CHAPTER 25

Object Detectors

25.1 DETECTOR ARCHITECTURES

25.1.1 Convolutional: Faster R-CNN

Faster R-CNN is a model object detector that exploits the principles of Section 24.4.1. It was originally described in *****. There are three main steps: sample the world of boxes; decide what boxes are worth looking at; and decide what, if anything, is in each box and precisely where the box is. Faster R-CNN is built on top of a convolutional image encoder, and is somewhat agnostic as to which encoder you use.

Sampling the world of boxes: Specifying an axis aligned box in 2D requires four coordinates (for example, top left and bottom right corner). But configuration – equivalently, the world of axis aligned boxes – needs to be sampled carefully. Represent a box by two position variables giving the center, an aspect ratio and a scale, yielding a four dimensional configuration space. To build a set of sample locations, construct a grid of reasonably spaced locations in the image (for example, on a stride of 16 pixels). For a given image, assume this grid is $N_x \times N_y$. At each location in this grid, place a fixed number of different boxes. The original paper used three aspect ratios (square, short and wide, tall and thin) and three scales (small, medium and big) for a total of nine boxes, referred to as *anchor boxes*. These anchor boxes will serve as approximate configurations for the detected object; the configuration will be refined during the detection process. Write k for the number of anchor boxes, and assume there will be no more than one object per anchor box.

What boxes are worth looking at: You should see this as a classification problem – is there something in this box? – linked to a regression problem – by how much should the box be offset to fit the object better? The component of the detector that produces this information is usually referred to as a *region proposal network* or *RPN*. It should produce two blocks of data. One – the *cls* block – is $N_x \times N_y \times (2k)$, and scores for each anchor box whether there is or is not an object in that box. The other – the *reg* block – is $N_x \times N_y \times (4k)$, and encodes an offset from the anchor box for each anchor box. You should think of this offset as giving a somewhat improved location of the anchor box. These feature blocks encode all possible object boxes (recall there is only one object per anchor box) and the extent to which there is an object in each. These two blocks are easily obtained from the output of an encoder. Pass the encoding through a convolutional layer and a ReLU, then pass the result through two 1×1 convolutional layers, one to produce the *cls* block and the other to produce the *reg* block. Then N_x, N_y and the spacing of the anchor boxes are all determined by the encoder. Now identify which boxes in the *cls* block are worth looking at – for example, you could choose the 300 anchor boxes with the top objectness score. For these boxes, apply the offset in the *reg* block to the original configuration to obtain an improved configuration estimate. These

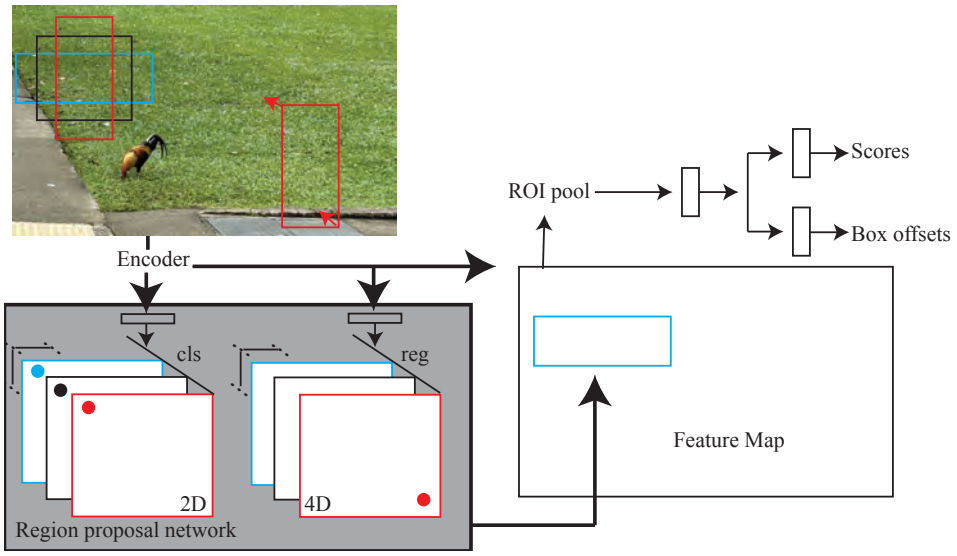

FIGURE 25.1: *Faster R-CNN takes a feature description from an encoder, and produces a set of boxes with object category scores for each. Anchor boxes (colored boxes on* **top left** *of image) are samples in configuration space. Their $N_x \times N_y$ centers are on an x-y grid, and there are three aspect rations (shown) and three scales (suppressed). A region proposal network (RPN) produces object/no object scores for each box in a* cls *block (size $N_x \times N_y \times (2 \times 9)$, so two scores per box; the* **color** *shows how a location in* cls *corresponds to a box in an image). The RPN produces a* reg *block consisting of four offset values for each box (size $N_x \times N_y \times (4 \times 9)$; the* **red** *shows how the offset adjusts the basic anchor box). The boxes are supplied to a ROI pooling layer, which computes a fixed size vector for each box from the piece of the encoded image that corresponds to the underlying box. This vector is passed to a fully connected layer that constructs a large feature, which in turn is fed to two fully connected layers. One produces a score for each category in a box, and the other produces a per category offset for the box.*

boxes must be assessed for what object lies inside them.

What is in the box, and precisely where: The final outcome of the RPN is a $B \times 4$ data block that describes B axis aligned boxes that are worth looking at. Given this block and a feature encoding of the image, you must decide what, if any, objects present and where they are. The procedure that does so is *Fast R-CNN*. Each entry in the block represents an *ROI* (for region of interest). For each ROI, an *ROI pooling layer* constructs a feature of fixed size $H \times W$ from the feature encoding. Write $h \times w$ for the size of the ROI. Assume that the feature encoding of the image is registered to the image. It is likely spatially smaller than the image; for the moment, assume it is smaller by a factor of 16 in each direction. Now cut an $\lfloor (h/16) \rfloor \times \lfloor (w/16) \rfloor$ window corresponding to the ROI out of the feature encoding ($\lfloor x \rfloor$ is the largest integer smaller than x). Divide this into a set of $H \times W$ boxes, each of which will be about $(\frac{h}{H}) \times (\frac{w}{W})$ in size. In each of these windows, choose

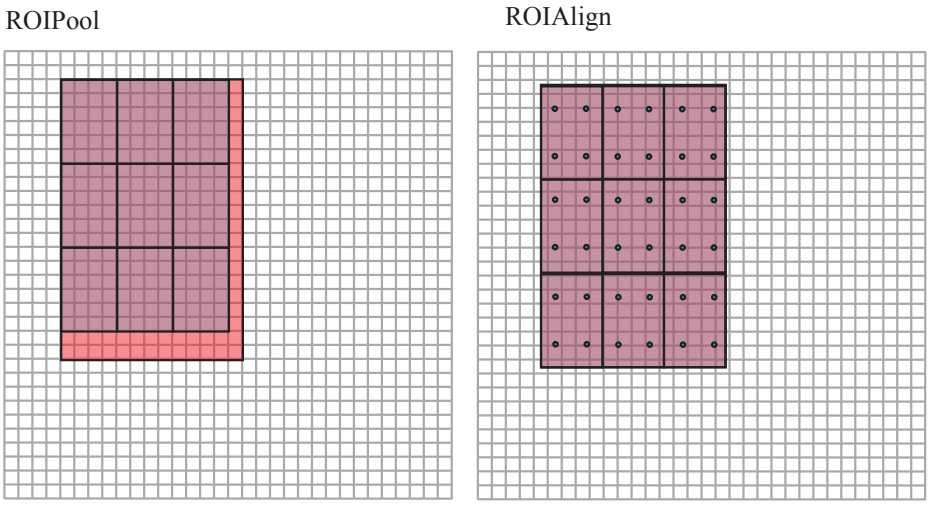

FIGURE 25.2: *On the* **left**, *ROIPool reduces a feature map to a fixed size feature by max pooling in a set of boxes (the translucent* **gray** *boxes). The feature sample is located at the center of each box. The ROI on the feature map (the translucent* **red** *box) is determined by rounding as is the placement of the pooling boxes, leading to a loss of spatial information. Notice how some parts of the ROI don't contribute, because of the rounding. On the* **right**, *ROIAlign places the ROI exactly on the feature map, samples at a grid of points (***blue***) inside each box using bilinear interpolation, then pools these samples inside each box. This preserves spatial information. Numerical details in the text.*

the largest value (equivalently, max-pool). Now straighten the $H \times W$ values into a vector that describes the ROI.

Figure 25.2 shows what happens when a 1024×1024 image produces a 32×32 feature map which is ROIPooled to a 9 dimensional vector for a given ROI. The top left corner of the ROI is at $(75, 140)$ in the original image, and the size of the ROI is 650×430. In this case, ROIPool would round the corner location to $(2, 4) = (\lfloor 75/32 \rfloor, \lfloor 140/32 \rfloor)$, the size to $20 \times 13 = \lfloor 650/32 \rfloor \times \lfloor 430/32 \rfloor$.

Pass the vector describing the ROI through some fully connected layers to produce a feature describing the ROI. Now you want a score for each object and an offset to ensure the box is accurately placed. Pass the feature through a fully connected layer to produce a C dimensional score vector – this is the score for the object. Pass the feature through a different fully connected layer to produce a $C \times 4$ dimensional offset vector. There is one offset per category.

You now have raw detector output, which will need some housekeeping. A single object could overlap several anchor boxes, and so output boxes require some non-maximum suppression. This is relatively straightforward: if two boxes have an IOU over some threshold, reject the box with lower score. The final position of a box is given by offsetting the original box with the category dependent offset from the highest scoring class. Finally, only boxes where the score for the highest scoring

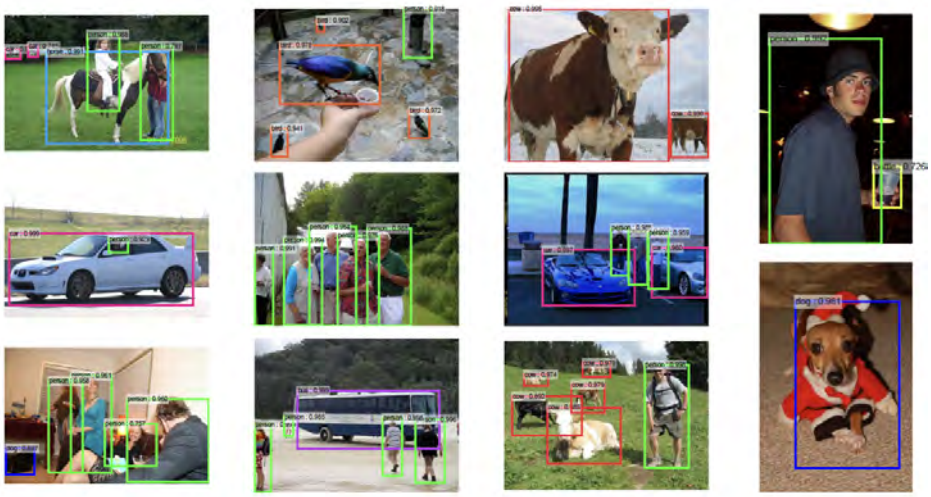

FIGURE 25.3: *Detection results using Faster R-CNN on parts of the PASCAL VOC 2007 test set.* **Image Credit:**Figure 2 of Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun

object category is over some threshold are reported.

Training Faster R-CNN requires some care. Usually, the two are built on top of a pretrained image encoder. The RPN and Fast R-CNN do somewhat different things (score all boxes for objectness vs decide what is in a box). Experience suggests that training the two together with SGD is relatively fast, but produces slightly worse results than a more elaborate strategy.

Start by training the RPN end-to-end, initializing with a pre-trained encoder using an image centric strategy. Select the images for the batch; now you must select labelled anchor boxes in the chosen ratio. Many anchor boxes overlap the image boundaries, and experience shows these should be ignored in training. The remaining anchor boxes are of three kinds. Positive boxes either have IoU with a ground truth box greater than 0.7 *or* are the box with the largest IoU with some ground truth box. Negative boxes have an IoU less than 0.3 with every ground truth box. All others are indeterminate, and are ignored in training. To force the boxes to get the right labels, use a cross-entropy loss between the objectness scores and the true labels for the selected anchor boxes. For positive anchor boxes, use a regression loss to force the *reg* block to get the right offset. Notice the encoder weights will change to produce a better RPN.

Now train the Fast R-CNN end-to-end, using the proposals generated by the RPN and initializing with a separate copy of the pre-trained encoder. Train this to produce the correct object label in each positive box, and to predict the correct offset. Again, the encoder weights will change.

Now fine-tune the RPN layers, using the encoder that resulted from

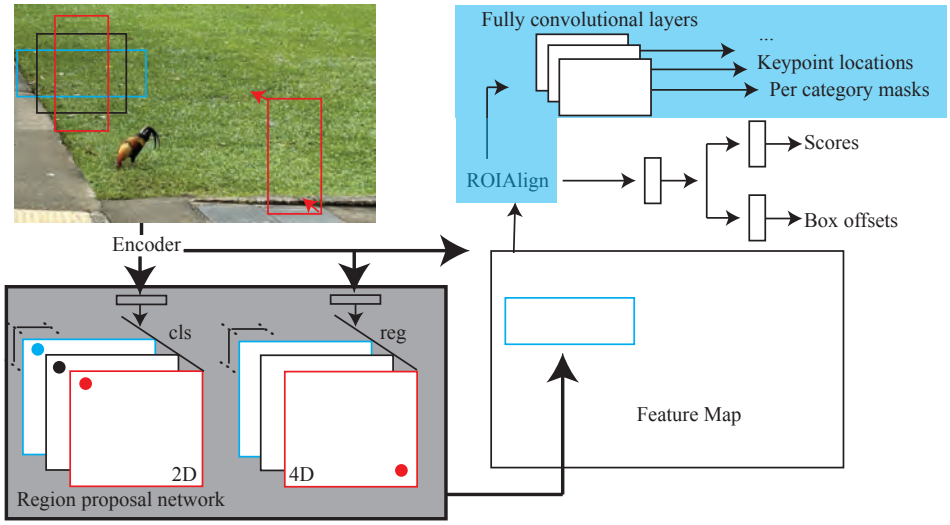

FIGURE 25.4: *MaskRCNN involves small but significant modifications to Faster-RCNN which are indicated in the* **blue** *boxes (compare to Figure 25.1). ROIPool is replaced with ROIAlign; and the results are passed through a fully convolutional network to produce a set of fixed size masks, one per category. This procedure can be applied to other predictions as well. I describe keypoint predictions in the text, but the figure leaves room for other possibilities.*

training Fast R-CNN.

Finally, fine-tune Fast R-CNN, using the encoder and the RPN from the previoius step.

Remember this:

25.1.2 Convolutional: YOLO

All the detectors I have described so far come up with a list of boxes that are likely to be useful. *YOLO* (You Only Look Once) is a family of detectors (variants pay off accuracy against speed, and incorporate a variety of speedups and improvements) that uses an entirely different approach to boxes. The image is divided into an $S \times S$ grid of tiles. Each tile is responsible for predicting the box of any object whose center lies inside the tile. Each tile is required to report B boxes, where each box is represented by the location of its center in the tile together with its width and its height. For each of these boxes (write b), each tile must also report a box confidence score $c(b(\text{tile}))$. The method is trained to produce a confidence score of zero if no object has its center in the tile, and the IoU for the box with ground

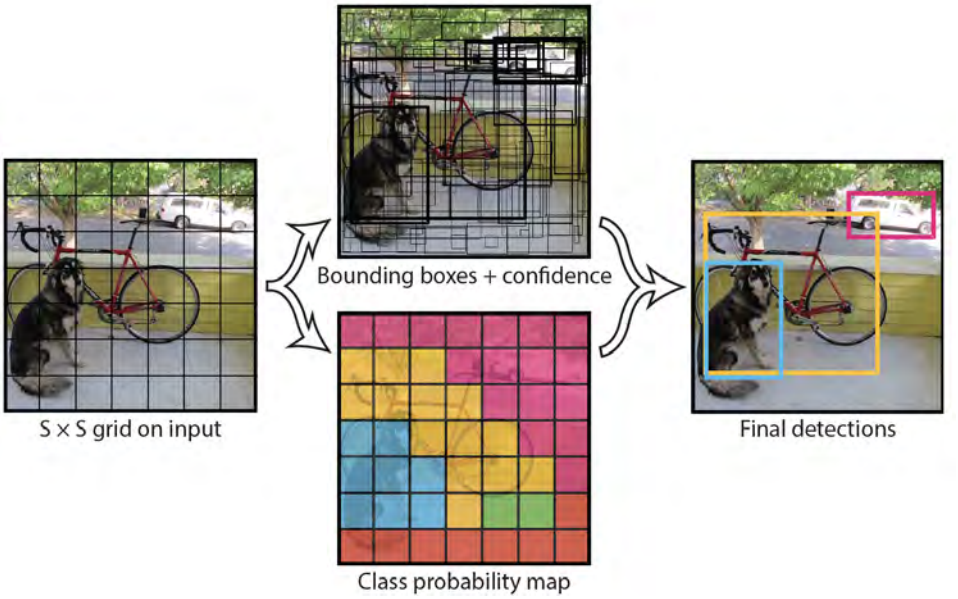

FIGURE 25.5: *YOLO predicts a confidence score for a set of B boxes in each of an $S \times S$ grid of tiles together with a class posterior score for each object category at each tile. These scores are combined to produce the score for boxes. The resulting boxes are then post processed (non-maximum suppression) and reported.* **Image Credit:**Figure 2 of You Only Look Once: Unified, Real-Time Object Detection Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi

truth if there is such an object (of course, at run time it might not report this score correctly).

Each tile also reports a class-posterior, $p(\text{class}|\text{tile})$ for that tile (Figure 25.6). The score linking each of the boxes b in a tile to a class is then computed as

$$c(b(\text{tile})) \times p(\text{class}|\text{tile}).$$

Notice how the box scoring process has been decoupled from the object class process. Each tile is scoring *what* object overlaps the tile and also scoring which boxes linked to the tile are important. But these scores are computed separately – the method does not know which box is being used when it computes the object scores. This means the method can be extremely fast, and YOLO offers relatively easy tradeoffs between speed and accuracy, which are often helpful (for example, one can use more or fewer network layers to make features; more or fewer boxes per tile; and so on).

Decoupling boxes from classes comes with problems. YOLO tends to handle small objects poorly. There is a limited number of boxes, and so the method has difficulties with large numbers of small objects. The decision as to whether an object is present or not is based on the whole tile, so if the object is small compared to the tile, the decision might be quite inaccurate. YOLO tends not to do well with

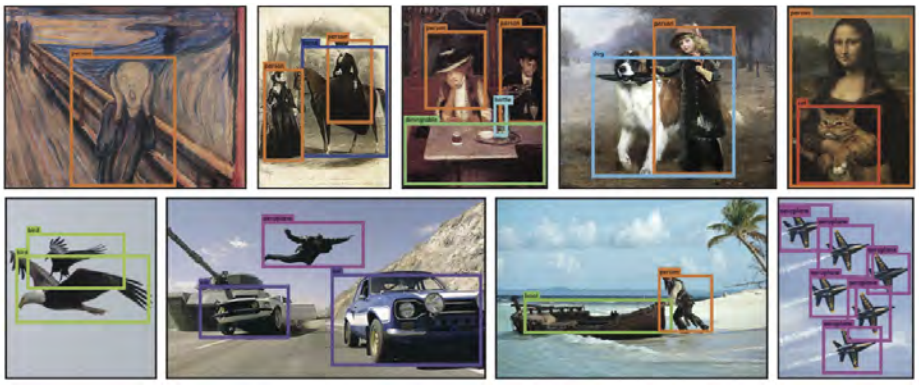

FIGURE 25.6: *I can't improve on the caption of the YOLO paper for this figure: "YOLO running on sample artwork and natural images from the internet. It is mostly accurate although it does think one person is an airplane."* **Image Credit:**Figure 5 of You Only Look Once: Unified, Real-Time Object Detection Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi

new aspects or new configurations of familiar objects. This is caused by the box prediction process. If the method is trained on (say) all vertical views of trees (tall thin boxes), it can have trouble with a tree lying on its side (short wide box).

Remember this: *The YOLO family of detectors work very differently from the R-CNN family. In Yolo, image tiles produce objectness scores for boxes and a classification score for objects independently; these are then multiplied. The advantage is speed, and tunable payoffs between speed and accuracy. The disadvantages are that many small objects are hard to detect, and new configurations of familiar objects are often missed.*

25.1.3 Convolutional: MaskRCNN

It's useful to put boxes on objects, but even more useful would be a *mask*, specifying the extent of the object in as much detail as possible. MaskRCNN adds a head to Faster R-CNN to predict masks. Doing so requires a small but very important modification to ROI-pooling, because ROI-pooling suppresses close detail about object location. This is because it quantizes location. ROI-pooling places a mask the size of the ROI on the feature grid, *at an integer location*, and fills in pooled values by max-pooling everything in each box of the mask. Doing so suppresses fine detail in the location information in at least two ways: first, the mask is placed at an integer location, so might be shifted from where it should be; second, the values pooled in each cell of the mask are the values that lie inside the cell. This is

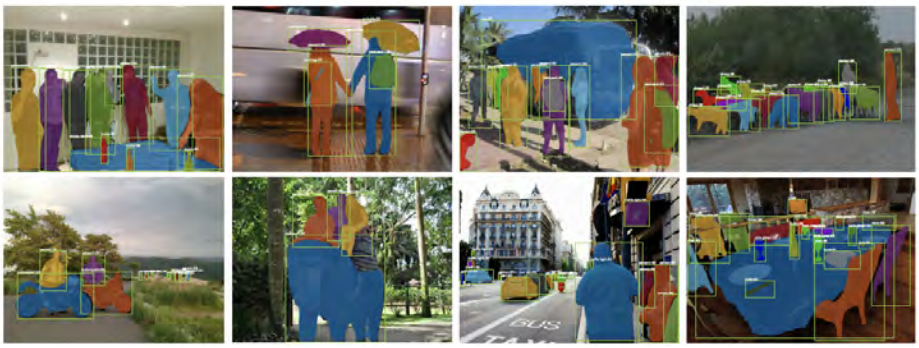

FIGURE 25.7: *MaskRCNN produces both object boxes and masks, which give a detailed domain of support for each object. Results here are on parts of the PASCAL VOC 2007 test set.* **Image Credit:**Figure 7 Mask R-CNN Kaiming He Georgia Gkioxari Piotr Doll´ar Ross Girshick

not a problem if you want to report boxes – small localization effects shouldn't be important – but it is a real problem if you want to report accurate masks.

You could preserve more detailed information by sampling an interpolated version of the feature map, then pooling those samples. This is *ROIAlign*. Rather than rounding locations, ROIAlign places an ROI that doesn't have to have integer size, at a location that doesn't have to be an integer location. The ROI is subdivided into a fixed size grid of boxes (3×3 in the Figure), where the size of the boxes doesn't have to be integer either. In turn, each box contains four sample points, where samples from the feature map are bilinearly interpolated. The values are pooled to yield a value for the box. Notice how, in Figure **??** which shows the difference between ROIAlign and ROIPool, the samples are more precisely recovered from the feature map. Furthermore, the whole ROI contributes to the feature. One might expect aliasing effects **exercises**, but I am aware of no evidence they occur.

Rather than straightening the feature that ROIAlign produces into a vector (which could lose spatial information), MaskRCNN passes it into a series of convolutional layers that produce a stack of $m \times m$ masks, one per category. At inference time, the category chosen for a box then chooses its mask as well. Training is very like training FasterRCNN, but the embedding is fixed; the RPN is trained separately from the FastRCNN head; and there is a loss requiring that the mask for the true category be right. The ROIAlign layer produces a spatially meaningful feature that is accurately aligned to the original image and to an object proposal. This feature can be used to produce other predictions than masks. Figure **??** shows keypoint predictions. These are predicted (like the mask) by an fully convolutional network from the feature, and trained (like the mask) by imposing a loss that requires results are accurate.

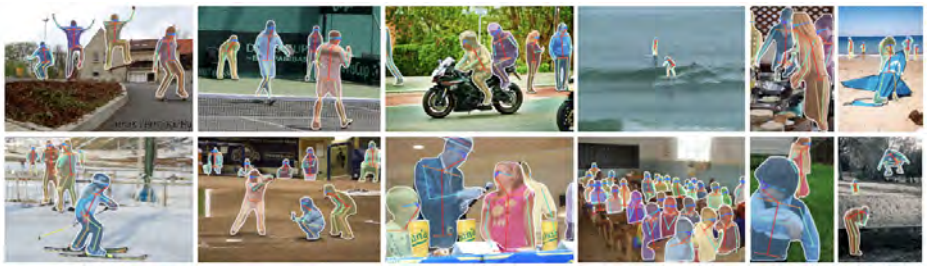

FIGURE 25.8: *The feature produced by ROIAlign can be decoded into a range of useful predictions. This figure shows masks predicted for humans. On top of the masks are predicted locations for* keypoints *(points on the body, typically at joints). These are obtained by passing the ROIAlign feature through a convolutional network and interpreting the resulting data block as keypoint locations.* **Image Credit:**Part of Figure 5 of Mask R-CNN Kaiming He Georgia Gkioxari Piotr Doll´ar Ross Girshick

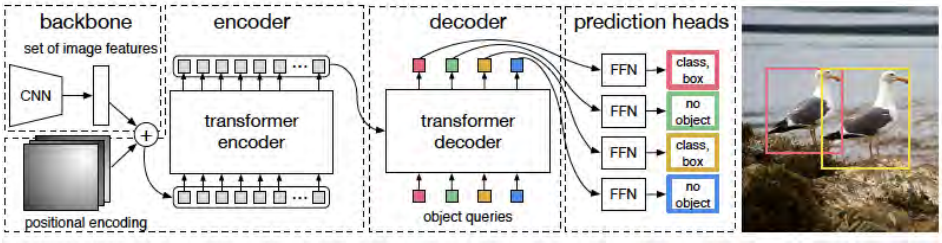

FIGURE 25.9: **Image Credit:**Part of Figure 1 of End-to-End Object Detection with Transformers Nicolas Carion?, Francisco Massa?, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko

25.1.4 Transformers: DetR

Chapter 21.3 described transformer encoders in some detail. A transformer can detect objects in a very natural way. Think of transformer as something that maps a sequence of image windows to a sequence of latent tokens. Assume that no image contains more than some maximum number of object instances, N (for example, $N = 100$ is quite practical). You could then build a transformer that produces a sequence of N latent tokens, and decode each into either a "nothing here" statement or the category and location of an object.

If you simply tokenize image windows to form the input sequence, the transformer is permutation equivariant (permute the order of the input tokens, and the output tokens will permute in order). This means the position of an input token in a sequence does not carry information about where it is in an image. But it is important for a detector to have information about where objects are, so when you tokenize the image windows, you should attach a positional encoding to the tokens.

Each token is individually decoded into a prediction.

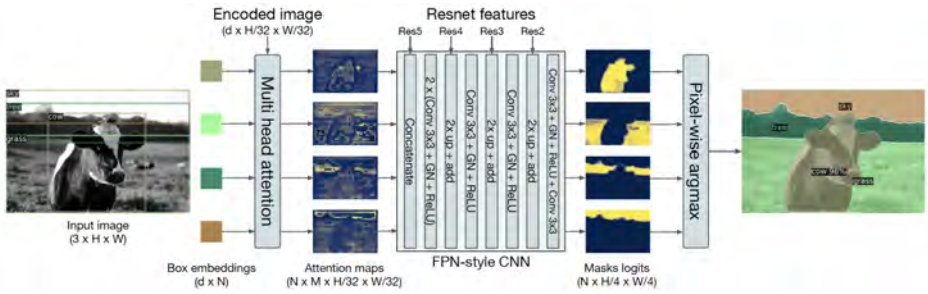

FIGURE 25.10: **Image Credit:**Part of Figure 8 of End-to-End Object Detection with Transformers Nicolas Carion?, Francisco Massa?, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko

tokens - predictions

Training this detector requires care, because you cannot know which token that the transformer produces should correspond to which of the true positive boxes. You can deal with this by using a bipartite matching strategy. Pass a training image through the transformer, to get a set of N tokens. Pass each through the decoder to get a prediction (y_i for the i'th token). For simplicity, ensure that there are N ground truth boxes by appending "nothing here" boxes (which have no location) as required. Write $\hat{y}_j$ for the j'th ground truth box, and $\mathcal{L}(y_i, \hat{y}_j)$ for a weight that compares the prediction of the i'th token with the j'th ground truth box. For simplicity, assume this weight is larger for a better match. You must now choose the matching between predicted tokens and ground truth boxes that maximizes the sum of weights. This is a *maximum weighted bipartite matching* problem. The so-called *Hungarian algorithm* solves this problem exactly in $O(N^3)$ time (a greedy algorithm gives quite a good estimate in $O(N)$ time **exercises**). Once you have the best matching, you can compute loss and backpropagate.

, the input sequence of image windows should carry information about where they are.

25.1.5 Generalizing to Panoptic Segmentation

25.2 RESOURCES

Detectron2, YOLO, Detr

`https://github.com/lyuwenyu/RT-DETR` `https://github.com/facebookresearch/detr`

FIGURE 25.11: **Image Credit:**Part of Figure 9 of End-to-End Object Detection with Transformers Nicolas Carion?, Francisco Massa?, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko

CHAPTER 26

Transformer Encoders for Images

This chapter describes a new procedure to encode an image which has deep roots in the ideas of Chapter 10. The encoder represents an image as a collection of patches, and builds a representation out of patch relations. Remarkably, the encoder does not use any explicit representation of spatial relations between the patches. In fact, it originates in the *transformer* used to encode and transform word sequences in natural language processing. A *vision transformer* or *ViT* can be used in slightly different forms for image to image mapping and for image classification.

Vision transformers are now very widely used. They integrate well with language representations, so they simplify building methods to map pictures to captions or text prompts to images. Experience shows vision transformers produce superior encodings of images for most tasks. Early efforts to use transformers for vision problems tended to work poorly, apparently because it took some time to realize just how much data (immense amounts) was required to train a vision transformer that worked well. Vision transformers also are very demanding of computation and change the nature of the computations required compared to convolutional encoders, so the performance gains may not be justified

26.1 VISION TRANSFORMER ENCODERS

The original transformer is an architecture that encodes a sequence of tokens into another sequence of latent representations. Tokens can be pretty much anything that can be represented by a fixed length vector. The first tokens were word embeddings – representations of words as vectors. The latent representations are then decoded into an output sequence. The decoder can be constructed so that the output sequence is *auto-regressive* – the $i+1$'th generated token is a function of the input and all or some earlier tokens. This property is extremely helpful if you intend to generate long sequences of words (paragraphs, books, etc) because it means you do not need to look at future words when you make the current word.

Figure 26.1 shows the overall structure of a vision transformer. The image is mapped into a set of N_t tokens. These tokens pass through a series of transformer layers, and then on to a decoder. The key feature of a transformer layer is attention. The layer turns a set of tokens into another set of tokens, *where each token in the resulting sequence is affected by every input token.* The output of the encoder will be a set of N_t tokens. There are various ways to decode this result, depending on the application.

26.1.1 Attention

The key operation in a transformer maps a token sequence to a token sequence. A token sequence is represented by a matrix. Each image will be tokenized into N_t tokens $\mathbf{v}_1 \dots \mathbf{v}_{N_t}$. It turns out to be helpful to have an extra, learnable, token $\mathbf{v}_0$.