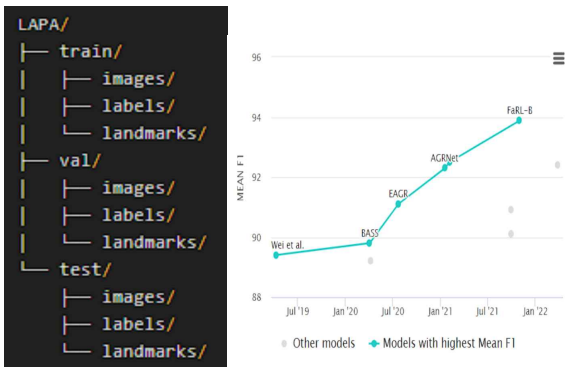
2. 수행 내용

2.1 LaPa dataset

LaPa 데이터셋[1]은 22,000개 이상의 고해상도 얼굴 이미지로 구성되어 있습니다. 각 이미지에는 11개의 얼굴 부위에 대한 정밀한 마스크 label이 포함되어 있습니다. 이는 Face Parsing 및 얼굴 속성 인식 연구에 매우 유용합니다.

LaPa 데이터셋은 아래와 같은 파일과 디렉토리로 구성되어 있습니다.

1. 이미지 파일
2. 마스크 파일
3. 속성 파일(각 이미지에 대한 속성 주석 파일)



데이터셋 구성 & Face Parsing on LaPa

2.2 Face Transformer 구현

다음은 Baseline Model로 사용한 Face Transformer의 구현입니다. Face Transformer의 주요 구성 요소와 그 역할을 차례로 서술하겠습니다.

1. Patch Embedding
: 이미지를 패치로 나누고 각 패치를 임베딩 벡터로 변환합니다.

```
class ViT(nn.Module):
    def __init__(self, image_size, patch_size, num_classes, dim, depth, heads, mlp_dim, dropout, emb_dropout):
        super(ViT, self).__init__()

        assert image_size % patch_size == 0
        num_patches = (image_size // patch_size) ** 2
        patch_dim = 3 * patch_size * patch_size

        self.conv = nn.Conv2d(3, dim, kernel_size=patch_size, stride=patch_size)
```

- **patch_size** : 패치의 크기
- **num_patches** : 이미지를 나눈 패치의 개수
- **self.conv :** con2D레이어를 사용하여 이미지를 패치로 나누고, 각 패치를 임베딩 벡터로 변환

2. Transformer Encoder

: 패치 임베딩 벡터를 Transformer 인코더에 입력하여 각 패치의 특징을 추출합니다. 이를 위해서 여러 Transformer Encoder 레이어를 사용합니다.

```
self.transformer = Transformer(dim, depth, heads, mlp_dim, dropo
```

```
class Transformer(nn.Module):
    def __init__(self, dim, depth, heads, mlp_dim, dropout):
        super(Transformer, self).__init__()
        self.layers = nn.ModuleList([nn.TransformerEncoderLay
        self.norm = nn.LayerNorm(dim)

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        return self.norm(x)
```

- **dim** : 패치 임베딩 벡터의 차원
- **depth** : Transformer 인코더 블록의 깊이(레이어 수)
- **heads** : Multi-head Attention의 헤드 수
- **mlp_dim** : Feedforward 네트워크의 숨겨진 차원
- **dropout** : 드롭아웃의 비율
- **self.transformer** : Transformer Encoder 블록을 정의

3. 패치 임베딩 재구성 및 최종 출력

: Transformer Encoder를 통해 얻은 출력을 원래 이미지 형태로 재구성하고, 최종적으로 Segmentation을 수행합니다.

```
self.to_patch_embedding = nn.Conv2d(dim, dim, kernel_size=1)
self.deconv = nn.ConvTranspose2d(dim, num_classes, kernel_size=patch_size, stride=patch_
self.dropout = nn.Dropout(dropout)
```

- **self.to_patch_embedding** : Transformer인코더의 출력을 다시 이미지 형태로 변환하기 위한 Conv2D레이어.
- **self.deconv** : 최종 출력을 위한 ConvTranspose2D레이어. 이 레이어는 임베딩 벡터를 다시 원래 이미지 크기로 변환
- **self.dropout** : 드롭아웃 레이어, 과적합을 방지

```
def forward(self, x):
    x = self.conv(x)  # [batch_size, dim, h', w']
    x = x.flatten(2).transpose(1, 2)  # [batch_size, h'*w', dim]
    x = self.transformer(x)  # [batch_size, h'*w', dim]
    x = x.transpose(1, 2).view(x.size(0), -1, int(x.size(1)**0.5), int(x.size(1)**
    x = self.to_patch_embedding(x)  # [batch_size, dim, h', w']
    x = self.deconv(x)  # [batch_size, num_classes, H, W]
    return x
```

최종적으로 ViT모델의 'forward' 메서드를 통해 위의 단계를 실행합니다. 이 모델은 입력 얼굴 이미지를 패치로 나누고, Transformer Encoder를 통해 특징을 추출한 후, 최종저긍로 얼굴의 다양한 부위를 Segmentation하거나 속성을 인식하는 데 사용됩니다.

2.3 학습

이전 단계와 같이 Face Transformer를 구현하고, 그것을 이용하여 학습을 진행하였습니다. 학습 또한 3단계로 분류하여 서술하였습니다.

1. 모델 정의 및 초기화

: SemanticSegmentationModel클래스는 PyTorch Lightning을 사용하여 정의한 ViT 기반의 Segmentation 모델입니다.

```
class SemanticSegmentationModel(LightningModule):
    def __init__(
        self,
        image_size: int = 256,
        patch_size: int = 32,
        num_classes: int = 11,
        dim: int = 1024,
        depth: int = 6,
        heads: int = 16,
        mlp_dim: int = 2048,
        dropout: float = 0.1,
        emb_dropout: float = 0.1,
        optimizer: str = "adam",
        lr: float = 1e-4,
        betas: Tuple[float, float] = (0.9, 0.999),
        momentum: float = 0.9,
        weight_decay: float = 0.0,
        scheduler: str = "none",
        warmup_steps: int = 0,
        weights: Optional[str] = None,
        prefix: str = "net",
    ):
```

- **'__init__'** : 모델의 하이퍼파라미터와 네트워크 구조를 설정합니다.

2. 학습 및 검증 단계

: shared_step, training_step, validation_step, test_step 이렇게 총 4개의 메서드를 통해 각각 학습, 검증, 테스트 단계에서 수행할 작업을 정의하였습니다.

- **'shared_step'** : 학습, 검증, 테스트 단계에서 공통으로 수행되는 작업을 정의합니다. 입력 데이터와 타겟 데이터를 가져와 모델에 통과시키고, 손실을 계산하며, 메트릭(Dice)을 기록합니다.

```
def shared_step(self, batch, mode="train"):
    x, y = batch
    device = self.device if self.device else 'cpu'
    x, y = x.to(device), y.to(device)

    pred = self.forward(x)

    y = y.squeeze(1).long()  # [N, 1, H, W] -> [N, H, W]

    assert y.min() >= 0, f"Invalid target value: {y.min()}"

    loss = torch.nn.functional.cross_entropy(pred, y)

    metrics = getattr(self, f"{mode}_metrics")(pred, y)

    self.log(f"{mode}_loss", loss, on_epoch=True, prog_bar=True)
    for k, v in metrics.items():
        self.log(f"{mode}_{k.lower()}", v, on_epoch=True)

    return loss
```

위는 일반적인 segmentation을 수행하는 shared_step 함수입니다.

이와 다르게 Reinforce Learning의 Reward 기반의 segmentation을 수행하는 다른 버전의 shared_step 함수는 아래와 같습니다.

```python
def shared_step(self, batch, mode="train"):
    x, y = batch
    device = self.device if self.device else 'cpu'
    x, y = x.to(device), y.to(device)

    pred = self.forward(x)

    # 크기 맞추기: [N, num_classes, H, W] -> [N, H, W]
    pred = torch.argmax(pred, dim=1)

    # 크기 변환 확인
    y_flat = y.flatten()
    pred_flat = pred.flatten()

    # F1 점수 계산
    reward = f1_score(y_flat.cpu(), pred_flat.cpu(), average='micro')

    # Bernoulli 로프 샘플
    dist = torch.distributions.Bernoulli(logits=pred.float())
    sample = dist.sample()
    log_prob = dist.log_prob(sample)

    # 로스 계산
    loss = -torch.mean(log_prob * reward)

    # Ensure loss requires gradient
    if not loss.requires_grad:
        loss.requires_grad = True

    # 메트릭 계산
    metrics = getattr(self, f"{mode}_metrics")(pred, y.long())

    self.log(f"{mode}_loss", loss, on_epoch=True, prog_bar=True)
    for k, v in metrics.items():
        self.log(f"{mode}_{k.lower()}", v, on_epoch=True)

    return loss
```

- 위는 f1_score를 사용하여 flatten된 타겟 데이터와 예측 결과의 F1 score를 계산합니다. 이를 통해 reward를 기반으로 하여 손실을 측정합니다.

```python
def training_step(self, batch, _):
    self.log("lr", self.trainer.optimizers[0].param_groups[0]["lr"], prog_bar=T
    return self.shared_step(batch, "train")

def validation_step(self, batch, _):
    return self.shared_step(batch, "val")

def test_step(self, batch, _):
    return self.shared_step(batch, "test")
```

- 위의 3개의 메서드는 각각 학습, 검증, 테스트 단계에서 호출되어 작업을 수행합니다.

3. 옵티마이저 및 학습 설정

: configure_optimizers 메서드를 통해 옵티마이저를 설정하고, 학습 설정을 위한 Trainer를 정의합니다.

```python
def configure_optimizers(self):
    optimizer = torch.optim.Adam(self.parameters(), lr=self.lr, betas=self.betas, weight_decay=self.weight_de
    return optimizer
```

- **'configure_optimizers'** : Adam 옵티마이저를 설정하고, 이를 통해 모델의 파라미터를 업데이트 합니다.

위의 클래스와 메서드를 통해 학습은 아래와 같은 과정으로 진행됩니다.

<데이터 디렉토리 & 배치 크기 설정>

<데이터 모듈 생성>

<체크포인트 경로 설정>

<모델 생성>

<로그 설정>

<콜백 설정>

<트레이너 생성>

<학습 시작>

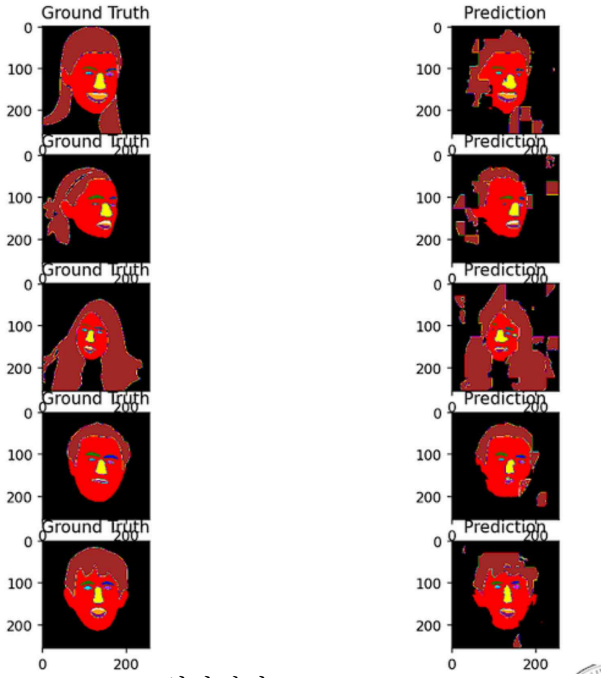
<모델 평가>

<평가 후 시각화>

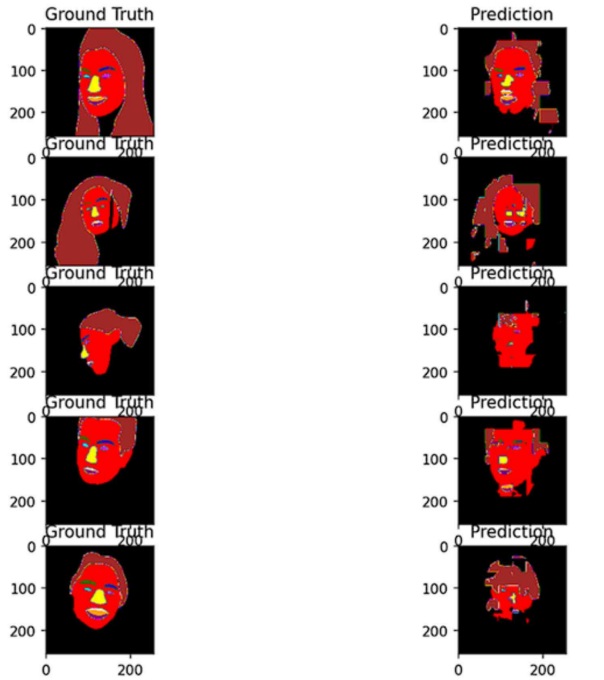
결과적으로 두가지 버전의 shared_step을 따로 구현하여 학습을 진행하였습니다.
즉, 일반적인 cross-entropy를 손실함수로 설정하여 학습한 segmentation결과와 강화 학습의 reward를 기반으로 학습한 segmentation 결과의 성능을 비교해 보았습니다.

(하이퍼파라미터, 메트릭, 시각화 함수, 체크포인트 등의 자세한 내용은 소스코드를 통해 확인하실 수 있습니다)

3. 실험 결과 및 분석



일반적인 Segmentation



reward기반 Segmentation

- 위는 랜덤으로 5개의 test data의 GT와 모델의 prediction을 시각화한 이미지 입니다.

- 손실함수를 제외한 모든 부분에서 동일한 조건(하이퍼 파라미터, 옵티마이저 등등)으로 학습한 결과입니다.

- 정성적으로 평가해보았을 때, Cross-Entropy를 손실함수로 사용한 segmentation 결과의 성능이 뛰어나다는 것을 확인할 수 있었습니다.

- Dice 메트릭을 통해 Face Parsing 성능을 정량적으로 평가하기위하여, tensorboard를 통해 시각화하려 시도했으나, tensorboard와 연결되지 않는 오류를 해결하지 못하여 정량적인 평가를 진행하지 못한 점 죄송합니다. (소스코드에는 Dice 메트릭을 계산하는 부분은 알맞게 구현되었으나 그 메트릭을 시각화하고 확인하는 부분에서 발생한 오류를 해결하지 못했습니다)

- Reward 기반 학습 방법이 성능이 나오지 않는 이유를 우선적으로 Face Parsing에 맞지 않는 reward를 설정했다고 생각하여, 위의 reward 함수를 제외한 2가지 정도로 더 시도하여 같은 조건하에 학습을 진행하였으나, 큰 성능 향상을 보이지 못했습니다.

<Reward 기반 학습 방법의 문제점>

- Reward 기반 손실 : 예측의 정확도에 따라 보상을 계산하고 손실을 설정한 방법 자체는 간접적이며, 보상의 변동성과 불확실성에 의하여 최적화가 더 어려울 수 있습니다.

- 보상 설계의 문제 : F1 점수 기반의 보상이 최적의 학습 신호를 제공하지 않을 수 있다. 이는 보상이 예측의 세부적인 차이를 충분히 반영하지 못하기 때문입니다.

4. 결론

Reward 기반의 학습 방법 자체의 아이디어는 나쁘지는 않지만, 단순하게 설정된 reward는 모델의 예측의 디테일한 부분을 반영하지 못한다는 점을 크게 깨달았습니다.

오히려 기존의 모델에서 파인 튜닝을 할 때 reward 기반의 학습이 더 효과적일 수 있겠다는 점을 생각할 수 있었고, 특정 task에 맞는 정교한 보상 신호를 설계한다는 점은 쉽지 않은 것임을 깨달았습니다.

또한, 학습 진행 간에 하이퍼파라미터를 수정하거나 모델의 복잡도를 높이는 등의 시도를 함으로써 모델 학습과 평가 과정에 좀 더 능숙해지고 성능 향상을 위한 다양한 방법에 대해 알 수 있는 기회였습니다.

참고문헌

[1] https://paperswithcode.com/dataset/lapa