#include <stdlib.h>

#include "darknet.h"

#include "network.h"

#include "region\_layer.h"

#include "cost\_layer.h"

#include "utils.h"

#include "parser.h"

#include "box.h"

#include "demo.h"

#include "option\_list.h"

#ifndef \_\_COMPAR\_FN\_T

#define \_\_COMPAR\_FN\_T

typedef int (\*\_\_compar\_fn\_t)(const void\*, const void\*);

#ifdef \_\_USE\_GNU

typedef \_\_compar\_fn\_t comparison\_fn\_t;

#endif

#endif

#include "http\_stream.h"

int check\_mistakes = 0;

static int coco\_ids[] = { 1,2,3,4,5,6,7,8,9,10,11,13,14,15,16,17,18,19,20,21,22,23,24,25,27,28,31,32,33,34,35,36,37,38,39,40,41,42,43,44,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,67,70,72,73,74,75,76,77,78,79,80,81,82,84,85,86,87,88,89,90 };

void train\_detector(char \*datacfg, char \*cfgfile, char \*weightfile, int \*gpus, int ngpus, int clear, int dont\_show, int calc\_map, int mjpeg\_port, int show\_imgs, int benchmark\_layers, char\* chart\_path)

{

list \*options = read\_data\_cfg(datacfg);

char \*train\_images = option\_find\_str(options, "train", "data/train.txt");

char \*valid\_images = option\_find\_str(options, "valid", train\_images);

char \*backup\_directory = option\_find\_str(options, "backup", "/backup/");

network net\_map;

if (calc\_map) {

FILE\* valid\_file = fopen(valid\_images, "r");

if (!valid\_file) {

printf("\n Error: There is no %s file for mAP calculation!\n Don't use -map flag.\n Or set valid=%s in your %s file. \n", valid\_images, train\_images, datacfg);

getchar();

exit(-1);

}

else fclose(valid\_file);

cuda\_set\_device(gpus[0]);

printf(" Prepare additional network for mAP calculation...\n");

net\_map = parse\_network\_cfg\_custom(cfgfile, 1, 1);

net\_map.benchmark\_layers = benchmark\_layers;

const int net\_classes = net\_map.layers[net\_map.n - 1].classes;

int k; // free memory unnecessary arrays

for (k = 0; k < net\_map.n - 1; ++k) free\_layer\_custom(net\_map.layers[k], 1);

char \*name\_list = option\_find\_str(options, "names", "data/names.list");

int names\_size = 0;

char \*\*names = get\_labels\_custom(name\_list, &names\_size);

if (net\_classes != names\_size) {

printf("\n Error: in the file %s number of names %d that isn't equal to classes=%d in the file %s \n",

name\_list, names\_size, net\_classes, cfgfile);

if (net\_classes > names\_size) getchar();

}

free\_ptrs((void\*\*)names, net\_map.layers[net\_map.n - 1].classes);

}

srand(time(0));

char \*base = basecfg(cfgfile);

printf("%s\n", base);

float avg\_loss = -1;

network\* nets = (network\*)xcalloc(ngpus, sizeof(network));

srand(time(0));

int seed = rand();

int k;

for (k = 0; k < ngpus; ++k) {

srand(seed);

#ifdef GPU

cuda\_set\_device(gpus[k]);

#endif

nets[k] = parse\_network\_cfg(cfgfile);

nets[k].benchmark\_layers = benchmark\_layers;

if (weightfile) {

load\_weights(&nets[k], weightfile);

}

if (clear) {

\*nets[k].seen = 0;

\*nets[k].cur\_iteration = 0;

}

nets[k].learning\_rate \*= ngpus;

}

srand(time(0));

network net = nets[0];

const int actual\_batch\_size = net.batch \* net.subdivisions;

if (actual\_batch\_size == 1) {

printf("\n Error: You set incorrect value batch=1 for Training! You should set batch=64 subdivision=64 \n");

getchar();

}

else if (actual\_batch\_size < 8) {

printf("\n Warning: You set batch=%d lower than 64! It is recommended to set batch=64 subdivision=64 \n", actual\_batch\_size);

}

int imgs = net.batch \* net.subdivisions \* ngpus;

printf("Learning Rate: %g, Momentum: %g, Decay: %g\n", net.learning\_rate, net.momentum, net.decay);

data train, buffer;

layer l = net.layers[net.n - 1];

int classes = l.classes;

float jitter = l.jitter;

list \*plist = get\_paths(train\_images);

int train\_images\_num = plist->size;

char \*\*paths = (char \*\*)list\_to\_array(plist);

const int init\_w = net.w;

const int init\_h = net.h;

const int init\_b = net.batch;

int iter\_save, iter\_save\_last, iter\_map;

iter\_save = get\_current\_iteration(net);

iter\_save\_last = get\_current\_iteration(net);

iter\_map = get\_current\_iteration(net);

float mean\_average\_precision = -1;

float best\_map = mean\_average\_precision;

load\_args args = { 0 };

args.w = net.w;

args.h = net.h;

args.c = net.c;

args.paths = paths;

args.n = imgs;

args.m = plist->size;

args.classes = classes;

args.flip = net.flip;

args.jitter = jitter;

args.num\_boxes = l.max\_boxes;

net.num\_boxes = args.num\_boxes;

net.train\_images\_num = train\_images\_num;

args.d = &buffer;

args.type = DETECTION\_DATA;

args.threads = 64; // 16 or 64

args.angle = net.angle;

args.gaussian\_noise = net.gaussian\_noise;

args.blur = net.blur;

args.mixup = net.mixup;

args.exposure = net.exposure;

args.saturation = net.saturation;

args.hue = net.hue;

args.letter\_box = net.letter\_box;

if (dont\_show && show\_imgs) show\_imgs = 2;

args.show\_imgs = show\_imgs;

#ifdef OPENCV

args.threads = 6 \* ngpus; // 3 for - Amazon EC2 Tesla V100: p3.2xlarge (8 logical cores) - p3.16xlarge

//args.threads = 12 \* ngpus; // Ryzen 7 2700X (16 logical cores)

mat\_cv\* img = NULL;

float max\_img\_loss = 5;

int number\_of\_lines = 100;

int img\_size = 1000;

char windows\_name[100];

sprintf(windows\_name, "chart\_%s.png", base);

img = draw\_train\_chart(windows\_name, max\_img\_loss, net.max\_batches, number\_of\_lines, img\_size, dont\_show, chart\_path);

#endif //OPENCV

if (net.track) {

args.track = net.track;

args.augment\_speed = net.augment\_speed;

if (net.sequential\_subdivisions) args.threads = net.sequential\_subdivisions \* ngpus;

else args.threads = net.subdivisions \* ngpus;

args.mini\_batch = net.batch / net.time\_steps;

printf("\n Tracking! batch = %d, subdiv = %d, time\_steps = %d, mini\_batch = %d \n", net.batch, net.subdivisions, net.time\_steps, args.mini\_batch);

}

//printf(" imgs = %d \n", imgs);

pthread\_t load\_thread = load\_data(args);

int count = 0;

double time\_remaining, avg\_time = -1, alpha\_time = 0.01;

//while(i\*imgs < N\*120){

while (get\_current\_iteration(net) < net.max\_batches) {

if (l.random && count++ % 10 == 0) {

float rand\_coef = 1.4;

if (l.random != 1.0) rand\_coef = l.random;

printf("Resizing, random\_coef = %.2f \n", rand\_coef);

float random\_val = rand\_scale(rand\_coef); // \*x or /x

int dim\_w = roundl(random\_val\*init\_w / net.resize\_step + 1) \* net.resize\_step;

int dim\_h = roundl(random\_val\*init\_h / net.resize\_step + 1) \* net.resize\_step;

if (random\_val < 1 && (dim\_w > init\_w || dim\_h > init\_h)) dim\_w = init\_w, dim\_h = init\_h;

int max\_dim\_w = roundl(rand\_coef\*init\_w / net.resize\_step + 1) \* net.resize\_step;

int max\_dim\_h = roundl(rand\_coef\*init\_h / net.resize\_step + 1) \* net.resize\_step;

// at the beginning (check if enough memory) and at the end (calc rolling mean/variance)

if (avg\_loss < 0 || get\_current\_iteration(net) > net.max\_batches - 100) {

dim\_w = max\_dim\_w;

dim\_h = max\_dim\_h;

}

if (dim\_w < net.resize\_step) dim\_w = net.resize\_step;

if (dim\_h < net.resize\_step) dim\_h = net.resize\_step;

int dim\_b = (init\_b \* max\_dim\_w \* max\_dim\_h) / (dim\_w \* dim\_h);

int new\_dim\_b = (int)(dim\_b \* 0.8);

if (new\_dim\_b > init\_b) dim\_b = new\_dim\_b;

args.w = dim\_w;

args.h = dim\_h;

int k;

if (net.dynamic\_minibatch) {

for (k = 0; k < ngpus; ++k) {

(\*nets[k].seen) = init\_b \* net.subdivisions \* get\_current\_iteration(net); // remove this line, when you will save to weights-file both: seen & cur\_iteration

nets[k].batch = dim\_b;

int j;

for (j = 0; j < nets[k].n; ++j)

nets[k].layers[j].batch = dim\_b;

}

net.batch = dim\_b;

imgs = net.batch \* net.subdivisions \* ngpus;

args.n = imgs;

printf("\n %d x %d (batch = %d) \n", dim\_w, dim\_h, net.batch);

}

else

printf("\n %d x %d \n", dim\_w, dim\_h);

pthread\_join(load\_thread, 0);

train = buffer;

free\_data(train);

load\_thread = load\_data(args);

for (k = 0; k < ngpus; ++k) {

resize\_network(nets + k, dim\_w, dim\_h);

}

net = nets[0];

}

double time = what\_time\_is\_it\_now();

pthread\_join(load\_thread, 0);

train = buffer;

if (net.track) {

net.sequential\_subdivisions = get\_current\_seq\_subdivisions(net);

args.threads = net.sequential\_subdivisions \* ngpus;

printf(" sequential\_subdivisions = %d, sequence = %d \n", net.sequential\_subdivisions, get\_sequence\_value(net));

}

load\_thread = load\_data(args);

/\*

int k;

for(k = 0; k < l.max\_boxes; ++k){

box b = float\_to\_box(train.y.vals[10] + 1 + k\*5);

if(!b.x) break;

printf("loaded: %f %f %f %f\n", b.x, b.y, b.w, b.h);

}

image im = float\_to\_image(448, 448, 3, train.X.vals[10]);

int k;

for(k = 0; k < l.max\_boxes; ++k){

box b = float\_to\_box(train.y.vals[10] + 1 + k\*5);

printf("%d %d %d %d\n", truth.x, truth.y, truth.w, truth.h);

draw\_bbox(im, b, 8, 1,0,0);

}

save\_image(im, "truth11");

\*/

const double load\_time = (what\_time\_is\_it\_now() - time);

printf("Loaded: %lf seconds", load\_time);

if (load\_time > 0.1 && avg\_loss > 0) printf(" - performance bottleneck on CPU or Disk HDD/SSD");

printf("\n");

time = what\_time\_is\_it\_now();

float loss = 0;

#ifdef GPU

if (ngpus == 1) {

int wait\_key = (dont\_show) ? 0 : 1;

loss = train\_network\_waitkey(net, train, wait\_key);

}

else {

loss = train\_networks(nets, ngpus, train, 4);

}

#else

loss = train\_network(net, train);

#endif

if (avg\_loss < 0 || avg\_loss != avg\_loss) avg\_loss = loss; // if(-inf or nan)

avg\_loss = avg\_loss\*.9 + loss\*.1;

const int iteration = get\_current\_iteration(net);

//i = get\_current\_batch(net);

int calc\_map\_for\_each = 4 \* train\_images\_num / (net.batch \* net.subdivisions); // calculate mAP for each 4 Epochs

calc\_map\_for\_each = fmax(calc\_map\_for\_each, 100);

int next\_map\_calc = iter\_map + calc\_map\_for\_each;

next\_map\_calc = fmax(next\_map\_calc, net.burn\_in);

//next\_map\_calc = fmax(next\_map\_calc, 400);

if (calc\_map) {

printf("\n (next mAP calculation at %d iterations) ", next\_map\_calc);

if (mean\_average\_precision > 0) printf("\n Last accuracy mAP@0.5 = %2.2f %%, best = %2.2f %% ", mean\_average\_precision \* 100, best\_map \* 100);

}

if (net.cudnn\_half) {

if (iteration < net.burn\_in \* 3) fprintf(stderr, "\n Tensor Cores are disabled until the first %d iterations are reached.", 3 \* net.burn\_in);

else fprintf(stderr, "\n Tensor Cores are used.");

}

printf("\n %d: %f, %f avg loss, %f rate, %lf seconds, %d images, %f hours left\n", iteration, loss, avg\_loss, get\_current\_rate(net), (what\_time\_is\_it\_now() - time), iteration\*imgs, avg\_time);

int draw\_precision = 0;

if (calc\_map && (iteration >= next\_map\_calc || iteration == net.max\_batches)) {

if (l.random) {

printf("Resizing to initial size: %d x %d ", init\_w, init\_h);

args.w = init\_w;

args.h = init\_h;

int k;

if (net.dynamic\_minibatch) {

for (k = 0; k < ngpus; ++k) {

for (k = 0; k < ngpus; ++k) {

nets[k].batch = init\_b;

int j;

for (j = 0; j < nets[k].n; ++j)

nets[k].layers[j].batch = init\_b;

}

}

net.batch = init\_b;

imgs = init\_b \* net.subdivisions \* ngpus;

args.n = imgs;

printf("\n %d x %d (batch = %d) \n", init\_w, init\_h, init\_b);

}

pthread\_join(load\_thread, 0);

free\_data(train);

train = buffer;

load\_thread = load\_data(args);

for (k = 0; k < ngpus; ++k) {

resize\_network(nets + k, init\_w, init\_h);

}

net = nets[0];

}

copy\_weights\_net(net, &net\_map);

// combine Training and Validation networks

//network net\_combined = combine\_train\_valid\_networks(net, net\_map);

iter\_map = iteration;

mean\_average\_precision = validate\_detector\_map(datacfg, cfgfile, weightfile, 0.25, 0.5, 0, net.letter\_box, &net\_map);// &net\_combined);

printf("\n mean\_average\_precision (mAP@0.5) = %f \n", mean\_average\_precision);

if (mean\_average\_precision > best\_map) {

best\_map = mean\_average\_precision;

printf("New best mAP!\n");

char buff[256];

sprintf(buff, "%s/%s\_best.weights", backup\_directory, base);

save\_weights(net, buff);

}

draw\_precision = 1;

}

time\_remaining = (net.max\_batches - iteration)\*(what\_time\_is\_it\_now() - time + load\_time) / 60 / 60;

// set initial value, even if resume training from 10000 iteration

if (avg\_time < 0) avg\_time = time\_remaining;

else avg\_time = alpha\_time \* time\_remaining + (1 - alpha\_time) \* avg\_time;

#ifdef OPENCV

draw\_train\_loss(windows\_name, img, img\_size, avg\_loss, max\_img\_loss, iteration, net.max\_batches, mean\_average\_precision, draw\_precision, "mAP%", dont\_show, mjpeg\_port, avg\_time);

#endif // OPENCV

//if (i % 1000 == 0 || (i < 1000 && i % 100 == 0)) {

//if (i % 100 == 0) {

if (iteration >= (iter\_save + 1000) || iteration % 1000 == 0) {

iter\_save = iteration;

#ifdef GPU

if (ngpus != 1) sync\_nets(nets, ngpus, 0);

#endif

char buff[256];

sprintf(buff, "%s/%s\_%d.weights", backup\_directory, base, iteration);

save\_weights(net, buff);

}

if (iteration >= (iter\_save\_last + 100) || (iteration % 100 == 0 && iteration > 1)) {

iter\_save\_last = iteration;

#ifdef GPU

if (ngpus != 1) sync\_nets(nets, ngpus, 0);

#endif

char buff[256];

sprintf(buff, "%s/%s\_last.weights", backup\_directory, base);

save\_weights(net, buff);

}

free\_data(train);

}

#ifdef GPU

if (ngpus != 1) sync\_nets(nets, ngpus, 0);

#endif

char buff[256];

sprintf(buff, "%s/%s\_final.weights", backup\_directory, base);

save\_weights(net, buff);

#ifdef OPENCV

release\_mat(&img);

destroy\_all\_windows\_cv();

#endif

// free memory

pthread\_join(load\_thread, 0);

free\_data(buffer);

free\_load\_threads(&args);

free(base);

free(paths);

free\_list\_contents(plist);

free\_list(plist);

free\_list\_contents\_kvp(options);

free\_list(options);

for (k = 0; k < ngpus; ++k) free\_network(nets[k]);

free(nets);

//free\_network(net);

if (calc\_map) {

net\_map.n = 0;

free\_network(net\_map);

}

}

static int get\_coco\_image\_id(char \*filename)

{

char \*p = strrchr(filename, '/');

char \*c = strrchr(filename, '\_');

if (c) p = c;

return atoi(p + 1);

}

static void print\_cocos(FILE \*fp, char \*image\_path, detection \*dets, int num\_boxes, int classes, int w, int h)

{

int i, j;

//int image\_id = get\_coco\_image\_id(image\_path);

char \*p = basecfg(image\_path);

int image\_id = atoi(p);

for (i = 0; i < num\_boxes; ++i) {

float xmin = dets[i].bbox.x - dets[i].bbox.w / 2.;

float xmax = dets[i].bbox.x + dets[i].bbox.w / 2.;

float ymin = dets[i].bbox.y - dets[i].bbox.h / 2.;

float ymax = dets[i].bbox.y + dets[i].bbox.h / 2.;

if (xmin < 0) xmin = 0;

if (ymin < 0) ymin = 0;

if (xmax > w) xmax = w;

if (ymax > h) ymax = h;

float bx = xmin;

float by = ymin;

float bw = xmax - xmin;

float bh = ymax - ymin;

for (j = 0; j < classes; ++j) {

if (dets[i].prob[j] > 0) {

char buff[1024];

sprintf(buff, "{\"image\_id\":%d, \"category\_id\":%d, \"bbox\":[%f, %f, %f, %f], \"score\":%f},\n", image\_id, coco\_ids[j], bx, by, bw, bh, dets[i].prob[j]);

fprintf(fp, buff);

//printf("%s", buff);

}

}

}

}

void print\_detector\_detections(FILE \*\*fps, char \*id, detection \*dets, int total, int classes, int w, int h)

{

int i, j;

for (i = 0; i < total; ++i) {

float xmin = dets[i].bbox.x - dets[i].bbox.w / 2. + 1;

float xmax = dets[i].bbox.x + dets[i].bbox.w / 2. + 1;

float ymin = dets[i].bbox.y - dets[i].bbox.h / 2. + 1;

float ymax = dets[i].bbox.y + dets[i].bbox.h / 2. + 1;

if (xmin < 1) xmin = 1;

if (ymin < 1) ymin = 1;

if (xmax > w) xmax = w;

if (ymax > h) ymax = h;

for (j = 0; j < classes; ++j) {

if (dets[i].prob[j]) fprintf(fps[j], "%s %f %f %f %f %f\n", id, dets[i].prob[j],

xmin, ymin, xmax, ymax);

}

}

}

void print\_imagenet\_detections(FILE \*fp, int id, detection \*dets, int total, int classes, int w, int h)

{

int i, j;

for (i = 0; i < total; ++i) {

float xmin = dets[i].bbox.x - dets[i].bbox.w / 2.;

float xmax = dets[i].bbox.x + dets[i].bbox.w / 2.;

float ymin = dets[i].bbox.y - dets[i].bbox.h / 2.;

float ymax = dets[i].bbox.y + dets[i].bbox.h / 2.;

if (xmin < 0) xmin = 0;

if (ymin < 0) ymin = 0;

if (xmax > w) xmax = w;

if (ymax > h) ymax = h;

for (j = 0; j < classes; ++j) {

int myclass = j;

if (dets[i].prob[myclass] > 0) fprintf(fp, "%d %d %f %f %f %f %f\n", id, j + 1, dets[i].prob[myclass],

xmin, ymin, xmax, ymax);

}

}

}

static void print\_kitti\_detections(FILE \*\*fps, char \*id, detection \*dets, int total, int classes, int w, int h, char \*outfile, char \*prefix)

{

char \*kitti\_ids[] = { "car", "pedestrian", "cyclist" };

FILE \*fpd = 0;

char buffd[1024];

snprintf(buffd, 1024, "%s/%s/data/%s.txt", prefix, outfile, id);

fpd = fopen(buffd, "w");

int i, j;

for (i = 0; i < total; ++i)

{

float xmin = dets[i].bbox.x - dets[i].bbox.w / 2.;

float xmax = dets[i].bbox.x + dets[i].bbox.w / 2.;

float ymin = dets[i].bbox.y - dets[i].bbox.h / 2.;

float ymax = dets[i].bbox.y + dets[i].bbox.h / 2.;

if (xmin < 0) xmin = 0;

if (ymin < 0) ymin = 0;

if (xmax > w) xmax = w;

if (ymax > h) ymax = h;

for (j = 0; j < classes; ++j)

{

//if (dets[i].prob[j]) fprintf(fpd, "%s 0 0 0 %f %f %f %f -1 -1 -1 -1 0 0 0 %f\n", kitti\_ids[j], xmin, ymin, xmax, ymax, dets[i].prob[j]);

if (dets[i].prob[j]) fprintf(fpd, "%s -1 -1 -10 %f %f %f %f -1 -1 -1 -1000 -1000 -1000 -10 %f\n", kitti\_ids[j], xmin, ymin, xmax, ymax, dets[i].prob[j]);

}

}

fclose(fpd);

}

static void eliminate\_bdd(char \*buf, char \*a)

{

int n = 0;

int i, k;

for (i = 0; buf[i] != '\0'; i++)

{

if (buf[i] == a[n])

{

k = i;

while (buf[i] == a[n])

{

if (a[++n] == '\0')

{

for (k; buf[k + n] != '\0'; k++)

{

buf[k] = buf[k + n];

}

buf[k] = '\0';

break;

}

i++;

}

n = 0; i--;

}

}

}

static void get\_bdd\_image\_id(char \*filename)

{

char \*p = strrchr(filename, '/');

eliminate\_bdd(p, ".jpg");

eliminate\_bdd(p, "/");

strcpy(filename, p);

}

static void print\_bdd\_detections(FILE \*fp, char \*image\_path, detection \*dets, int num\_boxes, int classes, int w, int h)

{

char \*bdd\_ids[] = { "bike" , "bus" , "car" , "motor" ,"person", "rider", "traffic light", "traffic sign", "train", "truck" };

get\_bdd\_image\_id(image\_path);

int i, j;

for (i = 0; i < num\_boxes; ++i)

{

float xmin = dets[i].bbox.x - dets[i].bbox.w / 2.;

float xmax = dets[i].bbox.x + dets[i].bbox.w / 2.;

float ymin = dets[i].bbox.y - dets[i].bbox.h / 2.;

float ymax = dets[i].bbox.y + dets[i].bbox.h / 2.;

if (xmin < 0) xmin = 0;

if (ymin < 0) ymin = 0;

if (xmax > w) xmax = w;

if (ymax > h) ymax = h;

float bx1 = xmin;

float by1 = ymin;

float bx2 = xmax;

float by2 = ymax;

for (j = 0; j < classes; ++j)

{

if (dets[i].prob[j])

{

fprintf(fp, "\t{\n\t\t\"name\":\"%s\",\n\t\t\"category\":\"%s\",\n\t\t\"bbox\":[%f, %f, %f, %f],\n\t\t\"score\":%f\n\t},\n", image\_path, bdd\_ids[j], bx1, by1, bx2, by2, dets[i].prob[j]);

}

}

}

}

void validate\_detector(char \*datacfg, char \*cfgfile, char \*weightfile, char \*outfile)

{

int j;

list \*options = read\_data\_cfg(datacfg);

char \*valid\_images = option\_find\_str(options, "valid", "data/train.list");

char \*name\_list = option\_find\_str(options, "names", "data/names.list");

char \*prefix = option\_find\_str(options, "results", "results");

char \*\*names = get\_labels(name\_list);

char \*mapf = option\_find\_str(options, "map", 0);

int \*map = 0;

if (mapf) map = read\_map(mapf);

network net = parse\_network\_cfg\_custom(cfgfile, 1, 1); // set batch=1

if (weightfile) {

load\_weights(&net, weightfile);

}

//set\_batch\_network(&net, 1);

fuse\_conv\_batchnorm(net);

calculate\_binary\_weights(net);

fprintf(stderr, "Learning Rate: %g, Momentum: %g, Decay: %g\n", net.learning\_rate, net.momentum, net.decay);

srand(time(0));

list \*plist = get\_paths(valid\_images);

char \*\*paths = (char \*\*)list\_to\_array(plist);

layer l = net.layers[net.n - 1];

int classes = l.classes;

char buff[1024];

char \*type = option\_find\_str(options, "eval", "voc");

FILE \*fp = 0;

FILE \*\*fps = 0;

int coco = 0;

int imagenet = 0;

int bdd = 0;

int kitti = 0;

if (0 == strcmp(type, "coco")) {

if (!outfile) outfile = "coco\_results";

snprintf(buff, 1024, "%s/%s.json", prefix, outfile);

fp = fopen(buff, "w");

fprintf(fp, "[\n");

coco = 1;

}

else if (0 == strcmp(type, "bdd")) {

if (!outfile) outfile = "bdd\_results";

snprintf(buff, 1024, "%s/%s.json", prefix, outfile);

fp = fopen(buff, "w");

fprintf(fp, "[\n");

bdd = 1;

}

else if (0 == strcmp(type, "kitti")) {

char buff2[1024];

if (!outfile) outfile = "kitti\_results";

printf("%s\n", outfile);

snprintf(buff, 1024, "%s/%s", prefix, outfile);

int mkd = make\_directory(buff, 0777);

snprintf(buff2, 1024, "%s/%s/data", prefix, outfile);

int mkd2 = make\_directory(buff2, 0777);

kitti = 1;

}

else if (0 == strcmp(type, "imagenet")) {

if (!outfile) outfile = "imagenet-detection";

snprintf(buff, 1024, "%s/%s.txt", prefix, outfile);

fp = fopen(buff, "w");

imagenet = 1;

classes = 200;

}

else {

if (!outfile) outfile = "comp4\_det\_test\_";

fps = (FILE\*\*) xcalloc(classes, sizeof(FILE \*));

for (j = 0; j < classes; ++j) {

snprintf(buff, 1024, "%s/%s%s.txt", prefix, outfile, names[j]);

fps[j] = fopen(buff, "w");

}

}

int m = plist->size;

int i = 0;

int t;

float thresh = .001;

float nms = .45;

int nthreads = 4;

if (m < 4) nthreads = m;

image\* val = (image\*)xcalloc(nthreads, sizeof(image));

image\* val\_resized = (image\*)xcalloc(nthreads, sizeof(image));

image\* buf = (image\*)xcalloc(nthreads, sizeof(image));

image\* buf\_resized = (image\*)xcalloc(nthreads, sizeof(image));

pthread\_t\* thr = (pthread\_t\*)xcalloc(nthreads, sizeof(pthread\_t));

load\_args args = { 0 };

args.w = net.w;

args.h = net.h;

args.c = net.c;

args.type = IMAGE\_DATA;

//args.type = LETTERBOX\_DATA;

for (t = 0; t < nthreads; ++t) {

args.path = paths[i + t];

args.im = &buf[t];

args.resized = &buf\_resized[t];

thr[t] = load\_data\_in\_thread(args);

}

time\_t start = time(0);

for (i = nthreads; i < m + nthreads; i += nthreads) {

fprintf(stderr, "%d\n", i);

for (t = 0; t < nthreads && i + t - nthreads < m; ++t) {

pthread\_join(thr[t], 0);

val[t] = buf[t];

val\_resized[t] = buf\_resized[t];

}

for (t = 0; t < nthreads && i + t < m; ++t) {

args.path = paths[i + t];

args.im = &buf[t];

args.resized = &buf\_resized[t];

thr[t] = load\_data\_in\_thread(args);

}

for (t = 0; t < nthreads && i + t - nthreads < m; ++t) {

char \*path = paths[i + t - nthreads];

char \*id = basecfg(path);

float \*X = val\_resized[t].data;

network\_predict(net, X);

int w = val[t].w;

int h = val[t].h;

int nboxes = 0;

int letterbox = (args.type == LETTERBOX\_DATA);

detection \*dets = get\_network\_boxes(&net, w, h, thresh, .5, map, 0, &nboxes, letterbox);

if (nms) {

if (l.nms\_kind == DEFAULT\_NMS) do\_nms\_sort(dets, nboxes, l.classes, nms);

else diounms\_sort(dets, nboxes, l.classes, nms, l.nms\_kind, l.beta\_nms);

}

if (coco) {

print\_cocos(fp, path, dets, nboxes, classes, w, h);

}

else if (imagenet) {

print\_imagenet\_detections(fp, i + t - nthreads + 1, dets, nboxes, classes, w, h);

}

else if (bdd) {

print\_bdd\_detections(fp, path, dets, nboxes, classes, w, h);

}

else if (kitti) {

print\_kitti\_detections(fps, id, dets, nboxes, classes, w, h, outfile, prefix);

}

else {

print\_detector\_detections(fps, id, dets, nboxes, classes, w, h);

}

free\_detections(dets, nboxes);

free(id);

free\_image(val[t]);

free\_image(val\_resized[t]);

}

}

if (fps) {

for (j = 0; j < classes; ++j) {

fclose(fps[j]);

}

free(fps);

}

if (coco) {

#ifdef WIN32

fseek(fp, -3, SEEK\_CUR);

#else

fseek(fp, -2, SEEK\_CUR);

#endif

fprintf(fp, "\n]\n");

}

if (bdd) {

#ifdef WIN32

fseek(fp, -3, SEEK\_CUR);

#else

fseek(fp, -2, SEEK\_CUR);

#endif

fprintf(fp, "\n]\n");

fclose(fp);

}

if (fp) fclose(fp);

if (val) free(val);

if (val\_resized) free(val\_resized);

if (thr) free(thr);

if (buf) free(buf);

if (buf\_resized) free(buf\_resized);

fprintf(stderr, "Total Detection Time: %f Seconds\n", (double)time(0) - start);

}

void validate\_detector\_recall(char \*datacfg, char \*cfgfile, char \*weightfile)

{

network net = parse\_network\_cfg\_custom(cfgfile, 1, 1); // set batch=1

if (weightfile) {

load\_weights(&net, weightfile);

}

//set\_batch\_network(&net, 1);

fuse\_conv\_batchnorm(net);

srand(time(0));

//list \*plist = get\_paths("data/coco\_val\_5k.list");

list \*options = read\_data\_cfg(datacfg);

char \*valid\_images = option\_find\_str(options, "valid", "data/train.txt");

list \*plist = get\_paths(valid\_images);

char \*\*paths = (char \*\*)list\_to\_array(plist);

//layer l = net.layers[net.n - 1];

int j, k;

int m = plist->size;

int i = 0;

float thresh = .001;

float iou\_thresh = .5;

float nms = .4;

int total = 0;

int correct = 0;

int proposals = 0;

float avg\_iou = 0;

for (i = 0; i < m; ++i) {

char \*path = paths[i];

image orig = load\_image(path, 0, 0, net.c);

image sized = resize\_image(orig, net.w, net.h);

char \*id = basecfg(path);

network\_predict(net, sized.data);

int nboxes = 0;

int letterbox = 0;

detection \*dets = get\_network\_boxes(&net, sized.w, sized.h, thresh, .5, 0, 1, &nboxes, letterbox);

if (nms) do\_nms\_obj(dets, nboxes, 1, nms);

char labelpath[4096];

replace\_image\_to\_label(path, labelpath);

int num\_labels = 0;

box\_label \*truth = read\_boxes(labelpath, &num\_labels);

for (k = 0; k < nboxes; ++k) {

if (dets[k].objectness > thresh) {

++proposals;

}

}

for (j = 0; j < num\_labels; ++j) {

++total;

box t = { truth[j].x, truth[j].y, truth[j].w, truth[j].h };

float best\_iou = 0;

for (k = 0; k < nboxes; ++k) {

float iou = box\_iou(dets[k].bbox, t);

if (dets[k].objectness > thresh && iou > best\_iou) {

best\_iou = iou;

}

}

avg\_iou += best\_iou;

if (best\_iou > iou\_thresh) {

++correct;

}

}

//fprintf(stderr, " %s - %s - ", paths[i], labelpath);

fprintf(stderr, "%5d %5d %5d\tRPs/Img: %.2f\tIOU: %.2f%%\tRecall:%.2f%%\n", i, correct, total, (float)proposals / (i + 1), avg\_iou \* 100 / total, 100.\*correct / total);

free(id);

free\_image(orig);

free\_image(sized);

}

}

typedef struct {

box b;

float p;

int class\_id;

int image\_index;

int truth\_flag;

int unique\_truth\_index;

} box\_prob;

int detections\_comparator(const void \*pa, const void \*pb)

{

box\_prob a = \*(const box\_prob \*)pa;

box\_prob b = \*(const box\_prob \*)pb;

float diff = a.p - b.p;

if (diff < 0) return 1;

else if (diff > 0) return -1;

return 0;

}

float validate\_detector\_map(char \*datacfg, char \*cfgfile, char \*weightfile, float thresh\_calc\_avg\_iou, const float iou\_thresh, const int map\_points, int letter\_box, network \*existing\_net)

{

int j;

list \*options = read\_data\_cfg(datacfg);

char \*valid\_images = option\_find\_str(options, "valid", "data/train.txt");

char \*difficult\_valid\_images = option\_find\_str(options, "difficult", NULL);

char \*name\_list = option\_find\_str(options, "names", "data/names.list");

int names\_size = 0;

char \*\*names = get\_labels\_custom(name\_list, &names\_size); //get\_labels(name\_list);

//char \*mapf = option\_find\_str(options, "map", 0);

//int \*map = 0;

//if (mapf) map = read\_map(mapf);

FILE\* reinforcement\_fd = NULL;

network net;

//int initial\_batch;

if (existing\_net) {

char \*train\_images = option\_find\_str(options, "train", "data/train.txt");

valid\_images = option\_find\_str(options, "valid", train\_images);

net = \*existing\_net;

remember\_network\_recurrent\_state(\*existing\_net);

free\_network\_recurrent\_state(\*existing\_net);

}

else {

net = parse\_network\_cfg\_custom(cfgfile, 1, 1); // set batch=1

if (weightfile) {

load\_weights(&net, weightfile);

}

//set\_batch\_network(&net, 1);

fuse\_conv\_batchnorm(net);

calculate\_binary\_weights(net);

}

if (net.layers[net.n - 1].classes != names\_size) {

printf("\n Error: in the file %s number of names %d that isn't equal to classes=%d in the file %s \n",

name\_list, names\_size, net.layers[net.n - 1].classes, cfgfile);

getchar();

}

srand(time(0));

printf("\n calculation mAP (mean average precision)...\n");

list \*plist = get\_paths(valid\_images);

char \*\*paths = (char \*\*)list\_to\_array(plist);

char \*\*paths\_dif = NULL;

if (difficult\_valid\_images) {

list \*plist\_dif = get\_paths(difficult\_valid\_images);

paths\_dif = (char \*\*)list\_to\_array(plist\_dif);

}

layer l = net.layers[net.n - 1];

int classes = l.classes;

int m = plist->size;

int i = 0;

int t;

const float thresh = .005;

const float nms = .45;

//const float iou\_thresh = 0.5;

int nthreads = 4;

if (m < 4) nthreads = m;

image\* val = (image\*)xcalloc(nthreads, sizeof(image));

image\* val\_resized = (image\*)xcalloc(nthreads, sizeof(image));

image\* buf = (image\*)xcalloc(nthreads, sizeof(image));

image\* buf\_resized = (image\*)xcalloc(nthreads, sizeof(image));

pthread\_t\* thr = (pthread\_t\*)xcalloc(nthreads, sizeof(pthread\_t));

load\_args args = { 0 };

args.w = net.w;

args.h = net.h;

args.c = net.c;

if (letter\_box) args.type = LETTERBOX\_DATA;

else args.type = IMAGE\_DATA;

//const float thresh\_calc\_avg\_iou = 0.24;

float avg\_iou = 0;

int tp\_for\_thresh = 0;

int fp\_for\_thresh = 0;

box\_prob\* detections = (box\_prob\*)xcalloc(1, sizeof(box\_prob));

int detections\_count = 0;

int unique\_truth\_count = 0;

int\* truth\_classes\_count = (int\*)xcalloc(classes, sizeof(int));

// For multi-class precision and recall computation

float \*avg\_iou\_per\_class = (float\*)xcalloc(classes, sizeof(float));

int \*tp\_for\_thresh\_per\_class = (int\*)xcalloc(classes, sizeof(int));

int \*fp\_for\_thresh\_per\_class = (int\*)xcalloc(classes, sizeof(int));

for (t = 0; t < nthreads; ++t) {

args.path = paths[i + t];

args.im = &buf[t];

args.resized = &buf\_resized[t];

thr[t] = load\_data\_in\_thread(args);

}

time\_t start = time(0);

for (i = nthreads; i < m + nthreads; i += nthreads) {

fprintf(stderr, "\r%d", i);

for (t = 0; t < nthreads && (i + t - nthreads) < m; ++t) {

pthread\_join(thr[t], 0);

val[t] = buf[t];

val\_resized[t] = buf\_resized[t];

}

for (t = 0; t < nthreads && (i + t) < m; ++t) {

args.path = paths[i + t];

args.im = &buf[t];

args.resized = &buf\_resized[t];

thr[t] = load\_data\_in\_thread(args);

}

for (t = 0; t < nthreads && i + t - nthreads < m; ++t) {

const int image\_index = i + t - nthreads;

char \*path = paths[image\_index];

char \*id = basecfg(path);

float \*X = val\_resized[t].data;

network\_predict(net, X);

int nboxes = 0;

float hier\_thresh = 0;

detection \*dets;

if (args.type == LETTERBOX\_DATA) {

dets = get\_network\_boxes(&net, val[t].w, val[t].h, thresh, hier\_thresh, 0, 1, &nboxes, letter\_box);

}

else {

dets = get\_network\_boxes(&net, 1, 1, thresh, hier\_thresh, 0, 0, &nboxes, letter\_box);

}

//detection \*dets = get\_network\_boxes(&net, val[t].w, val[t].h, thresh, hier\_thresh, 0, 1, &nboxes, letter\_box); // for letter\_box=1

if (nms) {

if (l.nms\_kind == DEFAULT\_NMS) do\_nms\_sort(dets, nboxes, l.classes, nms);

else diounms\_sort(dets, nboxes, l.classes, nms, l.nms\_kind, l.beta\_nms);

}

//if (nms) do\_nms\_obj(dets, nboxes, l.classes, nms);

char labelpath[4096];

replace\_image\_to\_label(path, labelpath);

int num\_labels = 0;

box\_label \*truth = read\_boxes(labelpath, &num\_labels);

int j;

for (j = 0; j < num\_labels; ++j) {

truth\_classes\_count[truth[j].id]++;

}

// difficult

box\_label \*truth\_dif = NULL;

int num\_labels\_dif = 0;

if (paths\_dif)

{

char \*path\_dif = paths\_dif[image\_index];

char labelpath\_dif[4096];

replace\_image\_to\_label(path\_dif, labelpath\_dif);

truth\_dif = read\_boxes(labelpath\_dif, &num\_labels\_dif);

}

const int checkpoint\_detections\_count = detections\_count;

int i;

for (i = 0; i < nboxes; ++i) {

int class\_id;

for (class\_id = 0; class\_id < classes; ++class\_id) {

float prob = dets[i].prob[class\_id];

if (prob > 0) {

detections\_count++;

detections = (box\_prob\*)xrealloc(detections, detections\_count \* sizeof(box\_prob));

detections[detections\_count - 1].b = dets[i].bbox;

detections[detections\_count - 1].p = prob;

detections[detections\_count - 1].image\_index = image\_index;

detections[detections\_count - 1].class\_id = class\_id;

detections[detections\_count - 1].truth\_flag = 0;

detections[detections\_count - 1].unique\_truth\_index = -1;

int truth\_index = -1;

float max\_iou = 0;

for (j = 0; j < num\_labels; ++j)

{

box t = { truth[j].x, truth[j].y, truth[j].w, truth[j].h };

//printf(" IoU = %f, prob = %f, class\_id = %d, truth[j].id = %d \n",

// box\_iou(dets[i].bbox, t), prob, class\_id, truth[j].id);

float current\_iou = box\_iou(dets[i].bbox, t);

if (current\_iou > iou\_thresh && class\_id == truth[j].id) {

if (current\_iou > max\_iou) {

max\_iou = current\_iou;

truth\_index = unique\_truth\_count + j;

}

}

}

// best IoU

if (truth\_index > -1) {

detections[detections\_count - 1].truth\_flag = 1;

detections[detections\_count - 1].unique\_truth\_index = truth\_index;

}

else {

// if object is difficult then remove detection

for (j = 0; j < num\_labels\_dif; ++j) {

box t = { truth\_dif[j].x, truth\_dif[j].y, truth\_dif[j].w, truth\_dif[j].h };

float current\_iou = box\_iou(dets[i].bbox, t);

if (current\_iou > iou\_thresh && class\_id == truth\_dif[j].id) {

--detections\_count;

break;

}

}

}

// calc avg IoU, true-positives, false-positives for required Threshold

if (prob > thresh\_calc\_avg\_iou) {

int z, found = 0;

for (z = checkpoint\_detections\_count; z < detections\_count - 1; ++z) {

if (detections[z].unique\_truth\_index == truth\_index) {

found = 1; break;

}

}

if (truth\_index > -1 && found == 0) {

avg\_iou += max\_iou;

++tp\_for\_thresh;

avg\_iou\_per\_class[class\_id] += max\_iou;

tp\_for\_thresh\_per\_class[class\_id]++;

}

else{

fp\_for\_thresh++;

fp\_for\_thresh\_per\_class[class\_id]++;

}

}

}

}

}

unique\_truth\_count += num\_labels;

//static int previous\_errors = 0;

//int total\_errors = fp\_for\_thresh + (unique\_truth\_count - tp\_for\_thresh);

//int errors\_in\_this\_image = total\_errors - previous\_errors;

//previous\_errors = total\_errors;

//if(reinforcement\_fd == NULL) reinforcement\_fd = fopen("reinforcement.txt", "wb");

//char buff[1000];

//sprintf(buff, "%s\n", path);

//if(errors\_in\_this\_image > 0) fwrite(buff, sizeof(char), strlen(buff), reinforcement\_fd);

free\_detections(dets, nboxes);

free(id);

free\_image(val[t]);

free\_image(val\_resized[t]);

}

}

//for (t = 0; t < nthreads; ++t) {

// pthread\_join(thr[t], 0);

//}

if ((tp\_for\_thresh + fp\_for\_thresh) > 0)

avg\_iou = avg\_iou / (tp\_for\_thresh + fp\_for\_thresh);

int class\_id;

for(class\_id = 0; class\_id < classes; class\_id++){

if ((tp\_for\_thresh\_per\_class[class\_id] + fp\_for\_thresh\_per\_class[class\_id]) > 0)

avg\_iou\_per\_class[class\_id] = avg\_iou\_per\_class[class\_id] / (tp\_for\_thresh\_per\_class[class\_id] + fp\_for\_thresh\_per\_class[class\_id]);

}

// SORT(detections)

qsort(detections, detections\_count, sizeof(box\_prob), detections\_comparator);

typedef struct {

double precision;

double recall;

int tp, fp, fn;

} pr\_t;

// for PR-curve

pr\_t\*\* pr = (pr\_t\*\*)xcalloc(classes, sizeof(pr\_t\*));

for (i = 0; i < classes; ++i) {

pr[i] = (pr\_t\*)xcalloc(detections\_count, sizeof(pr\_t));

}

printf("\n detections\_count = %d, unique\_truth\_count = %d \n", detections\_count, unique\_truth\_count);

int\* detection\_per\_class\_count = (int\*)xcalloc(classes, sizeof(int));

for (j = 0; j < detections\_count; ++j) {

detection\_per\_class\_count[detections[j].class\_id]++;

}

int\* truth\_flags = (int\*)xcalloc(unique\_truth\_count, sizeof(int));

int rank;

for (rank = 0; rank < detections\_count; ++rank) {

if (rank % 100 == 0)

printf(" rank = %d of ranks = %d \r", rank, detections\_count);

if (rank > 0) {

int class\_id;

for (class\_id = 0; class\_id < classes; ++class\_id) {

pr[class\_id][rank].tp = pr[class\_id][rank - 1].tp;

pr[class\_id][rank].fp = pr[class\_id][rank - 1].fp;

}

}

box\_prob d = detections[rank];

// if (detected && isn't detected before)

if (d.truth\_flag == 1) {

if (truth\_flags[d.unique\_truth\_index] == 0)

{

truth\_flags[d.unique\_truth\_index] = 1;

pr[d.class\_id][rank].tp++; // true-positive

} else

pr[d.class\_id][rank].fp++;

}

else {

pr[d.class\_id][rank].fp++; // false-positive

}

for (i = 0; i < classes; ++i)

{

const int tp = pr[i][rank].tp;

const int fp = pr[i][rank].fp;

const int fn = truth\_classes\_count[i] - tp; // false-negative = objects - true-positive

pr[i][rank].fn = fn;

if ((tp + fp) > 0) pr[i][rank].precision = (double)tp / (double)(tp + fp);

else pr[i][rank].precision = 0;

if ((tp + fn) > 0) pr[i][rank].recall = (double)tp / (double)(tp + fn);

else pr[i][rank].recall = 0;

if (rank == (detections\_count - 1) && detection\_per\_class\_count[i] != (tp + fp)) { // check for last rank

printf(" class\_id: %d - detections = %d, tp+fp = %d, tp = %d, fp = %d \n", i, detection\_per\_class\_count[i], tp+fp, tp, fp);

}

}

}

free(truth\_flags);

double mean\_average\_precision = 0;

for (i = 0; i < classes; ++i) {

double avg\_precision = 0;

// MS COCO - uses 101-Recall-points on PR-chart.

// PascalVOC2007 - uses 11-Recall-points on PR-chart.

// PascalVOC2010-2012 - uses Area-Under-Curve on PR-chart.

// ImageNet - uses Area-Under-Curve on PR-chart.

// correct mAP calculation: ImageNet, PascalVOC 2010-2012

if (map\_points == 0)

{

double last\_recall = pr[i][detections\_count - 1].recall;

double last\_precision = pr[i][detections\_count - 1].precision;

for (rank = detections\_count - 2; rank >= 0; --rank)

{

double delta\_recall = last\_recall - pr[i][rank].recall;

last\_recall = pr[i][rank].recall;

if (pr[i][rank].precision > last\_precision) {

last\_precision = pr[i][rank].precision;

}

avg\_precision += delta\_recall \* last\_precision;

}

}

// MSCOCO - 101 Recall-points, PascalVOC - 11 Recall-points

else

{

int point;

for (point = 0; point < map\_points; ++point) {

double cur\_recall = point \* 1.0 / (map\_points-1);

double cur\_precision = 0;

for (rank = 0; rank < detections\_count; ++rank)

{

if (pr[i][rank].recall >= cur\_recall) { // > or >=

if (pr[i][rank].precision > cur\_precision) {

cur\_precision = pr[i][rank].precision;

}

}

}

//printf("class\_id = %d, point = %d, cur\_recall = %.4f, cur\_precision = %.4f \n", i, point, cur\_recall, cur\_precision);

avg\_precision += cur\_precision;

}

avg\_precision = avg\_precision / map\_points;

}

printf("class\_id = %d, name = %s, ap = %2.2f%% \t (TP = %d, FP = %d) \n",

i, names[i], avg\_precision \* 100, tp\_for\_thresh\_per\_class[i], fp\_for\_thresh\_per\_class[i]);

float class\_precision = (float)tp\_for\_thresh\_per\_class[i] / ((float)tp\_for\_thresh\_per\_class[i] + (float)fp\_for\_thresh\_per\_class[i]);

float class\_recall = (float)tp\_for\_thresh\_per\_class[i] / ((float)tp\_for\_thresh\_per\_class[i] + (float)(truth\_classes\_count[i] - tp\_for\_thresh\_per\_class[i]));

//printf("Precision = %1.2f, Recall = %1.2f, avg IOU = %2.2f%% \n\n", class\_precision, class\_recall, avg\_iou\_per\_class[i]);

mean\_average\_precision += avg\_precision;

}

const float cur\_precision = (float)tp\_for\_thresh / ((float)tp\_for\_thresh + (float)fp\_for\_thresh);

const float cur\_recall = (float)tp\_for\_thresh / ((float)tp\_for\_thresh + (float)(unique\_truth\_count - tp\_for\_thresh));

const float f1\_score = 2.F \* cur\_precision \* cur\_recall / (cur\_precision + cur\_recall);

printf("\n for conf\_thresh = %1.2f, precision = %1.2f, recall = %1.2f, F1-score = %1.2f \n",

thresh\_calc\_avg\_iou, cur\_precision, cur\_recall, f1\_score);

printf(" for conf\_thresh = %0.2f, TP = %d, FP = %d, FN = %d, average IoU = %2.2f %% \n",

thresh\_calc\_avg\_iou, tp\_for\_thresh, fp\_for\_thresh, unique\_truth\_count - tp\_for\_thresh, avg\_iou \* 100);

mean\_average\_precision = mean\_average\_precision / classes;

printf("\n IoU threshold = %2.0f %%, ", iou\_thresh \* 100);

if (map\_points) printf("used %d Recall-points \n", map\_points);

else printf("used Area-Under-Curve for each unique Recall \n");

printf(" mean average precision (mAP@%0.2f) = %f, or %2.2f %% \n", iou\_thresh, mean\_average\_precision, mean\_average\_precision \* 100);

for (i = 0; i < classes; ++i) {

free(pr[i]);

}

free(pr);

free(detections);

free(truth\_classes\_count);

free(detection\_per\_class\_count);

free(avg\_iou\_per\_class);

free(tp\_for\_thresh\_per\_class);

free(fp\_for\_thresh\_per\_class);

fprintf(stderr, "Total Detection Time: %d Seconds\n", (int)(time(0) - start));

printf("\nSet -points flag:\n");

printf(" `-points 101` for MS COCO \n");

printf(" `-points 11` for PascalVOC 2007 (uncomment `difficult` in voc.data) \n");

printf(" `-points 0` (AUC) for ImageNet, PascalVOC 2010-2012, your custom dataset\n");

if (reinforcement\_fd != NULL) fclose(reinforcement\_fd);

// free memory

free\_ptrs((void\*\*)names, net.layers[net.n - 1].classes);

free\_list\_contents\_kvp(options);

free\_list(options);

if (existing\_net) {

//set\_batch\_network(&net, initial\_batch);

//free\_network\_recurrent\_state(\*existing\_net);

restore\_network\_recurrent\_state(\*existing\_net);

//randomize\_network\_recurrent\_state(\*existing\_net);

}

else {

free\_network(net);

}

if (val) free(val);

if (val\_resized) free(val\_resized);

if (thr) free(thr);

if (buf) free(buf);

if (buf\_resized) free(buf\_resized);

return mean\_average\_precision;

}

typedef struct {

float w, h;

} anchors\_t;

int anchors\_comparator(const void \*pa, const void \*pb)

{

anchors\_t a = \*(const anchors\_t \*)pa;

anchors\_t b = \*(const anchors\_t \*)pb;

float diff = b.w\*b.h - a.w\*a.h;

if (diff < 0) return 1;

else if (diff > 0) return -1;

return 0;

}

int anchors\_data\_comparator(const float \*\*pa, const float \*\*pb)

{

float \*a = (float \*)\*pa;

float \*b = (float \*)\*pb;

float diff = b[0] \* b[1] - a[0] \* a[1];

if (diff < 0) return 1;

else if (diff > 0) return -1;

return 0;

}

void calc\_anchors(char \*datacfg, int num\_of\_clusters, int width, int height, int show)

{

printf("\n num\_of\_clusters = %d, width = %d, height = %d \n", num\_of\_clusters, width, height);

if (width < 0 || height < 0) {

printf("Usage: darknet detector calc\_anchors data/voc.data -num\_of\_clusters 9 -width 416 -height 416 \n");

printf("Error: set width and height \n");

return;

}

//float pointsdata[] = { 1,1, 2,2, 6,6, 5,5, 10,10 };

float\* rel\_width\_height\_array = (float\*)xcalloc(1000, sizeof(float));

list \*options = read\_data\_cfg(datacfg);

char \*train\_images = option\_find\_str(options, "train", "data/train.list");

list \*plist = get\_paths(train\_images);

int number\_of\_images = plist->size;

char \*\*paths = (char \*\*)list\_to\_array(plist);

int classes = option\_find\_int(options, "classes", 1);

int\* counter\_per\_class = (int\*)xcalloc(classes, sizeof(int));

srand(time(0));

int number\_of\_boxes = 0;

printf(" read labels from %d images \n", number\_of\_images);

int i, j;

for (i = 0; i < number\_of\_images; ++i) {

char \*path = paths[i];

char labelpath[4096];

replace\_image\_to\_label(path, labelpath);

int num\_labels = 0;

box\_label \*truth = read\_boxes(labelpath, &num\_labels);

//printf(" new path: %s \n", labelpath);

char \*buff = (char\*)xcalloc(6144, sizeof(char));

for (j = 0; j < num\_labels; ++j)

{

if (truth[j].x > 1 || truth[j].x <= 0 || truth[j].y > 1 || truth[j].y <= 0 ||

truth[j].w > 1 || truth[j].w <= 0 || truth[j].h > 1 || truth[j].h <= 0)

{

printf("\n\nWrong label: %s - j = %d, x = %f, y = %f, width = %f, height = %f \n",

labelpath, j, truth[j].x, truth[j].y, truth[j].w, truth[j].h);

sprintf(buff, "echo \"Wrong label: %s - j = %d, x = %f, y = %f, width = %f, height = %f\" >> bad\_label.list",

labelpath, j, truth[j].x, truth[j].y, truth[j].w, truth[j].h);

system(buff);

if (check\_mistakes) getchar();

}

if (truth[j].id >= classes) {

classes = truth[j].id + 1;

counter\_per\_class = (int\*)xrealloc(counter\_per\_class, classes \* sizeof(int));

}

counter\_per\_class[truth[j].id]++;

number\_of\_boxes++;

rel\_width\_height\_array = (float\*)xrealloc(rel\_width\_height\_array, 2 \* number\_of\_boxes \* sizeof(float));

rel\_width\_height\_array[number\_of\_boxes \* 2 - 2] = truth[j].w \* width;

rel\_width\_height\_array[number\_of\_boxes \* 2 - 1] = truth[j].h \* height;

printf("\r loaded \t image: %d \t box: %d", i + 1, number\_of\_boxes);

}

free(buff);

}

printf("\n all loaded. \n");

printf("\n calculating k-means++ ...");

matrix boxes\_data;

model anchors\_data;

boxes\_data = make\_matrix(number\_of\_boxes, 2);

printf("\n");

for (i = 0; i < number\_of\_boxes; ++i) {

boxes\_data.vals[i][0] = rel\_width\_height\_array[i \* 2];

boxes\_data.vals[i][1] = rel\_width\_height\_array[i \* 2 + 1];

//if (w > 410 || h > 410) printf("i:%d, w = %f, h = %f \n", i, w, h);

}

// Is used: distance(box, centroid) = 1 - IoU(box, centroid)

// K-means

anchors\_data = do\_kmeans(boxes\_data, num\_of\_clusters);

qsort((void\*)anchors\_data.centers.vals, num\_of\_clusters, 2 \* sizeof(float), (\_\_compar\_fn\_t)anchors\_data\_comparator);

//gen\_anchors.py = 1.19, 1.99, 2.79, 4.60, 4.53, 8.92, 8.06, 5.29, 10.32, 10.66

//float orig\_anch[] = { 1.19, 1.99, 2.79, 4.60, 4.53, 8.92, 8.06, 5.29, 10.32, 10.66 };

printf("\n");

float avg\_iou = 0;

for (i = 0; i < number\_of\_boxes; ++i) {

float box\_w = rel\_width\_height\_array[i \* 2]; //points->data.fl[i \* 2];

float box\_h = rel\_width\_height\_array[i \* 2 + 1]; //points->data.fl[i \* 2 + 1];

//int cluster\_idx = labels->data.i[i];

int cluster\_idx = 0;

float min\_dist = FLT\_MAX;

float best\_iou = 0;

for (j = 0; j < num\_of\_clusters; ++j) {

float anchor\_w = anchors\_data.centers.vals[j][0]; // centers->data.fl[j \* 2];

float anchor\_h = anchors\_data.centers.vals[j][1]; // centers->data.fl[j \* 2 + 1];

float min\_w = (box\_w < anchor\_w) ? box\_w : anchor\_w;

float min\_h = (box\_h < anchor\_h) ? box\_h : anchor\_h;

float box\_intersect = min\_w\*min\_h;

float box\_union = box\_w\*box\_h + anchor\_w\*anchor\_h - box\_intersect;

float iou = box\_intersect / box\_union;

float distance = 1 - iou;

if (distance < min\_dist) {

min\_dist = distance;

cluster\_idx = j;

best\_iou = iou;

}

}

float anchor\_w = anchors\_data.centers.vals[cluster\_idx][0]; //centers->data.fl[cluster\_idx \* 2];

float anchor\_h = anchors\_data.centers.vals[cluster\_idx][1]; //centers->data.fl[cluster\_idx \* 2 + 1];

if (best\_iou > 1 || best\_iou < 0) { // || box\_w > width || box\_h > height) {

printf(" Wrong label: i = %d, box\_w = %f, box\_h = %f, anchor\_w = %f, anchor\_h = %f, iou = %f \n",

i, box\_w, box\_h, anchor\_w, anchor\_h, best\_iou);

}

else avg\_iou += best\_iou;

}

char buff[1024];

FILE\* fwc = fopen("counters\_per\_class.txt", "wb");

if (fwc) {

sprintf(buff, "counters\_per\_class = ");

printf("\n%s", buff);

fwrite(buff, sizeof(char), strlen(buff), fwc);

for (i = 0; i < classes; ++i) {

sprintf(buff, "%d", counter\_per\_class[i]);

printf("%s", buff);

fwrite(buff, sizeof(char), strlen(buff), fwc);

if (i < classes - 1) {

fwrite(", ", sizeof(char), 2, fwc);

printf(", ");

}

}

printf("\n");

fclose(fwc);

}

else {

printf(" Error: file counters\_per\_class.txt can't be open \n");

}

avg\_iou = 100 \* avg\_iou / number\_of\_boxes;

printf("\n avg IoU = %2.2f %% \n", avg\_iou);

FILE\* fw = fopen("anchors.txt", "wb");

if (fw) {

printf("\nSaving anchors to the file: anchors.txt \n");

printf("anchors = ");

for (i = 0; i < num\_of\_clusters; ++i) {

float anchor\_w = anchors\_data.centers.vals[i][0]; //centers->data.fl[i \* 2];

float anchor\_h = anchors\_data.centers.vals[i][1]; //centers->data.fl[i \* 2 + 1];

if (width > 32) sprintf(buff, "%3.0f,%3.0f", anchor\_w, anchor\_h);

else sprintf(buff, "%2.4f,%2.4f", anchor\_w, anchor\_h);

printf("%s", buff);

fwrite(buff, sizeof(char), strlen(buff), fw);

if (i + 1 < num\_of\_clusters) {

fwrite(", ", sizeof(char), 2, fw);

printf(", ");

}

}

printf("\n");

fclose(fw);

}

else {

printf(" Error: file anchors.txt can't be open \n");

}

if (show) {

#ifdef OPENCV

show\_acnhors(number\_of\_boxes, num\_of\_clusters, rel\_width\_height\_array, anchors\_data, width, height);

#endif // OPENCV

}

free(rel\_width\_height\_array);

free(counter\_per\_class);

getchar();

}

void test\_detector(char \*datacfg, char \*cfgfile, char \*weightfile, char \*filename, float thresh,

float hier\_thresh, int dont\_show, int ext\_output, int save\_labels, char \*outfile, int letter\_box, int benchmark\_layers)

{

list \*options = read\_data\_cfg(datacfg);

char \*name\_list = option\_find\_str(options, "names", "data/names.list");

int names\_size = 0;

char \*\*names = get\_labels\_custom(name\_list, &names\_size); //get\_labels(name\_list);

image \*\*alphabet = load\_alphabet();

network net = parse\_network\_cfg\_custom(cfgfile, 1, 1); // set batch=1

if (weightfile) {

load\_weights(&net, weightfile);

}

net.benchmark\_layers = benchmark\_layers;

fuse\_conv\_batchnorm(net);

calculate\_binary\_weights(net);

if (net.layers[net.n - 1].classes != names\_size) {

printf("\n Error: in the file %s number of names %d that isn't equal to classes=%d in the file %s \n",

name\_list, names\_size, net.layers[net.n - 1].classes, cfgfile);

if (net.layers[net.n - 1].classes > names\_size) getchar();

}

srand(2222222);

char buff[256];

char \*input = buff;

char \*json\_buf = NULL;

int json\_image\_id = 0;

FILE\* json\_file = NULL;

if (outfile) {

json\_file = fopen(outfile, "wb");

if(!json\_file) {

error("fopen failed");

}

char \*tmp = "[\n";

fwrite(tmp, sizeof(char), strlen(tmp), json\_file);

}

int j;

float nms = .45; // 0.4F

while (1) {

if (filename) {

strncpy(input, filename, 256);

if (strlen(input) > 0)

if (input[strlen(input) - 1] == 0x0d) input[strlen(input) - 1] = 0;

}

else {

printf("Enter Image Path: ");

fflush(stdout);

input = fgets(input, 256, stdin);

if (!input) break;

strtok(input, "\n");

}

//image im;

//image sized = load\_image\_resize(input, net.w, net.h, net.c, &im);

image im = load\_image(input, 0, 0, net.c);

image sized;

if(letter\_box) sized = letterbox\_image(im, net.w, net.h);

else sized = resize\_image(im, net.w, net.h);

layer l = net.layers[net.n - 1];

//box \*boxes = calloc(l.w\*l.h\*l.n, sizeof(box));

//float \*\*probs = calloc(l.w\*l.h\*l.n, sizeof(float\*));

//for(j = 0; j < l.w\*l.h\*l.n; ++j) probs[j] = (float\*)xcalloc(l.classes, sizeof(float));

float \*X = sized.data;

//time= what\_time\_is\_it\_now();

double time = get\_time\_point();

network\_predict(net, X);

//network\_predict\_image(&net, im); letterbox = 1;

printf("%s: Predicted in %lf milli-seconds.\n", input, ((double)get\_time\_point() - time) / 1000);

//printf("%s: Predicted in %f seconds.\n", input, (what\_time\_is\_it\_now()-time));

int nboxes = 0;

detection \*dets = get\_network\_boxes(&net, im.w, im.h, thresh, hier\_thresh, 0, 1, &nboxes, letter\_box);

if (nms) {

if (l.nms\_kind == DEFAULT\_NMS) do\_nms\_sort(dets, nboxes, l.classes, nms);

else diounms\_sort(dets, nboxes, l.classes, nms, l.nms\_kind, l.beta\_nms);

}

draw\_detections\_v3(im, dets, nboxes, thresh, names, alphabet, l.classes, ext\_output);

save\_image(im, "predictions");

if (!dont\_show) {

show\_image(im, "predictions");

}

if (json\_file) {

if (json\_buf) {

char \*tmp = ", \n";

fwrite(tmp, sizeof(char), strlen(tmp), json\_file);

}

++json\_image\_id;

json\_buf = detection\_to\_json(dets, nboxes, l.classes, names, json\_image\_id, input);

fwrite(json\_buf, sizeof(char), strlen(json\_buf), json\_file);

free(json\_buf);

}

// pseudo labeling concept - fast.ai

if (save\_labels)

{

char labelpath[4096];

replace\_image\_to\_label(input, labelpath);

FILE\* fw = fopen(labelpath, "wb");

int i;

for (i = 0; i < nboxes; ++i) {

char buff[1024];

int class\_id = -1;

float prob = 0;

for (j = 0; j < l.classes; ++j) {

if (dets[i].prob[j] > thresh && dets[i].prob[j] > prob) {

prob = dets[i].prob[j];

class\_id = j;

}

}

if (class\_id >= 0) {

sprintf(buff, "%d %2.4f %2.4f %2.4f %2.4f\n", class\_id, dets[i].bbox.x, dets[i].bbox.y, dets[i].bbox.w, dets[i].bbox.h);

fwrite(buff, sizeof(char), strlen(buff), fw);

}

}

fclose(fw);

}

free\_detections(dets, nboxes);

free\_image(im);

free\_image(sized);

if (!dont\_show) {

wait\_until\_press\_key\_cv();

destroy\_all\_windows\_cv();

}

if (filename) break;

}

if (json\_file) {

char \*tmp = "\n]";

fwrite(tmp, sizeof(char), strlen(tmp), json\_file);

fclose(json\_file);

}

// free memory

free\_ptrs((void\*\*)names, net.layers[net.n - 1].classes);

free\_list\_contents\_kvp(options);

free\_list(options);

int i;

const int nsize = 8;

for (j = 0; j < nsize; ++j) {

for (i = 32; i < 127; ++i) {

free\_image(alphabet[j][i]);

}

free(alphabet[j]);

}

free(alphabet);

free\_network(net);

}

#if defined(OPENCV) && defined(GPU)

// adversarial attack dnn

void draw\_object(char \*datacfg, char \*cfgfile, char \*weightfile, char \*filename, float thresh, int dont\_show, int it\_num,

int letter\_box, int benchmark\_layers)

{

list \*options = read\_data\_cfg(datacfg);

char \*name\_list = option\_find\_str(options, "names", "data/names.list");

int names\_size = 0;

char \*\*names = get\_labels\_custom(name\_list, &names\_size); //get\_labels(name\_list);

image \*\*alphabet = load\_alphabet();

network net = parse\_network\_cfg(cfgfile);// parse\_network\_cfg\_custom(cfgfile, 1, 1); // set batch=1

net.adversarial = 1;

set\_batch\_network(&net, 1);

if (weightfile) {

load\_weights(&net, weightfile);

}

net.benchmark\_layers = benchmark\_layers;

//fuse\_conv\_batchnorm(net);

//calculate\_binary\_weights(net);

if (net.layers[net.n - 1].classes != names\_size) {

printf("\n Error: in the file %s number of names %d that isn't equal to classes=%d in the file %s \n",

name\_list, names\_size, net.layers[net.n - 1].classes, cfgfile);

if (net.layers[net.n - 1].classes > names\_size) getchar();

}

srand(2222222);

char buff[256];

char \*input = buff;

int j;

float nms = .45; // 0.4F

while (1) {

if (filename) {

strncpy(input, filename, 256);

if (strlen(input) > 0)

if (input[strlen(input) - 1] == 0x0d) input[strlen(input) - 1] = 0;

}

else {

printf("Enter Image Path: ");

fflush(stdout);

input = fgets(input, 256, stdin);

if (!input) break;

strtok(input, "\n");

}

//image im;

//image sized = load\_image\_resize(input, net.w, net.h, net.c, &im);

image im = load\_image(input, 0, 0, net.c);

image sized;

if (letter\_box) sized = letterbox\_image(im, net.w, net.h);

else sized = resize\_image(im, net.w, net.h);

image src\_sized = copy\_image(sized);

layer l = net.layers[net.n - 1];

net.num\_boxes = l.max\_boxes;

int num\_truth = l.truths;

float \*truth\_cpu = (float \*)xcalloc(num\_truth, sizeof(float));

int \*it\_num\_set = (int \*)xcalloc(1, sizeof(int));

float \*lr\_set = (float \*)xcalloc(1, sizeof(float));

int \*boxonly = (int \*)xcalloc(1, sizeof(int));

cv\_draw\_object(sized, truth\_cpu, net.num\_boxes, num\_truth, it\_num\_set, lr\_set, boxonly, l.classes, names);

net.learning\_rate = \*lr\_set;

it\_num = \*it\_num\_set;

float \*X = sized.data;

mat\_cv\* img = NULL;

float max\_img\_loss = 5;

int number\_of\_lines = 100;

int img\_size = 1000;

char windows\_name[100];

char \*base = basecfg(cfgfile);

sprintf(windows\_name, "chart\_%s.png", base);

img = draw\_train\_chart(windows\_name, max\_img\_loss, it\_num, number\_of\_lines, img\_size, dont\_show, NULL);

int iteration;

for (iteration = 0; iteration < it\_num; ++iteration)

{

forward\_backward\_network\_gpu(net, X, truth\_cpu);

float avg\_loss = get\_network\_cost(net);

draw\_train\_loss(windows\_name, img, img\_size, avg\_loss, max\_img\_loss, iteration, it\_num, 0, 0, "mAP%", dont\_show, 0, 0);

float inv\_loss = 1.0 / max\_val\_cmp(0.01, avg\_loss);

//net.learning\_rate = \*lr\_set \* inv\_loss;

if (\*boxonly) {

int dw = truth\_cpu[2] \* sized.w, dh = truth\_cpu[3] \* sized.h;

int dx = truth\_cpu[0] \* sized.w - dw / 2, dy = truth\_cpu[1] \* sized.h - dh / 2;

image crop = crop\_image(sized, dx, dy, dw, dh);

copy\_image\_inplace(src\_sized, sized);

embed\_image(crop, sized, dx, dy);

}

show\_image\_cv(sized, "image\_optimization");

wait\_key\_cv(20);

}

net.train = 0;

quantize\_image(sized);

network\_predict(net, X);

save\_image\_png(sized, "drawn");

//sized = load\_image("drawn.png", 0, 0, net.c);

int nboxes = 0;

detection \*dets = get\_network\_boxes(&net, sized.w, sized.h, thresh, 0, 0, 1, &nboxes, letter\_box);

if (nms) {

if (l.nms\_kind == DEFAULT\_NMS) do\_nms\_sort(dets, nboxes, l.classes, nms);

else diounms\_sort(dets, nboxes, l.classes, nms, l.nms\_kind, l.beta\_nms);

}

draw\_detections\_v3(sized, dets, nboxes, thresh, names, alphabet, l.classes, 1);

save\_image(sized, "pre\_predictions");

if (!dont\_show) {

show\_image(sized, "pre\_predictions");

}

free\_detections(dets, nboxes);

free\_image(im);

free\_image(sized);

free\_image(src\_sized);

if (!dont\_show) {

wait\_until\_press\_key\_cv();

destroy\_all\_windows\_cv();

}

free(lr\_set);

free(it\_num\_set);

if (filename) break;

}

// free memory

free\_ptrs((void\*\*)names, net.layers[net.n - 1].classes);

free\_list\_contents\_kvp(options);

free\_list(options);

int i;

const int nsize = 8;

for (j = 0; j < nsize; ++j) {

for (i = 32; i < 127; ++i) {

free\_image(alphabet[j][i]);

}

free(alphabet[j]);

}

free(alphabet);

free\_network(net);

}

#else // defined(OPENCV) && defined(GPU)

void draw\_object(char \*datacfg, char \*cfgfile, char \*weightfile, char \*filename, float thresh, int dont\_show, int it\_num,

int letter\_box, int benchmark\_layers)

{

printf(" ./darknet detector draw ... can't be used without OpenCV and CUDA! \n");

getchar();

}

#endif // defined(OPENCV) && defined(GPU)

void run\_detector(int argc, char \*\*argv)

{

int dont\_show = find\_arg(argc, argv, "-dont\_show");

int benchmark = find\_arg(argc, argv, "-benchmark");

int benchmark\_layers = find\_arg(argc, argv, "-benchmark\_layers");

//if (benchmark\_layers) benchmark = 1;

if (benchmark) dont\_show = 1;

int show = find\_arg(argc, argv, "-show");

int letter\_box = find\_arg(argc, argv, "-letter\_box");

int calc\_map = find\_arg(argc, argv, "-map");

int map\_points = find\_int\_arg(argc, argv, "-points", 0);

check\_mistakes = find\_arg(argc, argv, "-check\_mistakes");

int show\_imgs = find\_arg(argc, argv, "-show\_imgs");

int mjpeg\_port = find\_int\_arg(argc, argv, "-mjpeg\_port", -1);

int json\_port = find\_int\_arg(argc, argv, "-json\_port", -1);

char \*http\_post\_host = find\_char\_arg(argc, argv, "-http\_post\_host", 0);

int time\_limit\_sec = find\_int\_arg(argc, argv, "-time\_limit\_sec", 0);

char \*out\_filename = find\_char\_arg(argc, argv, "-out\_filename", 0);

char \*outfile = find\_char\_arg(argc, argv, "-out", 0);

char \*prefix = find\_char\_arg(argc, argv, "-prefix", 0);

float thresh = find\_float\_arg(argc, argv, "-thresh", .25); // 0.24

float iou\_thresh = find\_float\_arg(argc, argv, "-iou\_thresh", .5); // 0.5 for mAP

float hier\_thresh = find\_float\_arg(argc, argv, "-hier", .5);

int cam\_index = find\_int\_arg(argc, argv, "-c", 0);

int frame\_skip = find\_int\_arg(argc, argv, "-s", 0);

int num\_of\_clusters = find\_int\_arg(argc, argv, "-num\_of\_clusters", 5);

int width = find\_int\_arg(argc, argv, "-width", -1);

int height = find\_int\_arg(argc, argv, "-height", -1);

// extended output in test mode (output of rect bound coords)

// and for recall mode (extended output table-like format with results for best\_class fit)

int ext\_output = find\_arg(argc, argv, "-ext\_output");

int save\_labels = find\_arg(argc, argv, "-save\_labels");

char\* chart\_path = find\_char\_arg(argc, argv, "-chart", 0);

if (argc < 4) {

fprintf(stderr, "usage: %s %s [train/test/valid/demo/map] [data] [cfg] [weights (optional)]\n", argv[0], argv[1]);

return;

}

char \*gpu\_list = find\_char\_arg(argc, argv, "-gpus", 0);

int \*gpus = 0;

int gpu = 0;

int ngpus = 0;

if (gpu\_list) {

printf("%s\n", gpu\_list);

int len = (int)strlen(gpu\_list);

ngpus = 1;

int i;

for (i = 0; i < len; ++i) {

if (gpu\_list[i] == ',') ++ngpus;

}

gpus = (int\*)xcalloc(ngpus, sizeof(int));

for (i = 0; i < ngpus; ++i) {

gpus[i] = atoi(gpu\_list);

gpu\_list = strchr(gpu\_list, ',') + 1;

}

}

else {

gpu = gpu\_index;

gpus = &gpu;

ngpus = 1;

}

int clear = find\_arg(argc, argv, "-clear");

char \*datacfg = argv[3];

char \*cfg = argv[4];

char \*weights = (argc > 5) ? argv[5] : 0;

if (weights)

if (strlen(weights) > 0)

if (weights[strlen(weights) - 1] == 0x0d) weights[strlen(weights) - 1] = 0;

char \*filename = (argc > 6) ? argv[6] : 0;

if (0 == strcmp(argv[2], "test")) test\_detector(datacfg, cfg, weights, filename, thresh, hier\_thresh, dont\_show, ext\_output, save\_labels, outfile, letter\_box, benchmark\_layers);

else if (0 == strcmp(argv[2], "train")) train\_detector(datacfg, cfg, weights, gpus, ngpus, clear, dont\_show, calc\_map, mjpeg\_port, show\_imgs, benchmark\_layers, chart\_path);

else if (0 == strcmp(argv[2], "valid")) validate\_detector(datacfg, cfg, weights, outfile);

else if (0 == strcmp(argv[2], "recall")) validate\_detector\_recall(datacfg, cfg, weights);

else if (0 == strcmp(argv[2], "map")) validate\_detector\_map(datacfg, cfg, weights, thresh, iou\_thresh, map\_points, letter\_box, NULL);

else if (0 == strcmp(argv[2], "calc\_anchors")) calc\_anchors(datacfg, num\_of\_clusters, width, height, show);

else if (0 == strcmp(argv[2], "draw")) {

int it\_num = 100;

draw\_object(datacfg, cfg, weights, filename, thresh, dont\_show, it\_num, letter\_box, benchmark\_layers);

}

else if (0 == strcmp(argv[2], "demo")) {

list \*options = read\_data\_cfg(datacfg);

int classes = option\_find\_int(options, "classes", 20);

char \*name\_list = option\_find\_str(options, "names", "data/names.list");

char \*\*names = get\_labels(name\_list);

if (filename)

if (strlen(filename) > 0)

if (filename[strlen(filename) - 1] == 0x0d) filename[strlen(filename) - 1] = 0;

demo(cfg, weights, thresh, hier\_thresh, cam\_index, filename, names, classes, frame\_skip, prefix, out\_filename,

mjpeg\_port, json\_port, dont\_show, ext\_output, letter\_box, time\_limit\_sec, http\_post\_host, benchmark, benchmark\_layers);

free\_list\_contents\_kvp(options);

free\_list(options);

}

else printf(" There isn't such command: %s", argv[2]);

if (gpus && gpu\_list && ngpus > 1) free(gpus);

}