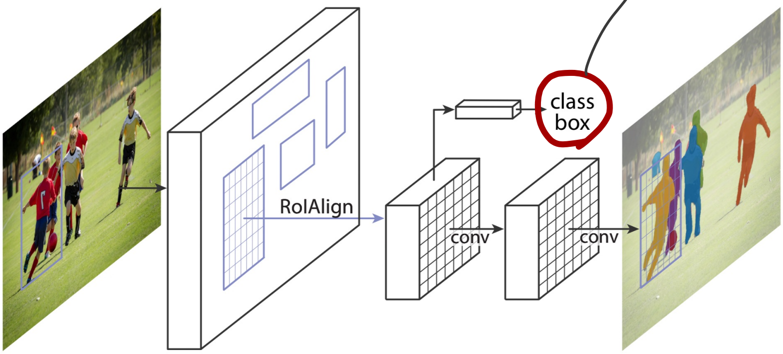
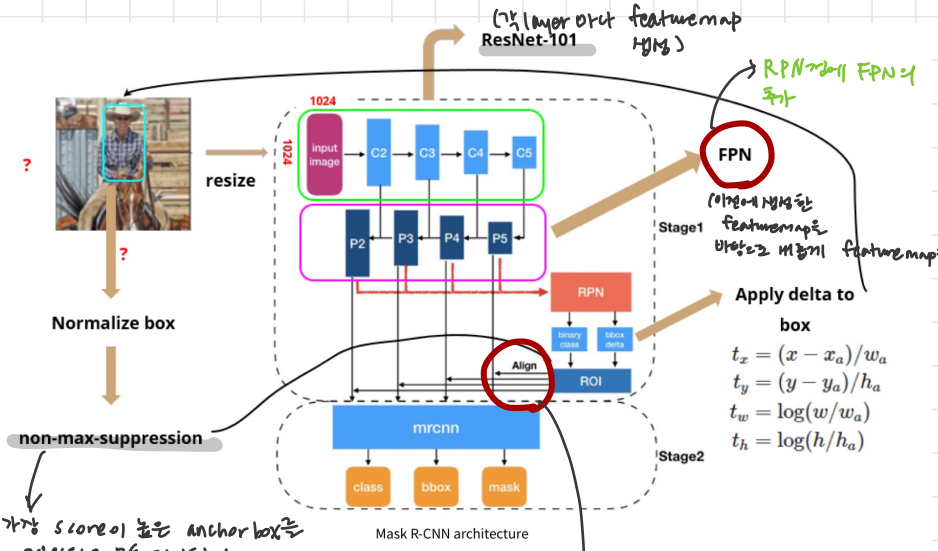
Mask R-CNN = Segmentation mask + RoI + Faster R-CNN

another extended branch for predicting segmentation masks on each RoI



특히나 "Instance segmentation"의 기능을 수행하는데에 성능↑



Mask R-CNN architecture

(각 layer 마다 featuremap 생성)

RPN거쳐 FPN의 추가

(이전에 생성한 featuremap을 바탕으로 새롭게 featuremap을 생성)

가장 score이 높은 anchor box는 제외하고 모두 제거한다.
(이경에는 IoU ratio가 0.5보다 작은 모든 gtbox에 음수값의 anchor은 적용하였다)

[ROI Pooling 대신에 사용]

RoI Align (각각의 크기가 다른 anchor box의 크기를 일정하게 맞춰줌)

: mask branch는 제대로 적용하기 위해서 적용

Faster R-CNN은 픽셀-픽셀끼리의 alignment가 아니었다.

input tensor 값의 크기가 [N, C, H, W] 일때

output tensor 값의 크기는 [K, C, output_size[0], output_size[1]]

반환해주 (즉, RoI 영역의 개수)

< RPN (Region Proposal Network) >

- 이미지를 입력받아서 각각 성적 (점수가 매겨진)을 fully convolution network을 이용해서 수행한다.

$$t_x = (x - x_a)/w_a$$
$$t_y = (y - y_a)/h_a$$
$$t_w = \log(w/w_a)$$
$$t_h = \log(h/h_a)$$

이 값에 anchor 정보를 연산해서 원래 이미지에 대응되는 anchor bounding box 생성

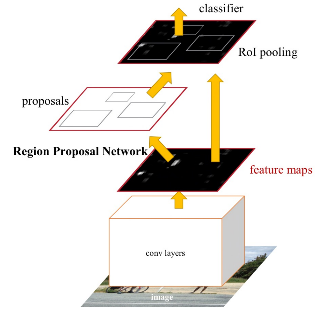
$$t_x^* = (x^* - x_a)/w_a$$
$$t_y^* = (y^* - y_a)/h_a$$
$$t_w^* = \log(w^*/w_a)$$
$$t_h^* = \log(h^*/h_a)$$

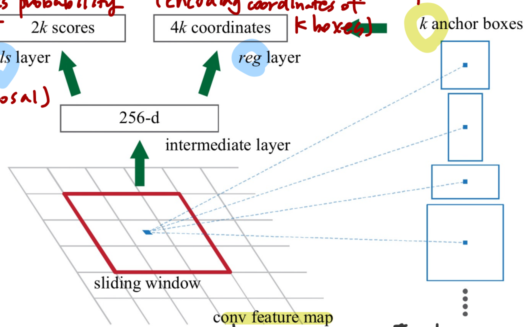
t_x, t_y : 박스의 center coordinates
t_w, t_h : 박스의 width, height
x, y, w, h : predicted box
x_a, y_a, w_a, h_a : anchor box
x^*, y^*, w^*, h^* : ground-truth box

bbox regression

① Anchors

- 각각의 sliding window location에 대해서 multiple region proposal을 진행





(estimates probability of object or not object for each proposal)

(encoding coordinates of k boxes)

number of maximum possible proposals for each location

Feature map의 크기가 [W×H]이면 WHK개 anchor이 생겨

② Loss Function

$$L(\{p_i\},\{t_i\}) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i^*) + \lambda \frac{1}{N_{reg}} \sum_i p_i^* L_{reg}(t_i, t_i^*).$$

- L_{cls} → log loss over two classes (object vs non object)
- t_i^* → 정답 값
- t_i → 예측 값
- p_i → predicted probability of anchor i being an object
- i → index of the anchor in a mini batch

$R(t_i - t_i^*)$
→ Smooth L1 defined as robust loss function

p_i^*의 값은 1과 0으로만 나타나며
예측을 제대로 한 경우 (아마 IoU score ≥ 0.3 일때만 anchor 값을 양수로 적용한다고 했으므로)
1이기 때문에
결국 cls와 reg layer의 출력값은 {p_i}와 {t_i}에 의존

Mask R-CNN: Mask R-CNN adopts the same two-stage procedure, with an identical first stage (which is RPN). In the second stage, *in parallel* to predicting the class and box offset, Mask R-CNN also outputs a binary mask for each RoI. This is in contrast to most recent systems, where classification *depends* on mask predictions (*e.g.* [33, 10, 26]). Our approach follows the spirit of Fast R-CNN [12] that applies bounding-box classification and regression in *parallel* (which turned out to largely simplify the multi-stage pipeline of original R-CNN [13]).

Formally, during training, we define a multi-task loss on each sampled RoI as $L = L_{cls} + L_{box} + L_{mask}$. The classification loss L_{cls} and bounding-box loss L_{box} are identical as those defined in [12]. The mask branch has a Km^2-dimensional output for each RoI, which encodes K binary masks of resolution $m \times m$, one for each of the K classes. To this we apply a per-pixel sigmoid, and define L_{mask} as the average binary cross-entropy loss. For an RoI associated with ground-truth class k, L_{mask} is only defined on the k-th mask (other mask outputs do not contribute to the loss).

기존의 R-CNN 모델들이 RoI에서 나온 값을 계산한 Lcls+Lbox에 뒤에 추가한 mask의 loss도 함께 계산해 준다.
K개의 class마다 mxm의 크기 mask로 resolution을 진행해 주면 Km²이라는 개수/크기의 loss값이 도출이 된다.

Our definition of L_{mask} allows the network to generate masks for every class without competition among classes; we rely on the dedicated classification branch to predict the class label used to select the output mask. This *decouples* mask and class prediction. This is different from common practice when applying FCNs [30] to semantic segmentation, which typically uses a per-pixel *softmax* and a *multinomial* cross-entropy loss. In that case, masks across classes compete; in our case, with a per-pixel *sigmoid* and a *binary* loss, they do not. We show by experiments that this formulation is key for good instance segmentation results.

기존의 모델들은 오히려 classification이 mask prediction 값에 의존했었다면 mask R-CNN의 경우에는 classification 값에 의존해서 output mask의 값을 결정할수 있도록 한다.

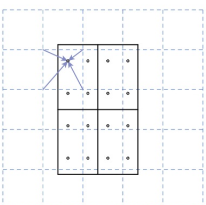
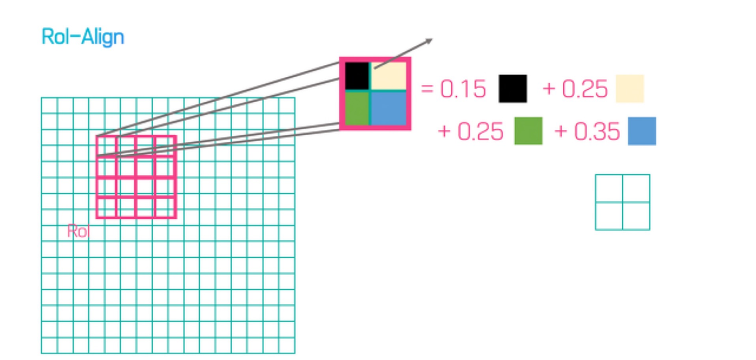


Figure 3. **RoIAlign:** The dashed grid represents a feature map, the solid lines an RoI (with 2×2 bins in this example), and the dots the 4 sampling points in each bin. RoIAlign computes the value of each sampling point by bilinear interpolation from the nearby grid points on the feature map. No quantization is performed on any coordinates involved in the RoI, its bins, or the sampling points.

결론적으로 class끼리 경쟁하지 않게되어서
per pixel sigmoid
binary loss
이 두가지 방법을 적용하여서 성능을 높인다.

<RoI- Align>



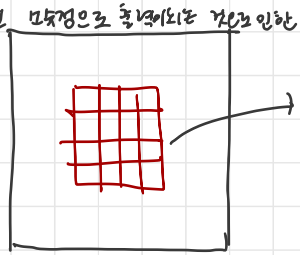
RoI align

RoIAlign: RoIPool [12] is a standard operation for extracting a small feature map (*e.g.*, 7×7) from each RoI. RoIPool first *quantizes* a floating-number RoI to the discrete granularity of the feature map, this quantized RoI is then subdivided into spatial bins which are themselves quantized, and finally feature values covered by each bin are aggregated (usually by max pooling). Quantization is performed, *e.g.*, on a continuous coordinate x by computing $[x/16]$, where 16 is a feature map stride and $[\cdot]$ is rounding; likewise, quantization is performed when dividing into bins (*e.g.*, 7×7). These quantizations introduce misalignments between the RoI and the extracted features. While this may not impact classification, which is robust to small translations, it has a large negative effect on predicting pixel-accurate masks.

To address this, we propose an *RoIAlign* layer that removes the harsh quantization of RoIPool, properly *aligning* the extracted features with the input. Our proposed change is simple: we avoid any quantization of the RoI boundaries or bins (*i.e.*, we use $x/16$ instead of $[x/16]$). We use bilinear interpolation [22] to compute the exact values of the input features at four regularly sampled locations in each RoI bin, and aggregate the result (using max or average), see Figure 3 for details. We note that the results are not sensitive to the exact sampling locations, or how many points are sampled, *as long as* no quantization is performed.

RoIAlign leads to large improvements as we show in §4.2. We also compare to the RoIWarp operation proposed in [10]. Unlike RoIAlign, RoIWarp overlooked the alignment issue and was implemented in [10] as quantizing RoI just like RoIPool. So even though RoIWarp also adopts bilinear resampling motivated by [22], it performs on par with RoIPool as shown by experiments (more details in Table 2c), demonstrating the crucial role of alignment.

기존에 사용하던 RoI Pooling과 달리 RoI Align을 사용하면 소수점으로 출력이되는 것으로 인한 오류를 해결할수 있다고 한다. 기존에는

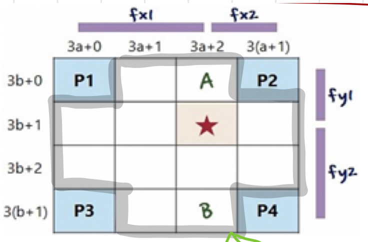


이런식으로 소수점 좌표를 갖게되면 그냥 반올림해서 구하는데, 그렇게 하면 input 이미지의 왜곡이 생길수 있어서 비효율적이다.

<Non-Max Suppression>

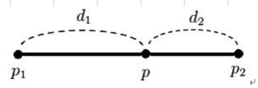
- object 마다 대응되는 수천개의 anchor들 중에서 가장 성능이 좋은, 즉 점수가 높은 anchor만을 남겨두고 나머지는 모두 제거하는 방법을 사용한다.

<Resize Input Image>



bilinear interpolation

Mask R-CNN에서는 backbone으로 Res-Net 101을 사용하는데 ResNet에서는 input 이미지의 크기가 800~1024일때 성능이 제일 좋다. 따라서 이미지는 해당 size로 맞춰주는 과정이 필요하다.



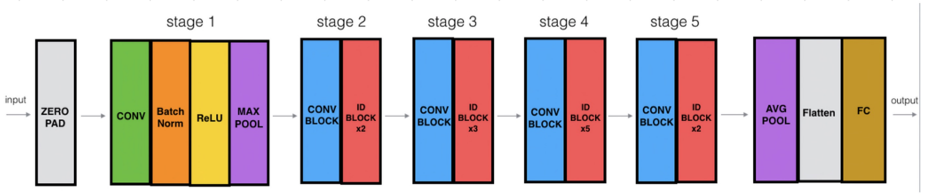
일반적으로 두 지점 p1, p2 에서의 데이터 값이 각각 f(p1), f(p2)일 때, p1, p2 사이의 임의의 지점 p 에서의 데이터 값 f(p) 는 다음과 같이 계산할 수 있다.

$$f(p) = \frac{d_2}{d_1 + d_2} f(p_1) + \frac{d_1}{d_1 + d_2} f(p_2)$$

이 회색부분을 f(p) 값으로 채워준다

만약 1024에서 모자라게 size를 바꿔주었다면 나머지 값들은 zero padding을 사용해서 채워준다.

<Backbone Network – ResNet 101>



ResNet Structure

<Feature Pyramid Network>

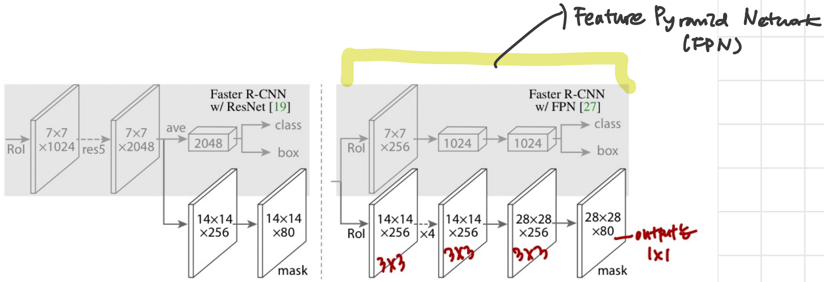
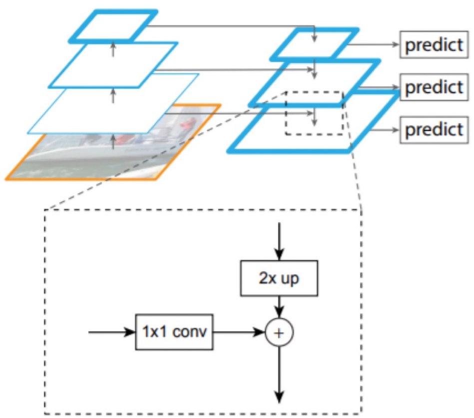


Figure 4. **Head Architecture**: We extend two existing Faster R-CNN heads [19, 27]. Left/Right panels show the heads for the ResNet C4 and FPN backbones, from [19] and [27], respectively, to which a mask branch is added. Numbers denote spatial resolution and channels. Arrows denote either conv, deconv, or *fc* layers as can be inferred from context (conv preserves spatial dimension while deconv increases it). All convs are 3×3, except the output conv which is 1×1, deconvs are 2×2 with stride 2, and we use ReLU [31] in hidden layers. *Left*: 'res5' denotes ResNet's fifth stage, which for simplicity we altered so that the first conv operates on a 7×7 RoI with stride 1 (instead of 14×14 / stride 2 as in [19]). *Right*: '×4' denotes a stack of four consecutive convs.



FPN detail (Upsampling

오른쪽 C5로부터 생성되는 featuremap

Feature Map을 $F_1 \sim F_6$으로 표현해보면

F_1

$F_2 \sim F_5$ → 추출해서 RPN에 보내기에 앞서 3x3 Max pooling을 진행한다.

F_6

→ F_6의 경우에는 F_5에서 3x3 Maxpooling을 진행해서 추출을 진행한 결과이기 때문에 굳이 다시 RPN에 보내기 이전에 pooling을 해주지 않는다.

We denote the *backbone* architecture using the nomenclature *network-depth-features*. We evaluate ResNet [19] and ResNeXt [45] networks of depth 50 or 101 layers. The original implementation of Faster R-CNN with ResNets [19] extracted features from the final convolutional layer of the 4-th stage, which we call C4. This backbone with ResNet-50, for example, is denoted by ResNet-50-C4. This is a common choice used in [19, 10, 21, 39].

아래에 나오는 모델 설명의 경우에는 mobilenet_v2를 (사전 학습된), 그것의 feature 추출 값은 넣었다.

backbone network로 ResNet을 사용하거나 (그 중에서도 Resnet-FPN을 사용하는 것이 성능이 좋다)

논문에서는 ResNet 또는 FPN을 이용한다고 언급되어있다.

```
MaskRCNN(
  (transform): GeneralizedRCNNTransform(
      Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
      Resize(min_size=(800,), max_size=1333, mode='bilinear')
  )
  (backbone): Sequential(
    (0): ConvBNActivation(
      (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
      (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU6(inplace=True)
    )
    (1): InvertedResidual(
      (conv): Sequential(
        (0): ConvBNActivation(
          (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
groups=32, bias=False)
          (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): Conv2d(32, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (2): InvertedResidual(
      (conv): Sequential(
        (0): ConvBNActivation(
          (0): Conv2d(16, 96, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (1): ConvBNActivation(
          (0): Conv2d(96, 96, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
groups=96, bias=False)
          (1): BatchNorm2d(96, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
          (2): ReLU6(inplace=True)
        )
        (2): Conv2d(96, 24, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (3): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (3): InvertedResidual(
      (conv): Sequential(
        (0): ConvBNActivation(
          (0): Conv2d(24, 144, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (1): BatchNorm2d(144, eps=1e-05, momentum=0.1, affine=True,
```

For the network *head* we closely follow architectures presented in previous work to which we add a fully convolutional mask prediction branch. Specifically, we extend the Faster R-CNN box heads from the ResNet [19] and FPN [27] papers. Details are shown in Figure 4. The head on the ResNet-C4 backbone includes the 5-th stage of ResNet (namely, the 9-layer 'res5' [19]), which is compute-intensive. For FPN, the backbone already includes res5 and thus allows for a more efficient head that uses fewer filters.

We note that our mask branches have a straightforward structure. More complex designs have the potential to improve performance but are not the focus of this work.

```
  (rpn): RegionProposalNetwork(
    (anchor_generator): AnchorGenerator()
    (head): RPNHead(
      (conv): Conv2d(1280, 1280, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (cls_logits): Conv2d(1280, 15, kernel_size=(1, 1), stride=(1, 1))
      (bbox_pred): Conv2d(1280, 60, kernel_size=(1, 1), stride=(1, 1))
    )
  )
  (roi_heads): RoIHeads(
    (box_roi_pool): MultiScaleRoIAlign(featmap_names=['0'], output_size=(7, 7),
sampling_ratio=2)
    (box_head): TwoMLPHead(
      (fc6): Linear(in_features=62720, out_features=1024, bias=True)
      (fc7): Linear(in_features=1024, out_features=1024, bias=True)
    )
    (box_predictor): FastRCNNPredictor(
      (cls_score): Linear(in_features=1024, out_features=2, bias=True)
      (bbox_pred): Linear(in_features=1024, out_features=8, bias=True)
    )
    (mask_roi_pool): MultiScaleRoIAlign(featmap_names=['0'], output_size=(14, 14),
sampling_ratio=2)
    (mask_head): MaskRCNNHeads(
      (mask_fcn1): Conv2d(1280, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (relu1): ReLU(inplace=True)
      (mask_fcn2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (relu2): ReLU(inplace=True)
      (mask_fcn3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (relu3): ReLU(inplace=True)
      (mask_fcn4): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (relu4): ReLU(inplace=True)
    )
    (mask_predictor): MaskRCNNPredictor(
      (conv5_mask): ConvTranspose2d(256, 256, kernel_size=(2, 2), stride=(2, 2))
      (relu): ReLU(inplace=True)
      (mask_fcn_logits): Conv2d(256, 2, kernel_size=(1, 1), stride=(1, 1))
    )
  )
)
```