#include "data.h"

#include "utils.h"

#include "image.h"

#include "dark\_cuda.h"

#include "box.h"

#include "http\_stream.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

extern int check\_mistakes;

#define NUMCHARS 37

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

list \*get\_paths(char \*filename)

{

char \*path;

FILE \*file = fopen(filename, "r");

if(!file) file\_error(filename);

list \*lines = make\_list();

while((path=fgetl(file))){

list\_insert(lines, path);

}

fclose(file);

return lines;

}

/\*

char \*\*get\_random\_paths\_indexes(char \*\*paths, int n, int m, int \*indexes)

{

char \*\*random\_paths = calloc(n, sizeof(char\*));

int i;

pthread\_mutex\_lock(&mutex);

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

int index = random\_gen()%m;

indexes[i] = index;

random\_paths[i] = paths[index];

if(i == 0) printf("%s\n", paths[index]);

}

pthread\_mutex\_unlock(&mutex);

return random\_paths;

}

\*/

char \*\*get\_sequential\_paths(char \*\*paths, int n, int m, int mini\_batch, int augment\_speed)

{

int speed = rand\_int(1, augment\_speed);

if (speed < 1) speed = 1;

char\*\* sequentia\_paths = (char\*\*)xcalloc(n, sizeof(char\*));

int i;

pthread\_mutex\_lock(&mutex);

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

unsigned int \*start\_time\_indexes = (unsigned int \*)xcalloc(mini\_batch, sizeof(unsigned int));

for (i = 0; i < mini\_batch; ++i) {

start\_time\_indexes[i] = random\_gen() % m;

//printf(" start\_time\_indexes[i] = %u, ", start\_time\_indexes[i]);

}

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

do {

int time\_line\_index = i % mini\_batch;

unsigned int index = start\_time\_indexes[time\_line\_index] % m;

start\_time\_indexes[time\_line\_index] += speed;

//int index = random\_gen() % m;

sequentia\_paths[i] = paths[index];

//if(i == 0) printf("%s\n", paths[index]);

//printf(" index = %u - grp: %s \n", index, paths[index]);

if (strlen(sequentia\_paths[i]) <= 4) printf(" Very small path to the image: %s \n", sequentia\_paths[i]);

} while (strlen(sequentia\_paths[i]) == 0);

}

free(start\_time\_indexes);

pthread\_mutex\_unlock(&mutex);

return sequentia\_paths;

}

char \*\*get\_random\_paths(char \*\*paths, int n, int m)

{

char\*\* random\_paths = (char\*\*)xcalloc(n, sizeof(char\*));

int i;

pthread\_mutex\_lock(&mutex);

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

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

do {

int index = random\_gen() % m;

random\_paths[i] = paths[index];

//if(i == 0) printf("%s\n", paths[index]);

//printf("grp: %s\n", paths[index]);

if (strlen(random\_paths[i]) <= 4) printf(" Very small path to the image: %s \n", random\_paths[i]);

} while (strlen(random\_paths[i]) == 0);

}

pthread\_mutex\_unlock(&mutex);

return random\_paths;

}

char \*\*find\_replace\_paths(char \*\*paths, int n, char \*find, char \*replace)

{

char\*\* replace\_paths = (char\*\*)xcalloc(n, sizeof(char\*));

int i;

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

char replaced[4096];

find\_replace(paths[i], find, replace, replaced);

replace\_paths[i] = copy\_string(replaced);

}

return replace\_paths;

}

matrix load\_image\_paths\_gray(char \*\*paths, int n, int w, int h)

{

int i;

matrix X;

X.rows = n;

X.vals = (float\*\*)xcalloc(X.rows, sizeof(float\*));

X.cols = 0;

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

image im = load\_image(paths[i], w, h, 3);

image gray = grayscale\_image(im);

free\_image(im);

im = gray;

X.vals[i] = im.data;

X.cols = im.h\*im.w\*im.c;

}

return X;

}

matrix load\_image\_paths(char \*\*paths, int n, int w, int h)

{

int i;

matrix X;

X.rows = n;

X.vals = (float\*\*)xcalloc(X.rows, sizeof(float\*));

X.cols = 0;

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

image im = load\_image\_color(paths[i], w, h);

X.vals[i] = im.data;

X.cols = im.h\*im.w\*im.c;

}

return X;

}

matrix load\_image\_augment\_paths(char \*\*paths, int n, int use\_flip, int min, int max, int w, int h, float angle, float aspect, float hue, float saturation, float exposure, int dontuse\_opencv)

{

int i;

matrix X;

X.rows = n;

X.vals = (float\*\*)xcalloc(X.rows, sizeof(float\*));

X.cols = 0;

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

int size = w > h ? w : h;

image im;

if(dontuse\_opencv) im = load\_image\_stb\_resize(paths[i], 0, 0, 3);

else im = load\_image\_color(paths[i], 0, 0);

image crop = random\_augment\_image(im, angle, aspect, min, max, size);

int flip = use\_flip ? random\_gen() % 2 : 0;

if (flip)

flip\_image(crop);

random\_distort\_image(crop, hue, saturation, exposure);

image sized = resize\_image(crop, w, h);

//show\_image(im, "orig");

//show\_image(sized, "sized");

//show\_image(sized, paths[i]);

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

//printf("w = %d, h = %d \n", sized.w, sized.h);

free\_image(im);

free\_image(crop);

X.vals[i] = sized.data;

X.cols = sized.h\*sized.w\*sized.c;

}

return X;

}

box\_label \*read\_boxes(char \*filename, int \*n)

{

box\_label\* boxes = (box\_label\*)xcalloc(1, sizeof(box\_label));

FILE \*file = fopen(filename, "r");

if (!file) {

printf("Can't open label file. (This can be normal only if you use MSCOCO): %s \n", filename);

//file\_error(filename);

FILE\* fw = fopen("bad.list", "a");

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

char \*new\_line = "\n";

fwrite(new\_line, sizeof(char), strlen(new\_line), fw);

fclose(fw);

if (check\_mistakes) {

printf("\n Error in read\_boxes() \n");

getchar();

}

\*n = 0;

return boxes;

}

float x, y, h, w;

int id;

int count = 0;

while(fscanf(file, "%d %f %f %f %f", &id, &x, &y, &w, &h) == 5){

boxes = (box\_label\*)xrealloc(boxes, (count + 1) \* sizeof(box\_label));

boxes[count].id = id;

boxes[count].x = x;

boxes[count].y = y;

boxes[count].h = h;

boxes[count].w = w;

boxes[count].left = x - w/2;

boxes[count].right = x + w/2;

boxes[count].top = y - h/2;

boxes[count].bottom = y + h/2;

++count;

}

fclose(file);

\*n = count;

return boxes;

}

void randomize\_boxes(box\_label \*b, int n)

{

int i;

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

box\_label swap = b[i];

int index = random\_gen()%n;

b[i] = b[index];

b[index] = swap;

}

}

void correct\_boxes(box\_label \*boxes, int n, float dx, float dy, float sx, float sy, int flip)

{

int i;

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

if(boxes[i].x == 0 && boxes[i].y == 0) {

boxes[i].x = 999999;

boxes[i].y = 999999;

boxes[i].w = 999999;

boxes[i].h = 999999;

continue;

}

if ((boxes[i].x + boxes[i].w / 2) < 0 || (boxes[i].y + boxes[i].h / 2) < 0 ||

(boxes[i].x - boxes[i].w / 2) > 1 || (boxes[i].y - boxes[i].h / 2) > 1)

{

boxes[i].x = 999999;

boxes[i].y = 999999;

boxes[i].w = 999999;

boxes[i].h = 999999;

continue;

}

boxes[i].left = boxes[i].left \* sx - dx;

boxes[i].right = boxes[i].right \* sx - dx;

boxes[i].top = boxes[i].top \* sy - dy;

boxes[i].bottom = boxes[i].bottom\* sy - dy;

if(flip){

float swap = boxes[i].left;

boxes[i].left = 1. - boxes[i].right;

boxes[i].right = 1. - swap;

}

boxes[i].left = constrain(0, 1, boxes[i].left);

boxes[i].right = constrain(0, 1, boxes[i].right);

boxes[i].top = constrain(0, 1, boxes[i].top);

boxes[i].bottom = constrain(0, 1, boxes[i].bottom);

boxes[i].x = (boxes[i].left+boxes[i].right)/2;

boxes[i].y = (boxes[i].top+boxes[i].bottom)/2;

boxes[i].w = (boxes[i].right - boxes[i].left);

boxes[i].h = (boxes[i].bottom - boxes[i].top);

boxes[i].w = constrain(0, 1, boxes[i].w);

boxes[i].h = constrain(0, 1, boxes[i].h);

}

}

void fill\_truth\_swag(char \*path, float \*truth, int classes, int flip, float dx, float dy, float sx, float sy)

{

char labelpath[4096];

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

int count = 0;

box\_label \*boxes = read\_boxes(labelpath, &count);

randomize\_boxes(boxes, count);

correct\_boxes(boxes, count, dx, dy, sx, sy, flip);

float x,y,w,h;

int id;

int i;

for (i = 0; i < count && i < 30; ++i) {

x = boxes[i].x;

y = boxes[i].y;

w = boxes[i].w;

h = boxes[i].h;

id = boxes[i].id;

if (w < .0 || h < .0) continue;

int index = (4+classes) \* i;

truth[index++] = x;

truth[index++] = y;

truth[index++] = w;

truth[index++] = h;

if (id < classes) truth[index+id] = 1;

}

free(boxes);

}

void fill\_truth\_region(char \*path, float \*truth, int classes, int num\_boxes, int flip, float dx, float dy, float sx, float sy)

{

char labelpath[4096];

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

int count = 0;

box\_label \*boxes = read\_boxes(labelpath, &count);

randomize\_boxes(boxes, count);

correct\_boxes(boxes, count, dx, dy, sx, sy, flip);

float x,y,w,h;

int id;

int i;

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

x = boxes[i].x;

y = boxes[i].y;

w = boxes[i].w;

h = boxes[i].h;

id = boxes[i].id;

if (w < .001 || h < .001) continue;

int col = (int)(x\*num\_boxes);

int row = (int)(y\*num\_boxes);

x = x\*num\_boxes - col;

y = y\*num\_boxes - row;

int index = (col+row\*num\_boxes)\*(5+classes);

if (truth[index]) continue;

truth[index++] = 1;

if (id < classes) truth[index+id] = 1;

index += classes;

truth[index++] = x;

truth[index++] = y;

truth[index++] = w;

truth[index++] = h;

}

free(boxes);

}

int fill\_truth\_detection(const char \*path, int num\_boxes, float \*truth, int classes, int flip, float dx, float dy, float sx, float sy,

int net\_w, int net\_h)

{

char labelpath[4096];

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

int count = 0;

int i;

box\_label \*boxes = read\_boxes(labelpath, &count);

int min\_w\_h = 0;

float lowest\_w = 1.F / net\_w;

float lowest\_h = 1.F / net\_h;

randomize\_boxes(boxes, count);

correct\_boxes(boxes, count, dx, dy, sx, sy, flip);

if (count > num\_boxes) count = num\_boxes;

float x, y, w, h;

int id;

int sub = 0;

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

x = boxes[i].x;

y = boxes[i].y;

w = boxes[i].w;

h = boxes[i].h;

id = boxes[i].id;

// not detect small objects

//if ((w < 0.001F || h < 0.001F)) continue;

// if truth (box for object) is smaller than 1x1 pix

char buff[256];

if (id >= classes) {

printf("\n Wrong annotation: class\_id = %d. But class\_id should be [from 0 to %d], file: %s \n", id, (classes-1), labelpath);

sprintf(buff, "echo %s \"Wrong annotation: class\_id = %d. But class\_id should be [from 0 to %d]\" >> bad\_label.list", labelpath, id, (classes-1));

system(buff);

if (check\_mistakes) getchar();

++sub;

continue;

}

if ((w < lowest\_w || h < lowest\_h)) {

//sprintf(buff, "echo %s \"Very small object: w < lowest\_w OR h < lowest\_h\" >> bad\_label.list", labelpath);

//system(buff);

++sub;

continue;

}

if (x == 999999 || y == 999999) {

printf("\n Wrong annotation: x = 0, y = 0, < 0 or > 1, file: %s \n", labelpath);

sprintf(buff, "echo %s \"Wrong annotation: x = 0 or y = 0\" >> bad\_label.list", labelpath);

system(buff);

++sub;

if (check\_mistakes) getchar();

continue;

}

if (x <= 0 || x > 1 || y <= 0 || y > 1) {

printf("\n Wrong annotation: x = %f, y = %f, file: %s \n", x, y, labelpath);

sprintf(buff, "echo %s \"Wrong annotation: x = %f, y = %f\" >> bad\_label.list", labelpath, x, y);

system(buff);

++sub;

if (check\_mistakes) getchar();

continue;

}

if (w > 1) {

printf("\n Wrong annotation: w = %f, file: %s \n", w, labelpath);

sprintf(buff, "echo %s \"Wrong annotation: w = %f\" >> bad\_label.list", labelpath, w);

system(buff);

w = 1;

if (check\_mistakes) getchar();

}

if (h > 1) {

printf("\n Wrong annotation: h = %f, file: %s \n", h, labelpath);

sprintf(buff, "echo %s \"Wrong annotation: h = %f\" >> bad\_label.list", labelpath, h);

system(buff);

h = 1;

if (check\_mistakes) getchar();

}

if (x == 0) x += lowest\_w;

if (y == 0) y += lowest\_h;

truth[(i-sub)\*5+0] = x;

truth[(i-sub)\*5+1] = y;

truth[(i-sub)\*5+2] = w;

truth[(i-sub)\*5+3] = h;

truth[(i-sub)\*5+4] = id;

if (min\_w\_h == 0) min\_w\_h = w\*net\_w;

if (min\_w\_h > w\*net\_w) min\_w\_h = w\*net\_w;

if (min\_w\_h > h\*net\_h) min\_w\_h = h\*net\_h;

}

free(boxes);

return min\_w\_h;

}

void print\_letters(float \*pred, int n)

{

int i;

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

int index = max\_index(pred+i\*NUMCHARS, NUMCHARS);

printf("%c", int\_to\_alphanum(index));

}

printf("\n");

}

void fill\_truth\_captcha(char \*path, int n, float \*truth)

{

char \*begin = strrchr(path, '/');

++begin;

int i;

for(i = 0; i < strlen(begin) && i < n && begin[i] != '.'; ++i){

int index = alphanum\_to\_int(begin[i]);

if(index > 35) printf("Bad %c\n", begin[i]);

truth[i\*NUMCHARS+index] = 1;

}

for(;i < n; ++i){

truth[i\*NUMCHARS + NUMCHARS-1] = 1;

}

}

data load\_data\_captcha(char \*\*paths, int n, int m, int k, int w, int h)

{

if(m) paths = get\_random\_paths(paths, n, m);

data d = {0};

d.shallow = 0;

d.X = load\_image\_paths(paths, n, w, h);

d.y = make\_matrix(n, k\*NUMCHARS);

int i;

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

fill\_truth\_captcha(paths[i], k, d.y.vals[i]);

}

if(m) free(paths);

return d;

}

data load\_data\_captcha\_encode(char \*\*paths, int n, int m, int w, int h)

{

if(m) paths = get\_random\_paths(paths, n, m);

data d = {0};

d.shallow = 0;

d.X = load\_image\_paths(paths, n, w, h);

d.X.cols = 17100;

d.y = d.X;

if(m) free(paths);

return d;

}

void fill\_truth(char \*path, char \*\*labels, int k, float \*truth)

{

int i;

memset(truth, 0, k\*sizeof(float));

int count = 0;

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

if(strstr(path, labels[i])){

truth[i] = 1;

++count;

}

}

if (count != 1) {

printf("Too many or too few labels: %d, %s\n", count, path);

count = 0;

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

if (strstr(path, labels[i])) {

printf("\t label %d: %s \n", count, labels[i]);

count++;

}

}

}

}

void fill\_truth\_smooth(char \*path, char \*\*labels, int k, float \*truth, float label\_smooth\_eps)

{

int i;

memset(truth, 0, k \* sizeof(float));

int count = 0;

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

if (strstr(path, labels[i])) {

truth[i] = (1 - label\_smooth\_eps);

++count;

}

else {

truth[i] = label\_smooth\_eps / (k - 1);

}

}

if (count != 1) {

printf("Too many or too few labels: %d, %s\n", count, path);

count = 0;

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

if (strstr(path, labels[i])) {

printf("\t label %d: %s \n", count, labels[i]);

count++;

}

}

}

}

void fill\_hierarchy(float \*truth, int k, tree \*hierarchy)

{

int j;

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

if(truth[j]){

int parent = hierarchy->parent[j];

while(parent >= 0){

truth[parent] = 1;

parent = hierarchy->parent[parent];

}

}

}

int i;

int count = 0;

for(j = 0; j < hierarchy->groups; ++j){

//printf("%d\n", count);

int mask = 1;

for(i = 0; i < hierarchy->group\_size[j]; ++i){

if(truth[count + i]){

mask = 0;

break;

}

}

if (mask) {

for(i = 0; i < hierarchy->group\_size[j]; ++i){

truth[count + i] = SECRET\_NUM;

}

}

count += hierarchy->group\_size[j];

}

}

matrix load\_labels\_paths(char \*\*paths, int n, char \*\*labels, int k, tree \*hierarchy, float label\_smooth\_eps)

{

matrix y = make\_matrix(n, k);

int i;

for(i = 0; i < n && labels; ++i){

fill\_truth\_smooth(paths[i], labels, k, y.vals[i], label\_smooth\_eps);

if(hierarchy){

fill\_hierarchy(y.vals[i], k, hierarchy);

}

}

return y;

}

matrix load\_tags\_paths(char \*\*paths, int n, int k)

{

matrix y = make\_matrix(n, k);

int i;

int count = 0;

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

char label[4096];

find\_replace(paths[i], "imgs", "labels", label);

find\_replace(label, "\_iconl.jpeg", ".txt", label);

FILE \*file = fopen(label, "r");

if(!file){

find\_replace(label, "labels", "labels2", label);

file = fopen(label, "r");

if(!file) continue;

}

++count;

int tag;

while(fscanf(file, "%d", &tag) == 1){

if(tag < k){

y.vals[i][tag] = 1;

}

}

fclose(file);

}

printf("%d/%d\n", count, n);

return y;

}

char \*\*get\_labels\_custom(char \*filename, int \*size)

{

list \*plist = get\_paths(filename);

if(size) \*size = plist->size;

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

free\_list(plist);

return labels;

}

char \*\*get\_labels(char \*filename)

{

return get\_labels\_custom(filename, NULL);

}

void free\_data(data d)

{

if(!d.shallow){

free\_matrix(d.X);

free\_matrix(d.y);

}else{

free(d.X.vals);

free(d.y.vals);

}

}

data load\_data\_region(int n, char \*\*paths, int m, int w, int h, int size, int classes, float jitter, float hue, float saturation, float exposure)

{

char \*\*random\_paths = get\_random\_paths(paths, n, m);

int i;

data d = {0};

d.shallow = 0;

d.X.rows = n;

d.X.vals = (float\*\*)xcalloc(d.X.rows, sizeof(float\*));

d.X.cols = h\*w\*3;

int k = size\*size\*(5+classes);

d.y = make\_matrix(n, k);

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

image orig = load\_image\_color(random\_paths[i], 0, 0);

int oh = orig.h;

int ow = orig.w;

int dw = (ow\*jitter);

int dh = (oh\*jitter);

int pleft = rand\_uniform(-dw, dw);

int pright = rand\_uniform(-dw, dw);

int ptop = rand\_uniform(-dh, dh);

int pbot = rand\_uniform(-dh, dh);

int swidth = ow - pleft - pright;

int sheight = oh - ptop - pbot;

float sx = (float)swidth / ow;

float sy = (float)sheight / oh;

int flip = random\_gen()%2;

image cropped = crop\_image(orig, pleft, ptop, swidth, sheight);

float dx = ((float)pleft/ow)/sx;

float dy = ((float)ptop /oh)/sy;

image sized = resize\_image(cropped, w, h);

if(flip) flip\_image(sized);

random\_distort\_image(sized, hue, saturation, exposure);

d.X.vals[i] = sized.data;

fill\_truth\_region(random\_paths[i], d.y.vals[i], classes, size, flip, dx, dy, 1./sx, 1./sy);

free\_image(orig);

free\_image(cropped);

}

free(random\_paths);

return d;

}

data load\_data\_compare(int n, char \*\*paths, int m, int classes, int w, int h)

{

if(m) paths = get\_random\_paths(paths, 2\*n, m);

int i,j;

data d = {0};

d.shallow = 0;

d.X.rows = n;

d.X.vals = (float\*\*)xcalloc(d.X.rows, sizeof(float\*));

d.X.cols = h\*w\*6;

int k = 2\*(classes);

d.y = make\_matrix(n, k);

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

image im1 = load\_image\_color(paths[i\*2], w, h);

image im2 = load\_image\_color(paths[i\*2+1], w, h);

d.X.vals[i] = (float\*)xcalloc(d.X.cols, sizeof(float));

memcpy(d.X.vals[i], im1.data, h\*w\*3\*sizeof(float));

memcpy(d.X.vals[i] + h\*w\*3, im2.data, h\*w\*3\*sizeof(float));

int id;

float iou;

char imlabel1[4096];

char imlabel2[4096];

find\_replace(paths[i\*2], "imgs", "labels", imlabel1);

find\_replace(imlabel1, "jpg", "txt", imlabel1);

FILE \*fp1 = fopen(imlabel1, "r");

while(fscanf(fp1, "%d %f", &id, &iou) == 2){

if (d.y.vals[i][2\*id] < iou) d.y.vals[i][2\*id] = iou;

}

find\_replace(paths[i\*2+1], "imgs", "labels", imlabel2);

find\_replace(imlabel2, "jpg", "txt", imlabel2);

FILE \*fp2 = fopen(imlabel2, "r");

while(fscanf(fp2, "%d %f", &id, &iou) == 2){

if (d.y.vals[i][2\*id + 1] < iou) d.y.vals[i][2\*id + 1] = iou;

}

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

if (d.y.vals[i][2\*j] > .5 && d.y.vals[i][2\*j+1] < .5){

d.y.vals[i][2\*j] = 1;

d.y.vals[i][2\*j+1] = 0;

} else if (d.y.vals[i][2\*j] < .5 && d.y.vals[i][2\*j+1] > .5){

d.y.vals[i][2\*j] = 0;

d.y.vals[i][2\*j+1] = 1;

} else {

d.y.vals[i][2\*j] = SECRET\_NUM;

d.y.vals[i][2\*j+1] = SECRET\_NUM;

}

}

fclose(fp1);

fclose(fp2);

free\_image(im1);

free\_image(im2);

}

if(m) free(paths);

return d;

}

data load\_data\_swag(char \*\*paths, int n, int classes, float jitter)

{

int index = random\_gen()%n;

char \*random\_path = paths[index];

image orig = load\_image\_color(random\_path, 0, 0);

int h = orig.h;

int w = orig.w;

data d = {0};

d.shallow = 0;

d.w = w;

d.h = h;

d.X.rows = 1;

d.X.vals = (float\*\*)xcalloc(d.X.rows, sizeof(float\*));

d.X.cols = h\*w\*3;

int k = (4+classes)\*30;

d.y = make\_matrix(1, k);

int dw = w\*jitter;

int dh = h\*jitter;

int pleft = rand\_uniform(-dw, dw);

int pright = rand\_uniform(-dw, dw);

int ptop = rand\_uniform(-dh, dh);

int pbot = rand\_uniform(-dh, dh);

int swidth = w - pleft - pright;

int sheight = h - ptop - pbot;

float sx = (float)swidth / w;

float sy = (float)sheight / h;

int flip = random\_gen()%2;

image cropped = crop\_image(orig, pleft, ptop, swidth, sheight);

float dx = ((float)pleft/w)/sx;

float dy = ((float)ptop /h)/sy;

image sized = resize\_image(cropped, w, h);

if(flip) flip\_image(sized);

d.X.vals[0] = sized.data;

fill\_truth\_swag(random\_path, d.y.vals[0], classes, flip, dx, dy, 1./sx, 1./sy);

free\_image(orig);

free\_image(cropped);

return d;

}

void blend\_truth(float \*new\_truth, int boxes, float \*old\_truth)

{

const int t\_size = 4 + 1;

int count\_new\_truth = 0;

int t;

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

float x = new\_truth[t\*(4 + 1)];

if (!x) break;

count\_new\_truth++;

}

for (t = count\_new\_truth; t < boxes; ++t) {

float \*new\_truth\_ptr = new\_truth + t\*t\_size;

float \*old\_truth\_ptr = old\_truth + (t - count\_new\_truth)\*t\_size;

float x = old\_truth\_ptr[0];

if (!x) break;

new\_truth\_ptr[0] = old\_truth\_ptr[0];

new\_truth\_ptr[1] = old\_truth\_ptr[1];

new\_truth\_ptr[2] = old\_truth\_ptr[2];

new\_truth\_ptr[3] = old\_truth\_ptr[3];

new\_truth\_ptr[4] = old\_truth\_ptr[4];

}

//printf("\n was %d bboxes, now %d bboxes \n", count\_new\_truth, t);

}

void blend\_truth\_mosaic(float \*new\_truth, int boxes, float \*old\_truth, int w, int h, float cut\_x, float cut\_y, int i\_mixup,

int left\_shift, int right\_shift, int top\_shift, int bot\_shift)

{

const int t\_size = 4 + 1;

int count\_new\_truth = 0;

int t;

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

float x = new\_truth[t\*(4 + 1)];

if (!x) break;

count\_new\_truth++;

}

int new\_t = count\_new\_truth;

for (t = count\_new\_truth; t < boxes; ++t) {

float \*new\_truth\_ptr = new\_truth + new\_t\*t\_size;

new\_truth\_ptr[0] = 0;

float \*old\_truth\_ptr = old\_truth + (t - count\_new\_truth)\*t\_size;

float x = old\_truth\_ptr[0];

if (!x) break;

float xb = old\_truth\_ptr[0];

float yb = old\_truth\_ptr[1];

float wb = old\_truth\_ptr[2];

float hb = old\_truth\_ptr[3];

// shift 4 images

if (i\_mixup == 0) {

xb = xb - (float)(w - cut\_x - right\_shift) / w;

yb = yb - (float)(h - cut\_y - bot\_shift) / h;

}

if (i\_mixup == 1) {

xb = xb + (float)(cut\_x - left\_shift) / w;

yb = yb - (float)(h - cut\_y - bot\_shift) / h;

}

if (i\_mixup == 2) {

xb = xb - (float)(w - cut\_x - right\_shift) / w;

yb = yb + (float)(cut\_y - top\_shift) / h;

}

if (i\_mixup == 3) {

xb = xb + (float)(cut\_x - left\_shift) / w;

yb = yb + (float)(cut\_y - top\_shift) / h;

}

int left = (xb - wb / 2)\*w;

int right = (xb + wb / 2)\*w;

int top = (yb - hb / 2)\*h;

int bot = (yb + hb / 2)\*h;

// fix out of bound

if (left < 0) {

float diff = (float)left / w;

xb = xb - diff / 2;

wb = wb + diff;

}

if (right > w) {

float diff = (float)(right - w) / w;

xb = xb - diff / 2;

wb = wb - diff;

}

if (top < 0) {

float diff = (float)top / h;

yb = yb - diff / 2;

hb = hb + diff;

}

if (bot > h) {

float diff = (float)(bot - h) / h;

yb = yb - diff / 2;

hb = hb - diff;

}

left = (xb - wb / 2)\*w;

right = (xb + wb / 2)\*w;

top = (yb - hb / 2)\*h;

bot = (yb + hb / 2)\*h;

// leave only within the image

if(left >= 0 && right <= w && top >= 0 && bot <= h &&

wb > 0 && wb < 1 && hb > 0 && hb < 1 &&

xb > 0 && xb < 1 && yb > 0 && yb < 1)

{

new\_truth\_ptr[0] = xb;

new\_truth\_ptr[1] = yb;

new\_truth\_ptr[2] = wb;

new\_truth\_ptr[3] = hb;

new\_truth\_ptr[4] = old\_truth\_ptr[4];

new\_t++;

}

}

//printf("\n was %d bboxes, now %d bboxes \n", count\_new\_truth, t);

}

#ifdef OPENCV

#include "http\_stream.h"

data load\_data\_detection(int n, char \*\*paths, int m, int w, int h, int c, int boxes, int classes, int use\_flip, int use\_gaussian\_noise, int use\_blur, int use\_mixup,

float jitter, float hue, float saturation, float exposure, int mini\_batch, int track, int augment\_speed, int letter\_box, int show\_imgs)

{

const int random\_index = random\_gen();

c = c ? c : 3;

if (use\_mixup == 2) {

printf("\n cutmix=1 - isn't supported for Detector \n");

exit(0);

}

if (use\_mixup == 3 && letter\_box) {

printf("\n Combination: letter\_box=1 & mosaic=1 - isn't supported, use only 1 of these parameters \n");

exit(0);

}

if (random\_gen() % 2 == 0) use\_mixup = 0;

int i;

int \*cut\_x = NULL, \*cut\_y = NULL;

if (use\_mixup == 3) {

cut\_x = (int\*)calloc(n, sizeof(int));

cut\_y = (int\*)calloc(n, sizeof(int));

const float min\_offset = 0.2; // 20%

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

cut\_x[i] = rand\_int(w\*min\_offset, w\*(1 - min\_offset));

cut\_y[i] = rand\_int(h\*min\_offset, h\*(1 - min\_offset));

}

}

data d = {0};

d.shallow = 0;

d.X.rows = n;

d.X.vals = (float\*\*)xcalloc(d.X.rows, sizeof(float\*));

d.X.cols = h\*w\*c;

float r1 = 0, r2 = 0, r3 = 0, r4 = 0, r\_scale = 0;

float dhue = 0, dsat = 0, dexp = 0, flip = 0, blur = 0;

int augmentation\_calculated = 0, gaussian\_noise = 0;

d.y = make\_matrix(n, 5\*boxes);

int i\_mixup = 0;

for (i\_mixup = 0; i\_mixup <= use\_mixup; i\_mixup++) {

if (i\_mixup) augmentation\_calculated = 0; // recalculate augmentation for the 2nd sequence if(track==1)

char \*\*random\_paths;

if (track) random\_paths = get\_sequential\_paths(paths, n, m, mini\_batch, augment\_speed);

else random\_paths = get\_random\_paths(paths, n, m);

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

float \*truth = (float\*)xcalloc(5 \* boxes, sizeof(float));

const char \*filename = random\_paths[i];

int flag = (c >= 3);

mat\_cv \*src;

src = load\_image\_mat\_cv(filename, flag);

if (src == NULL) {

if (check\_mistakes) {

printf("\n Error in load\_data\_detection() - OpenCV \n");

getchar();

}

continue;

}

int oh = get\_height\_mat(src);

int ow = get\_width\_mat(src);

int dw = (ow\*jitter);

int dh = (oh\*jitter);

if (!augmentation\_calculated || !track)

{

augmentation\_calculated = 1;

r1 = random\_float();

r2 = random\_float();

r3 = random\_float();

r4 = random\_float();

r\_scale = random\_float();

dhue = rand\_uniform\_strong(-hue, hue);

dsat = rand\_scale(saturation);

dexp = rand\_scale(exposure);

flip = use\_flip ? random\_gen() % 2 : 0;

if (use\_blur) {

int tmp\_blur = rand\_int(0, 2); // 0 - disable, 1 - blur background, 2 - blur the whole image

if (tmp\_blur == 0) blur = 0;

else if (tmp\_blur == 1) blur = 1;

else blur = use\_blur;

}

if (use\_gaussian\_noise && rand\_int(0, 1) == 1) gaussian\_noise = use\_gaussian\_noise;

else gaussian\_noise = 0;

}

int pleft = rand\_precalc\_random(-dw, dw, r1);

int pright = rand\_precalc\_random(-dw, dw, r2);

int ptop = rand\_precalc\_random(-dh, dh, r3);

int pbot = rand\_precalc\_random(-dh, dh, r4);

//printf("\n pleft = %d, pright = %d, ptop = %d, pbot = %d, ow = %d, oh = %d \n", pleft, pright, ptop, pbot, ow, oh);

//float scale = rand\_precalc\_random(.25, 2, r\_scale); // unused currently

if (letter\_box)

{

float img\_ar = (float)ow / (float)oh;

float net\_ar = (float)w / (float)h;

float result\_ar = img\_ar / net\_ar;

//printf(" ow = %d, oh = %d, w = %d, h = %d, img\_ar = %f, net\_ar = %f, result\_ar = %f \n", ow, oh, w, h, img\_ar, net\_ar, result\_ar);

if (result\_ar > 1) // sheight - should be increased

{

float oh\_tmp = ow / net\_ar;

float delta\_h = (oh\_tmp - oh)/2;

ptop = ptop - delta\_h;

pbot = pbot - delta\_h;

//printf(" result\_ar = %f, oh\_tmp = %f, delta\_h = %d, ptop = %f, pbot = %f \n", result\_ar, oh\_tmp, delta\_h, ptop, pbot);

}

else // swidth - should be increased

{

float ow\_tmp = oh \* net\_ar;

float delta\_w = (ow\_tmp - ow)/2;

pleft = pleft - delta\_w;

pright = pright - delta\_w;

//printf(" result\_ar = %f, ow\_tmp = %f, delta\_w = %d, pleft = %f, pright = %f \n", result\_ar, ow\_tmp, delta\_w, pleft, pright);

}

}

int swidth = ow - pleft - pright;

int sheight = oh - ptop - pbot;

float sx = (float)swidth / ow;

float sy = (float)sheight / oh;

float dx = ((float)pleft / ow) / sx;

float dy = ((float)ptop / oh) / sy;

int min\_w\_h = fill\_truth\_detection(filename, boxes, truth, classes, flip, dx, dy, 1. / sx, 1. / sy, w, h);

if ((min\_w\_h / 8) < blur && blur > 1) blur = min\_w\_h / 8; // disable blur if one of the objects is too small

image ai = image\_data\_augmentation(src, w, h, pleft, ptop, swidth, sheight, flip, dhue, dsat, dexp,

gaussian\_noise, blur, boxes, truth);

if (use\_mixup == 0) {

d.X.vals[i] = ai.data;

memcpy(d.y.vals[i], truth, 5 \* boxes \* sizeof(float));

}

else if (use\_mixup == 1) {

if (i\_mixup == 0) {

d.X.vals[i] = ai.data;

memcpy(d.y.vals[i], truth, 5 \* boxes \* sizeof(float));

}

else if (i\_mixup == 1) {

image old\_img = make\_empty\_image(w, h, c);

old\_img.data = d.X.vals[i];

//show\_image(ai, "new");

//show\_image(old\_img, "old");

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

blend\_images\_cv(ai, 0.5, old\_img, 0.5);

blend\_truth(d.y.vals[i], boxes, truth);

free\_image(old\_img);

d.X.vals[i] = ai.data;

}

}

else if (use\_mixup == 3) {

if (i\_mixup == 0) {

image tmp\_img = make\_image(w, h, c);

d.X.vals[i] = tmp\_img.data;

}

if (flip) {

int tmp = pleft;

pleft = pright;

pright = tmp;

}

const int left\_shift = min\_val\_cmp(cut\_x[i], max\_val\_cmp(0, (-pleft\*w / ow)));

const int top\_shift = min\_val\_cmp(cut\_y[i], max\_val\_cmp(0, (-ptop\*h / oh)));

const int right\_shift = min\_val\_cmp((w - cut\_x[i]), max\_val\_cmp(0, (-pright\*w / ow)));

const int bot\_shift = min\_val\_cmp(h - cut\_y[i], max\_val\_cmp(0, (-pbot\*h / oh)));

int k, x, y;

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

for (y = 0; y < h; ++y) {

int j = y\*w + k\*w\*h;

if (i\_mixup == 0 && y < cut\_y[i]) {

int j\_src = (w - cut\_x[i] - right\_shift) + (y + h - cut\_y[i] - bot\_shift)\*w + k\*w\*h;

memcpy(&d.X.vals[i][j + 0], &ai.data[j\_src], cut\_x[i] \* sizeof(float));

}

if (i\_mixup == 1 && y < cut\_y[i]) {

int j\_src = left\_shift + (y + h - cut\_y[i] - bot\_shift)\*w + k\*w\*h;

memcpy(&d.X.vals[i][j + cut\_x[i]], &ai.data[j\_src], (w-cut\_x[i]) \* sizeof(float));

}

if (i\_mixup == 2 && y >= cut\_y[i]) {

int j\_src = (w - cut\_x[i] - right\_shift) + (top\_shift + y - cut\_y[i])\*w + k\*w\*h;

memcpy(&d.X.vals[i][j + 0], &ai.data[j\_src], cut\_x[i] \* sizeof(float));

}

if (i\_mixup == 3 && y >= cut\_y[i]) {

int j\_src = left\_shift + (top\_shift + y - cut\_y[i])\*w + k\*w\*h;

memcpy(&d.X.vals[i][j + cut\_x[i]], &ai.data[j\_src], (w - cut\_x[i]) \* sizeof(float));

}

}

}

blend\_truth\_mosaic(d.y.vals[i], boxes, truth, w, h, cut\_x[i], cut\_y[i], i\_mixup, left\_shift, right\_shift, top\_shift, bot\_shift);

free\_image(ai);

ai.data = d.X.vals[i];

}

if (show\_imgs && i\_mixup == use\_mixup) // delete i\_mixup

{

image tmp\_ai = copy\_image(ai);

char buff[1000];

//sprintf(buff, "aug\_%d\_%d\_%s\_%d", random\_index, i, basecfg((char\*)filename), random\_gen());

sprintf(buff, "aug\_%d\_%d\_%d", random\_index, i, random\_gen());

int t;

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

box b = float\_to\_box\_stride(d.y.vals[i] + t\*(4 + 1), 1);

if (!b.x) break;

int left = (b.x - b.w / 2.)\*ai.w;

int right = (b.x + b.w / 2.)\*ai.w;

int top = (b.y - b.h / 2.)\*ai.h;

int bot = (b.y + b.h / 2.)\*ai.h;

draw\_box\_width(tmp\_ai, left, top, right, bot, 1, 150, 100, 50); // 3 channels RGB

}

save\_image(tmp\_ai, buff);

if (show\_imgs == 1) {

//char buff\_src[1000];

//sprintf(buff\_src, "src\_%d\_%d\_%s\_%d", random\_index, i, basecfg((char\*)filename), random\_gen());

//show\_image\_mat(src, buff\_src);

show\_image(tmp\_ai, buff);

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

}

printf("\nYou use flag -show\_imgs, so will be saved aug\_...jpg images. Click on window and press ESC button \n");

free\_image(tmp\_ai);

}

release\_mat(&src);

free(truth);

}

if (random\_paths) free(random\_paths);

}

return d;

}

#else // OPENCV

void blend\_images(image new\_img, float alpha, image old\_img, float beta)

{

int data\_size = new\_img.w \* new\_img.h \* new\_img.c;

int i;

#pragma omp parallel for

for (i = 0; i < data\_size; ++i)

new\_img.data[i] = new\_img.data[i] \* alpha + old\_img.data[i] \* beta;

}

data load\_data\_detection(int n, char \*\*paths, int m, int w, int h, int c, int boxes, int classes, int use\_flip, int gaussian\_noise, int use\_blur, int use\_mixup, float jitter,

float hue, float saturation, float exposure, int mini\_batch, int track, int augment\_speed, int letter\_box, int show\_imgs)

{

const int random\_index = random\_gen();

c = c ? c : 3;

char \*\*random\_paths;

char \*\*mixup\_random\_paths = NULL;

if(track) random\_paths = get\_sequential\_paths(paths, n, m, mini\_batch, augment\_speed);

else random\_paths = get\_random\_paths(paths, n, m);

//assert(use\_mixup < 2);

if (use\_mixup == 2) {

printf("\n cutmix=1 - isn't supported for Detector \n");

exit(0);

}

if (use\_mixup == 3) {

printf("\n mosaic=1 - compile Darknet with OpenCV for using mosaic=1 \n");

exit(0);

}

int mixup = use\_mixup ? random\_gen() % 2 : 0;

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

if (mixup) {

if (track) mixup\_random\_paths = get\_sequential\_paths(paths, n, m, mini\_batch, augment\_speed);

else mixup\_random\_paths = get\_random\_paths(paths, n, m);

}

int i;

data d = { 0 };

d.shallow = 0;

d.X.rows = n;

d.X.vals = (float\*\*)xcalloc(d.X.rows, sizeof(float\*));

d.X.cols = h\*w\*c;

float r1 = 0, r2 = 0, r3 = 0, r4 = 0, r\_scale;

float dhue = 0, dsat = 0, dexp = 0, flip = 0;

int augmentation\_calculated = 0;

d.y = make\_matrix(n, 5 \* boxes);

int i\_mixup = 0;

for (i\_mixup = 0; i\_mixup <= mixup; i\_mixup++) {

if (i\_mixup) augmentation\_calculated = 0;

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

float \*truth = (float\*)xcalloc(5 \* boxes, sizeof(float));

char \*filename = (i\_mixup) ? mixup\_random\_paths[i] : random\_paths[i];

image orig = load\_image(filename, 0, 0, c);

int oh = orig.h;

int ow = orig.w;

int dw = (ow\*jitter);

int dh = (oh\*jitter);

if (!augmentation\_calculated || !track)

{

augmentation\_calculated = 1;

r1 = random\_float();

r2 = random\_float();

r3 = random\_float();

r4 = random\_float();

r\_scale = random\_float();

dhue = rand\_uniform\_strong(-hue, hue);

dsat = rand\_scale(saturation);

dexp = rand\_scale(exposure);

flip = use\_flip ? random\_gen() % 2 : 0;

}

int pleft = rand\_precalc\_random(-dw, dw, r1);

int pright = rand\_precalc\_random(-dw, dw, r2);

int ptop = rand\_precalc\_random(-dh, dh, r3);

int pbot = rand\_precalc\_random(-dh, dh, r4);

float scale = rand\_precalc\_random(.25, 2, r\_scale); // unused currently

if (letter\_box)

{

float img\_ar = (float)ow / (float)oh;

float net\_ar = (float)w / (float)h;

float result\_ar = img\_ar / net\_ar;

//printf(" ow = %d, oh = %d, w = %d, h = %d, img\_ar = %f, net\_ar = %f, result\_ar = %f \n", ow, oh, w, h, img\_ar, net\_ar, result\_ar);

if (result\_ar > 1) // sheight - should be increased

{

float oh\_tmp = ow / net\_ar;

float delta\_h = (oh\_tmp - oh) / 2;

ptop = ptop - delta\_h;

pbot = pbot - delta\_h;

//printf(" result\_ar = %f, oh\_tmp = %f, delta\_h = %d, ptop = %f, pbot = %f \n", result\_ar, oh\_tmp, delta\_h, ptop, pbot);

}

else // swidth - should be increased

{

float ow\_tmp = oh \* net\_ar;

float delta\_w = (ow\_tmp - ow) / 2;

pleft = pleft - delta\_w;

pright = pright - delta\_w;

//printf(" result\_ar = %f, ow\_tmp = %f, delta\_w = %d, pleft = %f, pright = %f \n", result\_ar, ow\_tmp, delta\_w, pleft, pright);

}

}

int swidth = ow - pleft - pright;

int sheight = oh - ptop - pbot;

float sx = (float)swidth / ow;

float sy = (float)sheight / oh;

image cropped = crop\_image(orig, pleft, ptop, swidth, sheight);

float dx = ((float)pleft / ow) / sx;

float dy = ((float)ptop / oh) / sy;

image sized = resize\_image(cropped, w, h);

if (flip) flip\_image(sized);

distort\_image(sized, dhue, dsat, dexp);

//random\_distort\_image(sized, hue, saturation, exposure);

fill\_truth\_detection(filename, boxes, truth, classes, flip, dx, dy, 1. / sx, 1. / sy, w, h);

if (i\_mixup) {

image old\_img = sized;

old\_img.data = d.X.vals[i];

//show\_image(sized, "new");

//show\_image(old\_img, "old");

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

blend\_images(sized, 0.5, old\_img, 0.5);

blend\_truth(truth, boxes, d.y.vals[i]);

free\_image(old\_img);

}

d.X.vals[i] = sized.data;

memcpy(d.y.vals[i], truth, 5 \* boxes \* sizeof(float));

if (show\_imgs)// && i\_mixup)

{

char buff[1000];

sprintf(buff, "aug\_%d\_%d\_%s\_%d", random\_index, i, basecfg(filename), random\_gen());

int t;

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

box b = float\_to\_box\_stride(d.y.vals[i] + t\*(4 + 1), 1);

if (!b.x) break;

int left = (b.x - b.w / 2.)\*sized.w;

int right = (b.x + b.w / 2.)\*sized.w;

int top = (b.y - b.h / 2.)\*sized.h;

int bot = (b.y + b.h / 2.)\*sized.h;

draw\_box\_width(sized, left, top, right, bot, 1, 150, 100, 50); // 3 channels RGB

}

save\_image(sized, buff);

if (show\_imgs == 1) {

show\_image(sized, buff);

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

}

printf("\nYou use flag -show\_imgs, so will be saved aug\_...jpg images. Press Enter: \n");

//getchar();

}

free\_image(orig);

free\_image(cropped);

free(truth);

}

}

free(random\_paths);

if (mixup\_random\_paths) free(mixup\_random\_paths);

return d;

}

#endif // OPENCV

void \*load\_thread(void \*ptr)

{

//srand(time(0));

//printf("Loading data: %d\n", random\_gen());

load\_args a = \*(struct load\_args\*)ptr;

if(a.exposure == 0) a.exposure = 1;

if(a.saturation == 0) a.saturation = 1;

if(a.aspect == 0) a.aspect = 1;

if (a.type == OLD\_CLASSIFICATION\_DATA){

\*a.d = load\_data\_old(a.paths, a.n, a.m, a.labels, a.classes, a.w, a.h);

} else if (a.type == CLASSIFICATION\_DATA){

\*a.d = load\_data\_augment(a.paths, a.n, a.m, a.labels, a.classes, a.hierarchy, a.flip, a.min, a.max, a.w, a.h, a.angle, a.aspect, a.hue, a.saturation, a.exposure, a.mixup, a.blur, a.show\_imgs, a.label\_smooth\_eps, a.dontuse\_opencv);

} else if (a.type == SUPER\_DATA){

\*a.d = load\_data\_super(a.paths, a.n, a.m, a.w, a.h, a.scale);

} else if (a.type == WRITING\_DATA){

\*a.d = load\_data\_writing(a.paths, a.n, a.m, a.w, a.h, a.out\_w, a.out\_h);

} else if (a.type == REGION\_DATA){

\*a.d = load\_data\_region(a.n, a.paths, a.m, a.w, a.h, a.num\_boxes, a.classes, a.jitter, a.hue, a.saturation, a.exposure);

} else if (a.type == DETECTION\_DATA){

\*a.d = load\_data\_detection(a.n, a.paths, a.m, a.w, a.h, a.c, a.num\_boxes, a.classes, a.flip, a.gaussian\_noise, a.blur, a.mixup, a.jitter,

a.hue, a.saturation, a.exposure, a.mini\_batch, a.track, a.augment\_speed, a.letter\_box, a.show\_imgs);

} else if (a.type == SWAG\_DATA){

\*a.d = load\_data\_swag(a.paths, a.n, a.classes, a.jitter);

} else if (a.type == COMPARE\_DATA){

\*a.d = load\_data\_compare(a.n, a.paths, a.m, a.classes, a.w, a.h);

} else if (a.type == IMAGE\_DATA){

\*(a.im) = load\_image(a.path, 0, 0, a.c);

\*(a.resized) = resize\_image(\*(a.im), a.w, a.h);

}else if (a.type == LETTERBOX\_DATA) {

\*(a.im) = load\_image(a.path, 0, 0, a.c);

\*(a.resized) = letterbox\_image(\*(a.im), a.w, a.h);

} else if (a.type == TAG\_DATA){

\*a.d = load\_data\_tag(a.paths, a.n, a.m, a.classes, a.flip, a.min, a.max, a.w, a.h, a.angle, a.aspect, a.hue, a.saturation, a.exposure);

}

free(ptr);

return 0;

}

pthread\_t load\_data\_in\_thread(load\_args args)

{

pthread\_t thread;

struct load\_args\* ptr = (load\_args\*)xcalloc(1, sizeof(struct load\_args));

\*ptr = args;

if(pthread\_create(&thread, 0, load\_thread, ptr)) error("Thread creation failed");

return thread;

}

static const int thread\_wait\_ms = 5;

static volatile int flag\_exit;

static volatile int \* run\_load\_data = NULL;

static load\_args \* args\_swap = NULL;

static pthread\_t\* threads = NULL;

pthread\_mutex\_t mtx\_load\_data = PTHREAD\_MUTEX\_INITIALIZER;

void \*run\_thread\_loop(void \*ptr)

{

const int i = \*(int \*)ptr;

while (!custom\_atomic\_load\_int(&flag\_exit)) {

while (!custom\_atomic\_load\_int(&run\_load\_data[i])) {

if (custom\_atomic\_load\_int(&flag\_exit)) {

free(ptr);

return 0;

}

this\_thread\_sleep\_for(thread\_wait\_ms);

}

pthread\_mutex\_lock(&mtx\_load\_data);

load\_args \*args\_local = (load\_args \*)xcalloc(1, sizeof(load\_args));

\*args\_local = args\_swap[i];

pthread\_mutex\_unlock(&mtx\_load\_data);

load\_thread(args\_local);

custom\_atomic\_store\_int(&run\_load\_data[i], 0);

}

free(ptr);

return 0;

}

void \*load\_threads(void \*ptr)

{

//srand(time(0));

int i;

load\_args args = \*(load\_args \*)ptr;

if (args.threads == 0) args.threads = 1;

data \*out = args.d;

int total = args.n;

free(ptr);

data\* buffers = (data\*)xcalloc(args.threads, sizeof(data));

if (!threads) {

threads = (pthread\_t\*)xcalloc(args.threads, sizeof(pthread\_t));

run\_load\_data = (volatile int \*)xcalloc(args.threads, sizeof(int));

args\_swap = (load\_args \*)xcalloc(args.threads, sizeof(load\_args));

fprintf(stderr, " Create %d permanent cpu-threads \n", args.threads);

for (i = 0; i < args.threads; ++i) {

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

\*ptr = i;

if (pthread\_create(&threads[i], 0, run\_thread\_loop, ptr)) error("Thread creation failed");

}

}

for (i = 0; i < args.threads; ++i) {

args.d = buffers + i;

args.n = (i + 1) \* total / args.threads - i \* total / args.threads;

pthread\_mutex\_lock(&mtx\_load\_data);

args\_swap[i] = args;

pthread\_mutex\_unlock(&mtx\_load\_data);

custom\_atomic\_store\_int(&run\_load\_data[i], 1); // run thread

}

for (i = 0; i < args.threads; ++i) {

while (custom\_atomic\_load\_int(&run\_load\_data[i])) this\_thread\_sleep\_for(thread\_wait\_ms); // join

}

/\*

pthread\_t\* threads = (pthread\_t\*)xcalloc(args.threads, sizeof(pthread\_t));

for(i = 0; i < args.threads; ++i){

args.d = buffers + i;

args.n = (i+1) \* total/args.threads - i \* total/args.threads;

threads[i] = load\_data\_in\_thread(args);

}

for(i = 0; i < args.threads; ++i){

pthread\_join(threads[i], 0);

}

\*/

\*out = concat\_datas(buffers, args.threads);

out->shallow = 0;

for(i = 0; i < args.threads; ++i){

buffers[i].shallow = 1;

free\_data(buffers[i]);

}

free(buffers);

//free(threads);

return 0;

}

void free\_load\_threads(void \*ptr)

{

load\_args args = \*(load\_args \*)ptr;

if (args.threads == 0) args.threads = 1;

int i;

if (threads) {

custom\_atomic\_store\_int(&flag\_exit, 1);

for (i = 0; i < args.threads; ++i) {

pthread\_join(threads[i], 0);

}

free((void\*)run\_load\_data);

free(args\_swap);

free(threads);

threads = NULL;

custom\_atomic\_store\_int(&flag\_exit, 0);

}

}

pthread\_t load\_data(load\_args args)

{

pthread\_t thread;

struct load\_args\* ptr = (load\_args\*)xcalloc(1, sizeof(struct load\_args));

\*ptr = args;

if(pthread\_create(&thread, 0, load\_threads, ptr)) error("Thread creation failed");

return thread;

}

data load\_data\_writing(char \*\*paths, int n, int m, int w, int h, int out\_w, int out\_h)

{

if(m) paths = get\_random\_paths(paths, n, m);

char \*\*replace\_paths = find\_replace\_paths(paths, n, ".png", "-label.png");

data d = {0};

d.shallow = 0;

d.X = load\_image\_paths(paths, n, w, h);

d.y = load\_image\_paths\_gray(replace\_paths, n, out\_w, out\_h);

if(m) free(paths);

int i;

for(i = 0; i < n; ++i) free(replace\_paths[i]);

free(replace\_paths);

return d;

}

data load\_data\_old(char \*\*paths, int n, int m, char \*\*labels, int k, int w, int h)

{

if(m) paths = get\_random\_paths(paths, n, m);

data d = {0};

d.shallow = 0;

d.X = load\_image\_paths(paths, n, w, h);

d.y = load\_labels\_paths(paths, n, labels, k, 0, 0);

if(m) free(paths);

return d;

}

/\*

data load\_data\_study(char \*\*paths, int n, int m, char \*\*labels, int k, int min, int max, int size, float angle, float aspect, float hue, float saturation, float exposure)

{

data d = {0};

d.indexes = calloc(n, sizeof(int));

if(m) paths = get\_random\_paths\_indexes(paths, n, m, d.indexes);

d.shallow = 0;

d.X = load\_image\_augment\_paths(paths, n, flip, min, max, size, angle, aspect, hue, saturation, exposure);

d.y = load\_labels\_paths(paths, n, labels, k);

if(m) free(paths);

return d;

}

\*/

data load\_data\_super(char \*\*paths, int n, int m, int w, int h, int scale)

{

if(m) paths = get\_random\_paths(paths, n, m);

data d = {0};

d.shallow = 0;

int i;

d.X.rows = n;

d.X.vals = (float\*\*)xcalloc(n, sizeof(float\*));

d.X.cols = w\*h\*3;

d.y.rows = n;

d.y.vals = (float\*\*)xcalloc(n, sizeof(float\*));

d.y.cols = w\*scale \* h\*scale \* 3;

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

image im = load\_image\_color(paths[i], 0, 0);

image crop = random\_crop\_image(im, w\*scale, h\*scale);

int flip = random\_gen()%2;

if (flip) flip\_image(crop);

image resize = resize\_image(crop, w, h);

d.X.vals[i] = resize.data;

d.y.vals[i] = crop.data;

free\_image(im);

}

if(m) free(paths);

return d;

}

data load\_data\_augment(char \*\*paths, int n, int m, char \*\*labels, int k, tree \*hierarchy, int use\_flip, int min, int max, int w, int h, float angle, float aspect, float hue, float saturation, float exposure, int use\_mixup, int use\_blur, int show\_imgs, float label\_smooth\_eps, int dontuse\_opencv)

{

char \*\*paths\_stored = paths;

if(m) paths = get\_random\_paths(paths, n, m);

data d = {0};

d.shallow = 0;

d.X = load\_image\_augment\_paths(paths, n, use\_flip, min, max, w, h, angle, aspect, hue, saturation, exposure, dontuse\_opencv);

d.y = load\_labels\_paths(paths, n, labels, k, hierarchy, label\_smooth\_eps);

if (use\_mixup && rand\_int(0, 1)) {

char \*\*paths\_mix = get\_random\_paths(paths\_stored, n, m);

data d2 = { 0 };

d2.shallow = 0;

d2.X = load\_image\_augment\_paths(paths\_mix, n, use\_flip, min, max, w, h, angle, aspect, hue, saturation, exposure, dontuse\_opencv);

d2.y = load\_labels\_paths(paths\_mix, n, labels, k, hierarchy, label\_smooth\_eps);

free(paths\_mix);

data d3 = { 0 };

d3.shallow = 0;

data d4 = { 0 };

d4.shallow = 0;

if (use\_mixup >= 3) {

char \*\*paths\_mix3 = get\_random\_paths(paths\_stored, n, m);

d3.X = load\_image\_augment\_paths(paths\_mix3, n, use\_flip, min, max, w, h, angle, aspect, hue, saturation, exposure, dontuse\_opencv);

d3.y = load\_labels\_paths(paths\_mix3, n, labels, k, hierarchy, label\_smooth\_eps);

free(paths\_mix3);

char \*\*paths\_mix4 = get\_random\_paths(paths\_stored, n, m);

d4.X = load\_image\_augment\_paths(paths\_mix4, n, use\_flip, min, max, w, h, angle, aspect, hue, saturation, exposure, dontuse\_opencv);

d4.y = load\_labels\_paths(paths\_mix4, n, labels, k, hierarchy, label\_smooth\_eps);

free(paths\_mix4);

}

// mix

int i, j;

for (i = 0; i < d2.X.rows; ++i) {

int mixup = use\_mixup;

if (use\_mixup == 4) mixup = rand\_int(2, 3); // alternate CutMix and Mosaic

// MixUp -----------------------------------

if (mixup == 1) {

// mix images

for (j = 0; j < d2.X.cols; ++j) {

d.X.vals[i][j] = (d.X.vals[i][j] + d2.X.vals[i][j]) / 2.0f;

}

// mix labels

for (j = 0; j < d2.y.cols; ++j) {

d.y.vals[i][j] = (d.y.vals[i][j] + d2.y.vals[i][j]) / 2.0f;

}

}

// CutMix -----------------------------------

else if (mixup == 2) {

const float min = 0.3; // 0.3\*0.3 = 9%

const float max = 0.8; // 0.8\*0.8 = 64%

const int cut\_w = rand\_int(w\*min, w\*max);

const int cut\_h = rand\_int(h\*min, h\*max);

const int cut\_x = rand\_int(0, w - cut\_w - 1);

const int cut\_y = rand\_int(0, h - cut\_h - 1);

const int left = cut\_x;

const int right = cut\_x + cut\_w;

const int top = cut\_y;

const int bot = cut\_y + cut\_h;

assert(cut\_x >= 0 && cut\_x <= w);

assert(cut\_y >= 0 && cut\_y <= h);

assert(cut\_w >= 0 && cut\_w <= w);

assert(cut\_h >= 0 && cut\_h <= h);

assert(right >= 0 && right <= w);

assert(bot >= 0 && bot <= h);

assert(top <= bot);

assert(left <= right);

const float alpha = (float)(cut\_w\*cut\_h) / (float)(w\*h);

const float beta = 1 - alpha;

int c, x, y;

for (c = 0; c < 3; ++c) {

for (y = top; y < bot; ++y) {

for (x = left; x < right; ++x) {

int j = x + y\*w + c\*w\*h;

d.X.vals[i][j] = d2.X.vals[i][j];

}

}

}

//printf("\n alpha = %f, beta = %f \n", alpha, beta);

// mix labels

for (j = 0; j < d.y.cols; ++j) {

d.y.vals[i][j] = d.y.vals[i][j] \* beta + d2.y.vals[i][j] \* alpha;

}

}

// Mosaic -----------------------------------

else if (mixup == 3)

{

const float min\_offset = 0.2; // 20%

const int cut\_x = rand\_int(w\*min\_offset, w\*(1 - min\_offset));

const int cut\_y = rand\_int(h\*min\_offset, h\*(1 - min\_offset));

float s1 = (float)(cut\_x \* cut\_y) / (w\*h);

float s2 = (float)((w - cut\_x) \* cut\_y) / (w\*h);

float s3 = (float)(cut\_x \* (h - cut\_y)) / (w\*h);

float s4 = (float)((w - cut\_x) \* (h - cut\_y)) / (w\*h);

int c, x, y;

for (c = 0; c < 3; ++c) {

for (y = 0; y < h; ++y) {

for (x = 0; x < w; ++x) {

int j = x + y\*w + c\*w\*h;

if (x < cut\_x && y < cut\_y) d.X.vals[i][j] = d.X.vals[i][j];

if (x >= cut\_x && y < cut\_y) d.X.vals[i][j] = d2.X.vals[i][j];

if (x < cut\_x && y >= cut\_y) d.X.vals[i][j] = d3.X.vals[i][j];

if (x >= cut\_x && y >= cut\_y) d.X.vals[i][j] = d4.X.vals[i][j];

}

}

}

for (j = 0; j < d.y.cols; ++j) {

const float max\_s = 1;// max\_val\_cmp(s1, max\_val\_cmp(s2, max\_val\_cmp(s3, s4)));

d.y.vals[i][j] = d.y.vals[i][j] \* s1 / max\_s + d2.y.vals[i][j] \* s2 / max\_s + d3.y.vals[i][j] \* s3 / max\_s + d4.y.vals[i][j] \* s4 / max\_s;

}

}

}

free\_data(d2);

if (use\_mixup >= 3) {

free\_data(d3);

free\_data(d4);

}

}

#ifdef OPENCV

if (use\_blur) {

int i;

for (i = 0; i < d.X.rows; ++i) {

if (random\_gen() % 2) {

image im = make\_empty\_image(w, h, 3);

im.data = d.X.vals[i];

int ksize = use\_blur;

if (use\_blur == 1) ksize = 17;

image blurred = blur\_image(im, ksize);

free\_image(im);

d.X.vals[i] = blurred.data;

//if (i == 0) {

// show\_image(im, "Not blurred");

// show\_image(blurred, "blurred");

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

//}

}

}

}

#endif // OPENCV

if (show\_imgs) {

int i, j;

for (i = 0; i < d.X.rows; ++i) {

image im = make\_empty\_image(w, h, 3);

im.data = d.X.vals[i];

char buff[1000];

sprintf(buff, "aug\_%d\_%s\_%d", i, basecfg((char\*)paths[i]), random\_gen());

save\_image(im, buff);

char buff\_string[1000];

sprintf(buff\_string, "\n Classes: ");

for (j = 0; j < d.y.cols; ++j) {

if (d.y.vals[i][j] > 0) {

char buff\_tmp[100];

sprintf(buff\_tmp, " %d (%f), ", j, d.y.vals[i][j]);

strcat(buff\_string, buff\_tmp);

}

}

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

if (show\_imgs == 1) {

show\_image(im, buff);

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

}

}

printf("\nYou use flag -show\_imgs, so will be saved aug\_...jpg images. Click on window and press ESC button \n");

}

if (m) free(paths);

return d;

}

data load\_data\_tag(char \*\*paths, int n, int m, int k, int use\_flip, int min, int max, int w, int h, float angle, float aspect, float hue, float saturation, float exposure)

{

if(m) paths = get\_random\_paths(paths, n, m);

data d = {0};

d.w = w;

d.h = h;

d.shallow = 0;

d.X = load\_image\_augment\_paths(paths, n, use\_flip, min, max, w, h, angle, aspect, hue, saturation, exposure, 0);

d.y = load\_tags\_paths(paths, n, k);

if(m) free(paths);

return d;

}

matrix concat\_matrix(matrix m1, matrix m2)

{

int i, count = 0;

matrix m;

m.cols = m1.cols;

m.rows = m1.rows+m2.rows;

m.vals = (float\*\*)xcalloc(m1.rows + m2.rows, sizeof(float\*));

for(i = 0; i < m1.rows; ++i){

m.vals[count++] = m1.vals[i];

}

for(i = 0; i < m2.rows; ++i){

m.vals[count++] = m2.vals[i];

}

return m;

}

data concat\_data(data d1, data d2)

{

data d = {0};

d.shallow = 1;

d.X = concat\_matrix(d1.X, d2.X);

d.y = concat\_matrix(d1.y, d2.y);

return d;

}

data concat\_datas(data \*d, int n)

{

int i;

data out = {0};

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

data newdata = concat\_data(d[i], out);

free\_data(out);

out = newdata;

}

return out;

}

data load\_categorical\_data\_csv(char \*filename, int target, int k)

{

data d = {0};

d.shallow = 0;

matrix X = csv\_to\_matrix(filename);

float \*truth\_1d = pop\_column(&X, target);

float \*\*truth = one\_hot\_encode(truth\_1d, X.rows, k);

matrix y;

y.rows = X.rows;

y.cols = k;

y.vals = truth;

d.X = X;

d.y = y;

free(truth\_1d);

return d;

}

data load\_cifar10\_data(char \*filename)

{

data d = {0};

d.shallow = 0;

long i,j;

matrix X = make\_matrix(10000, 3072);

matrix y = make\_matrix(10000, 10);

d.X = X;

d.y = y;

FILE \*fp = fopen(filename, "rb");

if(!fp) file\_error(filename);

for(i = 0; i < 10000; ++i){

unsigned char bytes[3073];

fread(bytes, 1, 3073, fp);

int class\_id = bytes[0];

y.vals[i][class\_id] = 1;

for(j = 0; j < X.cols; ++j){

X.vals[i][j] = (double)bytes[j+1];

}

}

//translate\_data\_rows(d, -128);

scale\_data\_rows(d, 1./255);

//normalize\_data\_rows(d);

fclose(fp);

return d;

}

void get\_random\_batch(data d, int n, float \*X, float \*y)

{

int j;

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

int index = random\_gen()%d.X.rows;

memcpy(X+j\*d.X.cols, d.X.vals[index], d.X.cols\*sizeof(float));

memcpy(y+j\*d.y.cols, d.y.vals[index], d.y.cols\*sizeof(float));

}

}

void get\_next\_batch(data d, int n, int offset, float \*X, float \*y)

{

int j;

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

int index = offset + j;

memcpy(X+j\*d.X.cols, d.X.vals[index], d.X.cols\*sizeof(float));

memcpy(y+j\*d.y.cols, d.y.vals[index], d.y.cols\*sizeof(float));

}

}

void smooth\_data(data d)

{

int i, j;

float scale = 1. / d.y.cols;

float eps = .1;

for(i = 0; i < d.y.rows; ++i){

for(j = 0; j < d.y.cols; ++j){

d.y.vals[i][j] = eps \* scale + (1-eps) \* d.y.vals[i][j];

}

}

}

data load\_all\_cifar10()

{

data d = {0};

d.shallow = 0;

int i,j,b;

matrix X = make\_matrix(50000, 3072);

matrix y = make\_matrix(50000, 10);

d.X = X;

d.y = y;

for(b = 0; b < 5; ++b){

char buff[256];

sprintf(buff, "data/cifar/cifar-10-batches-bin/data\_batch\_%d.bin", b+1);

FILE \*fp = fopen(buff, "rb");

if(!fp) file\_error(buff);

for(i = 0; i < 10000; ++i){

unsigned char bytes[3073];

fread(bytes, 1, 3073, fp);

int class\_id = bytes[0];

y.vals[i+b\*10000][class\_id] = 1;

for(j = 0; j < X.cols; ++j){

X.vals[i+b\*10000][j] = (double)bytes[j+1];

}

}

fclose(fp);

}

//normalize\_data\_rows(d);

//translate\_data\_rows(d, -128);

scale\_data\_rows(d, 1./255);

smooth\_data(d);

return d;

}

data load\_go(char \*filename)

{

FILE \*fp = fopen(filename, "rb");

matrix X = make\_matrix(3363059, 361);

matrix y = make\_matrix(3363059, 361);

int row, col;

if(!fp) file\_error(filename);

char \*label;

int count = 0;

while((label = fgetl(fp))){

int i;

if(count == X.rows){

X = resize\_matrix(X, count\*2);

y = resize\_matrix(y, count\*2);

}

sscanf(label, "%d %d", &row, &col);

char \*board = fgetl(fp);

int index = row\*19 + col;

y.vals[count][index] = 1;

for(i = 0; i < 19\*19; ++i){

float val = 0;

if(board[i] == '1') val = 1;

else if(board[i] == '2') val = -1;

X.vals[count][i] = val;

}

++count;

free(label);

free(board);

}

X = resize\_matrix(X, count);

y = resize\_matrix(y, count);

data d = {0};

d.shallow = 0;

d.X = X;

d.y = y;

fclose(fp);

return d;

}

void randomize\_data(data d)

{

int i;

for(i = d.X.rows-1; i > 0; --i){

int index = random\_gen()%i;

float \*swap = d.X.vals[index];

d.X.vals[index] = d.X.vals[i];

d.X.vals[i] = swap;

swap = d.y.vals[index];

d.y.vals[index] = d.y.vals[i];

d.y.vals[i] = swap;

}

}

void scale\_data\_rows(data d, float s)

{

int i;

for(i = 0; i < d.X.rows; ++i){

scale\_array(d.X.vals[i], d.X.cols, s);

}

}

void translate\_data\_rows(data d, float s)

{

int i;

for(i = 0; i < d.X.rows; ++i){

translate\_array(d.X.vals[i], d.X.cols, s);

}

}

void normalize\_data\_rows(data d)

{

int i;

for(i = 0; i < d.X.rows; ++i){

normalize\_array(d.X.vals[i], d.X.cols);

}

}

data get\_data\_part(data d, int part, int total)

{

data p = {0};

p.shallow = 1;

p.X.rows = d.X.rows \* (part + 1) / total - d.X.rows \* part / total;

p.y.rows = d.y.rows \* (part + 1) / total - d.y.rows \* part / total;

p.X.cols = d.X.cols;

p.y.cols = d.y.cols;

p.X.vals = d.X.vals + d.X.rows \* part / total;

p.y.vals = d.y.vals + d.y.rows \* part / total;

return p;

}

data get\_random\_data(data d, int num)

{

data r = {0};

r.shallow = 1;

r.X.rows = num;

r.y.rows = num;

r.X.cols = d.X.cols;

r.y.cols = d.y.cols;

r.X.vals = (float\*\*)xcalloc(num, sizeof(float\*));

r.y.vals = (float\*\*)xcalloc(num, sizeof(float\*));

int i;

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

int index = random\_gen()%d.X.rows;

r.X.vals[i] = d.X.vals[index];

r.y.vals[i] = d.y.vals[index];

}

return r;

}

data \*split\_data(data d, int part, int total)

{

data\* split = (data\*)xcalloc(2, sizeof(data));

int i;

int start = part\*d.X.rows/total;

int end = (part+1)\*d.X.rows/total;

data train ={0};

data test ={0};

train.shallow = test.shallow = 1;

test.X.rows = test.y.rows = end-start;

train.X.rows = train.y.rows = d.X.rows - (end-start);

train.X.cols = test.X.cols = d.X.cols;

train.y.cols = test.y.cols = d.y.cols;

train.X.vals = (float\*\*)xcalloc(train.X.rows, sizeof(float\*));

test.X.vals = (float\*\*)xcalloc(test.X.rows, sizeof(float\*));

train.y.vals = (float\*\*)xcalloc(train.y.rows, sizeof(float\*));

test.y.vals = (float\*\*)xcalloc(test.y.rows, sizeof(float\*));

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

train.X.vals[i] = d.X.vals[i];

train.y.vals[i] = d.y.vals[i];

}

for(i = start; i < end; ++i){

test.X.vals[i-start] = d.X.vals[i];

test.y.vals[i-start] = d.y.vals[i];

}

for(i = end; i < d.X.rows; ++i){

train.X.vals[i-(end-start)] = d.X.vals[i];

train.y.vals[i-(end-start)] = d.y.vals[i];

}

split[0] = train;

split[1] = test;

return split;

}