Neural Computation Assignment

Brendan Case - bkc721 - 1801421

Guy Coop - gtc434 - 1447634

Vasileios Mizaridis - vxm716 - 1844216

Priyanka Mohata - pxm374 - 1341274

Liangye Yu - lxy736 - 1810736

Abstract

Image Recognition has, in recent history been relatively "solved" in the sense that algorithms have now been recorded outperforming humans in simple image recogntion tasks. However regional image recognition where the algorithm is tasked with recognizing multiple images inside a whole chaotic scene is still an emerging field. Our team analysed the most promising methods of regional image recognition, and implemented our own solution to a simplified regional recognition task. Our solution was able to locate and classify objects with some degree of accuracy, however was not able to submit an accepted result to the kaggle competition.

1 Introduction

For this project, our team was assigned the task of implementing a regional image recognition system.

1.1 Data Sets

The data set provided was given in the following format:

- Data: 400x400 RGB image files in .jpg format
- Labels: each image has a corresponding text file that describes the location of each of the predefined objects in the image. This location was given as pairs of integers describing horizontal runs of pixels that form a rectangular bounding box around the object. If the object was not present in the image it was given as [object 1 0] meaning that it had a run length of 0 pixels.

1.1.1 Training Set:

The training set provided contains 14,625 (image, label) pairs that includes image files and label files that describe the objects and bouding boxes. for conducting experiments, this data set should be subdivided into a Training set and a Validation set to measure how parameter changes affect the accuracy of the results.

1.1.2 Test Set:

The test set contained 2500 images that are given to simulate unseen data coming into the system. These images do not have the associated label files, and our task is to produce label files that describe the location and class of objects in each of the test image files.

1.2 Our Aims

Our aims for this project are as follows:

- Discover and Analyse currently exisitng method of performing regional image recognition
- Produce our own implementation of one of these methods, using any necessary packages or source code segments as necessary.
- Conduct experiments to optimize the recognition system in terms of bounding box locations, and object classification.
- Produce a set of conclusions about the effectiveness of the various recognition methods, and the optimal parameters of our implementation for the data set provided.

2 Design

For this task our Initial instict was to attempt to solve it with a simple feedforward neural network. And whilst this would have been adequate for a simple classification task, we determined it would not provide a good solution to this region bounded classification task. From there we exaimned other possible methods of solving the task. The first sub-method we investigated was using Selective Search to identify regions of interest inside the image. We then expanded this to determine other pre-established algorithms that make use of selective search and how they compare to other similar algorithms.

2.1 Selective Search

Selective Search is an algorithm used for regional image searching. It works by performing image processing to segment an image by multiple factors such as: Colour, Texture, an Brightness, in order to try and seperate multiple object inside an image. once these sections have been seperated. rectangular bounding boxes can be formed around the objects so that they can be passed to an image recognizer network.

2.2 Regional Convolutional Neural Networks (RCNNs)

Given the regional nature of this task, The first option that should be analysed is "Regional Convultional Neural Networks" and their successors. There are three implementations of this algorithm that will be examined:

- R-CNN [GDDM14]
- Fast R-CNN [Gir15]
- Faster R-CNN [RHGS15]

2.2.1 R-CNN

R-CNN [GDDM14] makes use of selective search to generate the region proposals. it typically produces around 2000 region proposals per image, these regions are then sent forward to the CNN in order to determine if they contain an object in the dataset, and what that object is. Once the objects have been detected in the bounding boxes, regression algorithms are used to tighten the bounding boxes more accurately around the objects.

2.2.2 Fast R-CNN

Fast R-CNN [Gir15] is an update on the original R-CNN technique that was developed in 2015, it acheives approximately a 9x speed-up on the original at train time, and over 200x speed-up at test time. It does this by unifying the training phase of the boudinding boxes, and the object classification algorithm into a single round of training, rather than having to train the two algorithms seperately. It also makes use of Region-of-Interest (RoI) pooling layers. "RoI max pooling works by reducing the $h * w$ RoI window into an HW *grid of sub-windows of approximate size h/H* w/W and then max-pooling the values in each sub-window into the corresponding output grid cell." [Gir15]

2.2.3 Faster R-CNN

Faster R-CNN [RHGS15] is another significat update on the Fast R-CNN technique that acheives another dramatic speedup. This algorithm was designed as part of an attempt at real time regional image recognition, and as such is able to operate in almost real time. Similar to Fast R-CNN it makes use of the RoI pooling layer to create a significant imporvement in performance over traditional R-CNN. Faster R-CNN also introduces a Region Proposal Network (RPN). The RPN shares convolutional features with the detection network. This allows it to provide almost "cost-free" region proposals to the system. Fast R-CNN is recorded to be able to operate at approximately 5 frames-per-second (fps) when running on a GPU, meaning that it could be used for real time applications.

2.3 You Only Look Once (YOLO)

Another option we explored is a technique called “You Only Look Once” (YOLO) [RF16]. YOLO was named as such because the algorithm centers around only performing a single pass across the image, rather than having to analyse the same data multiple times. This techniques, introduced in 2015, was unlike any other object detection model currently in-use. Unlike R-CNN techniques which pass the images through multiple networks, YOLO only requires one network evaluation.

In 2016, YOLO9000(also called YOLOv2) was introduced which had most of the same functionality offered in the initial version of YOLO, however it was modified to improve the localization and recall issue present in YOLO while trying to maintain the classification accuracy and speed. In [RF16] the YOLOv2 technique is recorded to be able to operate close to 45 frames-per-second (fps) on a normal CPU. This is faster than the original YOLO and also it understands more generalized object representations.

3 Implementation

A first attempt resembled the original R-CNN algorithm outlined in [GDDM14]. This was chosen from the initial intuition that by training a CNN with 3 convolutional layers and 2 fully connected layers, we could achieve good accuracy for the classification component of the problem on simple image sets such as MNIST. This CNN was implemented using the layers module in TensorFlow [Goo17]. We felt perhaps decent accuracy could be obtained by using this trained CNN on multiple proposed regions, then giving these regions a ’score’ based on how confident the classification was, and taking the best scoring region among largely intersecting regions as the output region. This is essentially the R-CNN algorithm. We decided to use the same region proposal algorithm, SelectionSearch, as the original authors, but with different parameters to reduce region proposals

and favor larger bounding boxes. Finding which parameters favored these preferences was one source of experimentation throughout the project. In addition, we found simply removing region proposals with certain extreme aspect ratios was a reasonable assumption for this data set.

These regions, along with the regions provided in the training text files, were converted to 120 * 120 arrays of rgb tuples using Skimage (SciKit Image processing library [sidt17]) and numpy reshaping tools, which could then be used to train the CNN.

In the interest of time and sanity, this implementation was largely discarded in favor of the more ready-to-use implementation of YOLO from Darknet. In order to comply with the requirement to implement code in Python, we made use of a Python wrapper provided by Darknet. To use this, we wrote a script which loaded a trained network with provided weights and made predictions for each image in the test set, appending the predictions in the proper format to a submission.txt file, used to submit our predictions. We also adjusted one of the provided networks, 'tiny-yolo,' and trained our own weights using the training image and labels provided; however, given the limited time these weights never reached the point of having an average loss below 100.

The following class: "data_handling" includes: - a settable image_size variable that can be changed for different data sets. - a full list of the names of each object that is part of the data set. - a method to get a dictionary of corner locations of bouding boxes based on the label files given in the dataset. - a pair of methods to convert the full set of label files to the style used by YOLO. - a method to convert the output of our systems to the label format required by the data set. - a method to load the data set into the program to be manipulated.

```
In [5]: import numpy as np
        import os
        import skimage.io as skio
        import skimage.transform as transform

        class data_handler:
        # A class to manage data type changes from one format to another:
        # Manages input text labels into format used by predesigned networks
        # Also manages outputs produced and converts them to the output format.

            image_size = [400, 400]

            object_list = [
                "aeroplane",
                "bicycle",
                "bird",
                "boat",
                "bottle",
                "bus",
                "car",
                "cat",
                "chair",
                "cow",
                "diningtable",
                "dog",
                "horse",
                "motorbike",
```

```
    "person",
    "pottedplant",
    "sheep",
    "sofa",
    "train",
    "tvmonitor"
]


@staticmethod
def get_bounding_boxes(label):
    # from an input pixel list file, generate a dictionary of bounding boxes
    # the dictionary contains a list of TL, BR bounding boxes under each bounding_bo
    # i.e. bounding boxes['aeroplane'] will return a list of all the bounding boxes
    def pixel_list_to_bboxes(pixel_list):
        pixel_loc_corners = []
        bboxes = []
        #start pixel location, current left pixel location, run length
        if pixel_list[0][0] == 1 and pixel_list[0][1] == 0:
            return bounding_boxes
        pixel_loc_corners.append([pixel_list[0][0], pixel_list[0][0], pixel_list[0][
        for pair in pixel_list[1:]:
            break_var = 0
            for ind, objs_found in enumerate(pixel_loc_corners):
                if objs_found[1] + data_handler.image_size[1] == pair[0]:
                    # not a new object
                    pixel_loc_corners[ind][1] = pair[0]
                    break_var = 1
                    break
            if break_var == 0:
                pixel_loc_corners.append([pair[0], pair[0], pair[1]])

        # finished building pixel_loc_corners
        for corner in pixel_loc_corners:
            x1 = ((corner[0]) % data_handler.image_size[0]) + 1
            y1 = int(corner[0] / data_handler.image_size[1])
            x2 = ((corner[1] + corner[2] - 1) % data_handler.image_size[0]) + 1
            y2 = int((corner[1] + corner[2]) / data_handler.image_size[1])
            bboxes.append([[x1, y1],[x2, y2]])

        return bboxes


    pixel_list_dict = {}
    for ind, line in enumerate(label.readlines()):
        pixel_list_dict[data_handler.object_list[ind]] = map(int, (line.split(',')[1
        pixel_list_dict[data_handler.object_list[ind]] = np.reshape(
            pixel_list_dict[data_handler.object_list[ind]],
```

```
                [len(pixel_list_dict[data_handler.object_list[ind]])/2, 2])

        bounding_boxes = {}
        for obj in data_handler.object_list:
            bounding_boxes[obj] = pixel_list_to_bboxes(pixel_list_dict[obj])

        return bounding_boxes


    @staticmethod
    def get_yolo_text_files(input_file_location, output_file_location):
        # Reformats a single text data file into the format required to train for the YO
        output_string = ""
        f = open(input_file_location)
        bboxes = data_handler.get_bounding_boxes(f)
        for ind, obj in enumerate(data_handler.object_list):
            if bboxes[obj] != bboxes:
                for bbox in bboxes[obj]:
                    x = 0.00125 * (float(bbox[0][0]) + float(bbox[1][0]))
                    y = 0.00125 * (float(bbox[0][1]) + float(bbox[1][1]))
                    width = (float(bbox[1][0]) - float(bbox[0][0])) / 400.
                    height = (float(bbox[1][1]) - float(bbox[0][1])) / 400.
                    output_string += (str(ind) + " " + str(x) + " " + str(y) + " " + str

        outfile = open(output_file_location, 'w')
        outfile.write(output_string)
        outfile.close()
        return output_string

    @staticmethod
    def batch_create_yolo_labels(input_dir, output_dir):
        # converts a full set of labels into the yolo required format
        txt_files = [file for file in os.listdir(input_dir) if file.endswith(".txt")]
        for file in txt_files:
            data_handler.get_yolo_text_files(input_dir + file, output_dir + file)
        return

    @staticmethod
    def generate_output_file(bounding_boxes, filename):
        #generate an output pixel list from the bounding boxes dictionary
        # inputs: a list of bounding boxes for a single image
        #         the file name of the image, i.e. ('2007_00042.jpg')
        #
        # output: a string that contains the entire output information for the file

        def get_one_pixel_list(bounding_box):
            # generates a list of pixels for a rectangular bounding box
            pixel_list = []
```

```
            h_run = bounding_box[1][0] - bounding_box[0][0]
            for j in range(bounding_box[0][1], bounding_box[1][1]):
                # perform vertical steps down
                pixel_list.append([bounding_box[0][0] + (data_handler.image_size[1] * j)
            return pixel_list

        def sort_pixel_list(pixel_list):
            # sorts pixel list and manages possible overlaps
            pixel_list = sorted(pixel_list, key=lambda x:x[0])

            #check if overlap is possible, if so perform compressions stage
            return pixel_list

        def str_pixel_list(pixel_list):
            # convert a pixel list into a space seperated string
            pix_string = ""
            for pair in pixel_list:
                pix_string += str(pair[0]) + " " + str(pair[1]) + " "
            return pix_string

        def get_full_pixel_list(bounding_boxes):
            if bounding_boxes == []:
                return [[0, 1]]

            else:
                pixel_list = []
                for bounding_box in bounding_boxes:
                    pixel_list += get_one_pixel_list(bounding_box)

                pixel_list = sort_pixel_list(pixel_list)
                return pixel_list

        output_string = ""
        for obj in data_handler.object_list:
            output_string = output_string + filename + "_" + obj + ","
            pixel_list = get_full_pixel_list(bounding_boxes[obj])
            output_string += str_pixel_list(pixel_list)
            output_string += '\n'

        return output_string


    @staticmethod
    def get_training_data():
        labels = []
        data = []
        for f in os.listdir(data_handler.file_path.format('train'))[:100]:
            if f.count('.txt'):
```

```
                    bboxes = data_handler.get_bounding_boxes(open(data_handler.file_path.for
                    img = data_handler.file_path.format('train') + '/' + f.replace('.txt', '
                    label, datum = data_handler.build_training_array_single(bboxes, skio.imr
                    labels.extend(label)
                    data.extend(datum)
            return data, labels
```

The following code is a python wrapper for the YOLO image recognition system: taken from the darknet source code ../python/

```
In [6]: from ctypes import *
        import math
        import random
        import os

        def sample(probs):
            s = sum(probs)
            probs = [a/s for a in probs]
            r = random.uniform(0, 1)
            for i in range(len(probs)):
                r = r - probs[i]
                if r <= 0:
                    return i
            return len(probs)-1

        def c_array(ctype, values):
            arr = (ctype*len(values))()
            arr[:] = values
            return arr

        class BOX(Structure):
            _fields_ = [("x", c_float),
                        ("y", c_float),
                        ("w", c_float),
                        ("h", c_float)]

        class IMAGE(Structure):
            _fields_ = [("w", c_int),
                        ("h", c_int),
                        ("c", c_int),
                        ("data", POINTER(c_float))]

        class METADATA(Structure):
            _fields_ = [("classes", c_int),
                        ("names", POINTER(c_char_p))]
```

```
#lib = CDLL("/home/pjreddie/documents/darknet/libdarknet.so", RTLD_GLOBAL)
lib = CDLL("../libdarknet.so", RTLD_GLOBAL)
lib.network_width.argtypes = [c_void_p]
lib.network_width.restype = c_int
lib.network_height.argtypes = [c_void_p]
lib.network_height.restype = c_int

predict = lib.network_predict
predict.argtypes = [c_void_p, POINTER(c_float)]
predict.restype = POINTER(c_float)

set_gpu = lib.cuda_set_device
set_gpu.argtypes = [c_int]

make_image = lib.make_image
make_image.argtypes = [c_int, c_int, c_int]
make_image.restype = IMAGE

make_boxes = lib.make_boxes
make_boxes.argtypes = [c_void_p]
make_boxes.restype = POINTER(BOX)

free_ptrs = lib.free_ptrs
free_ptrs.argtypes = [POINTER(c_void_p), c_int]

num_boxes = lib.num_boxes
num_boxes.argtypes = [c_void_p]
num_boxes.restype = c_int

make_probs = lib.make_probs
make_probs.argtypes = [c_void_p]
make_probs.restype = POINTER(POINTER(c_float))

detect = lib.network_predict
detect.argtypes = [c_void_p, IMAGE, c_float, c_float, c_float, POINTER(BOX), POINTER(POI

reset_rnn = lib.reset_rnn
reset_rnn.argtypes = [c_void_p]

load_net = lib.load_network
load_net.argtypes = [c_char_p, c_char_p, c_int]
load_net.restype = c_void_p

free_image = lib.free_image
free_image.argtypes = [IMAGE]

letterbox_image = lib.letterbox_image
```

```
    letterbox_image.argtypes = [IMAGE, c_int, c_int]
    letterbox_image.restype = IMAGE

    load_meta = lib.get_metadata
    lib.get_metadata.argtypes = [c_char_p]
    lib.get_metadata.restype = METADATA

    load_image = lib.load_image_color
    load_image.argtypes = [c_char_p, c_int, c_int]
    load_image.restype = IMAGE

    rgbgr_image = lib.rgbgr_image
    rgbgr_image.argtypes = [IMAGE]

    predict_image = lib.network_predict_image
    predict_image.argtypes = [c_void_p, IMAGE]
    predict_image.restype = POINTER(c_float)

    network_detect = lib.network_detect
    network_detect.argtypes = [c_void_p, IMAGE, c_float, c_float, c_float, POINTER(BOX), POI


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

    OSError                                   Traceback (most recent call last)

    <ipython-input-6-3f4888eb93db> in <module>()
     38
     39 #lib = CDLL("/home/pjreddie/documents/darknet/libdarknet.so", RTLD_GLOBAL)
---> 40 lib = CDLL("../libdarknet.so", RTLD_GLOBAL)
     41 lib.network_width.argtypes = [c_void_p]
     42 lib.network_width.restype = c_int


    /usr/lib/python2.7/ctypes/__init__.pyc in __init__(self, name, mode, handle, use_errno,
    360
    361         if handle is None:
--> 362             self._handle = _dlopen(self._name, mode)
    363         else:
    364             self._handle = handle


    OSError: ../libdarknet.so: cannot open shared object file: No such file or directory
```

This next section of code was written by our team to run the darknet system

```
In [7]: """
        Test function to try out the wrapper for simple classification.
```

```
    Calls the internal predict_image and then rates each class, returning the most likely cl
    """
def classify(net, meta, im):
    out = predict_image(net, im)
    res = []
    for i in range(meta.classes):
        res.append((meta.names[i], out[i]))
    res = sorted(res, key=lambda x: -x[1])
    return res

"""
Wrapper for yolo-detection. Takes in a neural net, class info, and an image and creates
predictions for each region.
Returns a list of predictions, each containing a label, a confidence rating, and the cen
"""
def detect(net, meta, image, thresh=.5, hier_thresh=.5, nms=.45):
    im = load_image(image, 0, 0)
    boxes = make_boxes(net)
    probs = make_probs(net)
    num =   num_boxes(net)
    print num
    network_detect(net, im, thresh, hier_thresh, nms, boxes, probs)
    res = []
    for j in range(num):
        for i in range(meta.classes):
            if probs[j][i] > 0:
                res.append((meta.names[i], probs[j][i], (boxes[j].x, boxes[j].y, boxes[j
    res = sorted(res, key=lambda x: -x[1])
    free_image(im)
    free_ptrs(cast(probs, POINTER(c_void_p)), num)
    print res
    print
    return res


if __name__ == "__main__":
    os.chdir("../darknet")
    test_fpath = '/Users/brendan/Documents/Neural_2017/Documents/NC_data/test/'
    sub = open('submission.txt', 'w+')
    sub.write('image_cat,pixels\n')
    net = load_net("cfg/yolo.cfg", 'yolo.weights', 0) #use a pretrained neural net for n
    meta = load_meta("cfg/coco.data")
    for f in os.listdir(test_fpath):
        if f.count('.jpg'):
            bbox_dict = {}
            #get the detections
            detections = detect(net, meta, test_fpath + f)
            for d in detections:
```

```
                    #fix up the data for the data handler
                    dw = int(d[2][2]) // 2
                    dh = int(d[2][3]) // 2
                    tl = [int(d[2][0] - dw), int(d[2][1]) - dh]
                    br = [int(d[2][0]) + dw, int(d[2][1]) + dh]
                    if d[0] in bbox_dict.keys():
                        bbox_dict[d[0]].append([tl, br])
                    else:
                        bbox_dict[d[0]] = [[tl, br]]
                for label in data_handler.object_list:
                    if label not in bbox_dict.keys():
                        bbox_dict[label] = []
                line = data_handler.generate_output_file(bbox_dict, f)
                print line
                sub.write(line)


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

    OSError                                   Traceback (most recent call last)

    <ipython-input-7-e5afd3f15531> in <module>()
     28
     29 if __name__ == "__main__":
---> 30     os.chdir("../darknet")
     31     test_fpath = '/Users/brendan/Documents/Neural_2017/Documents/NC_data/test/'
     32     sub = open('submission.txt', 'w+')


    OSError: [Errno 2] No such file or directory: '../darknet'
```

4 Experiments

Because of the continuing struggle with implementation, there were few opportunities for rigorous experimentation during this project. Despite this, we were able to tweak with a few parameters throughout the project. When using SelectiveSearch, we were looking for parameter values which would produce a smaller number of boxes while still keeping the good ones with high probability. The algorithm took the parameters scale, sigma, and minsize, where scale determined the tendency for larger regions, sigma the bias towards grouping nearby objects of similar color together, and minsize the minimum length of the flattened pixel array. We ran some tests on random small groups of images from the test set, and found a scale of 400, a sigma of 0.8, and a minsize of 1000 to produce promising results, but a minsize of 400 largely kept the good large region proposals, while also producing many smaller regions to increase likelyhood of finding smaller objects.

We also had the opportunity to experiment with training times for the neural network used in the yolo algorithm, though never test the performance resulting from this training due to time

and memory constraints. As mentioned above, we chose to use the tiny-yolo network as a basis, which uses about 5 fewer convolutional layers than other yolo configurations such as standard yolo or yolo-voc. Running on CPU, tiny-yolo ran an iteration in about 5 minutes, while the extra convolutional layers took around 20 minutes. In all examples, a batch size of 64 was used.

5 Conclusions

One of our first, but most important findings, was the R-CNN is unusably slow for anything to akin to real time applications when run from a CPU. We had a number of difficulties testing and implement GPU implementations, mostly due to a lack of access to machines with GPUs (The machines in the lab were fairly uncooprative: Keras, TensorFlow, and YOLO_v2 could not be made to run due to memory issues, and none of our team had their own sutiable hardware).

YOLO_v2 overcomes this problem, and runs in an acceptable time on a CPU, however the output produced from the vanilla YOLO system is incompatible with the desired output format for the competition.

Our code was unable to sucessfully produce a result to the competition. despite producing seemingly close to correct values. The kaggle competition rejected the submission with the error: "Overlapping values detected: Submitted values must be unique". Because of this our best official performance was "NULL", despite attempts to forcibly rectify the output labels and remove collisions. Our belief is that this error was due to the fact that our modified version of YOLO allows overlapping bounding boxes, (i.e. the pottedplant is on the diningtable and therefore the pottedplant pixels will be a subset of the diningtable pixels) but the required format of the output data does not appear to allow this.

Overall, this project can be said to give a useful insight into existing methods for regional image recognition, however it made little progress towards optimizing a new system. This was in part due to the time constraints as implementing and retraining a new style of network can take over a day in some cases and there was a very limited amount of time to run experiments of this length once our team was getting close to functioning implementations.

6 Description of Collaboration

Brendan Case

- Wrote the python wrapper for YOLO.
- Set up training YOLO on data with Vasileios.
- Investigated RCNN and tested working with region proposal systems.

Guy Coop

- Produced data handler python class used by multiple systems to reformat the training data into the required input format for the neural network. And reformat the output from the network back into a format that matches the input.
- Lead the writing of the report, and collated it into a Jupyter Notebook.

Vasileios Mizaridis

- performed research into R-CNN using TensorFlow and Keras
- Assisted writing a python Wrapper for YOLO.
- Set up training and testing YOLO with our competition data.

Priyanka Mohata

- Tried to implement multiple versions different version of YOLO using pre existing neural networks. This proved to be very difficult since there were many issues during this due to many of the codes not run correctly on a windows machine. I did try it on the Linux machine is the UG lab but were unsuccessful due to limited disk storage quota.
- Ran darknet(YOLO) on a virtual linux machine.Initially ran it with its pretrained weights and tested it on a few images. After determining that YOLO was indeed a good technique, tried to modify it to fit it into our dataset and file structure.
- Contributed some sections of the report

Liangye Yu

- Found and collected information about the You Only Look Once (YOLO) System
- Analysed and tested a pre-trained network
- Analysed the results and attempted to retrain the network for our own data

7 References

[GDDM14] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. R-cnn for object detection. 2014.

[Gir15] Ross Girshick. Fast r-cnn. arXiv:1504.08083, 2015.

[Goo17] Google. Tensorflow 1.0, 2017.

[RDGF15] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. arXiv:1506.02640, 2015.

[RF16] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. arXiv preprint arXiv:1612.08242, 2016.

[RHGS15] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. arXiv:1506.01497, 2015.

[sidt17] scikit-image development team. skimage 0.13.1, 2017.