ImageNet Object Detection, Localization and Labeling from Video

Pratik Patre | Rahul Chandra
patre.p@husky.neu.edu, chandra.ra@husky.neu.edu
CSYE 7245, Fall 2017, Northeastern University

Abstract: This project deals with the field of computer vision, mainly for the application of deep learning in object detection task. On the one hand, there is a simple summary of the datasets and deep learning algorithms commonly used in computer vision. On the other hand, a new dataset is built according to those commonly used datasets, and choose one of the network called YOLO to work on this new dataset. Through the experiment to strengthen the understanding of these networks, and through the analysis of the results learn the importance of deep learning technology, and the importance of the dataset for deep learning.

Link to Project -

https://drive.google.com/drive/folders/1fjP8cFGRdDmgVckSfHaT1CcO9xEhQJ61?usp=sharing

I. INTRODUCTION

In recent years, with the rapid development of deep learning, many research areas have achieved good results, and accompanied by the continuous improvement of convolution neural networks, computer vision has arrived at a new peak. From the ALexNet in 2012 years to the ZF Net in 2013 years, and then to the VGG Net, the ResNet, the Inception and so on, the architecture of convolution neural network is constantly improving. In addition, the return of the convolution neural network also makes the application of computer vision greatly improve, such as face recognition, object detection, object tracking, semantic segmentation, and so on.

Object detection as one of the important applications in the field of computer vision has been the focus of research, and convolution neural network has made great progress in object detection. Object detection is developing from the single object recognition to the multi-object recognition. The meaning of the first is just from an image to identify a single object, it can be said that it is a problem of classification, and the meaning of the later is not only can identify all the objects in an image, including the exact location of the objects. Further application is implementing object detection on videos. Deep learning has formed a mainstream object recognition algorithm based on YOLO, and this algorithm is refreshing the higher accuracy in many famous datasets.

In this report, we first summarize some algorithms related to deep learning for object detection, and then apply one of the algorithms to a imageNet dataset to verify its wide applicability.

II DATASET AND CONVOLUTIONAL NEURAL NETWORKS

Dataset

Dataset is one of the foundations of deep learning, for many researchers to get enough data to carry out the experiment just by themselves is a big problem, so we need a lot of open source dataset for everyone to use. Some commonly used datasets in computer vision is the following.

ImageNet: The Imagenet dataset has more than 14 million images covering more than 20,000 categories. There are more than a million pictures with explicit class annotations and annotations of object locations in the image. The Imagenet dataset is one of the most widely used datasets in the field of deep learning. Most of the research work such as image classification, location, and detection is based on this dataset. The Imagenet dataset is detailed and is very easy to use. It is very widely used in the field of computer vision research, and has become the "standard" dataset of the current deep learning of image domain to test algorithm performance. There is a well-known challenge called "ImageNet International Computer Vision Challenge" (ILSVRC) based on the Imagenet dataset. It is worth mentioning that the winners of ILSVRC2016 are Chinese teams for all projects.

Convolutional Neural Networks

Deep learning used by the network has been constantly improving, in addition to the changes in the network structure, the more is to do some tune based on the original network or apply some trick to make the network performance to enhance. The more well-known algorithms of object detection are a series of algorithms based on R-CNN, mainly in the following.

1) *R-CNN:* Paper which the R-CNN (Regions with Convolutional Neural Network) is in has been the state-of-art papers in field of object detection in 2014 years. The idea of this paper has changed the general idea of object detection. Later, algorithms in many literatures on deep learning of object detection basically inherited this idea which is the core algorithm for object detection with deep learning. One of the most noteworthy points of this paper is that the CNN is applied to the candidate box to extract the feature vector, and the second is to propose a way to effectively train large CNNs. It is supervised pre-training on large dataset such ILSVRC, and then do some fine-tuning training in a specific range on a small dataset such PASCAL.

2) *SPP-Net*: SPP-Net is an improvement based on the R-CNN with faster speed. SPP-Net proposed a spatial pyramid pooling (SPP) layer that removes restrictions on network fixed size. SPP-Net only needs to run the convolution layer once (the whole image, regardless of size), and then use the SPP layer to extract features, compared to the R-CNN, to avoid repeat convolution operation the candidate area, reducing the number of convolution times. The speed for SPP-Net calculating the convolution on the Pascal VOC 2007 dataset by 30-170 times faster than the R-CNN, and the overall speed is 24-64 times faster than the R-CNN.

3) *Fast R-CNN*: For the shortcomings of R-CNN and SPP-Net, Fast R-CNN did the following improvements: higher detection quality (mAP) than R-CNN and SPP-Net; write the loss function of multiple tasks together to achieve single-level training process; in the training can update all the layers; do not need to store features in the disk. Fast R-CNN can improve the speed of training deeper neural networks, such as VGG16. Compared to R-CNN, the speed for Fast R-CNN training stage is 9 times faster and the speed for test is 213 times faster. The speed for Fast R-CNN training stage is 3 times faster than SPP-net and the speed for test is 10 times faster, the accuracy rate also has a certain increase.

4) *Faster R-CNN*: The emergence of SPP-net and Fast R-CNN has greatly reduced the running time of the object detection network. However, the time they take for the regional proposal method is too long, and the task of getting regional proposal is a bottleneck. Faster R-CNN presents a solution to this problem by converting traditional practices (such as Selective Search, SS) to use a deep network to compute a proposal box (such as Region Proposal Network, RPN). In this paper, our experiment chooses the Faster R-CNN network. Table I shows the comparison of mean Average Precision (mAP) of above four kinds of network structure on the VOC2007 dataset.

5) *YOLO*: YOLO9000, a state-of-the-art, real-time object detection system that can detect over 9000 object categories. Darknet-19 is the base of YOLO. Darknet-19 has 19 convolutional layers and 5 maxpooling layers. Training for classification includes training the network on the standard ImageNet 1000 class classification dataset for 160 epochs using stochastic gradient descent with a starting learning rate of 0.1. Training for detection includes modification of this network by removing the last convolutional layer and adding on three 3×3 convolutional layers with 1024 filters each followed by a final 1×1 convolutional layer with the number of outputs we need for detection.

YOLOv2 is a combined classification-bounding box prediction framework where we directly predict the objects in each cell and the corrections on anchor boxes. More specifically, YOLOv2 divides the entire image into 13X13 grid cells, next places 5 anchor boxes at each location and finally predicts corrections on these anchor boxes. YOLOv2 makes 5 predictions corresponding to corrections on location of center (x and y), height and width, and finally the intersection over union (IOU) between predicted bounding boxes and ground truth boxes. A unique feature of YOLOv2 is that all the predictions are have magnitude less than 1, as a result the chance of one type of cost dominating the optimization is less likely. A unique feature of YOLOv2 is that the anchor boxes are designed specifically for the given dataset using K-means clustering. Unlike other anchor boxes (or prior) based methods, like Single Shot Detection, YOLOv2 does not assume the aspect ratios or shapes of the boxes. As a result, the YOLOv2 in general has lower localization loss and has higher intersection over union (IOU) between the target and network prediction.

YOLOv2 is state-of-the-art and faster than other detection systems across a variety of detection datasets. Furthermore, it can be run at a variety of image sizes to provide a smooth tradeoff between speed and accuracy.

6) *COMPARISON BETWEEN THE CNN MODELS:*

Metric Used is mAP- Mean Average Precisions: Its use is different in the field of Information Retrieval and Multi-Class classification (Object Detection) settings. To calculate it for Object Detection, you calculate the average precision for each class in your data based on your model predictions. Average precision is related to the area under the precision-recall curve for a class. Then Taking the mean of these average individual-class-precision gives you the Mean Average Precision.

Mean average precision for a set of queries is the mean of the average precision scores for each query.

$$\mathrm{MAP} = \frac{\sum_{q=1}^{Q} \mathrm{AveP(q)}}{Q}$$

where Q is the number of queries.

Mean average precision (MAP) of different network on the imagenet dataset:

Network	R-CNN	SPP-Net	Fast R-CNN	Faster R-CNN	YOLOv2
VOC07 mAP	0.66	0.631	0.669	0.732	0.579

III. APPLICATION OF YOLO

Preprocessing Data: In the experiment, we must need a new dataset, to train YOLO. We have ImageNet data set which has a lot of classes and a lot of data. But to train the Yolo Model for new images and to adjust the weights again for new images, the model must be fed with images and bounding boxes coordinates. Each image will have a text file which stores the class to which the object belongs to and the coordinates of the bounding boxes in Yolov2 format.

Steps for Object Localization and Labeling in Video:

Step 1: Image Recognition and Detection in image

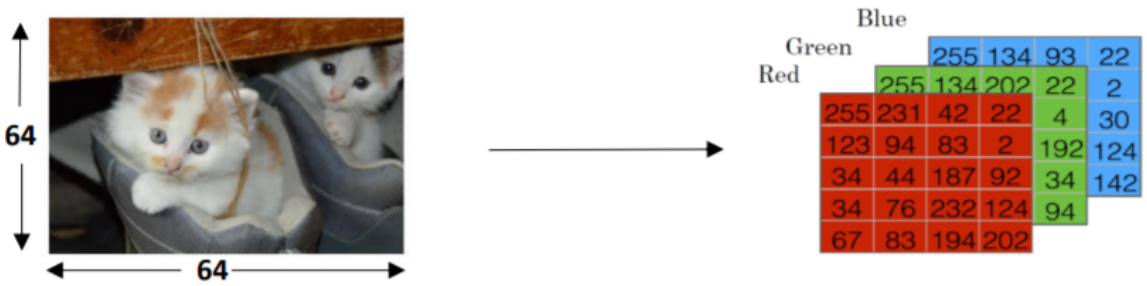
Step 2: Image localization of objects in Image

Step 3: Image labeling of objects in Image

Step 4: Object detection and localization in Video

Procedure for Object Localization and Labeling in Video

Procedure 1: Object Classifier using Numpy implementation of Neural Net (NN_Numpy.py)
Image preprocessing included Reshape the training and test data sets so that images of size (num_px, num_px, 3) are flattened into single vectors of shape (num_px ∗ num_px ∗3, 1). For example, an RGB image of 64x64 will get converted to 64 * 64 * 3 = 12288



$$x = \begin{bmatrix} 255 \\ 231 \\ 42 \\ \vdots \\ 255 \\ 134 \\ 202 \\ \vdots \\ 255 \\ 134 \\ 93 \\ \vdots \end{bmatrix} \begin{matrix} \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{matrix} \begin{matrix} \\ \\ \text{red} \\ \\ \\ \\ \text{green} \\ \\ \\ \\ \text{blue} \\ \\ \end{matrix}$$

After getting the pixel values, we initialize weights and biases of our model. We perform forward propagation by simply multiplying input with weights and add bias before applying activation function (sigmoid in here) at each node. We calculate the following formula:

$$z^{(i)} = w^T x^{(i)} + b$$

We then use the sigmoid function which is our activation function to calculate y bar.

$$\hat{y}^{(i)} = a^{(i)} = sigmoid(z^{(i)})$$

We then compute the cost function and summing up over all the training examples:

$$\mathcal{L}(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1 - y^{(i)}) \log(1 - a^{(i)})$$

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)})$$

The important part is to optimize. Optimization is always the goal whether you are dealing with a real-life problem or building a software product. I, as a computer science student, always fiddled with optimizing my code to the extent that I could brag about its fast execution. Optimization in machine learning has a slight difference. Generally, while optimizing, we know exactly how our data looks like and what areas we want to improve. But in machine learning we have no clue how our "new data" looks like, let alone try to optimize on it. So, in machine learning, we perform optimization on the training data and check its performance on a new validation data.

Gradient descent is an optimization algorithm used to find the values of parameters (coefficients) of a function (f) that minimizes a cost function (cost). Gradient descent is best used when the parameters cannot be calculated analytically (e.g. using linear algebra) and must be searched for by an optimization algorithm.

Now there are many types of gradient descent algorithms. They can be classified by two methods mainly:

- Based on data ingestion
    1. Full Batch Gradient Descent Algorithm
    2. Stochastic Gradient Descent Algorithm

In full batch gradient descent algorithms, you use whole data at once to compute the gradient, whereas in stochastic you take a sample while computing the gradient.

- Based on differentiation techniques
    1. First order Differentiation
    2. Second order Differentiation

Gradient descent requires calculation of gradient by differentiation of cost function. We can either use first order differentiation or second order differentiation.

Gradient Descent is a sound technique which works in most of the cases. But there are many cases where gradient descent does not work properly or fails to work altogether. There are three main reasons when this would happen:

1. Data challenges
2. Gradient challenges
3. Implementation challenges

Data Challenges

- If the data is arranged in a way that it poses a non-convex optimization problem. It is very difficult to perform optimization using gradient descent. Gradient descent only works for problems which have a well-defined convex optimization problem.
- Even when optimizing a convex optimization problem, there may be numerous minimal points. The lowest point is called global minimum, whereas rest of the points are called local minima. Our aim is to go to global minimum while avoiding local minima.
- There is also a saddle point problem. This is a point in the data where the gradient is zero but is not an optimal point. We don't have a specific way to avoid this point and is still an active area of research.

Gradient Challenges

- If the execution is not done properly while using gradient descent, it may lead to problems like vanishing gradient or exploding gradient problems. These problems occur when the gradient is too small or too large. And because of this problem the algorithms do not converge.

Implementation Challenges

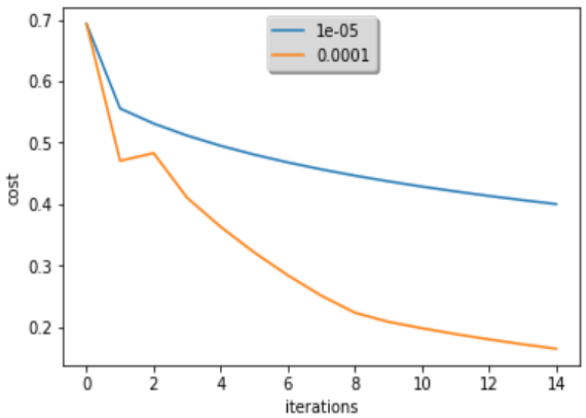
- Most of the neural network practitioners don't generally pay attention to implementation, but it's very important to look at the resource utilization by networks. For e.g.: When implementing gradient descent, it is very important to note how many resources you would require. If the memory is too small for your application, then the network would fail.
- Also, it's important to keep track of things like floating point considerations and hardware /software prerequisites.

Finally, after optimizing Cost function i.e. after 2000 iterations, we get the following accuracy.

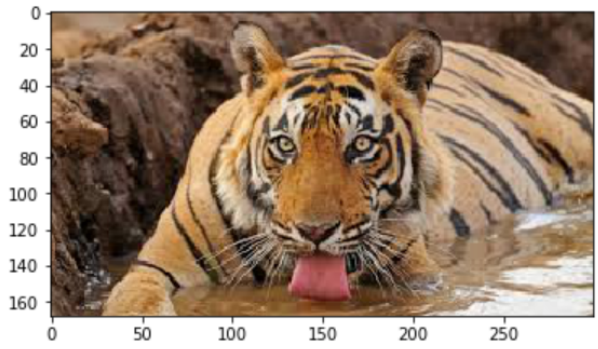
```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.470520
Cost after iteration 200: 0.482923
Cost after iteration 300: 0.410153
Cost after iteration 400: 0.362819
Cost after iteration 500: 0.321208
Cost after iteration 600: 0.283985
Cost after iteration 700: 0.250734
Cost after iteration 800: 0.223059
Cost after iteration 900: 0.208178
Cost after iteration 1000: 0.197709
Cost after iteration 1100: 0.188215
Cost after iteration 1200: 0.179551
Cost after iteration 1300: 0.171613
Cost after iteration 1400: 0.164312
Cost after iteration 1500: 0.157577
Cost after iteration 1600: 0.151344
Cost after iteration 1700: 0.145560
Cost after iteration 1800: 0.140180
Cost after iteration 1900: 0.135163
train accuracy: 98.09825673534073 %
test accuracy: 75.31645569620252 %
```

We tested the model on two learning rates 0.0001 and 0.00001 and compared the accuracy level.

```
learning rate is: 1e-05
train accuracy: 83.35974643423138 %
test accuracy: 73.41772151898735 %

-------------------------------------------------------

learning rate is: 0.0001
train accuracy: 97.62282091917591 %
test accuracy: 75.9493670886076 %

-------------------------------------------------------
```

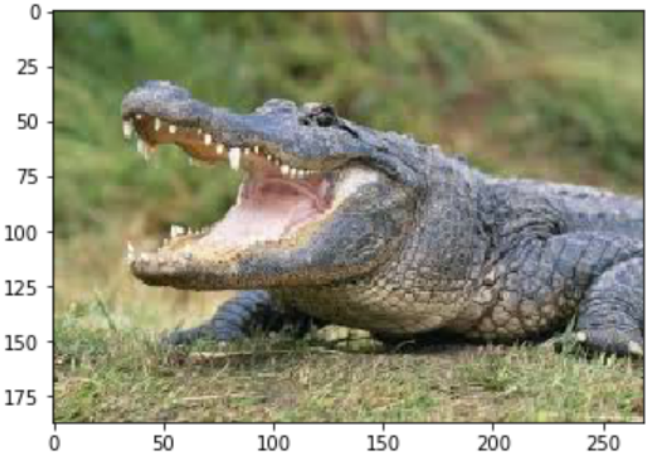


The model classified birds correctly with an accuracy of 75%. The output shows y = 1 as bird and y = 0 as not a bird.



```
y = 0.0
```

```
y = 1.0
```



```
y = 1.0
```

Procedure 2: Object detection using Tensor Flow and CNN. (CNN_tensorflow.py)

Stride: Stride controls how the filter convolves around the input volume. In the example, we had in part 1, the filter convolves around the input volume by shifting one unit at a time. The amount by which the filter shifts is the stride. In that case, the stride was implicitly set at 1. Stride is normally set in a way so that the output volume is an integer and not a fraction.

Padding: Zero padding pads the input volume with zeros around the border. This is usually used when we do not want input image size to decrease.

ReLU (Rectified Linear Units) Layers: After each conv layer, it is convention to apply a nonlinear layer (or **activation layer**) immediately afterward. The purpose of this layer is to introduce nonlinearity to a system that basically has just been computing linear operations during the conv layers (just element wise multiplications and summations). In the past, nonlinear functions like tanh and sigmoid were used, but researchers found out that **ReLU layers** work far better because the network can train a lot faster (because of the computational efficiency) without making a significant difference to the accuracy. It also helps to alleviate the vanishing gradient problem, which is the issue where the lower layers of the network train very slowly because the gradient decreases exponentially through the layers. (The ReLU layer applies the function f(x) = max (0, x) to all of the values in the input volume. In basic terms, this layer just changes all the negative activations to 0. This layer increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the conv layer.

Pooling Layers: After some ReLU layers, programmers may choose to apply a pooling layer. It is also referred to as a down sampling layer. In this category, there are also several layer options, with maxpooling being the most popular. This basically takes a filter (normally of size 2x2) and a stride of the same length. It then applies it to the input volume and outputs the maximum number in every subregion that the filter convolves around.
Other options for pooling layers are average pooling and L2-norm pooling. The intuitive reasoning behind this layer is that once we know that a specific feature is in the original input volume (there will be a high activation value), its exact location is not as important as its relative location to the other features. As you can imagine, this layer drastically reduces the spatial dimension (the length and the width change but not the depth) of the input volume. This serves two main purposes. The first is that the number of parameters or weights is reduced by 75%, thus lessening the computation cost. The second is that it will control **overfitting**. This term refers to when a model is so tuned to the training examples that it is not able to generalize well for the validation and test sets. A symptom of overfitting is having a model that gets 100% or 99% on the training set, but only 50% on the test data.

The formula for calculating the output size for any given conv layer is

$$O = \frac{(W - K + 2P)}{S} + 1$$

where O is the output height/length, W is the input height/length, K is the filter size, P is the padding, and S is the stride.

Steps for object detection using CNN:

We create placeholders for the input image X and the output Y using
X = tf.placeholder(tf.float32, [None, n_H0, n_W0, n_C0], name = "Placeholder")
Y = tf.placeholder(tf.float32, [None, n_y], name = "Placeholder")

We initialize weights/filters W1 and W2 using
tf.contrib.layers.xavier_initializer(seed = 0)
We implement forward propagation by using the following built in functions to build the following model: CONV2D -> RELU -> MAXPOOL -> CONV2D -> RELU -> MAXPOOL -> FLATTEN -> FULLYCONNECTED.

- **tf.nn.conv2d(X,W1, strides = [1,s,s,1], padding = 'SAME'):** given an input X and a group of filters W1, this function convolves W1's filters on X. The third input ([1, f, f, 1]) represents the strides for each dimension of the input (m, n_H_prev, n_W_prev, n_C_prev).
- **tf.nn.max_pool(A, ksize = [1,f,f,1], strides = [1,s,s,1], padding = 'SAME'):** given an input A, this function uses a window of size (f, f) and strides of size (s, s) to carry out max pooling over each window.
- **tf.nn.relu(Z1):** computes the elementwise ReLU of Z1 (which can be any shape).
- **tf.contrib.layers.flatten(P)**: given an input P, this function flattens each example into a 1D vector it while maintaining the batch-size. It returns a flattened tensor with shape [batch_size, k].
- **tf.contrib.layers.fully_connected(F, num_outputs):** given a flattened input F, it returns the output computed using a fully connected layer.

In detail, we use the following parameters for all the steps:

- Conv2D: stride 1, padding is "SAME"

- ReLU

- Max pool: Use an 8 by 8 filter size and an 8 by 8 stride, padding is "SAME"

- Conv2D: stride 1, padding is "SAME"

- ReLU

- Max pool: Use a 4 by 4 filter size and a 4 by 4 stride, padding is "SAME"

- Flatten the previous output.

- FULLYCONNECTED (FC) layer.

We compute the cost as below:

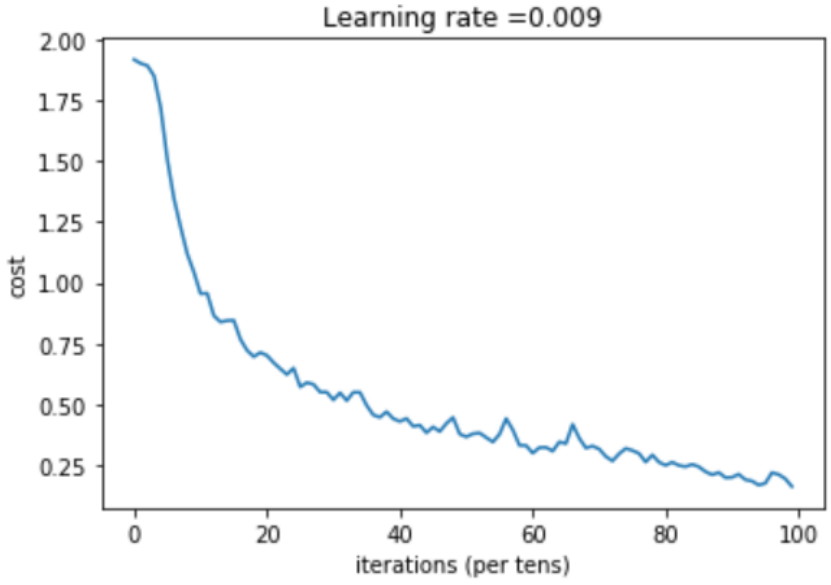
- **tf.nn.softmax_cross_entropy_with_logits(logits = Z3, labels = Y):** computes the softmax entropy loss. This function both computes the softmax activation function as well as the resulting loss.
- **tf.reduce_mean:** computes the mean of elements across dimensions of a tensor. Use this to sum the losses over all the examples to get the overall cost.

Further we create a model which implements the below functions:

- create placeholders
- initialize parameters
- forward propagate
- compute the cost
- create an optimizer

We run the model for 100 epochs and learning rate as 0.009 which took us more than three hours to train.

```
Cost after epoch 0: 1.917929
Cost after epoch 5: 1.506757
Cost after epoch 10: 0.955359
Cost after epoch 15: 0.845802
Cost after epoch 20: 0.701174
Cost after epoch 25: 0.571977
Cost after epoch 30: 0.518435
Cost after epoch 35: 0.495806
Cost after epoch 40: 0.429827
Cost after epoch 45: 0.407291
Cost after epoch 50: 0.366394
Cost after epoch 55: 0.376922
Cost after epoch 60: 0.299491
Cost after epoch 65: 0.338870
Cost after epoch 70: 0.316400
Cost after epoch 75: 0.310413
Cost after epoch 80: 0.249549
Cost after epoch 85: 0.243457
Cost after epoch 90: 0.200031
Cost after epoch 95: 0.175452
```




```
Tensor("Mean_1:0", shape=(), dtype=float32)
Train Accuracy: 0.940741
Test Accuracy: 0.783333
```

Procedure 3: Object Localization and Labelling in video using the YOLO model. (CNN_YOLO.py)

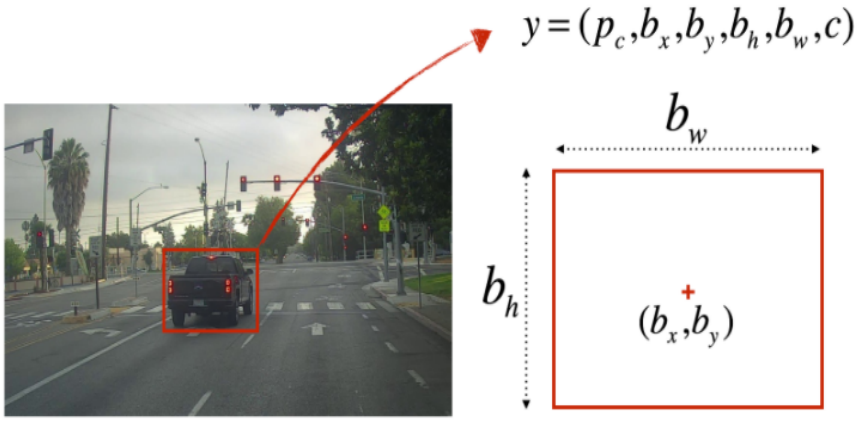
Breaking the Video into Frames:

In [5]:
```
import cv2
import numpy as np
import os
```

In [12]:
```
# Playing video from file:
cap = cv2.VideoCapture('Initial_Video.mp4')

try:
    if not os.path.exists('images'):
        os.makedirs('images')
except OSError:
    print ('Error: Creating directory of data')

currentFrame = 0
while(True):
    # Capture frame-by-frame
    ret, frame = cap.read()

    # Saves image of the current frame in jpg file
    name = './data/frame' + str(currentFrame) + '.jpg'
    print ('Creating...' + name)
    cv2.imwrite(name, frame)

    # To stop duplicate images
    currentFrame += 1
```

The **input** is a batch of images of shape (m, 608, 608, 3)

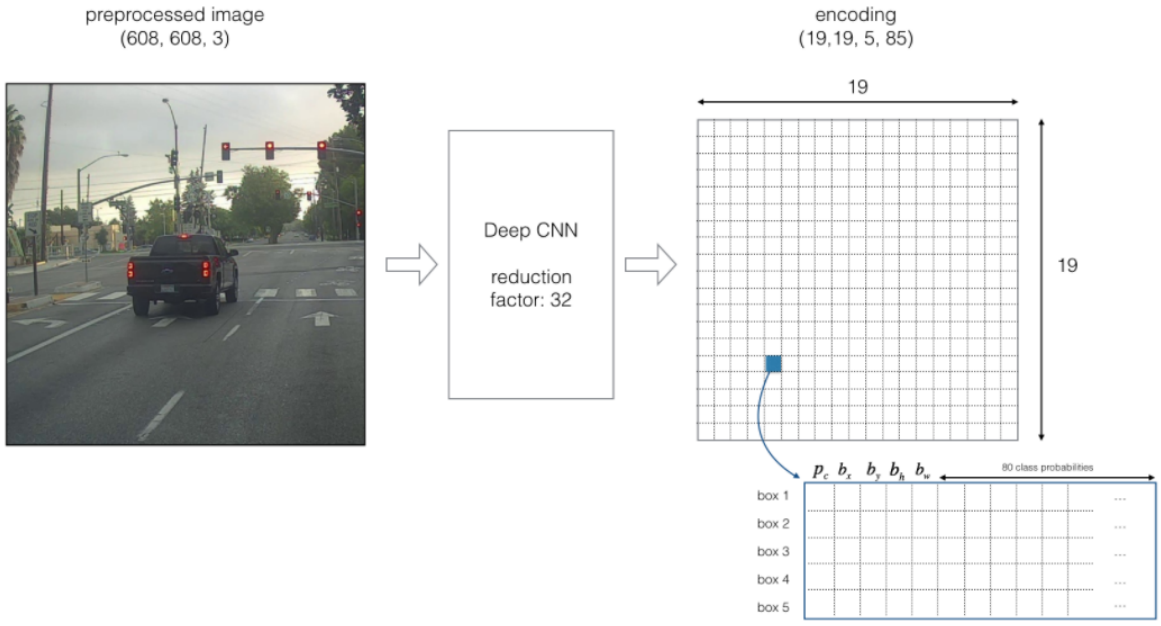
The **output** is a list of bounding boxes along with the recognized classes. Each bounding box is represented by 6 numbers (pc,bx,by,bh,bw,c) If you expand c into an 80-dimensional vector, each bounding box is then represented by 85 numbers.



$p_c = 1$: confidence of an object being present in the bounding box

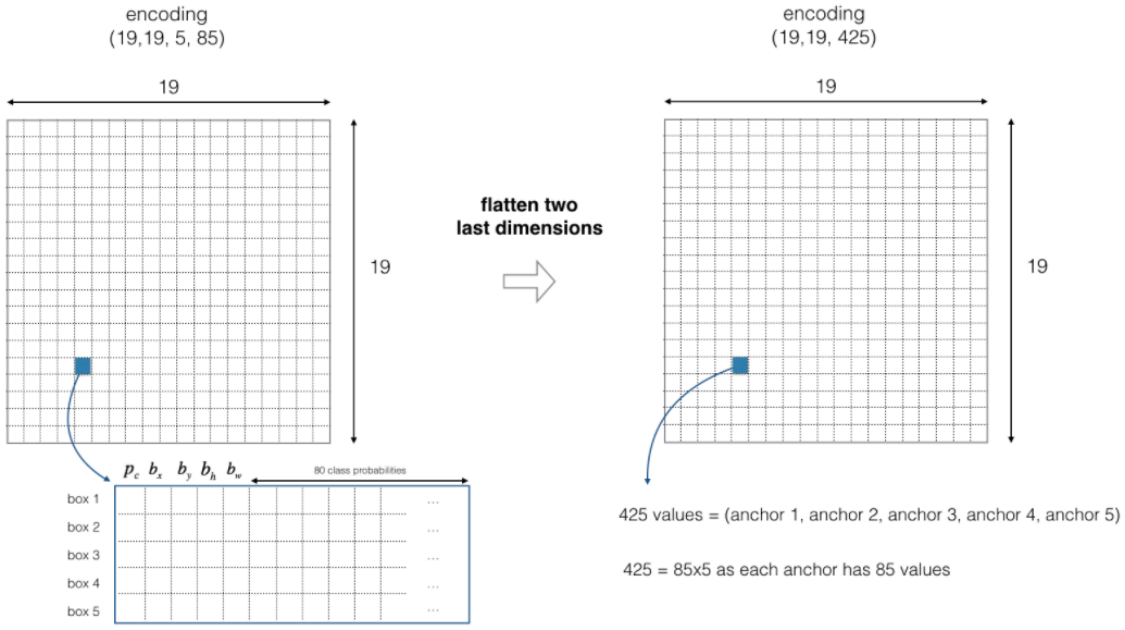
$c = 3$: class of the object being detected (here 3 for "car")

We have used 5 anchor boxes. So, we can think of the YOLO architecture as the following: IMAGE (m, 608, 608, 3) -> DEEP CNN -> ENCODING (m, 19, 19, 5, 85).

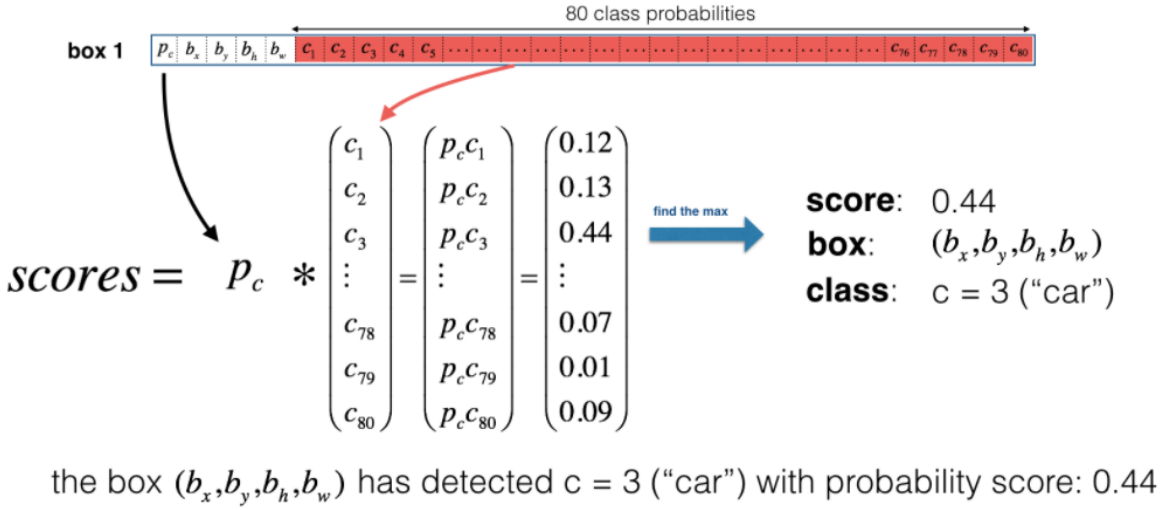


If the center/midpoint of an object falls into a grid cell, that grid cell is responsible for detecting that object. Since we are using 5 anchor boxes, each of the 19 x19 cells thus encodes information about 5 boxes. Anchor boxes are defined only by their width and height.

For simplicity, we will flatten the last two last dimensions of the shape (19, 19, 5, 85) encoding. So, the output of the Deep CNN is (19, 19, 425).



Now, for each box (of each cell) we will compute the following elementwise product and extract a probability that the box contains a certain class.



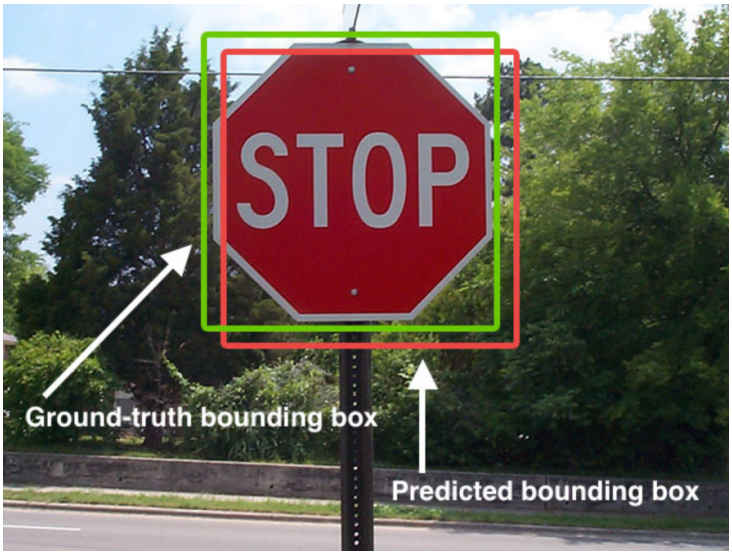
The algorithm has three basic functions to implement bounding boxes.

1) Thresholding: We get rid of any box which has a score less than a chosen threshold. The model gives tensor of shape 19x19x5x85 numbers. It'll be convenient to rearrange the (19,19,5,85) (or (19,19,425)) dimensional tensor into the following variables:

- box_confidence: tensor of shape (19×19,5,1) containing pc (confidence probability that there's some object) for each of the 5 boxes predicted in each of the 19x19 cells.
- boxes: tensor of shape (19×19,5,4) containing (bx,by,bh,bw) for each of the 5 boxes per cell.
- box_class_probs: tensor of shape (19×19,5,80) containing the detection probabilities (c1, c2, ...c80) for each of the 80 classes for each of the 5 boxes per cell.

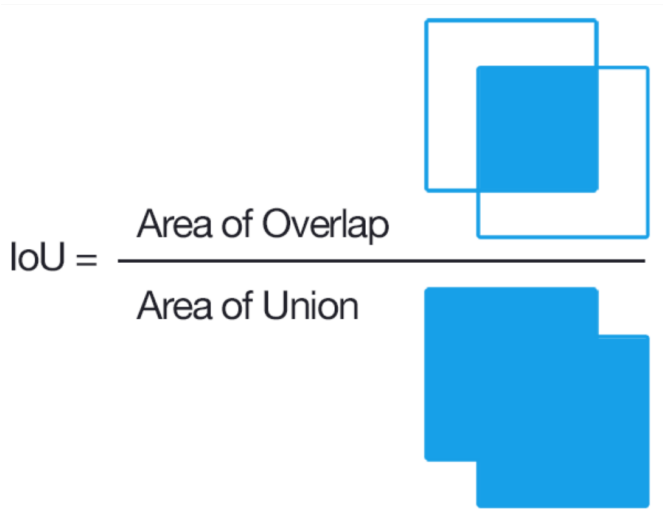
2) Intersection Over Union: Intersection over Union is an evaluation metric used to measure the accuracy of an object detector on a dataset. More formally, to apply Intersection over Union to evaluate an (arbitrary) object detector we need:

- The *ground-truth bounding boxes* (i.e., the hand labeled bounding boxes from the testing set that specify *where* in the image our object is).
- The *predicted bounding boxes* from our model.



In the figure above we can see that our object detector has detected the presence of a stop sign in an image.

The *predicted* bounding box is drawn in *red* while the *ground-truth* (i.e., hand labeled) bounding box is drawn in green.
Computing Intersection over Union can therefore be determined via:



In the numerator, we compute the ***area of overlap*** between the *predicted* bounding box and the *ground-truth* bounding box.

The denominator is the ***area of union***, or more simply, the area encompassed by *both* the predicted bounding box and the ground-truth bounding box.

Dividing the area of overlap by the area of union yields our final score — *the Intersection over Union.*

3) Non-Max Suppression: Even after filtering by thresholding over the classes scores, we still end up a lot of overlapping boxes. A second filter for selecting the right boxes is called non-maximum suppression (NMS).

Non-max suppression uses the very important function called **"Intersection over Union"**, or IoU.

The key steps are:

1. Select the box that has the highest score.
2. Compute its overlap with all other boxes, and remove boxes that overlap it more than iou_threshold.
3. Go back to step 1 and iterate until there's no more boxes with a lower score than the current selected box.

We load Yolo Model with its pretrained weights.

```
yolo_model.summary()
```

```
__________________________________________________________________________________________________
Layer (type)                    Output Shape          Param #     Connected to
==================================================================================================
input_1 (InputLayer)            (None, 608, 608, 3)   0
__________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 608, 608, 32)  864         input_1[0][0]
__________________________________________________________________________________________________
batch_normalization_1 (BatchNorm (None, 608, 608, 32)  128         conv2d_1[0][0]
__________________________________________________________________________________________________
leaky_re_lu_1 (LeakyReLU)       (None, 608, 608, 32)  0           batch_normalization_1[0][0]
__________________________________________________________________________________________________
max_pooling2d_1 (MaxPooling2D)  (None, 304, 304, 32)  0           leaky_re_lu_1[0][0]
__________________________________________________________________________________________________
conv2d_2 (Conv2D)               (None, 304, 304, 64)  18432       max_pooling2d_1[0][0]
__________________________________________________________________________________________________
batch_normalization_2 (BatchNorm (None, 304, 304, 64)  256         conv2d_2[0][0]
__________________________________________________________________________________________________
leaky_re_lu_2 (LeakyReLU)       (None, 304, 304, 64)  0           batch_normalization_2[0][0]
__________________________________________________________________________________________________
max_pooling2d_2 (MaxPooling2D)  (None, 152, 152, 64)  0           leaky_re_lu_2[0][0]
__________________________________________________________________________________________________
conv2d_3 (Conv2D)               (None, 152, 152, 128) 73728       max_pooling2d_2[0][0]
__________________________________________________________________________________________________
```

Forming Video from the processed Images:

In [19]:
```
from PIL import Image
import cv2
import numpy as np
import os

Data = []
D =[]
time = 0
import glob
for name in glob.glob(os.getcwd()+'//images/*'):
    D.append(name)
D.sort()
for i in D:
    Data.append([time,i])
    time = time + 0.25
Data = Data[::-1]

FPS = 60 # Sets the FPS of the entire video
currentFrame = 0 # The animation hasn't moved yet, so we're going to leave it as zero
startFrame = 0 # The animation of the "next" image starts at "startFrame", at most
trailingSeconds = 0.0005 # Sets the amount of time we give our last image (in seconds)
blendingDuration = 0.00002 # Sets the amount of time that each transition should last for
                           # This could be more dynamic, but for now, a constant transition period is chosen
blendingStart = 1 # Sets the time in which the image starts blending before songFile

for i in Data:
    i[0] = i[0] * FPS # Makes it so that iterating frame-by-frame will result in properly timed slideshows
```

IV. RESULTS

After performing the algorithm and running yolo model, we get the following output:

Image and Video Before Processing:

Initial_Video.mp4

Image and Video After Processing:

Video_with_Labeling_and_Localisation.mp4

Acknowledgment:

We would like to thank to Professor Nik Bear Brown for supporting us and guiding us throughout the project.

References:

[1]. Andrew Ng, Kian Katanforoosh, Younes Bensouda Mourri "Convolutional Neural Networks"
[2]. Amir-Abbas Sadeghian Pranav Rajpurkar, Ramtin Keramati, Timon Ruban "MIT Research papers"
[3]. PJReddie "YOLO-Real-Time Object Detection" https://pjreddie.com/darknet/yolo/
[4]. Alexey https://timebutt.github.io/static/how-to-train-yolov2-to-detect-custom-objects/