ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

UNIVERSITÉ DU QUÉBEC

SYS843

Réseaux de Neurones et Systèmes Flous

Synthèse de Littérature: Real-Time Object Detection

PRÉSENTÉ À: Ismail Ben Ayed

PAR

Bozan XU - XUXB02079305

MONTRÉAL, 11 Novembre 2019

Table of Contents

OVERVIEW **3**

Domains of Applications 3
Current Challenges 3
Objective 3
Methodology 3
Structure of this Document 4

SYNTHESIS **4**

Convolutional Neural Network (CNN) 4
Image Classification 5
Object Classification and Localization 5
Multiple Objects Detection and Localization 6
You Only Look Once (YOLO) 6
Single Shot MultiBox Detector (SSD) 10
YOLOv2 13
YOLOv3 16
Summary 17

ANALYSIS **18**

CONCLUSION **20**

BIBLIOGRAPHY **21**

Table of Figures

Figure 1: Image classification using CNN 5
Figure 2: In the output, there is an addition of centroid x and y coordinates, width, and height of the bounding box. 5
Figure 3: Starting with a small sliding window, we are on the lookout for small objects. We gradually increase the size of the sliding window to detect larger and larger object. Finally, the output is a set of regions that contains objects and their bounding boxes. 6
Figure 4: Image is divided into a 7x7 grid, where the yellow grid is responsible for detecting the person object because the centroid coordinates (blue dot) of that person is in the yellow grid. 7
Figure 5: Each yellow column vector represents class scores for a bounding box. We first define a threshold value and set those below the threshold value to 0. Then, we sort the class scores vectors in descending order. Finally, we apply a Non-max suppression algorithm to set redundant detections to 0. 8
Figure 6: The network structure looks like any CNN, with convolutional and max pooling layers, followed by two fully connected layers. 8
Figure 7: The architecture was designed for the Pascal VOC dataset, ... 9
Figure 8: SSD architecture featuring VGG16 serving as backbone model. 11
Figure 9: (a) A training data containing ground truth boxes for the cat and the dog. (b) In a feature map that is divided into an 8x8 grid, the anchor boxes of different aspect ratios cover the smaller areas of the raw input. (c) In a coarser grid, the anchor boxes detect objects in larger areas of the input. 11
Figure 10: Choosing more diverse anchor boxes. 14
Figure 11: Constrained bounding box prediction. 15
Figure 12: YOLOv2 architecture featuring a passthrough layer. 15
Figure 13: Accuracy improvement after applying the 7 changes. 16
Figure 14: YOLOv3 architecture 16
Figure 15: Similar architectures. 17
Figure 16: Performance of different configurations of YOLOv2 and SSD. 18
Figure 17: Accuracy and Speed on VOC 2007. 19
Figure 18: Performance of various algorithms on the COCO dataset. 19
Figure 19: Tradeoff between speed/accuracy of different algorithms on the COCO 2015 dataset. 20

Overview

Artificial intelligence has gained huge amounts of traction in recent years, especially since Deep Learning (DL). Within the field of DL, processes to identify objects through image data are grouped into the sub-field of Objection Detection.

Domains of Applications

Today, detection algorithms have taken center stage as many big corporations are racing toward building the first generation of autonomous vehicles. Of course, detection algorithms are not only limited to autopilot applications. The same techniques are also used for medical image analysis, anomaly detection, video surveillance, object tracking, and face detection amongst plenty other use cases. In this review, I am mainly interested in shining light on the state-of-the-art algorithms used for autopilot applications.

Current Challenges

One of the biggest problems in self driving cars is perception, that is, to being able to fully understand the environment around it. Other from needing to process billions of image, lidar, radar, and map data, there is a critical need for faster and more robust algorithms that will improve the safety and reliability of these autonomous systems. There are many concerns about how quickly they can come to market even if they work some of the time -- The technology has to work perfectly.

The recent advances in image recognition are powered by the formation of large-scale datasets, the development of powerful models, and the availability of vast computational resources. The obvious challenge, then, arises when there is no large-scale labeled data. In the case for autopilot applications, amongst thousands of possible scenarios, let's imagine the instance for which, at a crosswalk, a pedestrian on the curb is looking down at his phone. A human driver will be able to decide whether he should wait or hit the gas pedal. What about a biker's hand sign to signal a turn? An eye contact at a four way stop sign can decide who goes first, but how is the autonomous car going to react? The truth is, there is still a long way to go before robots can accurately differentiate complex social interactions. Furthermore, from an analytical point of view, current leading algorithms are, while accurate, very computationally intensive, such that they are too slow for real-time applications, and simply do not run on embedded systems[1].

Objective

Two years ago, I built a quadcopter from scratch and attempted to implement statistical object detection algorithms in an attempt to make it fully autopilot. It did not recognize certain tree branches and leaves as it came crashing down. Updated with DL, I want to build an autopilot system that navigates traffic. To do this, I will explore two main algorithms used in the industry. They are You Only Look Once (YOLO), and Single Shot Detector (SSD). By the end of this literature review, I will pick the best detector and configurations that give the best balance between speed and accuracy to proceed with.

Methodology

In order to establish a controlled environment to compare the results from the above-mentioned algorithms, I plan on dividing the workload in three. I will start by creating a custom training, validation, and testing dataset from the Waymo Open Dataset. The second step is to implement the models in TensorFlow. For that, I will use scikit-image and OpenCV library with Python bindings for image processing, and Keras library with a Tensorflow backend to build and train networks. Then, once trained, I will test these algorithms, on a NVIDIA Jetson Nano, in real time through the video feed of a Raspberry Pi Camera Module v2 to settle the best algorithm for real-time onboard object detection. I will highlight the tradeoff between speed and accuracy and pick the most balanced solution.

Structure of this Document

The rest of the synthesis is organized as follows. In the next section, I will dive deeper into YOLO, and SSD. I will break down the inner workings of both of these detectors and highlight some of their differences from one another.

Following that section, I will analyze the performance of each detector. One of my analysis criteria will be their speed at inference allowing real-time analysis. This section will shed light upon the advantages and drawback for each algorithm.

To conclude this review, in the very last section, I will discuss the significance of findings and add my interpretation of the tradeoff between speed and accuracy. I will then choose the best configurations for onboard autopilot systems to proceed with in my final project.

Synthesis

Object detection is one area in computer vision that is maturing very rapidly. Every year, new algorithms outperform previous ones. There exist many robust techniques for classification problems, but convolutional neural networks (CNN) have been proven to be one of the best architectures.
Currently, every state-of-the-art model incorporates CNN in some way.

Convolutional Neural Network (CNN)

Unlike object recognition, whose sole mission is to classify the object in an image, object detection solves the problem of localizing an object in an image. In this is report, I will provide an intuition of the workings of CNN.

In a CNN, there are four main operations: the convolution, the activation function, the pooling, and the fully connected layer for classification.

First, the purpose of the convolutional step is to extract features from the input image. Convolution, in the field of AI, is strictly speaking, cross-correlation. Without going too much into the math, convolution performs element-wise multiplication and additions. This is done by sliding a filter over a larger matrix. And at every location, it computes element-wise multiplication, sums the products, and outputs in a single element in the output matrix. This output matrix is called a feature map. By sliding different kinds of filters, we detect different features, such as edges and curves.

Following convolution, the next operation is the activation function. The purpose of the activation function is to determine the output of the layer. In the case of object detection, we want to predict probabilities as an output, we want to introduce some form of non-linearity to help generalize data.
For example, ReLU simply replaces every negative value in the feature map to zero. The resulting matrix is called a rectified feature map.

The third step is the pooling step. The purpose of pooling is to reduce the dimensionality of feature maps, hence reducing computation complexity while only retaining the most important information. The most common form of pooling is max pooling. When we apply max pooling on a rectified feature map, only the largest value in each region is retained. Every other value is discarded. This step also prevents overfitting.

The last step is the classification step. A fully connected layer uses typically, a softmax function on the output from the pooling layers to classify the input image. In the case of object detection, its output is a set of probabilities that describes the likelihood of each class. The sum of these probabilities is 1.

In the rest of this section, we will see how researchers use these steps in different combinations to build state-of-the-art algorithms.

Image Classification

Before we jump straight into state-of-the-art detection algorithms, I want to clarify some big ideas and we will slowly work our way to how these algorithms are implemented.

Starting with the most common problem in computer vision, the applications of image classification range from face detection on social media to cancer detection in medicine. The mission of an image classification algorithm is to look at an image and to classify the object in it.

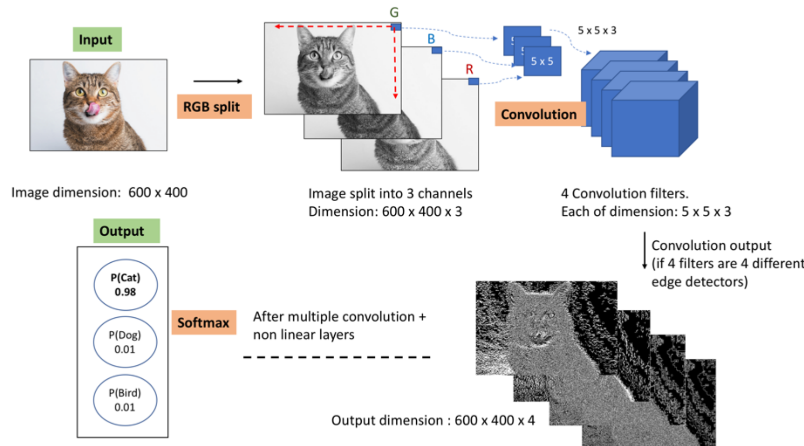


Figure 1: Image classification using CNN

Given an input image of some width, height, and channel depth, we convolve the image by some number of filters (4 in Figure 1). The output of the convolution is treated with non-linear transformations like Max Pool and ReLU. Generally, a network has multiple layers of convolution and non-linear transformations. The output of the final layer is sent to a Softmax layer that gives the probability of the input image being of a particular class. The network then aims to minimize a loss function to make predictions more accurate.

Object Classification and Localization

Following image recognition, the next step toward object detection is to localize an object in a given image and the draw a bounding box centered around the object. In other words, the algorithm has to learn the centroid position of the object and the proportion between the width and height of its bounding box. We add 4 more parameters to the output layer.

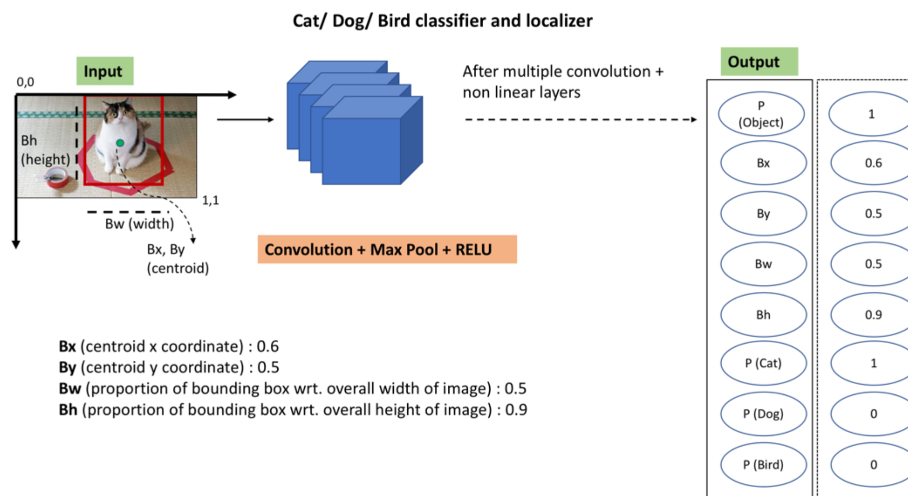


Figure 2: In the output, there is an addition of centroid x and y coordinates, width, and height of the bounding box.

And similarly to image recognition algorithms, in the input image is sent through multiple layers of convolution, Max Pool, and RELU layers before returning the coordinates of different objects we want to recognize.

Multiple Objects Detection and Localization

In the previous subsection, we described how an algorithm could detect a single object in an input image. To classify and localize multiple objects in an image, we apply the same technique, but we are dividing the input image into many smaller images before running CNN on every cropped image for object detection. The CNN could be any convolution neural network, such as VGG16.

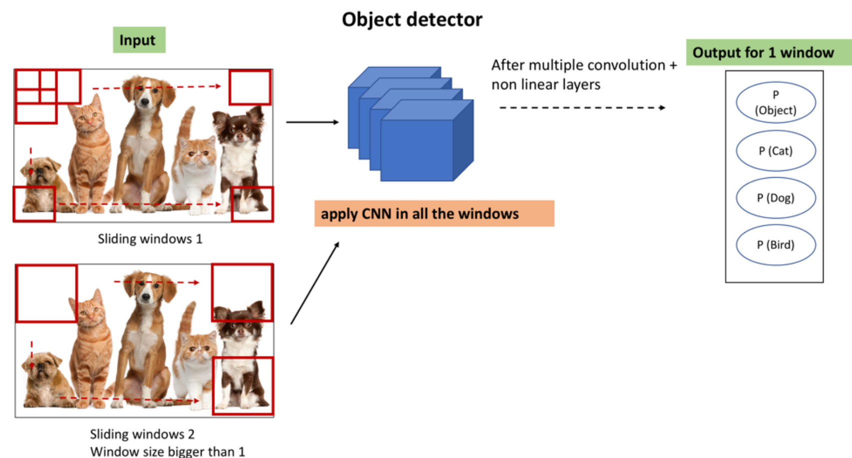


Figure 3: Starting with a small sliding window, we are on the lookout for small objects. We gradually increase the size of the sliding window to detect larger and larger object. Finally, the output is a set of regions that contains objects and their bounding boxes.

The way that the cropping is done is by sliding a window across the entire image. Starting with a small sliding window, the algorithm crops out a part of the input image and passes it to a CNN to make a prediction before sliding to another part of the image. Once every portion of the input image has gone through the CNN, the algorithm repeats every step with a bigger sliding window. In the end, the algorithm returns a set of cropped regions that contains an object along with its class and bounding box.
The downside with this algorithm is performance. We need to ask the model for a prediction every time we slide the window. One solution could be to increase the stride of the sliding window, but we might fail to detect some objects. However, if we use a small stride, many pixels are going to be shared and a lot of information are going to be redundantly calculated.
In the past, it was not a problem because models were mostly linear and had features designed by hand, but nowadays, models like VGG16 has hundreds of millions of parameters. It is then impossible to do real-time object detection, which is what my project is about.
To reduce computational complexity of passing cropped images one at a time, we replace the fully connected layer in the CNN with 1x1 convolution layers that passes the complete image at once and we do it for every window size.

You Only Look Once (YOLO)

A better solution to the sliding window algorithm is YOLO. It is based on a minor tweak from the algorithm described above. Again, the input images are labeled with four additional variables: the centroid coordinates (x, y), height(h), and width(w) of the bounding box. Then, the idea is to divide the input image into an S by S grid where every bounding box can be part of several grids, but we assign the object and bounding box to the grid that contains the centroid coordinates of the object. The model can then predict the geometry and position of the bounding box, on top of the object class. And because

the model learns to mark objects with bounding boxes, the bounding boxes are more accurate.

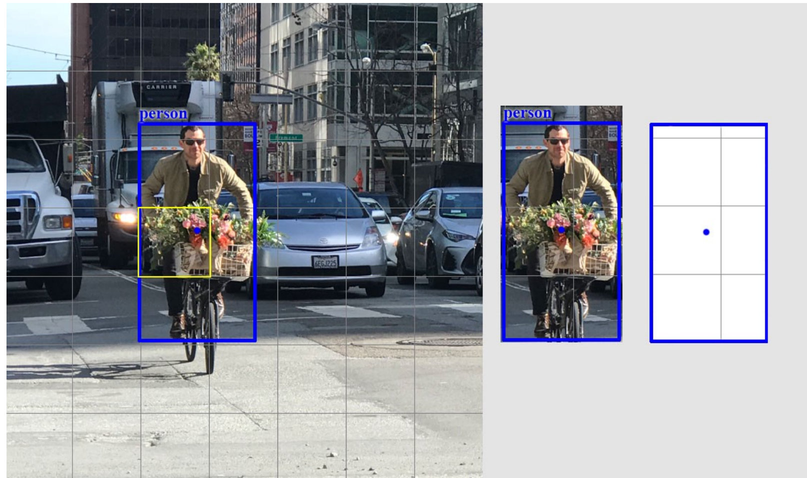


Figure 4: Image is divided into a 7x7 grid, where the yellow grid is responsible for detecting the person object because the centroid coordinates (blue dot) of that person is in the yellow grid.

Each grid cell predicts a fixed number (B) of bounding boxes, but only detects one object per grid. This sets a hard limit on how close objects can be.

For every bounding box prediction, there are five predicted values: $\{x, y, w, h, confidence\}$. The first four are the spatial coordinates of the object and the aspect ratio of its bounding box. The last element is the bounding box confidence score defined as:

$$P(Object) * IoU_{prediction}^{truth}$$

where IoU means intersection over union. It represents the area of intersection over the area of union of the predicted bounding box and the ground truth box. In practice, any IoU over 50% is usually considered a correct prediction.
Looking from the above equation, the confidence score measures the probability that the bounding box contains an object and how well the predicted box matches the ground truth. If there is no object in that grid cell, the box confidence score is 0.

The output tensor also predicts a set of conditional class probabilities (C):

$$P(Class_i|Object)$$

which is the conditional probability of an object to belong to class i. This means that if no object is present in the grid cell, the loss function will not count it as a wrong prediction.

Next, we calculate the class confidence scores by multiplying the conditional class probabilities with the bounding box confidence scores:

$$P(Class_i|Object) * P(Object) * IoU_{prediction}^{truth} = P(Class_i) * IoU_{prediction}^{truth}$$

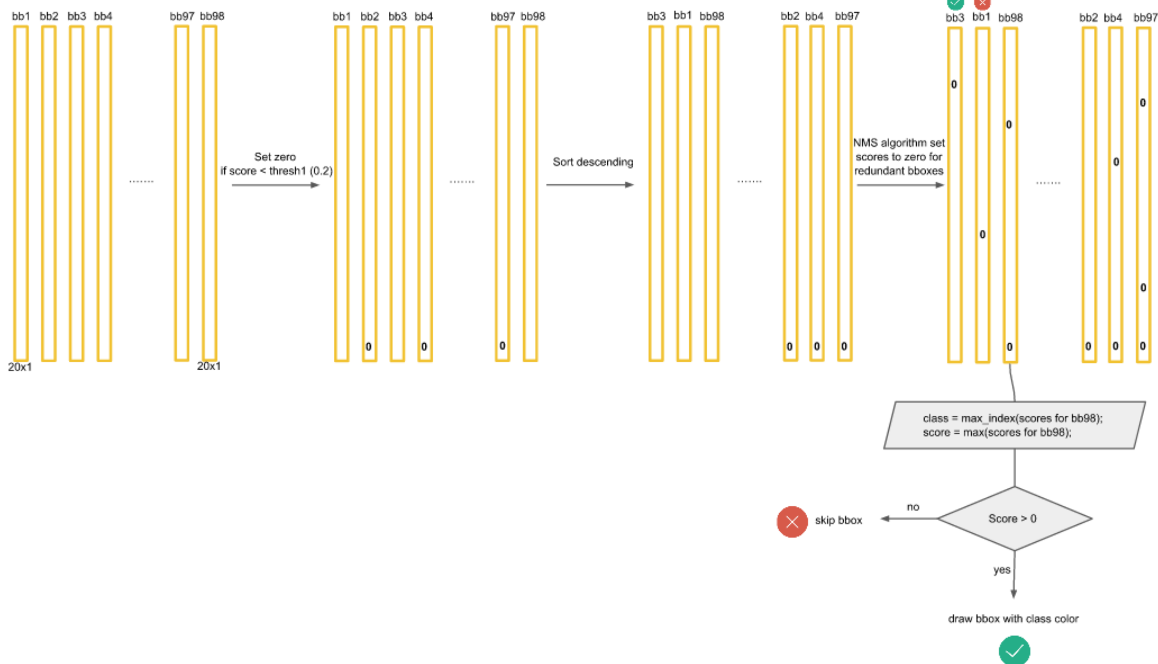


Figure 5: Each yellow column vector represents class scores for a bounding box. We first define a threshold value and set those below the threshold value to 0. Then, we sort the class scores vectors in descending order. Finally, we apply a Non-max suppression algorithm to set redundant detections to 0.

After calculating the class confidence scores for each bounding box, we set a threshold value of scores and sort them in a descending fashion for each class. To get rid of multiple boxes detection the same object, we apply a Non-max suppression algorithm to keep the boxes with maximum probability and suppress the boxes with high IoU with them.
The final tensor output for a S by S grid, is then $S \times S \times (B * 5 + C)$, where B denotes the number of bounding boxes and C represents the number of class predictions.

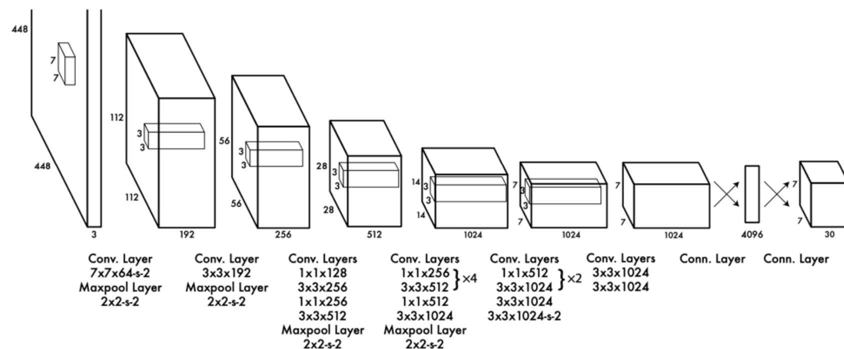


Figure 6: The network structure looks like any CNN, with convolutional and max pooling layers, followed by two fully connected layers.

Once we understand how the predictions are encoded, YOLO's architecture is simple to understand. The network comprises of 24 convolutional layers followed by two fully connecter layers. The alternating use of 1x1 reduction layers to reduce the depth of the features space followed by a 3x3 convolutional layer was inspired by the GoogLeNet (Inception) model[2]. As for activation functions, the final layer uses a linear activation function, but every other layer employs leaky RELU as activation function:

$$\phi(x) = \begin{cases} x & when\ x > 0 \\ 0.1x & otherwise \end{cases}$$

Name	Filters	Output Dimension
Conv 1	7 x 7 x 64, stride=2	224 x 224 x 64
Max Pool 1	2 x 2, stride=2	112 x 112 x 64
Conv 2	3 x 3 x 192	112 x 112 x 192
Max Pool 2	2 x 2, stride=2	56 x 56 x 192
Conv 3	1 x 1 x 128	56 x 56 x 128
Conv 4	3 x 3 x 256	56 x 56 x 256
Conv 5	1 x 1 x 256	56 x 56 x 256
Conv 6	1 x 1 x 512	56 x 56 x 512
Max Pool 3	2 x 2, stride=2	28 x 28 x 512
Conv 7	1 x 1 x 256	28 x 28 x 256
Conv 8	3 x 3 x 512	28 x 28 x 512
Conv 9	1 x 1 x 256	28 x 28 x 256
Conv 10	3 x 3 x 512	28 x 28 x 512
Conv 11	1 x 1 x 256	28 x 28 x 256
Conv 12	3 x 3 x 512	28 x 28 x 512
Conv 13	1 x 1 x 256	28 x 28 x 256
Conv 14	3 x 3 x 512	28 x 28 x 512
Conv 15	1 x 1 x 512	28 x 28 x 512
Conv 16	3 x 3 x 1024	28 x 28 x 1024
Max Pool 4	2 x 2, stride=2	14 x 14 x 1024
Conv 17	1 x 1 x 512	14 x 14 x 512
Conv 18	3 x 3 x 1024	14 x 14 x 1024
Conv 19	1 x 1 x 512	14 x 14 x 512
Conv 20	3 x 3 x 1024	14 x 14 x 1024
Conv 21	3 x 3 x 1024	14 x 14 x 1024
Conv 22	3 x 3 x 1024, stride=2	7 x 7 x 1024
Conv 23	3 x 3 x 1024	7 x 7 x 1024
Conv 24	3 x 3 x 1024	7 x 7 x 1024
FC 1	-	4096
FC 2	-	7 x 7 x 30 (1470)

Figure 7: The architecture was designed for the Pascal VOC dataset,

where the researchers used S=7, B=2, and C=20.

During training, a weighted sum between the localization error and classification error is calculated. The researchers chose to calculate sum-square errors because it is easy to optimize. They use a weighted sum because when there is no object in a grid cell, the confidence score is 0 and that can lead the training to diverge[2]. The multi-part loss function is:

$$\lambda_{coord}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathrm{I}_{ij}^{obj}[(x_i-\hat{x}_i)^2+(y_i-\hat{y}_i)^2]$$
$$+\lambda_{coord}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathrm{I}_{ij}^{obj}\left[\left(\sqrt{w}_i-\sqrt{\widehat{w}}_i\right)^2+\left(\sqrt{h}_i-\sqrt{\hat{h}}_i\right)^2\right]$$
$$+\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathrm{I}_{ij}^{obj}\left(C_i-\hat{C}_i\right)^2+\ \lambda_{noObj}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathrm{I}_{ij}^{noObj}\left(C_i-\hat{C}_i\right)^2$$
$$+\sum_{i=0}^{S^2}\mathrm{I}_{ij}^{obj}\sum_{c\in classes}(p_i(c)-\hat{p}_i(c))^2$$

The researchers set $\lambda_{coord}=5$ and $\lambda_{noObj}=\mathbf{0.5}$ as parameters to balance the localization loss and the confidence loss.
To digest the whole loss function, we are going to break it down into parts. The first term of the equation represents the loss related to the position of the predicted bounding box:

$$\lambda_{coord}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathrm{I}_{ij}^{obj}[(x_i-\hat{x}_i)^2+(y_i-\hat{y}_i)^2]$$

where $\mathbb{I}_{ij}^{obj}$ represents whether an object is in grid cell i and j is the bounding box that is responsible for that prediction. Multiplying by this tensor will avoid counting the error on bounding boxes that are not responsible for the object.
This term computes a sum over each bounding box j for each grid cell i.
Moving on to the second term:

$$\lambda_{coord}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{I}_{ij}^{obj}\left[\left(\sqrt{w}_i-\sqrt{\widehat{w}}_i\right)^2+\left(\sqrt{h}_i-\sqrt{\widehat{h}}_i\right)^2\right]$$

The second term represents the loss related to the width and height of the bounding box. The function calculates in the same way as the one above, but it does not predict the width and height of the bounding box. By predicting the square root of w and h, a deviation in a larger bounding box is going to matter less than the same deviation in a smaller bounding box.
The third part of the loss function is the loss reflecting the confidence score for each bounding box:

$$\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{I}_{ij}^{obj}\left(C_i-\hat{C}_i\right)^2+\lambda_{noObj}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{I}_{ij}^{noObj}\left(C_i-\hat{C}_i\right)^2$$

where C is the confidence score and $\hat{C}$ is the IoU of the predicted bounding box with the ground truth. Here $\mathbb{I}_{ij}^{noObj}$ does the opposite of $\mathbb{I}_{ij}^{obj}$. It is 1 when there is no object in grid cell i.
Finally, the last term of the loss function:

$$\sum_{i=0}^{S^2}\mathbb{I}_{ij}^{obj}\sum_{c\in classes}(p_i(c)-\hat{p}_i(c))^2$$

represents the classification loss. The function first computes the sum-squared error for class classification, and only sums over the grid cells that contain an object. This will avoid counting classification error from grid cells that do not contain an object.

So how did the researchers train the network? The researchers pretrained the first 20 convolutional layers followed by an average-pooling layer and a fully connected layer on the ImageNet 1000-class competition dataset, using an input resolution of 224x224. They then increased the resolution to 448x448 for detection.
After a week of pretraining, they then trained the full network for 135 epochs using a batch size of 64, momentum 0.9, and decay of 0.0005. For the first few epochs, they slowly raised the learning rate from 0.001 to 0.01 to avoid unstable gradients. They kept 0.01 for the next 75 epochs. After that, they decreased the learning rate back to 0.001 for 30 epochs, before finally decreasing it further down to 0.0001 for another 30 epochs.
To combat overfitting, the researchers placed a dropout layer with a rate of 50% after the first connected layer to prevent neurons to rely on each other. The researchers also introduced data augmentation with random scaling and translations of up to 20% of the original image size, as well as randomly adjusted the exposure and saturation of the original images.

Single Shot MultiBox Detector (SSD)

Developed in 2016 by Liu et al., SSD is also a one-step approach, hence the name "single shot detector". SSD is comprised of two main components:

1) A backbone model, that functions as a feature extractor. It is usually a pre-trained image classification network like ResNet or Mobilenet, but whose final fully connected classification layer has been removed. In the paper [1], Liu et al used VGG16.
2) And, an SSD head, made out of convolutional layers that progressively decreases in size to allow detection at multiple scales, unlike YOLO that operates on one

single scale. It outputs the bounding boxes and classes of objects in the spatial location of the final layer activations.

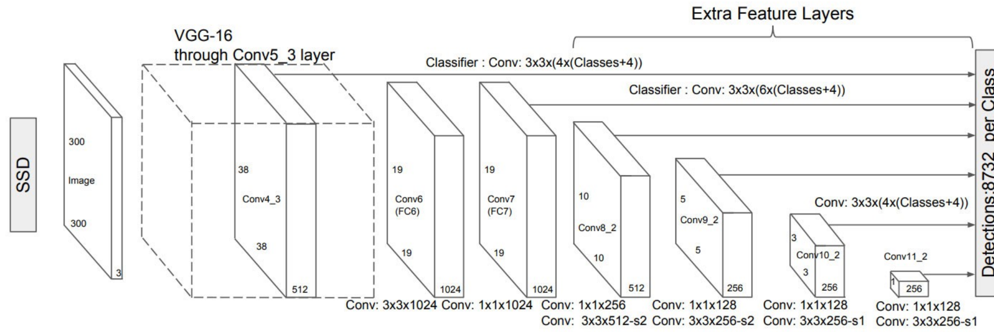


Figure 8: SSD architecture featuring VGG16 serving as backbone model.

First, SSD also works by dividing the input image into a grid of cells. Each grid cell is responsible for predicting the class and position of objects within it. If there is no object, it will predict the background class, and no location will be returned.
SSD detects objects through multi-scale feature maps. As CNN reduces the spatial dimension gradually, the resolution of the feature maps also decreases, but in the SSD framework, lower resolution layers are used to detect larger scale objects, as shown in Figure 9 (c).

To detect objects with different aspect ratio, each grid cell is assigned with multiple (4 or 6) anchor boxes of different aspect ratio centered at the grid cell. The researchers refer to these as default boxes. The predictions on boundary boxes are relative to the center of those default boxes. We thus define the offsets to the default box at each cell $\{\Delta x, \Delta y, w, h\}$.

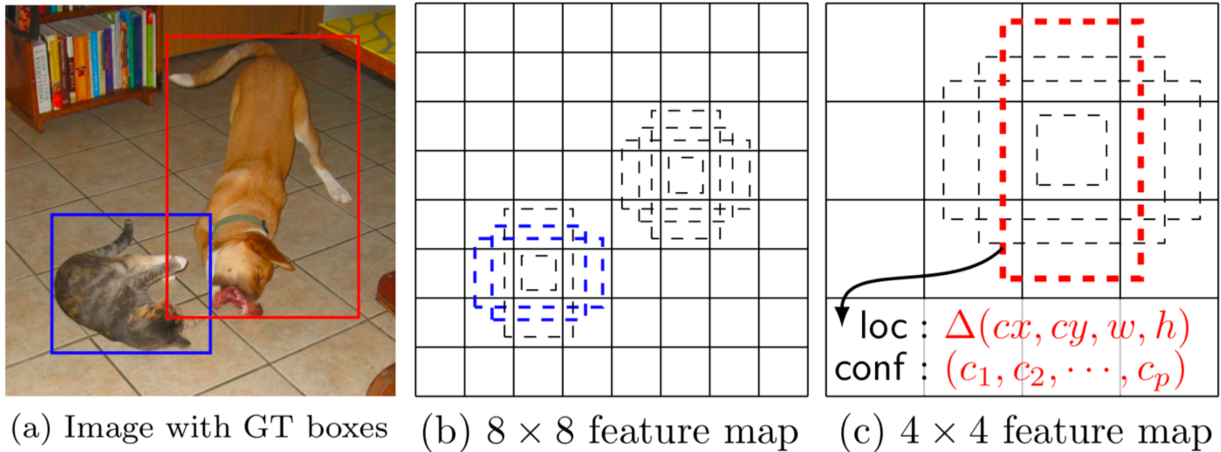


Figure 9: (a) A training data containing ground truth boxes for the cat and the dog. (b) In a feature map that is divided into an 8x8 grid, the anchor boxes of different aspect ratios cover the smaller areas of the raw input. (c) In a coarser grid, the anchor boxes detect objects in larger areas of the input.

Default boxes are chosen manually. SSD defines a scaling factor for each feature map. Suppose there are m feature maps, and we define index $k \in [1, m]$ to represent the k-th feature map. Then,

$$s_k = s_{min} + \frac{s_{max} - s_{min}}{m - 1}(k - 1)$$

where $s_{min} = 0.1$, $s_{max} = 0.7$, and $s_{m+1} = 1$. This means that starting from the left, the layers in the SSD head detect objects from the smallest scale of 0.1 all the way up to a scale of 0.7.
We can then compute the width and height of the default boxes:

$$w = s * \sqrt{aspect\ ratio}$$
$$h = s/\sqrt{aspect\ ratio}$$

where aspect ratio ∈ $\left\{1, 2, 3, \frac{1}{2}, \frac{1}{3}\right\}$
For the aspect ratio of 1, the SSD adds an extra default box with scale:

$$s_k{}' = \sqrt{s_k * s_{k+1}}$$

And the center of the default boxes is written as:

$$\left(\frac{i + 0.5}{f_k}, \frac{j + 0.5}{f_k}\right)$$

where f_k is the size of the k-th feature map and $i, j\ \in [0, f_k)$.

It is by combining the predictions from every default box of different scales and aspect ratios coming from every grid cell on every feature map that SSD can detect objects of every shape and size. This number is usually in the thousands, and as for larger models like SSD512, the output makes 24564 predictions[1].

During training, the default boxes with the highest degree of IoU with the ground truth box are matched with that object. In other words, the more overlap there is between a default box and a ground truth box, the better match it is. In Figure 9 (b), there are two matched anchor boxes with the cat. These anchor boxes are called positive matches, whereas the two other ones are negative matches.
Similarly to YOLO, the loss function for SSD is a weighted sum of classification loss and localization loss:

$$L = \frac{1}{N}(L_{classification} + \propto L_{localization})$$

where N is the number of matched anchor boxes and ∝ balances the weights between the losses.

The localization loss is a loss between the predicted box correction (d_m^i) and the true values g_m^j:

$$L_{localization} = \sum_{ij} \sum_{m \in \{x,y,w,h\}} \mathrm{I}_{ij}^k\, L_{smoothL1}(d_m^i - t_m^j)^2$$

where I_{ij}^k represents the matching between i^{th} anchor box with coordinates $(p_x^i, p_y^i, p_w^i, p_h^i)$ and j^{th} ground truth box with coordinates $(g_x^i, g_y^i, g_w^i, g_h^i)$ for an object in class k. It is 1 if if $IoU > 0.5$, and 0 otherwise.
$L_{smoothL1}$ is a Smooth L1 loss defined as:

$$L_{smoothL1}(s) = \begin{cases} 0.5s^2 & when\ |s| < 1 \\ |s| - 0.5 & otherwise \end{cases}$$

t_m^j represent the offsets for the center, width, and height of the anchor boxes. They are defined as:

$$t_x^j = (g_x^j - p_x^i)/p_w^i$$
$$t_y^j = (g_y^j - p_y^i)/p_h^i$$
$$t_w^j = \log\left(\frac{g_w^j}{p_w^i}\right)$$

$$t_h^j = \log\left(\frac{g_h^j}{p_h^i}\right)$$

As for the classification loss, we apply a softmax to make it a nice probability function:

$$L_{classification} = -\sum_{i \in Pos}^{N} \mathrm{I}_{ij}^{k} \log(\hat{c}_i^k) - \sum_{i \in Neg}^{N} \log(\hat{c}_i^0)$$

where $\hat{c}_i^k = softmax(c_i^k) = \frac{exp(c_i^k)}{\sum_k c_i^k}$

For positive matches, the classification loss is penalized according to the confidence score of class k. For negative matches, SSD predicts class $\mathbf{0}$, or background, meaning no object was detected.

As mentioned above, there is a very large number of predictions that are made for the number of objects present in a given image. This means that there will be many times more negative matches than positive matches. This imbalance causes the model to learn about the background space rather than detecting objects. Rather than using all the negative matches, SSD only keeps the matches with highest confidence (false positives) as to keep a ratio of 3:1 between negatives and positives. Apparently, this ratio leads to faster optimization and even makes the training process more stable[12].

To improve accuracy, the researchers used data augmentation on every input image.

- Use the original
- Sample a patch with IoU of 0.1, 0.3, 0.5, 0.7, or 0.9
- Sample a patch randomly

The sampled patch is then resized to a fixed size before undergoing two photometric distortions[4].

YOLOv2

A year after YOLO was published, Redmon et al. came up with a faster and more accurate version of YOLO. YOLOv2, the modified YOLO, highlights 7 major changes.

The first major modification is the addition of batch normalization to every layer. Batch normalization prevents activations from blowing up or going to zero, so it helps the training to converge. It also introduces some noise to each hidden layer's activations, hence removing the need for dropout. The best part is that mAP was pushed up by over 2% [29000].

The second change is a fine tune on the classification network by training at 448x448 resolution for 10 epochs after the usual 224x224. This change increases mAP by almost 4%[29000].

The third change is the replacing of the two fully connected layers that makes arbitrary guesses on the boundary boxes with convolution layers. In early training, different shaped boxes are fighting each other on what shape to focus on. To remedy this problem, the researchers suggest handpicking a set of boundary boxes that are diverse in shape. With convolution layers, we predict offsets and confidence to each of the anchor boxes instead of predicting a set of anchor boxes.

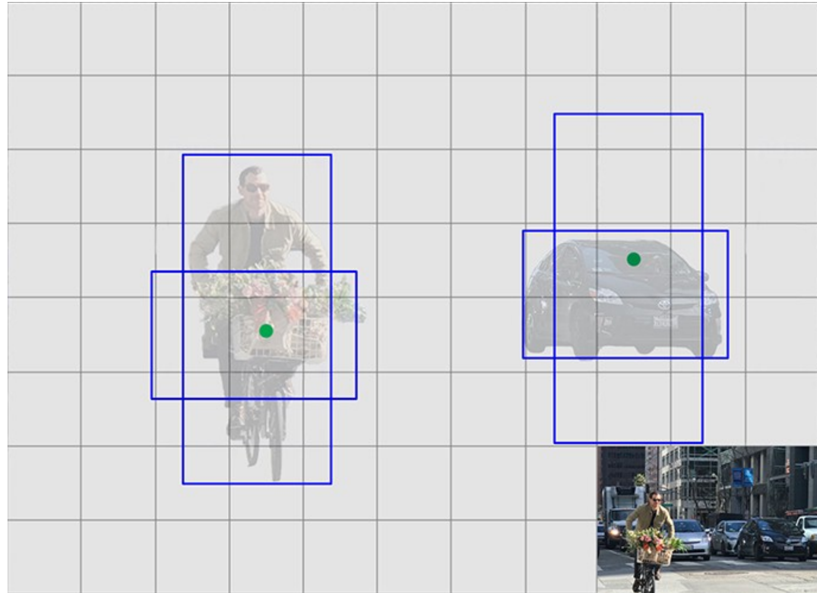


Figure 10: Choosing more diverse anchor boxes.

So if we handpicked 5 anchor boxes of diverse shapes, each prediction is going to consist of the 4 spatial parameters for the predicted bounding box, 1 confidence score, and c class probabilities. Since each boundary box has $5 + c$ parameters and there are 5 predictions per grid cell, we calculate $5 * (5 + c)$ parameters per grid cell. Just like YOLO, the confidence scores are still the overlap between the predicted box and the ground truth box. And the class prediction:

$$P(Class_i|Object)$$

is still given by the conditional probability of $Class_i$ given $Object$.

And because large objects usually occupy the center of an image, the researchers shrank the input images from 448x448 to 416x416. This yields a 7x7 grid, instead of the 8x8 grid in YOLO. This allows for a single center grid to be responsible for a large object.

By adding these anchor boxes, YOLOv2 can predict $19 * 19 * 5 = 1805$ boxes, compared to the 98 in the original work[3]. The precision dropped a little, but the recall is increased by a large margin.

The fourth change arises from the choice of anchor boxes. Rather than picking anchor boxes by hand, the researchers ran k-means clustering to pick the most appropriate anchor boxes. They found that setting $k = 5$ gives the best tradeoff between recall and complexity.

The fifth modification works at putting constraints on the location of the bounding box. In the original YOLO, the predicted bounding box can be far from the original grid location, resulting in instability in the early iterations. YOLOv2 makes following predictions:

$$P(object) * IoU(b, object) = \sigma(t_o)$$
$$b_x = \sigma(t_x) + c_x$$
$$b_y = \sigma(t_y) + c_y$$
$$b_h = p_h e^{t_h}$$
$$b_w = p_w e^{t_w}$$

where t are the values predicted by anchor boxes, (c_x, c_y) is the centroid coordinates of the box, and p_h, p_w are the height and width of the anchor box. By using σ to bound the location, we observe a 1% increase in mAP.

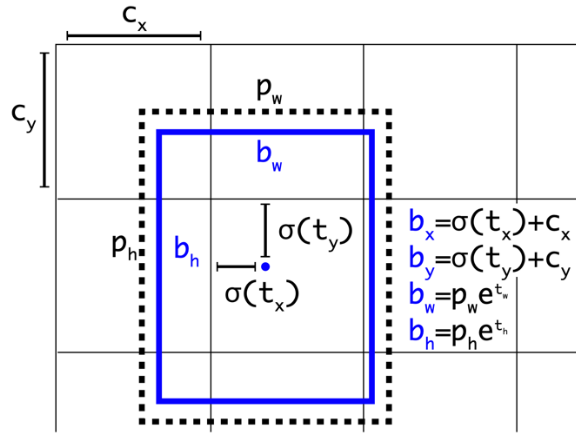


Figure 11: Constrained bounding box prediction.

The sixth modification is the addition of a passthrough layer that brings features from a higher resolution layer directly to the detector. This change helps YOLOv2 to detect smaller objects, hence increasing the mAP by 1%.

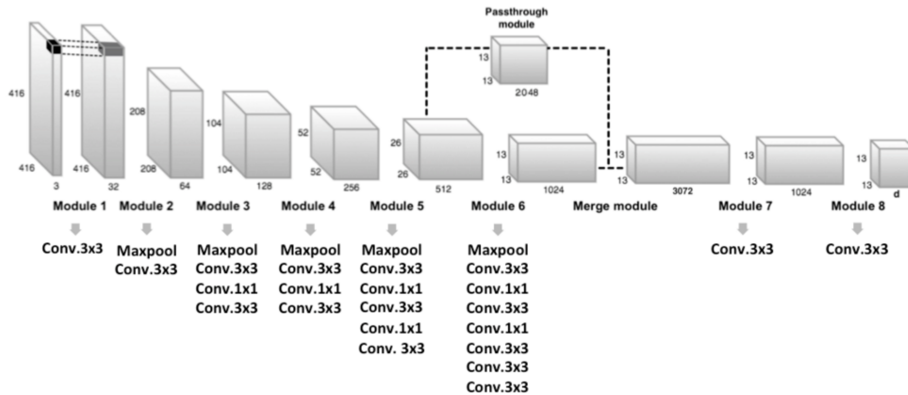


Figure 12: YOLOv2 architecture featuring a passthrough layer.

The final change is at the level of training. During training, at every 10 batches, new image dimensions are randomly chosen to increase robustness. This regime improves predictions at different input resolutions.

	YOLO								YOLOv2
batch norm?		✓	✓	✓	✓	✓	✓	✓	✓
hi-res classifier?			✓	✓	✓	✓	✓	✓	✓
convolutional?				✓	✓	✓	✓	✓	✓
anchor boxes?				✓	✓				
new network?					✓	✓	✓	✓	✓
dimension priors?						✓	✓	✓	✓
location prediction?						✓	✓	✓	✓
passthrough?							✓	✓	✓
multi-scale?								✓	✓
hi-res detector?									✓
VOC2007 mAP	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8	**78.6**

Figure 13: Accuracy improvement after applying the 7 changes.

YOLOv3
Redmon et al. claim to have "phone[d] it in for a year" and "didn't do a whole lot of research" in 2018[5]. They made some tweaks to the model and YOLOv3 is better and stronger, but slower than YOLOv2.

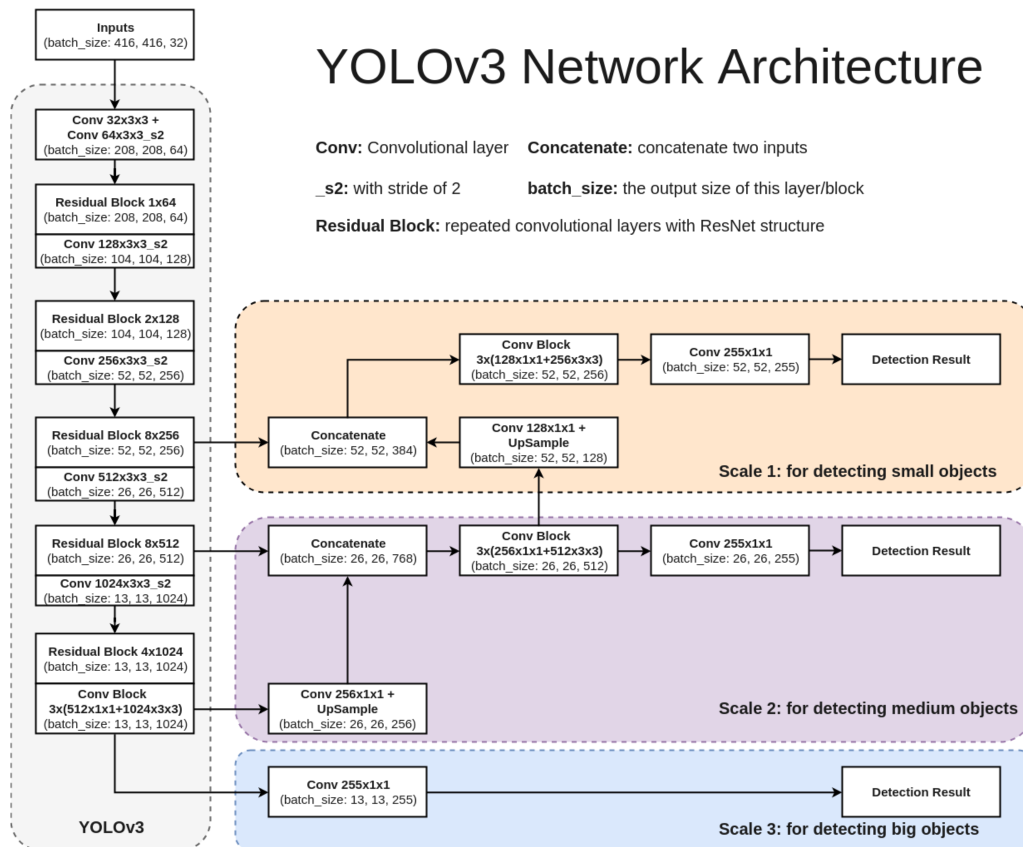


Figure 14: YOLOv3 architecture.

Walking through the whole system, YOLOv3 works in the following way:

1. YOLOv3, just like YOLOv2, predicts a confidence score using logistic regression. This returns 1 if the anchor box overlaps the ground truth box better than any other anchor box. There is only one anchor box associated with each ground truth box. A cross-entropy error loss is used instead of the sum squared error loss.

2. Each bounding box may predict more than one class, because YOLOv3 now performs multilabel classification. This helps with classes that are not mutually exclusive, like “pedestrian” and “child”.
3. Instead of the usual softmax layer for class prediction, YOLOv3 now uses a logistic classifier for each class.
4. The most salient feature of YOLOv3 is the detection at 3 scales. YOLOv3 applies 1x1 kernels on feature maps at 3 different places in the network. At those 3 places, the dimensions of the input image get downsampled, respectively, by 32, 16, and 8.
5. In the last feature map layer, it goes back by 2 layers and upsamples it by 2. The upsampled layers are then concatenated with the previous layers to preserve the fine-grained features that improves the detections of small objects.
6. We are still using k-means clustering to choose 9 anchor boxes. These anchor boxes are divided into 3 groups of 3 by their size, 1 group for each scale. Redmon et al. “chose 9 clusters and 3 scales arbitrarily”[5].
7. As illustrated in Figure 14, the new feature extractor network, named Darknet-53, is a hybrid of successive 3x3 and 1x1 convolutional layers and residual network (ResNet). The residual blocks are passthrough layers essentially. The network consists of 53 convolutional layers, has less billion floating point operations (BFLOP) than ResNet-152, and achieves the same accuracy at twice the speed.
8. The training is still done on full images with no hard negative mining. Multi-scale training, data augmentation, and batch normalization are still used.

Summary

So far, we have examined the algorithmic changes in the YOLO family as well as studied SSD in detail.

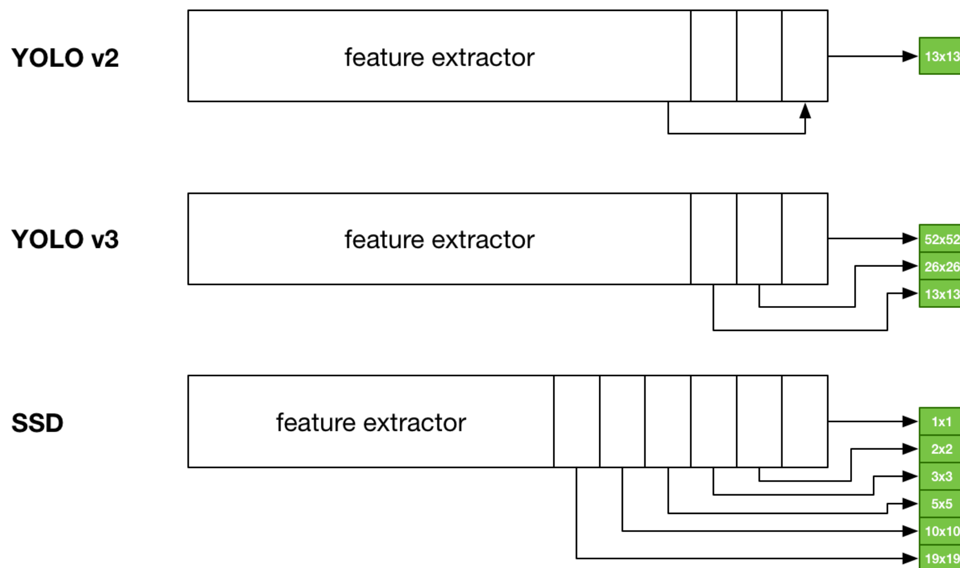


Figure 15: Similar architectures.

To summarize, we can see from Figure 15 that YOLOv3 is quite similar to SSD. The difference is that YOLOv3 uses upsampling to obtain fine-grained features whereas SSD gradually downsamples. SSD runs detection on 6 feature maps ranging from 1x1 to 19x19 to detect objects of every size. YOLOv3, like SSD, makes predictions at 3 different scales. YOLOv2, on the other hand, only has a 13x13 feature map. This limitation prevents YOLOv2 to accurately detect smaller objects, despite the passthrough layer.

Like YOLO, each SSD grid cell makes multiple predictions. It was found that making 5 detections per grid was optimal for YOLOv2. YOLOv3 makes a 3 detections per grid cell per scale and SSD uses 3-4 detectors per finer grid, and 6 on coarser grids.

The coordinate predictions are also a little different. For SSD, does not apply a sigmoid to the predicted centroid coordinates offset. This means that the centroid coordinates of the predicted box can be outside of the grid.

SSD also does not predict a confidence score. Instead it solves this by having a background class to signify that no object was found by the detector. It is treated in the same way as a low confidence score in YOLO.

The anchor boxes in SSD are also different from those in the YOLOv1 and YOLOv2. SSD uses different sizes of anchors on different size of grids. A finer grid will have smaller anchors than a coarser grid. Since YOLOv1 and YOLOv2 only has one grid, the sizes of their anchor boxes differ widely. Also, in SSD, the anchors do not specialize on an object size, different grids do that. In the YOLO family, anchor boxes compete with each other to specialize in both the shape and size of an object, which causes some instability in early training.

The choice of anchors in SSD is made via simple equations that we've seen above, whereas YOLOv2 and YOLOv3 run k-means clustering on the training data to choose the best set of anchor boxes.

SSD outputs many more predictions than YOLO because of its greater number of grids and anchors, so it has an advantage of finding every object in the image, but the downside to that is that it has to run hard negative mining to determine which predictions to keep.

Analysis

In the previous section, we have looked at state-of-the-art single stage object detection models. We have also seen that some minor differences between each one of them can lead to interesting architectural differences.

In this section we will qualitative examine the performance of these algorithms. The main focus is this analysis will be on speed and accuracy.

Model	Train	Test	mAP	FLOPS	FPS	Cfg	Weights
Old YOLO	VOC 2007+2012	2007	63.4	40.19 Bn	45		link
SSD300	VOC 2007+2012	2007	74.3	-	46		link
SSD500	VOC 2007+2012	2007	76.8	-	19		link
YOLOv2	VOC 2007+2012	2007	76.8	34.90 Bn	67	cfg	weights
YOLOv2 544x544	VOC 2007+2012	2007	78.6	59.68 Bn	40	cfg	weights
Tiny YOLO	VOC 2007+2012	2007	57.1	6.97 Bn	207	cfg	weights
SSD300	COCO trainval	test-dev	41.2	-	46		link
SSD500	COCO trainval	test-dev	46.5	-	19		link
YOLOv2 608x608	COCO trainval	test-dev	48.1	62.94 Bn	40	cfg	weights
Tiny YOLO	COCO trainval	-	-	7.07 Bn	200	cfg	weights

Figure 16: Performance of different configurations of YOLOv2 and SSD.

Figure 16 is taken from Redmon's personal website, but both Redmon and Liu ran their algorithms on a GeForce GTX Titan X. Immediately we see that YOLOv1 labeled "Old YOLO" is as fast but still out of the race with the lowest accuracy score. We see that YOLOv2 is overall faster and more accurate than SSD on the VOC 2007 dataset. YOLOv2

with an input size of 416x416 performs as accurately as SSD with 500x500 input dimension, but YOLOv2 runs at 67 FPS, whereas SSD500 runs at 19.

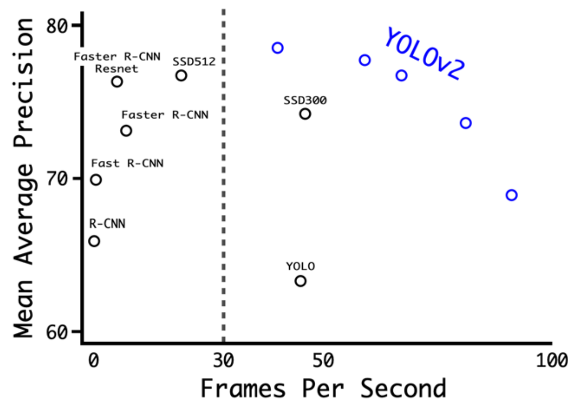


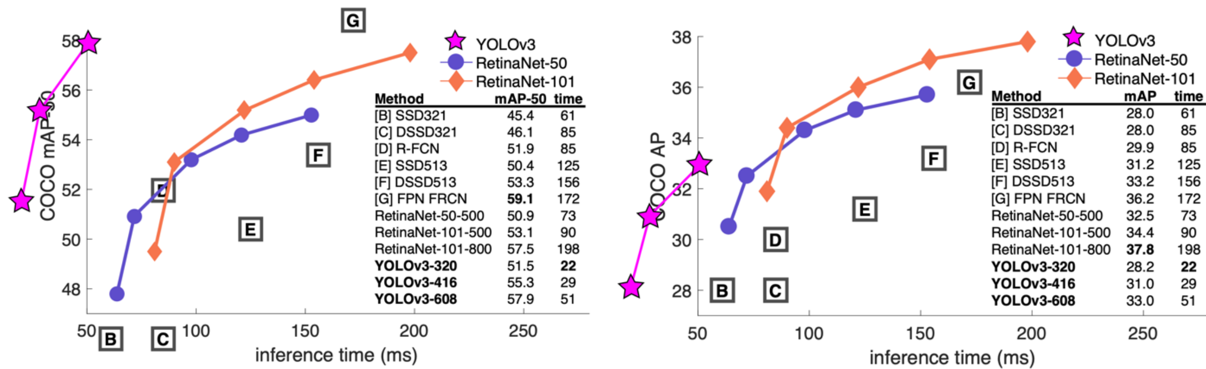
Figure 17: Accuracy and Speed on VOC 2007.

From Figure 17 we see that YOLOv2 is exceptionally fast. There is also the performance of many regional based CNN methods plotted on the graph. We did not consider those algorithms in this review because while accurate, they are too slow to be of any use for real-time applications.
The reason YOLOv2 is this fast is because it only computes $13 * 13 * 5 = 845$ boxes for every input image, whereas SSD512 predicts 24564 bounding boxes.

Model	Train	Test	mAP	FLOPS	FPS	Cfg	Weights
SSD300	COCO trainval	test-dev	41.2	-	46		link
SSD500	COCO trainval	test-dev	46.5	-	19		link
YOLOv2 608x608	COCO trainval	test-dev	48.1	62.94 Bn	40	cfg	weights
Tiny YOLO	COCO trainval	test-dev	23.7	5.41 Bn	244	cfg	weights
SSD321	COCO trainval	test-dev	45.4	-	16		link
DSSD321	COCO trainval	test-dev	46.1	-	12		link
R-FCN	COCO trainval	test-dev	51.9	-	12		link
SSD513	COCO trainval	test-dev	50.4	-	8		link
DSSD513	COCO trainval	test-dev	53.3	-	6		link
FPN FRCN	COCO trainval	test-dev	59.1	-	6		link
Retinanet-50-500	COCO trainval	test-dev	50.9	-	14		link
Retinanet-101-500	COCO trainval	test-dev	53.1	-	11		link
Retinanet-101-800	COCO trainval	test-dev	57.5	-	5		link
YOLOv3-320	COCO trainval	test-dev	51.5	38.97 Bn	45	cfg	weights
YOLOv3-416	COCO trainval	test-dev	55.3	65.86 Bn	35	cfg	weights
YOLOv3-608	COCO trainval	test-dev	57.9	140.69 Bn	20	cfg	weights
YOLOv3-tiny	COCO trainval	test-dev	33.1	5.56 Bn	220	cfg	weights
YOLOv3-spp	COCO trainval	test-dev	60.6	141.45 Bn	20	cfg	weights

Figure 18: Performance of various algorithms on the COCO dataset.

In Figure 18, we notice that YOLOv3 is not as fast as YOLOv2 but it achieves around 10 points higher mAP than YOLOv2 and SSD. This makes sense given that YOLOv3 uses a 53 layers CNN as feature extractor whereas YOLOv2 only uses a CNN with 19 layers. YOLOv3 also makes 10647 bounding box predictions for every image. Given an input image of size 608x608, YOLOv2 performs with an accuracy of 48.1 mAP at 40 FPS. In comparison, YOLOv3 scores 57.9 in mAP, but runs at 20 FPS. However, YOLOv3 is still the superior algorithm because given an input image of only 320x320, it scores a higher mAP and runs at a higher FPS than YOLOv2 with an input of 608x608.



Method	mAP-50	time
[B] SSD321	45.4	61
[C] DSSD321	46.1	85
[D] R-FCN	51.9	85
[E] SSD513	50.4	125
[F] DSSD513	53.3	156
[G] FPN FRCN	**59.1**	172
RetinaNet-50-500	50.9	73
RetinaNet-101-500	53.1	90
RetinaNet-101-800	57.5	198
YOLOv3-320	51.5	**22**
YOLOv3-416	55.3	29
YOLOv3-608	57.9	51

Method	mAP	time
[B] SSD321	28.0	61
[C] DSSD321	28.0	85
[D] R-FCN	29.9	85
[E] SSD513	31.2	125
[F] DSSD513	33.2	156
[G] FPN FRCN	36.2	172
RetinaNet-50-500	32.5	73
RetinaNet-101-500	34.4	90
RetinaNet-101-800	**37.8**	198
YOLOv3-320	28.2	**22**
YOLOv3-416	31.0	29
YOLOv3-608	33.0	51

Figure 19: Tradeoff between speed/accuracy of different algorithms on the COCO 2015 dataset.

We see that YOLOv3 achieve accuracy that tops the best of every other detector while running at a fraction of their speed under the mAP at 0.5 IoU metric. We also notice that YOLOv3 does not perform as well under the new COCO metric (average AP between 0.5 and 0.95 IoU). To be fair, variations of SSD also experienced a significant drop in accuracy. Yet, YOLOv3 is still many points more accurate than the SSD variants. In fact, YOLOv3 is more accurate and faster than certain configurations of RetinaNet. For RetinaNet to increase its accuracy to 37.8 mAP, its inference time rises to 198ms, which is almost 4 times as slow as YOLOv3 that predicts at a mAP of 33.0 in 51ms.

Conclusion

In this report, we explored YOLO and SSD algorithms in depth. Specifically, we observed how the simplistic YOLO algorithm gradually evolved into a model that resembles SSD. We also noted the architectural differences that give each algorithm their edge when it comes to the speed and accuracy tradeoff. We concluded that for real-time applications, the superior algorithm is YOLOv3 due to its unmatched accuracy (amongst the single stage detectors) and relatively higher FPS.

The optimal configuration for real-time object detection is going to depend on the embedded system on which the project is built. The most crucial specification is with the GPU. Not only are we talking about the number of cores, bandwidth, and GPU clock speed, we also have to be careful on the size of the model as memory is probably going to be the limiting factor. SSD512 returning 24564 bounding boxes is going to have weights that will blow up the RAM on any microprocessor.

Following this report, my objective remains the same: to build an embedded object detection system for autopilot applications. I am still going to run the model on an NVIDIA Jetson Nano. I can envision myself reducing the size of the input images and maybe even modify the model by taking out some layers to reduce its dimensionality.

As a plan B, I am thinking of bringing one or two additional Coral Edge TPU to the table to accelerate the computation, but in a TensorFlow Lite framework.

Bibliography

1. W. Liu, D. Anguelov, D. Erhan, C. Szegedy, and S. Reed. SSD: Single Shot MultiBox Detector. *arXiv preprint arXiv:1512.02325, 2016.*

2. J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. *arXiv preprint arXiv:1506.02640 v4, 2015.*

3. J. Redmon, S. Divvala, R. Girshick, and A. Farhadi.YOLO9000: Better, Faster, Stronger. *arXiv preprint arXiv:1612.08242, 2016.*

4. G. Howard. Some improvements on deep convolutional neural network based image classification. *arXiv preprint arXiv:1312.5402, 2013.*

5. J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. *arXiv preprint arXiv:1804.02767, 2018.*