Human Protein Atlas Image Classification

1. A CNN Classifier and a Metric Learning Model, 1st Place Solution

💡 **Keypoint**: Loss Functions(FocalLoss+Lovasz, ArcFaceLoss)과 Metric Learning을 중심적으로

▼ Challenges

- 불균형이 심한 데이터, 클래스가 희귀해 모델 학습과 훈련에 어려움
- 데이터의 분포가 훈련, 테스트, HPA v 18에서 일관되지 않음
- 고화질의 이미지 → 모델의 효율성과 정확성 사이의 균형

▼ Validation for CNNs

- 전체 val 셋에 대해 F1 score보다 Focal loss 가 더 좋은 메트릭
  - F1은 train과 val set의 분포에 따라 달라지는 threshold에 민감.
- 각 클래스의 비율을 train ,set과 동일하게 설정하여 모델을 평가.

▼ Loss Functions

- batch size가 작고, 일부 클래스가 rare 하여 F1 loss가 아닌 FocalLoss + Lovaz를 사용함.

  IOU와 F1이 동일하지는 않지만, recall과 precision의 균형을 어느정도 확장할 수 있다고 생각하여 lovasz 손실 함수를 사용함.

```
class FocalLoss(nn.Module):
    def __init__(self, gamma=2):
        super().__init__()
        self.gamma = gamma

    def forward(self, logit, target, epoch=0):
        target = target.float()
        max_val = (-logit).clamp(min=0)
        loss = logit - logit * target + max_val + \
               ((-max_val).exp() + (-logit - max_val).exp()).log()

        invprobs = F.logsigmoid(-logit * (target * 2.0 - 1.0))
        loss = (invprobs * self.gamma).exp() * loss
        if len(loss.size())==2:
            loss = loss.sum(dim=1)
        return loss.mean()

class ArcFaceLoss(nn.modules.Module):
    def __init__(self,s=30.0,m=0.5):
        super(ArcFaceLoss, self).__init__()
        self.classify_loss = nn.CrossEntropyLoss()
        self.s = s
        self.easy_margin = False
        self.cos_m = math.cos(m)
        self.sin_m = math.sin(m)
        self.th = math.cos(math.pi - m)
        self.mm = math.sin(math.pi - m) * m

    def forward(self, logits, labels, epoch=0):
        cosine = logits
        sine = torch.sqrt(1.0 - torch.pow(cosine, 2))
        phi = cosine * self.cos_m - sine * self.sin_m
        if self.easy_margin:
            phi = torch.where(cosine > 0, phi, cosine)
        else:
            phi = torch.where(cosine > self.th, phi, cosine - self.mm)

        one_hot = torch.zeros(cosine.size(), device='cuda')
        one_hot.scatter_(1, labels.view(-1, 1).long(), 1)
        # -------------torch.where(out_i = {x_i if condition_i else y_i) -------------
        output = (one_hot * phi) + ((1.0 - one_hot) * cosine)
        output *= self.s
        loss1 = self.classify_loss(output, labels)
```

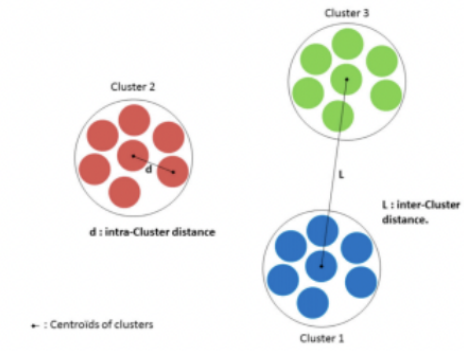
```
        loss2 = self.classify_loss(cosine, labels)
        gamma=1
        loss=(loss1+gamma*loss2)/(1+gamma)
        return loss
def lovasz_softmax(probas, labels, only_present=False, per_image=False, ignore=None):
    """
    Multi-class Lovasz-Softmax loss
      probas: [B, C, H, W] Variable, class probabilities at each prediction (between 0 and 1)
      labels: [B, H, W] Tensor, ground truth labels (between 0 and C - 1)
      only_present: average only on classes present in ground truth
      per_image: compute the loss per image instead of per batch
      ignore: void class labels
    """
    if per_image:
        loss = mean(lovasz_softmax_flat(*flatten_probas(prob.unsqueeze(0), lab.unsqueeze(0), ignore), only_present=only_present)
                    for prob, lab in zip(probas, labels))
    else:
        loss = lovasz_softmax_flat(*flatten_probas(probas, labels, ignore), only_present=only_present)
    return loss


def lovasz_softmax_flat(probas, labels, only_present=False):
    """
    Multi-class Lovasz-Softmax loss
      probas: [P, C] Variable, class probabilities at each prediction (between 0 and 1)
      labels: [P] Tensor, ground truth labels (between 0 and C - 1)
      only_present:  average only on classes present in ground truth
    """
    C = probas.size(1)
    losses = []
    for c in range(C):
        fg = (labels == c).float() # foreground for class c
        if only_present and fg.sum() == 0:
            continue
        errors = (Variable(fg) - probas[:, c]).abs()
        errors_sorted, perm = torch.sort(errors, 0, descending=True)
        perm = perm.data
        fg_sorted = fg[perm]
        losses.append(torch.dot(errors_sorted, Variable(lovasz_grad(fg_sorted))))
    return mean(losses)
```

▼ **Metric Learning**

- 데이터 간의 유사도를 잘 수치화하는 거리 함수 (metric function)을 학습하는 것.
  - 학습할 때 '분류'를 하는 것이 아니라, 같다/ 다르다를 나타내는 distance를 계산하는 metric 사용방식
- clustering, few shot learning 등 여러 가지 분야에 쓰임.
- **loss의 발전**
  - **초기** : Softmax를 활용한 Feature 추출, BUT discriminative features에 부적합

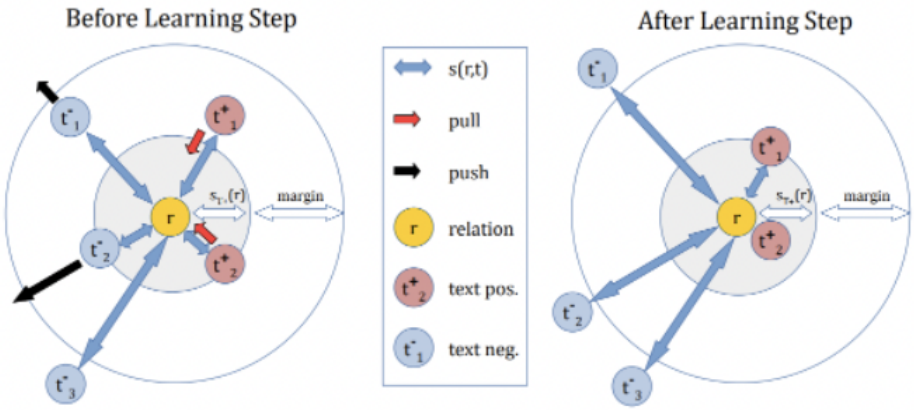
    * discriminative features(= Inter-difference) : 파란눈, 점 주름 등의 차별적인 특징
  - **중기 : Euclidean-distance 기반**

    Inter-Variance는 키우고, Intra-Variance는 줄이는 Euclidean Space로 임베딩 시키는 Metric Learning



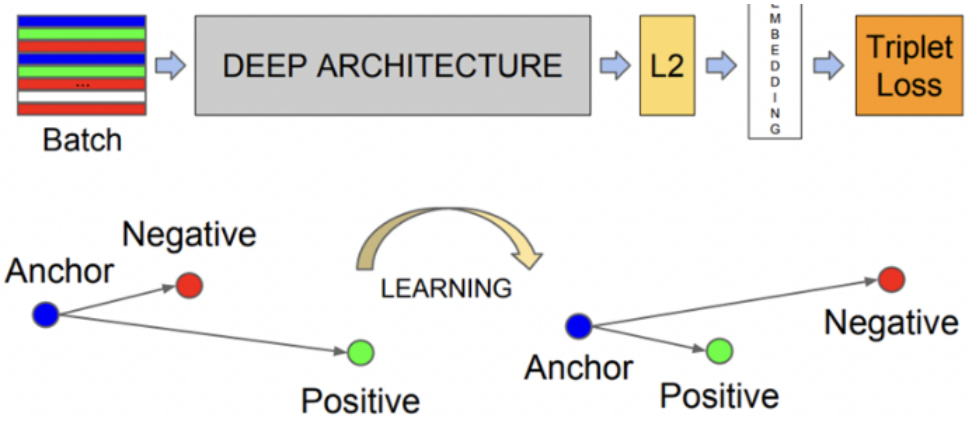
관련 loss로 Contrastive loss와 Triplet Loss가 있음.

- Contrastive loss



1. 같은 쌍(pos pair)는 거리가 0
2. 다른 쌍 (neg pair)는 거리가 margin이 되도록 학습.

- Triplet Loss



Contrastive는 절대적 거리(margin)을 고려하는 대신, **Triplet Loss는 상대적 거리** 차이를 학습

- **최근** : Angular/Consine-Margin 기반 + Sample Selection 및 쉬운 Parameter를 통한 학습 절차

  좀 더 어려운 데이터에 대해, 엄격하게 분리시키고자 함.

  decision boundary 도입.

  L-softmax/A-softmax: Softmax Loss에서 weight와 cos를 결합.

2. part of 4th place solution: GAPNet & dual loss ResNet

💡 **Keypoint**: GAPNet, Dual Loss ResNet

▼ GAPNet

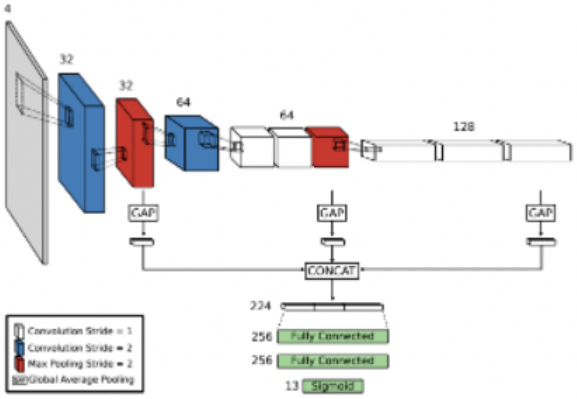
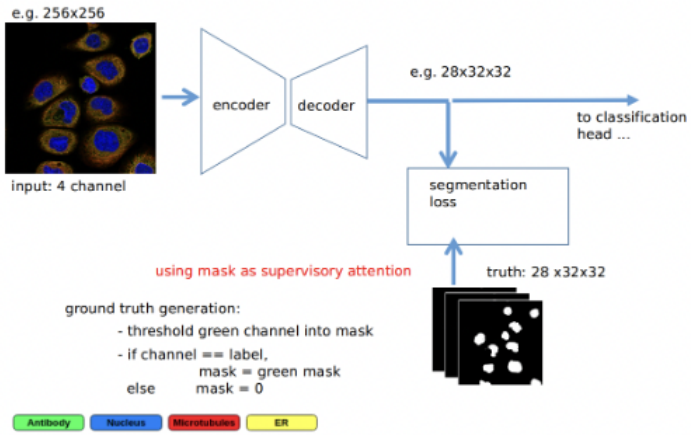


Figure 4: GapNet-PL architecture

- multiscale 가능.
- Average Pooling layers 앞에 SE-Blocks 추가 하여 성능 개선
- weighted bce, f1 score, cosine annealing lr schedule

▼ Dual Loss Resnet



- ResNet34 내의 마지막 32*32*128 레이어의 출력을 이용해 32*32*28 Conv2D

- segmentation loss를 얻기 위해 ground truth로 해당 레이블이 있는 녹색 채널의 다운샘플링 진행.
- s.l는 layer의 activation이 “nice”한지 확인하는 regularizer처럼 작동.
- supervised attention → 정규화에 도움

```
def build_resnet(
     repetitions=(2, 2, 2, 2),
     include_top=True,
     input_tensor=None,
     input_shape=None,
     classes=1000,
     block_type='usual'):

    """
    TODO
    """

    # Determine proper input shape
    input_shape = _obtain_input_shape(input_shape,
                                      default_size=224,
                                      min_size=197,
                                      data_format='channels_last',
                                      require_flatten=include_top)

    if input_tensor is None:
        img_input = Input(shape=input_shape, name='data')
    else:
        if not K.is_keras_tensor(input_tensor):
            img_input = Input(tensor=input_tensor, shape=input_shape)
        else:
            img_input = input_tensor

    # get parameters for model layers
    no_scale_bn_params = get_bn_params(scale=False)
    bn_params = get_bn_params()
    conv_params = get_conv_params()
    init_filters = 64

    if block_type == 'basic':
        conv_block = basic_conv_block
        identity_block = basic_identity_block
    else:
        conv_block = usual_conv_block
        identity_block = usual_identity_block

    # resnet bottom
    x = BatchNormalization(name='bn_data', **no_scale_bn_params)(img_input)
    x = ZeroPadding2D(padding=(3, 3))(x)
    x = Conv2D(init_filters, (7, 7), strides=(2, 2), name='conv0', **conv_params)(x)
    x = BatchNormalization(name='bn0', **bn_params)(x)
    x = Activation('relu', name='relu0')(x)
    x = ZeroPadding2D(padding=(1, 1))(x)
    x = MaxPooling2D((3, 3), strides=(2, 2), padding='valid', name='pooling0')(x)

    # resnet body
    for stage, rep in enumerate(repetitions):
        for block in range(rep):

            filters = init_filters * (2**stage)

            # first block of first stage without strides because we have maxpooling before
            if block == 0 and stage == 0:
                x = conv_block(filters, stage, block, strides=(1, 1))(x)

            elif block == 0:
                x = conv_block(filters, stage, block, strides=(2, 2))(x)

            else:
                x = identity_block(filters, stage, block)(x)

    x = BatchNormalization(name='bn1', **bn_params)(x)
    x = Activation('relu', name='relu1')(x)

    # resnet top
```

```
    if include_top:
        x = GlobalAveragePooling2D(name='pool1')(x)
        x = Dense(classes, name='fc1')(x)
        x = Activation('softmax', name='softmax')(x)

    # Ensure that the model takes into account any potential predecessors of `input_tensor`.
    if input_tensor is not None:
        inputs = get_source_inputs(input_tensor)
    else:
        inputs = img_input

    # Create model.
    model = Model(inputs, x)

    return model
```