General object detection framework

Object detection is a CV task that involves both main tasks: localizing one or more objects within an image and classifying each object in the image (see table 7.1). This is done by drawing a bounding box around the identified object with its predicted class. This means the system doesn't just predict the class of the image, as in image classification tasks; it also predicts the coordinates of the bounding box that fits the detected object. This is a challenging CV task because it requires both successful object localization, in order to locate and draw a bounding box around each object in an image, and object classification to predict the correct class of object that was localized.

Table 7.1 Image classification vs. object detection

Image classification	Object detection
The goal is to predict the type or class of an object in an image. ▪ Input: an image with a single object ▪ Output: a class label (cat, dog, etc.) ▪ Example output: class probability (for example, 84% cat)	The goal is to predict the location of objects in an image via bounding boxes and the classes of the located objects. ▪ Input: an image with one or more objects ▪ Output: one or more bounding boxes (defined by coordinates) and a class label for each bounding box ▪ Example output for an image with two objects: – box1 coordinates (x, y, w, h) and class probability – box2 coordinates and class probability Note that the image coordinates (x, y, w, h) are as follows: (x and y) are the coordinates of the bounding-box center point, and (w and h) are the width and height of the box.

Object detection is widely used in many fields. For example, in self-driving technology, we need to plan routes by identifying the locations of vehicles, pedestrians, roads, and obstacles in a captured video image. Robots often perform this type of task to detect targets of interest. And systems in the security field need to detect abnormal targets, such as intruders or bombs.

General object detection framework:

Typically, an object detection framework has four components:

1 ***Region proposal*****—**An algorithm or a DL model is used to generate regions of interest (RoIs) to be further processed by the system. These are regions that the network believes might contain an object; the output is a large number of bounding boxes, each of which has an *objectness score*. Boxes with large objectness scores are then passed along the network layers for further processing.

2 ***Feature extraction and network predictions*****—**Visual features are extracted for each of the bounding boxes. They are evaluated, and it is determined whether and which objects are present in the proposals based on visual features (for example, an object classification component).

3 ***Non-maximum suppression (NMS)*****—**In this step, the model has likely found multiple bounding boxes for the same object. NMS helps avoid repeated detection of the same instance by combining overlapping boxes into a single bounding box for each object.

4 ***Evaluation metrics*****—**Similar to accuracy, precision, and recall metrics in image classification tasks (see chapter 4), object detection systems have their own metrics to evaluate their detection performance. In this section, we will explain the most popular metrics, like mean average precision (mAP), precision-recall curve (PR curve), and intersection over union (IoU).

Now, let's dive one level deeper into each one:

1. ***Region proposals***

In this step, the system looks at the image and proposes RoIs for further analysis. RoIs are regions that the system believes have a high likelihood of containing an object, called the *objectness score* (figure 7.2). Regions with high objectness scores are passed to the next steps; regions with low scores are abandoned.

Objectness Score:

- **Definition:** The **objectness score** is a metric that represents the likelihood of a particular RoI containing any object (not necessarily a specific one). It is a confidence measure output by the system.
- **Range:** Typically, the score ranges from 0 to 1, where:
  - **1:** Indicates high confidence that the region contains an object.
  - **0:** Indicates low confidence that the region contains any object.

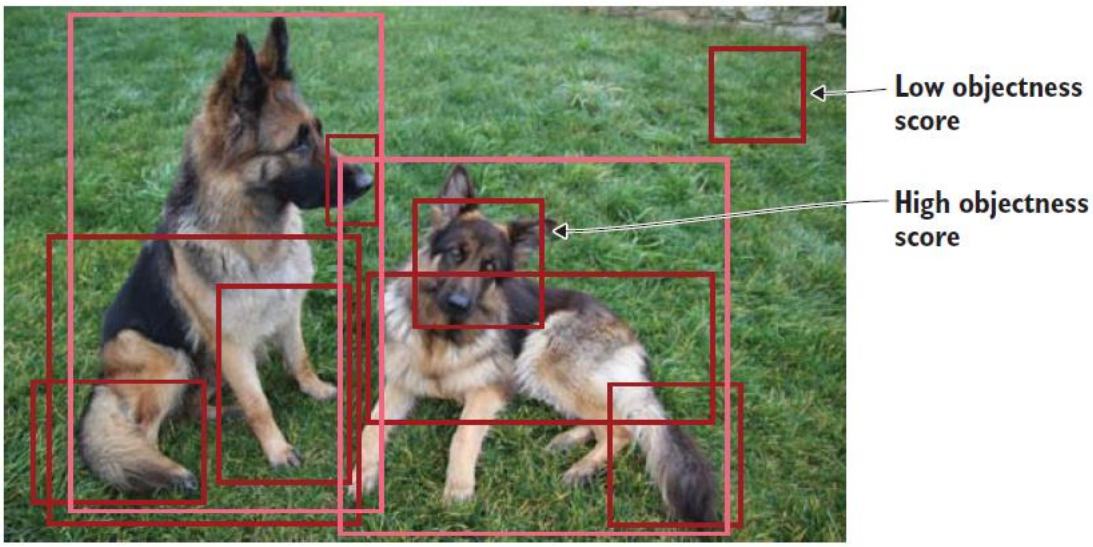


Figure 7.2 Regions of interest (RoIs) proposed by the system. Regions with high objectness score represent areas of high likelihood to contain objects (foreground), and the ones with low objectness score are ignored because they have a low likelihood of containing objects (background).

There are several approaches to generate region proposals. Originally, the *selective search* algorithm was used to generate object proposals; we will talk more about this algorithm when we discuss the R-CNN network.

2. ***Network predictions***

This component includes the pretrained CNN network that is used for feature extraction to extract features from the input image that are representative for the task at hand and to use these features to determine the class of the image. In object detection frameworks, people typically use pretrained image classification models to extract visual features, as these tend to generalize fairly well. For example, a model trained on the MS COCO or ImageNet dataset is able to extract fairly generic features.

In this step, the network analyzes all the regions that have been identified as having a high likelihood of containing an object and makes two predictions for each region:

- *Bounding-box prediction*—The coordinates that locate the box surrounding the object. The bounding box coordinates are represented as the tuple (x, y, w, h), where x and y are the

coordinates of the center point of the bounding box and *w* and *h* are the width and height of the box.

- *Class prediction*: The classic softmax function that predicts the class probability for each object.

Since thousands of regions are proposed, each object will always have multiple bounding boxes surrounding it with the correct classification. For example, take a look at the image of the dog in figure below. The network was clearly able to find the object (dog) and successfully classify it. But the detection fired a total of five times because the dog was present in the five RoIs produced in the previous step: hence the five bounding boxes around the dog in the figure. Although the detector was able to successfully locate the dog in the image and classify it correctly, this is not exactly what we need. We need just one bounding box for each object for most problems.

In some problems, we only want the one box that fits the object the most. What if we are building a system to count dogs in an image? Our current system will count five dogs. We don't want that. This is when the non-maximum suppression technique comes in handy.

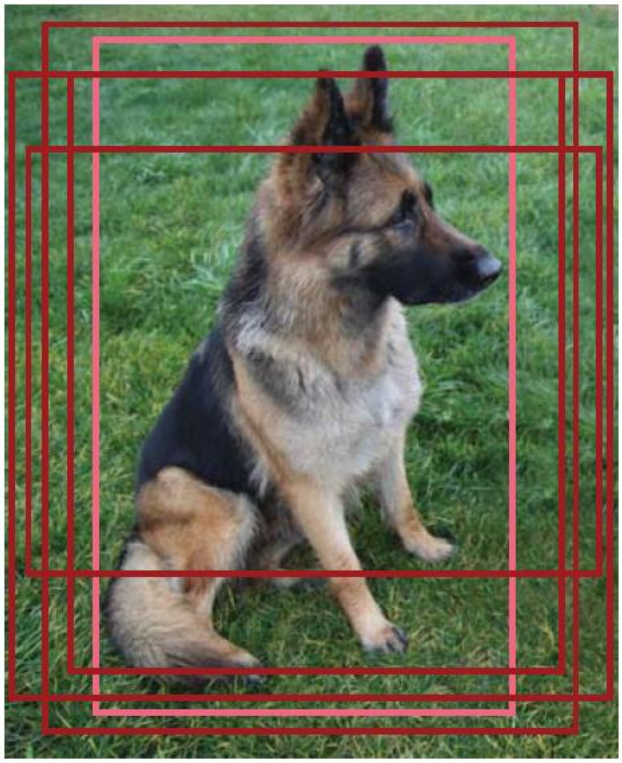


Figure 7.3 The bounding-box detector produces more than one bounding box for an object. We want to consolidate these boxes into one bounding box that fits the object the most.

3. ***Non-maximum suppression (NMS)***

As you can see in figure 7.4, one of the problems of an object detection algorithm is that it may find multiple detections of the same object. So, instead of creating only one bounding box around the object, it draws multiple boxes for the same object. NMS is a technique that makes sure the detection algorithm detects each object only once. As the name implies, NMS looks at all the boxes surrounding an object to find the box that has the *maximum* prediction probability, and it *suppresses* or eliminates the other boxes (hence the name).

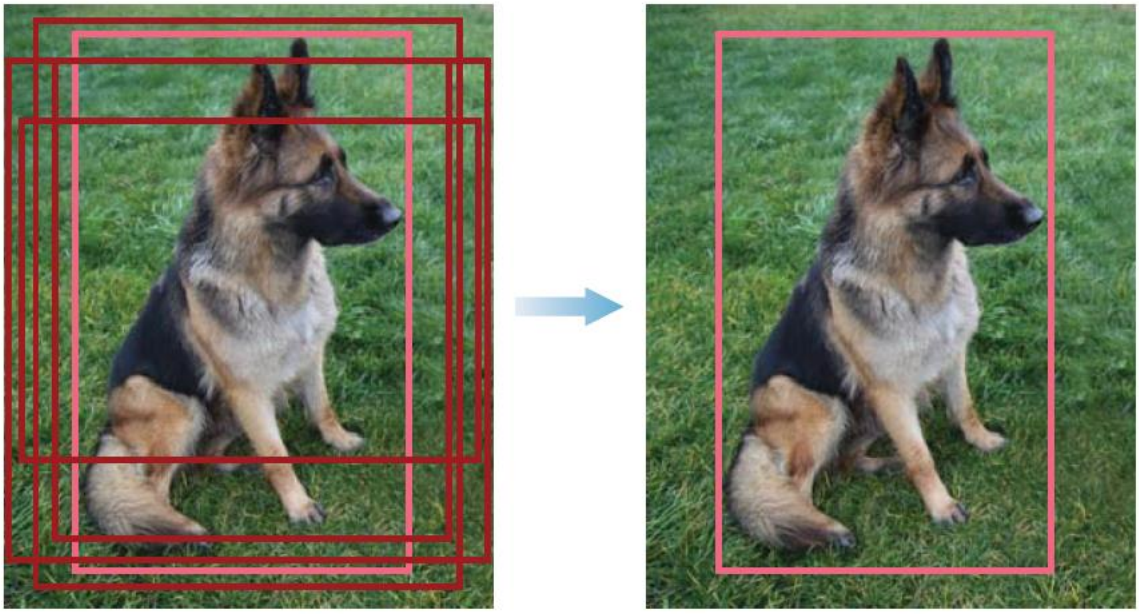


Figure 7.4 Multiple regions are proposed for the same object. After NMS, only the box that fits the object the best remains; the rest are ignored, as they have large overlaps with the selected box.

Let's see the steps of how the NMS algorithm works:

1 Discard all bounding boxes that have predictions that are less than a certain threshold, called the *confidence threshold*. This threshold is tunable, which means a box will be suppressed if the prediction probability is less than the set threshold.

2 Look at all the remaining boxes, and select the bounding box with the highest probability.

3 Calculate the overlap of the remaining boxes that have the same class prediction. Bounding boxes that have high overlap with each other and that predict the same class are averaged together. This overlap metric is called *intersection over union (IoU)*. IoU is explained in detail in the next section.

4 Suppress any box that has an IoU value smaller than a certain threshold (called the *NMS threshold*). Usually the NMS threshold is equal to 0.5, but it is tunable as well if you want to output fewer or more bounding boxes. NMS techniques are typically standard across the different detection frameworks, but it is an important step that may require tweaking hyperparameters such as the confidence threshold and the NMS threshold based on the scenario.

EXAMPLE:

We have:

Bounding Boxes: Four bounding boxes (A, B, C, D) with confidence scores.

Coordinates (x1, y1, x2, y2):

- A:(1,1,4,4), confidence=0.9
- B:(2,2,5,5), confidence=0.8
- C:(3,3,6,6), confidence=0.7
- D:(7,7,9,9), confidence=0.6

Thresholds:

- **Confidence Threshold:** 0.5.
- **IoU Threshold:** 0.5.

Step 1: Apply the Confidence Threshold

- Discard bounding boxes with confidence < 0.5.
- Result: All boxes are retained since their confidence scores are 0.9,0.8,0.7,0.6, all above 0.5.

Retained boxes: A, B, C, D

Step 2: Select the Box with the Highest Confidence

- Identify the box with the highest confidence score.
- Box A has the highest confidence 0.9
- **Retain box A** as a detection.

Step 3: Calculate IoU for Remaining Boxes

- Calculate the **IoU** of box A with all other boxes B,C,D.

$$\text{IoU} = \frac{\text{Area of Intersection}}{\text{Area of Union}}$$

1. **IOU of Box A and Box B**:
   - **Intersection**: From (2,2) to (4,4), area = $(4-2)\times(4-2)=4$
   - **Union**: Area of A = 9, Area of B = 9, Intersection = 4
   - Union = $9+9-4=14$.
   - **IoU** = $4/14\approx0.286$
2. **Box A and Box C**:
   - **Intersection**: None (no overlap).
   - **IoU** = 0.
3. **Box A and Box D**:
   - **Intersection**: None (no overlap).
   - **IoU** = 0.

Visualization of the Steps:

Box	Confidence	IoU (with A)	Retained?
A	0.9	-	Yes
B	0.8	0.286	Yes
C	0.7	0	Yes
D	0.6	0	Yes

- None of the IoUs calculated exceed 0.5.
- **Result:** No boxes are suppressed.

Object-detector evaluation metrics

When evaluating the performance of an object detector, we use two main evaluation metrics: frames per second and mean average precision.

FRAMES PER SECOND (FPS) TO MEASURE DETECTION SPEED

The most common metric used to measure detection speed is the number of frames per second (FPS). For example, Faster R-CNN operates at only 7 FPS, whereas SSD operates at 59 FPS. In benchmarking experiments, you will see the authors of a paper state their network results as: "Network X achieves mAP of Y% at Z FPS," where X is the network name, Y is the mAP percentage, and Z is the FPS.

MEAN AVERAGE PRECISION (MAP) TO MEASURE NETWORK PRECISION

The most common evaluation metric used in object recognition tasks is *mean average precision (mAP)*. It is a percentage from 0 to 100, and higher values are typically better, but its value is different from the accuracy metric used in classification. To understand how mAP is calculated, you first need to understand intersection over union (IoU) and the precision-recall curve (PR curve). Let's explain IoU and the PR curve and then come back to mAP.

INTERSECTION OVER UNION (IOU)

This measure evaluates the overlap between two bounding boxes: the ground truth bounding box (*B*ground truth) and the predicted bounding box (*B*predicted). By applying the IoU, we can tell whether a detection is valid (True Positive) or not (False Positive). Figure below illustrates the IoU between a ground truth bounding box and a predicted bounding box.

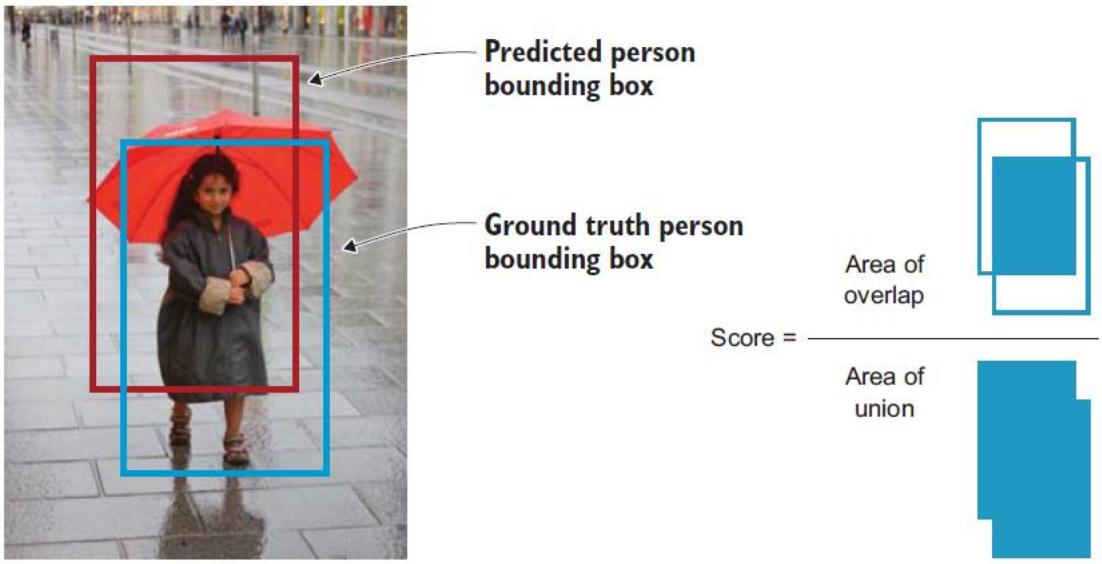


Figure 7.5 The IoU score is the overlap between the ground truth bounding box and the predicted bounding box.

The intersection over the union value ranges from 0 (no overlap at all) to 1 (the two bounding boxes overlap each other 100%). The higher the overlap between the two bounding boxes (IoU value), the better (figure 7.6).

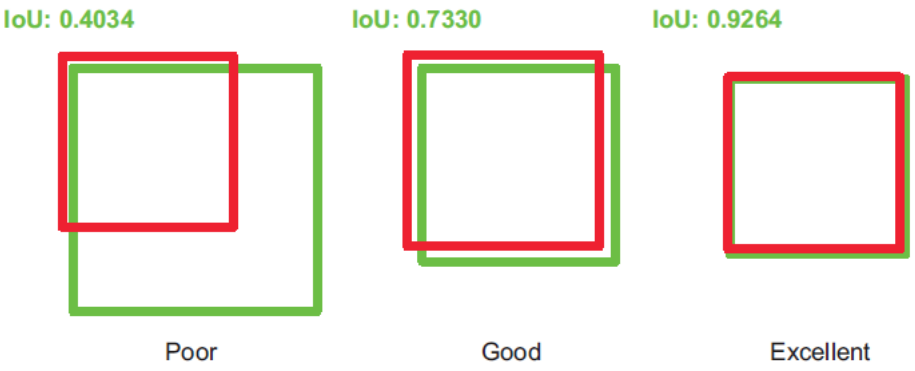


Figure 7.6 IoU scores range from 0 (no overlap) to 1 (100% overlap). The higher the overlap (IoU) between the two bounding boxes, the better.

To calculate the IoU of a prediction, we need the following:

- The ground truth bounding box (*B*ground truth): the hand-labeled bounding box created during the labeling process
- The predicted bounding box (*B*predicted) from our model We calculate IoU by dividing the area of overlap by the area of the union, as in the following equation:

$$\text{IoU} = \frac{B_{\text{ground truth}} \cap B_{\text{predicted}}}{B_{\text{ground truth}} \cup B_{\text{predicted}}}$$

IoU is used to define a *correct prediction*, meaning a prediction (True Positive) that has an IoU greater than some threshold. This threshold is a tunable value depending on the challenge, but 0.5 is a standard value. For example, some challenges, like Microsoft COCO, use mAP@0.5 (IoU threshold of 0.5) or mAP@0.75 (IoU threshold of 0.75). If the IoU value is above this threshold, the prediction is considered a True Positive (TP); and if it is below the threshold, it is considered a False Positive (FP).

PRECISION-RECALL CURVE (PR CURVE)

With the TP and FP defined, we can now calculate the precision and recall of our detection for a given class across the testing dataset. As explained in chapter 4, we calculate the precision and recall as follows (recall that FN stands for False Negative):

$$\text{Recall} = \frac{TP}{TP + FN}$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

After calculating the precision and recall for all classes, the PR curve is then plotted as shown in figure 7.7.

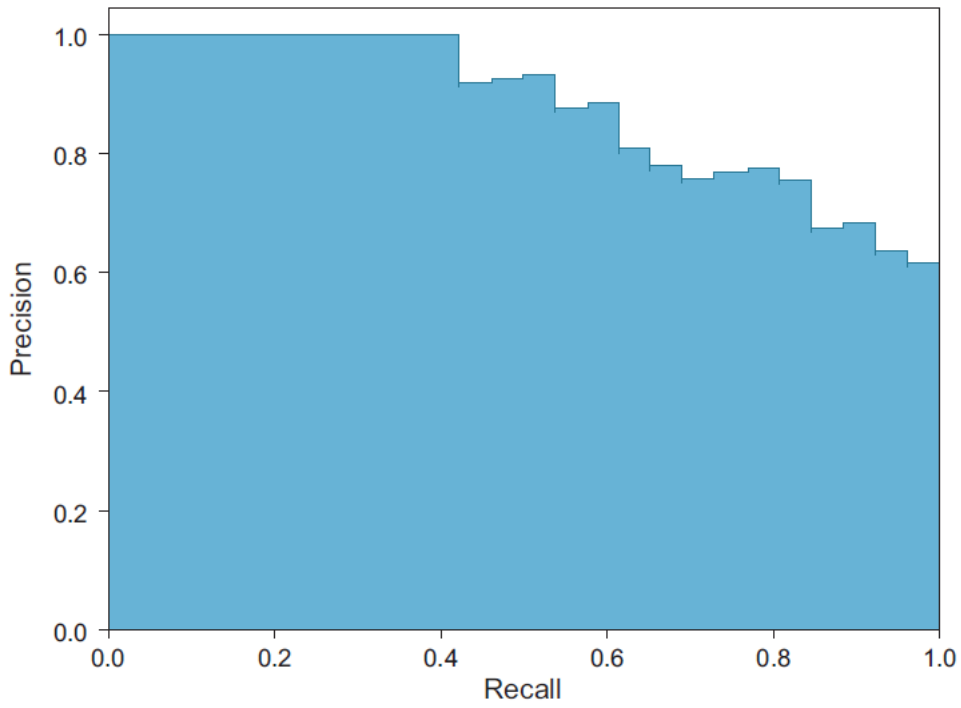


Figure 7.7 A precision-recall curve is used to evaluate the performance of an object detector.

Now that we have the PR curve, we can calculate the average precision (AP) by calculating the area under the curve (AUC). Finally, the mAP for object detection is the average of the AP calculated for all the classes. It is also important to note that some research papers use AP and mAP interchangeably.

RECAP:

To recap, the mAP is calculated as follows:

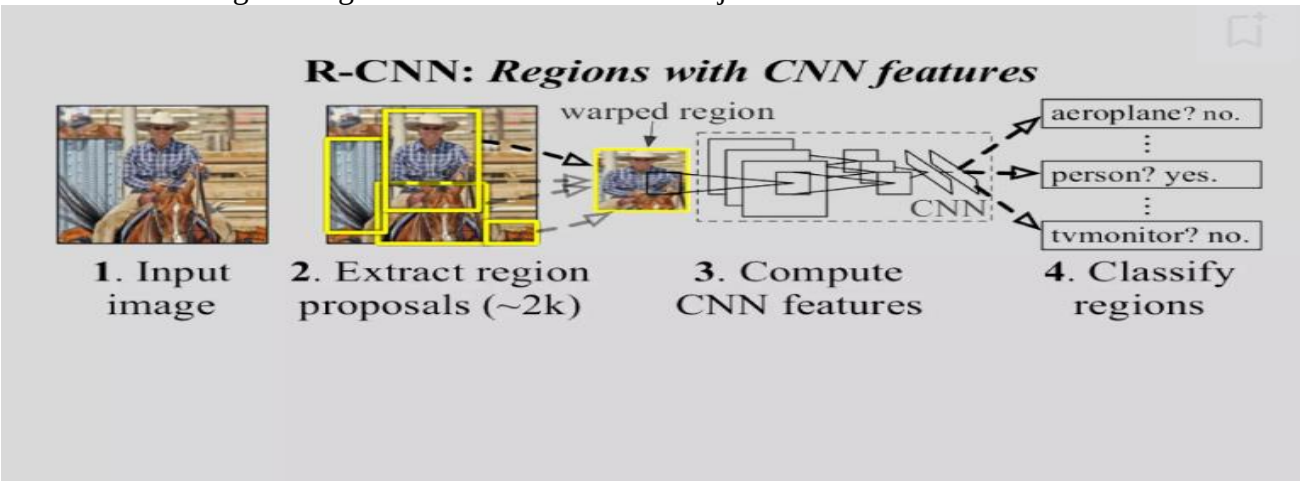
1. Get each bounding box's associated objectness score (probability of the box containing an object).
2. Calculate precision and recall.
3. Compute the PR curve for each class by varying the score threshold.
4. Calculate the AP: the area under the PR curve. In this step, the AP is computed for each class.
5. Calculate the mAP: the average AP over all the different classes.

The last thing to note about mAP is that it is more complicated to calculate than other traditional metrics like accuracy.

Region-based CNNs (R-CNNs)

R-CNNs, which is short for *region-based convolutional neural networks*, was developed by Ross Girshick et al. in 2014.

- R-CNNs divides an input image into regions, called "region proposals", and then pass each proposal through a CNN to extract features.
- The CNN features are used to classify the object's presence and class using Support Vector Machines (SVMs).
- A bounding box regressor then fine-tunes the object's location.



The R-CNN model consists of four components:

- Extract regions of interest
- Feature extraction module
- Classification module
- Localization module

The *R-CNN* first extracts many (e.g., 2000) *region proposals* from the input image (e.g., anchor boxes can also be considered as region proposals), labeling their classes and bounding boxes (e.g., offsets). (Girshick *et al.*, 2014)

Then a CNN is used to perform forward propagation on each region proposal to extract its features. Next, features of each region proposal are used for predicting the class and bounding box of this region proposal.

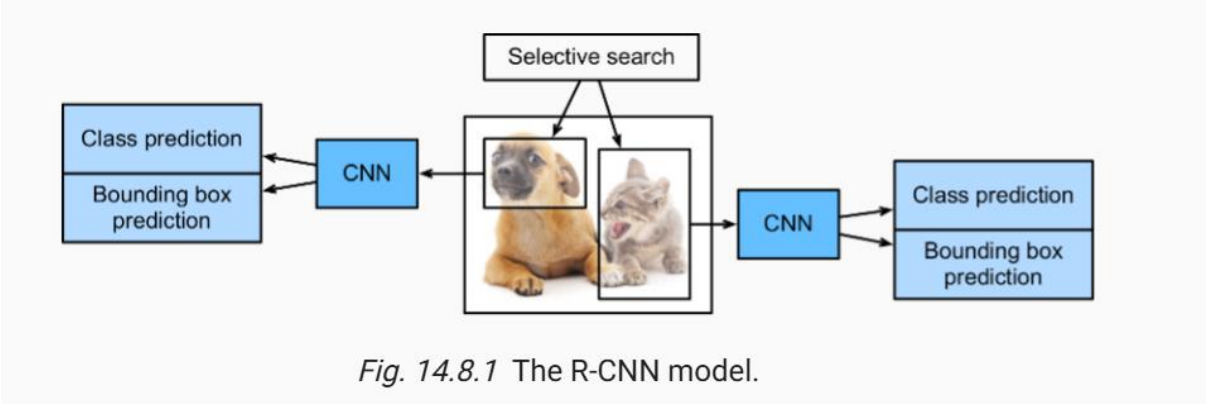


Fig. 14.8.1 The R-CNN model.

Fig. 14.8.1 shows the R-CNN model. More concretely, the R-CNN consists of the following four steps:

1. Perform *selective search* to extract multiple high-quality region proposals on the input image. These proposed regions are usually selected at multiple scales with different shapes and sizes. Each region proposal will be labeled with a class and a ground-truth bounding box.

2. Choose a pretrained CNN and truncate it before the output layer. Resize each region proposal to the input size required by the network, and output the extracted features for the region proposal through forward propagation.

3. Take the extracted features and labeled class of each region proposal as an example. Train multiple support vector machines to classify objects, where each support vector machine individually determines whether the example contains a specific class.

4. Take the extracted features and labeled bounding box of each region proposal as an example. Train a linear regression model to predict the ground-truth bounding box.

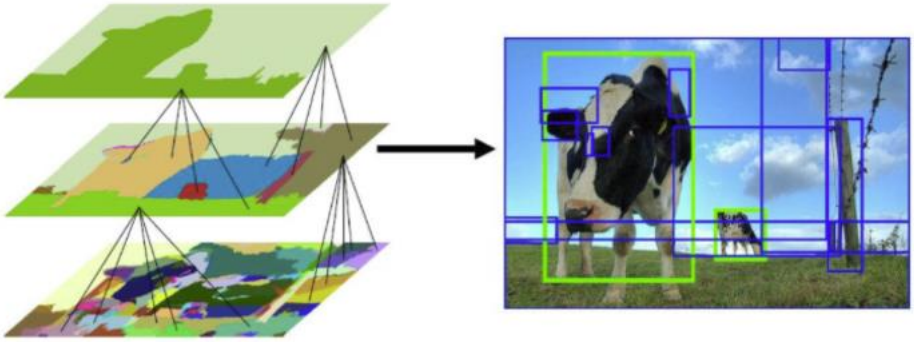
Key Features of R-CNNs

1. Region Proposals

R-CNNs begin by generating *region proposals*, which are smaller sections of the image that may contain the objects we are searching for.

The algorithm employs a method called *selective search*, a greedy approach that generates approximately 2,000 region proposals per image. Selective search effectively balances the number of proposals while maintaining high object recall, ensuring efficient object detection.

By limiting the number of regions for detailed analysis, this method enhances the overall performance of the R-CNN in detecting objects within images.



2. Selective Search

Selective Search is a greedy algorithm that generates region proposals by combining smaller segmented regions. It takes an image as input and produces region proposals that are crucial for object detection. This method offers significant advantages over random proposal generation by limiting the number of proposals to approximately 2,000 while ensuring high object recall.

Algorithm Steps:

1. **Generate Initial Segmentation**: The algorithm starts by performing an initial sub-segmentation of the input image.

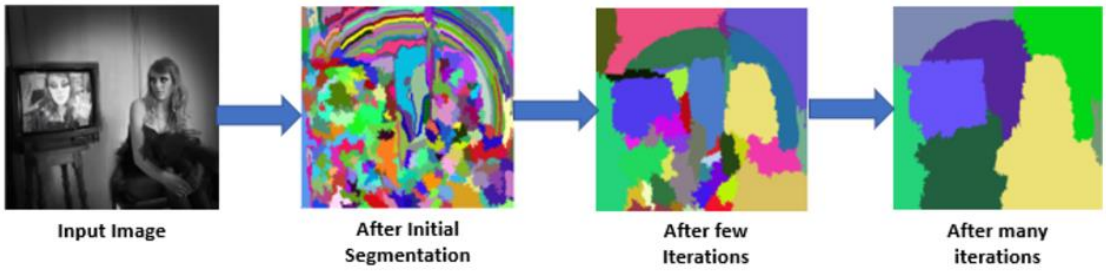
2. **Combine Similar Regions**: It then recursively combines similar bounding boxes into larger ones. Similarities are evaluated based on factors such as color, texture, and region size.

3. **Generate Region Proposals**: Finally, these larger bounding boxes are used to create region proposals for object detection.

The selective search algorithm provides an efficient way to identify potential object regions, enhancing the overall effectiveness of the detection process.

Selective Search, for example, operates by merging or splitting segments of the image based on various image cues like color, texture, and shape to create a diverse set of region proposals.

Below we show how Selective Search works, which shows an input image, then an image with many segmented masks, then fewer masks, then masks that comprise the main components in the image.

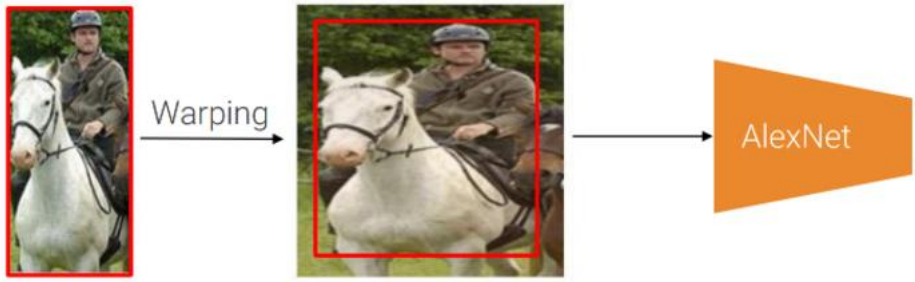


Feature Extraction

Once the region proposals are generated, approximately 2,000 regions are extracted and anisotropically warped to a consistent input size that the CNN expects (e.g., 224x224 pixels) and then it is passed through the CNN to extract features.

Before warping, the region size is expanded to a new size that will result in 16 pixels of context in the warped frame. The CNN used is AlexNet and it is typically fine-tuned on a large dataset like ImageNet for generic feature representation.

The output of the CNN is a high-dimensional feature vector representing the content of the region proposal.



An example of how warping works in R-CNN. Note the addition of 16 pixels of context in the middle image.

Object Classification

The extracted feature vectors from the region proposals are fed into a separate machine learning classifier for each object class of interest. R-CNN typically uses Support Vector Machines (SVMs) for classification. For each class, a unique SVM is trained to determine whether or not the region proposal contains an instance of that class.

During training, positive samples are regions that contain an instance of the class. Negative samples are regions that do not.

Bounding Box Regression

In addition to classifying objects, R-CNN also performs bounding box regression. For each class, a separate regression model is trained to refine the location and size of the bounding box around the detected object. The bounding box regression helps improve the accuracy of object localization by adjusting the initially proposed bounding box to better fit the object's actual boundaries.

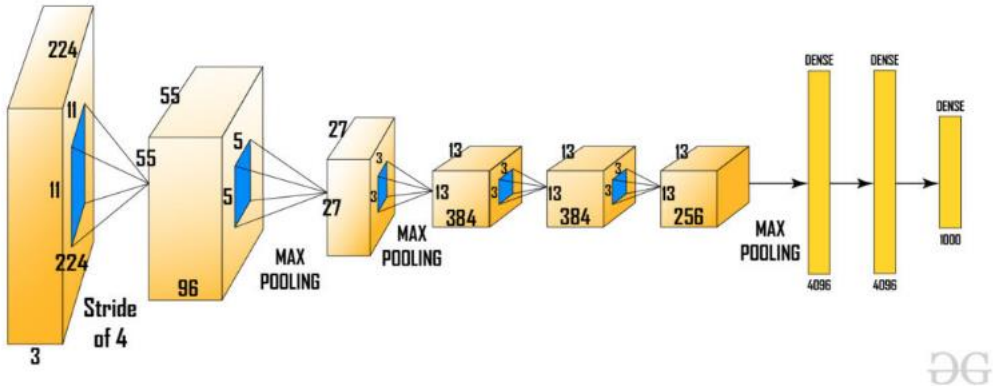
Non-Maximum Suppression (NMS)

After classifying and regressing bounding boxes for each region proposal, R-CNN applies non-maximum suppression to eliminate duplicate or highly overlapping bounding boxes. NMS ensures that only the most confident and non-overlapping bounding boxes are retained as final object detections.

3. Input Preparation in R-CNN

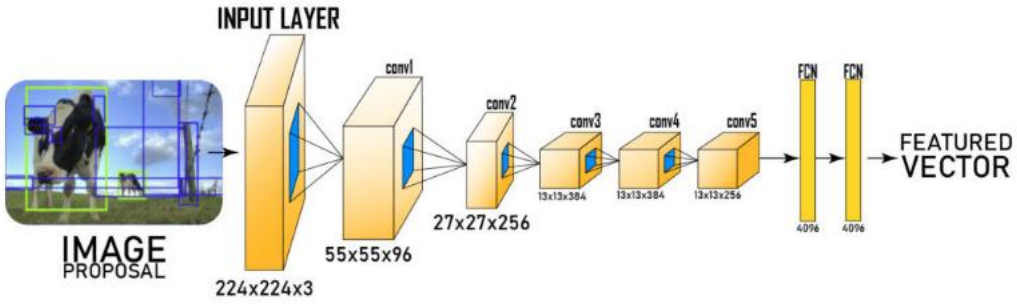
After generating the region proposals, these regions are warped into a uniform square shape to match the input dimensions required by the CNN model.

In this case, we use the pre-trained ***AlexNet model***, which was considered the state-of-the-art CNN for image classification at the time.



The input size for AlexNet is (227, 227, 3), meaning each input image must be resized to these dimensions. Consequently, whether the region proposals are small or large, they need to be adjusted accordingly to fit the specified input size.

From the above architecture, we remove the final softmax layer to obtain a (1, 4096) feature vector. This feature vector is then fed into both the Support Vector Machine (SVM) for classification and the bounding box regressor for improved localization.



4. SVM (Support Vector Machine)

The feature vector generated by the CNN is then utilized by a binary ***Support Vector Machine (SVM)***, which is trained independently for each class. This SVM model takes the feature vector produced by the previous CNN architecture and outputs a confidence score indicating the likelihood of an object being present in that region.

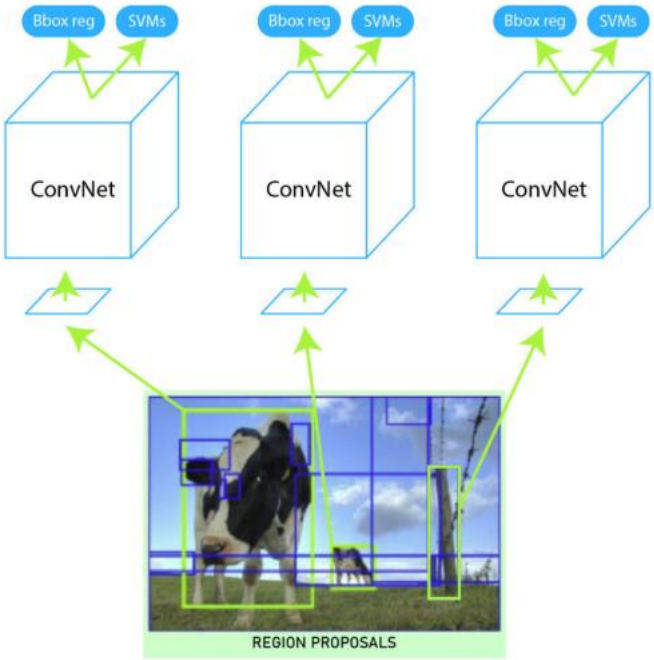
However, a challenge arises during the training process with the SVM: it requires the AlexNet feature vectors for each class. As a result, we cannot train AlexNet and the SVM independently and in parallel.

5. Bounding Box Regressor

To accurately locate the *bounding box* within the image, we utilize a scale-invariant linear regression model known as the ***bounding box regressor***.

For training this model, we use pairs of predicted and ground truth values for four dimensions of localization: . Here, and represent the pixel coordinates of the center of the bounding box, while and indicate the width and height of the bounding boxes, respectively.

*This method enhances the **Mean Average Precision (mAP)** of the results by 3-4%.*

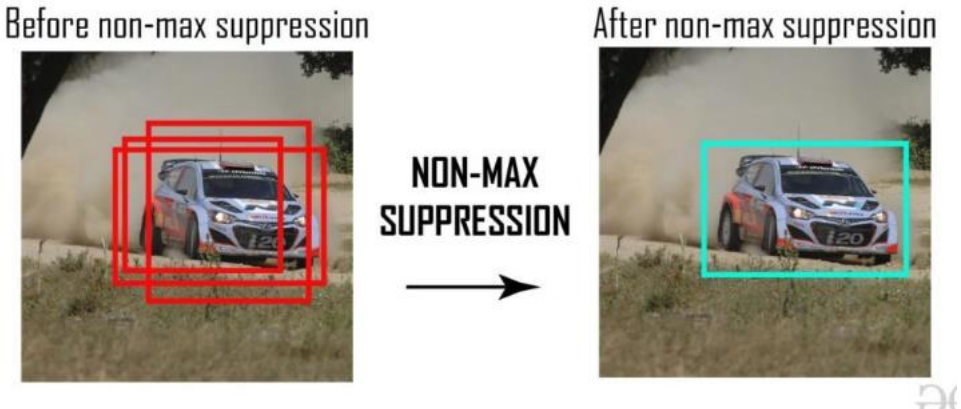


To further optimize detection, R-CNNs apply ***Non-Maximum Suppression (NMS)***:

1. Remove proposals with confidence scores below a threshold (e.g., 0.5).
2. Select the highest-probability region among candidates for each object.
3. Discard overlapping regions with an IoU (Intersection over Union) above 0.5 to eliminate duplicate detections, where IoU is defined as:

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

By combining region proposals, selective search, CNN-based feature extraction, SVM classification, and bounding box refinement, R-CNN achieves high accuracy in object detection, making it suitable for various applications.



After that, we can obtain output by plotting these bounding boxes on the input image and labeling objects that are present in bounding boxes.

Results of R-CNN Model

The R-CNN gives a Mean Average Precision (mAPs) of 53.7% on VOC 2010 dataset. On 200-class ILSVRC 2013 object detection dataset it gives an mAP of 31.4% which is a large improvement from the previous best of 24.3%. However, this architecture is very slow to train and takes ~ 49 sec to generate test results on a single image of the VOC 2007 dataset.

Challenges of R-CNN

R-CNN faces several challenges in its implementation:

1. **Rigid Selective Search Algorithm**: The selective search algorithm is inflexible and does not involve any learning. This rigidity can result in poor region proposal generation for object detection.

2. **Time-Consuming Training**: With approximately 2,000 candidate proposals, training the network becomes time-intensive. Additionally, multiple components need to be trained separately, including the CNN architecture, SVM model, and bounding box regressor. This multi-step training process slows down implementation.

3. **Inefficiency for Real-Time Applications**: R-CNN is not suitable for real-time applications, as it takes around 50 seconds to process a single image with the bounding box regressor.

4. **Increased Memory Requirements**: Storing feature maps for all region proposals significantly increases the disk memory needed during the training phase.

Fast R-CNN

Fast R-CNN was an immediate descendant of R-CNN, developed in 2015 by Ross Girshick.

Although the R-CNN model uses pretrained CNNs to effectively extract image features, it is slow. Imagine that we select thousands of region proposals from a single input image: this requires thousands of CNN forward propagations to perform object detection. This massive computing load makes it infeasible to widely use R-CNNs in real-world applications.

Fast R-CNN resembled the R-CNN technique in many ways but improved on its detection speed while also increasing detection accuracy through two main changes:

- Instead of starting with the regions proposal module and then using the feature extraction module, like R-CNN, Fast-RCNN proposes that we apply the CNN feature extractor first to the entire input image and then propose regions. This way, we run only one ConvNet over the entire image instead of 2,000 ConvNets over 2,000 overlapping regions.
- It extends the ConvNet's job to do the classification part as well, by replacing the traditional SVM machine learning algorithm with a softmax layer. This way, we have a single model to perform both tasks: feature extraction and object classification.

FAST R-CNN ARCHITECTURE:

As shown in figure below, Fast R-CNN generates region proposals based on the last feature map of the network, not from the original image like R-CNN. As a result, we can train just one ConvNet for the entire image. In addition, instead of training many different SVM algorithms to classify each object class, a single softmax layer outputs the class probabilities directly. Now we only have one neural net to train, as opposed to one neural net and many SVMs.

- Fast R-CNN generates region proposals based on the last feature map of the network, not from the original image like R-CNN.
- As a result, we can train just one ConvNet for the entire image.
- In addition, instead of training many different SVM algorithms to classify each object class, a single softmax layer outputs the class probabilities directly.
- Now we only have one neural net to train, as opposed to one neural net and many SVMs.

The architecture of Fast R-CNN consists of the following modules:

1. *Feature extractor module*—The network starts with a ConvNet to extract features from the full image.
2. *RoI extractor*—The selective search algorithm proposes about 2,000 region candidates per image.
3. *RoI pooling layer*—This is a new component that was introduced to extract a fixed-size window from the feature map before feeding the RoIs to the fully connected layers. It uses

max pooling to convert the features inside any valid RoI into a small feature map with a fixed spatial extent of height × width (*H* × *W*). Understand that it is applied on the last feature map layer extracted from the CNN, and its goal is to extract fixed-size RoIs to feed to the fully connected layers and then the output layers.

4. *Two-head output layer*—The model branches into two heads:
   – A softmax classifier layer that outputs a discrete probability distribution per RoI
   – A bounding-box regressor layer to predict offsets relative to the original RoI

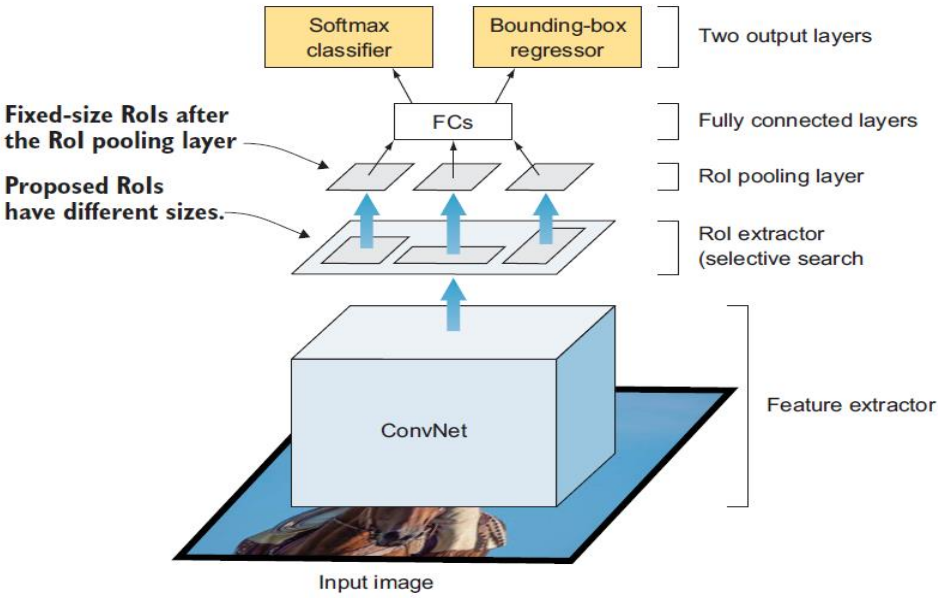


Figure 7.10 The Fast R-CNN architecture consists of a feature extractor ConvNet, RoI extractor, RoI pooling layers, fully connected layers, and a two-head output layer. Note that, unlike R-CNNs, Fast R-CNNs apply the feature extractor to the entire input image before applying the regions proposal module.

MULTI-TASK LOSS FUNCTION IN FAST R-CNNS

Since Fast R-CNN is an end-to-end learning architecture to learn the class of an object as well as the associated bounding box position and size, the loss is *multi-task loss*. With multi-task loss, the output has the softmax classifier and bounding-box regressor, as shown in figure 7.10.

In any optimization problem, we need to define a loss function that our optimizer algorithm is trying to minimize. (Chapter 2 gives more details about optimization and loss functions.) In object detection problems, our goal is to optimize for two goals: object classification and object localization. Therefore, we have two loss functions in this problem: *Lcls* for the classification loss and *Lloc* for the bounding box prediction defining the object location.

A Fast R-CNN network has two sibling output layers with two loss functions:

- *Classification*—The first outputs a discrete probability distribution (per RoI) over *K* + 1 categories (we add one class for the background). The probability *P* is computed by a softmax over the *K* + 1 outputs of a fully connected layer. The classification loss function is

a log loss for the true class *u*

$$L_{cls}(p,u) = -\log p_u$$

where *u* is the true label, $u \in 0, 1, 2, \ldots (K + 1)$; where u = 0 is the background;
and *p* is the discrete probability distribution per RoI over $K + 1$ classes.

- *Regression*—The second sibling layer outputs bounding box regression offsets $v = (x, y, w, h)$ for each of the *K* object classes. The loss function is the loss for bounding box for class *u*

$$L_{loc}(t^u, u) = \sum L1_{\text{smooth}}(t_i^u - v_i)$$

where:

- v is the true bounding box, $v = (x, y, w, h)$.
- t^u is the prediction bounding box correction:

$$t^u = (t_x^u, t_y^u, t_w^u, t_h^u)$$

- $L1_{\text{smooth}}$ is the bounding box loss that measures the difference between t_i^u and v_i using the smooth L1 loss function. It is a robust function and is claimed to be less sensitive to outliers than other regression losses like L2.

The overall loss function is

$$L = L_{cls} + L_{loc}$$

$$L(p, u, t^u, v) = L_{cls}(p, u) + [u \geq 1] l_{\text{box}}(t^u, v)$$

Note that $[u \geq 1]$ is added before the regression loss to indicate 0 when the region inspected doesn't contain any object and contains a background. It is a way of ignoring the bounding box regression when the classifier labels the region as a background. The indicator function $[u \geq 1]$ is defined as

$$[u \geq 1] = \begin{cases} 1 & \text{if } u \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

Fast R-CNN resembled the R-CNN technique in many ways but improved on its detection speed while also increasing detection accuracy through two main changes:

- Instead of starting with the regions proposal module and then using the feature extraction module, like R-CNN, Fast-RCNN proposes that we apply the CNN feature extractor first to the entire input image and then propose regions. This way, we run only one ConvNet over the entire image instead of 2,000 ConvNets over 2,000 overlapping regions.

- It extends the ConvNet's job to do the classification part as well, by replacing the traditional SVM machine learning algorithm with a softmax layer. This way, we have a single model to perform both tasks: feature extraction and object classification.

DISADVANTAGES OF FAST R-CNN

- Fast R-CNN is much faster in terms of testing time, because we don't have to feed 2,000 region proposals to the convolutional neural network for every image. Instead, a convolution operation is done only once per image, and a feature map is generated from it. Training is also faster because all the components are in one CNN network: feature extractor, object classifier, and bounding-box regressor. However, there is a big bottleneck remaining: the selective search algorithm for generating region proposals is very slow and is generated separately by another model. The last step to achieve a complete end-to-end object detection system using DL is to find a way to combine the region proposal algorithm into our end-to-end DL network. This is what Faster R-CNN does.

Faster R-CNN

Faster R-CNN is the third iteration of the R-CNN family, developed in 2016 by Shaoqing Ren et al. Similar to Fast R-CNN, the image is provided as an input to a convolutional network that provides a convolutional feature map. Instead of using a selective search algorithm on the feature map to identify the region proposals, a *region proposal network (RPN)* is used to predict the region proposals as part of the training process.

The predicted region proposals are then reshaped using an RoI pooling layer and used to classify the image within the proposed region and predict the offset values for the bounding boxes. These improvements both reduce the number of region proposals and accelerate the test-time operation of the model to near real-time with thenstate- of-the-art performance.

FASTER R-CNN ARCHITECTURE

The architecture of Faster R-CNN can be described using two main networks:

- *Region proposal network (RPN)*—Selective search is replaced by a ConvNet that proposes RoIs from the last feature maps of the feature extractor to be considered for investigation. The RPN has two outputs: the objectness score (object or no object) and the box location.
- *Fast R-CNN*—It consists of the typical components of Fast R-CNN:
  – Base network for the feature extractor: a typical pretrained CNN model to extract features from the input image
  – RoI pooling layer to extract fixed-size RoIs
  – Output layer that contains two fully connected layers: a softmax classifier to output the class probability and a bounding box regression CNN for the bounding box predictions

As you can see in figure below, the input image is presented to the network, and its features are extracted via a pretrained CNN. These features, in parallel, are sent to two different components of the Faster R-CNN architecture:

- The RPN to determine where in the image a potential object could be. At this point, we do not know what the object is, just that there is potentially an object at a certain location in the image.
- RoI pooling to extract fixed-size windows of features.

The output is then passed into two fully connected layers: one for the object classifier and one for the bounding box coordinate predictions to obtain our final localizations. This architecture achieves an end-to-end trainable, complete object detection pipeline where all of the required components are inside the network:

- Base network feature extractor
- Regions proposal
- RoI pooling
- Object classification
- Bounding-box regressor

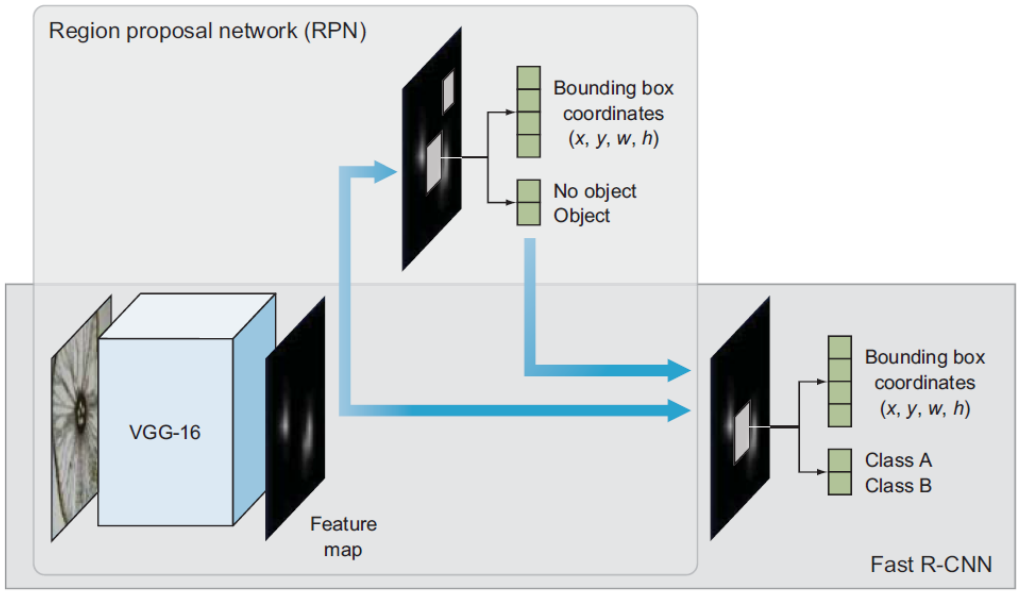


Figure 7.11 The Faster R-CNN architecture has two main components: an RPN that identifies regions that may contain objects of interest and their approximate location, and a Fast R-CNN network that classifies objects and refines their location defined using bounding boxes. The two components share the convolutional layers of the pretrained VGG16.

BASE NETWORK TO EXTRACT FEATURES

Similar to Fast R-CNN, the first step is to use a pretrained CNN and slice off its classification part. The base network is used to extract features from the input image. We covered how this works in detail in chapter 6. In this component, you can use any of the popular CNN architectures based on the problem you are trying to solve. The original Faster R-CNN paper used ZF4 and VGG5 pretrained networks on ImageNet; but since then, there have been lots of different networks with a varying number of weights. For example, MobileNet,6 a smaller and efficient network architecture optimized for speed, has approximately 3.3 million parameters, whereas ResNet-152 (152 layers)—once the state of the art in the ImageNet classification competition—has around 60 million. Most recently, new architectures like DenseNet7 are both improving results and reducing the number of parameters.

REGION PROPOSAL NETWORK (RPN)

The RPN identifies regions that could potentially contain objects of interest, based on the last feature map of the pretrained convolutional neural network. An RPN is also known as an *attention network* because it guides the network's attention to interesting regions in the image. Faster R-CNN uses an RPN to bake the region proposal directly into the R-CNN architecture instead of running a selective

search algorithm to extract RoIs.

The architecture of the RPN is composed of two layers (figure 7.12):

 A 3 × 3 fully convolutional layer with 512 channels

 Two parallel 1 × 1 convolutional layers: a classification layer that is used to predict whether the region contains an object (the score of it being background or foreground), and a layer for regression or bounding box prediction.

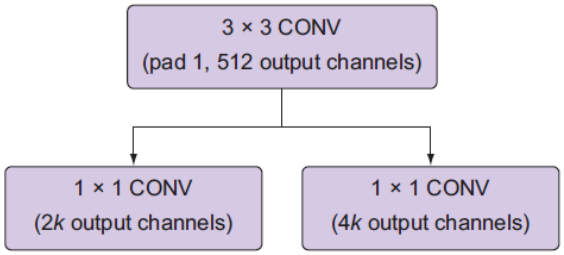


Figure 7.12 Convolutional implementation of an RPN architecture, where *k* is the number of anchors

Fully convolutional networks (FCNs)

One important aspect of object detection networks is that they should be fully convolutional. A fully convolutional neural network means that the network does not contain any fully connected layers, typically found at the end of a network prior to making output predictions. In the context of image classification, removing the fully connected layers is normally accomplished by applying average pooling across the entire volume prior to using a single dense softmax classifier to output the final predictions. An FCN has two main benefits:

 It is faster because it contains only convolution operations and no fully connected layers.

 It can accept images of any spatial resolution (width and height), provided the image and network can fit into the available memory.

Being an FCN makes the network invariant to the size of the input image. However, in practice, we might want to stick to a constant input size due to issues that only become apparent when we are implementing the algorithm. A significant such problem is that if we want to process images in batches (because images in batches can be processed in parallel by the GPU, leading to speed boosts), all of the images must have a fixed height and width.

The 3 × 3 convolutional layer is applied on the last feature map of the base network where a sliding window of size 3 × 3 is passed over the feature map. The output is then passed to two 1 × 1 convolutional layers: a classifier and a bounding-box regressor. Note that the classifier and the regressor of the RPN are not trying to predict the class of the object and its bounding box; this comes later, after the RPN. Remember, the goal of the RPN is to determine whether the region has an object to be investigated afterward by the fully connected layers. In the RPN, we use a binary classifier to

predict the objectness score of the region, to determine the probability of this region being a foreground (contains an object) or a background (doesn't contain an object). It basically looks at the region and asks, "Does this region contain an object?" If the answer is yes, then the region is passed along for further investigation by RoI pooling and the final output layers (see figure 7.13).

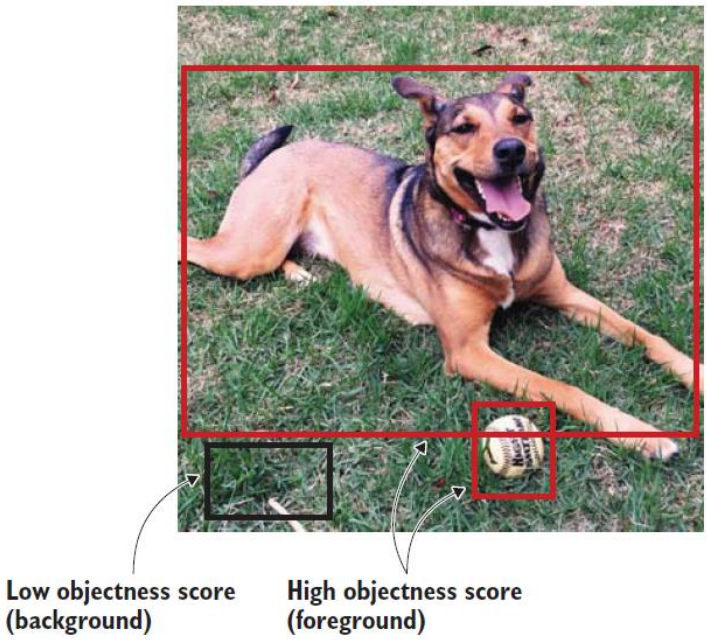


Figure 7.13 The RPN classifier predicts the objectness score, which is the probability of an image containing an object (foreground) or a background.

How does the regressor predict the bounding box?

To answer this question, let's first define the bounding box. It is the box that surrounds the object and is identified by the tuple (x, y, w, h), where x and y are the coordinates in the image that describes the center of the bounding box and h and w are the height and width of the bounding box. Researchers have found that defining the (x, y) coordinates of the center point can be challenging because we have to enforce some rules to make sure the network predicts values *inside* the boundaries of the image. Instead, we can create reference boxes called *anchor boxes* in the image and make the regression layer predict offsets from these boxes called *deltas* (Δx, Δy, Δw, Δh) to adjust the anchor boxes to better fit the object to get final proposals (figure 7.14).

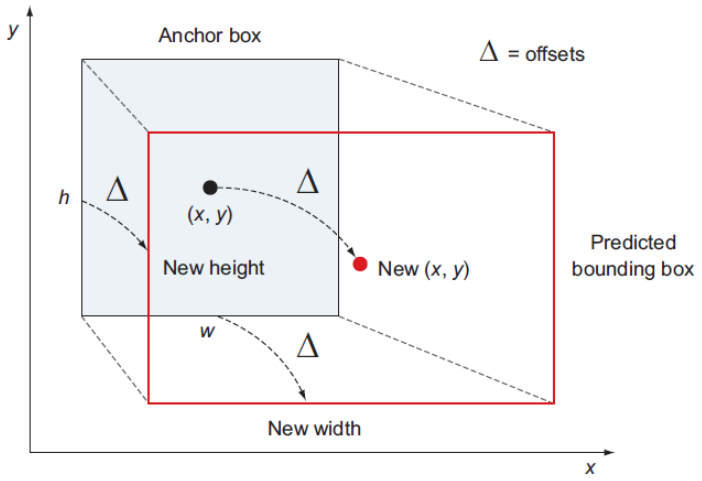


Figure 7.14 Illustration of predicting the delta shift from the anchor boxes and the bounding box coordinates

Anchor boxes

Using a sliding window approach, the RPN generates *k* regions for each location in the feature map. These regions are represented as *anchor boxes*. The anchors are centered in the middle of their corresponding sliding window and differ in terms of scale and aspect ratio to cover a wide variety of objects. They are fixed bounding boxes that are placed throughout the image to be used for reference when first predicting object locations. In their paper, Ren et. al. generated nine anchor boxes that all had the same center but that had three different aspect ratios and three different scales.

Figure 7.15 shows an example of how anchor boxes are applied. Anchors are at the center of the sliding windows; each window has *k* anchor boxes with the anchor at their center.

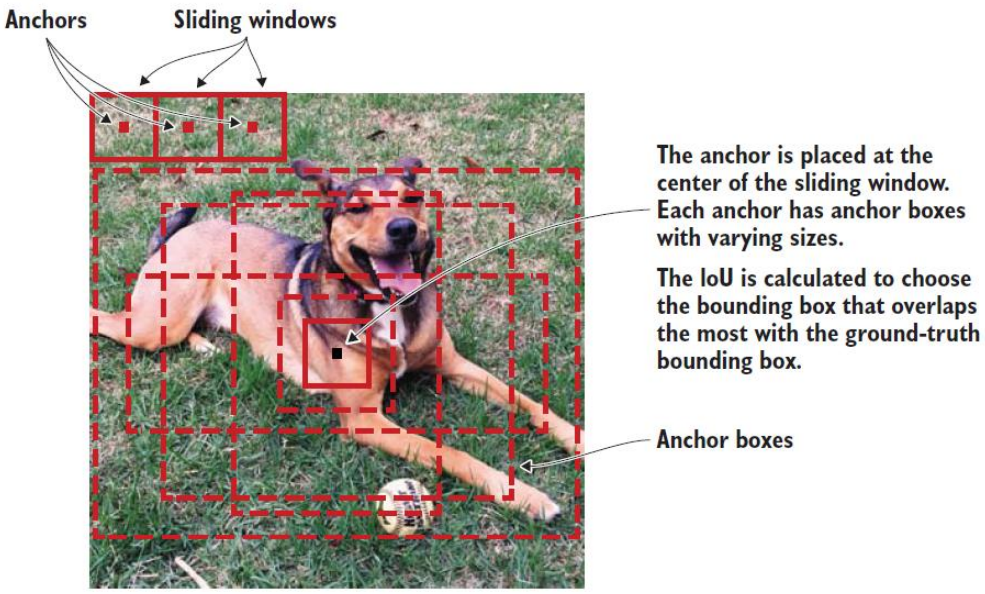


Figure 7.15 Anchors are at the center of each sliding window. IoU is calculated to select the bounding box that overlaps the most with the ground truth.

Training the RPN

The RPN is trained to classify an anchor box to output an objectness score and to approximate the four coordinates of the object (location parameters). It is trained using human annotators to label the bounding boxes. A labeled box is called the ground truth. For each anchor box, the overlap probability value (p) is computed, which indicates how much these anchors overlap with the ground-truth bounding boxes:

$$p = \begin{cases} 1 & \text{if} \quad \text{IoU} > 0.7 \\ -1 & \text{if} \quad \text{IoU} < 0.3 \\ 0 & \text{otherwise} \end{cases}$$

If an anchor has high overlap with a ground-truth bounding box, then it is likely that the anchor box includes an object of interest, and it is labeled as positive with respect to the *object versus no object* classification task. Similarly, if an anchor has small overlap with a ground-truth bounding box, it is labeled as negative. During the training process, the positive and negative anchors are passed as input to two fully connected layers corresponding to the classification of anchors as containing an object or no object, and to the regression of location parameters (four coordinates), respectively. Corresponding to the *k* number of anchors from a location, the RPN network outputs 2*k* scores and 4*k* coordinates. Thus, for example, if the number of anchors per sliding window (*k*) is 9, then the RPN outputs 18 objectness scores and 36 location coordinates (figure 7.16).

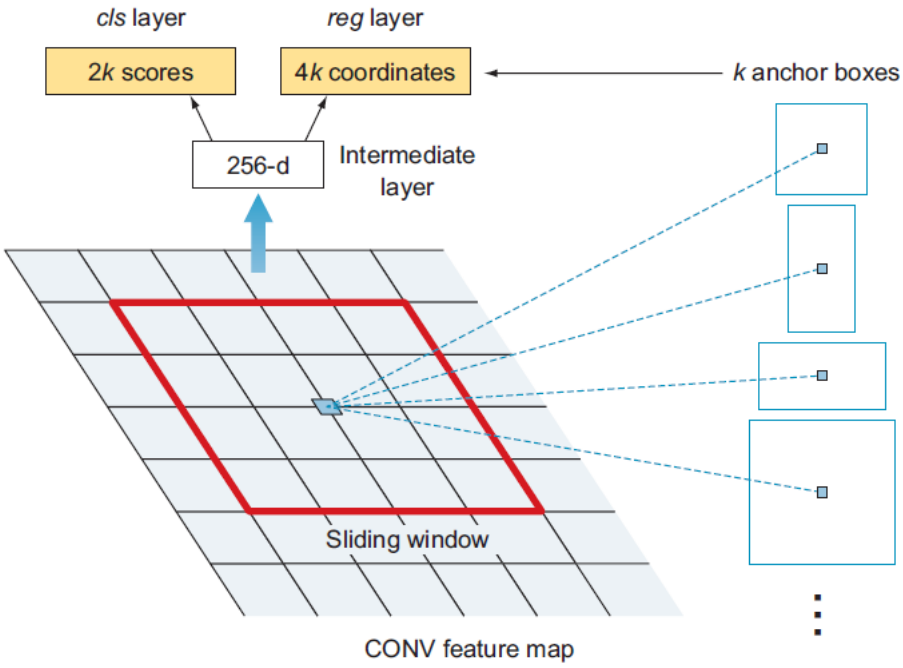


Figure 7.16 Region proposal network

MULTI-TASK LOSS FUNCTION

Similar to Fast R-CNN, Faster R-CNN is optimized for a multi-task loss function that combines the losses of classification and bounding box regression:

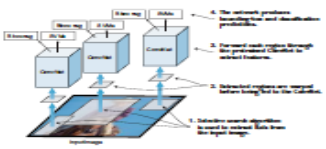
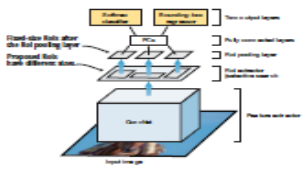
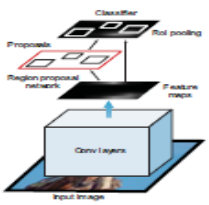
$$L = L_{cls} + L_{loc}$$

$$L(\{p_i\},\{t_i\}) = \frac{1}{N_{cls}} \sum L_{cls}(p_i, p_i^*) + \frac{\lambda}{N_{loc}} \sum p_i^* \cdot L1_{\text{smooth}}(t_i - t_i^*)$$

Table 7.2 Multi-task loss function symbols

Symbol	Explanation
p_i and p_i^*	p_i is the predicted probability of the anchor (i) being an object and the ground, and p_i^* is the binary ground truth (0 or 1) of the anchor being an object.
t_i and t_i^*	t_i is the predicted four parameters that define the bounding box, and t_i^* is the ground-truth parameters.
N_{cls}	Normalization term for the classification loss. Ren et al. set it to be a mini-batch size of ~256.
N_{loc}	Normalization term for the bounding box regression. Ren et al. set it to the number of anchor locations, ~2400.
$L_{cls}(p_i, p_i^*)$	The log loss function over two classes. We can easily translate a multi-class classification into a binary classification by predicting whether a sample is a target object: $L_{cls}(p_i, p_i^*) = -p_i^* \log p_i - (1 - p_i^*) \log (1 - p_i)$
$L1_{\text{smooth}}$	As described in section 7.2.2, the bounding box loss measures the difference between the predicted and true location parameters (t_i, t_i^*) using the smooth L1 loss function. It is a robust function and is claimed to be less sensitive to outliers than other regression losses like L2.
λ	A balancing parameter, set to be ~10 in Ren et al. (so the L_{cls} and L_{loc} terms are roughly equally weighted).

Table 7.3 The evolution of the CNN family of networks from R-CNN to Fast R-CNN to Faster R-CNN

	R-CNN	Fast R-CNN	Faster R-CNN
			
mAP on the PASCAL Visual Object Classes Challenge 2007	66.0%	66.9%	66.9%
Features	1 Applies selective search to extract RoIs (~2,000) from each image. 2 A ConvNet is used to extract features from each of the ~2,000 regions extracted. 3 Uses classification and bounding box predictions.	Each image is passed only once to the CNN, and feature maps are extracted. 1 A ConvNet is used to extract feature maps from the input image. 2 Selective search is used on these maps to generate predictions. This way, we run only one ConvNet over the entire image instead of ~2,000 ConvNets over 2000 overlapping regions.	Replaces the selective search method with a region proposal network, which makes the algorithm much faster. An end-to-end DL network.
Limitations	High computation time, as each region is passed to the CNN separately. Also, uses three different models for making predictions.	Selective search is slow and, hence, computation time is still high.	Object proposal takes time. And as there are different systems working one after the other, the performance of systems depends on how the previous system performed.
Test time per image	50 seconds	2 seconds	0.2 seconds
Speed-up from R-CNN	1x	25x	250x

Single-shot detector (SSD)

The SSD paper was released in 2016 by Wei Liu et al.8 The SSD network reached new records in terms of performance and precision for object detection tasks, scoring over 74% mAP at 59 FPS on standard datasets such as the PASCAL VOC and Microsoft COCO.

We learned earlier that the R-CNN family are multi-stage detectors: the network first predicts the objectness score of the bounding box and then passes this box through a classifier to predict the class probability. In single-stage detectors like SSD and YOLO (discussed in section 7.4), the convolutional layers make both predictions directly in one shot: hence the name single-shot detector. The image is passed once through the network, and the objectness score for each bounding box is predicted using logistic regression to indicate the level of overlap with the ground truth. If the bounding box overlaps 100% with the ground truth, the objectness score is 1; and if there is no overlap, the objectness score is 0. We then set a threshold value (0.5) that says, "If the objectness score is above 50%, this bounding box likely has an object of interest, and we get predictions. If it is less than 50%, we ignore the predictions."

High-level SSD architecture:

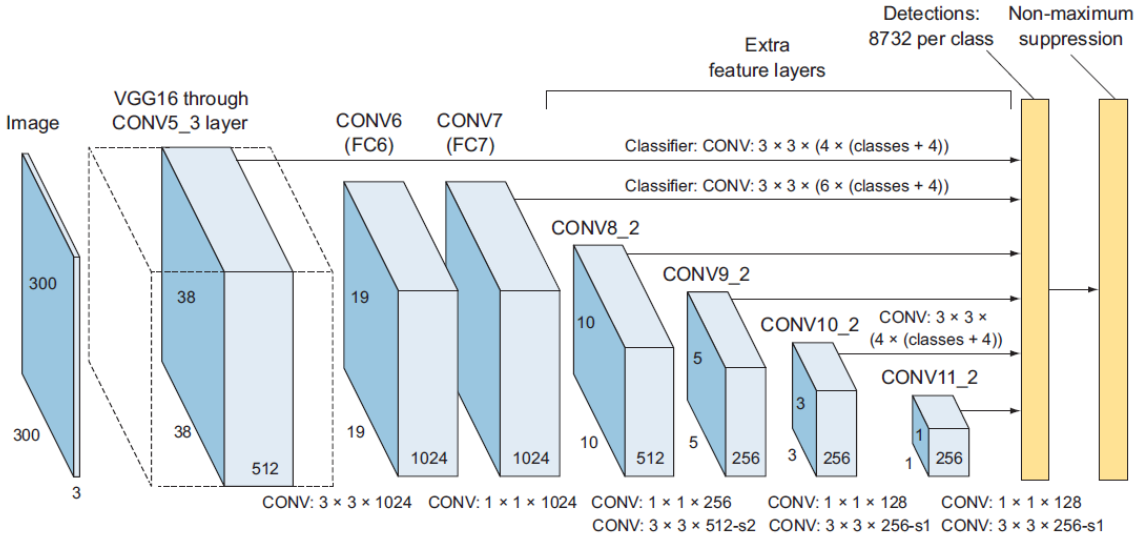


Figure 7.17 The SSD architecture is composed of a base network (VGG16), extra convolutional layers for object detection, and a non-maximum suppression (NMS) layer for final detections. Note that convolution layers 7, 8, 9, 10, and 11 make predictions that are directly fed to the NMS layer. (*Source:* Liu et al., 2016.)

The SSD approach is based on a feed-forward convolutional network that produces a fixed-size collection of bounding boxes and scores for the presence of object class instances in those boxes, followed by a NMS step to produce the final detections. The architecture of the SSD model is composed of three main parts:

 Base network to extract feature maps—A standard pretrained network used for high-quality image classification, which is truncated before any classification layers. In their paper, Liu et al. used a

VGG16 network. Other networks like VGG19 and ResNet can be used and should produce good results.

 Multi-scale feature layers—A series of convolution filters are added after the base network. These layers decrease in size progressively to allow predictions of detections at multiple scales.

 Non-maximum suppression—NMS is used to eliminate overlapping boxes and keep only one box for each object detected.

As you can see in figure 7.17, layers 4_3, 7, 8_2, 9_2, 10_2, and 11_2 make predictions directly to the NMS layer.

You can see in figure 7.17, that the network makes a total of 8,732 detections per class that are then fed to an NMS layer to reduce down to one detection per object. Where did the number 8,732 come from? To have more accurate detection, different layers of feature maps also go through a small 3 × 3 convolution for object detection. For example, Conv4_3 is of size 38 × 38 × 512, and a 3 × 3 convolutional is applied. There are four bounding boxes, each of which has (*number of classes* + 4 box values) outputs. Suppose there are 20 object classes plus 1 background class; then the output number of bounding boxes is 38 × 38 × 4 = 5,776 bounding boxes. Similarly, we calculate the number of bounding boxes for the other convolutional layers:

 Conv7: 19 × 19 × 6 = 2,166 boxes (6 boxes for each location)

 Conv8_2: 10 × 10 × 6 = 600 boxes (6 boxes for each location)

 Conv9_2: 5 × 5 × 6 = 150 boxes (6 boxes for each location)

 Conv10_2: 3 × 3 × 4 = 36 boxes (4 boxes for each location)

 Conv11_2: 1 × 1 × 4 = 4 boxes (4 boxes for each location)

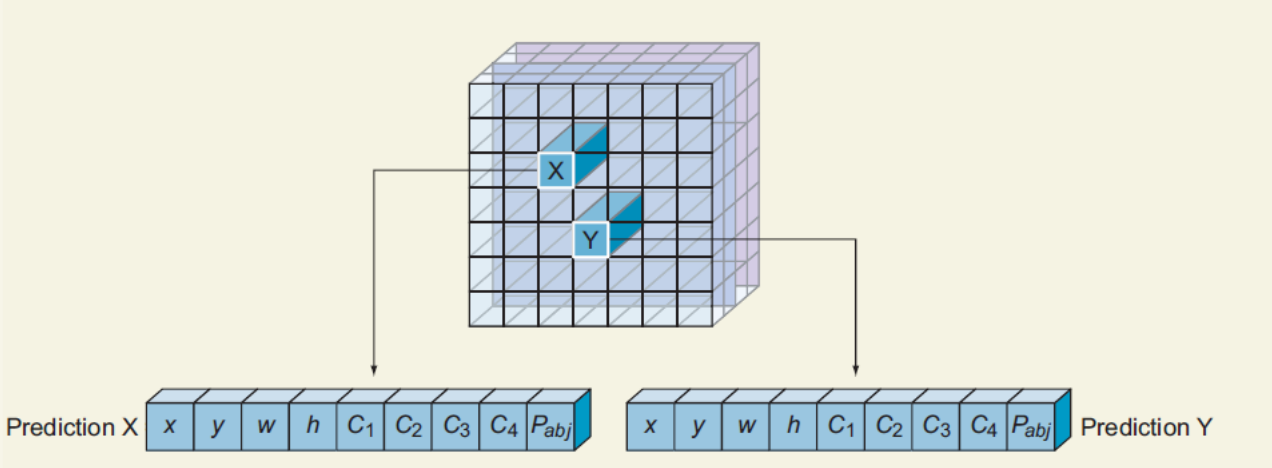
If we sum them up, we get 5,776 + 2,166 + 600 + 150 + 36 + 4 = 8,732 boxes produced. This is a huge number of boxes to show for our detector. That’s why we apply NMS to reduce the number of the output boxes. As you will see in section 7.4, in YOLO there are 7 × 7 locations at the end with two bounding boxes for each location: 7 × 7 × 2 = 98 boxes.

What does the output prediction look like?

For each feature, the network predicts the following:

- 4 values that describe the bounding box (x, y, w, h)
- 1 value for the objectness score
- C values that represent the probability of each class

That's a total of 5 + C prediction values. Suppose there are four object classes in our problem. Then each prediction will be a vector that looks like this: [x, y, w, h, *objectness score*, C_1, C_2, C_3, C_4].



An example visualization of the output prediction when we have four classes in our problem. The convolutional layer predicts the bounding box coordinates, objectness score, and four class probabilities: C_1, C_2, C_3, and C_4.

Base network:

As you can see in figure 7.17, the SSD architecture builds on the VGG16 architecture after slicing off the fully connected classification layers (VGG16 is explained in detail in chapter 5). VGG16 was used as the base network because of its strong performance in high-quality image classification tasks and its popularity for problems where transfer learning helps to improve results. Instead of the original VGG fully connected layers, a set of supporting convolutional layers (from Conv6 onward) was added to enable us to extract features at multiple scales and progressively decrease the size of the input to each subsequent layer.

HOW THE BASE NETWORK MAKES PREDICTIONS

Consider the following example. Suppose you have the image in figure 7.18, and the network's job is to draw bounding boxes around all the boats in the image. The process goes as follows:

1 Similar to the anchors concept in R-CNN, SSD overlays a grid of anchors around the image. For each anchor, the network creates a set of bounding boxes at its center. In SSD, anchors are called *priors*.

2 The base network looks at each bounding box as a separate image. For each bounding box, the network asks, "Is there a boat in this box?" Or in other words, "Did I extract any features of a boat in this box?"

3 When the network finds a bounding box that contains boat features, it sends its coordinates prediction and object classification to the NMS layer.

4 NMS eliminates all the boxes except the one that overlaps the most with the ground-truth bounding box.

NOTE: Liu et al. used VGG16 because of its strong performance in complex image classification tasks. You can use other networks like the deeper VGG19 or ResNet for the base network, and it should perform as well if not better in accuracy

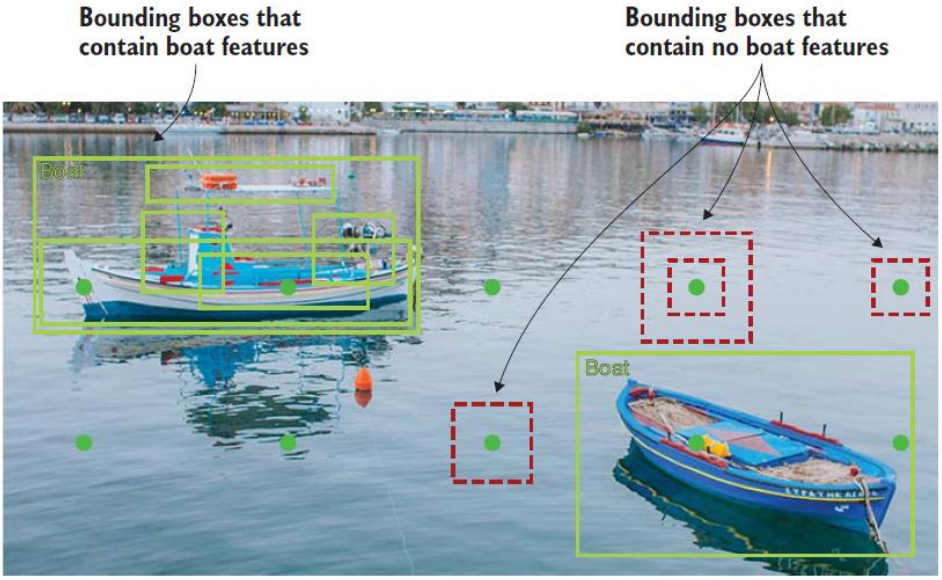


Figure 7.18 The SSD base network looks at the anchor boxes to find features of a boat. Solid boxes indicate that the network has found boat features. Dotted boxes indicate no boat features.

Multi-scale feature layers

These are convolutional feature layers that are added to the end of the truncated base network. These layers decrease in size progressively to allow predictions of detections at multiple scales.

MULTI-SCALE DETECTIONS

To understand the goal of the multi-scale feature layers and why they vary in size, let's look at the image of horses in figure 7.19. As you can see, the base network may be able to detect the horse features in the background, but it may fail to detect the horse that is closest to the camera. To understand why, take a close look at the dotted bounding box and try to imagine this box alone outside the context of the full image (see figure 7.20).

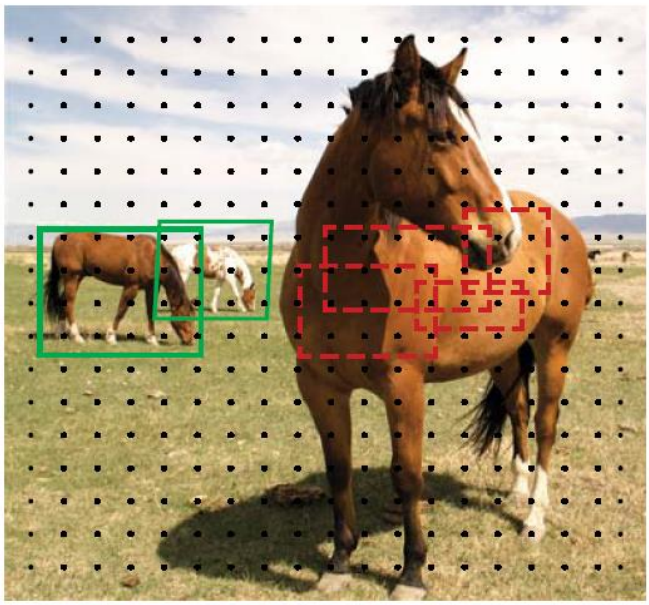


Figure 7.19 Horses at different scales in an image. The horses that are far from the camera are easier to detect because they are small in size and can fit inside the priors (anchor boxes). The base network might fail to detect the horse closest to the camera because it needs a different scale of anchors to be able to create priors that cover more identifiable features.

Figure 7.20 An isolated horse feature

To visualize this, imagine that the network reduces the image dimensions to be able to fit all of the horses inside its bounding boxes (figure 7.22). The multi-scale feature layers resize the image dimensions and keep the bounding-box sizes so that they can fit the larger horse. In reality, convolutional layers do not literally reduce the size of the image; this is just for illustration to help us intuitively understand the concept. The image is not just resized, it actually goes through the convolutional process and thus won't look anything like itself anymore. It will be a completely random-looking image, but it will preserve its features.

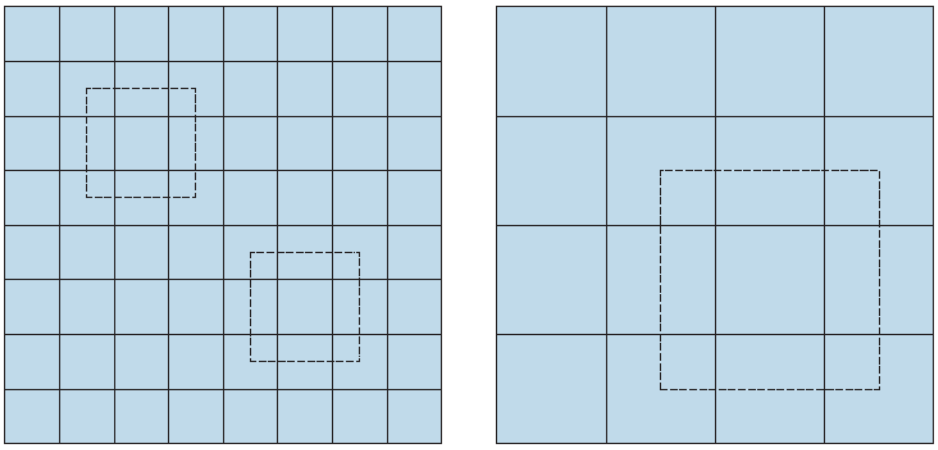


Figure 7.21 Lower-resolution feature maps detect larger-scale objects (right); higher-resolution feature maps detect smaller-scale objects (left).

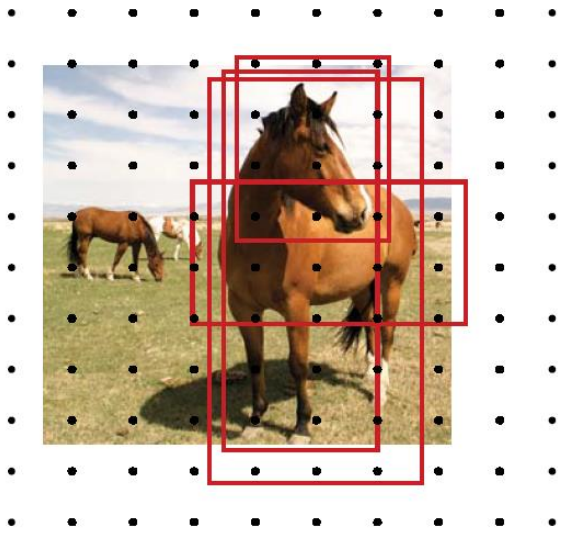


Figure 7.22 Multi-scale feature layers reduce the spatial dimensions of the input image to detect objects with different scales. In this image, you can see that the new priors are kind of zoomed out to cover more identifiable features of the horse close to the camera.

Non-maximum suppression

Given the large number of boxes generated by the detection layer per class during a forward pass of SSD at inference time, it is essential to prune most of the bounding box by applying the NMS technique. Boxes with a confidence loss and IoU less than a certain threshold are discarded, and only the top *N* predictions are kept (figure 7.24). This ensures that only the most likely predictions are retained by the network, while the noisier ones are removed.

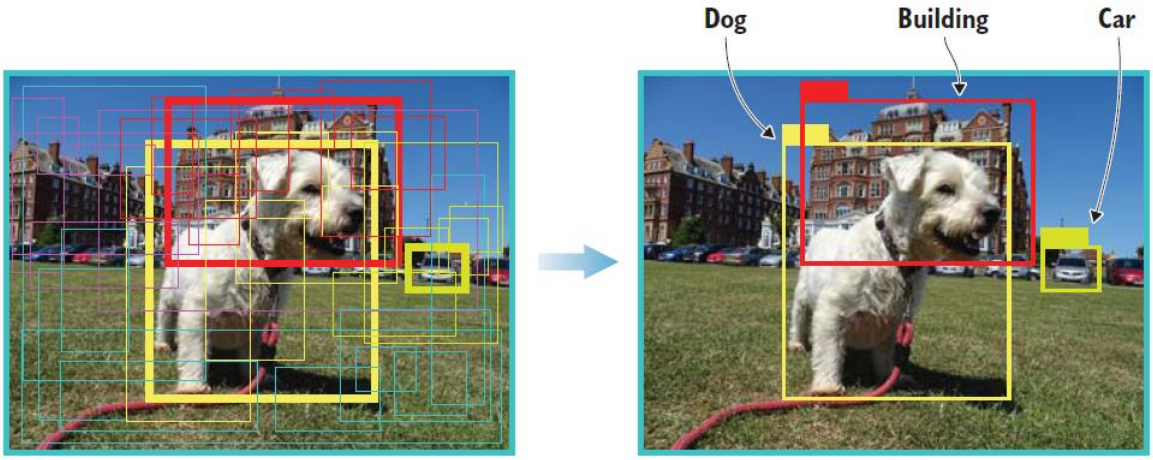


Figure 7.24 Non-maximum suppression reduces the number of bounding boxes to only one box for each object.

You only look once (YOLO)

Similar to the R-CNN family, YOLO is a family of object detection networks developed by Joseph Redmon et al. and improved over the years through the following versions:

- *YOLOv1*, published in 20169—Called "unified, real-time object detection" because it is a single-detection network that unifies the two components of a detector: object detector and class predictor.
- *YOLOv2* (also known as YOLO9000), published later in 201610—Capable of detecting over 9,000 objects; hence the name. It has been trained on ImageNet and COCO datasets and has achieved 16% mAP, which is not good; but it was very fast during test time.
- *YOLOv3*, published in 201811—Significantly larger than previous models and has achieved a mAP of 57.9%, which is the best result yet out of the YOLO family of object detectors.

The YOLO family is a series of end-to-end DL models designed for fast object detection, and it was among the first attempts to build a fast real-time object detector. It is one of the faster object detection algorithms out there. Although the accuracy of the models is close but not as good as R-CNNs, they are popular for object detection because of their detection speed, often demonstrated in real-time video or camera feed input.

The creators of YOLO took a different approach than the previous networks. YOLO does not undergo the region proposal step like R-CNNs. Instead, it only predicts over a limited number of bounding boxes by splitting the input into a grid of cells; each cell directly predicts a bounding box and object classification. The result is a large number of candidate bounding boxes that are consolidated into a final prediction using NMS (figure 7.25).

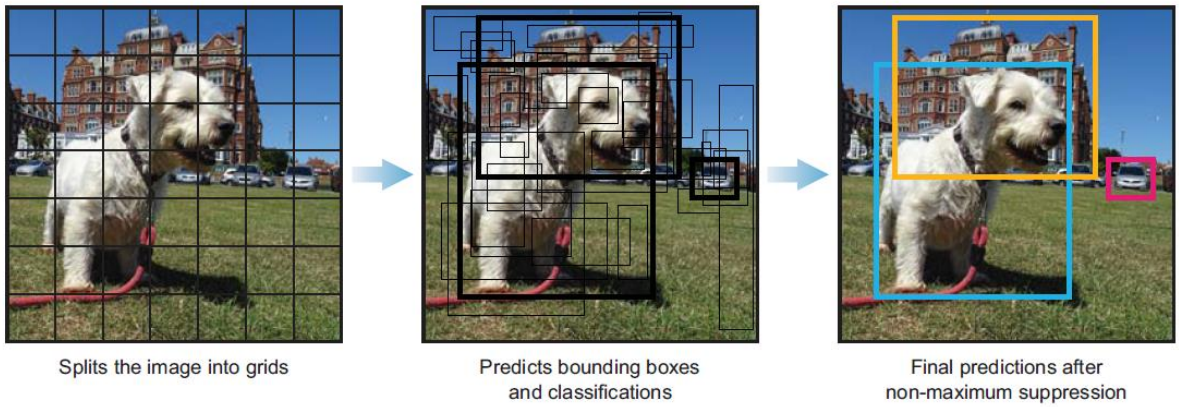


Figure 7.25 YOLO splits the image into grids, predicts objects for each grid, and then uses NMS to finalize predictions.

YOLOv1 proposed the general architecture, YOLOv2 refined the design and made use of predefined anchor boxes to improve bounding-box proposals, and YOLOv3 further refined the model architecture and training process. In this section, we are going to focus on YOLOv3 because it is currently the state-of-the-art architecture in the YOLO family.

How YOLOv3 works:

The YOLO network splits the input image into a grid of $S \times S$ cells. If the center of the ground-truth box falls into a cell, that cell is responsible for detecting the existence of that object. Each grid cell predicts B number of bounding boxes and their objectness score along with their class predictions, as follows:

 Coordinates of B bounding boxes—Similar to previous detectors, YOLO predicts four coordinates for each bounding box (*bx*, *by*, *bw*, *bh*), where *x* and *y* are set to be offsets of a cell location.

 Objectness score (*P*0)—indicates the probability that the cell contains an object. The objectness score is passed through a sigmoid function to be treated as a probability with a value range between 0 and 1. The objectness score is calculated as follows:

*P*0 = Pr (containing an object) × IoU (pred, truth)

 Class prediction—If the bounding box contains an object, the network predicts the probability of *K* number of classes, where *K* is the total number of classes in your problem.

It is important to note that before v3, YOLO used a softmax function for the class scores. In v3, Redmon et al. decided to use sigmoid instead. The reason is that softmax imposes the assumption that each box has exactly one class, which is often not the case. In other words, if an object belongs to one class, then it's guaranteed not to belong to another class. While this assumption is true for some datasets, it may not work when we have classes like Women and Person. A multilabel approach models the data more accurately.

As you can see in figure 7.26, for each bounding box (*B*), the prediction looks like this: [(*bounding box coordinates*), (*objectness score*), (*class predictions*)]. We've learned that the bounding box

coordinates are four values plus one value for the objectness score and K values for class predictions. Then the total number of values predicted for all bounding boxes is $5B + K$ multiplied by the number of cells in the grid $S \times S$: Total predicted values = $S \times S \times (5B + K)$

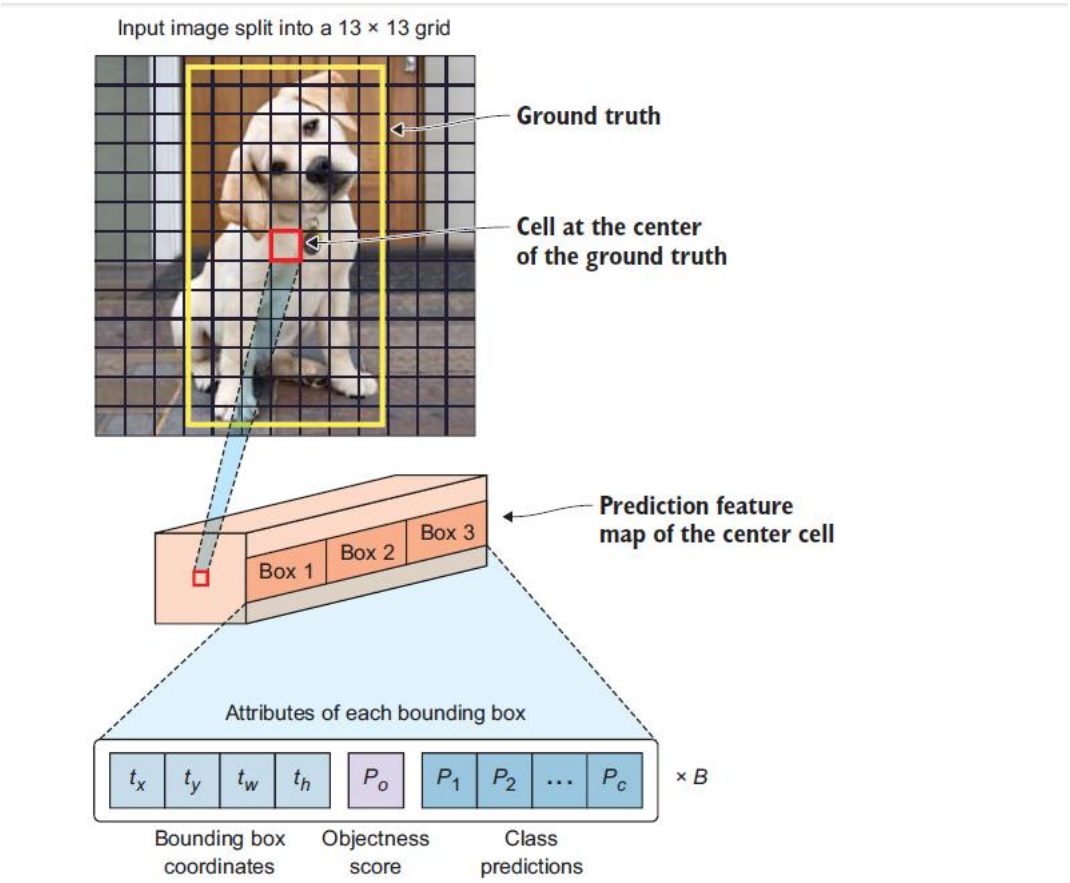


Figure 7.26 Example of a YOLOv3 workflow when applying a 13 × 13 grid to the input image. The input image is split into 169 cells. Each cell predicts *B* number of bounding boxes and their objectness score along with their class predictions. In this example, we show the cell at the center of the ground-truth making predictions for 3 boxes (*B* = 3). Each prediction has the following attributes: bounding box coordinates, objectness score, and class predictions.

PREDICTIONS ACROSS DIFFERENT SCALES

Look closely at figure 7.26. Notice that the prediction feature map has three boxes. You might have wondered why there are three boxes. Similar to the anchors concept in SSD, YOLOv3 has nine anchors to allow for prediction at three different scales per cell. The detection layer makes detections at feature maps of three different sizes having strides 32, 16, and 8, respectively. This means that with an input image of size 416 × 416, we make detections on scales 13 × 13, 26 × 26, and 52 × 52 (figure 7.27). The 13 × 13 layer is responsible for detecting large objects, the 26 × 26 layer is for detecting medium objects, and the 52 × 52 layer detects smaller objects.

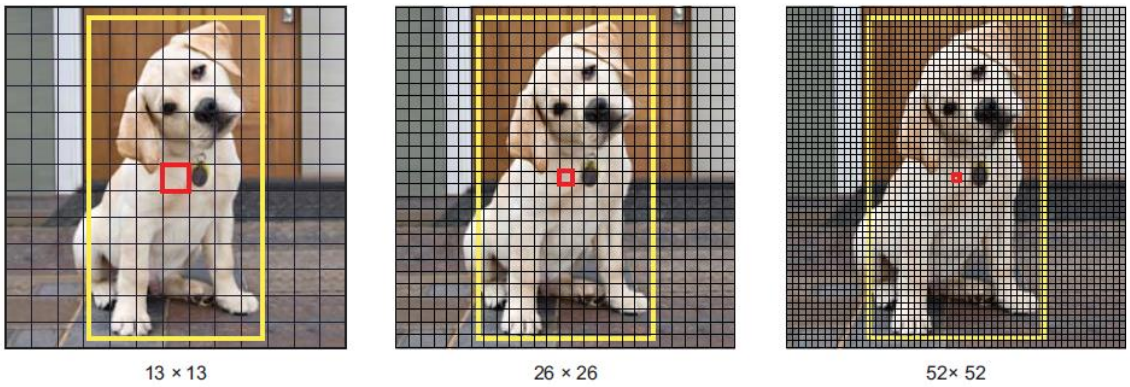


Figure 7.27 Prediction feature maps at different scales

YOLOV3 OUTPUT BOUNDING BOXES

For an input image of size 416 × 416, YOLO predicts ((52 × 52) + (26 × 26) + 13 × 13)) × 3 = 10,647 bounding boxes. That is a huge number of boxes for an output. In our dog example, we have only one object. We want only one bounding box around this object. How do we reduce the boxes from 10,647 down to 1? First, we filter the boxes based on their objectness score. Generally, boxes having scores below a threshold are ignored. Second, we use NMS to cure the problem of multiple detections of the same image. For example, all three bounding boxes of the outlined grid cell at the center of the image may detect a box, or the adjacent cells may detect the same object.

7.4.2 YOLOv3 architecture

Now that you understand how YOLO works, going through the architecture will be very simple and straightforward. YOLO is a single neural network that unifies object detection and classifications into one end-to-end network. The neural network architecture was inspired by the GoogLeNet model (Inception) for feature extraction. Instead of the Inception modules, YOLO uses 1 × 1 reduction layers followed by 3 × 3 convolutional layers. Redmon and Farhadi called this *DarkNet* (figure 7.28).

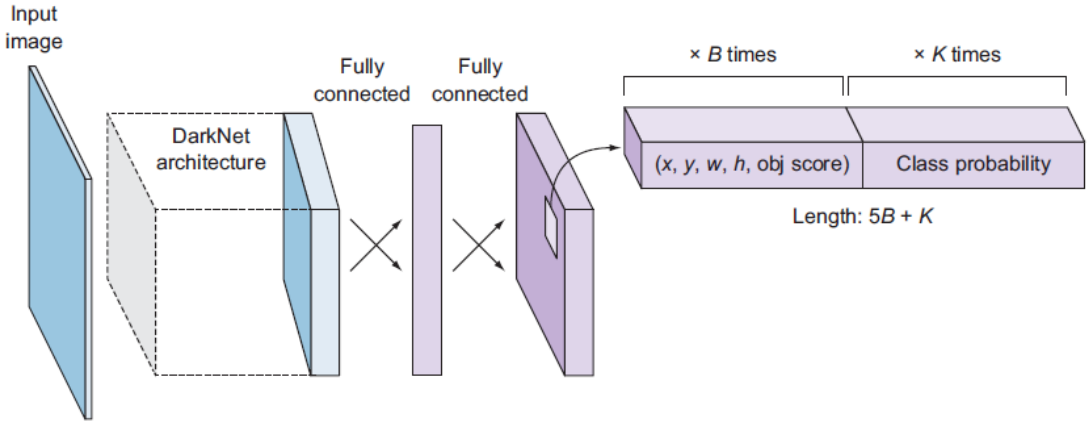


Figure 7.28 High-level architecture of YOLO

FULL ARCHITECTURE OF YOLOV3:

We just learned that YOLOv3 makes predictions across three different scales. This becomes a lot clearer when you see the full architecture, shown in figure 7.30. The input image goes through the DarkNet-53 feature extractor, and then the image is downsampled by the network until layer 79. The network branches out and continues to downsample the image until it makes its first prediction at layer 82. This detection is made on a grid scale of 13 × 13 that is responsible for detecting large objects, as we explained before.

Next the feature map from layer 79 is *upsampled* by 2x to dimensions 26 × 26 and *concatenated* with the feature map from layer 61. Then the second detection is made by layer 94 on a grid scale of 26 × 26 that is responsible for detecting medium objects. Finally, a similar procedure is followed again, and the feature map from layer 91 is subjected to few upsampling convolutional layers before being depth concatenated with a feature map from layer 36. A third prediction is made by layer 106 on a grid scale of 52 × 52, which is responsible for detecting small objects.

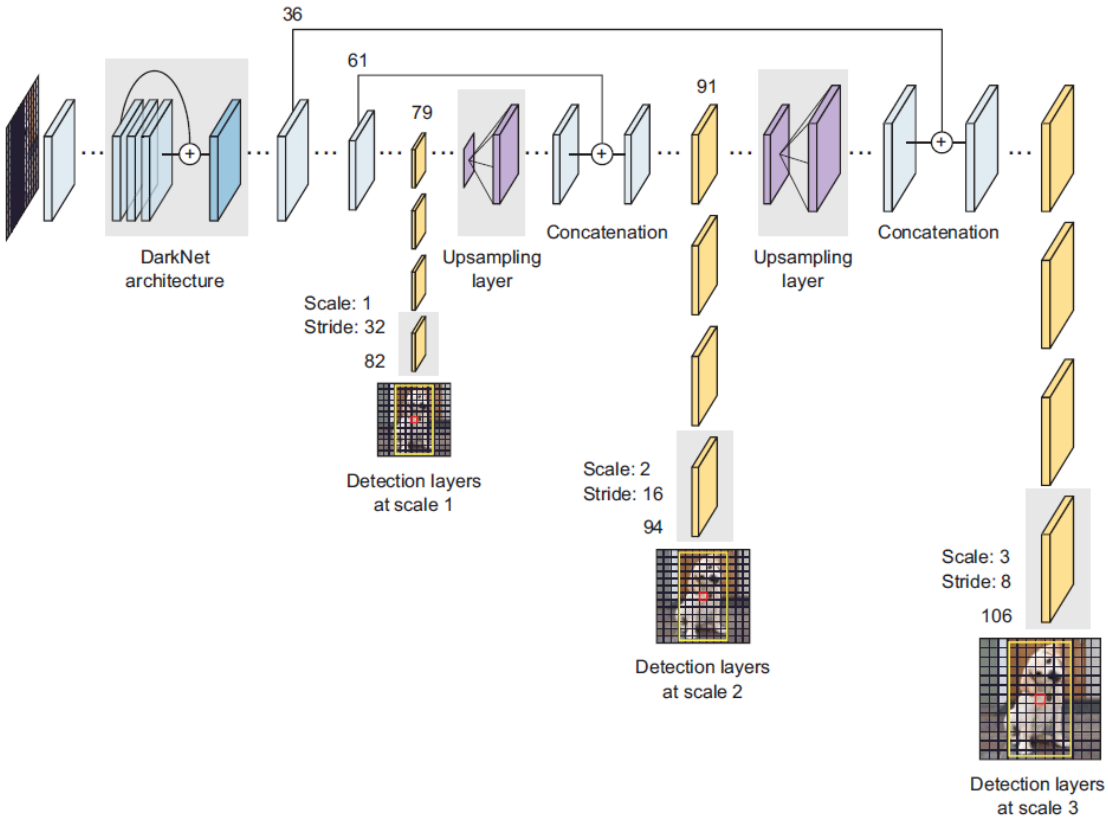


Figure 7.30 YOLOv3 network architecture. (Inspired by the diagram in Ayoosh Kathuria's post "What's new in YOLO v3?" *Medium*, 2018, http://mng.bz/lGN2.)

Segmentation:

Image Segmentation

Before we get to why U-Net is so popular when it comes to image segmentation tasks, let us understand what image segmentation is. Computer Vision has been one of the many exciting applications of machine intelligence. It has numerous applications in today's world and makes our lives easier. Two of the most common computer vision tasks are image classification and object detection.

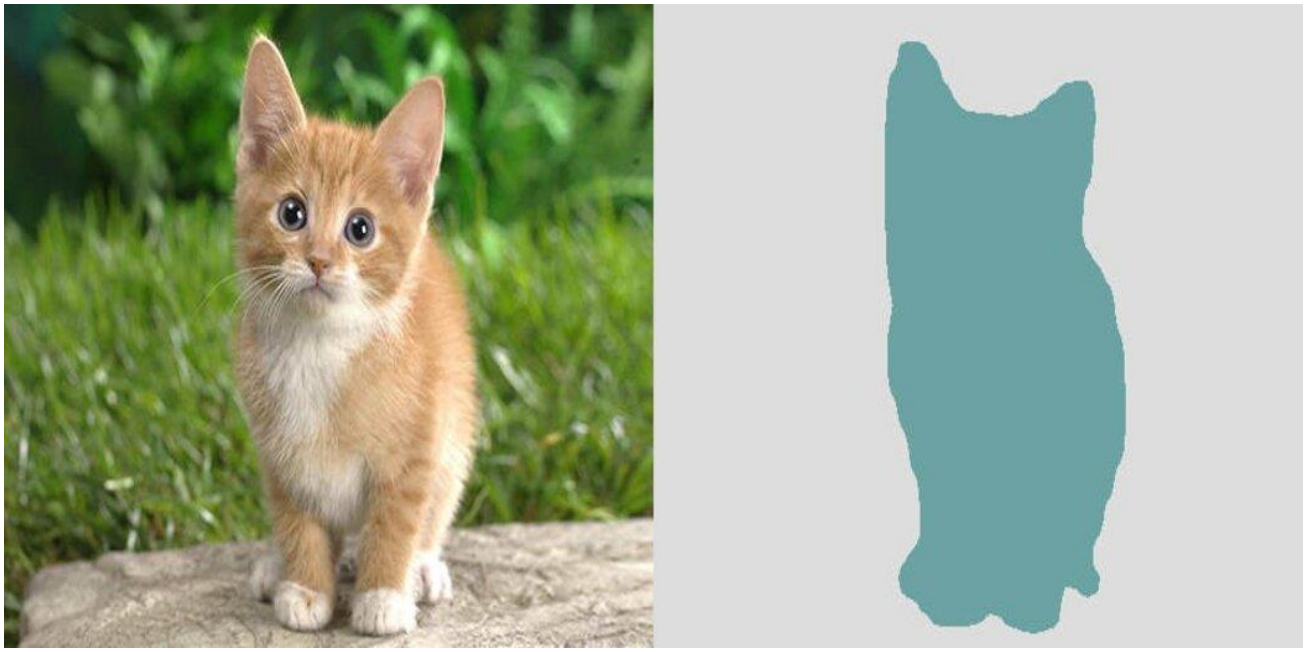
Image classification for two classes involves predicting whether the image belongs to class A or B. The predicted label is assigned to the entire image. Classification is helpful when we want to see what class is in the image.

Object detection, on the other hand, takes this further by predicting the object's location in our input image. We localize objects within an image by drawing a bounding box around them. Detection is useful for locating and tracking the contents of the image.

Understanding Image Segmentation

One could think of image segmentation as combining classification and localization.

Image segmentation involves partitioning the image into smaller parts called segments. Segmentation involves understanding what is given in an image at a pixel level. It provides fine-grained information about the image as well as the shapes and boundaries of the objects. The output of image segmentation is a mask where each element indicates which class that pixel belongs to. Let's understand this with an example.



1. Segmentation

What is Image Segmentation?

Image segmentation is a technique used in digital image processing to divide an image into regions or parts. It can be used to separate the background from the foreground, or to group pixels based on similar characteristics like color or shape.

Types of Image Segmentation

The high level categorization of image segmentation techniques are based on the nature of the segmentation. The main types of Image Segmentation are:

1. Semantic Segmentation
2. Instance Segmentation
3. Panoptic Segmentation

What is Semantic Segmentation?

Semantic segmentation is a foundational technique in computer vision that focuses on classifying each pixel in an image into specific categories or classes, such as objects, parts of objects, or background regions. Unlike instance segmentation, which differentiates between individual object instances, semantic segmentation provides a holistic understanding of the image by segmenting it into meaningful semantic regions based on the content and context of the scene.

- **Definition**: Assigns a class label to each pixel in the image. Every pixel belonging to the same object category is labeled the same, without distinguishing between individual objects of the same type.
- **Use Case**: Autonomous vehicles (e.g., identifying roads, cars, pedestrians), medical imaging (e.g., segmenting organs or tumors).
- **Output**: A single mask where each pixel corresponds to a specific class.

Workflow of Semantic Segmentation

1. **Data Analysis**: Analyze labeled training data to understand object classes and segmentation patterns.
2. **Network Design**: Create a semantic segmentation network with convolutional layers for feature extraction, contextual information integration, and upsampling layers for dense classification.
3. **Training**: Train the network using the annotated dataset to learn pixel-wise classification and optimize segmentation accuracy using loss functions like cross-entropy or Dice loss.
4. **Inference**: Deploy the trained model to process unseen images and generate segmentation masks by classifying each pixel into specific semantic categories.

Some of the Semantic Segmentation techniques are U-Net, FCN (Fully Convolutional Networks), DeepLab, PSPNet (Pyramid Scene Parsing Network) and SegNet.

Applications of Semantic Segmentation

- **Scene Understanding**: Semantic segmentation aids in understanding the content and context of complex scenes by identifying and categorizing various objects and regions within an image.
- **Autonomous Driving**: In autonomous vehicles, semantic segmentation enables scene perception by detecting and classifying objects like roads, pedestrians, vehicles, and obstacles to navigate safely.
- **Medical Image Analysis**: Semantic segmentation is crucial in medical imaging for identifying and segmenting anatomical structures or abnormalities, assisting in diagnosis and treatment planning.
- **Video Surveillance**: In video analytics systems, semantic segmentation facilitates object detection and tracking by segmenting and analyzing the motion and behavior of objects over time.

- **Image Editing and Augmentation**: Semantic segmentation powers advanced image editing and augmentation techniques by enabling precise selection and manipulation of specific objects or regions in the image.

Instance Segmentation

Instance segmentation is an advanced image analysis technique that combines elements of object detection and semantic segmentation to identify and delineate individual object instances within an image at a detailed pixel level. Unlike semantic segmentation, which classifies each pixel into broad categories without distinguishing between different instances of the same class, instance segmentation provides a more granular understanding by differentiating between individual objects and assigning a unique label to each object instance.

- **Definition**: Not only identifies the object category for each pixel but also differentiates between individual instances of objects.
- **Use Case**: Applications where distinguishing between objects of the same class is critical, such as counting the number of people in a crowd or objects in a warehouse.
- **Output**: Multiple masks, each representing an individual object instance, even if they belong to the same class.

Workflow of Instance Segmentation

1. **Object Detection**: The algorithm processes the input image and identifies potential objects by predicting bounding boxes and object classifications.
2. **Bounding Box Refinement**: Post-processing techniques may be employed to refine the predicted bounding boxes, ensuring accurate localization of object instances.
3. **Semantic Segmentation**: Within each refined bounding box, a semantic segmentation model segments the pixels to differentiate the object instance from its background, producing a segmentation mask for each object.
4. **Instance Labeling**: Finally, each segmented object instance is assigned a unique label, and the corresponding segmentation masks are combined to generate a comprehensive instance segmentation map for the entire image.

Some of the instance based segmentation techniques are Mask R-CNN, Faster R-CNN with Mask Branch, Cascade Mask R-CNN, SOLO (Segmenting Objects by Locations) and YOLACT (You Only Look At CoefficienTs).

Applications of instance segmentation

- **Object Detection and Recognition**: Instance segmentation facilitates accurate object detection, recognition, and classification in complex scenes with multiple overlapping objects.
- **Scene Understanding**: By providing detailed object-level segmentation, instance segmentation enhances scene understanding and context-aware image analysis.
- **Medical Imaging**: Instance segmentation aids in identifying and delineating specific anatomical structures or abnormalities in medical images for diagnosis and treatment planning.
- **Robotics and Autonomous Systems**: Instance segmentation is crucial for robotic vision systems and autonomous vehicles to perceive and interact with the surrounding environment effectively.

Panoptic Segmentation

- **Definition**: Combines both semantic and instance segmentation. It assigns a class label and distinguishes individual instances for countable objects while also labeling background and amorphous areas (e.g., sky, road).

- **Use Case**: Advanced applications like robotics, autonomous navigation, or urban planning, where a comprehensive understanding of the scene is necessary.
- **Output**: A single unified segmentation map that provides instance-level differentiation for objects and semantic labels for amorphous regions.

Original Image Semantic Segmentation Instance Segmentation

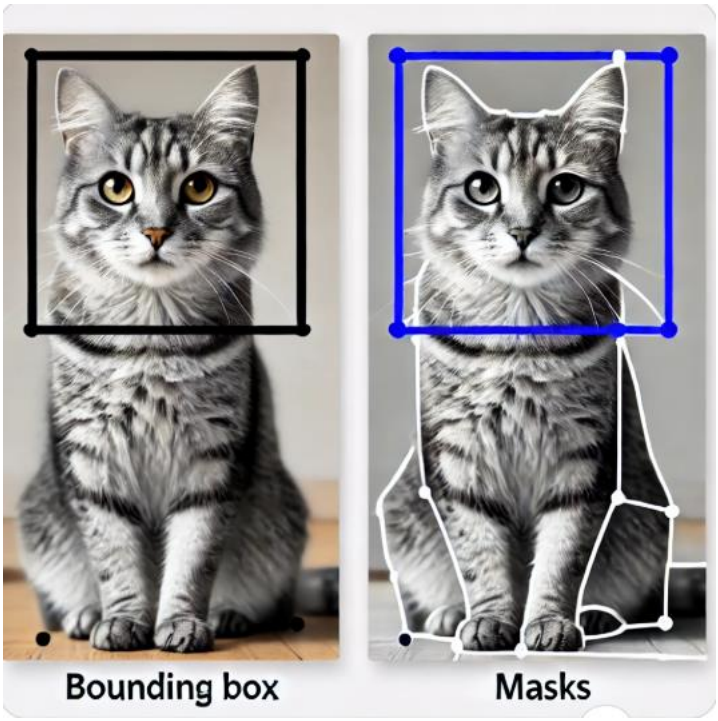
Mask R-CNN

In the context of **Mask R-CNN** and **instance segmentation**, a **mask** refers to a **binary image** or a matrix that represents the pixel-wise segmentation of an object in the image.

Key Details:

1. **Binary Representation:**
   - A mask is a 2D array (of the same height and width as the input image region).
   - Each element in the mask corresponds to a pixel in the image, where:
     - 1 (or white) indicates that the pixel belongs to the object.
     - 0 (or black) indicates that the pixel does not belong to the object.
2. **Per-Instance Mask:**
   - For **instance segmentation**, a separate mask is generated for each individual object instance in the image, even for objects belonging to the same class.
   - Example: In an image with two cars, Mask R-CNN will create two masks—one for each car.
3. **Use in Segmentation:**
   - Masks are used to precisely delineate object boundaries and distinguish overlapping objects.
   - Unlike bounding boxes (which provide a rectangular outline), masks allow for pixel-perfect segmentation of objects.

A **bounding box** around a cat would look like a rectangle roughly enclosing the cat.
A **mask** of the same cat would highlight the exact shape of the cat, including the outline of its body, legs, and tail.



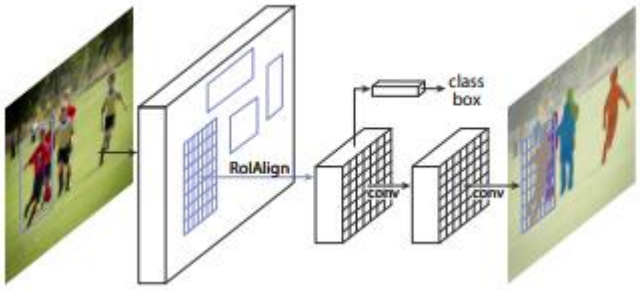
Mask R-CNN, or Mask Region-based Convolutional Neural Network, is a deep learning model that performs image segmentation and object detection:

What is Mask R-CNN?

Mask R-CNN (Mask Region-based Convolutional Neural Network) is an extension of the Faster R-CNN architecture that adds a branch for predicting segmentation masks on top of the existing object detection capabilities. It was introduced to address the task of instance segmentation, where the goal is not only to detect objects in an image but also to precisely segment the pixels corresponding to each object instance.

Mask R-CNN Architecture

Mask R-CNN was proposed by Kaiming He et al. in 2017. It is very similar to Faster R-CNN except there is another layer to predict segmented. The stage of region proposal generation is the same in both the architecture the second stage which works in parallel predicts the class generates a bounding box as well as outputs a binary mask for each RoI.



Mask R-CNN Architecture

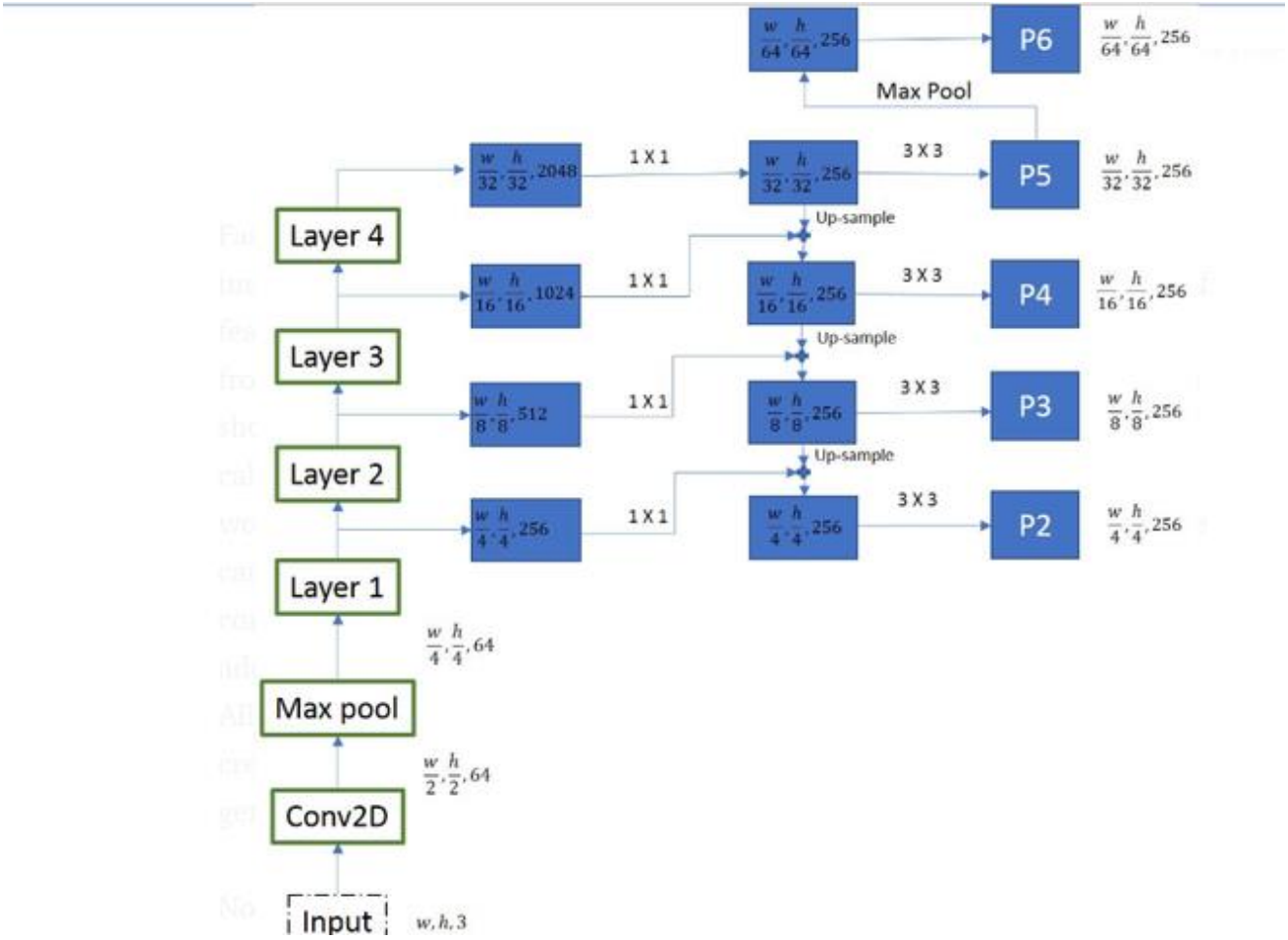
It comprises of –

- Backbone Network
- Region Proposal Network
- Mask Representation
- RoI Align

Backbone Network

The authors of Mask R-CNN experimented with two kinds of backbone networks. The first is standard ResNet architecture (ResNet-C4) and another is ResNet with a feature pyramid network. The standard ResNet architecture was similar to that of Faster R-CNN but the

ResNet-FPN has proposed some modification. This consists of a multi-layer RoI generation. This multi-layer feature pyramid network generates RoI of different scale which improves the accuracy of previous ResNet architecture.

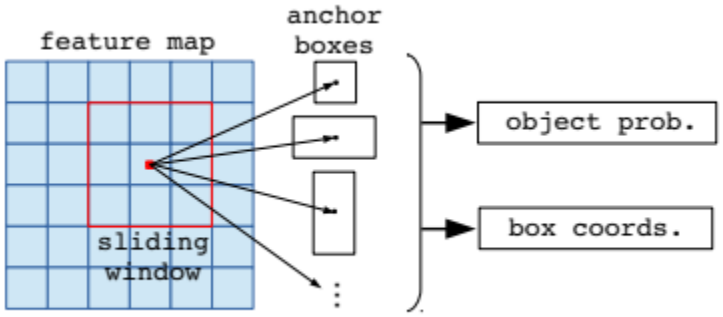


Mask R-CNN backbone architecture

At every layer, the feature map size is reduced by half and the number of feature maps is doubled. We took output from four layers *(layers – 1, 2, 3, and 4)*. To generate final feature maps, we use an approach called the top-bottom pathway. We start from the top feature map*(w/32, h/32, 256)* and work our way down to bigger ones, by upscale operations. Before sampling, we also apply the *1*1* convolution to bring down the number of channels to *256*. This is then added element-wise to the up-sampled output from the previous iteration. All the outputs are subjected to *3 X 3* convolution layers to create the final *4 feature maps(P2, P3, P4, P5)*. The *5th* feature map *(P6)* is generated from a max pooling operation from *P5*.

Region Proposal Network

All the convolution feature map that is generated by the previous layer is passed through a *3*3* convolution layer. The output of this is then passed into two parallel branches that determine the objectness score and regress the bounding box coordinates.



Anchor Generation Mask R-CNN

Here, we only use only one anchor stride and *3* anchor ratios for a feature pyramid (because we already have feature maps of different sizes to check for objects of different sizes).

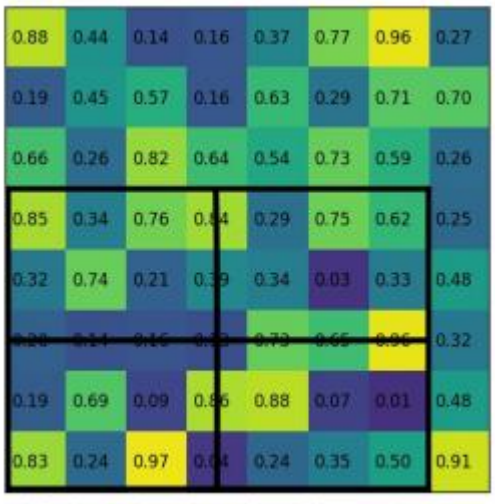
Mask Representation

A mask contains spatial information about the object. Thus, unlike the classification and bounding box regression layers, we could not collapse the output to a fully connected layer to improve since it requires pixel-to-pixel correspondence from the above layer. Mask R-CNN uses a fully connected network to predict the mask. This ConvNet takes an RoI as input and outputs the *m*m* mask representation. We also upscale this mask for inference on the input image and reduce the channels to *256* using *1*1* convolution. In order to generate input for this fully connected network that predicts mask, we use RoIAlign. The purpose of RoIAlign is to use convert different-size feature maps generated by the region proposal network into a fixed-size feature map. Mask R-CNN paper suggested two variants of architecture. In one variant, the input of mask generation CNN is passed after RoIAlign is applied (ResNet C4), but in another variant, the input is passed just before the fully connected layer (FPN Network).

This mask generation branch is a full convolution network and it output a *K * (m*m)*, where *K* is the number of classes (one for each class) and *m=14* for *ResNet-C4 and 28 for ResNet_FPN*.

RoI Align

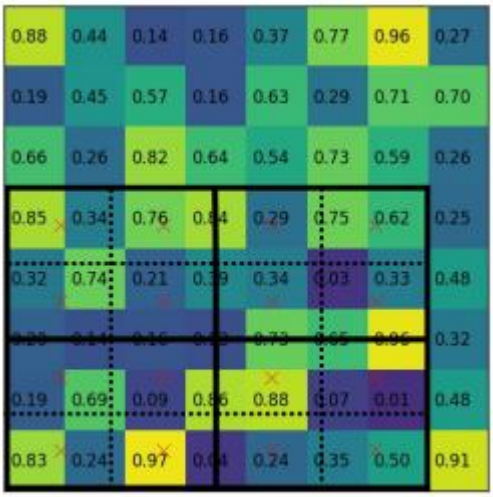
RoI align has the same motive as of RoI pool, to generate the fixed size regions of interest from region proposals. It works in the following steps:



Region projection and pooling sections

ROI Align

Given the feature map of the previous Convolution layer of size *h*w*, divide this feature map into *M * N* grids of equal size (we will NOT just take integer value).



The mask R-CNN inference speed is around *2 fps*, which is good considering the addition of a segmentation branch in the architecture.

U-Net and Semantic Segmentation

Semantic segmentation is a technique that enables us to associate each pixel of a digital image with a class label, such as trees, signboards, pedestrians, roads, buildings, cars, sky, etc. It is also considered an **image classification** task at a pixel level as it involves differentiating between objects in an image.

Semantic segmentation focuses on assigning a class label to each pixel in the image without distinguishing between individual instances of the same class.

Popular Models:

1. U-Net
   - Designed for biomedical segmentation but widely used for other domain
   - Features an encoder-decoder architecture with skip connection to preserve spatial information.
2. Fully Convolutional Networks
   - Replaces FCL with CL to produce pixel wise prediction.
   - Backbone: Pretrained model like VGG

U-Net

U-Net is a deep learning model that uses a convolutional neural network (CNN) to segment images. It was developed by Olaf Ronneberger's team in 2015 to address the challenge of working with limited training data, which is common in the medical field.

U-Net's architecture is a U-shaped encoder/decoder structure that incorporates encoder and decoder paths. It's used for a variety of tasks, including:

- **Cell detection**: U-Net can be used to detect cells in biomedical image data.
- **Shape measurements**: U-Net can be used to measure the shape of cells in biomedical image data.
- **Brain tumor segmentation**: U-Net can be used to predict the location of brain tumors.

Some advantages of U-Net include: It can work with limited training data and It can produce precise segmentation.

However, U-Net has some disadvantages, including:

It has many parameters, which can make it more prone to overfitting, especially when working with small datasets.

U-Net is a widely used deep learning architecture that was first introduced in the "U-Net: Convolutional Networks for Biomedical Image Segmentation" paper. The primary purpose of this architecture was to address the challenge of limited annotated data in the medical field. This network was designed to effectively leverage a smaller amount of data while maintaining speed and accuracy.

U-Net Architecture:

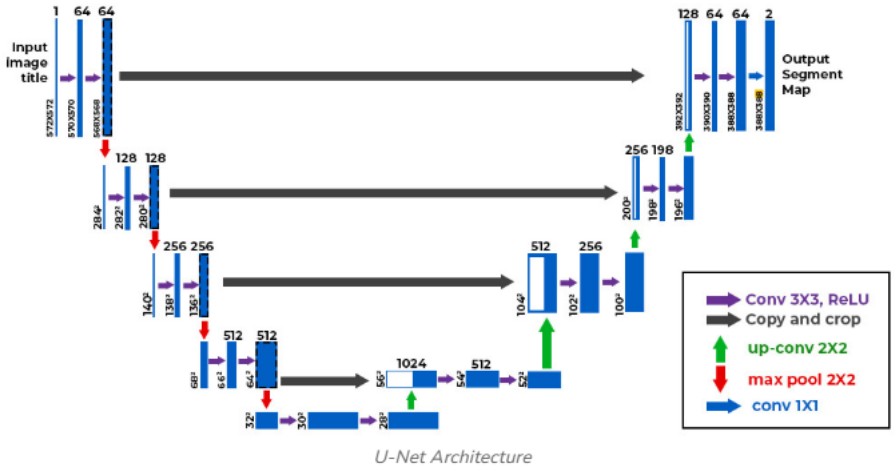
Understanding the Development and Advantages of U-Net:

Olaf Ronneberger and his team developed u net segmentation in 2015 for their work on biomedical images.

Sliding window architecture performs localization tasks well on any given training dataset. This

architecture creates a local patch for each pixel, generating separate class labels for each pixel. However, this architecture has two main drawbacks: firstly, it generates a lot of overall redundancy due to overlapping patches. Secondly, the training procedure was slow, taking a lot of time and resources. These reasons made the architecture not feasible for various tasks. U-Net architecture for image segmentation overcomes these two drawbacks.

The architecture of U-Net is unique in that it consists of a contracting path and an expansive path. The contracting path contains encoder layers that capture contextual information and reduce the spatial resolution of the input, while the expansive path contains decoder layers that decode the encoded data and use the information from the contracting path via skip connections to generate a segmentation map.



U-Net Architecture

The contracting path in U-Net is responsible for identifying the relevant features in the input image. The encoder layers perform convolutional operations that reduce the spatial resolution of the feature maps while increasing their depth, thereby capturing increasingly abstract representations of the input. This contracting path is similar to the feedforward layers in other convolutional neural networks. On the other hand, the expansive path works on decoding the encoded data and locating the features while maintaining the spatial resolution of the input. The decoder layers in the expansive path upsample the feature maps, while also performing convolutional operations. The skip connections from the contracting path help to preserve the spatial information lost in the contracting path, which helps the decoder layers to locate the features more accurately.

Figure 1 illustrates how the U-Net network converts a grayscale input image of size 572×572×1 into a binary segmented output map of size 388×388×2. We can notice that the output size is smaller than the input size because no padding is being used. However, if we use padding, we can maintain the input size. During the contracting path, the input image is progressively reduced in height and width but increased in the number of channels. This increase in channels allows the network to capture high-level features as it progresses down the path. At the bottleneck, a final convolution operation is performed to generate a 30×30×1024 shaped feature map. The expansive path then takes the feature map from the bottleneck and converts it back into an image of the same size as the original input. This is done using upsampling layers, which increase the spatial resolution of the feature map while reducing the number of channels. The skip connections from the contracting path are used to help the decoder layers locate and refine the features in the image. Finally, each pixel in the output image represents a label that corresponds to a particular object or class in the input image. In this case, the

output map is a binary segmentation map where each pixel represents a foreground or background region.

The encoder network is also called the contracting network. This network learns a feature map of the input image and tries to solve our first question- "what" is in the image? It is similar to any classification task we perform with convolutional neural networks except for the fact that in a U-Net, we do not have any fully connected layers in the end, as the output we require now is not the class label but a mask of the same size as our input image.

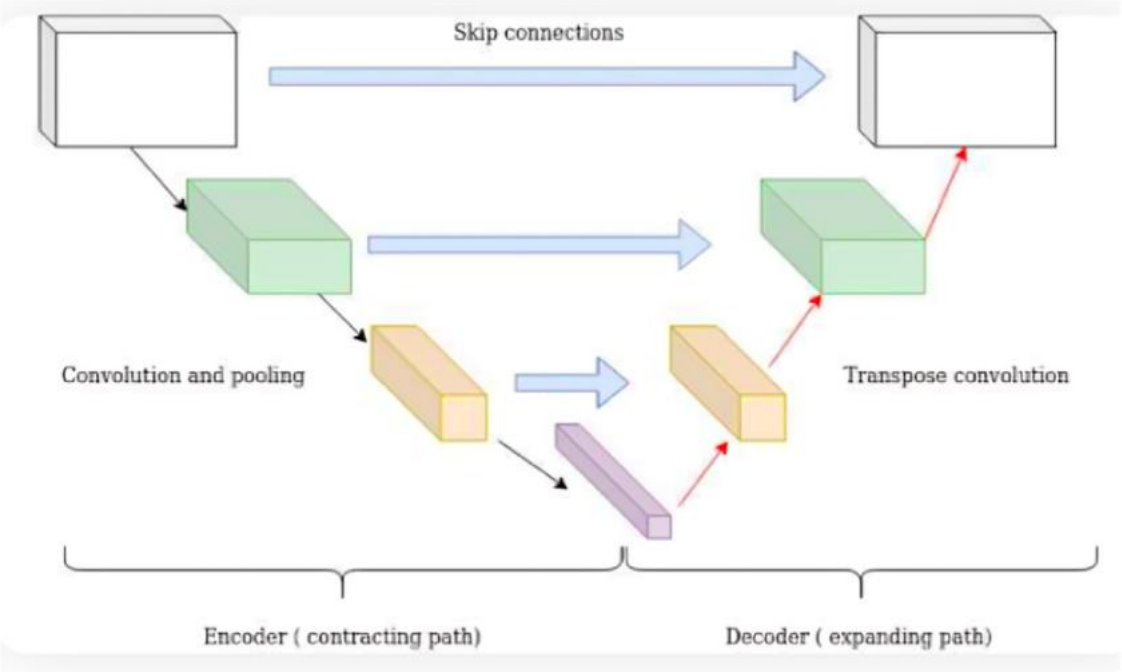
This encoder network consists of 4 encoder blocks. Each block contains two convolutional layers with a kernel size of 3*3 and valid padding, followed by a Relu activation function. This is inputted to a max pooling layer with a kernel size of 2*2. With the max pooling layer, we have halved the spatial dimensions learned, thereby reducing the computation cost of training the model.

In between the encoder and decoder network, we have the bottleneck layer. This is the bottommost layer, as we can see in the model above. It consists of 2 convolutional layers followed by Relu. The output of the bottleneck is the **final feature map representation**.

Mechanisms of Skip Connections and Decoder Networks:

The **decoder network** is also called the expansive network. Our idea is to up sample our feature maps to the size of our input image. This network takes the feature map from the bottleneck layer and generates a segmentation mask with the help of skip connections. The decoder network tries to solve our second question-"where" is the object in the image? It consists of 4 decoder blocks. Each block starts with a transpose convolution (indicated as up-conv in the diagram) with a kernel size of 2*2. After concatenating this output with the corresponding skip layer connection from the encoder block, the network utilizes two convolutional layers with a kernel size of 3*3, followed by a Relu activation function.

Skip connections are indicated with a grey arrow in the model architecture. Skip connections help us use the contextual feature information collected in the encoder blocks to generate our segmentation map. The idea is to use our high-resolution features learned from the encoder blocks (through skip connections) to help us project our feature map (output of the bottleneck layer). This helps us answer "where" is our object in the image?



A 1*1 convolution follows the last decoder block with sigmoid activation which gives the output of a segmentation mask containing pixel-wise classification. This way, it could be said that the contracting path passes across information to the expansive path. And thus, we can capture both the feature information and localization with the help of a U-Net.

Variational Autoencoder

What is a Variational Autoencoder?

Variational autoencoder was proposed in 2013 by Diederik P. Kingma and Max Welling at Google and Qualcomm. A variational autoencoder (VAE) provides a probabilistic manner for describing an observation in latent space. Thus, rather than building an encoder that outputs a single value to describe each latent state attribute, we'll formulate our encoder to describe a probability distribution for each latent attribute. It has many applications, such as data compression, synthetic data creation, etc.

Variational autoencoder is different from an autoencoder in a way that it provides a statistical manner for describing the samples of the dataset in latent space. Therefore, in the variational autoencoder, the encoder outputs a probability distribution in the bottleneck layer instead of a single output value.

- **Standard Autoencoder:** Deterministic, learns a compressed representation, good for reconstruction but poor for generation.
- **Variational Autoencoder:** Probabilistic, learns a structured latent space, good for both reconstruction and generating new data.

Variational autoencoders introduce a probabilistic interpretation in the latent space, allowing for the generation of diverse outputs by sampling from learned distributions. This contrasts with standard autoencoders, which use a deterministic mapping in the latent space.

Architecture of Variational Autoencoder

- The encoder-decoder architecture lies at the heart of Variational Autoencoders (VAEs), distinguishing them from traditional autoencoders. The encoder network takes raw input data and transforms it into a probability distribution within the latent space.

- The latent code generated by the encoder is a probabilistic encoding, allowing the VAE to express not just a single point in the latent space but a distribution of potential representations.

- The decoder network, in turn, takes a sampled point from the latent distribution and reconstructs it back into data space. During training, the model refines both the encoder and decoder parameters to minimize the reconstruction loss – the disparity between the input data and the decoded output. The goal is not just to achieve accurate reconstruction but also to regularize the latent space, ensuring that it conforms to a specified distribution.

- The process involves a delicate balance between two essential components: the reconstruction loss and the regularization term, often represented by the Kullback-Leibler divergence. The reconstruction loss compels the model to accurately reconstruct the input, while the regularization term encourages the latent space to adhere to the chosen distribution, preventing overfitting and promoting generalization.

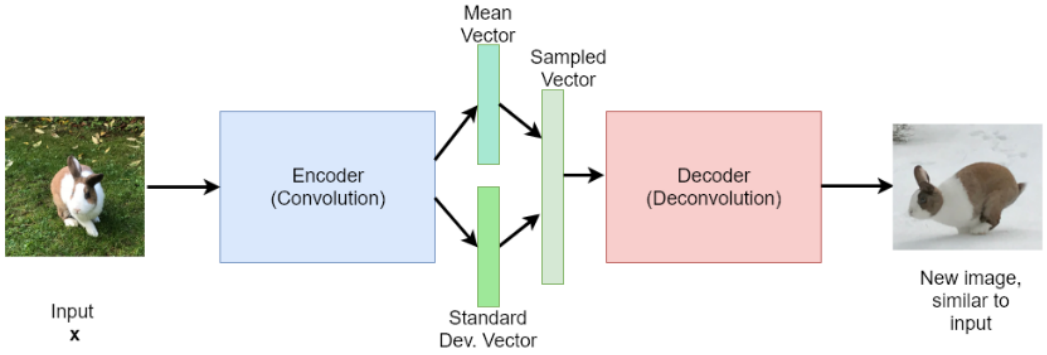
- By iteratively adjusting these parameters during training, the VAE learns to encode input data into a meaningful latent space representation. This optimized latent code encapsulates the underlying features and structures of the data, facilitating precise reconstruction. The probabilistic nature of the latent space also enables the generation of novel samples by

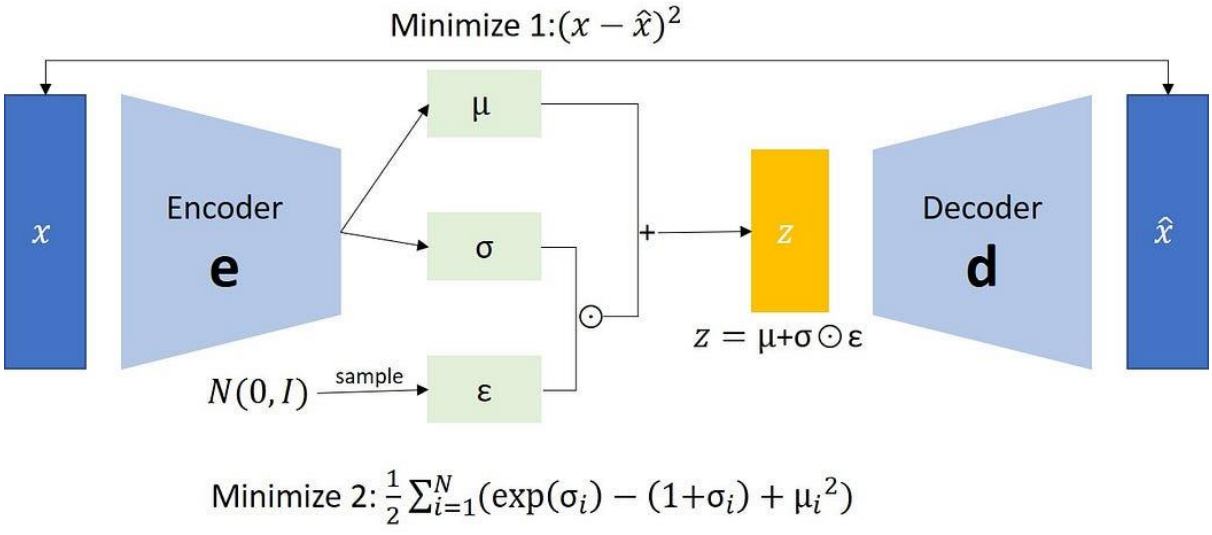
drawing random points from the learned distribution.

In general, if the probability distribution of one or multiple random variable(s) is known, then new samples, which arise from this distribution can be generated. For example, if the parameters *mean* and *standard-deviation* of a Gaussian-distributed random variable are known, then new samples can be generated, which correspond to this distribution.

Based on this concept, a standard autoencoder can be modified to a generative model - the Variational Autoencoder - as follows: Instead of mapping the input x to a latent representation y, the Encoder model maps the input to a probability distribution. More concrete: A type of probability distribution (e.g. a Gaussian Normal Distribution) is presumed, and the output of the encoder are the concrete parameters of this distribution (e.g. *mean* and *standard deviation* for Gaussian distribution type). Then a concrete sample is generated from this distribution. This sample constitutes the input to the decoder.

The weights of the encoder and decoder-module are learned such that the output of the decoder is as close as possible to input - like in the case of an autoencoder, but now with the new process in the latent layer: parameter estimation and sampling.





Minimize 2: $\frac{1}{2}\sum_{i=1}^{N}(\exp(\sigma_i)-(1+\sigma_i)+{\mu_i}^2)$

Loss Functions:

The VAE minimizes two types of loss to train the model:

1. **Reconstruction Loss:**
   - Measures how well the decoder reconstructs the input x from the latent variable z.
   - A common choice for this loss is the Mean Squared Error (MSE):
   $$L_{\text{reconstruction}} = ||x - \hat{x}||^2$$
2. **KL Divergence Loss:**
   - Encourages the learned latent distribution $q(z|x)$ (parameterized by μ and σ) to be close to a standard normal prior $p(z) = N(0, I)$.
   - The KL divergence loss is:
   $$L_{\text{KL}} = \frac{1}{2} \sum_{i=1}^{N} \left(\exp(\sigma_i) - 1 - \sigma_i + \mu_i^2 \right)$$

$$z = \mu + \sigma \odot \varepsilon$$

Purpose of the Equation:

This equation is used to sample a latent variable z in a way that makes the backpropagation process compatible with the stochastic nature of VAEs.

Reparameterization Trick:

- In a VAE, the latent space is modeled as a Gaussian distribution $q(z|x) = N(\mu, \sigma^2)$. However, sampling directly from this distribution makes the process non-differentiable, which breaks gradient-based optimization.
- The **reparameterization trick** rewrites the sampling as:
  $$z = \mu + \sigma \odot \epsilon$$
  where $\epsilon \sim N(0, I)$. This separates the randomness (ϵ) from the learnable parameters (μ and σ), enabling gradients to flow through μ and σ during training.

Implementing Variational Autoencoder

In this implementation, we will be using the Fashion-MNIST dataset, this dataset is already available in keras.datasets API, so we don't need to add or upload manually. You can also find the implementation in the from an.

1. **Importing Libraries**

- First, we need to import the necessary packages to our python environment. we will be using

Keras package with TensorFlow as a backend.

```
import numpy as np
import tensorflow as tf
import keras
from keras import layers
```

2. Creating a Sampling Layer

- For variational autoencoders, we need to define the architecture of two parts encoder and decoder but first, we will define the bottleneck layer of architecture, the sampling layer.

```
# this sampling layer is the bottleneck layer of variational autoencoder,
# it uses the output from two dense layers z_mean and z_log_var as input,
# convert them into normal distribution and pass them to the decoder layer

class Sampling(layers.Layer):
    """Uses (mean, log_var) to sample z, the vector encoding a digit."""

    def call(self, inputs):
        mean, log_var = inputs
        batch = tf.shape(mean)[0]
        dim = tf.shape(mean)[1]
        epsilon = tf.random.normal(shape=(batch, dim))
        return mean + tf.exp(0.5 * log_var) * epsilon
```

3. Define Encoder Block

- Now, we define the architecture of encoder part of our autoencoder, this part takes images as input and encodes their representation in the Sampling layer.

```
latent_dim = 2

encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(64, 3, activation="relu", strides=2,
padding="same")(encoder_inputs)
x = layers.Conv2D(128, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
mean = layers.Dense(latent_dim, name="mean")(x)
log_var = layers.Dense(latent_dim, name="log_var")(x)
z = Sampling()([mean, log_var])
encoder = keras.Model(encoder_inputs, [mean, log_var, z], name="encoder")
encoder.summary()
```

Output:

Model: "encoder"

Layer (type)	Output Shape	Param #	Connected to

```
==================================================================================================
input_9 (InputLayer)        [(None, 28, 28, 1)]       0         []
conv2d_8 (Conv2D)           (None, 14, 14, 64)        640       ['input_9[0][0]']
conv2d_9 (Conv2D)           (None, 7, 7, 128)         73856     ['conv2d_8[0][0]']
flatten_4 (Flatten)         (None, 6272)              0         ['conv2d_9[0][0]']
dense_8 (Dense)             (None, 16)                100368    ['flatten_4[0][0]']
mean (Dense)                (None, 2)                 34        ['dense_8[0][0]']
log_var (Dense)             (None, 2)                 34        ['dense_8[0][0]']
sampling_4 (Sampling)       (None, 2)                 0         ['mean[0][0]',
                                                                 'log_var[0][0]']
==================================================================================================
Total params: 174932 (683.33 KB)
Trainable params: 174932 (683.33 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

4. Define Decoder Block

- Now, we define the architecture of decoder part of our autoencoder, this part takes the output of the sampling layer as input and output an image of size (28, 28, 1) .

```
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(128, 3, activation="relu", strides=2,
padding="same")(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2,
padding="same")(x)
decoder_outputs = layers.Conv2DTranspose(1, 3, activation="sigmoid",
padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
decoder.summary()
```

Output:

```
Model: "decoder"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
input_10 (InputLayer)       [(None, 2)]               0
```

dense_9 (Dense)	(None, 3136)		9408
reshape_4 (Reshape)	(None, 7, 7, 64)		0
conv2d_transpose_12 (Conv2DTranspose)	(None, 14, 14, 128)		73856
conv2d_transpose_13 (Conv2DTranspose)	(None, 28, 28, 64)		73792
conv2d_transpose_14 (Conv2DTranspose)	(None, 28, 28, 1)		577

==========

Total params:	157633	(615.75	KB)
Trainable params:	157633	(615.75	KB)
Non-trainable params:	0	(0.00	Byte)

---

5. Define the VAE Model

- In this step, we combine the model and define the training procedure with loss functions.

```
class VAE(keras.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super().__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.total_loss_tracker = keras.metrics.Mean(name="total_loss")
        self.reconstruction_loss_tracker = keras.metrics.Mean(
            name="reconstruction_loss"
        )
        self.kl_loss_tracker = keras.metrics.Mean(name="kl_loss")

    @property
    def metrics(self):
        return [
            self.total_loss_tracker,
            self.reconstruction_loss_tracker,
            self.kl_loss_tracker,
        ]

    def train_step(self, data):
        with tf.GradientTape() as tape:
            mean,log_var, z = self.encoder(data)
            reconstruction = self.decoder(z)
            reconstruction_loss = tf.reduce_mean(
                tf.reduce_sum(
                    keras.losses.binary_crossentropy(data, reconstruction),
                    axis=(1, 2),
                )
```

```
            )
            kl_loss = -0.5 * (1 + log_var - tf.square(mean) -
tf.exp(log_var))
            kl_loss = tf.reduce_mean(tf.reduce_sum(kl_loss, axis=1))
            total_loss = reconstruction_loss + kl_loss
        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
        self.total_loss_tracker.update_state(total_loss)
        self.reconstruction_loss_tracker.update_state(reconstruction_loss)
        self.kl_loss_tracker.update_state(kl_loss)
        return {
            "loss": self.total_loss_tracker.result(),
            "reconstruction_loss":
self.reconstruction_loss_tracker.result(),
            "kl_loss": self.kl_loss_tracker.result(),
        }
```

6. Train the VAE

- Now it's the right time to train our variational autoencoder model, we will train it for 10 epochs. But first we need to import the fashion MNIST dataset.

```
(x_train, _), (x_test, _) = keras.datasets.fashion_mnist.load_data()
fashion_mnist = np.concatenate([x_train, x_test], axis=0)
fashion_mnist = np.expand_dims(fashion_mnist, -1).astype("float32") / 255

vae = VAE(encoder, decoder)
vae.compile(optimizer=keras.optimizers.Adam())
vae.fit(fashion_mnist, epochs=10, batch_size=128)
```

Output:

```
Epoch 1/10
547/547 [==============================] - 12s 15ms/step - loss: 279.0265 -
reconstruction_loss: 265.4565 - kl_loss: 7.4453
Epoch 2/10
547/547 [==============================] - 9s 16ms/step - loss: 270.6755 -
reconstruction_loss: 262.7192 - kl_loss: 7.3106
Epoch 3/10
547/547 [==============================] - 8s 15ms/step - loss: 268.7467 -
reconstruction_loss: 261.6213 - kl_loss: 7.2532
Epoch 4/10
547/547 [==============================] - 9s 17ms/step - loss: 268.4030 -
reconstruction_loss: 260.6729 - kl_loss: 7.1995
Epoch 5/10
547/547 [==============================] - 9s 16ms/step - loss: 267.5622 -
reconstruction_loss: 260.0038 - kl_loss: 7.1871
Epoch 6/10
547/547 [==============================] - 8s 15ms/step - loss: 266.9597 -
reconstruction_loss: 259.2417 - kl_loss: 7.1737
```

```
Epoch 7/10
547/547 [==============================] - 8s 15ms/step - loss: 266.0508 - reconstruction_loss: 258.7231 - kl_loss: 7.1072
Epoch 8/10
547/547 [==============================] - 8s 15ms/step - loss: 265.1775 - reconstruction_loss: 258.2050 - kl_loss: 7.0736
Epoch 9/10
547/547 [==============================] - 8s 16ms/step - loss: 264.4663 - reconstruction_loss: 257.8303 - kl_loss: 7.0655
Epoch 10/10
547/547 [==============================] - 8s 15ms/step - loss: 264.6552 - reconstruction_loss: 257.3342 - kl_loss: 7.0278
<keras.src.callbacks.History at 0x785cc25dd540>
```

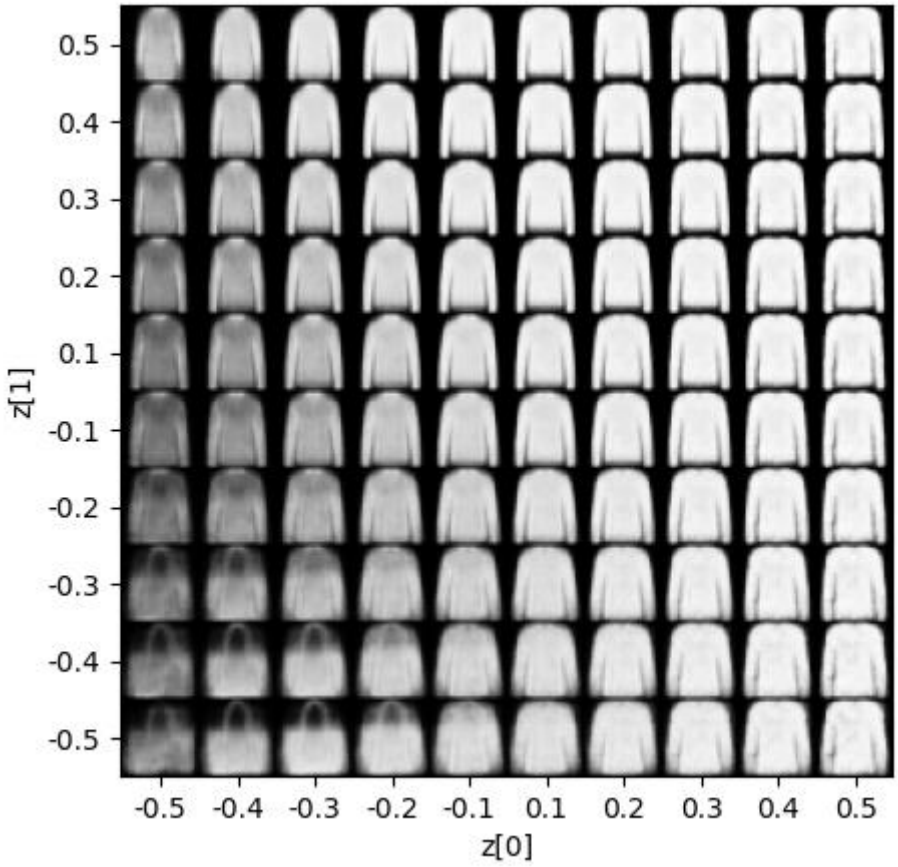
7. Display Sampled Images

- In this step, we display training results, we will be displaying these results according to their values in latent space vectors.

```
import matplotlib.pyplot as plt

def plot_latent_space(vae, n=10, figsize=5):
    # display a n*n 2D manifold of images
    img_size = 28
    scale = 0.5
    figure = np.zeros((img_size * n, img_size * n))
    # linearly spaced coordinates corresponding to the 2D plot
    # of images classes in the latent space
    grid_x = np.linspace(-scale, scale, n)
    grid_y = np.linspace(-scale, scale, n)[::-1]

    for i, yi in enumerate(grid_y):
        for j, xi in enumerate(grid_x):
            sample = np.array([[xi, yi]])
            x_decoded = vae.decoder.predict(sample, verbose=0)
            images = x_decoded[0].reshape(img_size, img_size)
            figure[
                i * img_size : (i + 1) * img_size,
                j * img_size : (j + 1) * img_size,
            ] = images

    plt.figure(figsize=(figsize, figsize))
    start_range = img_size // 2
    end_range = n * img_size + start_range
    pixel_range = np.arange(start_range, end_range, img_size)
    sample_range_x = np.round(grid_x, 1)
    sample_range_y = np.round(grid_y, 1)
    plt.xticks(pixel_range, sample_range_x)
    plt.yticks(pixel_range, sample_range_y)
    plt.xlabel("z[0]")
    plt.ylabel("z[1]")
```

```
    plt.imshow(figure, cmap="Greys_r")
    plt.show()

plot_latent_space(vae)
```

Output:



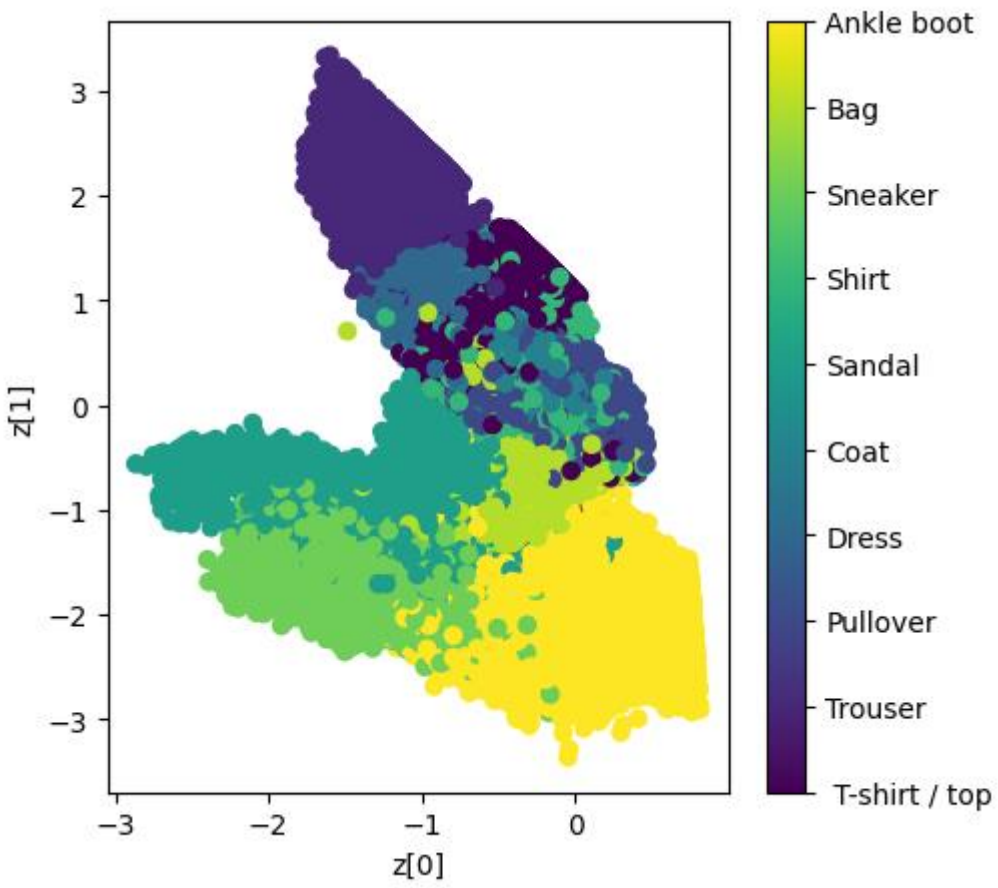
8. Display Latent Space Clusters

- To get a more clear view of our representational latent vectors values, we will be plotting the scatter plot of training data on the basis of their values of corresponding latent dimensions generated from the encoder .

```
def plot_label_clusters(encoder, decoder, data, test_lab):
    z_mean, _, _ = encoder.predict(data)
    plt.figure(figsize =(12, 10))
    sc = plt.scatter(z_mean[:, 0], z_mean[:, 1], c = test_lab)
    cbar = plt.colorbar(sc, ticks = range(10))
    cbar.ax.set_yticklabels([labels.get(i) for i in range(10)])
    plt.xlabel("z[0]")
    plt.ylabel("z[1]")
    plt.show()


labels = {0    :"T-shirt / top",
1:    "Trouser",
```

```
    "Pullover",
    "Dress",
    "Coat",
    "Sandal",
    "Shirt",
    "Sneaker",
    "Bag",
    "Ankle boot"}

(x_train, y_train), _ = keras.datasets.fashion_mnist.load_data()
x_train = np.expand_dims(x_train, -1).astype("float32") / 255
plot_label_clusters(encoder, decoder, x_train, y_train)
```

Output:



1. Reducing Noise with Auto encoders

What is an Autoencoder?
Autoencoders are neural networks that can learn to compress and reconstruct input data, such as images, using a hidden layer of neurons. An autoencoder model consists of two parts: an encoder and a decoder.
The encoder takes the input data and compresses it into a lower-dimensional representation called the latent space. The decoder then reconstructs the input data from the latent space representation. In an optimal scenario, the autoencoder performs as close to perfect reconstruction as possible.
There are several types of autoencoder models, each with its own unique approach to learning these compressed representations:

1. Autoencoding models: These are the simplest type of autoencoder model. They learn to encode input data into a lower-dimensional representation. Then, they decode this representation back into the original input.

2. Contractive autoencoder: This type of autoencoder model is designed to learn a compressed representation of the input data while being resistant to small perturbations in the input. This is achieved by adding a regularization term to the training objective. This term penalizes the network for changing the output with respect to small changes in the input.

3. Convolutional autoencoder (CAE): A Convolutional Autoencoder (CAE) is a type of neural network that uses convolutional layers for encoding and decoding of images. This autoencoder type aims to learn a compressed representation of an image by minimizing the reconstruction error between the input and output of the network. Such models are commonly used for image generation tasks, image denoising, compression, and image reconstruction.

4. Sparse autoencoder: A sparse autoencoder is similar to a regular autoencoder, but with an added constraint on the encoding process. In a sparse autoencoder, the encoder network is trained to produce sparse encoding vectors, which have many zero values. This forces the network to identify only the most important features of the input data.

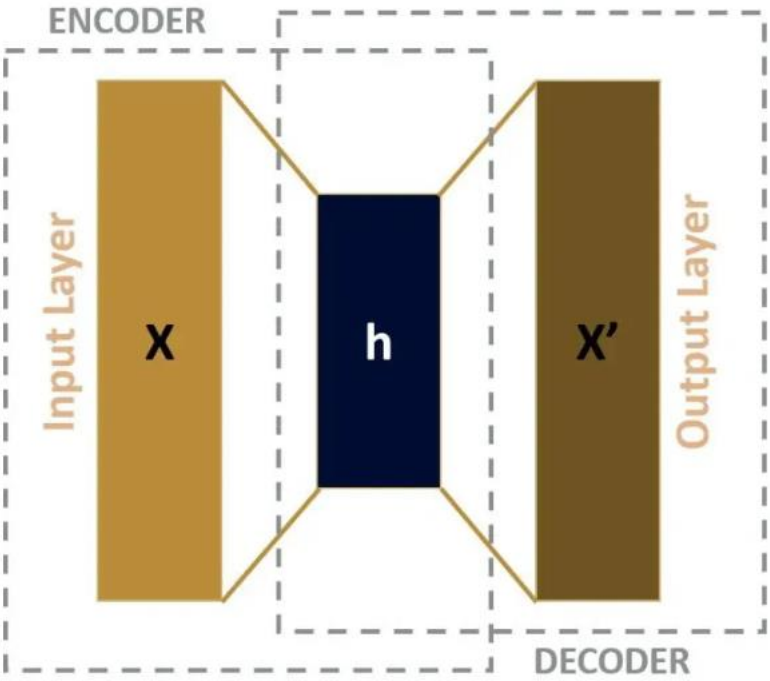
5. Denoising autoencoder: This type of autoencoder is designed to learn to reconstruct an input from a corrupted version of the input. The corrupted input is created by adding noise to the original input, and the network is trained to remove the noise and reconstruct the original input. For example, BART is a popular denoising autoencoder for pretraining sequence-to-sequence models. The model was trained by corrupting text with an arbitrary noising function and learning a model to reconstruct the original text. It is very effective for natural language generation, text translation, text generation and comprehension tasks.

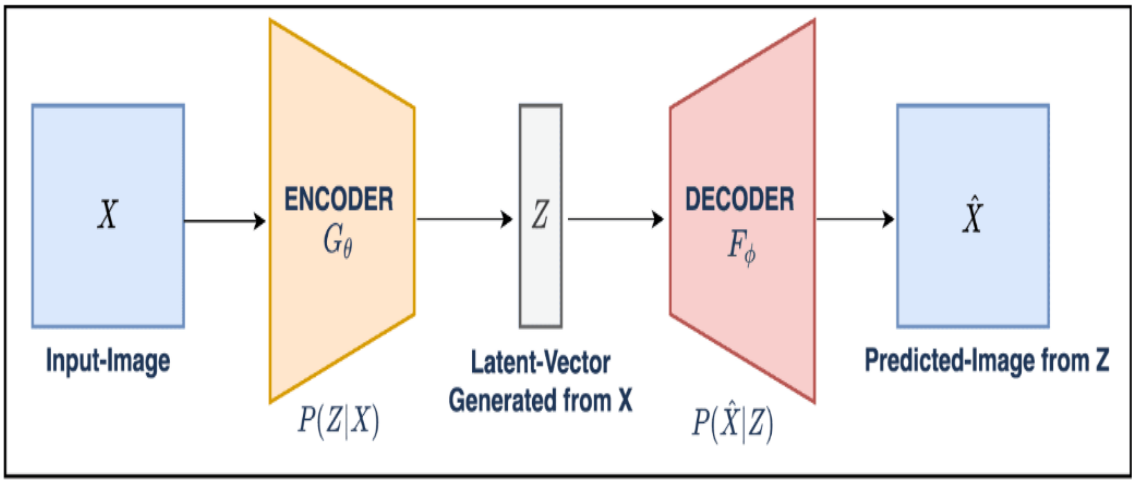
6. Variational autoencoders (VAE): Variational autoencoders are a type of generative model that learns a probabilistic representation of the input data. A VAE model is trained to learn a mapping from the input data to a probability distribution in a lower-dimensional latent space, and then to generate new samples from this distribution. VAEs are commonly used in image and text generation tasks.

7. Video Autoencoder: Video Autoencoder has been introduced for learning representations in a self-supervised manner. For example, a model was developed that can learn representations of 3D structure and camera pose in a sequence of video frames as input (see Pose Estimation). Hence, Video Autoencoder can be trained directly using a pixel reconstruction loss, without any ground truth 3D or camera pose annotations. This autoencoder type can be used for camera pose estimation and video generation by motion following.

8. Masked Autoencoders (MAE): A masked autoencoder is a simple autoencoding approach that reconstructs the original signal given its partial observation. A MAE variant includes masked autoencoders for point cloud self-supervised learning, named Point-MAE. This approach has shown great effectiveness and high generalization capability on various tasks, including object classification, few-show learning, and part-segmentation. Specifically, Point-MAE outperforms all the other self-supervised learning methods.

Autoencoders are composed of 2 key fully connected feedforward neural networks (Figure 1):





An Autoencoder

- **Encoder**: compresses the input data to remove any form of noise and generates a latent space/bottleneck. Therefore, the output neural network dimensions are smaller than the input and can be adjusted as a hyperparameter in order to decide how much lossy our compression should be.
- **Decoder**: making use of only the compressed data representation from the latent space, tries to reconstruct with as much fidelity as possible the original input data (the architecture of this

neural network is, therefore, generally a mirror image of the encoder). The "goodness" of the prediction can then be measured by calculating the reconstruction error between the input and output data using a loss function.

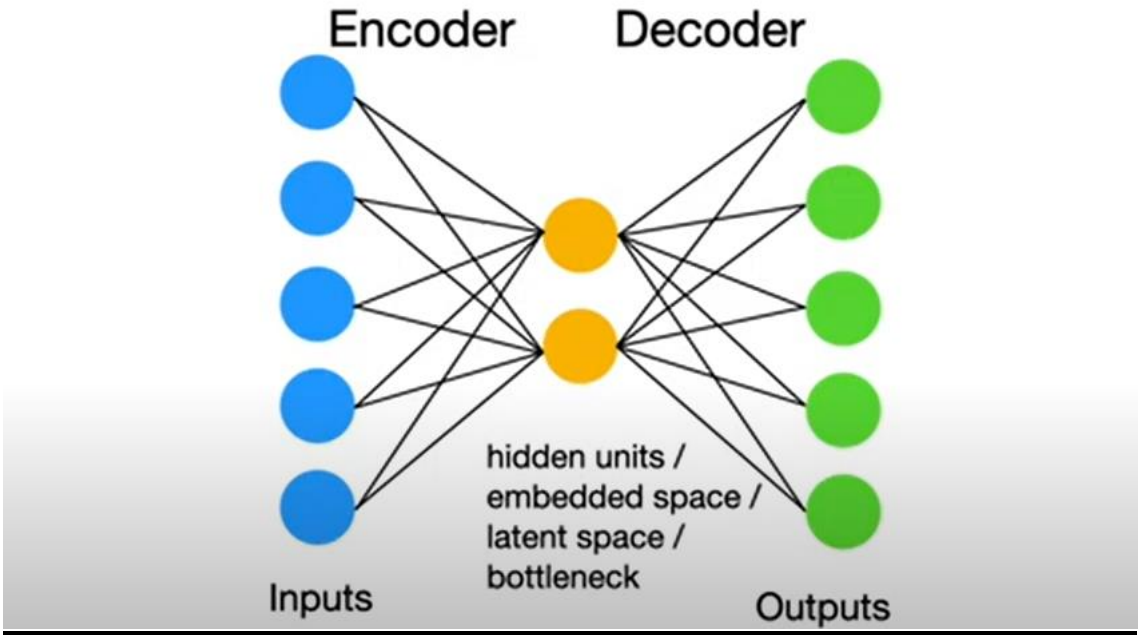
Repeating iteratively this process of passing data through the encoder and decoder and measuring the error to tune the parameters through backpropagation, the Autoencoder can, with time, correctly work with extremely difficult forms of data.

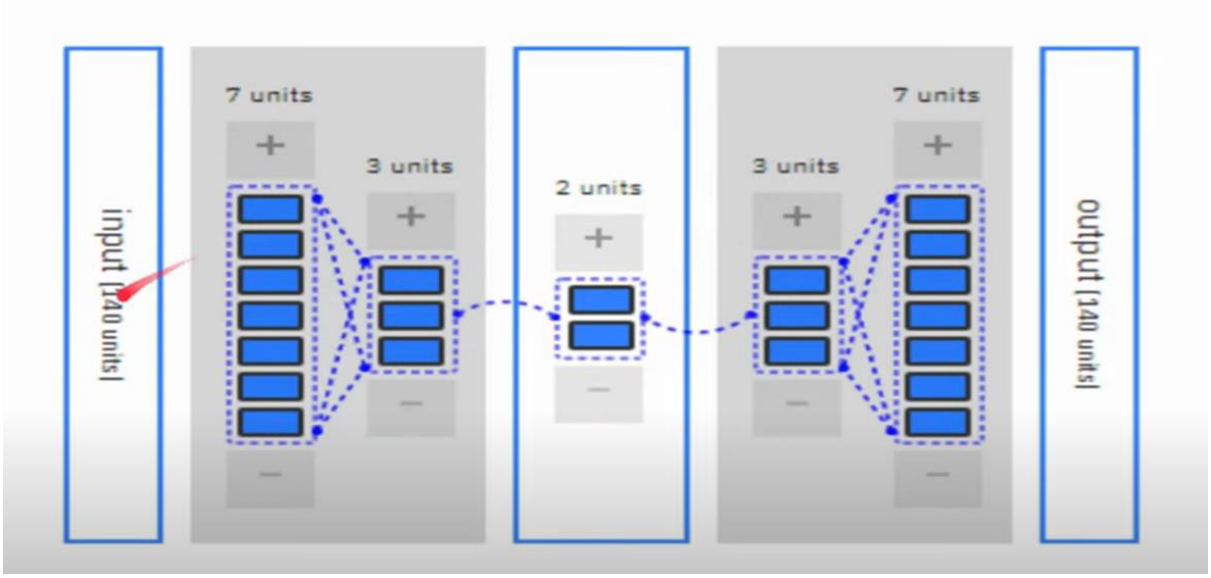
Some of the most common hyperparameters that can be tuned when optimizing your Autoencoder are:

- The number of layers for the Encoder and Decoder neural networks
- The number of nodes for each of these layers
- The loss function to use for the optimization process (e.g., binary cross-entropy or mean squared error)
- The size of the latent space (the smaller, the higher the compression, acting, therefore as a regularization mechanism)

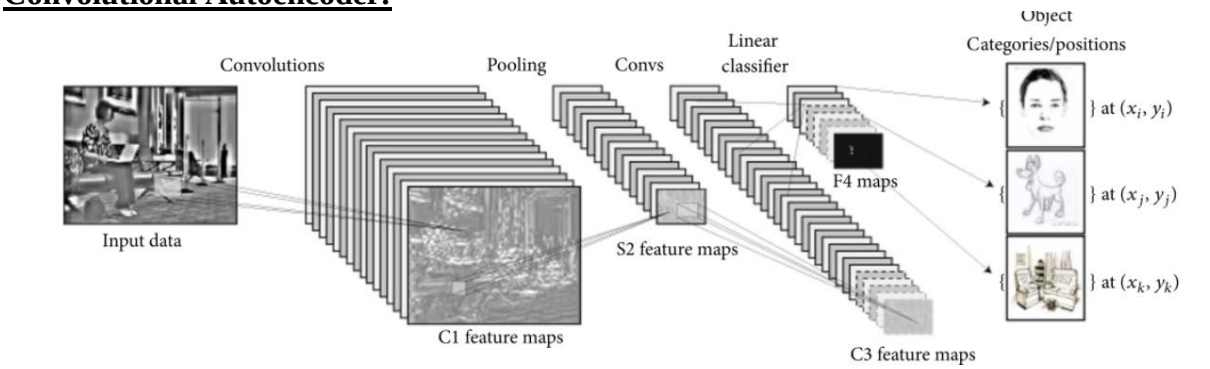
Ideally, a well-trained Autoencoder should be responsive enough to adapt to the input data in order to provide a tailor-made response but not so much as to just mimic the input data and not be able to generalize with unseen data (therefore overfitting).

A basic Fully connected (multilayer perceptron) Autoencoder:

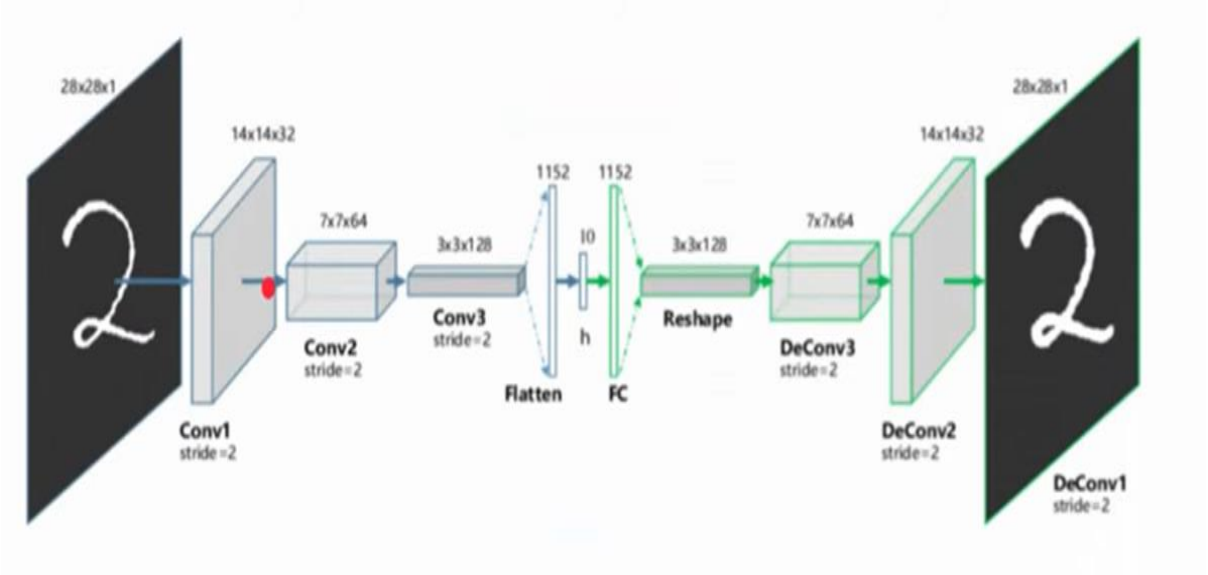




Convolutional Autoencoder:

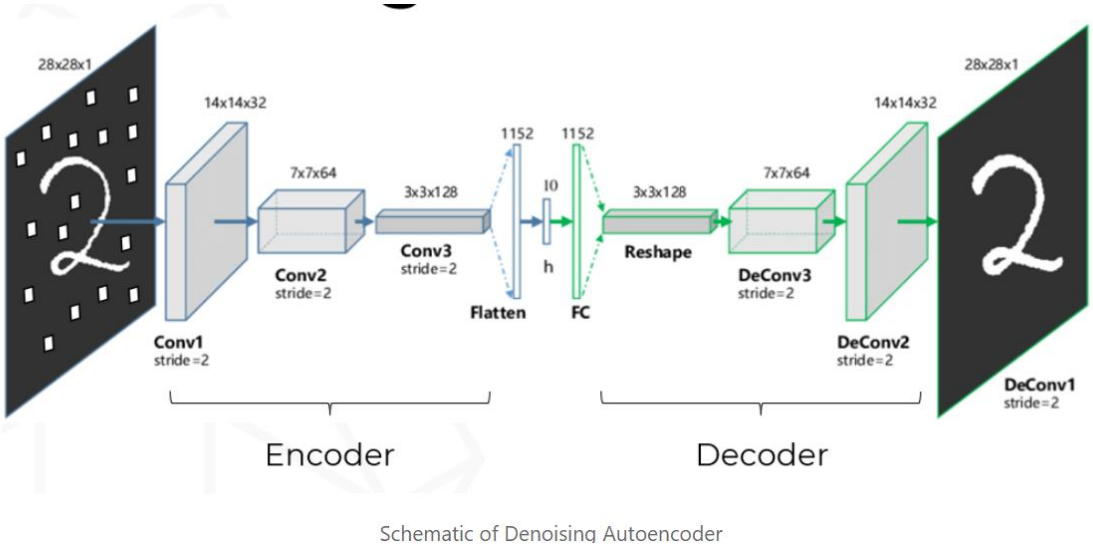


Exemplary Convolutional Neural Network (CNN) architecture for computer vision



Denoising Autoencoder:

- A denoising autoencoder is a neural network model that removes noise from corrupted or noisy data by learning to reconstruct the original data from the noisy version.
- It is trained to minimize the difference between the original and reconstructed data.
- Denoising autoencoders can be stacked to form deep networks for improved performance.



Schematic of Denoising Autoencoder

Structure of a Denoising Autoencoder

- The structure of a denoising autoencoder is similar to that of a traditional autoencoder. It consists of an input layer, one or more hidden layers forming the encoder, a code layer where the representation is compressed, one or more hidden layers forming the decoder, and an output layer.
- The key difference is in the training process, where the input data is deliberately corrupted before being fed into the network.

Training a Denoising Autoencoder

To train a denoising autoencoder, the following steps are typically followed:

1. **Corruption Process:** The original clean input data is corrupted using a stochastic process, which adds some form of noise. This could be Gaussian noise, masking noise (where part of the input is randomly set to zero), or salt-and-pepper noise (where random pixels are set to their maximum or minimum value).
2. **Encoding:** The corrupted data is passed through the encoder to produce a lower-dimensional code. The encoder learns to capture the essential features that are robust to the corruption process.
3. **Decoding:** The decoder then attempts to reconstruct the original clean data from the encoded representation. The reconstruction is compared to the original clean data (not the corrupted version), and a loss is computed.
4. **Backpropagation:** The loss is backpropagated through the network to update the weights, with the goal of minimizing the reconstruction error.

Advantages and Limitations

Advantages:

- Denoising autoencoders can learn more robust features compared to standard autoencoders.
- They can improve the performance of other machine learning models by providing cleaner data.
- The approach is unsupervised, which means it does not require labeled data for training.

Limitations:

- The choice of noise and its level can greatly affect the performance of the model and may require careful tuning.

- Like other neural networks, denoising autoencoders can be computationally intensive to train, especially for large datasets or complex architectures.
- While they can remove noise, they may also lose some detail or relevant information in the data if not properly regularized.

Conclusion

Denoising autoencoders are a powerful variant of autoencoders that have the added capability of learning to remove noise from data. Through their training process, they learn to capture the most significant features of the data distribution, leading to more robust representations that are useful for various machine learning tasks. Despite their limitations, denoising autoencoders remain a popular choice for unsupervised learning, especially in the field of image and signal processing.

To build a simple autoencoder for image processing, you can use TensorFlow/Keras. Here's a step-by-step guide:

Step 1: Import Libraries

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Flatten, Reshape
from tensorflow.keras.datasets import mnist
from tensorflow.keras.optimizers import Adam
```

Step 2: Load and Preprocess the Data

Here we use the MNIST dataset as an example.

```
# Load MNIST dataset
(x_train, _), (x_test, _) = mnist.load_data()

# Normalize pixel values to range [0, 1]
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Flatten the images for a fully connected autoencoder
x_train = x_train.reshape((x_train.shape[0], -1))
x_test = x_test.reshape((x_test.shape[0], -1))

input_dim = x_train.shape[1]  # Input size
```

Step 3: Build the Autoencoder

An autoencoder has two parts: an encoder and a decoder.

- **Encoder:**

```
encoding_dim = 64  # Dimension of the encoded representation

input_img = Input(shape=(input_dim,))
encoded = Dense(encoding_dim, activation='relu')(input_img)
```

- **Decoder:**

```
decoded = Dense(input_dim, activation='sigmoid')(encoded)
```

- **Combine into an Autoencoder Model:**

```
autoencoder = Model(input_img, decoded)

# Compile the model
autoencoder.compile(optimizer=Adam(), loss='binary_crossentropy')
```

Step 4: Train the Autoencoder

```
autoencoder.fit(
    x_train, x_train,  # Input and output are the same
    epochs=20,
    batch_size=256,
    shuffle=True,
    validation_data=(x_test, x_test)
)
```

Step 5: Visualize Results
Encode and Decode Test Images:

```
decoded_imgs = autoencoder.predict(x_test)

# Reshape for visualization
x_test_images = x_test.reshape((-1, 28, 28))
decoded_images = decoded_imgs.reshape((-1, 28, 28))
```

Plot Original and Reconstructed Images:

```
n = 10  # Number of images to display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test_images[i], cmap='gray')
    plt.title("Original")
    plt.axis('off')

    # Display reconstructed
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_images[i], cmap='gray')
    plt.title("Reconstructed")
    plt.axis('off')

plt.show()
```

- **Second example: Image denoising**

An autoencoder can also be trained to remove noise from images. In the following section, you will create a noisy version of the Fashion MNIST dataset by applying random noise to each image. You

will then train an autoencoder using the noisy image as input, and the original image as the target. Let's reimport the dataset to omit the modifications made earlier.

```
(x_train, _), (x_test, _) = fashion_mnist.load_data()


x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

x_train = x_train[..., tf.newaxis]
x_test = x_test[..., tf.newaxis]

print(x_train.shape)
```

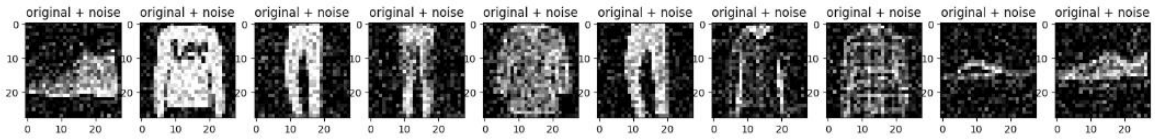
```
(60000, 28, 28, 1)
```

Adding random noise to the images

```
noise_factor = 0.2
x_train_noisy = x_train + noise_factor * tf.random.normal(shape=x_train.shape)
x_test_noisy = x_test + noise_factor * tf.random.normal(shape=x_test.shape)

x_train_noisy = tf.clip_by_value(x_train_noisy, clip_value_min=0., clip_value_max=1.)
x_test_noisy = tf.clip_by_value(x_test_noisy, clip_value_min=0., clip_value_max=1.)
```

Plot the noisy images.

```
n = 10
plt.figure(figsize=(20, 2))
for i in range(n):
    ax = plt.subplot(1, n, i + 1)
    plt.title("original + noise")
    plt.imshow(tf.squeeze(x_test_noisy[i]))
    plt.gray()
plt.show()
```



Define a convolutional autoencoder

In this example, you will train a convolutional autoencoder using Conv2D layers in the encoder, and Conv2DTranspose layers in the decoder.

```
class Denoise(Model):
  def __init__(self):
    super(Denoise, self).__init__()
    self.encoder = tf.keras.Sequential([
      layers.Input(shape=(28, 28, 1)),
      layers.Conv2D(16, (3, 3), activation='relu', padding='same', strides=2),
      layers.Conv2D(8, (3, 3), activation='relu', padding='same', strides=2)])

    self.decoder = tf.keras.Sequential([
      layers.Conv2DTranspose(8, kernel_size=3, strides=2, activation='relu', padding='same'),
      layers.Conv2DTranspose(16, kernel_size=3, strides=2, activation='relu', padding='same'),
      layers.Conv2D(1, kernel_size=(3, 3), activation='sigmoid', padding='same')])

  def call(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return decoded

autoencoder = Denoise()
```

```
autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())
```

```
autoencoder.fit(x_train_noisy, x_train,
                epochs=10,
                shuffle=True,
                validation_data=(x_test_noisy, x_test))
```

```
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.0077 - val_loss: 0.0076
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.0075 - val_loss: 0.0074
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.0073 - val_loss: 0.0073
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.0072 - val_loss: 0.0072
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.0071 - val_loss: 0.0071
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.0071 - val_loss: 0.0070
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - loss: 0.0070 - val_loss: 0.0070
<keras.src.callbacks.history.History at 0x7f8ad316bb20>
```

Let's take a look at a summary of the encoder. Notice how the images are downsampled from 28x28 to 7x7.

```
autoencoder.encoder.summary()
```

```
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 14, 14, 16)	160
conv2d_1 (Conv2D)	(None, 7, 7, 8)	1,160

Total params: 1,320 (5.16 KB)
Trainable params: 1,320 (5.16 KB)
Non-trainable params: 0 (0.00 B)
The decoder upsamples the images back from 7x7 to 28x28.

```
autoencoder.decoder.summary()
```

```
Model: "sequential_3"
```

Layer (type)	Output Shape	Param #
conv2d_transpose (Conv2DTranspose)	(32, 14, 14, 8)	584
conv2d_transpose_1 (Conv2DTranspose)	(32, 28, 28, 16)	1,168
conv2d_2 (Conv2D)	(32, 28, 28, 1)	145

Total params: 1,897 (7.41 KB)
Trainable params: 1,897 (7.41 KB)
Non-trainable params: 0 (0.00 B)
Plotting both the noisy images and the denoised images produced by the autoencoder.

```
encoded_imgs = autoencoder.encoder(x_test_noisy).numpy()
decoded_imgs = autoencoder.decoder(encoded_imgs).numpy()
```

```
1723784981.399198  160375 gpu_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced
1723784981.413427  160375 gpu_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced
1723784981.443673  160375 gpu_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced
1723784981.502975  160375 gpu_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced
1723784981.545023  160375 gpu_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced
1723784981.626188  160375 gpu_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced
1723784981.642306  160375 gpu_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced
1723784981.657118  160375 gpu_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced
1723784981.675985  160375 gpu_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced
1723784981.707544  160375 gpu_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced
1723784981.718098  160375 gpu_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced
1723784981.751437  160375 gpu_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced
1723784981.774578  160375 gpu_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced
```

```
n = 10
plt.figure(figsize=(20, 4))
for i in range(n):

    # display original + noise
    ax = plt.subplot(2, n, i + 1)
    plt.title("original + noise")
    plt.imshow(tf.squeeze(x_test_noisy[i]))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # display reconstruction
    bx = plt.subplot(2, n, i + n + 1)
    plt.title("reconstructed")
    plt.imshow(tf.squeeze(decoded_imgs[i]))
    plt.gray()
    bx.get_xaxis().set_visible(False)
    bx.get_yaxis().set_visible(False)
plt.show()
```