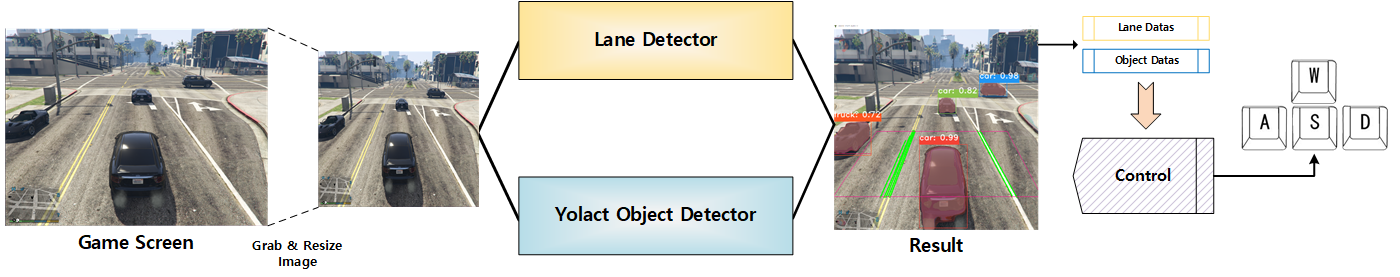
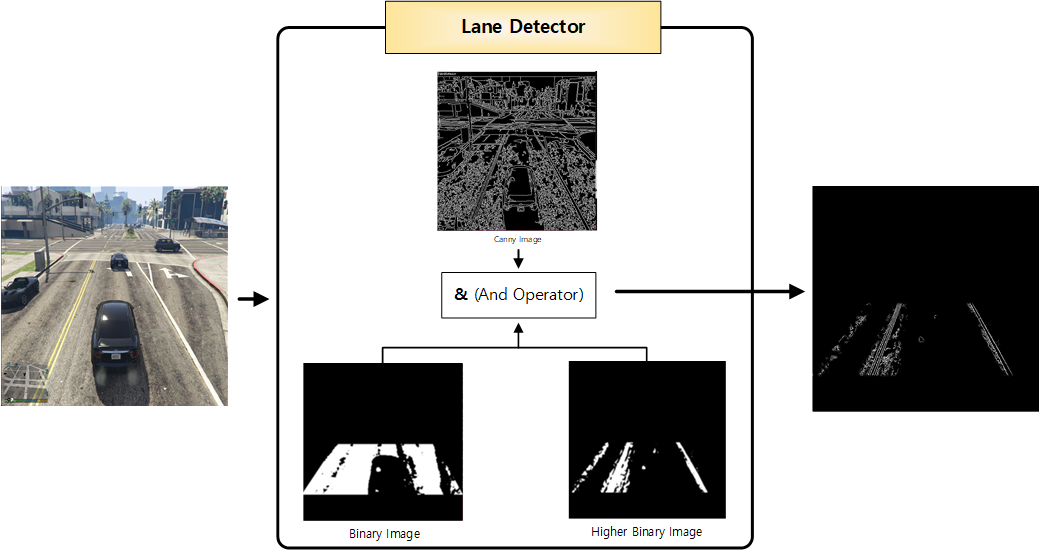
게임 프로젝트

*요약: 게임에서 Object Detection이랑 Lane Detection을 이용해서 자율주행 구현하기*

[*https://github.com/bobsiunn/Autodrivegame*](https://github.com/bobsiunn/Autodrivegame)

**1. Overall**

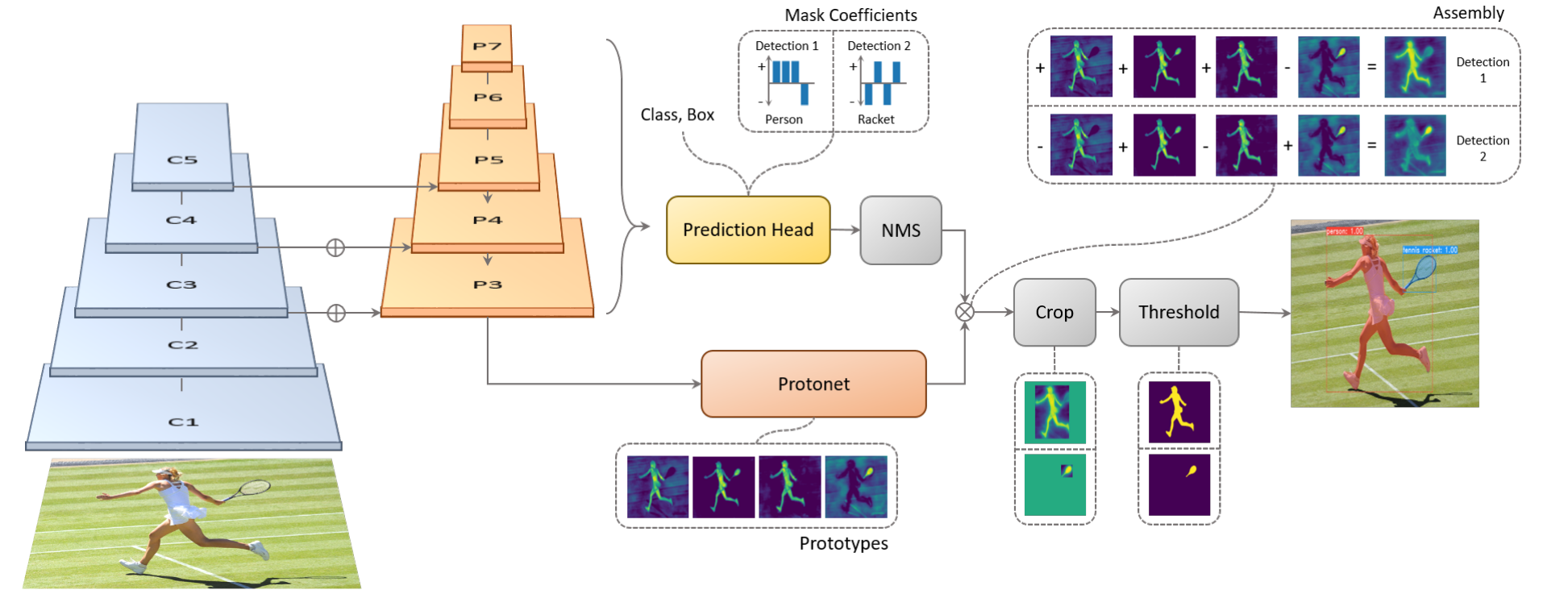
전체적인 진행 방식은 다음과 같다. 화면의 일정 부분을 Grab 툴을 이용해서 화면에 표시된 내용을 저장한다. 해당 프레임에 게임 화면을 위치하게 되면 게임 화면을 가져올 수 있다. 해당 이미지를 resize를 거친 후 각각 Lane Detector와 Yolact Object Detector로 입력이 들어가게 된다. Lane Detector는 ROI(Region of Interested) 부분의 차선을 인식하게 되고, Yolact Object Detector는 화면 내부의 사물들을 각각 인식하게 된다.

**2. Lane Detection**

Lane Detection의 동작 과정은 Figure2와 같다. 먼저, Canny 알고리즘을 활용해서 입력된 이미지의 테두리 선들만을 검출한다. 다음으로, 이미지를 이진화 시킨다. (Binary Image) 이진화 시키면 일정 임계값 이상의 밝은 부분의 픽셀은 1의 값을 가지고 어두운 부분의 이미지는 0의 값을 가진다. Higher Binary 이미지는 기존 Binary Image 보다 높은 레벨의 이미지로 Binary 이미지에서 1의 값들이 전체 영역에서 차지하는 비율에 따라서 임계값을 유동적으로 조절한다. 이렇게 하면, 밝은 낮일 때는 임계값을 높여서 인식 범위를 좁히고, 어두운 밤일때는 임계값을 낮춰서 인식 범위를 넓힌다.

이렇게 두가지 Image(Canny Image, Binary Image)를 And Operator를 이용해서 연산을 진행하면 하얀색 모서리인 부분이 뽑히게 된다. ROI는 도로 부분이기 때문에 도로 위에서 하얀색 모서리 부분이 검출되게 되고, Hough 연산을 진행하면 하얀 모서리 직선부분이 되고 이는 곧 차선일 확률이 높다. 이러한 과정을 통해서 차선을 검출하게 된다.

**3. Object Detection**

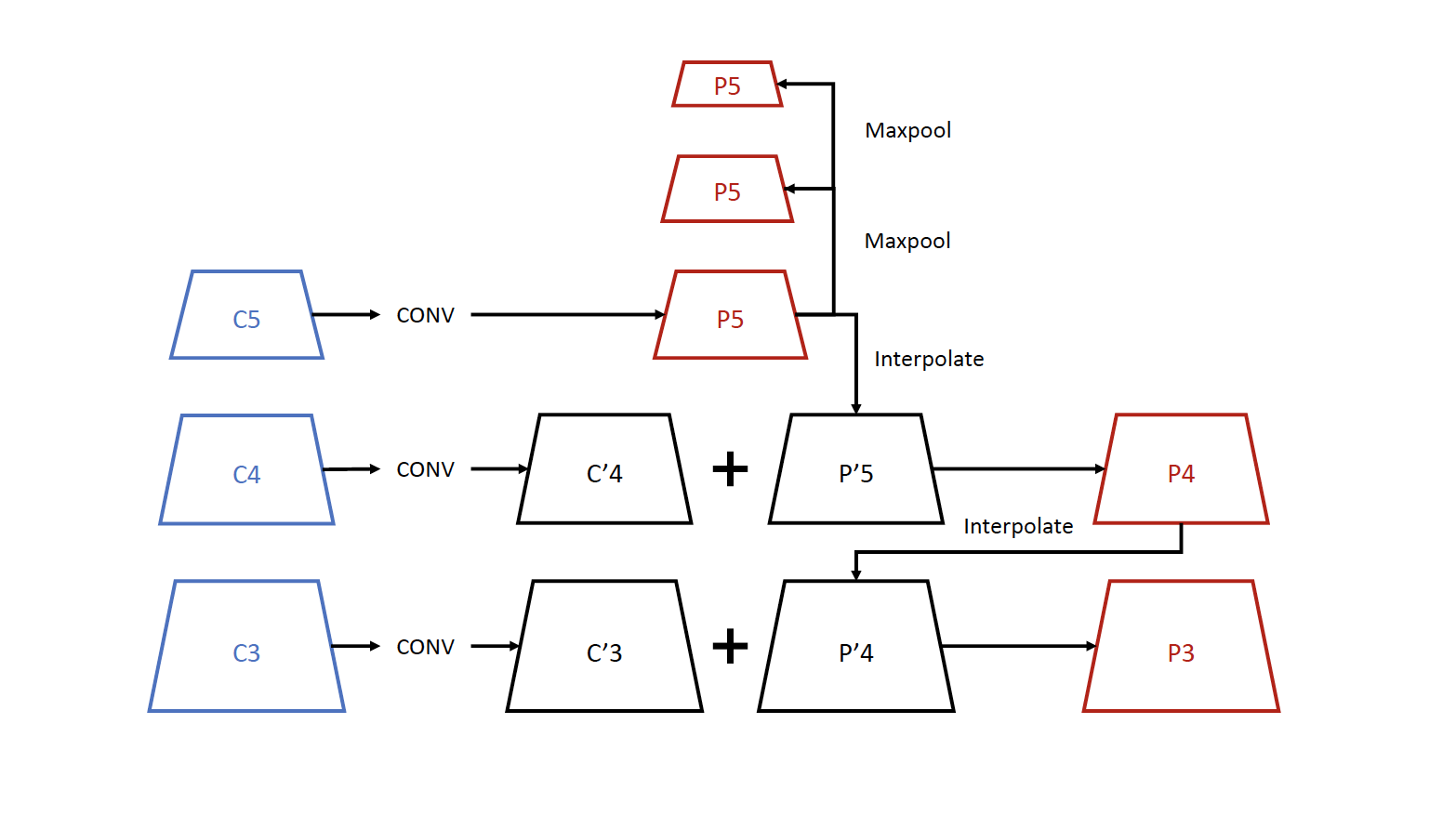
****

**Figure 3 Yolact Architecture**

Backbone:

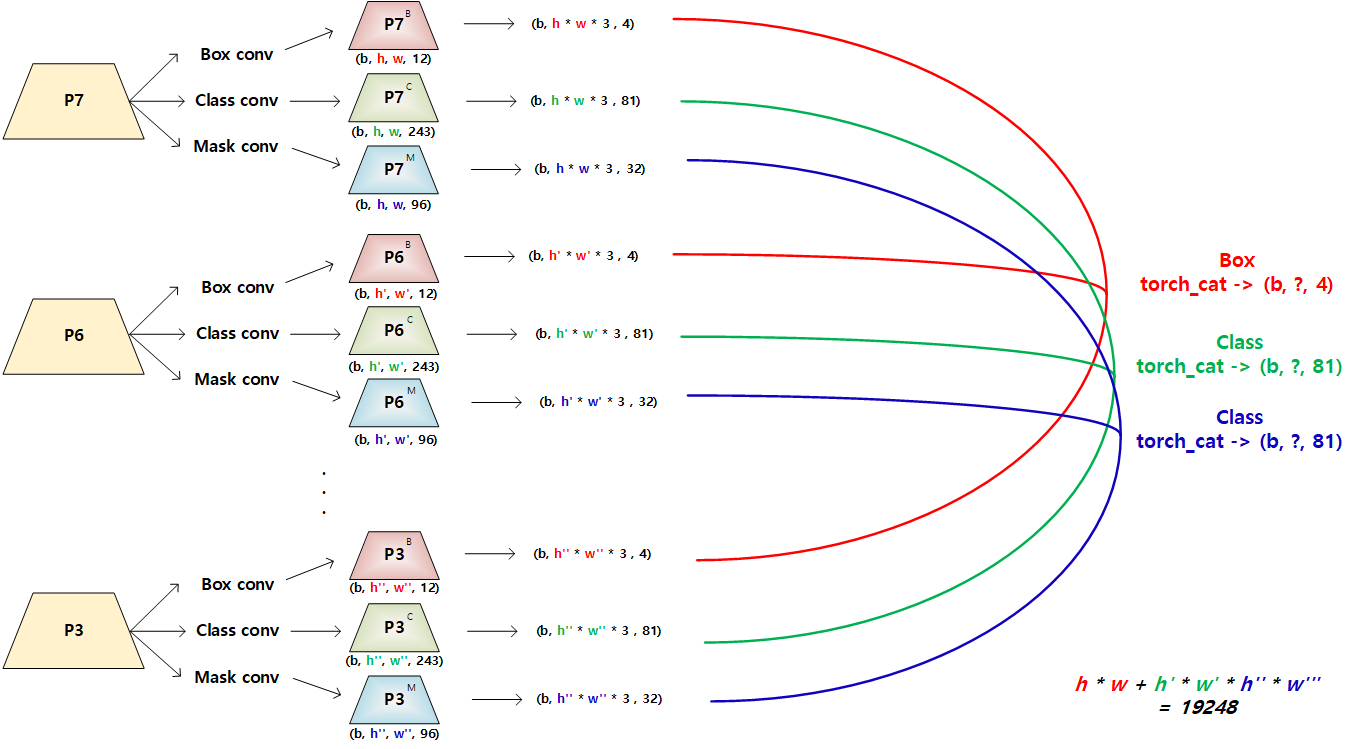
* Resnet101
* Bottleneck 기본 구조
  + conv layer(out\_channel = 64, kernel\_size = 1, stride = 1)
  + batch normalization
  + conv layer(out\_channel = 64, kernel\_size = 3, stride = 1, padding 1)
  + batch normalization
  + conv layer(out\_channel 256, kernel\_size = 1, stride = 1)
  + batch normalization
  + ReLU
  + 각 Sequential의 첫번째만 downsample
    - conv layer(out\_channel = 256, kernel\_size = 1, stride = 1
    - batch normalization
* C1 -> Bottleneck x 3 -> C2 -> Bottleneck x 4 -> C3 -> Bottleneck x 23 -> C4 -> Bottleneck x 3 -> C5

FPN

* C5 -> conv(out\_channel = 256, kernel\_size = 1) -> P5
* (P5 -> interpolate(size of C4)) + (C4 -> conv(out\_channel = 256, kernel\_size = 1)) -> P4
* (P4 -> interpolate(size of C3)) + (C3 -> conv(out\_channel = 256, kernel\_size = 1)) -> P3
* P5 -> Maxpooling(kernel\_size = 2) -> P6
* P6 -> Maxpooling(kernel\_size = 2) -> P7
* 
* 설명

: Feature Pyramid Network로 Top-down 방식으로 특징을 추출하며, 각 추출된 결과들인 low resolution 및 high-resolution들을 묶는 방식이다. 각 레벨에서 독립적으로 특징을 추출하여 객체를 탐지하게 되는데 상위 레벨의 이미 계산된 특징을 재사용하므로 멀티 스케일 특징들을 효율적으로 사용할 수 있다. 우선 Pn layer들을 만들기 전에 Cn layer들의 가로 세로 길이와 채널의 개수가 모두 다르다. 우선 모든 Cn layer들에게 똑같은 conv 연산(out\_channel = 256, kernel\_size = 1)을 해줌으로써 가로 세로 길이는 변하지 않고 channel의 개수를 256개로 맞춰준다. 그 후 더 high resolution(=P에서 n 숫자가 더 크면 high이다) layer를 interpolation(bilinear)한 것과 conv 연산된 layer를 더해준다. 예를 들어, C5를 conv 연산한 것을 P5로, P5를 interpolation한 것과 C4를 conv 연산한 것을 더해서 P4를 만들어준다. P3 또한 같은 원리로 생성된다. P6와 P7의 경우, 이전의 연산과는 다른 Maxpooling 연산을 하여 생성한다.

Prediction Head

* 
* ( 그림에 오타: 파란색 class -> mask)
* 설명

: P3부터 P7까지의 각 layer에 대해 box, class confidence, mask conv 연산을 한다. channel 개수만 바뀌고 가로,세로 길이는 변하지 않음

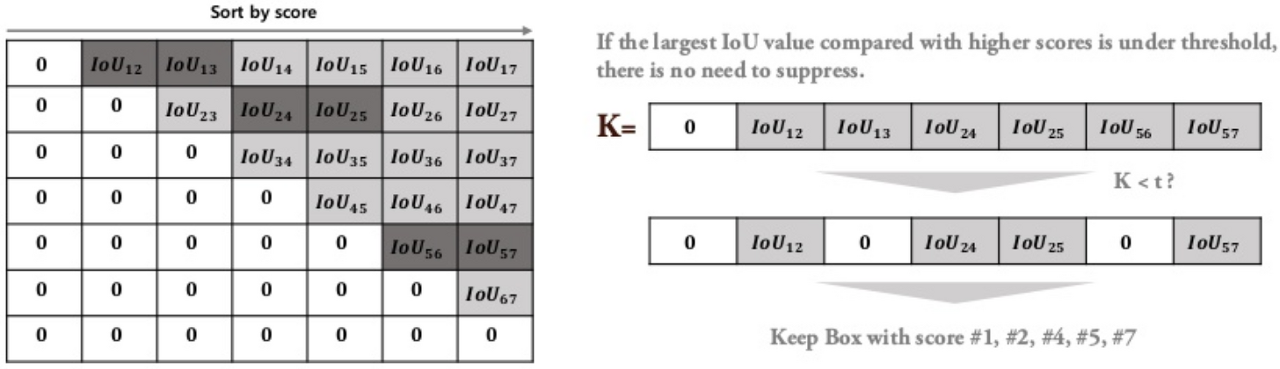
* box: conv(out\_channels = 3 x 4, kernel\_size = 3, padding = 1)
* class confidence: conv(out\_channels = 3 x 81, kernel\_size = 3, padding = 1)
* mask: conv(out\_channels = 3 x 32, kernel\_size = 3, padding = 1)

: 그 후 사이즈 조정을 한다. 연산된 결과는 배치 사이즈와 채널 개수만 같고 나머지 가로/세로 길이는 모두 다르다. (batch\_size, channel, h, w)여기서 h와 w만 다르다. box layer는 (batch\_size, 12, h, w) -> (batch\_size, h x w x 3, 4)로, class layer는 (batch\_size, 243, h, w) -> (batch\_size, h x w x 3, 81)로, mask layer는 (batch\_size, 96, h, w) -> (batch\_size, h x w x 3, 32)로 바꿔준다. 그리고 box는 box끼리, class는 class끼리, mask는 mask끼리 torch.cat을 통해 합쳐준다.

Protonet

* 구조
  + conv(out\_channels = 256, kernel\_size = 3, stride = 1, padding = 1)
  + ReLU
  + conv(out\_channels = 256, kernel\_size = 3, stride = 1, padding = 1)
  + ReLU
  + conv(out\_channels = 256, kernel\_size = 3, stride = 1, padding = 1)
  + ReLU
  + interpolate
  + ReLU
  + conv(out\_channels = 256, kernel\_size = 3, stride = 1, padding = 1)
  + ReLU
  + conv(out\_channels = 32, kernel\_size = 1, stride = 1)
* 연산 후, (batch\_size, 32, h, w) -> (batch\_size, h, w, 32)로 사이즈 조정

NMS

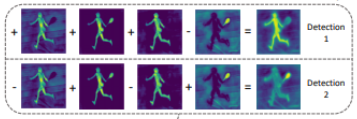


* 구성
  + box = (#pixel x #pixel x 12) = mask 픽셀 사이즈와 각 anchor box 3개의 좌표
  + class = (#pixel x #pixel x 243) = 각 anchor box와 81개 class score

- 과정

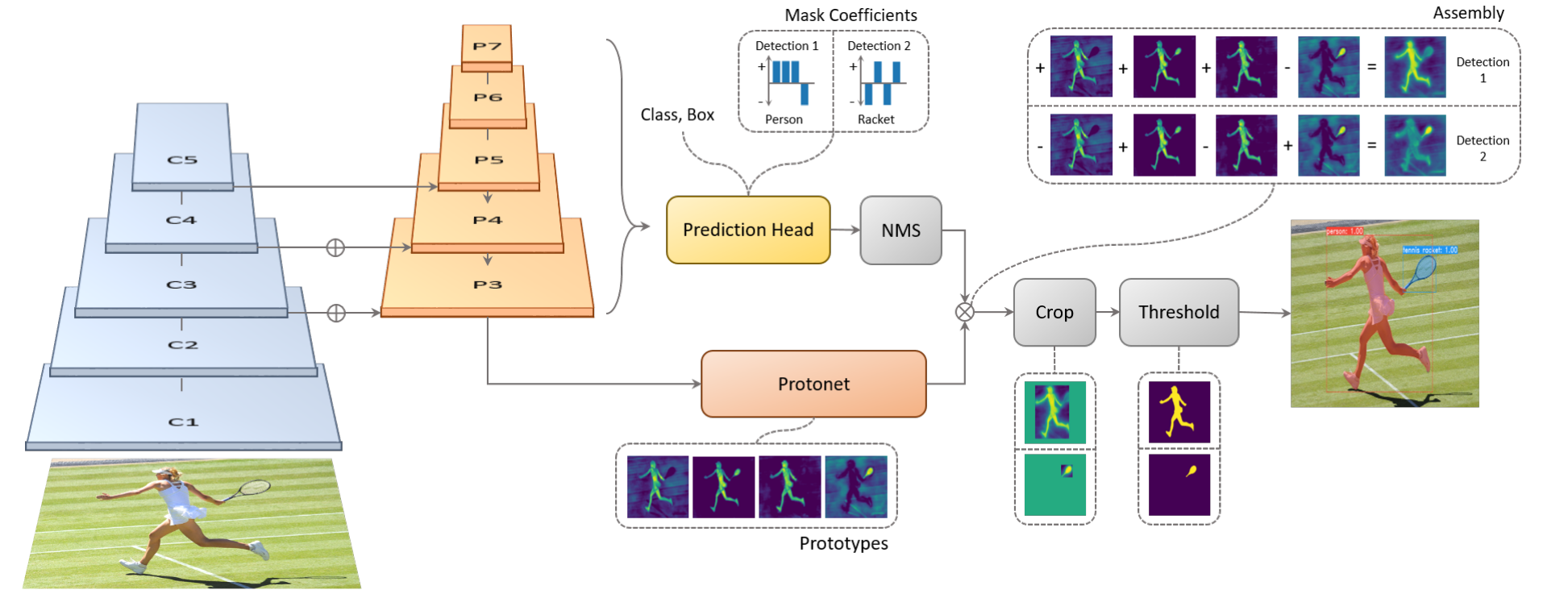
1. 각 class score를 기준으로 모든 anchor를 정렬
2. anchor끼리 IOU를 계산하고, 각 열에서 가장 score가 큰 anchor만 K matrix로 복사
3. K matrix에서 threshold값보다 작은 anchor만을 남기고 나머지를 배제
4. 남은 anchor들이 prediction head의 최종값으로 반환

Assembly



* 구성
  + k개의 protonet mask
  + 각 mask에 해당하는 Coefficient
* 과정
  + 각 protonet mask에 대해 Coefficient 간의 linear combination을 실시한다
  + 이때, 단순 합만으로는 1개의 object mask만을 남기기 어려우므로 mask 간 더하기 연산과 빼기 연산을 병용한다
  + 이를 위해 tanh 등 음수값을 가지는 activation 함수를 사용하여 학습 시 합차 연산이 모두 이루어지도록 해야 한다

Crop & Threshold

****

* 구성
  + masks = assembly를 통해 만들어진 mask
  + boxes = 해당 mask에 대해 가장 confidence가 높았던 anchor box의 좌표
* 과정
  + mask 상에서 boxes의 좌표값에 해당하는 지점들을 선택하고
  + 4개의 지점을 연결하는 mask 내 영역을 복사하여 반환한다
  + threshold 값 이하인 부분은 grey scale value로 변경하고, 이상인 부분은 그대로 취한 이후 반환한다.

**4. Loss**

Segmentation Loss

semantic segmentation loss 계산을 위해 사용되는 Layer는 P3에 channel = 80, kernel\_size = 1인 conv 연산을 거친 결과물인 80, 69, 69 사이즈를 가진 tensor이다. 쉽게 생각하면 한 클래스당 69x69 사이즈의 이미지가 사용된다는 것이다.(클래스의 개수는 background를 제외하면 80개이다) 비교를 위해 실제 mask(정답 데이터)를 interpolation을 통해 69x69로 사이즈 downsampling해주고 binary cross entropy로 pixel-wise하게 계산한다. 즉, 한 클래스당 모델에서 나온 결과값 이미지 한 장과 정답 이미지 한 장을 비교한다. 한 클래스당 정답 이미지 한장을 만들기 위해 같은 클래스의 경우 여러 object 사진들을 합치는 과정을 거치게 된다. pixel별 평균 loss를 이용하기 때문에 모든 pixel에서의 loss를 구한 후 69x69 영역으로 나눈다.

Box Loss

Bos loss 계산법은 간단하다. 를 계산을 하기 위해서 tensor로 나온 box들과 정답 데이터간의 일치도(jaccard)를 계산한다. 이후 smooth l1 loss를 이용해서 loss를 계산하게 된다.

Mask Loss

모든 anchor들에 대한 k개 coef를 담고있는 리스트에서 positive anchor들의 coef만 뽑은 변수에 protonet의 결과값 transpose를 곱하면 (138, 138, num\_pos)로 linear combination mask결과가 나온다. 여기서 실제 anchor box의 영역을 제외한 pixel의 mask는 0으로 값을 수정해준다.

이제 실제 ground truth mask와 binary cross entropy로 각 pixel별로 bce(binary cross entropy)를 계산한다.

각 pixel별로 구해진 loss를 각 object별로 다 더해서 모든 영역의 pixel loss를 더한 값이 positive anchor의 개수만큼 나온다. 이때 실제 anchor box의 영역으로 나눠주는데, 그 이유는 동일 엄청 큰 box에서의 loss와 작은 box에서의 loss가 같다고 가정할때, 당연히 적은 개수의 pixel로 loss를 계산한 작은 box가 pixel-wise하게 더 큰 loss를 갖기 때문이다.

이렇게 각각의 box 영역넓이로 loss를 나누고 positive anchor의 개수만큼 나온 loss들을 다 더한다. 마지막으로 prototype넓이인 (N\*N)로 나누고 Mask Loss에 부여된 가중치 alpha를 곱한다.

Class loss

Class loss는 생성된 19248개의 모든 anchor들에 대해서 softmax를 이용해서 classification loss를 구한다. positive(= class level), negative(0), neutral(-1)을 이용해서 negative 중에서 confidence가 높은 것들과, positive 것들만을 이용해서 정답데이터가의 loss를 계산한다.

**5. Control**

해당 과정에서 검출된 Lane 부분의 Data들과, Object Data들을 이용해서 Control하는 부분을 구현하면 되는데 아직 구현이 돼있지 않은 상태이다.

**6. Execution**

위 프로젝트를 실행하기 위해서는, 환경 설정이 필요하다.

먼저, Image Grab을 하기 위해서는 pillow library, cv2 library가 필요하다.

*“pip install pillow”*

*“pip install opencv-python”*

다음으로, Lane Detection은 cv2를 주로 활용하기 때문에 위 과정을 진행했다면 특별한 library가 필요하지 않다.

다음으로, Yolact를 위한 환경 세팅이 필요하다 이에 대한 자세한 내용은 [Yolact Github](https://github.com/dbolya/yolact)를 참고하여 설치하길 바란다. 필자는 Anaconda를 이용해서 간단히 설치하였다.

이렇게 Object Detection을 설치하고 나서, python main.py를 실행하면 프로젝트가 실행된다.