Unit-IV

Topics Covered

Object detection

1. General object detection framework,
2. Region-based convolutional neural networks(R-CNNs) ,
3. Single-shot detector (SSD) ,
4. You only look once (YOLO) ,

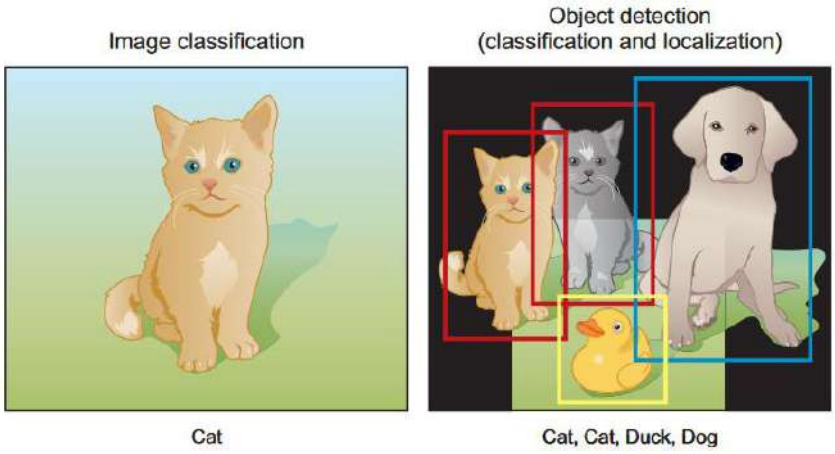
Project: Train an SSD network in a self-driving car application

Segmentation:

- Mask R-CNN and
- Instance Segmentation,
- U-Net and
- **Semantic Segmentation Reducing Noise with Auto encoders**
    - Creating a simple fully connected autoencoder,
    - creating a convolutional autoencoder,
    - Denoising images with autoencoders,
    - Spotting outliers using Autoencoders,
    - Variational Autoencoders,
    - creating an inverse image search index with deep learning,
    - Implementing a variational autoencoder.

Deep neural networks, are used for image classification tasks there is only one main target object in the image, and the model's sole focus in classifying or identify the target category.

But sometimes there will be multiple targets in the image we, model to classify them, but also obtain their specific positions in the image. In computer vision, we refer to such tasks as object detection.



Object detection:

The goal of object detection 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.

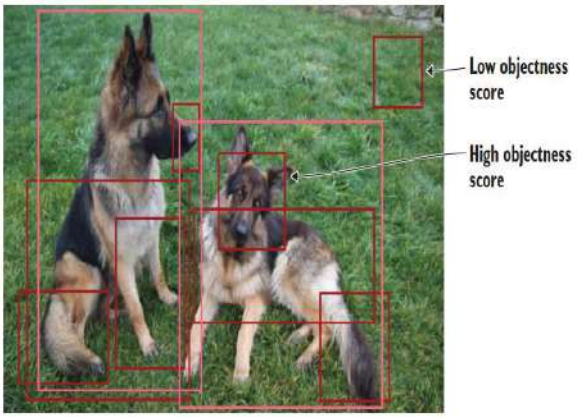
Applications of Object detection :

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

1. General object detection framework:

An object detection framework has four components:

1 region proposal—an algorithm like elective search algorithm is used to generate regions of interest (ROI) 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.

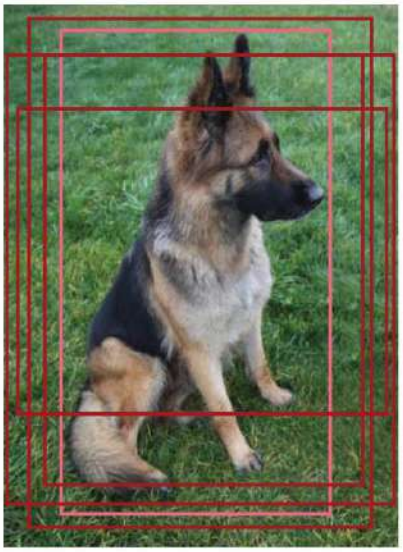


Regions of interest (RoIs) proposed by the system.

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).

it is very computationally expensive and will slow down detection.

So, the trade-off with generating region proposals is the number of regions versus computational complexity—and the right approach is to use problem-specific information to reduce the number of RoIs.



The network analyzes all the regions that have been identified as having a high likelihood of containing an object and makes two predictions foreach 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.

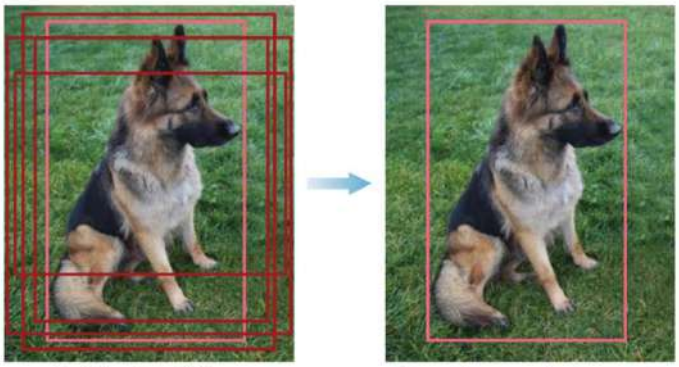
The bounding-box detector produces more than one bounding box for an object. We want to consolidate these boxes into one bounding box that fits the object the most.

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.

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)



Predictions before NMS

After applying non-maximum suppression

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.

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.

4 Object-detection evaluation metrics—similar to accuracy, precision, and recall metrics in image Classification tasks (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).

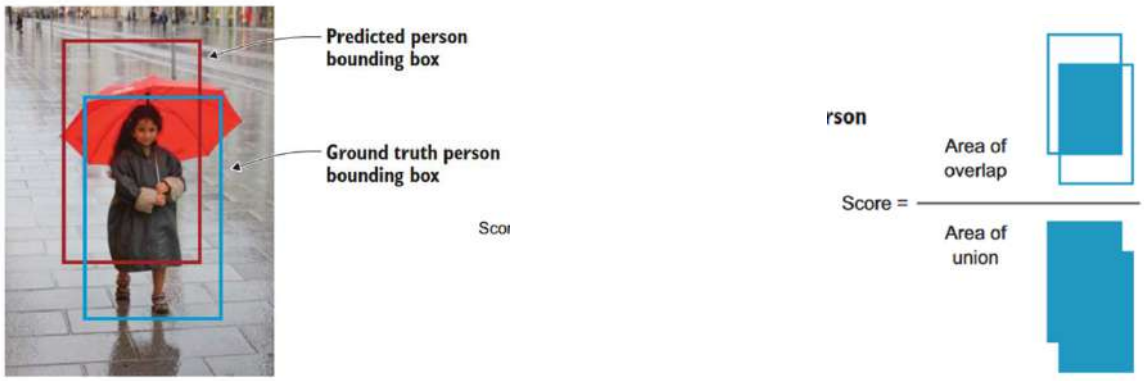
FRAMES PER SECOND (FPS) TO MEASURE DETECTION SPEED

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
evaluation metric used in object recognition tasks is mean average precision (mAP).

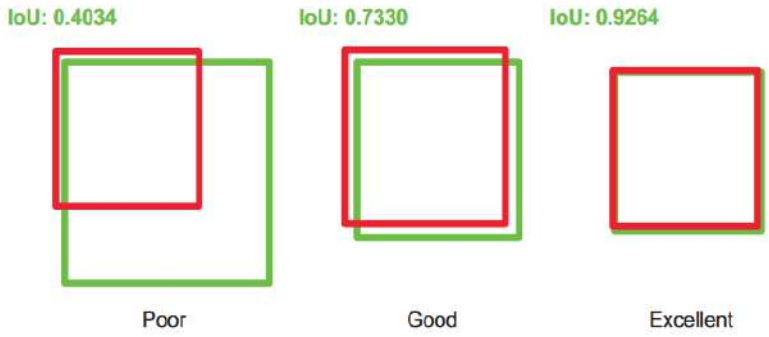
It is a percentage from 0 to 100, and higher values are typically better, but its value is different from the accuracy metric used in classification.
To understand how mAP is calculated, you first need to understand intersection over union (IoU) and the precision-recall curve (PR curve). Let's explain 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 (Bground truth) and the predicted bounding box (Bpredicted).
By applying the IoU, we can tell whether a detection is valid (True Positive) or not (False Positive).
Figure 7.5 illustrates the IoU between a ground truth bounding box and a predictedbounding 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.

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,

- The ground truth bounding box (Bground truth):
  the hand-labeled bounding box created during the labeling process
- The predicted bounding box (Bpredicted) 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):

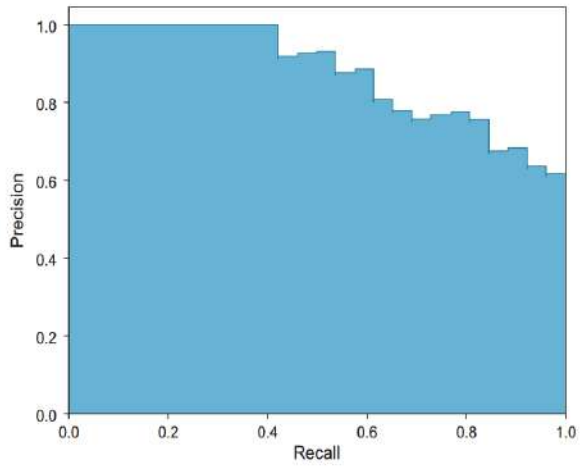
With the TP and FP defined, we can now calculate the precision and recall of our detection for a given class across the testing dataset.

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

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

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

To 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.



2a. Region-based convolutional neural networks(R-CNNs) :

R-CNNs was developed by Ross Girshick in 2014.

Family of R-CNN are Fast-RCNN 2 and Faster-RCN 3 developed in 2015 and 2016, respectively.

R-CNN is the least sophisticated *region-based architecture* in its family, but it is the basis for understanding *how multiple object-recognition algorithms* work for all of them.

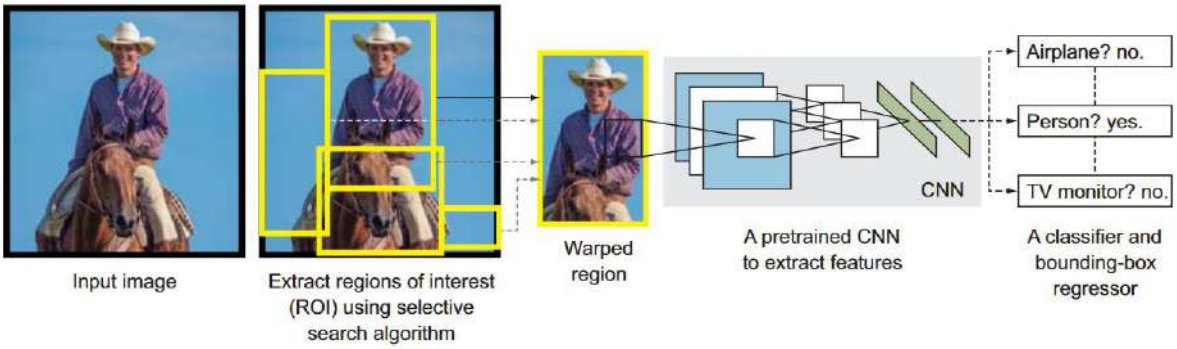
It was one of the first large, successful applications of CNN to the problem of *object detection and localization*, and it paved the way for the other advanced detection algorithms.

The approach was demonstrated on benchmark datasets, achieving then-state-of-the-art results on the PASCAL VOC-2012 dataset and the ILSVRC 2013 object detection challenge.

R-CNN model architecture:

R-CNNs are composed of **four modules:**

- Selective search region proposal,
- feature extractor,
- classifier,
- and bounding-box regressor.



The R-CNN model consists of four components:

- **Extract regions of interest** or region proposals. These regions have a high probability of containing an object.
  An algorithm called **selective search scans** the input image to find regions that contain blobs, and proposes them as RoIs to be processed by the next modules in the pipeline. The proposed RoIs are then warped to have a fixed size; they usually vary in size, but as we learned in previous chapters, CNNs require a fixed input image size.
  Selective Search:
    - Generate initial sub-segmentation, we generate many candidate regions
    - Use greedy algorithm to recursively combine similar regions into larger ones
    - Use the generated regions to produce the final candidate region proposals
- **Feature extraction module**—We run a pretrained convolutional network on top of the region proposals to extract features from each candidate region.
- **Classification module**—We train a classifier like a support vector machine (SVM), a traditional machine learning algorithm, to classify candidate detections based on the extracted features from the previous step.
- **Localization module**—Also known as a bounding-box regressor. Let's take a step back to understand regression. ML problems are categorized as classification or regression problems. Classification algorithms output discrete, predefined classes (dog,cat, elephant), whereas regression algorithms output continuous value predictions. In this module, we want to predict the location and size of the bounding box that surrounds the object.

The bounding box is represented by identifying four values: the x and y coordinates of the box's origin (x, y), the width, and the height of the box (w, h). Putting this together, the regressor predicts the four real-valued numbers that define the bounding box as the following tuple:(x, y, w, h).

Disadvantages with R-CNN:

- It still takes a huge amount of time to train the network as you would have to classify 2000 region proposals per image.
- It cannot be implemented real time as it takes around 47 seconds for each test image.
- The selective search algorithm is a fixed algorithm. Therefore, no learning is happening at that stage. This could lead to the generation of bad candidate region proposals.

2b. Fast R-CNN

It is resembles the R-CNN technique in many ways but improved on its ***detection speed*** while also increasing ***detection accuracy*** through two main changes:

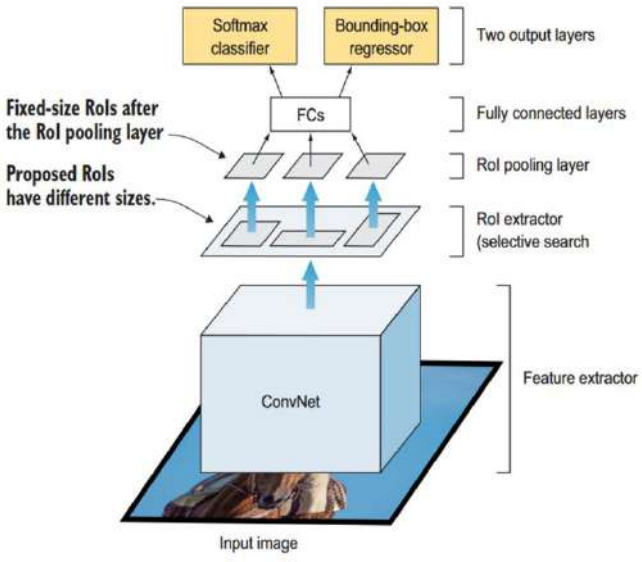
- Instead of starting with the ***regions proposal module*** and then using the ***feature extraction module***, like R-CNN,
- Fast-RCNN proposes that we apply the **CNN feature extractor first** to the entire input image and **then propose regions**. This way, we run only one **ConvNet** over the entire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

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
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:

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.'**

In any optimization problem, we need to define a loss function that our optimizer algorithm is trying to minimize.

To optimize object detection we use two goals: object classification and object localization.

Therefore, we have two loss functions in this problem:
L_{cls} for the classification loss and
L_{loc} 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:

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

– L1smooth is the bounding box loss that measures the difference between t iu 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 ≥ 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 ≥ 1] is defined as

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

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, as we will see next.

2c. 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 input to a convolutional network*** that provides a feature map. Instead of selective search algorithm to give feature map to identify the region proposals,

a region proposal network (RPN) is a part of the training process used to *predict the region proposals* .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 then state-of-the-art performance.

FASTER R-CNN ARCHITECTURE:

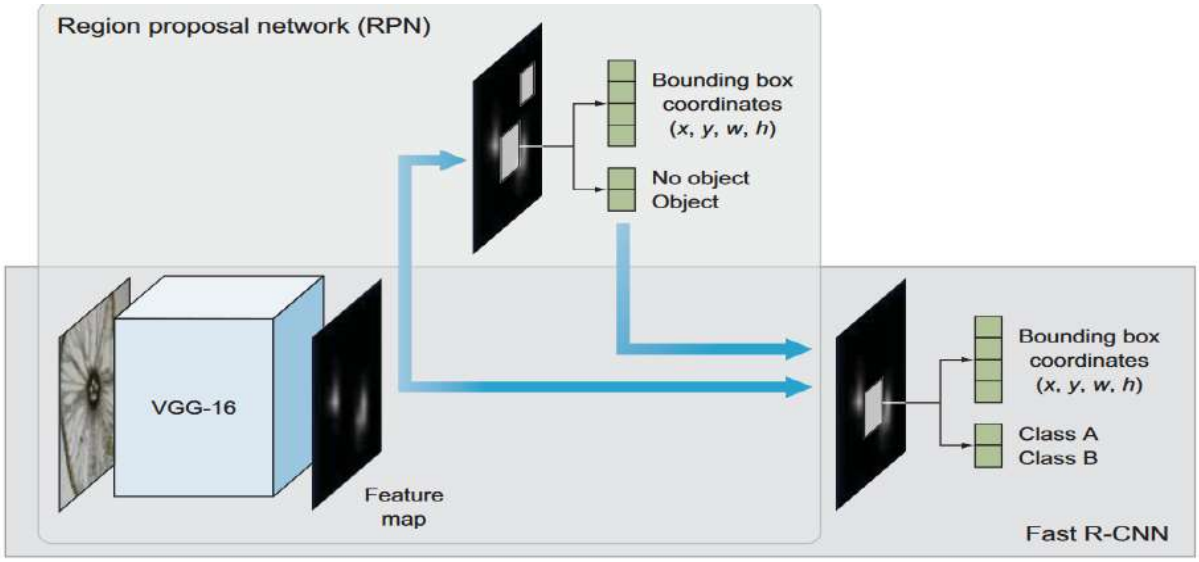
The architecture of Faster R-CNN has two main networks:

- 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 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 contains **two fully connected layers**: a *softmax classifier* to output the class probability and a *bounding box regression* CNN for the bounding box predictions



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.

3. Single-shot detector (SSD) :

SSD (Single Shot Detector) is specifically designed for real-time object detection, and it achieves this by making several key optimizations compared to the Faster R-CNN approach.

Faster R-CNN uses a two-stage process involving a region proposal network (RPN) to generate candidate object bounding boxes, followed by object classification.

While Faster R-CNN is renowned for its accuracy, it operates at a slower frame rate, typically around 7 frames per second, which may not meet the demands of real-time processing.

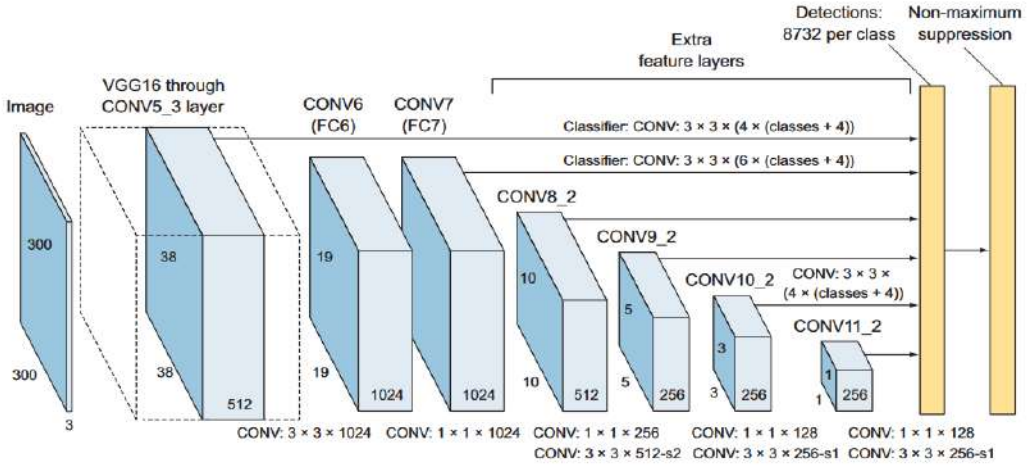
SSD *eliminates the need of RPN in object detection.*

It directly predicts object bounding boxes and their corresponding class labels in a single pass. This approach speeds up the detection process.

SSD incorporates several improvements:

1. ***Multi-Scale Features:*** SSD utilizes feature maps from multiple scales in the network. This allows it to detect objects of various sizes effectively. Objects of different sizes can be present in an image, and using multi-scale features helps ensure that they are all detected accurately.
2. ***Default Boxes:*** SSD introduces the concept of default boxes or anchor boxes. These default boxes are pre-defined bounding boxes at different aspect ratios and scales, enabling the model to predict objects more accurately. Each default box corresponds to a specific scale and aspect ratio, which helps improve detection precision.

3. These enhancements collectively enable **SSD to match the accuracy** of Faster R-CNN while processing images at a lower resolution. Lower-resolution images are faster to process, which further boosts the speed of SSD.



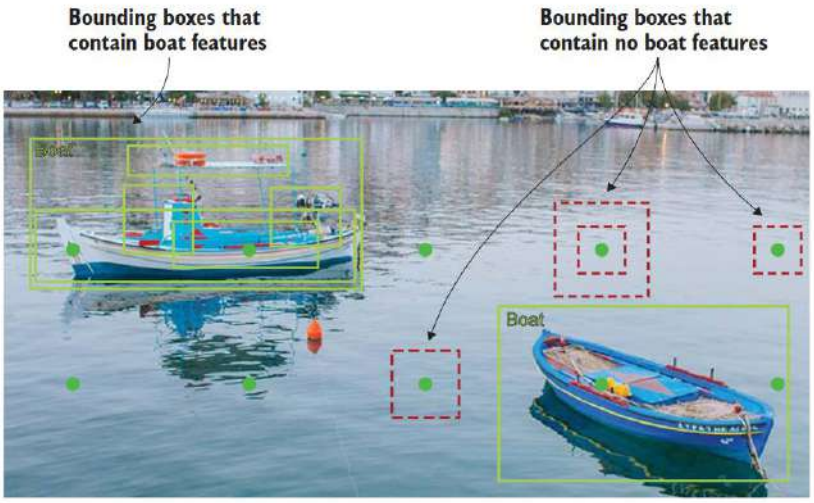
The SSD architecture is composed of a

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.

In summary, SSD achieves real-time processing speeds by simplifying the object detection pipeline and eliminating the need for a region proposal network.

It recovers any potential loss in accuracy by incorporating multi-scale features and default boxes.

This makes SSD a competitive choice for real-time object detection, often outperforming Faster R-CNN in terms of speed while maintaining comparable accuracy (measured as mean average precision, or mAP).

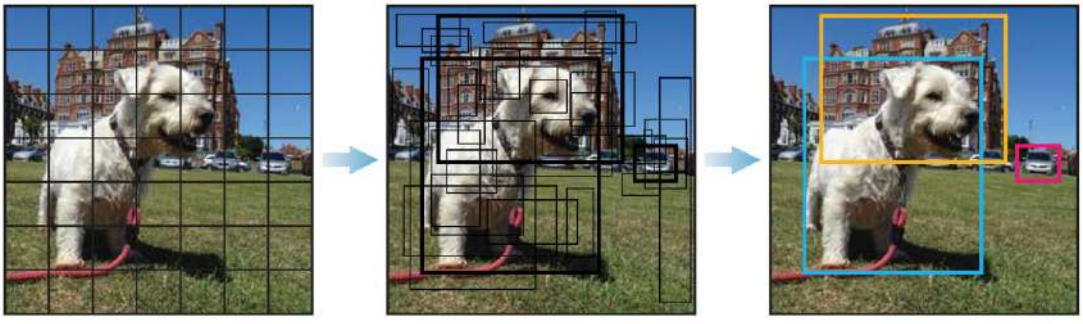


The SSD base network looks at the anchor boxes to find features of a boat. Solid boxes indicate that the network has found boat features. Dotted boxes indicate no boat features.

4. You only look once (YOLO) :

Another RCNN family is YOLO used for object detection networks developed by Joseph Redmon et al.

- YOLOv1, published in 2016 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 2016 —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 2018—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.
- 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



The YOLO network splits the input image into a grid of S × 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 objectnessscore along with their class predictions, as follows:

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

Objectness score (P0)—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:

$$P_0 = \Pr(\text{containing an object}) \times \text{IoU}(\text{pred}, \text{truth})$$

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 soft-max 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.

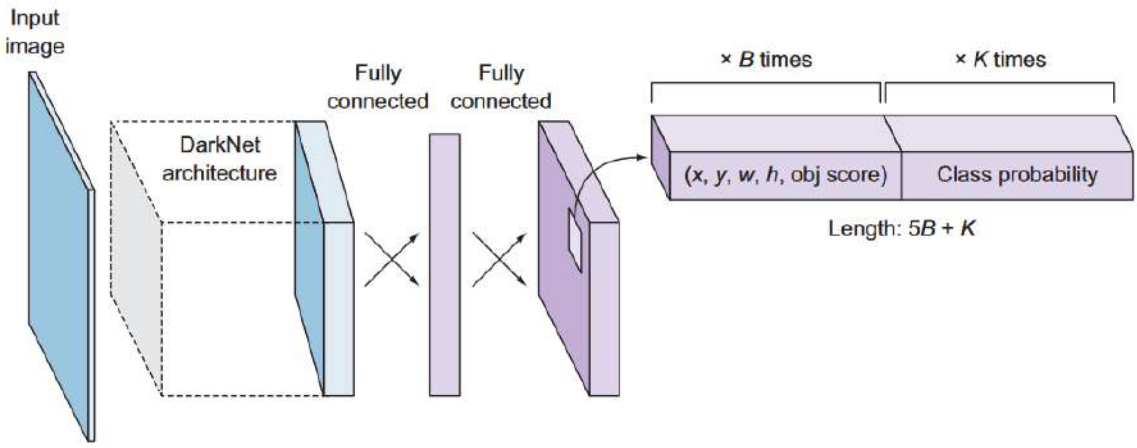
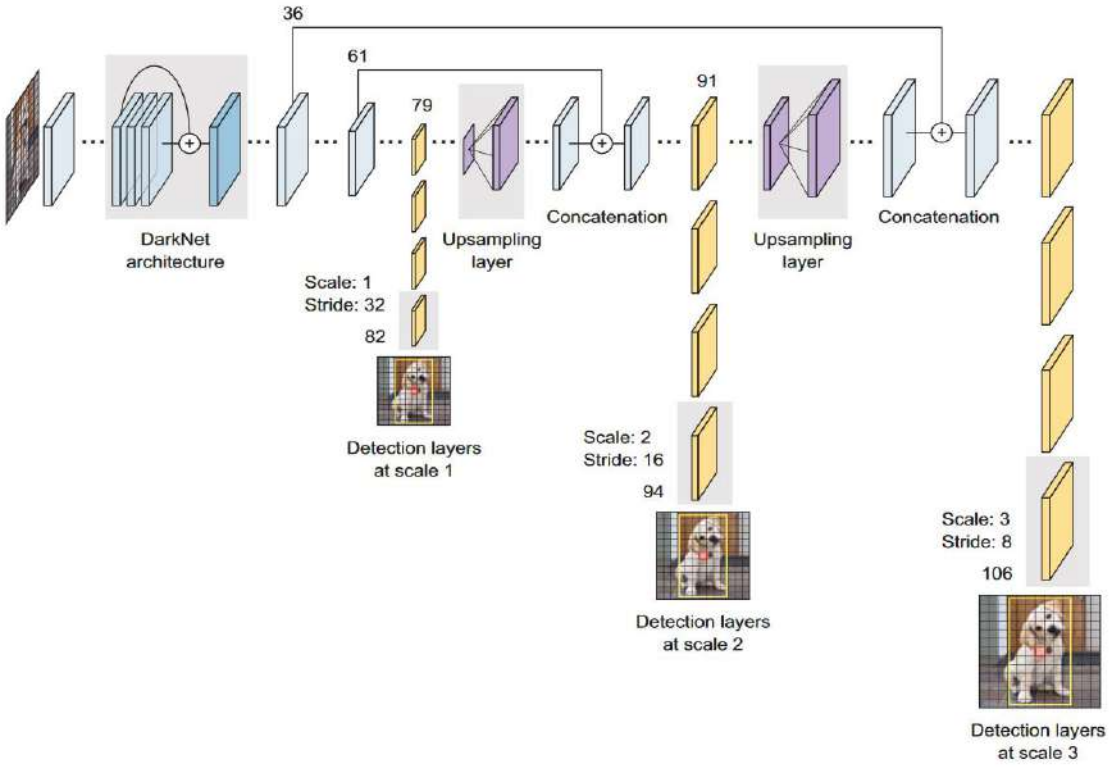


Fig :YOLO



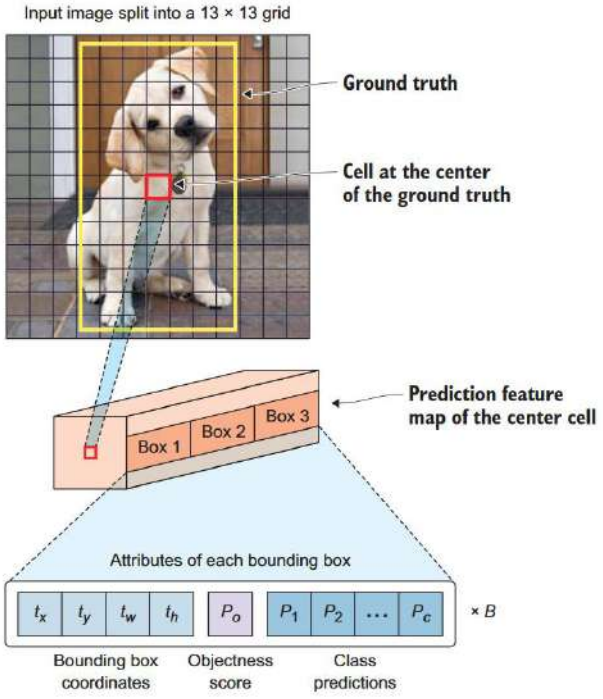
Example of a YOLOv3 workflow

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.



Segmentation

What is Image Segmentation?

Image segmentation is the task of identifying and classifying multiple categories of objects. But if you go deep into segmentation, we get confusion, but we will know considerable difference between different types of segmentation, mainly semantic and instance segmentation.

Semantic Segmentation

Instance Segmentation

1. What is 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.

It is essential to understand that semantic segmentation classifies image pixels of one or more classes rather than real-world objects which are not semantically interpretable. Due to its intricate working scheme, it is a difficult task in the computer vision ecosystem as you classify each pixel instead of objects, which is the case in **object detection**.

How Does Semantic Segmentation Work?

Semantic segmentation aims to extract features before using them to form distinct categories in an image. The steps involved are as follows:

- Analyze training data for classifying a specific object in the image.
- Create a semantic segmentation network to localize the objects and draw a bounding box around them.
- Train the **semantic segmentation network** to group the pixels in a localized image by creating a segmentation mask.

Applications of Semantic Segmentation

1. Medical Diagnostics: For detecting medical abnormalities in X-Rays, CT Scans, MRI Scans
2. GeoSensing: For land usage mapping from satellite imagery and monitoring areas of deforestation and urbanization
3. Autonomous Driving: For accurately detecting lanes, pedestrians, traffic signs, road, sky and other vehicles on the road

2. What is Instance Segmentation?

Instance Segmentation is a unique form of **image segmentation** that deals with detecting and delineating each distinct instance of an object appearing in an image. Instance segmentation detects all instances of a class with the extra functionality of demarcating separate instances of any segment class. Hence, it is also referred to as incorporating **object detection** and **semantic segmentation** functionality.

Instance segmentation has a richer output format as it creates a segment map for each category and instance of that class. Simply put, consider you have an image with dogs and cats. By running an **instance segmentation model** on that image, you can locate the bounding boxes of each dog and cat, plot segmentation maps for each dog and cat, and count how many dogs and cats are in the image.

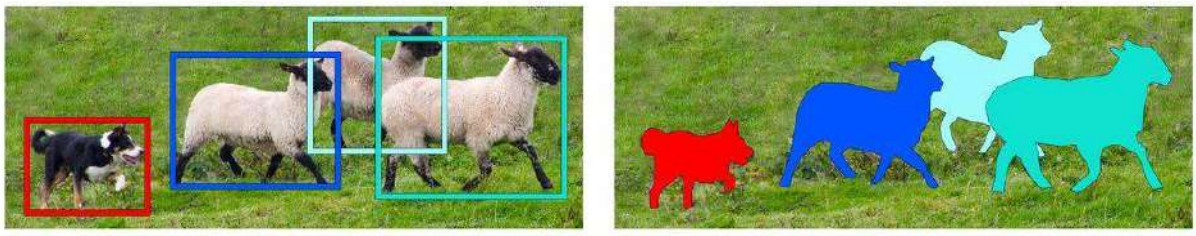
Street view in Instance Segmentation

How Does Instance Segmentation Work?

Instance segmentation involves identifying boundaries of the objects at the detailed pixel level, making it a complex task to perform. But as we saw earlier, instance segmentation contains 2 significant parts:

1. **Object Detection:** Firstly, it runs object detection to find all bounding boxes for every object in an image
2. **Semantic Segmentation:** After finding all the rectangles (bounding boxes), it uses a semantic segmentation model inside every rectangle

Note: Instance segmentation only differentiates all instances in each class; for example, it will separate every person into a different class.



Object Detection Instance Segmentation

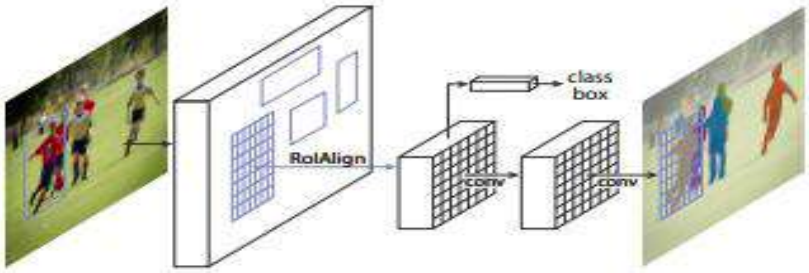
Applications of instance segmentation

Here are a few real-world applications of instance segmentation:

1. **Medical Domain:** Used to detect and segment tumors in MRI scans of the brain and nuclei in images
2. **Satellite Imagery:** Used to achieve a better separation between the objects, such as counting cars, detecting ships for maritime security, and sea pollution monitoring
3. **Self-Driving Cars:** Used in conjunction with dense distance to object estimation methods to provide high-resolution 3D depth estimation of a scene from monocular 2D images
4. **Robotics:** Used with self-supervised learning to segment visual observations into individual objects by interacting with the environment
5. **Automation:** Used for detecting dents on a car, separating buildings in a city, and more

3. 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.



It comprises of –

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

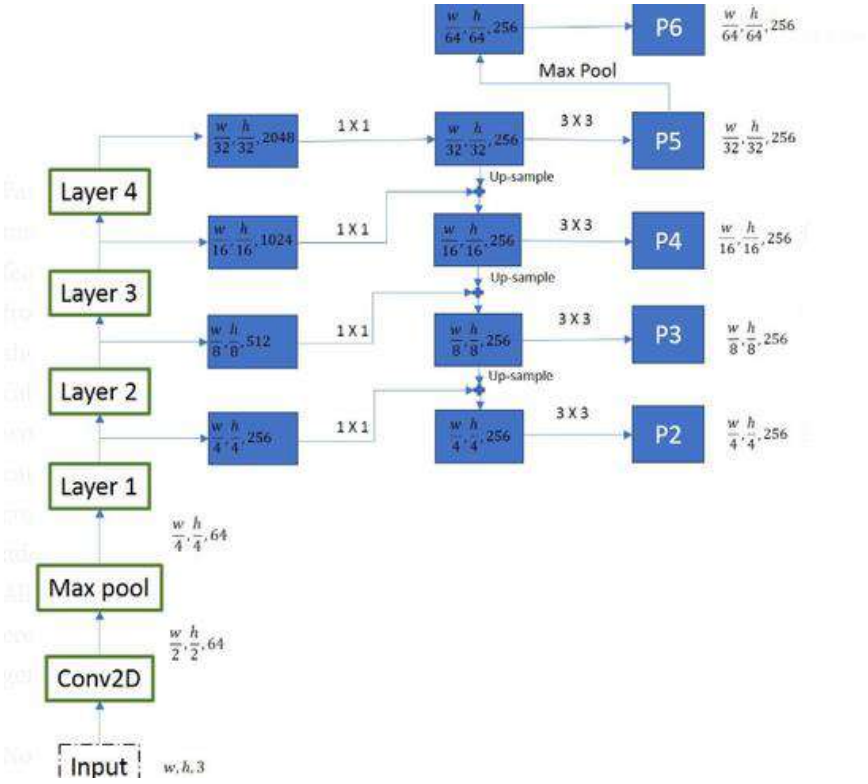
Backbone Network

Mask R-CNN is 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.

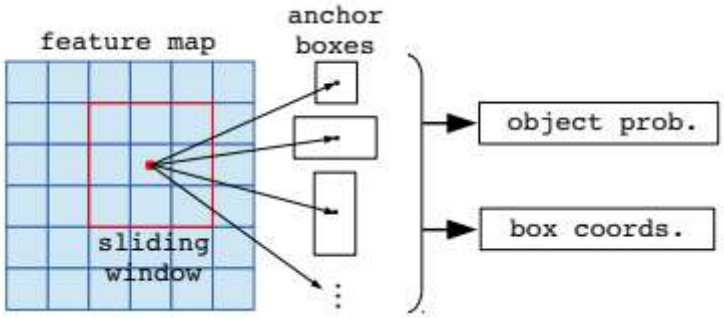


Mask R-CNN backbone architecture

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).

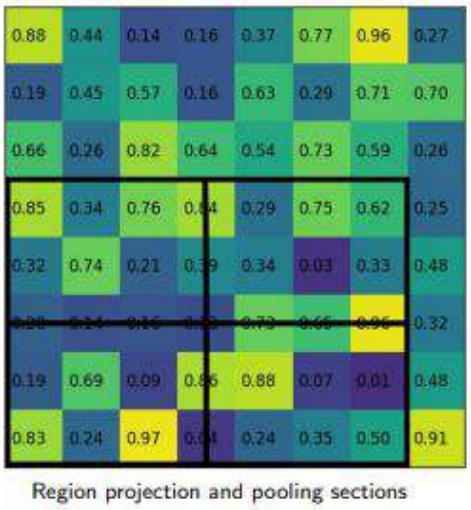
Mask Representation

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*.

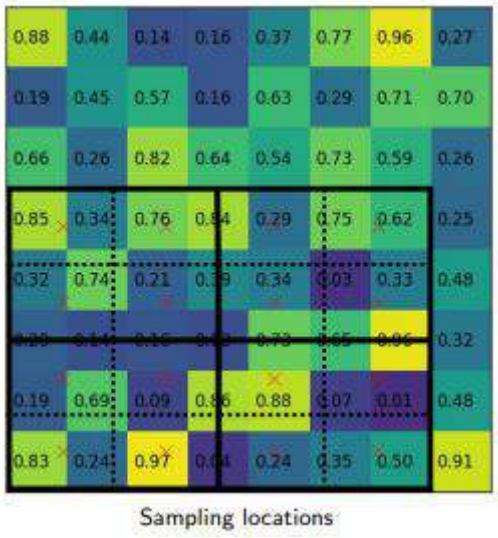
RoI Align

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:



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.

Applications of Mask R-CNN

Due to its additional capability to generate segmented masks, it is used in many computer vision applications such as:

- Human Pose Estimation
- Self Driving Car
- Drone Image Mapping etc.

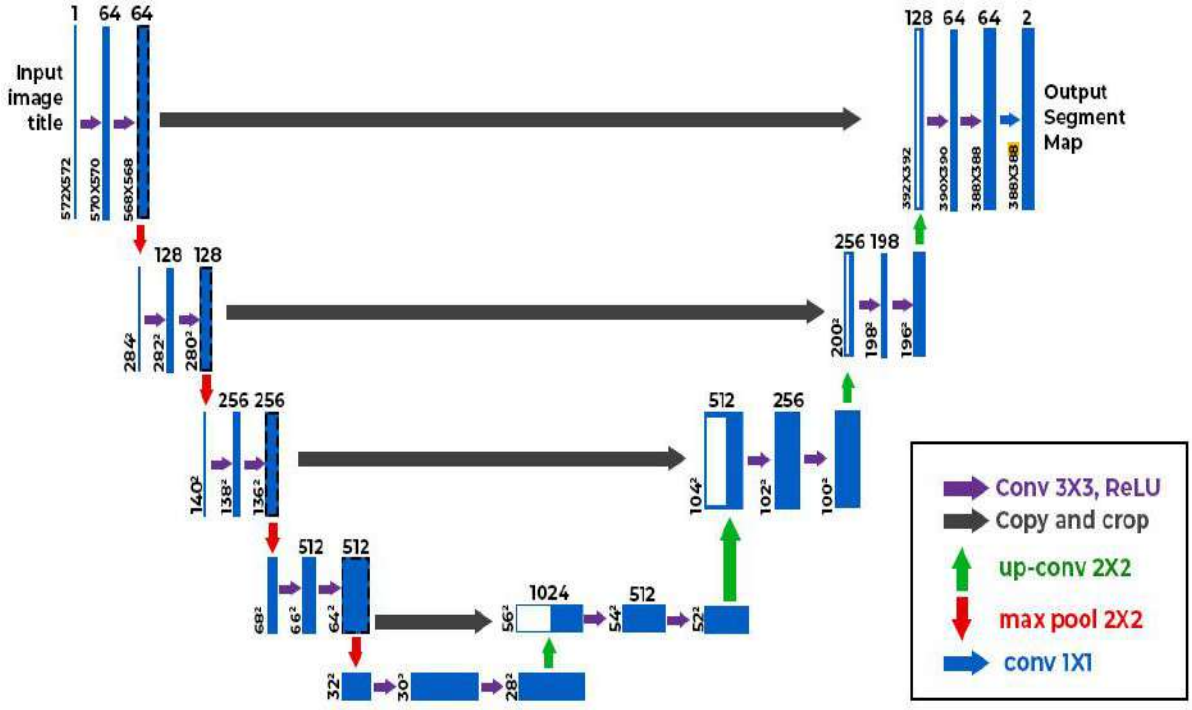
4. U-Net :

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:

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.

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.



U-Net Architecture

Figure 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.

Build the UNET Model:

implementation the U-Net architecture using Python 3 and the TensorFlow library.

The implementation can be divided into three parts.

First, we will define the encoder block used in the contraction path. This block consists of two 3×3 convolution layers followed by a ReLU activation layer and a 2×2 max pooling layer.

The second part is the decoder block, which takes the feature map from the lower layer, upconverts it, crops and concatenates it with the encoder data of the same level, and then performs two 3×3 convolution layers followed by ReLU activation.

The third part is defining the U-Net model using these blocks.

the code for the encoder block:

```
def encoder_block(inputs, num_filters):

# Convolution with 3x3 filter followed by ReLU activation
x = tf.keras.layers.Conv2D(num_filters,3,padding = 'valid')(inputs)
x = tf.keras.layers.Activation('relu')(x)

# Convolution with 3x3 filter followed by ReLU activation
x = tf.keras.layers.Conv2D(num_filters,3,padding = 'valid')(x)
x = tf.keras.layers.Activation('relu')(x)

# Max Pooling with 2x2 filter
x = tf.keras.layers.MaxPool2D(pool_size = (2, 2),strides = 2)(x)
return x
```

defining the decoder.

```
def decoder_block(inputs, skip_features, num_filters):
 # Upsampling with 2x2 filter
 x = tf.keras.layers.Conv2DTranspose(num_filters, (2, 2), strides = 2,
                                     padding = 'valid')(inputs)
 # Copy and crop the skip features
 # to match the shape of the upsampled input
 skip_features = tf.image.resize(skip_features, size = (x.shape[1], x.shape[2]))

 x = tf.keras.layers.Concatenate()([x, skip_features])

 # Convolution with 3x3 filter followed by ReLU activation
 x = tf.keras.layers.Conv2D(num_filters,3,padding = 'valid')(x)
 x = tf.keras.layers.Activation('relu')(x)

 # Convolution with 3x3 filter followed by ReLU activation
```

```
    x = tf.keras.layers.Conv2D(num_filters, 3, padding = 'valid')(x)
    x = tf.keras.layers.Activation('relu')(x)
    return x
```

Using these blocks and defining a U-Net model and printing model summary.

```
# Unet code
import tensorflow as tf

def unet_model(input_shape = (256, 256, 3), num_classes = 1):
    inputs = tf.keras.layers.Input(input_shape)

    # Contracting Path
    s1 = encoder_block(inputs, 64)
    s2 = encoder_block(s1, 128)
    s3 = encoder_block(s2, 256)
    s4 = encoder_block(s3, 512)

    # Bottleneck
    b1 = tf.keras.layers.Conv2D(1024, 3, padding = 'valid')(s4)
    b1 = tf.keras.layers.Activation('relu')(b1)
    b1 = tf.keras.layers.Conv2D(1024, 3, padding = 'valid')(b1)
    b1 = tf.keras.layers.Activation('relu')(b1)

    # Expansive Path
    s5 = decoder_block(b1, s4, 512)
    s6 = decoder_block(s5, s3, 256)
    s7 = decoder_block(s6, s2, 128)
    s8 = decoder_block(s7, s1, 64)

    # Output
    outputs = tf.keras.layers.Conv2D(num_classes,
                                     1,padding = 'valid',
                                     activation = 'sigmoid')(s8)

    model = tf.keras.models.Model(inputs = inputs,outputs = outputs,
                                  name = 'U-Net')
    return model

if __name__ == '__main__':
    model = unet_model(input_shape=(572, 572, 3), num_classes=2)
    model.summary()
```

5.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.

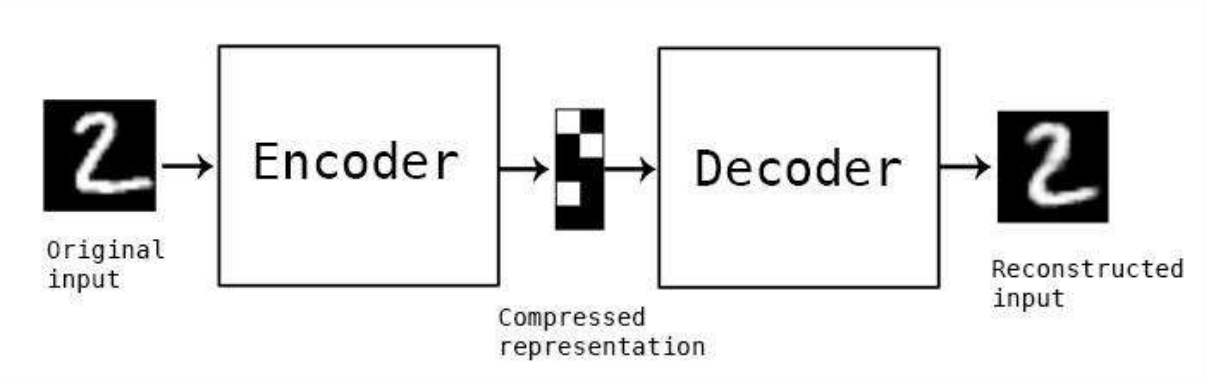
It is essential to understand that semantic segmentation classifies image pixels of one or more classes rather than real-world objects which are not semantically interpretable. Due to its intricate working scheme, it is a difficult task in the computer vision ecosystem as you classify each pixel instead of objects, which is the case in **object detection**.

Semantic Segmentation Reducing Noise with Auto encoders:

a. Creating a simple fully connected autoencoder,
b. creating a convolutional autoencoder,
c. Denoising images with autoencoders,
d. Spotting outliers using Autoencoders,
e. Variational Autoencoders,
f. creating an inverse image search index with deep learning,

g. Implementing a variational autoencoder.

a) What are autoencoders?



"Autoencoding" is a data compression algorithm where the compression and decompression functions are 1) data-specific, 2) lossy, and 3) *learned automatically from examples* rather than engineered by a human. Additionally, in almost all contexts where the term "autoencoder" is used, the compression and decompression functions are implemented with neural networks.

1) Autoencoders are data-specific, which means that they will only be able to compress data similar to what they have been trained on. This is different from, say, the MPEG-2 Audio Layer III (MP3) compression algorithm, which only holds assumptions about "sound" in general, but not about specific types of sounds. An autoencoder trained on pictures of faces would do a rather poor job of compressing pictures of trees, because the features it would learn would be face-specific.

2) Autoencoders are lossy, which means that the decompressed outputs will be degraded compared to the original inputs (similar to MP3 or JPEG compression). This differs from lossless arithmetic compression.

3) Autoencoders are learned automatically from data examples, which is a useful property: it means that it is easy to train specialized instances of the algorithm that will perform well on a specific type of input. It doesn't require any new engineering, just appropriate training data.

To build an autoencoder, you need three things:

- an encoding function,
- a decoding function,
- and a distance function between the amount of information loss between the compressed representation of your data and the decompressed representation (i.e. a "loss" function).

The encoder and decoder will be chosen to be parametric functions (typically neural networks), and to be differentiable with respect to the distance function, so the parameters of the encoding/decoding functions can be optimize to minimize the reconstruction loss, using Stochastic Gradient Descent. It's simple! And you don't even need to understand any of these words to start using autoencoders in practice.

Are they good at data compression?

Usually, not really. In picture compression for instance, it is pretty difficult to train an autoencoder that does a better job than a basic algorithm like JPEG, and typically the only way it can be achieved is by restricting yourself to a very specific type of picture (e.g. one for which JPEG does not do a good job). The fact that autoencoders are data-specific makes them generally impractical for real-world data compression problems: you can only use them on data that is similar to what they were trained on, and making them more general thus requires *lots* of training data. But future advances might change this, who knows.

What are autoencoders good for?

They are rarely used in practical applications. In 2012 they briefly found an application in greedy layer-wise pretraining for deep convolutional neural networks [1], but this quickly fell out of fashion as we started realizing that better random weight initialization schemes were sufficient for training deep networks from scratch. In 2014, batch normalization [2] started allowing for even deeper networks, and from late 2015 we could train arbitrarily deep networks from scratch using residual learning [3].

Today two interesting practical applications of autoencoders are **data denoising** (which we feature later in this post), and **dimensionality reduction for data visualization**. With appropriate dimensionality and sparsity constraints, autoencoders can learn data projections that are more interesting than PCA or other basic techniques.

For 2D visualization specifically, t-SNE (pronounced "tee-snee") is probably the best algorithm around, but it typically requires relatively low-dimensional data. So a good strategy for visualizing similarity relationships in high-dimensional data is to start by using an autoencoder to compress your data into a low-dimensional space (e.g. 32-dimensional), then use t-SNE for mapping the compressed data to a 2D plane.

Figure from Noroozi and Favaro (2016)

Fig. 1: What image representations do we learn by solving puzzles? Left: The image from which the tiles (marked with green lines) are extracted. Middle: A puzzle obtained by shuffling the tiles. Some tiles might be directly identifiable as object parts, but their identification is much more reliable once the correct ordering is found and the global figure emerges (Right).

Let's build the simplest possible autoencoder

We'll start simple, with a single fully-connected neural layer as

```
import keras
from keras import layers

# This is the size of our encoded representations
```

```
encoding_dim = 32  # 32 floats -> compression of factor 24.5,
assuming the input is 784 floats

# This is our input image
input_img = keras.Input(shape=(784,))
# "encoded" is the encoded representation of the input
encoded = layers.Dense(encoding_dim, activation='relu')(input_img)
# "decoded" is the lossy reconstruction of the input
decoded = layers.Dense(784, activation='sigmoid')(encoded)

# This model maps an input to its reconstruction
autoencoder = keras.Model(input_img, decoded)
```

Let's also create a separate encoder model:

```
# This model maps an input to its encoded representation
encoder = keras.Model(input_img, encoded)
```

As well as the decoder model:

```
# This is our encoded (32-dimensional) input
encoded_input = keras.Input(shape=(encoding_dim,))
# Retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# Create the decoder model
decoder = keras.Model(encoded_input, decoder_layer(encoded_input))
```

Now let's train our autoencoder to reconstruct MNIST digits.

First, we'll configure our model to use a per-pixel binary crossentropy loss, and the Adam optimizer:

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

Let's prepare our input data. We're using MNIST digits, and we're discarding the labels (since we're only interested in encoding/decoding the input images).

```
from keras.datasets import mnist
import numpy as np
(x_train, _), (x_test, _) = mnist.load_data()
```

We will normalize all values between 0 and 1 and we will flatten the 28x28 images into vectors of size 784.

```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
print(x_train.shape)
print(x_test.shape)
```

Now let's train our autoencoder for 50 epochs:

```
autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

After 50 epochs, the autoencoder seems to reach a stable train/validation loss value of about `0.09`. We can try to visualize the reconstructed inputs and the encoded representations. We will use Matplotlib.

```
# Encode and decode some digits
# Note that we take them from the *test* set
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)
# Use Matplotlib (don't ask)
import matplotlib.pyplot as plt

n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
```

```
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

Here's what we get. The top row is the original digits, and the bottom row is the reconstructed digits. We are losing quite a bit of detail with this basic approach.

Adding a sparsity constraint on the encoded representations

In the previous example, the representations were only constrained by the size of the hidden layer (32). In such a situation, what typically happens is that the hidden layer is learning an approximation of PCA (principal component analysis). But another way to constrain the representations to be compact is to add a sparsity contraint on the activity of the hidden representations, so fewer units would "fire" at a given time. In Keras, this can be done by adding an activity_regularizer to our Dense layer:

```
from keras import regularizers

encoding_dim = 32

input_img = keras.Input(shape=(784,))
# Add a Dense layer with a L1 activity regularizer
encoded = layers.Dense(encoding_dim, activation='relu',
                activity_regularizer=regularizers.l1(10e-5))(input_img)
decoded = layers.Dense(784, activation='sigmoid')(encoded)

autoencoder = keras.Model(input_img, decoded)
```

Let's train this model for 100 epochs (with the added regularization the model is less likely to overfit and can be trained longer). The models ends with a train loss of `0.11` and test loss of `0.10`. The difference between the two is mostly due to the regularization term being added to the loss during training (worth about 0.01).

Here's a visualization of our new results:

They look pretty similar to the previous model, the only significant difference being the sparsity of the encoded representations. `encoded_imgs.mean()` yields a value `3.33` (over our 10,000 test images), whereas with the previous model the same quantity was `7.30`. So our new model yields encoded representations that are twice sparser.

Deep autoencoder

We do not have to limit ourselves to a single layer as encoder or decoder, we could instead use a stack of layers, such as:

```
input_img = keras.Input(shape=(784,))
encoded = layers.Dense(128, activation='relu')(input_img)
encoded = layers.Dense(64, activation='relu')(encoded)
encoded = layers.Dense(32, activation='relu')(encoded)

decoded = layers.Dense(64, activation='relu')(encoded)
decoded = layers.Dense(128, activation='relu')(decoded)
decoded = layers.Dense(784, activation='sigmoid')(decoded)
```

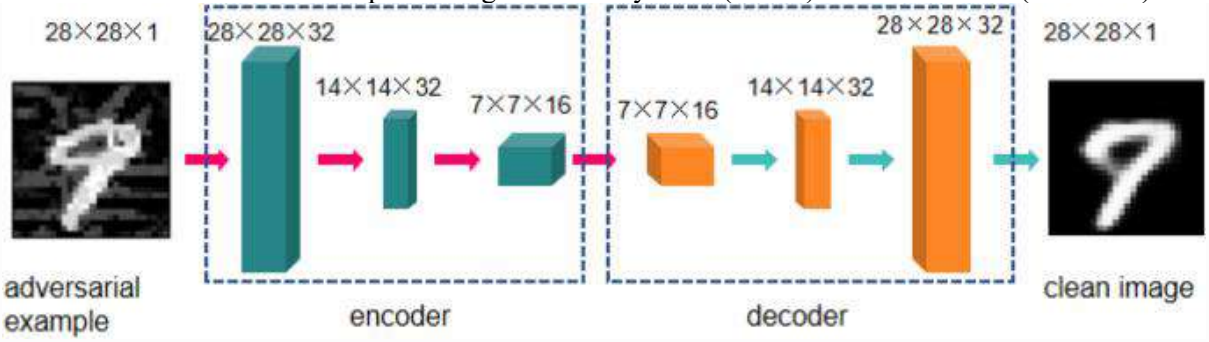
Let's try this:

```
autoencoder = keras.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

autoencoder.fit(x_train, x_train,
                epochs=100,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

After 100 epochs, it reaches a train and validation loss of ~0.08, a bit better than our previous models. Our reconstructed digits look a bit better too:

b. Convolutional autoencoder

Since our inputs are images, it makes sense to use convolutional neural networks (convnets) as encoders and decoders. In practical settings, autoencoders applied to images are always convolutional autoencoders --they simply perform much better.



Let's implement one. The encoder will consist in a stack of `Conv2D` and `MaxPooling2D` layers (max pooling being used for spatial down-sampling), while the decoder will consist in a stack of `Conv2D` and `UpSampling2D` layers.

```
import keras
from keras import layers

input_img = keras.Input(shape=(28, 28, 1))

x = layers.Conv2D(16, (3, 3), activation='relu', padding='same')(input_img)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = layers.MaxPooling2D((2, 2), padding='same')(x)

# at this point the representation is (4, 4, 8) i.e. 128-dimensional

x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = layers.UpSampling2D((2, 2))(x)
x = layers.Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = layers.UpSampling2D((2, 2))(x)
x = layers.Conv2D(16, (3, 3), activation='relu')(x)
x = layers.UpSampling2D((2, 2))(x)
decoded = layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = keras.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

To train it, we will use the original MNIST digits with shape `(samples, 3, 28, 28)`, and we will just normalize pixel values between 0 and 1.

```
from keras.datasets import mnist
import numpy as np

(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))
```

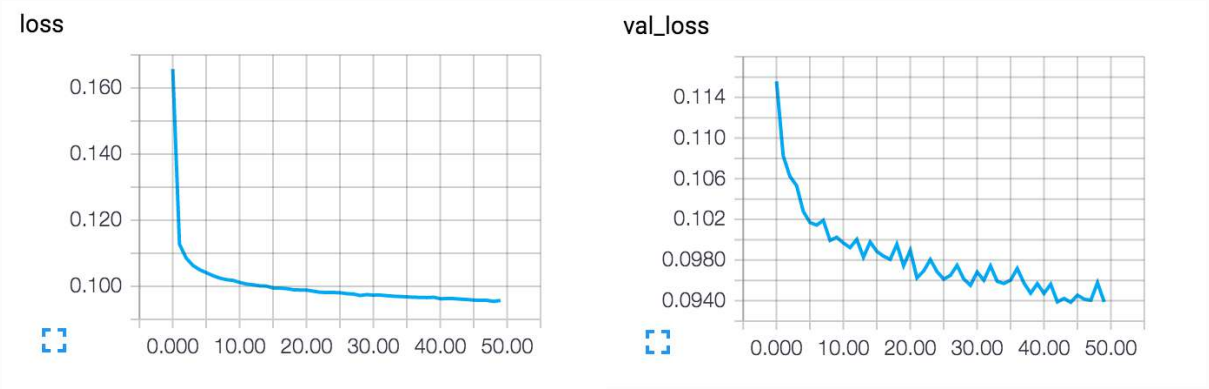
Let's train this model for 50 epochs. For the sake of demonstrating how to visualize the results of a model during training, we will be using the TensorFlow backend and the TensorBoard callback.
First, let's open up a terminal and start a TensorBoard server that will read logs stored at `/tmp/autoencoder`.

```
tensorboard --logdir=/tmp/autoencoder
```

Then let's train our model. In the `callbacks` list we pass an instance of the `TensorBoard` callback. After every epoch, this callback will write logs to `/tmp/autoencoder`, which can be read by our TensorBoard server.

```
from keras.callbacks import TensorBoard

autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=128,
                shuffle=True,
                validation_data=(x_test, x_test),
                callbacks=[TensorBoard(log_dir='/tmp/autoencoder')])
```

This allows us to monitor training in the TensorBoard web interface (by navighating to `http://0.0.0.0:6006`):



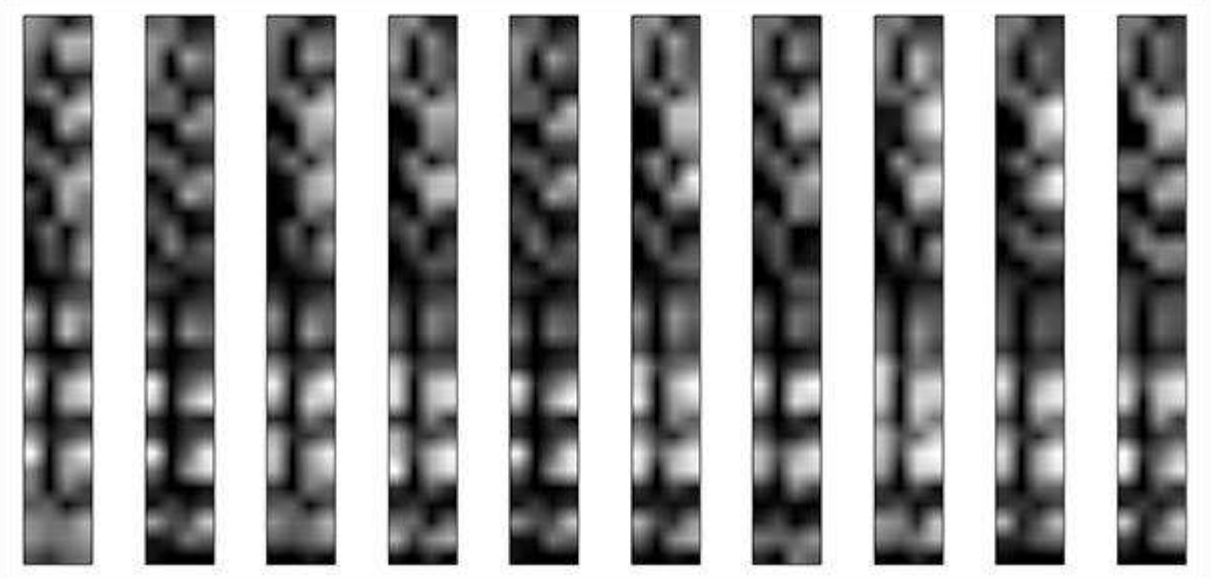
The model converges to a loss of 0.094, significantly better than our previous models (this is in large part due to the higher entropic capacity of the encoded representation, 128 dimensions vs. 32 previously). Let's take a look at the reconstructed digits:

```
decoded_imgs = autoencoder.predict(x_test)

n = 10
plt.figure(figsize=(20, 4))
for i in range(1, n + 1):
    # Display original
    ax = plt.subplot(2, n, i)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(2, n, i + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

We can also have a look at the 128-dimensional encoded representations. These representations are 8x4x4, so we reshape them to 4x32 in order to be able to display them as grayscale images.

```
encoder = keras.Model(input_img, encoded)
encoded_imgs = encoder.predict(x_test)
```

```
n = 10
plt.figure(figsize=(20, 8))
for i in range(1, n + 1):
    ax = plt.subplot(1, n, i)
    plt.imshow(encoded_imgs[i].reshape((4, 4 * 8)).T)
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```



---

c. Application to image denoising

Let's put our convolutional autoencoder to work on an image denoising problem. It's simple: we will train the autoencoder to map noisy digits images to clean digits images.

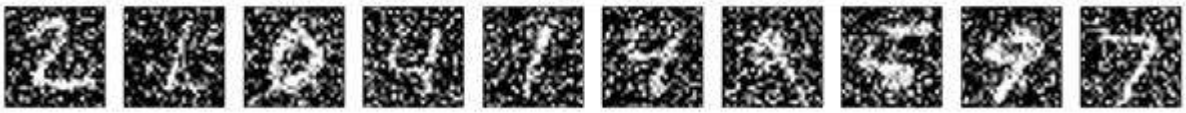
Here's how we will generate synthetic noisy digits: we just apply a gaussian noise matrix and clip the images between 0 and 1.

```
from keras.datasets import mnist
import numpy as np

(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))

noise_factor = 0.5
x_train_noisy = x_train + noise_factor * np.random.normal(loc=0.0, scale=1.0,
size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.normal(loc=0.0, scale=1.0,
size=x_test.shape)

x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
```

Here's what the noisy digits look like:

```
n = 10
plt.figure(figsize=(20, 2))
for i in range(1, n + 1):
    ax = plt.subplot(1, n, i)
    plt.imshow(x_test_noisy[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
```

```
    ax.get_yaxis().set_visible(False)
plt.show()
```



If you squint you can still recognize them, but barely. Can our autoencoder learn to recover the original digits? Let's find out.

Compared to the previous convolutional autoencoder, in order to improve the quality of the reconstructed, we'll use a slightly different model with more filters per layer:

```
input_img = keras.Input(shape=(28, 28, 1))

x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = layers.MaxPooling2D((2, 2), padding='same')(x)
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)
encoded = layers.MaxPooling2D((2, 2), padding='same')(x)

# At this point the representation is (7, 7, 32)

x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(encoded)
x = layers.UpSampling2D((2, 2))(x)
x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = layers.UpSampling2D((2, 2))(x)
decoded = layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = keras.Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

Let's train it for 100 epochs:

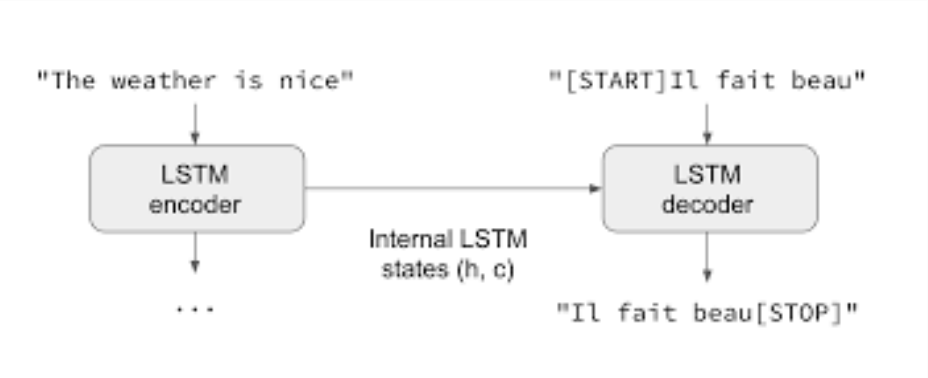
```
autoencoder.fit(x_train_noisy, x_train,
                epochs=100,
                batch_size=128,
                shuffle=True,
                validation_data=(x_test_noisy, x_test),
                callbacks=[TensorBoard(log_dir='/tmp/tb',
histogram_freq=0, write_graph=False)])
```

Now let's take a look at the results. Top, the noisy digits fed to the network, and bottom, the digits are reconstructed by the network.

It seems to work pretty well. If you scale this process to a bigger convnet, you can start building document denoising or audio denoising models.

d.Sequence-to-sequence autoencoder

If you inputs are sequences, rather than vectors or 2D images, then you may want to use as encoder and decoder a type of model that can capture temporal structure, such as a LSTM. To build a LSTM-based autoencoder, first use a LSTM encoder to turn your input sequences into a single vector that contains information about the entire sequence, then repeat this vector `n` times (where `n` is the number of timesteps in the output sequence), and run a LSTM decoder to turn this constant sequence into the target sequence.

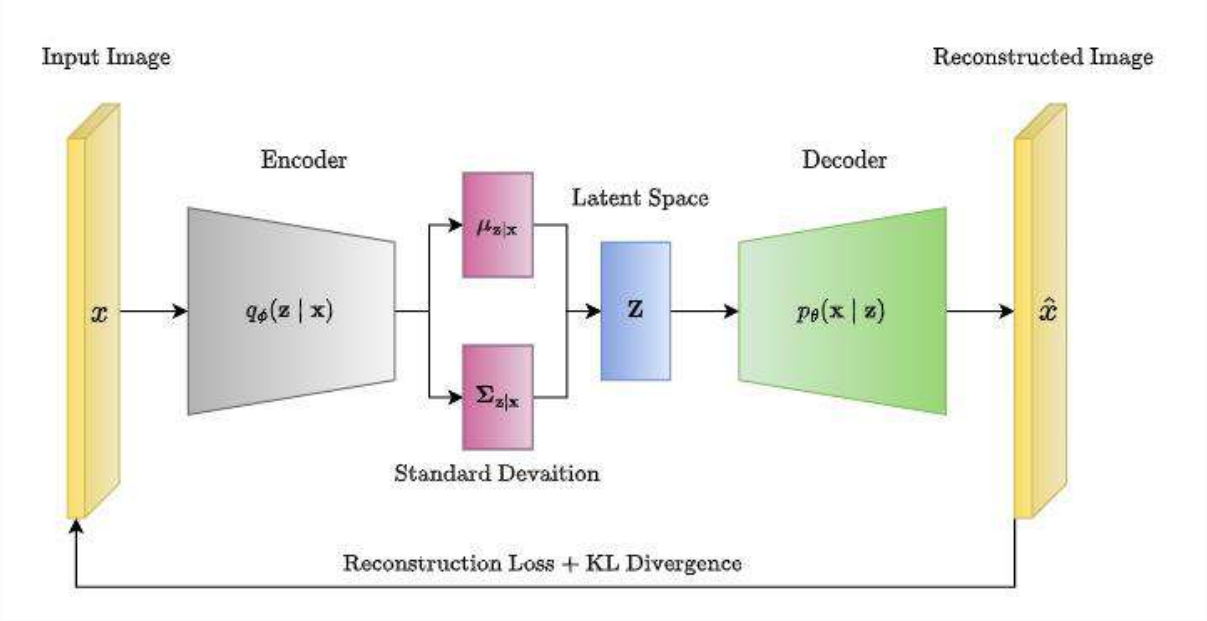



```
timesteps = ...  # Length of your sequences
input_dim = ...
latent_dim = ...

inputs = keras.Input(shape=(timesteps, input_dim))
encoded = layers.LSTM(latent_dim)(inputs)

decoded = layers.RepeatVector(timesteps)(encoded)
decoded = layers.LSTM(input_dim, return_sequences=True)(decoded)

sequence_autoencoder = keras.Model(inputs, decoded)
encoder = keras.Model(inputs, encoded)
```

e. Variational autoencoder (VAE)

Variational autoencoders are a slightly more modern and interesting take on autoencoding.

What is a variational autoencoder, is a type of autoencoder with added constraints on the encoded representations being learned. More precisely, it is an autoencoder that learns a latent variable model for its input data. So instead of letting your neural network learn an arbitrary function, you are learning the parameters of a probability distribution modeling your data. If you sample points from this distribution, you can generate new input data samples: a VAE is a "generative model".



How does a variational autoencoder work?

First, an encoder network turns the input samples x into two parameters in a latent space, which we will note z_mean and z_log_sigma. Then, we randomly sample similar points z from the latent normal distribution that is assumed to generate the data, via z = z_mean + exp(z_log_sigma) * epsilon, where epsilon is a random normal tensor. Finally, a decoder network maps these latent space points back to the original input data.

The parameters of the model are trained via two loss functions: a reconstruction loss forcing the decoded samples to match the initial inputs (just like in our previous autoencoders), and the KL divergence between the learned latent distribution and the prior distribution, acting as a regularization term. You could actually get rid of this latter term entirely, although it does help in learning well-formed latent spaces and reducing overfitting to the training data.

Because a VAE is a more complex example, we have made the code available on Github as a standalone script. Here we will review step by step how the model is created.
First, here's our encoder network, mapping inputs to our latent distribution parameters:

```
original_dim = 28 * 28
intermediate_dim = 64
latent_dim = 2

inputs = keras.Input(shape=(original_dim,))
h = layers.Dense(intermediate_dim, activation='relu')(inputs)
z_mean = layers.Dense(latent_dim)(h)
z_log_sigma = layers.Dense(latent_dim)(h)
```

We can use these parameters to sample new similar points from the latent space:

```
from keras import backend as K

def sampling(args):
    z_mean, z_log_sigma = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim),
```

```
                              mean=0., stddev=0.1)
    return z_mean + K.exp(z_log_sigma) * epsilon

z = layers.Lambda(sampling)([z_mean, z_log_sigma])
```

Finally, we can map these sampled latent points back to reconstructed inputs:

```
# Create encoder
encoder = keras.Model(inputs, [z_mean, z_log_sigma, z], name='encoder')

# Create decoder
latent_inputs = keras.Input(shape=(latent_dim,), name='z_sampling')
x = layers.Dense(intermediate_dim, activation='relu')(latent_inputs)
outputs = layers.Dense(original_dim, activation='sigmoid')(x)
decoder = keras.Model(latent_inputs, outputs, name='decoder')

# instantiate VAE model
outputs = decoder(encoder(inputs)[2])
vae = keras.Model(inputs, outputs, name='vae_mlp')
```

What we've done so far allows us to instantiate 3 models:

- an end-to-end autoencoder mapping inputs to reconstructions
- an encoder mapping inputs to the latent space
- a generator that can take points on the latent space and will output the corresponding reconstructed samples.

We train the model using the end-to-end model, with a custom loss function: the sum of a reconstruction term, and the KL divergence regularization term.

```
reconstruction_loss = keras.losses.binary_crossentropy(inputs, outputs)
reconstruction_loss *= original_dim
kl_loss = 1 + z_log_sigma - K.square(z_mean) - K.exp(z_log_sigma)
kl_loss = K.sum(kl_loss, axis=-1)
kl_loss *= -0.5
vae_loss = K.mean(reconstruction_loss + kl_loss)
vae.add_loss(vae_loss)
vae.compile(optimizer='adam')
```

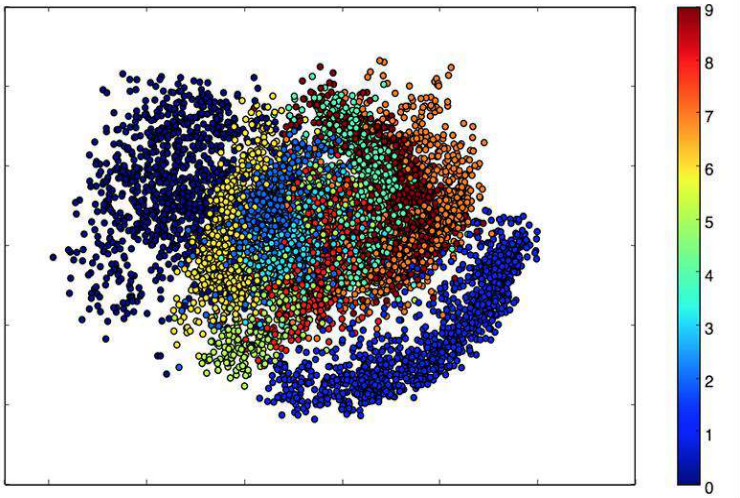
We train our VAE on MNIST digits:

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

vae.fit(x_train, x_train,
        epochs=100,
        batch_size=32,
        validation_data=(x_test, x_test))
```

Because our latent space is two-dimensional, there are a few cool visualizations that can be done at this point. One is to look at the neighborhoods of different classes on the latent 2D plane:

```
```

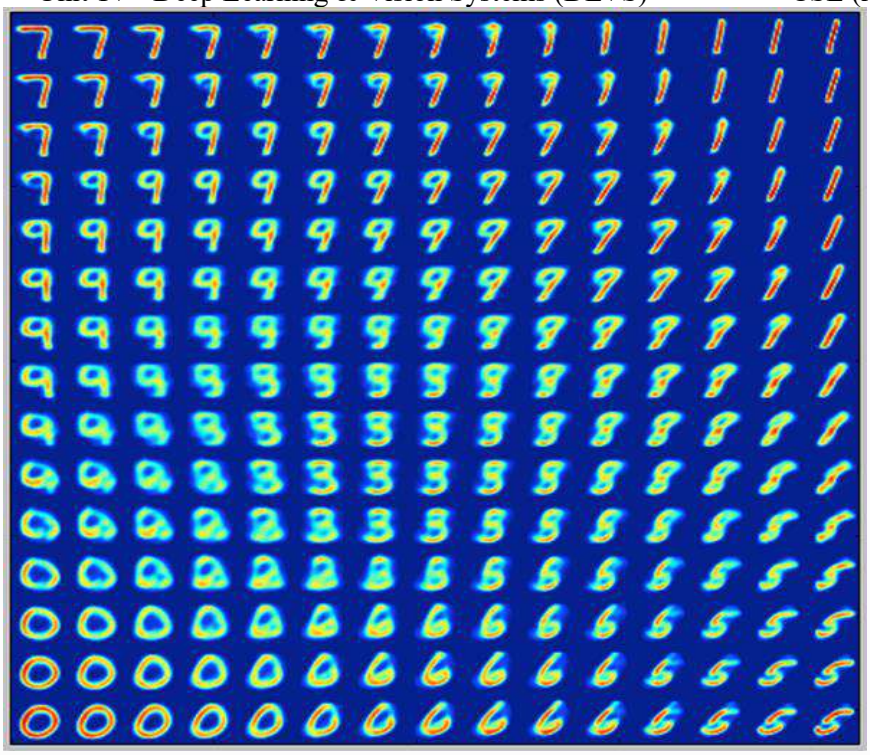
```
x_test_encoded = encoder.predict(x_test, batch_size=batch_size)
plt.figure(figsize=(6, 6))
plt.scatter(x_test_encoded[:, 0], x_test_encoded[:, 1], c=y_test)
plt.colorbar()
plt.show()
```



Each of these colored clusters is a type of digit. Close clusters are digits that are structurally similar (i.e. digits that share information in the latent space).

Because the VAE is a generative model, we can also use it to generate new digits! Here we will scan the latent plane, sampling latent points at regular intervals, and generating the corresponding digit for each of these points. This gives us a visualization of the latent manifold that "generates" the MNIST digits.

```
# Display a 2D manifold of the digits
n = 15  # figure with 15x15 digits
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
# We will sample n points within [-15, 15] standard deviations
grid_x = np.linspace(-15, 15, n)
grid_y = np.linspace(-15, 15, n)

for i, yi in enumerate(grid_x):
    for j, xi in enumerate(grid_y):
        z_sample = np.array([[xi, yi]])
        x_decoded = decoder.predict(z_sample)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size,
               j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(10, 10))
plt.imshow(figure)
plt.show()
```



10. Open-source datasets:

- we worked with some open source datasets like MNIST, CIFAR, and ImageNet.
- When applying transfer learning, you might need to do a little research to use a dataset that is close enough to your domain.

MNIST

- MNIST stands for “Modified National Institute of Standards and Technology”.
- It contains labeled handwritten images of digits from 0 to 9.
- The goal of this dataset is to classify handwritten digits. MNIST was popularly used by the research community to benchmark their classification algorithms.
- In fact, it is considered as the “hello, world!” of image datasets. Nowadays, MNIST dataset is pretty simple and a simple CNN can achieve more than 99% accuracy so it is not considered a benchmark for CNN performance anymore.
- MNIST consists of 60,000 training images and 10,000 test images. All are grayscale images (i.e. 1 channel). Each image has 28 pixel height and 28 pixel width. Figure below shows a sample image of the MNIST dataset.

KR

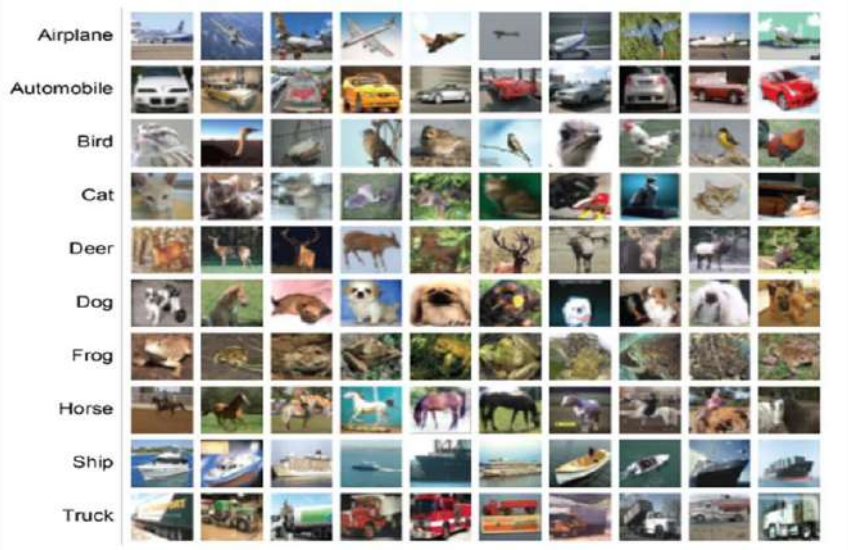
Fashion MNIST dataset

- The data is stored in the same format as the original MNIST dataset. It was made with the intention to replace the MNIST dataset because it became too simple for modern convolutional networks.
- Instead of handwritten digits, it contains 60,000 training images and 10,000 test images of 10 fashion clothing classes.
- The 10 supported classes are: t-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot.
- Below is a sample of the represented classes in this dataset.

CIFAR-10

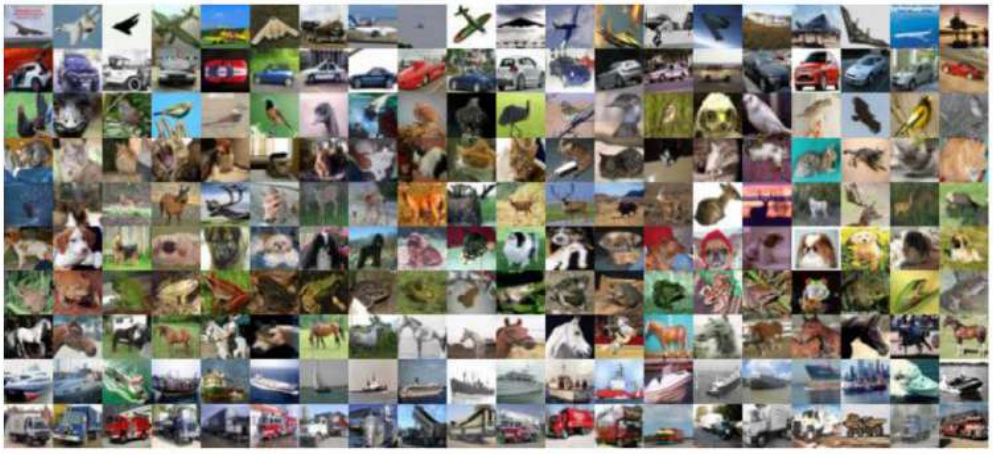
- CIFAR is considered as another benchmark dataset for image classification in the computer vision and machine learning literature.
- CIFAR images are more complex than MNIST in the sense that MNIST are all grayscale images and their objects are perfectly centered in the middle of the image while CIFAR's images are all colored (3 channels) with dramatic variance of how the objects appear in the image.
- The CIFAR-10 dataset consists of 60,000 32x32 color images in 10 classes, with 6000 images per class. There are 50,000 training images and 10,000 test images.
- CIFAR-100 is the bigger brother of CIFAR-10. It contains 100 classes with 600 images each. These 100 classes are grouped into 20 superclasses. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs)

KR



ImageNet

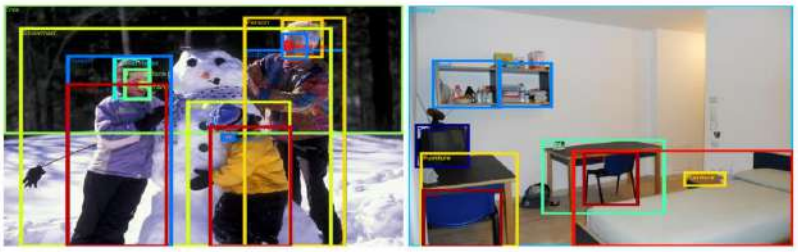
- ImageNet is considered the current benchmark that is widely used by computer vision researchers to evaluate their classification algorithms.
- ImageNet is a large visual database designed for use in visual object recognition software research. It is aimed at labeling and categorizing images into almost 22,000 categories based on a defined set of words and phrases. The images were collected from the web and labeled by human labelers using Amazon's Mechanical Turk crowdsourcing tool.



Microsoft's COCO

MS COCO is short for "Microsoft Common Objects in Context" that was created by Microsoft. It is an open-source database that aims to enable future research for object detection, instance segmentation, image captioning, and person keypoints localization. It contains 328K images. More than 200K of them are labeled, 2.5 million object instances, and 80 object categories that would be easily recognizable by a 4 year old. Read the original research paper that was provided by the creators of the dataset. It describes the motivation and content of this dataset very well.

Google's Open Images

- Open Images is an open-source image database that was created by Google. It contains more than 9 million images as of the writing of this chapter.
- What makes it stand out is that these images are mostly of complex scenes that span thousands of classes of objects. Additionally, more than 2 million of these images are hand-annotated with bounding boxes making Open Images by far the largest existing dataset with object location annotations. In this subset of images, there are ~15.4 million bounding boxes of 600 classes of object.
- Similar to ImageNet and ILSVRC, Open Images has a challenge called "Open Images Challenge".
- 

to build a smaller SSD network called SSD7.SSD7 is a seven-layer version of the SSD300 network. It is important to note that while an SSD7 network would yield some acceptable results, this is not an optimized network architecture. The goal is just to build a low-complexity network that is fast enough for you to train on your personal computer. It took me around 20 hours to train this network on the road traffic dataset; training could take a lot less time on a GPU.

The original repository created by Pierluigi Ferrari comes with implementation tutorials for SSD7, SSD300, and SSD512 networks. I encourage you to check it out.

In this project, we will use a toy dataset created by Udacity. You can visit Udacity's GitHub repository for more information on the dataset (https://github.com/udacity/self-driving-car/tree/master/annotations). It has more than 22,000 labeled images and 5 object classes: car, truck, pedestrian, bicyclist, and traffic light. All of the images have been resized to a height of 300 pixels and a width of 480 pixels. You can download the dataset as part of the book's code.

What makes this dataset very interesting is that these are real-time images taken while driving in Mountain View, California, and neighboring cities during daylight conditions. No image cleanup was done. Take a look at the image examples in figure 7.31.

Fig: Example images from the Udacity self-driving dataset (Image copyright © 2016 Udacity and published under MIT License.)

As stated on Udacity's page, the dataset was labeled by CrowdAI and Autti. You can find the labels in CSV format in the folder, split into three files: training, validation, and test datasets. The labeling format is straightforward, as follows:

frame	xmin	xmax	ymin	ymax	class_id
1478019952686311006.jpg	237	251	143	155	1

Xmin, xmax, ymin, and ymax are the bounding box coordinates. Class_id is the correct label, and frame is the image name.

Data annotation using LabelImg

If you are annotating your own data, there are several open source labeling applications that you can use, like LabelImg (https://pypi.org/project/labelImg). They are very easy to set up and use.

Example of using the labelImg application to annotate images

Step 1: Build the model

Before jumping into the model training, take a close look at the build_model method in the keras_ssd7.py file. This file builds a Keras model with the SSD architecture. As we learned earlier in this chapter, the model consists of convolutional feature layers and a number of convolutional predictor layers that make their input from different feature layers.

Here is what the build_model method looks like. Please read the comments in the keras_ssd7.py file to understand the arguments passed:

```
def build_model(image_size,
                mode='training',
```

```
l2_regularization=0.0,
min_scale=0.1,
max_scale=0.9,
scales=None,
aspect_ratios_global=[0.5, 1.0, 2.0],
aspect_ratios_per_layer=None,
two_boxes_for_ar1=True,
clip_boxes=False,
variances=[1.0, 1.0, 1.0, 1.0],
coords='centroids',
normalize_coords=False,
subtract_mean=None,
divide_by_stddev=None,
swap_channels=False,
confidence_thresh=0.01,
iou_threshold=0.45,
top_k=200,
nms_max_output_size=400,
return_predictor_sizes=False)
```

Step 2: Model configuration

In this section, we set the model configuration parameters. First we set the height, width, and number of color channels to whatever we want the model to accept as image input. If your input images have a different size than defined here, or if your images have non-uniform size, you must use the data generator's image transformations (resizing and/or cropping) so that your images end up having the ***required input size before they are fed to the model:***

img_height = 300

img_width = 480

img_channels = 3

intensity_mean = 127.5

intensity_range = 127.5

The number of classes is the number of positive classes in your dataset: for example, 20 for PASCAL VOC or 80 for COCO. Class ID 0 must always be reserved for the back ground class:

n_classes = 5

scales = [0.08, 0.16, 0.32, 0.64, 0.96]

aspect_ratios = [0.5, 1.0, 2.0]

steps = None

offsets = None

two_boxes_for_ar1 = True

clip_boxes = False

variances = [1.0, 1.0, 1.0, 1.0]

normalize_coords = True

Step 3: Create the model

Now we call the build_model() function to build our model:

model = build_model(image_size=(img_height, img_width, img_channels),

n_classes=n_classes,

mode='training',

l2_regularization=0.0005,

scales=scales,

aspect_ratios_global=aspect_ratios,

aspect_ratios_per_layer=None,

two_boxes_for_ar1=two_boxes_for_ar1,

steps=steps,

offsets=offsets,

clip_boxes=clip_boxes,

variances=variances,

normalize_coords=normalize_coords,

subtract_mean=intensity_mean,

divide_by_stddev=intensity_range)

You can optionally load saved weights. If you don't want to load weights, skip the following code snippet:

model.load_weights('<path/to/model.h5>', by_name=True)

Instantiate an Adam optimizer and the SSD loss function, and compile the model.

Here, we will use a custom Keras function called SSDLoss. It implements the multi-task log loss for classification and smooth L1 loss for localization. neg_pos_ratio and alpha are set as in the SSD paper (Liu et al., 2016):

adam = Adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)

ssd_loss = SSDLoss(neg_pos_ratio=3, alpha=1.0)

model.compile(optimizer=adam, loss=ssd_loss.compute_loss)

Step 4: Load the data

To load the data, follow these steps:

1 Instantiate two DataGenerator objects—one for training and one for validation:

train_dataset = DataGenerator(load_images_into_memory=False, hdf5_dataset_path=None)

val_dataset = DataGenerator(load_images_into_memory=False, hdf5_dataset_path=None)

2 Parse the image and label lists for the training and validation datasets:

```
images_dir = 'path_to_downloaded_directory'
train_labels_filename = 'path_to_dataset/labels_train.csv'
val_labels_filename = 'path_to_dataset/labels_val.csv'
train_dataset.parse_csv(images_dir=images_dir,
                        labels_filename=train_labels_filename,
                        input_format=['image_name', 'xmin', 'xmax', 'ymin',
                        'ymax', 'class_id'],
                        include_classes='all')
val_dataset.parse_csv(images_dir=images_dir,
                      labels_filename=val_labels_filename,
                      input_format=['image_name', 'xmin', 'xmax', 'ymin',
                      'ymax', 'class_id'],
                      include_classes='all')
train_dataset_size = train_dataset.get_dataset_size()
val_dataset_size = val_dataset.get_dataset_size()
print("Number of images in the training
    dataset:\t{:>6}".format(train_dataset_size))
print("Number of images in the validation
    dataset:\t{:>6}".format(val_dataset_size))
```

This cell should print out the size of your training and validation datasets as follows:

Number of images in the training dataset: 18000

Number of images in the validation dataset: 4241

3 Set the batch size:

```
batch_size = 16
```

As you learned in chapter 4, you can increase the batch size to get a boost in the computing speed based on the hardware that you are using for this training.

4. Define the data augmentation process:

```
data_augmentation_chain = DataAugmentationConstantInputSize(
                          random_brightness=(-48, 48, 0.5),
                          random_contrast=(0.5, 1.8, 0.5),
                          random_saturation=(0.5, 1.8, 0.5),
```

random_hue=(18, 0.5),

random_flip=0.5,

random_translate=((0.03,0.5),

(0.03,0.5), 0.5),

random_scale=(0.5, 2.0, 0.5),

n_trials_max=3,

clip_boxes=True,

overlap_criterion='area',

bounds_box_filter=(0.3, 1.0),

bounds_validator=(0.5, 1.0),

n_boxes_min=1,

background=(0,0,0))

5 Instantiate an encoder that can encode ground-truth labels into the format needed by the SSD loss function. Here, the encoder constructor needs the spatial dimensions of the model's predictor layers to create the anchor boxes:

```
predictor_sizes = [model.get_layer('classes4').output_shape[1:3],
                   model.get_layer('classes5').output_shape[1:3],
                   model.get_layer('classes6').output_shape[1:3],
                   model.get_layer('classes7').output_shape[1:3]]
ssd_input_encoder = SSDInputEncoder(img_height=img_height,
                                    img_width=img_width,
                                    n_classes=n_classes,
                                    predictor_sizes=predictor_sizes,
                                    scales=scales,
                                    aspect_ratios_global=aspect_ratios,
                                    two_boxes_for_ar1=two_boxes_for_ar1,
                                    steps=steps,
                                    offsets=offsets,
                                    clip_boxes=clip_boxes,
                                    variances=variances,
                                    matching_type='multi',
                                    pos_iou_threshold=0.5,
                                    neg_iou_limit=0.3,
                                    normalize_coords=normalize_coords)
```

6 Create the generator handles that will be passed to Keras's fit_generator() function:

```
train_generator = train_dataset.generate(batch_size=batch_size,
                                         shuffle=True,
                                         transformations=[
                                         data_augmentation_chain],
                                         label_encoder=ssd_input_encoder,
                                         returns={'processed_images',
                                         'encoded_labels'},
                                         keep_images_without_gt=False)
                                         val_generator =
                                         val_dataset.generate(batch_size=batch_size,
                                         shuffle=False,
                                         transformations=[],
                                         label_encoder=ssd_input_encoder,
                                         returns={'processed_images',
                                         'encoded_labels'},
                                         keep_images_without_gt=False)
```

Step 5: Train the model

Everything is set, and we are ready to train our SSD7 network. We've already chosen an optimizer and a learning rate and set the batch size; now let's set the remaining training parameters and train the network. There are no new parameters here that you haven't learned already. We will set the model checkpoint, early stopping, and learning rate reduction rate:

```
model_checkpoint =
ModelCheckpoint(filepath='ssd7_epoch-{epoch:02d}_loss-{loss:.4f}_val_loss-
        {val_loss:.4f}.h5',
                                         monitor='val_loss',
                                         verbose=1,
                                         save_best_only=True,
                                         save_weights_only=False,
                                         mode='auto',
                                         period=1)
csv_logger = CSVLogger(filename='ssd7_training_log.csv',
                                 separator=',',
                                 append=True)
```

```
early_stopping = EarlyStopping(monitor='val_loss',
                               min_delta=0.0,
                               patience=10,
                               verbose=1)
reduce_learning_rate = ReduceLROnPlateau(monitor='val_loss',
                                         factor=0.2,
                                         patience=8,
                                         verbose=1,
                                         epsilon=0.001,
                                         cooldown=0,
                                         min_lr=0.00001)
callbacks = [model_checkpoint, csv_logger, early_stopping, reduce_learning_rate]
```

Set one epoch to consist of 1,000 training steps. I've arbitrarily set the number of epochs to 20 here. This does not necessarily mean that 20,000 training steps is the optimum number. Depending on the model, dataset, learning rate, and so on, you might have to train much longer (or less) to achieve convergence:

```
initial_epoch = 0
final_epoch = 20
steps_per_epoch = 1000
history = model.fit_generator(generator=train_generator,
                              steps_per_epoch=steps_per_epoch,
                              epochs=final_epoch,
                              callbacks=callbacks,
                              validation_data=val_generator,
                              validation_steps=ceil(
                              val_dataset_size/batch_size),
                              initial_epoch=initial_epoch)
```

Step 6: Visualize the loss

Let's visualize the loss and val_loss values to look at how the training and validation loss evolved and check whether our training is going in the right direction (figure 7.32):

```
plt.figure(figsize=(20,12))
plt.plot(history.history['loss'], label='loss')
plt.plot(history.history['val_loss'], label='val_loss')
plt.legend(loc='upper right', prop={'size': 24})
```

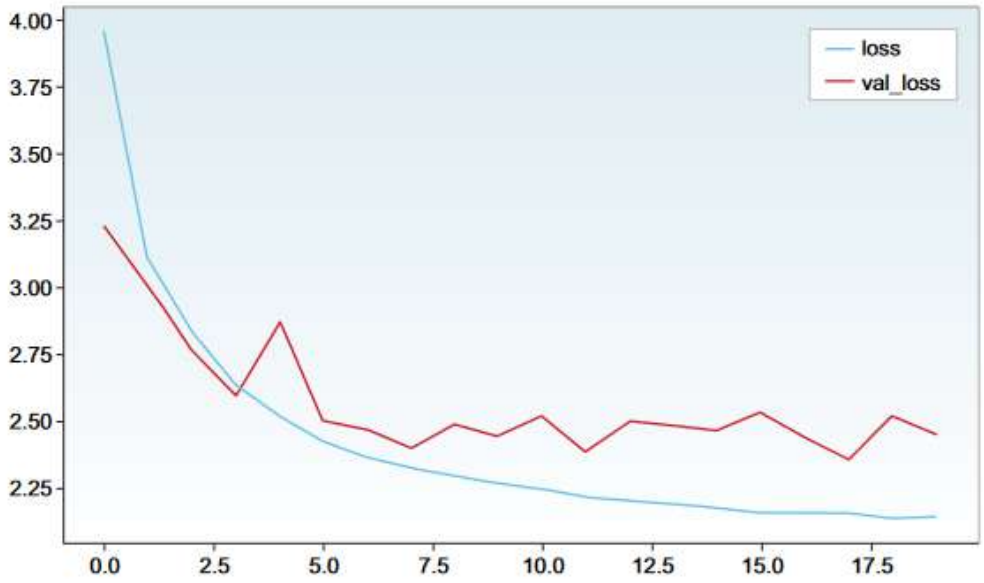


Figure : Visualized loss and val_loss values during SSD7 training for 20 epochs

Step 7: Make predictions

Now let's make some predictions on the validation dataset with the trained model. For convenience, we'll use the validation generator that we've already set up. Feel free to change the batch size:

```
predict_generator = val_dataset.generate(batch_size=1,
                                         shuffle=True,
                                         transformations=[],
                                         label_encoder=None,
                                         returns={'processed_images',
                                         'processed_labels',
                                         'filenames'},
                                         keep_images_without_gt=False)
batch_images, batch_labels, batch_filenames = next(predict_generator)
y_pred = model.predict(batch_images)
y_pred_decoded = decode_detections(y_pred,
                                   confidence_thresh=0.5,
                                   iou_threshold=0.45,
                                   top_k=200,
                                   normalize_coords=normalize_coords,
                                   img_height=img_height,
                                   img_width=img_width)
```

np.set_printoptions(precision=2, suppress=True, linewidth=90)

print("Predicted boxes:\n")

print(' class conf xmin ymin xmax ymax')

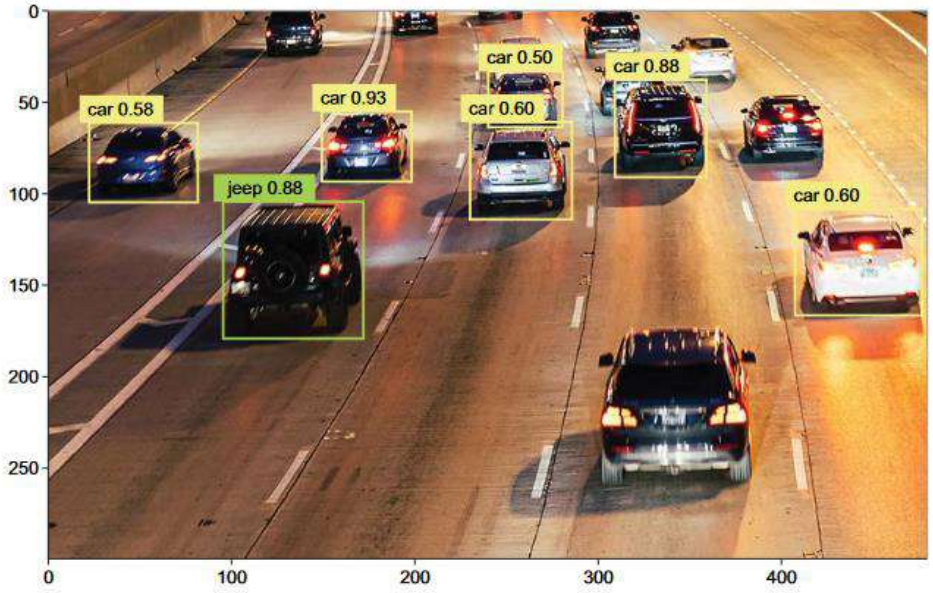
print(y_pred_decoded[i])

This code snippet prints the predicted bounding boxes along with their class and the level of confidence for each one, as shown in figure

```
   class   conf xmin   ymin   xmax   ymax
[[  1.     0.93 131.96 152.12 159.29 172.3 ]
 [  1.     0.88  52.39 151.89  87.44 179.34]
 [  1.     0.88 262.65 140.26 286.45 164.05]
 [  1.     0.6  234.53 148.43 267.19 170.34]
 [  1.     0.58   73.2 153.51  91.79 175.64]
 [  1.     0.5  225.06 130.93 274.15 169.79]
 [  2.     0.6  266.38 116.4  282.23 173.16]]
```

Figure 7.33 Predicted bounding boxes, confidence level, and class

When we draw these predicted boxes onto the image, as shown in figure 7.34, each predicted box has its confidence next to the category name. The ground-truth boxes are also drawn onto the image for comparison.



Predicted boxes drawn onto the image

<u>Important Points</u>

- Image classification is the task of predicting the type or class of an object in an image.
- Object detection is the task of predicting the location of objects in an image via bounding boxes and the classes of the located objects.

- The general framework of object detection systems consists of four main components: region proposals, feature extraction and predictions, non-maximum suppression, and evaluation metrics.
- Object detection algorithms are evaluated using two main metrics: frame per second (FPS) to measure the network's speed, and mean average precision (mAP) to measure the network's precision.
- The three most popular object detection systems are the R-CNN family of networks, SSD, and the YOLO family of networks.
- The R-CNN family of networks has three main variations: R-CNN, Fast R-CNN,and Faster R-CNN. R-CNN and Fast R-CNN use a selective search algorithm to propose RoIs, whereas Faster R-CNN is an end-to-end DL system that uses a region proposal network to propose RoIs.
- The YOLO family of networks include YOLOv1, YOLOv2 (or YOLO9000), and YOLOv3.
- R-CNN is a multi-stage detector: it separates the process to predict the objectness score of the bounding box and the object class into two different stages.
- SSD and YOLO are single-stage detectors: the image is passed once through the network to predict the objectness score and the object class.
- In general, single-stage detectors tend to be less accurate than two-stage detectors but are significantly faster.