Deep Learning and Transformer

Shenghao XU

April 16, 2024

Abstract

This report we will mainly focus on two parts: Deep learning based object detection and attention based Transformer. We will discuss two-stage methods (i.e., Faster R-CNN) and single-stage methods (i.e., YOLO series). Two of the most profound ideas are discussed: *better underlying networks* & *fusing features from different convolutional layers*, both of which are superimposed on the standard approach, giving birth to a large number of variants. Then, we will continue to discuss and analyze the latest variants of the approach and try to find the optimal approach for our own purposes. We will focus on the latest variant of the YOLO series- *YOLOX*. and compare its advantages and disadvantages with various YOLO methods. Finally, Transformer will be discussed, we will also discuss its advantages and disadvantages compared to deep learning based methods.

1 Introduction

Deep learning has helped objects detection reach an unprecedented peak. Since 2014, objects detection frameworks are divided into two main categories: two-stage and one-stage, the former represented by the classical method *Faster R-CNN*, the latter by *YOLO* and *SSD* as the main frameworks. In recent years, the two most profound ideas: better underlying networks (i.e., VGG, ResNet) or fusion the features from different convolutional layers superimposed on the classical method, have produced a large number of variants.

The Transformer, initially, the architecture was proposed for the task of machine translation. Inspired by the use of the self-attentive mechanism in Transformer to modeling of dependencies in text, many tasks in the field of computer vision propose to use the self-attentive mechanism to effectively overcome the problems in Convolutions.Transformer has revolutionized the field of computer vision by enabling significant improvements in the areas of object detection, video classification, image classification, and image generation. Some of these models using Transformer architecture can meet or exceed the results of state-of-the-art solutions in this field.

The following article will be arranged as follows, we will conduct a general review in the section 2, followed by two most profound ideas in section 3. In the section 4, we will analyze variants of the major classical methods. Finally, the Transformer will present in the section 5.

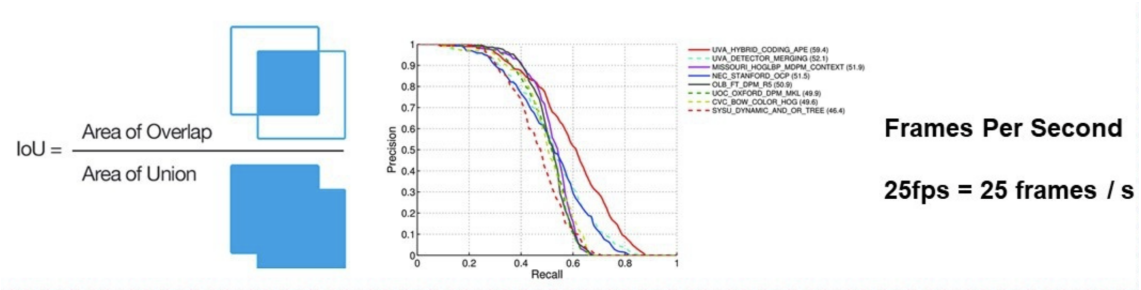


Figure 1: Three metrics for object detection

2 General Review

2.1 Task description for object detection

The so-called object detection is actually a subordinate task in computer vision. Its goal is to locate objects in an image and give their specific category. In autonomous vehicles, intelligent surveillance, object detection is crucial. As a criterion for judging a object detection system, we have three important metrics as shown in Fig.1 .

1. IoU, which is an indicator to identify how close the predicted box is to the ground-truth box.
2. mAP (mean Average Precision), in the detection of multiple categories, each category can adjust the threshold, calculate the *precision* of the *recall* from 0 to 1 (under the same recall, the highest precision will be chosen), calculate the average of the precision, and then for all classes to average to get mAP.
3. FPS (frames per second) used to judge how fast the system is.

The goal of the object detection method is: more accurate localization, faster and more precise classification.

With these metrics, data sets are also needed to evaluate the methods. The two most common types of generic data sets are described here:

1. PASCAL VOC dataset. There are 20 categories in this dataset, such as person, bird, cat, bicycle, etc.
2. Microsoft's COCO (MS COCO) dataset, a total of 80 categories in this dataset, is geared towards indoor, outdoor.

They are mostly oriented to such life scenarios. There are also specific datasets for domain-specific applications in objects detection, for example PLACES [Zhou et al., 2014], the scene recognition data set. The abundant and huge dataset has greatly driven the development of the object detection field. This is accompanied by a constant increase in computing capacity and methods improvements. In the next section, outstanding object detection frameworks/methods will be discussed

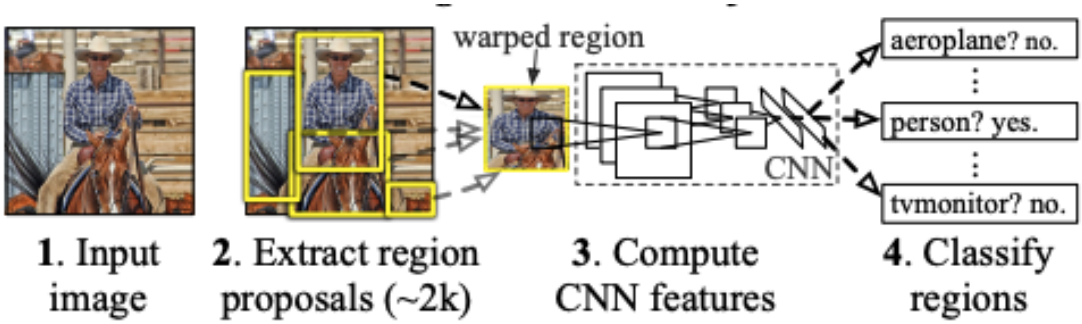


Figure 2: Two-stage framework:R-CNN Object Detection System

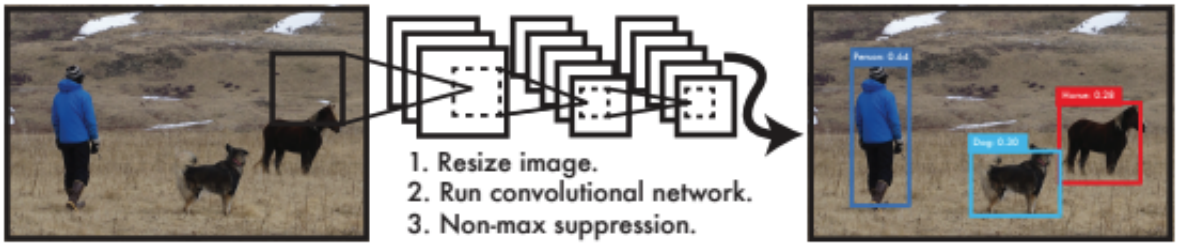


Figure 3: One-stage end-to-end framework:The YOLO Detection System

2.2 Classical Object Detection Methods

The overall objective detection framework is also divided into two categories as following:

1. Two-stage framework: first for region proposal and subsequently for object classification,as shown in Fig.2 [Girshick et al., 2014].

2. One-stage end-to-end framework, which applies a network to do everything and outputs results in one step ,as shown in Fig.3 [Redmon et al., 2016].

The proposal of the R-CNN framework in 2014 was one of the first approaches to apply convolutional networks to objects detection, and is also the earliest two-stage framework. The overview of the R-CNN system is shown in the Fig.2, the system (1) takes an input image, (2) extracts around 2000 bottom-up region proposals, (3) computes features for each proposal using a large convolutional neural network (CNN), and then (4) classifies each region using class-specific linear SVMs. To sum up, the R-CNN The region proposal is first performed using a selective search image segmentation algorithm, and then the regions are cropped down and scaled to feed the convolutional network to make the decision. On the PASCAL VOC dataset, it boosts the mAP by more than 20% at once, but this results in a slow speedup because the proposed regions are usually more than 2000 [Girshick et al., 2014].

The Fast R-CNN [Girshick, 2015] (Fig.4) improves on the R-CNN by adding a region pooling layer to the last layer of the convolutional layer, allowing the proposed regions (numerous candidate object locations) to be mapped directly to the pooling layer to extract features, reducing thousands of repetitive convolutional operations and greatly improving speed.

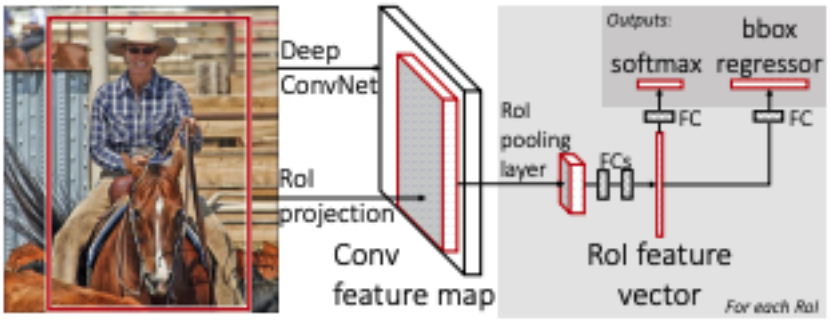


Figure 4: Fast R-CNN architecture

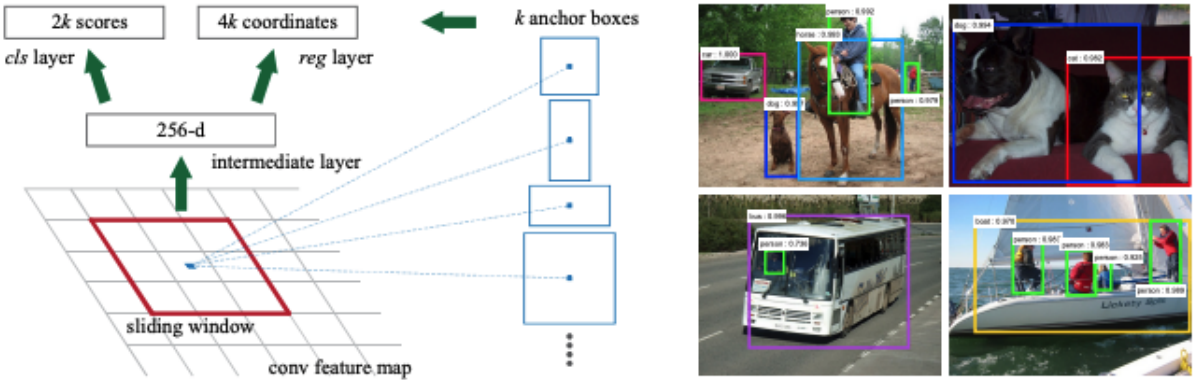


Figure 5: Faster R-CNN architecture

However, for Fast R-CNN, its region proposal method still uses selective search, which cannot share computation with CNN, and the speed bottleneck still exists. Faster R-CNN [Ren et al., 2015] (Fig.4) introduces the region proposal network (RPN), which uses CNN to perform this step as well, and lets the region proposal network and classification network share convolutional features, making the object detection framework almost real-time and reaching a very high mAP.

These three methods are all two-stage methods and can be summarized as follows:

1. R-CNN [Girshick et al., 2014] (2014): Pioneering work, 20% mAP improvement over traditional methods on PASCAL VOC dataset.

2. Fast R-CNN [Girshick, 2015] (2015): Reduce redundant computations by adding a region of interest (RoI)pooling layer.

3. Faster R-CNN [Ren et al., 2015] (2016): Region proposal via RPN network, considered advanced object detection framework up to now.

Next we will discuss the One-stage end-to-end framework, as shown in Fig.3, the YOLO system (1) resizes the input image to 448 × 448, (2) runs a single convolutional network on the image, and (3) thresholds the resulting detections by the model's confidence [Redmon et al., 2016]. YOLO divides the image into S × S cells (448 × 448 in Fig.3), each cell goes to predict the boxes separately, and uses the network to learn the real boxes and the categories to which they belongs directly for these

boxes.YOLO is very fast, with small models reaching 155 fps, but with this comes a reduction in mAP and the problem of inaccurate positioning.

3 The idea that can improve the performance of the object detector

3.1 Fusing the features of different convolutional layers

There is such an idea in SSD(Single Shot MultiBox Detector)[Liu et al., 2016] that the lower receptive field is small, and it is used to detect small objects, and the higher receptive field is larger, and it is used to detect large objects. However, it ignores the fact that the semantic information of the low convolutional layer is weak and cannot help well in subsequent classification, resulting in a detection improvement of small objects that is actually not significant. Due to the above reason, HyperNet [Kong et al., 2016] has been proposed,HyperNet takes a fused convolutional feature approach. What if the feature maps have different sizes? The larger ones are pooled and reduced, and the smaller ones are up-sampled and expanded, and then fused. With the above improvement strategy, HyperNet can guarantee a high *Recall* when generating about 100 region proposals, and the mAP of object detection is improved by about 6 % compared to Fast R-CNN.

However, the feature maps fused in this way by HyperNet have some losses in both spatial and semantic information.In 2017 FPN (Feature pyramid networks) network [Lin et al., 2017] was proposed, which uses a feature pyramid approach to both use fused features for detection and let the prediction be performed independently in different feature layers. In the above we mentioned that the semantic information of the low convolutional layer is weak and cannot contribute well to the subsequent classification, leading to the detection improvement of small objects is actually not much. Then FPN tries to compensate for the low layer with the semantics of the high layer, and achieves fairly good results. As shown in the figure above, there are 4 forms of utilizing features: (a) image pyramids, Features are computed on each of the image scales independently, which is slow, (b) features using only one layers for faster detection, (c) SSD style, and (d) FPN style,is fast like (b) and (c), but more accurate [Lin et al., 2017]. As we can see, FPN seems to be a combination and variant of SSD and HyperNet. Its path on the right side is usually called top-down path.It is worth noting that FPN is not a objects detection framework, and this structure can be incorporated into other object detection frameworks to improve the performance of the object detector.

3.2 A better underlying network

The second idea, a better underlying network. As we mentioned the deep residual network in last week. Traditional convolutional neural network models increase the depth of the network by cascading convolutional layers to improve detection accuracy. However, experiments have found that too deep networks suffer from degradation, as shown in Fig.7 [He et al., 2016] , with the network depth increasing, accu-

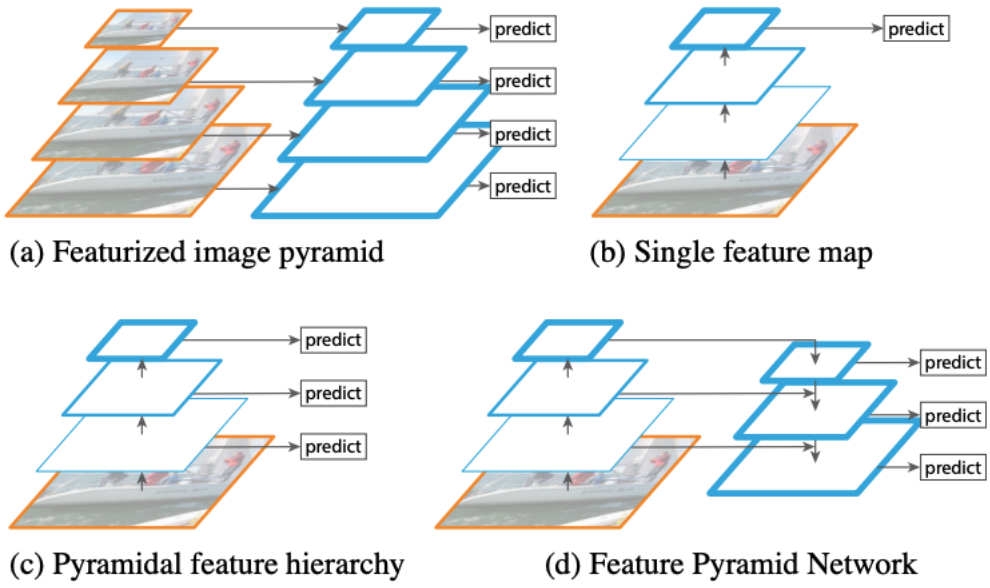


Figure 6: 4 forms of exploiting features

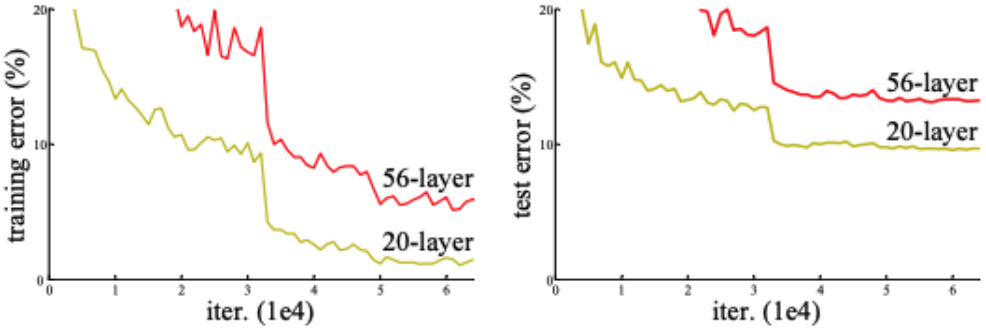


Figure 7: Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer networks.

racy gets saturated and then degrades rapidly.Such degradation is not caused by over-fitting,because if it is the case, deeper networks should only be higher in test error than shallow networks, not higher in both training and testing. And adding more layers to a suitably deep model leads to higher training error. In [He et al., 2016], authors address the degradation problem by introducing a deep residual learning framework, as shown in the Fig.8.Formally, denoting the desired underlying mapping as $\mathcal{H}(\mathbf{x})$, we let the stacked nonlinear layers fit another mapping of $\mathcal{F}(\mathbf{x}) := \mathcal{H}(\mathbf{x}) - \mathbf{x}$. The orig- inal mapping is recast into $\mathcal{F}(\mathbf{x}) + \mathbf{x}$. Formally, in this paper we consider a building block defined as:

$$\mathbf{y} = \mathcal{F}\left(\mathbf{x}, \{W_i\}\right) + \mathbf{x} \tag{1}$$

Here $\mathbf{x}$ and $\mathbf{y}$ are the input and output vectors of the layers considered.In short, the reason why ResNet can solve degradation is because residual learning can increase the gradient so that the gradient

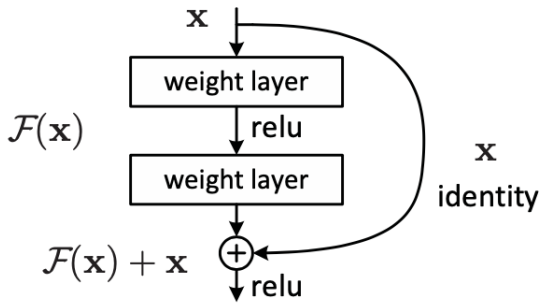


Figure 8: Residual learning: a building block.

does not become small, the proof is shown as follows:

$$\frac{\mathrm{d}L}{\mathrm{d}\mathbf{x}} = \frac{\mathrm{d}L}{\mathrm{d}\mathbf{y}} \times \frac{\mathrm{d}\mathbf{y}}{\mathrm{d}\mathbf{x}} \tag{2}$$

$$\frac{\mathrm{d}L}{\mathrm{d}\mathbf{y}} \times \frac{\mathrm{d}\mathbf{y}}{\mathrm{d}\mathbf{x}} = \frac{\mathrm{d}L}{\mathrm{d}\mathbf{y}} \left[\frac{\mathrm{d}\left(\mathcal{F}\left(\mathbf{x}, \{W_i\}\right) + \mathbf{x}\right)}{\mathrm{d}\mathbf{x}} \right] \tag{3}$$

$$\frac{\mathrm{d}L}{\mathrm{d}\mathbf{y}} \left[\frac{\mathrm{d}\left(\mathcal{F}\left(\mathbf{x}, \{W_i\}\right) + \mathbf{x}\right)}{\mathrm{d}\mathbf{x}} \right] = \frac{\mathrm{d}L}{\mathrm{d}\mathbf{y}} \left[1 + \frac{\mathrm{d}\left(\mathcal{F}\left(\mathbf{x}, \{W_i\}\right)\right)}{\mathrm{d}\mathbf{x}} \right] \tag{4}$$

where L denote the loss, and in equation.4, we can see that after taking the derivative, the gradient is 1 more than origin approach.

3.3 Deformable Convolutional Networks

The ResNet [He et al., 2016] solves the degradation problem (i.e., with the network depth increasing, accuracy gets saturated) of deeper networks. It is worth noting that a better underlying network is not only about how to stack the depth of the network, but also how to make the network better adapt to variations in objects, which is how to accommodate geometric variations or model geometric transformations in object scale, pose, viewpoint, and part deformation.

The CNNs is inherently limited to modeling large, unknown transformations. The limitations stem from the fixed geometry of the CNN module: the convolutional unit samples the input feature map at a fixed position; the pooling layer reduces the spatial resolution at a fixed ratio. How can we effectively model the variation of various poses of an object? One can certainly build training datasets with sufficiently diverse datasets, or use transformation-invariant features and algorithms, but there are always new and unknown geometric transformations, meanwhile, handcrafted design of invariant features and algorithms could be difficult or infeasible for overly complex transformations. In [Dai et al., 2017], the authors proposed two new modules that greatly enhance CNNs' capability of modeling geometric transformations.The first is *deformable convolution* and the second is *deformable RoI pooling*.Considering that convolutional kernels (filters) can extract features along what the object looks like (object's shape), it seems to solve the problem to some extent. This is the core idea of DCN (deformable convolution network), unlike normal convolutional kernels, it adds 2D offsets to

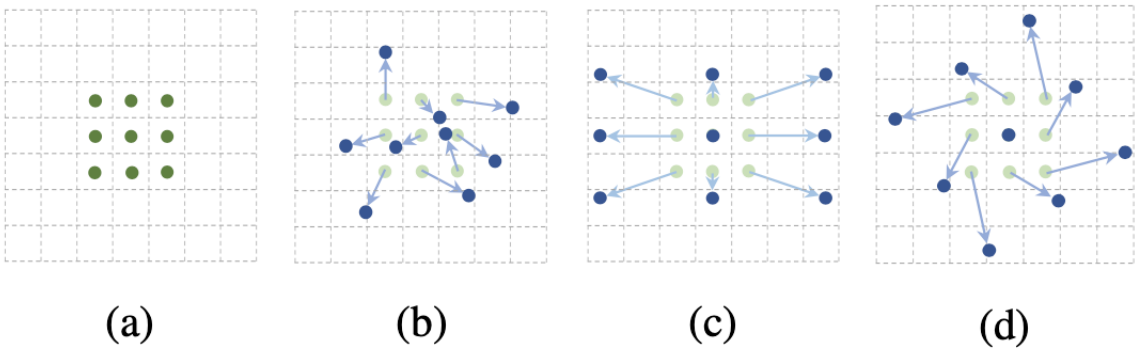


Figure 9: Illustration of the sampling locations in 3×3 standard and deformable convolutions.

the regular grid sampling locations in the standard convolution., and this offset is learned from the preceding feature map, by additional convolutional layers.

As shown in Fig.9, sub-figure (a) shows the regular sampling grid (green points) of standard convolution. (b) deformed sampling locations (dark blue points) with augmented offsets (light blue arrows) in deformable convolution. (c)(d) are special cases of (b), showing that if the offset can be learned, then it to be possible to represent/generalizes various transformations as scale (shown in c), aspect ratio and rotation(shown in d).

In short, a transposition variable is added to these convolutional or RoI pooling layers. This variable is learned according to the data, and after the offset, it corresponds to a scalable change in each square of the convolutional kernel/filter, thus changing the range of the receptive field, which becomes nearly a polygon.

This is illustrated in Figure 10 that for deformable convolution, one additional convolutional layer is needed to learn the offset which share the same input feature map. Then input feature maps and offset are jointly used as input to the deformable convolutional layer, and the deformable convolutional layer operates the sampling points to be offset and then convoluted.

How do apply it to the RoI Pooling layer? In this paper, authors proposed deformable RoI Pooling. Recall that RoI pooling is to unify the feature maps corresponding to different sizes $w \times h$ of RoI to a fixed size $k \times k$. In another word, RoI pooling divides the RoI into $k \times k$ (k is a free parameter) bins and outputs a $k \times k$ feature map $\mathbf{y}$. But for deformable RoI pooling, as shown in Fig 11, firstly, RoI pooling generates the pooled feature maps.From the maps, a fully connected (fc) layer generates the normalized offsets, then provide element-wise product with the RoI's width and height,and then pooling. In Fig 11, RoI is divided into 3×3 bins, which are fed to an additional fc layer to learn offset, and then operated by a deformable RoI pooling layer to make each bin offset.

In fact, the added offset in the deformable convolutional unit is part of the network structure and is computed by another parallel standard convolutional unit, which in turn can also be learned end-to-end by gradient back-propagation.After the learning of the offset, the size and position of the

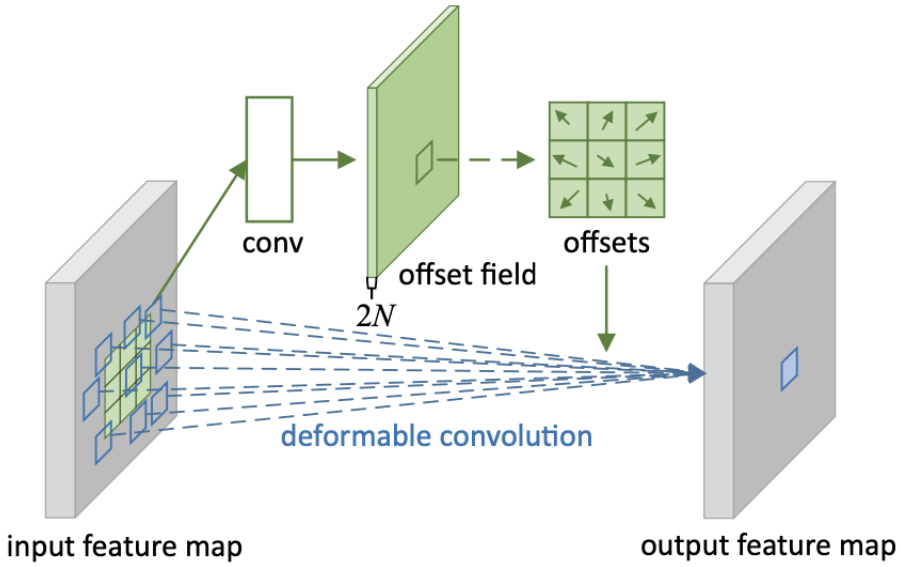


Figure 10: 3×3 deformable convolutions

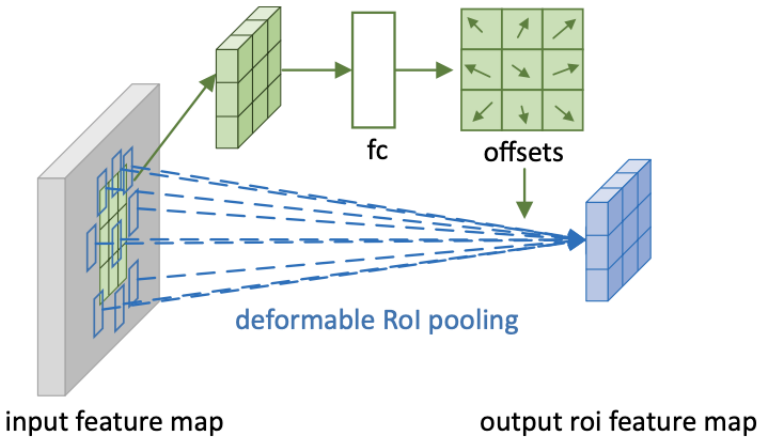


Figure 11: 3×3 deformable RoI pooling

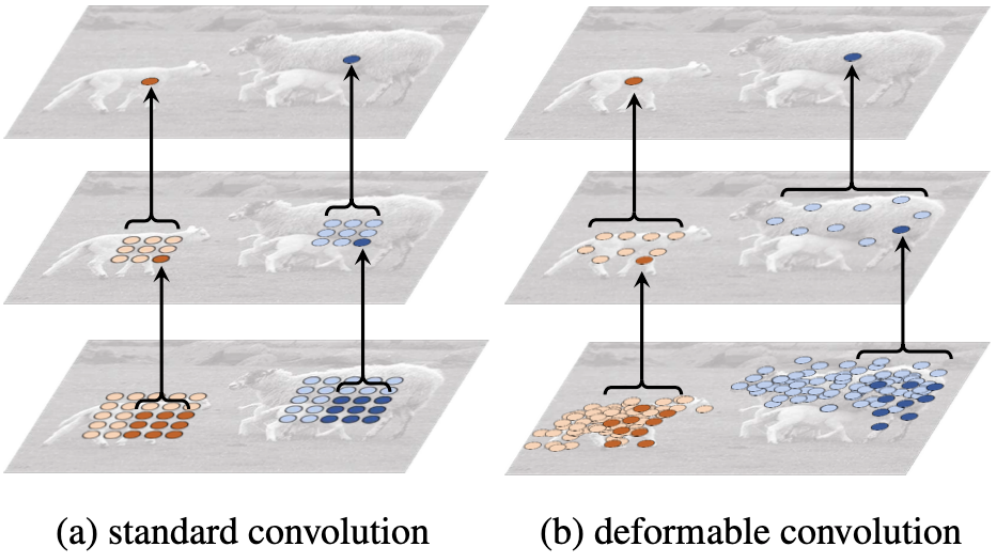


Figure 12: Mapping of 3 × 3 convolutional layers

Figure 13: Example of deformable convolution

deformable convolutional kernel can be dynamically adjusted according to the current image content to be recognized. The intuitive effect is that the position of the sampling points of the convolutional kernel at different locations will change adaptively according to the image content, thus adapting to the geometric deformation of different objects such as shape and size. Fig 12 shown the illustration of the fixed receptive field in standard convolution (a) and the adaptive receptive field in deformable convolution (b), using two layers. A point on the feature map of the back layer is mapped to the receptive field corresponding to the front layer, which is fixed and cannot take into account the different shapes and sizes of different targets. The deformable convolution, on the other hand, takes into account the deformation of the target, and the sampling points mapped to the previous layer will mostly cover the target, sampling more information which we are interested in.More examples are shown in Fig.13. Each image triplet shows the sampling locations ($93 = 729$ red points in each image) in three levels of 3×3 deformable filters (see Fig.12 as a reference) for three activation units (green points) on the background (left), a small object (middle), and a large object (right), respectively.

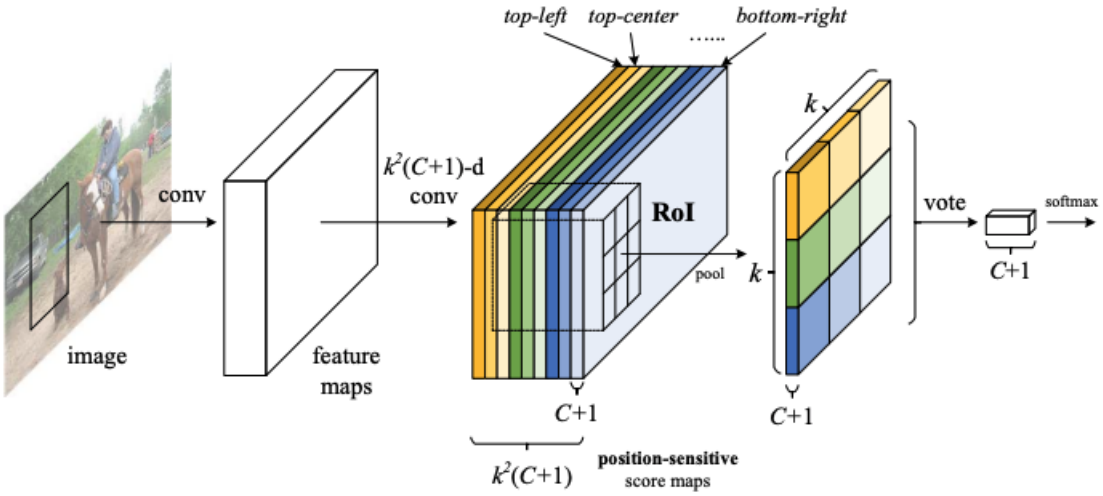


Figure 14: Key idea of R-FCN for object detection.

4 Variants of the major classical methods

4.1 Variants of one-stage methods(Faster R-CNN series)

4.1.1 R-FCN: Object Detection via Region-based Fully Convolutional Networks

The accuracy of Faster R-CNN [Ren et al., 2015] is actually very high, but the drawback is that it is not fast enough. Because it sends each proposed region into the region of interest (RoI) Pooling layer separately after the region proposal network, and there are several fully-connected layers behind it for classification and regression, this process slows down the speed. At the same time, object classification is required to have translation invariance, and locating objects is required to be able to respond to object translations, which creates a contradiction. In [Dai et al., 2016], address this dilemma between translation-invariance in image classification and translation-variance in object detection. The inspiration here is that if the computation of the fully connected layer behind is removed, the speed can be improved; if the higher-level feature maps with stronger semantic information can be made to have spatial information, better accuracy can be achieved. Based on the structure of the Faster R-CNN, in [Dai et al., 2016], position-sensitive score map and position-sensitive RoI pooling are proposed to solve the aforementioned problems.

For the key idea of R-FCN (Fig.9), if a RoI contains an object of category C, then the RoI will be divided into $K \times K$ regions, which represent each part of the object, for example, suppose the RoI contains a human object, $K = 3$, then "*human*" will be divided into 9 sub-regions, the top-center region should undoubtedly be the *head*, and the bottom-center should be the *feet*. The RoI is divided into $K \times K$ regions because it is expected that each region of the RoI should contain the parts of the object of the category C. That is, if it is a human, then the top-center region of the RoI must contain the head of the human. And after all these sub-regions contain the respective corresponding parts of that object, then the classifier will judge that RoI as that category.

So now the question is "How can the network determine that all the sub-regions of a RoI contain the

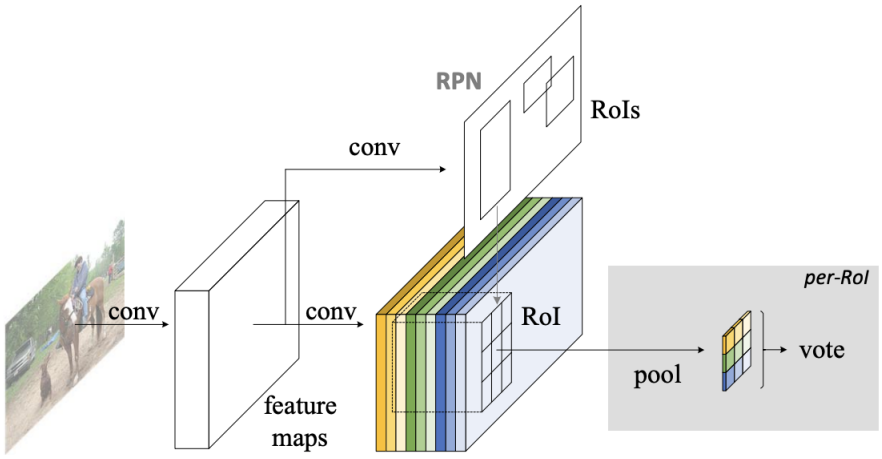


Figure 15: Overall architecture of R-FCN .

corresponding parts? This is the core idea of position-sensitive score map. R-FCN will add another convolutional layer at the end of the shared convolutional layer, and the convolutional layer is the "position-sensitive score map", what is the meaning of the score map? First of all, it is a convolutional layer, its height and width are the same as the shared convolutional layer, but its channels is $K^2\,(C+1)$ as shown in the Fig.9, then C denotes the number of object categories plus 1 background category, each of the categories has K^2 score maps. Assuming the category is human, then there are K^2 score maps, each score map indicates which locations in the original image contain a part of the human, and the score map will have *high response values* at locations containing a part of the human corresponding to that score map. Each score map is used to describe where one of the parts of the human body appears in the score map, and where it appears, it has a high response value.

Instead of a fully connected layer after position-sensitive RoI Pooling, R-FCN uses an average pooling to generate the vectors for classification. In this way, both spatial information is added and fully connected operations are reduced, thus R-FCN is slightly more accurate than Faster R-CNN and 2.5 times faster than Faster R-CNN [Dai et al., 2016].

4.2 Variants of one-stage methods(YOLO series)

The YOLO series has attracted a lot of attention as an enduring target detection approach. YOLO, as a one-stage approach, uses only a CNN network to directly predict the class and location of different objectives. In the following, we will focus on the most up-to-date YOLO series approach-**YOLOX** which published in August 2021.

4.2.1 YOLOX

Before we discuss the YOLOX [Ge et al., 2021] methodology, let's visualize its speed and detection effect in Fig.16:

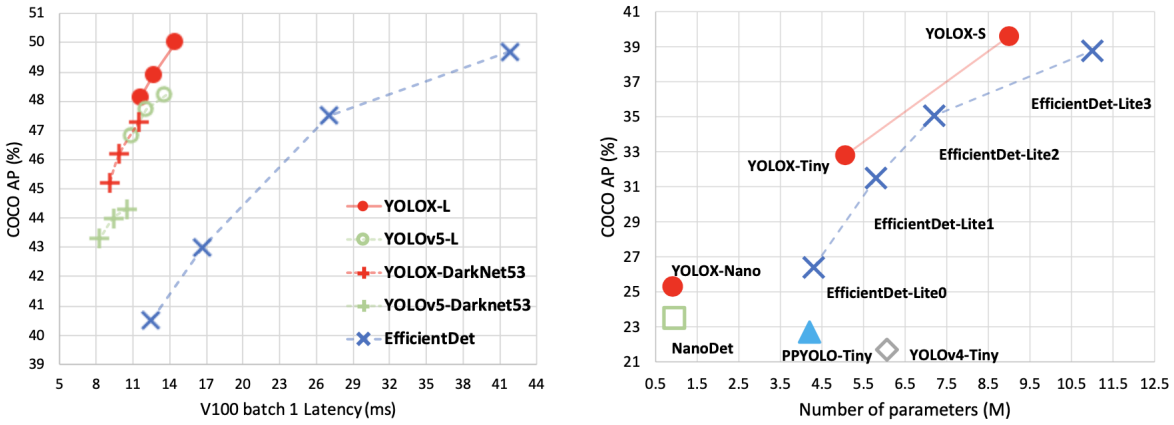


Figure 16: Comparison between YOLOX and other state-of-the-art object detectors.

- For YOLOX-L with roughly the same amount of parameters as YOLOv4-CSP [Wang et al., 2021], YOLOv5-L [Jocher et al., 2020], YOLOX-L achieve 50.0% AP on COCO at a speed of 68.9 FPS on Tesla V100, exceeding YOLOv5-L by 1.8% AP (average perception).
- YOLOX-Tiny and YOLOX-Nano (only 0.91M Parameters and 1.08G FLOPs) outperform the corresponding counterparts YOLOv4-Tiny [Bochkovskiy et al., 2020] and NanoDet3 [1] (fast and lightweight anchor-free object detection model) by 10% AP and 1.8% AP, respectively.

Notice that the authors use YOLOv3 [Redmon and Farhadi, 2018] as the baseline for improvement instead of YOLOv4 and YOLOv5 due to that the above tow approaches may be a little over-optimized for the anchor-based pipeline. At the same time, indeed, YOLOv3 is still one of the most widely used detectors in the industry due to the limited computation resources and the insufficient software support in various practical applications [Ge et al., 2021]. YOLOX contains the following major improvements:

1. **Decoupled head**

   Before we talk about the improvement about the decoupled head, let's talk about what is the coupled/decoupled head. Generally, in order to detect the location and category of the objects from the images, we will first extract some necessary features from the images, and then use these features to achieve localization and classification.In the case of deep learning, the network responsible for extracting features from images is called *backbone*. Of course, there is no way to know what kind of features are proposed here, after all, the "black box" feature of deep learning has not been able to really lift its veil so far. Although the two tasks of object detection and image classification have similarities, but they are not exactly equivalent. The goal of object detection is to achieve localization and classification of objects, while image classification only classifies objects in an image without locating them. To compensate for the inherent defect of the classification network's inability to localize, we need to add additional networks. In this way, we first migrate the classification network to be used as a feature extractor, and the subsequent

[1] https://github.com/RangiLyu/nanodet

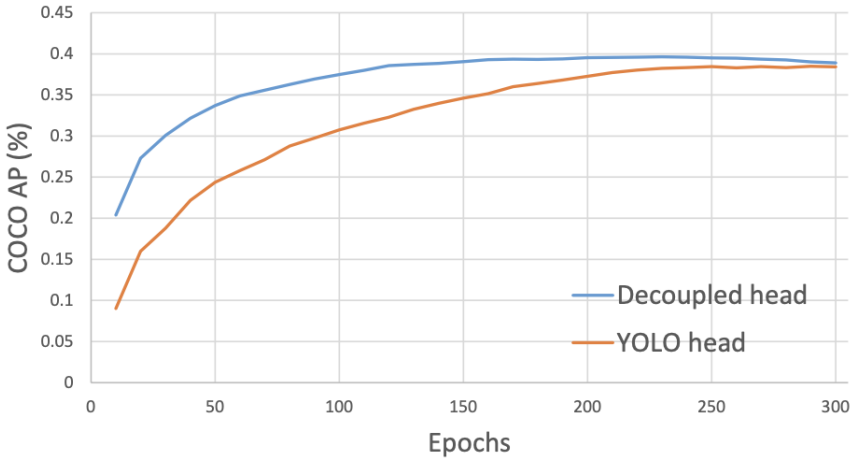


Figure 17: Training curves for detectors with YOLOv3 head or decoupled head.

network is responsible for detecting the location and class of the objects from these features. Then, it makes sense to call the part of the classification network "*Backbone*". The subsequent connected network layer is called "*Detection head*" because it mainly serves the detection task.

With the development of technology, in addition to the two parts of backbone and detection head, more novel technologies and modules have been proposed, the most famous, none other than the FPN (Feature Pyramid Networks) which we talked about before. The FPN structure is to extract information at different scales (actually, different size feature maps) and fuse them to make full use of all the feature information extracted by backbone, so that the network can better detect objects. With FPN, the information extracted by backbone can be utilized more adequately, enabling detector to cope well with multi-scale situations: images with objectives of different sizes, large, medium and small. The FPN structure can be inserted between the backbone and the detection head. Due to the delicate position of its insertion, it is called "*neck*". The role of the neck is to better fuse the features given by the backbone, which is then handed over to the subsequent detection head for detection, thus improving the performance of the network. Thus, a complete object detection network now consists of three main components:

$$Detector = Backbone + Neck + Head \tag{5}$$

The authors experimentally found that coupling the detection head may impair performance. There are two improvements:

- Replacing YOLO's head with a decoupled one greatly improves the converging speed as shown in Fig. 17.
- Compared to the non-decoupled end-to-end approach, decoupling delivers a 4.2% AP improvement.

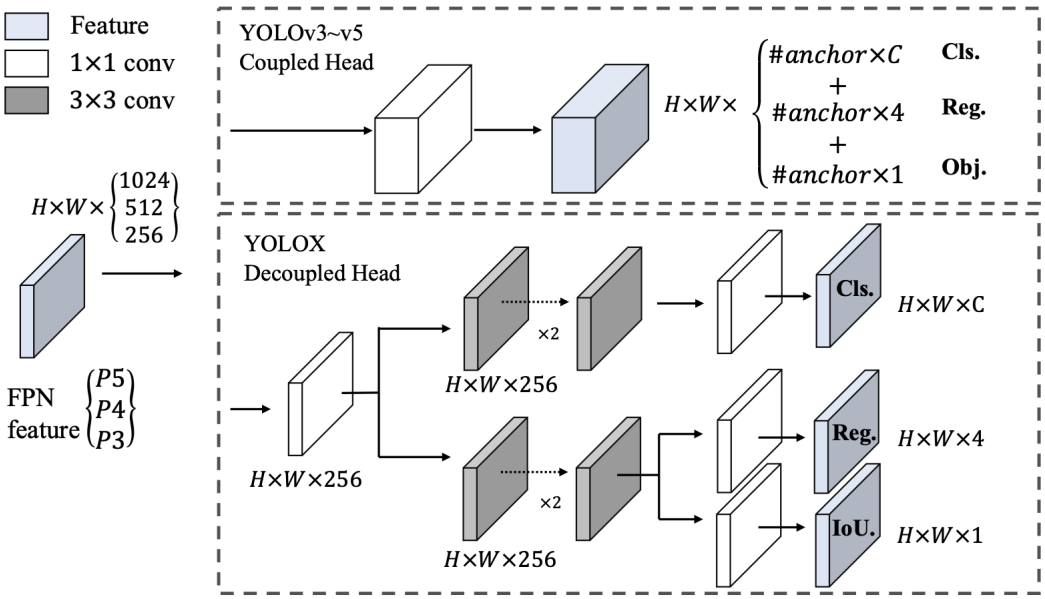


Figure 18: Illustration of the difference between YOLOv3 head and the proposed decoupled head in [Ge et al., 2021]

As shown in Fig. 18, in YOLOv3, for the COCO 80 categories detection task, each anchor will correspondingly produce prediction results in $H \times W \times 85$ dimensions, where **cls.** (predicting which of the 80 categories is the class) occupies 80 channel, **reg.** (coordinates) occupies 4 channels, and **obj.** (distinguishing foreground or background) occupies 1 channels. For YOLOX, for each level of FPN feature, it first adopt a 1×1 conv layer to reduce the feature channel to 256 and then add two parallel branches with two 3×3 conv layers each for classification and regression tasks respectively. IoU (intersection over unit) branch is added on the regression branch.

2. **Strong data augmentation**

   Mosaic and MixUp are added into augmentation strategies to boost YOLOX's performance. Mosaic is an efficient augmentation strategy proposed by ultralytics-YOLOv3 [2] which widely used in YOLOv4 [Bochkovskiy et al., 2020], YOLOv5 [Jocher et al., 2020]. As shown in Fig. 19, the mosaic augmentation uses four images, and stitching four images, each image has its corresponding box, after stitching four images to obtain a new image, the new image also obtain the corresponding box of this image, then we will pass such a new image into the neural network to learn, equivalent to passing four images at once to learn, which greatly enriches the background of detecting objects. MixUp [Zhang et al., 2018] is originally designed for image classification task but then modified for object detection training. The way MixUp is used in object detection is that the two images are blended with a certain ratio of RGB values, and the model is required to predict all the objects in the original two images

3. **Anchor-free** Anchor in computer vision has the meaning of anchor point or anchor box, the

[2] https://github.com/ultralytics/yolov3

Figure 19: Mosaic argumentation

anchor box often appears in object detection which indicating a fixed reference box. The most state-of-the-art object detection methods almost always use the anchor technique. First, a set of fixed reference boxes (named as region proposal network in Faster R-CNN) of different scales and positions is preset, covering almost all positions and scales, and each reference box is responsible for detecting targets whose intersection over unit is greater than a threshold (training preset, often 0.5 or 0.7). YOLOv4 and YOLOv5 both adopt the original anchor setting of YOLOv3. However, the anchor mechanism has several drawback:

- To achieve optimal detection performance, one needs to conduct clustering analysis to determine a set of optimal anchors before training. Those clustered anchors are domain-specific and less generalized.
- The anchor mechanism increases the complexity of the detection head and increases the number of predictions per image. For specific,for coco dataset, if yolov3 input is 416 ×416, it will eventually generate 13×13,26×26,52×52 three feature maps, and each feature map corresponds to 3 anchors, so there will be (13×13+26×26+52×52)×3=10647 anchors. Anchor-free mechanism significantly reduces the number of design parameters which need heuristic tuning and many tricks involved (e.g., Anchor Clustering [Redmon and Farhadi, 2017]).Switching YOLO to anchor-free by reduce from predicting 3 groups of anchors for one feature map to predicting only 1 group and make them directly predict four values, i.e., two offsets in terms of the left-top corner of the grid, and the height and width of the predicted box. Such modification reduces the parameters and GFLOPs of the detector and makes it faster, but obtains better performance – 42.9% AP compare with the baseline YOLOv3 which claim in the paper.

To sum up, YOLOX achieves a better trade-off between speed and accuracy than other counterparts across all model sizes. In the meantime, I have conducted some experiments based on YOLOX and plan to report next week.

5 Transformer

Transformer is the model used in [Vaswani et al., 2017], after years of industrial use and paper validation, has occupied an important position in the field of deep learning. In the rest of this chapter, first we will introduce the basic architecture of Transformer, then we will discuss some of its variants specifically for images and video, and finally we will discuss why we should embrace Transformer.

5.1 Basic Transformer architecture

5.1.1 General introduction for Transformer

Transformer [Vaswani et al., 2017], is the first model that uses only attention to do sequence transduction (sequence transduction: sequence-to-sequence generation. input a sequence, output a sequence. e.g. for machine translation, input a Chinese sentence, output an English sentence), replacing the previous structure of encoder-decoder with a multi-headed self-attention.

Through the experiments, the Transformer can be trained faster than other architectures based on recurrent or convolutional layers for translation tasks.

Most notably, the attention-based model can be applied not only to translation tasks, but also to image, audio, and video tasks to handle large inputs and outputs, and make generation less sequential.

5.1.2 Drawbacks of the previous method

Back in 2017, the most commonly used approaches in sequence modeling at that time was Recurrent Neural Network, Long-Short Term Memory [Hochreiter and Schmidhuber, 1997] and Gated Recurrent [Chung et al., 2014] neural networks(GRU) is a type of Recurrent Neural Network (RNN). Like LSTM (Long-Short Term Memory), it is also proposed to solve the problems of long-term memory and gradient in back-propagation.)There are two more dominant models in there: one is the recurrent language models, and the other is when you have more structured information output, encoder-decoder architectures will commonly be used. [Wu et al., 2016].

What are the features or drawbacks of RNN? In an RNN, if you are given a sequence, it is computed by taking the sequence from left to right step by step. Assuming your sequence is a sentence, the RNN will look at it word by word. As shown in Fig.20, For the t-th word, the RNN computes an h_t, which is the so called hidden state. And h_t depends on previous hidden state h_{t-1} and the input for position t (the t-th word itself). In this way, the RNN can take the previously learned historical information by h_{t-1} put into the present moment, and then do some computation with the current word t and get the output. This is the key to how RNNs can effectively handle sequential information. But that's where the problem comes in.

1. Difficult to parallelize

   When counting the t-th word, it is necessary to ensure that the h_{t-1} input of the previous word is completed. Assuming that the sentence has 100 words, it requires 100 steps in time,

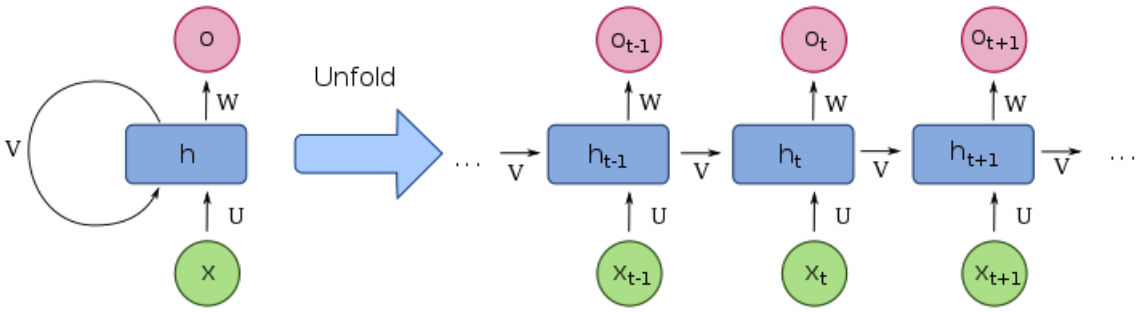


Figure 20: RNN architectures

making it impossible to parallelize in time. Inability to parallelize leads to poor computational performance.

2. Premature historical information may be dropped

   Since the RNN history information is passed backwards step by step, if you have a long time series, then those very early time series information may be dropped later on. If you don't want this information to be lost, then you need a larger h_t to store those information. But the problem with a larger h_t is that since h_t needs to be stored every time, it leads to a larger memory overhead.

Before that, there are also studies combining the attention mechanism with the recurrent network which allows the modeling of dependencies without regard to their distance in the input or output sequences [Bahdanau et al., 2014, Kim et al., 2017].

In [Vaswani et al., 2017], authors proposed a new model architecture which is the Transformer, that eschews a recurrence network and instead relies entirely on an attention mechanism to draw global dependencies between input and output. The Transformer allows for significantly more parallelization and can be trained to produce butter results in a relatively short period of time (8 P100 GPUs 12 hours).

5.1.3 Background

In order to reduce sequential computation, in some models, such as Extended Neural GPU [Łukasz Kaiser and Bengio, 2017], ByteNet [Kalchbrenner et al., 2016]and ConvS2S [Gehring et al., 2017], all of which use convolutional neural networks to replace the recurrent network. However, it is difficult to model longer sequences using convolutional neural networks, in another word, it is more difficult to learn dependencies between distant positions. This is due to the fact that when convolution does the calculation, it only looks at a relatively small window (e.g., receptive field, a 3x3 block of pixels) at a time, and if two pixels are far apart, it requires many layers of convolution, one layer at a time, to fuse the far apart pixels. But if you use the attention mechanism in Transformer, it is possible to see

all the pixels at a time, so you can see the whole sequence in just one layer.

However, one advantage of the existence of convolutional neural networks is that they can do multiple output channels, and one output channel can be considered as used to recognize different patterns. In order to make Transformer can also do multiple output channels, the authors proposed Multi-Head Attention to simulate the effect of a convolutional neural network with multiple output channels.

Although the attention mechanism is not the first time it has been proposed in Transformer, the Transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequence-aligned RNNs or convolution.

5.1.4 Model Architecture

Most competitive neural sequence transduction models have an encoder-decoder structure [Cho et al., 2014, Bahdanau et al., 2014].

The encoder maps $(x_1, \cdots, x_n)$ (original input) into $z = (z_1, \cdots, z_n)$ (vectors that can be understood by machine learning).For example, a sentence has n words, x_t is the t-th word, and z_t is the vector representation of the t-th word.

For decoder, it will get the output of the encode which is z, then decoder will generate a sequence $(y_1, \cdots, y_m)$ with length m. It is worth noting that n and m are not necessarily the same length. For example, the length may be different if translated from Chinese to English.

The difference between encoder and decoder is that encoder can look at the whole sentence at once, while decoder's output words are generated one by one, which is auto-regressive model, that is, the input to the model is also the model output. What is the meaning of 'input to the model is also the model output'? Given a $z = (z_1, \cdots, z_n)$, we can generate y_1 and after getting y_1, y_2 can be generated after getting y_1. When generating y_t, you have to get all the previous y_1 to y_{t-1}. When translating, the word will pop out one by one. So it means that your output in the past moment will also be used as your input in the current moment, so it is called auto-regressive.

Simply put, Transformer uses an encoder-decoder architecture tacked self-attention and point-wise, fully connected layers, shown in the left and right halves of Fig.21, respectively.

As shown in the Fig.21, if we use Chinese to English translation as an example,the *inputs* shown in the figure should be your Chinese sentence. For the input of the decoder,in fact the decoder has no input when making prediction, in fact it is some output of the decoder at the previous moment as input in this place, so it is written as *Output* in the figure.

The input goes through an Embedding layer(denoted by *Input Embedding* in the figure), i.e., a word comes in and is represented as a vector. The obtained vector values are summed with *Positional Encoding* which will talk in coming article.

As we can see the architecture of encoder in the Fig.22, where the $N\times$ stands for N such results are stacked together to form an encoder. Each encoder has two sub-layers, the first is a multi-head self-attention mechanism, and the second is a simple, position- wise, fully connected feed-forward network

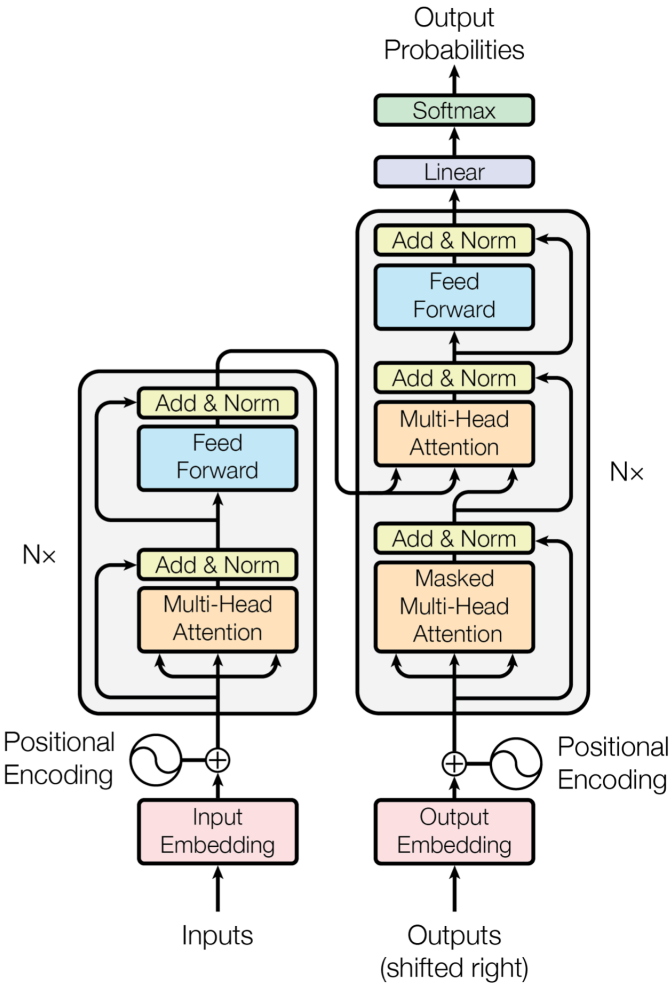


Figure 21: The Transformer - model architecture.

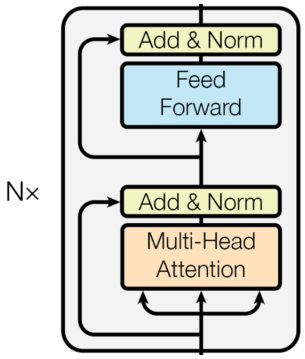


Figure 22: The Encoder architecture.

that can be considered a Multi-layer Perceptron(MLP). Also, a residual connection [He et al., 2016] have employed around each of the two sub-layers, followed by LayerNormalization.

After this, the output of the encoder goes to the decoder as the input of the decoder. The encoder and decoder have two identical sub-layers, while the decoder has an extra sub-layer containing Masked Multi-Head Attention.

The output of the decoder goes into a Linear layer, does a soft-max, and gets the final output. Adding Linear and soft-max to the end of the model is a standard neural network approach.

The above is the entire structure of the Transformer architecture, and we will then describe how each module is specifically implemented.

- **Encoder and Decoder Stacks**

  Encoder

  The encoder is built from two sub-layers, a multi-head self-attention mechanism, and a multi-layer perceptron (MLP) called position-wise fully connected feed-forward network.We can use the following formula to represent each sub-layer:

$$\text{LayerNorm } (x + \text{ Sublayer } (x)) \tag{6}$$

  where x is the input, and input will go into one of the sub-layer(multi-head self-attention or MLP), due to residual connection(shortcut connection in [He et al., 2016]) is adopted, so that we add the input and output together then we will get $(x + \text{ Sublayer } (x))$, and this value will use as the input to the *LayerNorm* and get the final output.

  Facilitate these residual connections that residual connections require consistent input and output dimensions, inconsistency requires projection. For simplicity, fix the output dimension of each layer $d_{model} = 512$. This means that each word it is 512 dimensions, regardless of which layer. Unlike CNN, which decreases the spatial dimension downwards and increases the channel dimension, Transformer uses a fixed length to represent the dimension of each layer, making the model relatively simple where only need consider one hyper-parameter.

  The LayerNorm mentioned in the previous section will be explained next. We will explain what LayerNorm is by comparing it with BatchNorm and why we do not choose batch norm.

  Consider the case of a two-dimensional input, where each row is a sample X and each column is a feature, as shown in Fig.23. BatchNorm: each time a column (1 feature, blue box in the figure) is placed in a mini-batch, let the mean becomes 0 and the variance becomes 1. BatchNorm also learns λ,β, by which BatchNorm can learn to deflate a vector into a vector with arbitrary mean and arbitrary variance.

  LayerNorm and BatchNorm are almost the same in many cases, except for the implementation method, which is a bit different. As shown in Fig.24, LayerNorm does Normalization for each sample (yellow box, turning each row to have mean 0 and variance 1), instead of doing normaliza-

tion for each feature. But for the Transformer or RNN, the common input is three-dimensional.

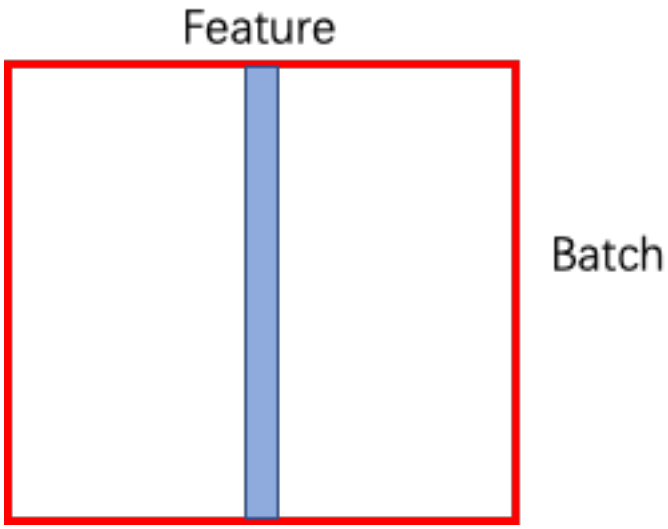


Figure 23: Two-dimensional BatchNorm .

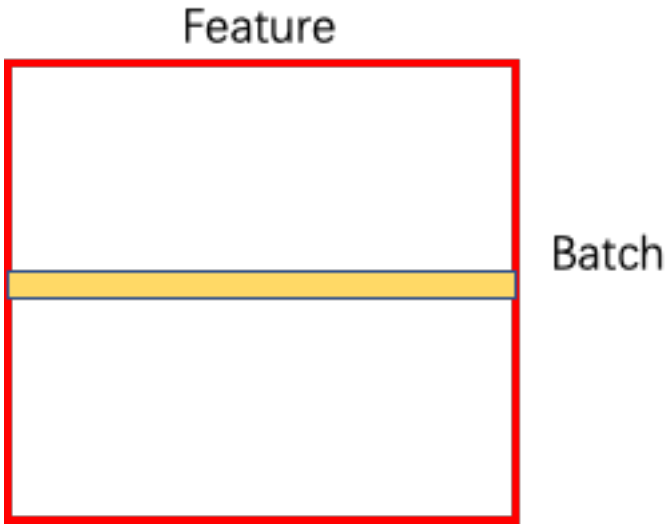


Figure 24: Two-dimensional LayerNorm .

The input is a sequence of samples with many elements in each sample. If there are n words in a sentence, each word corresponds to a vector, and there is a batch, then it is a three-dimensional representation, as shown in Fig.25. The column is the sequence length n (length of sentence); the 3rd dimensional-feature is an additional vector for each word, where $d = 512$ in the transformer.

BatchNorm in 3D takes one feature at a time, cuts a piece (blue line), and pulls it into a vector, normalized by a mean of 0 and a variance of 1. For LayerNorm, the above operation is performed for each sample (yellow line).

Why is LayerNorm used more often? Because the length of the samples in the time-series data may be different, different cuts will give different results. However, different samples may have

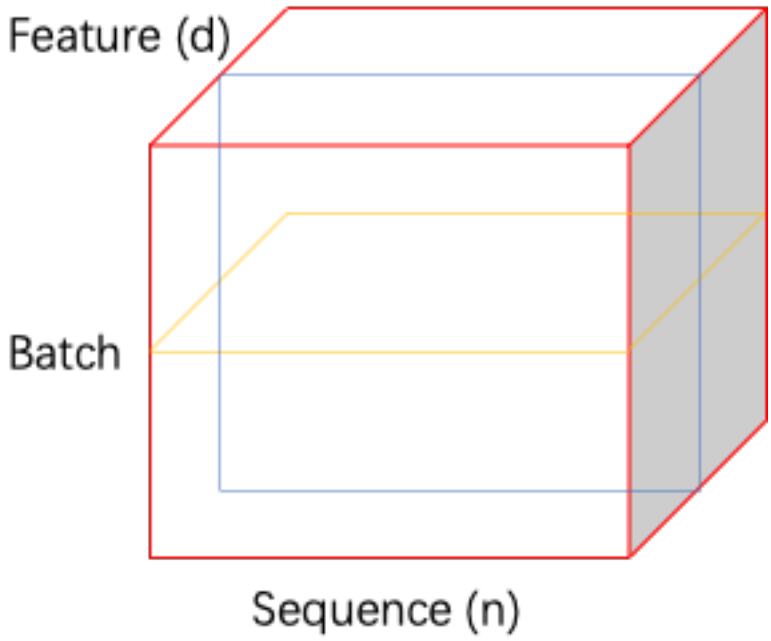


Figure 25: Three-dimensional LayerNorm (yellow) and BatchNorm (blue).

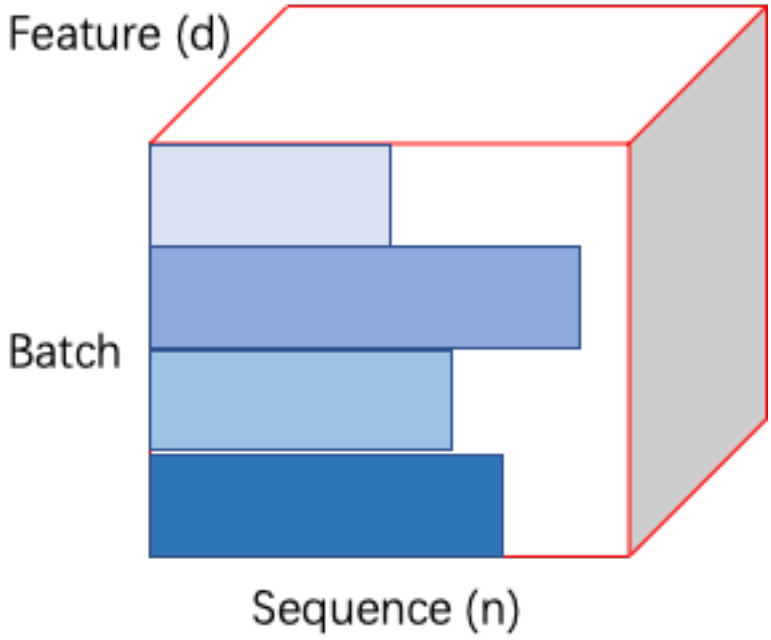
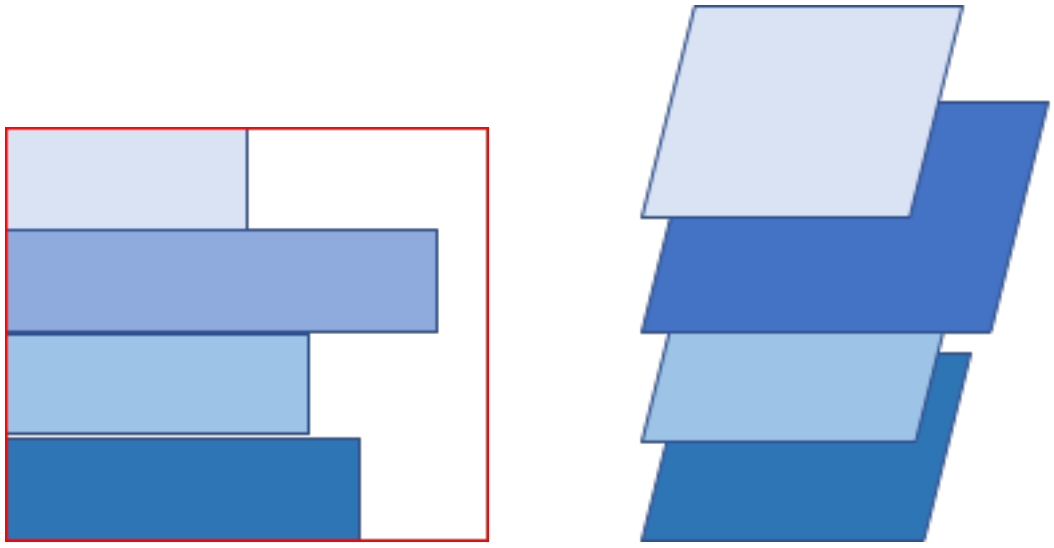


Figure 26: Different lengths for the samples.

different lengths, as the figure shows in Fig.26, 4 samples have 4 different lengths.



(a) BatchNorm result (b) LaterNorm result

Figure 27: Cut out results

The results of BatchNorm are shown in Fig.27(a), where the blue part indicates the valid part and the rest of the space is filled with 0. For the mean and variance of the Mini-batch calculated during training, if the sample length varies greatly, the jitters of the mean and variance are large for each calculation of a Mini-batch. When predicting, the global mean and variance need to be calculated and stored, but if a particularly long-length brand-new sample is encountered during prediction that has not been seen during training, the mean and variance calculated during training may not work well.

However for the LayerNorm, in Fig.27 (b), LayerNorm calculates the mean and variance for each sample, with no need to store the global mean and variance. So LayerNorm is more stable, and the mean and variance are calculated within each sample, regardless of whether the length of the sample is long or short. The above is the reason why LayerNorm be adopted in Transformer.

Decoder

The decoder is composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Slimier with an encoder, the decoder also has residual connections and LayerNorm.As described earlier the decoder is auto-regressive. The input of the current moment is the output of some precious moments. When making predictions, the decoder should not be able to see the output at subsequent times, However, the attention mechanism can see the complete input at once, so it is necessary to avoid this situation above. When the decoder is trained to predict the output at time t, the decoder should not see those inputs after time t (i.e., $t+1, ..., t+n$). Transformer does this by using an attention mechanism with masked

attention to ensure consistent behavior during training and prediction.

- **Attention**

  After understanding the architecture of the encoder and decoder, we will analyze how the sub-layers of each are defined, first and most importantly-the attention layer.

  An attention function is a function that maps a *query* and some *key-value* pair into an output, where all the *query*, *key*, *value* and *outputs* are vectors.

  Specifically, the output is a weighted sum of values, which results in the dimensionality of the output is equal to the dimensionality of the value. The weight of each value is calculated from the similarity (compatibility function) between the query and the corresponding key. As illustrated in Fig.28, under the assumption that *Query 1* is most similar with *key 1*, *key 2* is the second most similar, and *key 3* is the least similar. So, we can see that *Query 1* multiply with largest weight (red solid line) with *Value 1*, ands multiply with second largest weight (red dot line) with *Value 2*; for *Value 3*, it multiply *Query 1* with smallest weight (red underline-dot).

  Although the key-value pair does not change, the output will be different as the query changes because the weights are assigned differently, which is the attention mechanism.

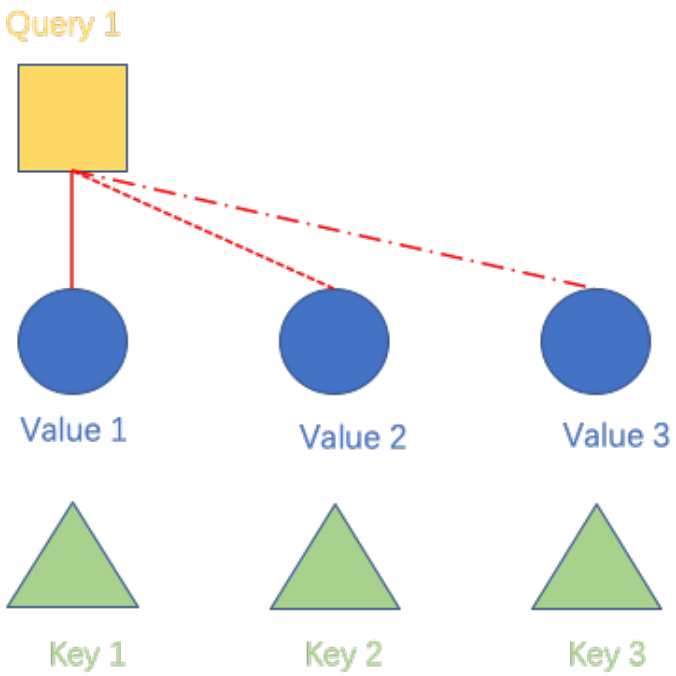


Figure 28: Calculation for weight.

- **Scaled Dot-Product Attention**

  Different compatibility functions result in different attention versions, so this section describes the attention versions used in the Transformer. The length of the query and key is equal to d_k. The dimension of value is d_k, so the output is also d_k. The specific calculation of attention is

to do the dot product for each query and key, and then use it as the similarity (compatibility functions), wherein the Euclidean distance:

$$\vec{Q} \cdot \vec{K} = |\vec{Q}||\vec{k}| \cos\theta \tag{7}$$

$$\cos\theta = \frac{\mathbf{Q} \cdot \mathbf{K}}{|\vec{Q}||\vec{K}|} \tag{8}$$

Through the equations that we can know the larger the value of the dot product is, the larger its cosine value is, and the more similar these two vectors are. After the similarity is calculated, divide each by $\sqrt{d_k}$, and apply a soft-max function to obtain the weights on the values. Suppose there is a query and n key-value pairs, this query will be a dot product with each key-value pair, which will generate n similarity values. Pass in soft-max to get n non-negative weights that sum to 1. Multiply the soft-max weights with the value matrix V to get the attention output. In practice, we do not compute one query at a time, because the computation is slow. Instead, we write multiple queries into a matrix and parallelize the operations. As shown in Fig.29,Q stands for query matrix with n queries, and the dimension is d_k, since the number of queries and key may not be the same, so for key matrix, the length is m with the same dimension as a query which is d_k; then we can get a $n \times m$ matrix, where the red line in the figure indicates the dot product of a query for all keys.

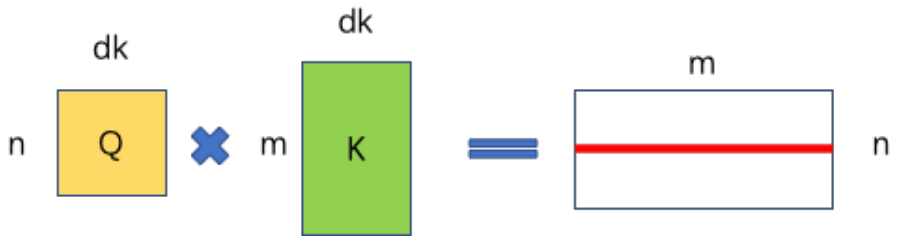


Figure 29: Matrix calculations for query and key.

After getting the $n \times m$ matrix, divided it by $\sqrt{d_k}$, and apply a soft-max function. Then, as shown in Fig.30, the processed matrix multiple with the $m \times d_v$ matrix V, finally, a $n \times d_v$ will be getting, and the purple line indicates the one attention value. So attention can be calculated with two times matrix multiplication, therefore, highly parallel operations are possible.

2 common attention mechanisms: additive attention mechanism [Bahdanau et al., 2014] (it can handle the case where your query and key are not equal in length; dot-product attention mechanism, Transformer uses scaled dot-product attention where it needs to be divided by $\sqrt{d_k}$, so you can see its name it's called scale. The dot product was chosen because of its simple and efficient implementation, requiring only two matrix multiplications. The reason is scale dot-product needs to be divided by $\sqrt{d_k}$ to prevent the gradient of the soft-max function from vanishing. When d_k is not very large, it is fine to divide or not. But for d_k is larger (when the

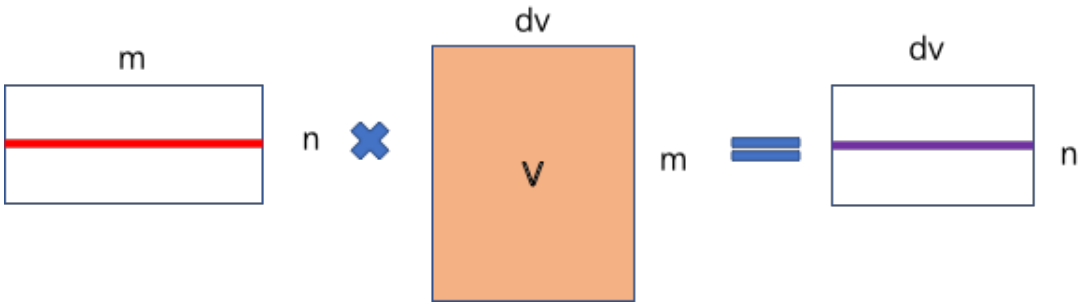


Figure 30: Matrix calculations to get Attention.

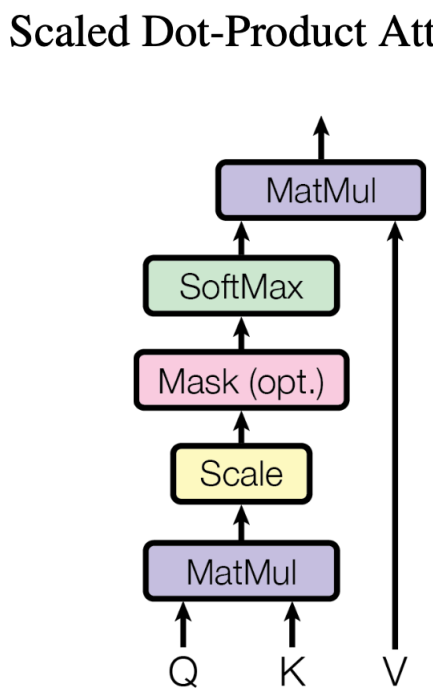


Figure 31: Scaled Dot-Product Attention.

length of the two vectors is longer, the column for matrix Q and matrix K), the value of the dot product will be larger or will be smaller. When you have a larger value, the relative gap will become larger, resulting in the maximum value through soft-max will closer to 1 and the remaining values being closer to 0. The values will be closer to the extremes and the gradient will be smaller when calculating the gradient. The d_k used in Trasformer is generally large (512 in this article), so dividing by $\sqrt{d_k}$ is a good choice.

- **How to do mask?**

  The intent of mask is to avoid seeing inputs after time t, at time t.Suppose our query and key are of equal length, both of length n, and can match in time. For the query Q_t at time t, only $K_1, ..., K_{t-1}$ should be seen. While in the attention mechanism, Q_t will see all the values in the K matrix to do the operation, the attention mechanism will always count until K_n. So, there the function of the mask is replacing the values of Q_t and K_t after time t with a large negative number. As shown in Fig.31, after the mask, it will pass to the soft-max, when such a huge negative number goes into soft-max for exponentiation, the result will become 0, that is, the weight will be 0; and multiplied with value V, only $V_1, ..., V_{t-1}$ have the value, rest of the time, the values are all 0. the mask makes my query at time t only look at the key-value pairs before time t during training so that the prediction can correspond to the times one by one.

- **Multi-Head Attention**

  Fig.32 depicted the Multi-Head Attention used in the transformer. The input is the original value, key, and query. Then it goes into a linear layer, which projects the value, key, and query to a lower dimension. Then make a scaled dot product attention (shown in Fig.31). The scaled dot product attention will be executed h times to get h outputs, then all h output vectors will be concatenated together, and finally, a linear projection Linear will be done to return to our multi-head attention.

  Why do we need a multi-headed attention mechanism? The reason is that inside the scale dot-product attention, there are no parameters that can be learned, and the specific function is the dot product. And there are times when a different way of computing similarity is desired in order to identify different patterns. In order to achieve this, the dot-product attention is first projected to the lower dimension, and the weight w of the projection is learnable. Multi-head attention gives h chances to learn different projection methods, so that it can match some similarity functions of different patterns in the projected metric space, and then put h heads together and finally do another projection to get back the original dimension. The equation of multi-headed attention can be described as:

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{Concat}\left(\text{head}_1, \ldots, \text{ head }_{\text{h}}\right) W^O \\ \text{where head}_{\text{i}} &= \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right) \end{aligned} \tag{9}$$

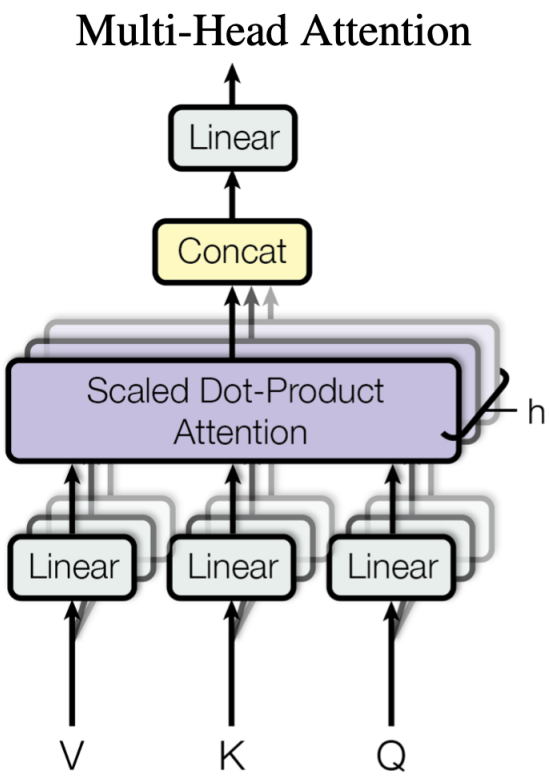


Figure 32: Multi-Head Attention consists of several attention layers running in parallel.

The input of multi-headed attention still are Q, K, V, but the output is concatenated with the output of the different headers and then projected into a W^O. Each $head_i$, is a projection of Q, K, V onto the low dimension through the learnable $\left(W_i^Q, W_i^K, W_i^V\right)$, and then through the attention function to get the $head_i$.

- **Applications of Attention in Transformer**

  As shown in Fig.21, there have three kinds of yellow boxes that indicate three kinds of attention layers. For the attention in the encoder, Suppose the sentence length is n and the input to the encoder is a vector of n vectors of length d, and each input word corresponds to a vector of length d. The attention layer of the encoder has three inputs, which represent key, value, and query. It is copied into three: the same thing, both as key and as value and as query, so it is called a self-attentive mechanism. key, value, and query are actually one thing, which is itself.

  If n queries are input, and each query gets an output, then there are n outputs. The output is a value-weighted sum (the weight is the similarity between the query and the key), and the dimension of the output is d because the input dimension is equal to the output dimension. As shown in Fig.33, if the green line represents the weight, if we use the second yellow vector, which is calculate similarity with other input vectors, the similarity with itself is the largest and the weight (green line) is the thickest. Assuming that the rightmost vector is more similar to itself, the weight will be larger and the green line will be thicker. Ignoring the multi-head and

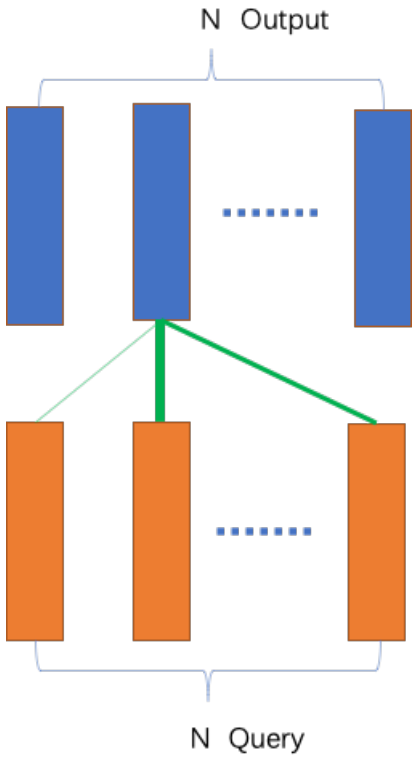


Figure 33: Attention in the encoder.

projection cases, the output is a weighted sum of the inputs, whose weights are derived from the similarity of each vector to the other vectors. If the multi-head and projection cases are considered, attention learns h different distance spaces, making the output vary.

For the masked multi-head attention in the decoder, the only difference with attention in the encoder is that multi-head attention, which does not see inputs after time t, has a weight of 0 (denoted by the green line in fig.33).

Another multi-head attention layer in the decoder is no longer self-attention, its key-value comes from the output of the encoder, and the query is the output from the masked multi-headed attention in the decoder. As illustrated in Fig.34, the red squares show the output value and key of the encoder (the output of the last layer of the encoder is n vectors of length d); The green square indicates the output of masked multi-head attention + Add & Norm of the decoder, which is a vector of m vectors of length d; The blue squares indicate the output of the decoder based on the query: weighted sum of value (the weight depends on the similarity of the red and green squares). The role of the third attention layer is to efficiently extract the encoder layer output based on the query, that is, to bring up useful information from the encoder output based on what I pay attention to(query).

- **Position-wise Feed-Forward Networks**

  In this section, the feed-forward network will be talked about as the blue blocks shown in Fig.22.

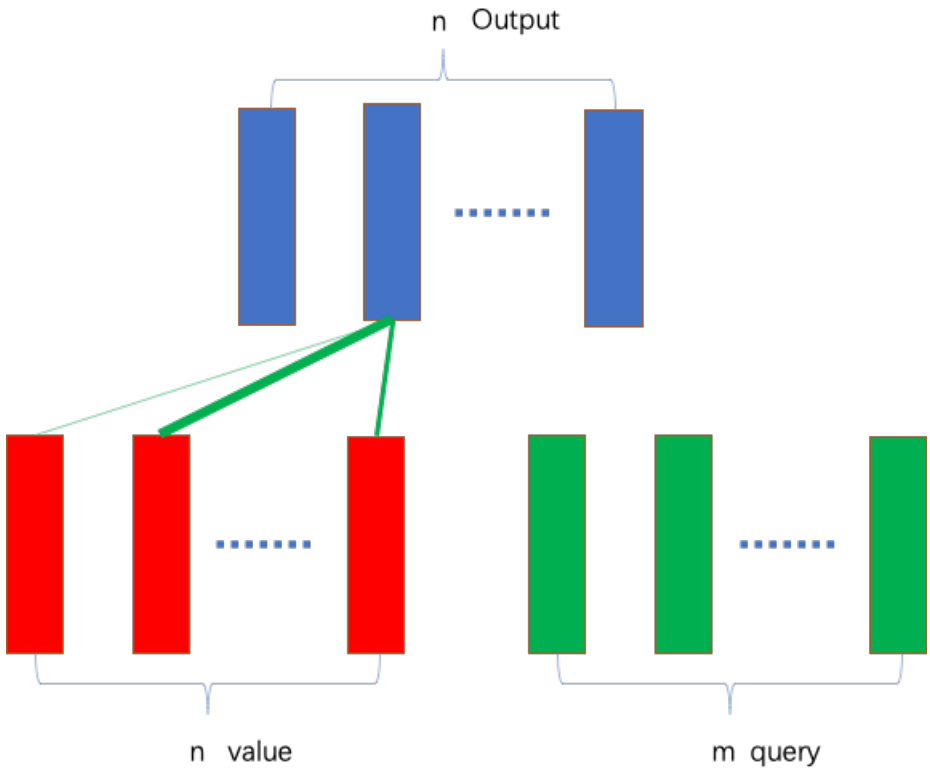


Figure 34: Attention in the decoder.

A fully connected feed-forward network can be considered as a Multi-layer perceptron(MLP), which is applied to each position separately and identically. Position means that the input sequence has many words, and each word is a position. Position-wise means that an MLP acts on each word (position) once, and the same MLP acts on each word. The formula of the position-wise feed-forward networks can be written as:

$$\mathrm{FFN}(x) = \max\left(0, xW_1 + b_1\right) W_2 + b_2 \tag{10}$$

where $W_1 + b_1$ is a linear layer, and $\max\left(0, xW_1 + b_1\right)$ is a ReLU activation layer and $W_2 + b_2$ is second linear layer.

Duo to that, the input $d_{model} = 512$, so that x is a vector of length 512. Since the FFN can be viewed as a single hidden layer MLP, the middle W_1 projects the input x expanded dimensionally to 2048, and finally W_2 projects back to 512-dimensional size to facilitate residual connectivity (input dimension equal to output dimension).

For better understanding, we represent FFN and attention together in a diagram, as well as explain what distinguishes it from the previous RNN (recurrent neural network). For simplicity, we consider the following consumption no residual connection, no LayerNorm, single-headed attention, and no projection. See the difference with RNN.

As Fig.35 shows that attention does a weighted sum of the inputs and the weighted sum goes

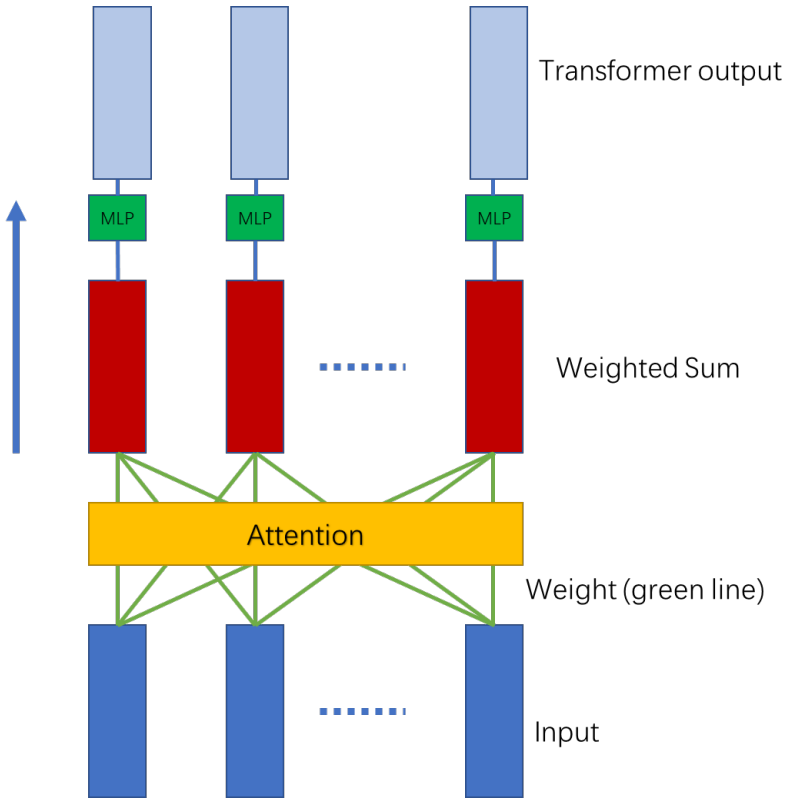


Figure 35: Transformer schematic diagram.

into a point-wise MLP. (Multiple green square MLPs are drawn, which is an MLP with the same weight). Position-wise MLP computes each input point (position)to get the output. The attention function is to grab the information in the whole sequence and do an aggregation. The green block in the diagram already has what is of interest in the sequence. So that when I do the projection or do the MLP mapping to the semantic space, due to the weighted sum (red block) already containing my sequence information, so each MLP just has to do it independently for each position. Since the sequence information has been aggregated by the time the attention layer is passed, the MLP can be done separately.

As a comparison, let's see how RNN does it. The green block represents the linear layer (MLP). In RNN, for the first position, the input goes through the MLP and get the output directly. But for the position after the first position, how are these locations using sequence information? the MLP has the same weight as the first position's MLP, as the red line shown in Fig.36, the RNN puts the output of the previous position back to the next position and merges it into the MLP together with the input of the next position, which is the transfer of information in RNN.

The similarity between RNN and Transformer is that both use a linear layer or an MLP to do the transformation of the semantic space. The difference between RNN and transformer lies in how to pass the information of the sequence. RNN is to pass the information output from the previous time to the next time as input. TTransformer uses an attention layer to globally get the information inside the whole sequence and then uses MLP to do the semantic transformation.

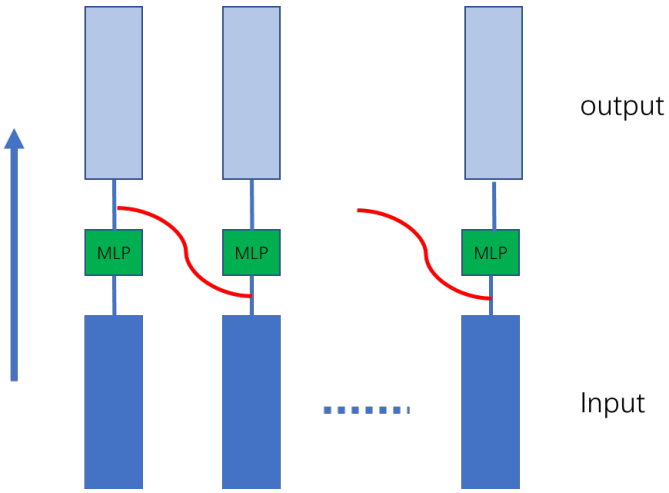


Figure 36: Recurrent Neural Network schematic diagram.

- **Embedding Layer**

  The role of the embedding layer is to map the input word into a vector. Embedding means that for any word, a vector of length d is learned to represent the whole word, where in the paper $d_{model} = 512$.

- **Positional Encoding Layer**

  Why do we need positional encoding? Because attention does not have temporal information. The output is a weighted sum of values (the weight is the similarity between the query and the key, and has nothing to do with sequential information), Transformer doesn't even look at where the key-value pairs are in the sequence. After the sequence is arbitrarily disordered in one sentence, the result is the same after the attention process. The order is changed, but the value is not, and that is the problem. When dealing with temporal data, if the words in a sentence are completely disrupted, then the semantics will definitely change, but attention will not handle this situation, so we need to add temporal information. For RNN, the output of the previous moment is used as the input of the next moment to pass the temporal information. And for attention, it is to add temporal information to the input, that is, positional encoding which is a word in position i will add the i position information to the input.

  So how do you represent position information? For example, if a computer represents a 32-bit integer, then it uses 32 bits, and each bit has a different value to represent. That is, a number is represented by a vector of length 32. A word is represented in the embedding layer as a 512-dimensional vector, while another 512-dimensional vector is used to represent a number, i.e., position information. Specifically, the position encoding is calculated by the following equations:

$$\begin{aligned} PE_{(pos,2i)} &= \sin\left(pos/10000^{2i/d_{\text{model}}}\right) \\ PE_{(pos,2i+1)} &= \cos\left(pos/10000^{2i/d_{\text{model}}}\right) \end{aligned} \tag{11}$$

The sin and cos function with different periods are calculated to obtain a vector of length 512 to represent any value. This is a positional encoding of 512 in length, which records the temporal information, and the embedded layers are summed to complete the addition of the temporal information to the input. After adding input embedding, no matter how to disrupt the order of the input sequence, after entering the following layer, the output values are unchanged.

5.2 Why Self-Attention

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O\left(n^2 \cdot d\right)$	$O(1)$	$O(1)$
Recurrent	$O\left(n \cdot d^2\right)$	$O(n)$	$O(n)$
Convolutional	$O\left(k \cdot n \cdot d^2\right)$	$O(1)$	$O\left(\log_k(n)\right)$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

The above table shows the Maximum path lengths which indicate how far one information point has to travel to another information point; per-layer complexity; and the minimum number of sequential operations which means that the next calculation step has to wait for the completion of the previous calculation steps for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions, and r is the size of the neighborhood in restricted self-attention.

For the self-attention layer query matrix($n \times d$) is multiplied by the key matrix($m \times d$), due to self-attention, so that ($n = m$), so the complexity of self-attention is $O\left(n^2 \cdot d\right)$. Since self-attention is mainly matrix multiplication, the parallelism is high, so sequential operations are $O(1)$. In attention, a query does operations with all keys, and its output is a weighted sum of all values, so any query and key-value pairs can be passed just once which is $O(1)$.

The recurrent layer, if the sequence contains n words, it does the operations one by one, each inside its main calculation is an n times n matrix, a dense layer (full connected layer) and then multiplied by d input, so it is an d^2, and then to do n times, so its complexity is $O\left(n \cdot d^2\right)$. Since the computation of the current moment of the RNN needs to wait for the completion of the computation of the previous moments, the sequential operations are $O(n)$. Also, the information from the initial point to the last point needs to go through n steps to be conveyed, which means the maximum path length is $O(n)$. So RNN does not perform well for particularly long sequences.

For the convolutional layer, k is generally small (e.g., 3, 5) and constant, so CNN and RNN have similar complexity.

For restricted self-attention, the complexity of the query is reduced by restricting it to the nearest r neighbors, but the maximum path length is increased because it requires several steps of message passing for relatively distant points.

By comparison, when the sequence length, model width, and depth are the same, the algorithm complexity of the three models of self-attention, recurrent, and convolutional are similar. But self-attention and convolutional calculation are more easy and fast. Finally, self-attention is better at

blending information.

References

Bolei Zhou, Agata Lapedriza, Jianxiong Xiao, Antonio Torralba, and Aude Oliva. Learning deep features for scene recognition using places database. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'14, page 487–495, 2014.

Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

Ross B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015. URL http://arxiv.org/abs/1504.08083.

Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 91–99, Cambridge, MA, USA, 2015. MIT Press.

Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. Ssd: Single shot multibox detector. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, pages 21–37, Cham, 2016. Springer International Publishing.

Tao Kong, Anbang Yao, Yurong Chen, and Fuchun Sun. Hypernet: Towards accurate region proposal generation and joint object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 845–853, 2016.

Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2117–2125, 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Li, Guodong Zhang, Han Hu, and Yichen Wei. Deformable convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 764–773, 2017.

Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks. In *Advances in neural information processing systems*, pages 379–387, 2016.

Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. Yolox: Exceeding yolo series in 2021, 2021.

Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. Scaled-yolov4: Scaling cross stage partial network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13029–13038, 2021.

Glenn Jocher, Alex Stoken, Jirka Borovec, NanoCode012, ChristopherSTAN, Liu Changyu, Laughing, tkianai, Adam Hogan, lorenzomammana, yxNONG, AlexWang1900, Laurentiu Diaconu, Marc, wanghaoyang0106, ml5ah, Doug, Francisco Ingham, Frederik, Guilhen, Hatovix, Jake Poznanski, Jiacong Fang, Lijun Yu, changyu98, Mingyu Wang, Naman Gupta, Osama Akhtar, PetrDvoracek, and Prashant Rai. ultralytics/yolov5: v3.1 - Bug Fixes and Performance Improvements, October 2020. URL https://doi.org/10.5281/zenodo.4154370.

Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection, 2020.

Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

Hongyi Zhang, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization, 2018.

Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural

machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016. URL http://arxiv.org/abs/1609.08144.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Yoon Kim, Carl Denton, Luong Hoang, and Alexander M. Rush. Structured attention networks. *CoRR*, abs/1702.00887, 2017. URL http://arxiv.org/abs/1702.00887.

Łukasz Kaiser and Samy Bengio. Can active memory replace attention?, 2017.

Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aäron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural machine translation in linear time. *CoRR*, abs/1610.10099, 2016. URL http://arxiv.org/abs/1610.10099.

Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. Convolutional sequence to sequence learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1243–1252. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/gehring17a.html.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014. URL http://arxiv.org/abs/1406.1078.