#ifndef \_GNU\_SOURCE

#define \_GNU\_SOURCE

#endif

#include "image.h"

#include "utils.h"

#include "blas.h"

#include "dark\_cuda.h"

#include <stdio.h>

#ifndef \_USE\_MATH\_DEFINES

#define \_USE\_MATH\_DEFINES

#endif

#include <math.h>

#ifndef STB\_IMAGE\_IMPLEMENTATION

#define STB\_IMAGE\_IMPLEMENTATION

#include "stb\_image.h"

#endif

#ifndef STB\_IMAGE\_WRITE\_IMPLEMENTATION

#define STB\_IMAGE\_WRITE\_IMPLEMENTATION

#include "stb\_image\_write.h"

#endif

extern int check\_mistakes;

//int windows = 0;

float colors[6][3] = { {1,0,1}, {0,0,1},{0,1,1},{0,1,0},{1,1,0},{1,0,0} };

float get\_color(int c, int x, int max)

{

float ratio = ((float)x/max)\*5;

int i = floor(ratio);

int j = ceil(ratio);

ratio -= i;

float r = (1-ratio) \* colors[i][c] + ratio\*colors[j][c];

//printf("%f\n", r);

return r;

}

static float get\_pixel(image m, int x, int y, int c)

{

assert(x < m.w && y < m.h && c < m.c);

return m.data[c\*m.h\*m.w + y\*m.w + x];

}

static float get\_pixel\_extend(image m, int x, int y, int c)

{

if (x < 0 || x >= m.w || y < 0 || y >= m.h) return 0;

/\*

if(x < 0) x = 0;

if(x >= m.w) x = m.w-1;

if(y < 0) y = 0;

if(y >= m.h) y = m.h-1;

\*/

if (c < 0 || c >= m.c) return 0;

return get\_pixel(m, x, y, c);

}

static void set\_pixel(image m, int x, int y, int c, float val)

{

if (x < 0 || y < 0 || c < 0 || x >= m.w || y >= m.h || c >= m.c) return;

assert(x < m.w && y < m.h && c < m.c);

m.data[c\*m.h\*m.w + y\*m.w + x] = val;

}

static void add\_pixel(image m, int x, int y, int c, float val)

{

assert(x < m.w && y < m.h && c < m.c);

m.data[c\*m.h\*m.w + y\*m.w + x] += val;

}

void composite\_image(image source, image dest, int dx, int dy)

{

int x,y,k;

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

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

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

float val = get\_pixel(source, x, y, k);

float val2 = get\_pixel\_extend(dest, dx+x, dy+y, k);

set\_pixel(dest, dx+x, dy+y, k, val \* val2);

}

}

}

}

image border\_image(image a, int border)

{

image b = make\_image(a.w + 2\*border, a.h + 2\*border, a.c);

int x,y,k;

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

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

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

float val = get\_pixel\_extend(a, x - border, y - border, k);

if(x - border < 0 || x - border >= a.w || y - border < 0 || y - border >= a.h) val = 1;

set\_pixel(b, x, y, k, val);

}

}

}

return b;

}

image tile\_images(image a, image b, int dx)

{

if(a.w == 0) return copy\_image(b);

image c = make\_image(a.w + b.w + dx, (a.h > b.h) ? a.h : b.h, (a.c > b.c) ? a.c : b.c);

fill\_cpu(c.w\*c.h\*c.c, 1, c.data, 1);

embed\_image(a, c, 0, 0);

composite\_image(b, c, a.w + dx, 0);

return c;

}

image get\_label(image \*\*characters, char \*string, int size)

{

if(size > 7) size = 7;

image label = make\_empty\_image(0,0,0);

while(\*string){

image l = characters[size][(int)\*string];

image n = tile\_images(label, l, -size - 1 + (size+1)/2);

free\_image(label);

label = n;

++string;

}

image b = border\_image(label, label.h\*.25);

free\_image(label);

return b;

}

image get\_label\_v3(image \*\*characters, char \*string, int size)

{

size = size / 10;

if (size > 7) size = 7;

image label = make\_empty\_image(0, 0, 0);

while (\*string) {

image l = characters[size][(int)\*string];

image n = tile\_images(label, l, -size - 1 + (size + 1) / 2);

free\_image(label);

label = n;

++string;

}

image b = border\_image(label, label.h\*.25);

free\_image(label);

return b;

}

void draw\_label(image a, int r, int c, image label, const float \*rgb)

{

int w = label.w;

int h = label.h;

if (r - h >= 0) r = r - h;

int i, j, k;

for(j = 0; j < h && j + r < a.h; ++j){

for(i = 0; i < w && i + c < a.w; ++i){

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

float val = get\_pixel(label, i, j, k);

set\_pixel(a, i+c, j+r, k, rgb[k] \* val);

}

}

}

}

void draw\_box\_bw(image a, int x1, int y1, int x2, int y2, float brightness)

{

//normalize\_image(a);

int i;

if (x1 < 0) x1 = 0;

if (x1 >= a.w) x1 = a.w - 1;

if (x2 < 0) x2 = 0;

if (x2 >= a.w) x2 = a.w - 1;

if (y1 < 0) y1 = 0;

if (y1 >= a.h) y1 = a.h - 1;

if (y2 < 0) y2 = 0;

if (y2 >= a.h) y2 = a.h - 1;

for (i = x1; i <= x2; ++i) {

a.data[i + y1\*a.w + 0 \* a.w\*a.h] = brightness;

a.data[i + y2\*a.w + 0 \* a.w\*a.h] = brightness;

}

for (i = y1; i <= y2; ++i) {

a.data[x1 + i\*a.w + 0 \* a.w\*a.h] = brightness;

a.data[x2 + i\*a.w + 0 \* a.w\*a.h] = brightness;

}

}

void draw\_box\_width\_bw(image a, int x1, int y1, int x2, int y2, int w, float brightness)

{

int i;

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

float alternate\_color = (w % 2) ? (brightness) : (1.0 - brightness);

draw\_box\_bw(a, x1 + i, y1 + i, x2 - i, y2 - i, alternate\_color);

}

}

void draw\_box(image a, int x1, int y1, int x2, int y2, float r, float g, float b)

{

//normalize\_image(a);

int i;

if(x1 < 0) x1 = 0;

if(x1 >= a.w) x1 = a.w-1;

if(x2 < 0) x2 = 0;

if(x2 >= a.w) x2 = a.w-1;

if(y1 < 0) y1 = 0;

if(y1 >= a.h) y1 = a.h-1;

if(y2 < 0) y2 = 0;

if(y2 >= a.h) y2 = a.h-1;

for(i = x1; i <= x2; ++i){

a.data[i + y1\*a.w + 0\*a.w\*a.h] = r;

a.data[i + y2\*a.w + 0\*a.w\*a.h] = r;

a.data[i + y1\*a.w + 1\*a.w\*a.h] = g;

a.data[i + y2\*a.w + 1\*a.w\*a.h] = g;

a.data[i + y1\*a.w + 2\*a.w\*a.h] = b;

a.data[i + y2\*a.w + 2\*a.w\*a.h] = b;

}

for(i = y1; i <= y2; ++i){

a.data[x1 + i\*a.w + 0\*a.w\*a.h] = r;

a.data[x2 + i\*a.w + 0\*a.w\*a.h] = r;

a.data[x1 + i\*a.w + 1\*a.w\*a.h] = g;

a.data[x2 + i\*a.w + 1\*a.w\*a.h] = g;

a.data[x1 + i\*a.w + 2\*a.w\*a.h] = b;

a.data[x2 + i\*a.w + 2\*a.w\*a.h] = b;

}

}

void draw\_box\_width(image a, int x1, int y1, int x2, int y2, int w, float r, float g, float b)

{

int i;

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

draw\_box(a, x1+i, y1+i, x2-i, y2-i, r, g, b);

}

}

void draw\_bbox(image a, box bbox, int w, float r, float g, float b)

{

int left = (bbox.x-bbox.w/2)\*a.w;

int right = (bbox.x+bbox.w/2)\*a.w;

int top = (bbox.y-bbox.h/2)\*a.h;

int bot = (bbox.y+bbox.h/2)\*a.h;

int i;

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

draw\_box(a, left+i, top+i, right-i, bot-i, r, g, b);

}

}

image \*\*load\_alphabet()

{

int i, j;

const int nsize = 8;

image\*\* alphabets = (image\*\*)xcalloc(nsize, sizeof(image\*));

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

alphabets[j] = (image\*)xcalloc(128, sizeof(image));

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

char buff[256];

sprintf(buff, "data/labels/%d\_%d.png", i, j);

alphabets[j][i] = load\_image\_color(buff, 0, 0);

}

}

return alphabets;

}

// Creates array of detections with prob > thresh and fills best\_class for them

detection\_with\_class\* get\_actual\_detections(detection \*dets, int dets\_num, float thresh, int\* selected\_detections\_num, char \*\*names)

{

int selected\_num = 0;

detection\_with\_class\* result\_arr = (detection\_with\_class\*)xcalloc(dets\_num, sizeof(detection\_with\_class));

int i;

for (i = 0; i < dets\_num; ++i) {

int best\_class = -1;

float best\_class\_prob = thresh;

int j;

for (j = 0; j < dets[i].classes; ++j) {

int show = strncmp(names[j], "dont\_show", 9);

if (dets[i].prob[j] > best\_class\_prob && show) {

best\_class = j;

best\_class\_prob = dets[i].prob[j];

}

}

if (best\_class >= 0) {

result\_arr[selected\_num].det = dets[i];

result\_arr[selected\_num].best\_class = best\_class;

++selected\_num;

}

}

if (selected\_detections\_num)

\*selected\_detections\_num = selected\_num;

return result\_arr;

}

// compare to sort detection\*\* by bbox.x

int compare\_by\_lefts(const void \*a\_ptr, const void \*b\_ptr) {

const detection\_with\_class\* a = (detection\_with\_class\*)a\_ptr;

const detection\_with\_class\* b = (detection\_with\_class\*)b\_ptr;

const float delta = (a->det.bbox.x - a->det.bbox.w/2) - (b->det.bbox.x - b->det.bbox.w/2);

return delta < 0 ? -1 : delta > 0 ? 1 : 0;

}

// compare to sort detection\*\* by best\_class probability

int compare\_by\_probs(const void \*a\_ptr, const void \*b\_ptr) {

const detection\_with\_class\* a = (detection\_with\_class\*)a\_ptr;

const detection\_with\_class\* b = (detection\_with\_class\*)b\_ptr;

float delta = a->det.prob[a->best\_class] - b->det.prob[b->best\_class];

return delta < 0 ? -1 : delta > 0 ? 1 : 0;

}

void draw\_detections\_v3(image im, detection \*dets, int num, float thresh, char \*\*names, image \*\*alphabet, int classes, int ext\_output)

{

static int frame\_id = 0;

frame\_id++;

int selected\_detections\_num;

detection\_with\_class\* selected\_detections = get\_actual\_detections(dets, num, thresh, &selected\_detections\_num, names);

// text output

qsort(selected\_detections, selected\_detections\_num, sizeof(\*selected\_detections), compare\_by\_lefts);

int i;

for (i = 0; i < selected\_detections\_num; ++i) {

const int best\_class = selected\_detections[i].best\_class;

printf("%s: %.0f%%", names[best\_class], selected\_detections[i].det.prob[best\_class] \* 100);

if (ext\_output)

printf("\t(left\_x: %4.0f top\_y: %4.0f width: %4.0f height: %4.0f)\n",

round((selected\_detections[i].det.bbox.x - selected\_detections[i].det.bbox.w / 2)\*im.w),

round((selected\_detections[i].det.bbox.y - selected\_detections[i].det.bbox.h / 2)\*im.h),

round(selected\_detections[i].det.bbox.w\*im.w), round(selected\_detections[i].det.bbox.h\*im.h));

else

printf("\n");

int j;

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

if (selected\_detections[i].det.prob[j] > thresh && j != best\_class) {

printf("%s: %.0f%%", names[j], selected\_detections[i].det.prob[j] \* 100);

if (ext\_output)

printf("\t(left\_x: %4.0f top\_y: %4.0f width: %4.0f height: %4.0f)\n",

round((selected\_detections[i].det.bbox.x - selected\_detections[i].det.bbox.w / 2)\*im.w),

round((selected\_detections[i].det.bbox.y - selected\_detections[i].det.bbox.h / 2)\*im.h),

round(selected\_detections[i].det.bbox.w\*im.w), round(selected\_detections[i].det.bbox.h\*im.h));

else

printf("\n");

}

}

}

// image output

qsort(selected\_detections, selected\_detections\_num, sizeof(\*selected\_detections), compare\_by\_probs);

for (i = 0; i < selected\_detections\_num; ++i) {

int width = im.h \* .006;

if (width < 1)

width = 1;

/\*

if(0){

width = pow(prob, 1./2.)\*10+1;

alphabet = 0;

}

\*/

//printf("%d %s: %.0f%%\n", i, names[selected\_detections[i].best\_class], prob\*100);

int offset = selected\_detections[i].best\_class \* 123457 % classes;

float red = get\_color(2, offset, classes);

float green = get\_color(1, offset, classes);

float blue = get\_color(0, offset, classes);

float rgb[3];

//width = prob\*20+2;

rgb[0] = red;

rgb[1] = green;

rgb[2] = blue;

box b = selected\_detections[i].det.bbox;

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

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

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

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

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

if (left < 0) left = 0;

if (right > im.w - 1) right = im.w - 1;

if (top < 0) top = 0;

if (bot > im.h - 1) bot = im.h - 1;

//int b\_x\_center = (left + right) / 2;

//int b\_y\_center = (top + bot) / 2;

//int b\_width = right - left;

//int b\_height = bot - top;

//sprintf(labelstr, "%d x %d - w: %d, h: %d", b\_x\_center, b\_y\_center, b\_width, b\_height);

// you should create directory: result\_img

//static int copied\_frame\_id = -1;

//static image copy\_img;

//if (copied\_frame\_id != frame\_id) {

// copied\_frame\_id = frame\_id;

// if (copy\_img.data) free\_image(copy\_img);

// copy\_img = copy\_image(im);

//}

//image cropped\_im = crop\_image(copy\_img, left, top, right - left, bot - top);

//static int img\_id = 0;

//img\_id++;

//char image\_name[1024];

//int best\_class\_id = selected\_detections[i].best\_class;

//sprintf(image\_name, "result\_img/img\_%d\_%d\_%d\_%s.jpg", frame\_id, img\_id, best\_class\_id, names[best\_class\_id]);

//save\_image(cropped\_im, image\_name);

//free\_image(cropped\_im);

if (im.c == 1) {

draw\_box\_width\_bw(im, left, top, right, bot, width, 0.8); // 1 channel Black-White

}

else {

draw\_box\_width(im, left, top, right, bot, width, red, green, blue); // 3 channels RGB

}

if (alphabet) {

char labelstr[4096] = { 0 };

strcat(labelstr, names[selected\_detections[i].best\_class]);

int j;

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

if (selected\_detections[i].det.prob[j] > thresh && j != selected\_detections[i].best\_class) {

strcat(labelstr, ", ");

strcat(labelstr, names[j]);

}

}

image label = get\_label\_v3(alphabet, labelstr, (im.h\*.03));

draw\_label(im, top + width, left, label, rgb);

free\_image(label);

}

if (selected\_detections[i].det.mask) {

image mask = float\_to\_image(14, 14, 1, selected\_detections[i].det.mask);

image resized\_mask = resize\_image(mask, b.w\*im.w, b.h\*im.h);

image tmask = threshold\_image(resized\_mask, .5);

embed\_image(tmask, im, left, top);

free\_image(mask);

free\_image(resized\_mask);

free\_image(tmask);

}

}

free(selected\_detections);

}

void draw\_detections(image im, int num, float thresh, box \*boxes, float \*\*probs, char \*\*names, image \*\*alphabet, int classes)

{

int i;

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

int class\_id = max\_index(probs[i], classes);

float prob = probs[i][class\_id];

if(prob > thresh){

//// for comparison with OpenCV version of DNN Darknet Yolo v2

//printf("\n %f, %f, %f, %f, ", boxes[i].x, boxes[i].y, boxes[i].w, boxes[i].h);

// int k;

//for (k = 0; k < classes; ++k) {

// printf("%f, ", probs[i][k]);

//}

//printf("\n");

int width = im.h \* .012;

if(0){

width = pow(prob, 1./2.)\*10+1;

alphabet = 0;

}

int offset = class\_id\*123457 % classes;

float red = get\_color(2,offset,classes);

float green = get\_color(1,offset,classes);

float blue = get\_color(0,offset,classes);

float rgb[3];

//width = prob\*20+2;

rgb[0] = red;

rgb[1] = green;

rgb[2] = blue;

box b = boxes[i];

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

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

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

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

if(left < 0) left = 0;

if(right > im.w-1) right = im.w-1;

if(top < 0) top = 0;

if(bot > im.h-1) bot = im.h-1;

printf("%s: %.0f%%", names[class\_id], prob \* 100);

//printf(" - id: %d, x\_center: %d, y\_center: %d, width: %d, height: %d",

// class\_id, (right + left) / 2, (bot - top) / 2, right - left, bot - top);

printf("\n");

draw\_box\_width(im, left, top, right, bot, width, red, green, blue);

if (alphabet) {

image label = get\_label(alphabet, names[class\_id], (im.h\*.03)/10);

draw\_label(im, top + width, left, label, rgb);

}

}

}

}

void transpose\_image(image im)

{

assert(im.w == im.h);

int n, m;

int c;

for(c = 0; c < im.c; ++c){

for(n = 0; n < im.w-1; ++n){

for(m = n + 1; m < im.w; ++m){

float swap = im.data[m + im.w\*(n + im.h\*c)];

im.data[m + im.w\*(n + im.h\*c)] = im.data[n + im.w\*(m + im.h\*c)];

im.data[n + im.w\*(m + im.h\*c)] = swap;

}

}

}

}

void rotate\_image\_cw(image im, int times)

{

assert(im.w == im.h);

times = (times + 400) % 4;

int i, x, y, c;

int n = im.w;

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

for(c = 0; c < im.c; ++c){

for(x = 0; x < n/2; ++x){

for(y = 0; y < (n-1)/2 + 1; ++y){

float temp = im.data[y + im.w\*(x + im.h\*c)];

im.data[y + im.w\*(x + im.h\*c)] = im.data[n-1-x + im.w\*(y + im.h\*c)];

im.data[n-1-x + im.w\*(y + im.h\*c)] = im.data[n-1-y + im.w\*(n-1-x + im.h\*c)];

im.data[n-1-y + im.w\*(n-1-x + im.h\*c)] = im.data[x + im.w\*(n-1-y + im.h\*c)];

im.data[x + im.w\*(n-1-y + im.h\*c)] = temp;

}

}

}

}

}

void flip\_image(image a)

{

int i,j,k;

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

for(i = 0; i < a.h; ++i){

for(j = 0; j < a.w/2; ++j){

int index = j + a.w\*(i + a.h\*(k));

int flip = (a.w - j - 1) + a.w\*(i + a.h\*(k));

float swap = a.data[flip];

a.data[flip] = a.data[index];

a.data[index] = swap;

}

}

}

}

image image\_distance(image a, image b)

{

int i,j;

image dist = make\_image(a.w, a.h, 1);

for(i = 0; i < a.c; ++i){

for(j = 0; j < a.h\*a.w; ++j){

dist.data[j] += pow(a.data[i\*a.h\*a.w+j]-b.data[i\*a.h\*a.w+j],2);

}

}

for(j = 0; j < a.h\*a.w; ++j){

dist.data[j] = sqrt(dist.data[j]);

}

return dist;

}

void embed\_image(image source, image dest, int dx, int dy)

{

int x,y,k;

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

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

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

float val = get\_pixel(source, x,y,k);

set\_pixel(dest, dx+x, dy+y, k, val);

}

}

}

}

image collapse\_image\_layers(image source, int border)

{

int h = source.h;

h = (h+border)\*source.c - border;

image dest = make\_image(source.w, h, 1);

int i;

for(i = 0; i < source.c; ++i){

image layer = get\_image\_layer(source, i);

int h\_offset = i\*(source.h+border);

embed\_image(layer, dest, 0, h\_offset);

free\_image(layer);

}

return dest;

}

void constrain\_image(image im)

{

int i;

for(i = 0; i < im.w\*im.h\*im.c; ++i){

if(im.data[i] < 0) im.data[i] = 0;

if(im.data[i] > 1) im.data[i] = 1;

}

}

void normalize\_image(image p)

{

int i;

float min = 9999999;

float max = -999999;

for(i = 0; i < p.h\*p.w\*p.c; ++i){

float v = p.data[i];

if(v < min) min = v;

if(v > max) max = v;

}

if(max - min < .000000001){

min = 0;

max = 1;

}

for(i = 0; i < p.c\*p.w\*p.h; ++i){

p.data[i] = (p.data[i] - min)/(max-min);

}

}

void normalize\_image2(image p)

{

float\* min = (float\*)xcalloc(p.c, sizeof(float));

float\* max = (float\*)xcalloc(p.c, sizeof(float));

int i,j;

for(i = 0; i < p.c; ++i) min[i] = max[i] = p.data[i\*p.h\*p.w];

for(j = 0; j < p.c; ++j){

for(i = 0; i < p.h\*p.w; ++i){

float v = p.data[i+j\*p.h\*p.w];

if(v < min[j]) min[j] = v;

if(v > max[j]) max[j] = v;

}

}

for(i = 0; i < p.c; ++i){

if(max[i] - min[i] < .000000001){

min[i] = 0;

max[i] = 1;

}

}

for(j = 0; j < p.c; ++j){

for(i = 0; i < p.w\*p.h; ++i){

p.data[i+j\*p.h\*p.w] = (p.data[i+j\*p.h\*p.w] - min[j])/(max[j]-min[j]);

}

}

free(min);

free(max);

}

void copy\_image\_inplace(image src, image dst)

{

memcpy(dst.data, src.data, src.h\*src.w\*src.c \* sizeof(float));

}

image copy\_image(image p)

{

image copy = p;

copy.data = (float\*)xcalloc(p.h \* p.w \* p.c, sizeof(float));

memcpy(copy.data, p.data, p.h\*p.w\*p.c\*sizeof(float));

return copy;

}

void rgbgr\_image(image im)

{

int i;

for(i = 0; i < im.w\*im.h; ++i){

float swap = im.data[i];

im.data[i] = im.data[i+im.w\*im.h\*2];

im.data[i+im.w\*im.h\*2] = swap;

}

}

void show\_image(image p, const char \*name)

{

#ifdef OPENCV

show\_image\_cv(p, name);

#else

fprintf(stderr, "Not compiled with OpenCV, saving to %s.png instead\n", name);

save\_image(p, name);

#endif // OPENCV

}

void save\_image\_png(image im, const char \*name)

{

char buff[256];

//sprintf(buff, "%s (%d)", name, windows);

sprintf(buff, "%s.png", name);

unsigned char\* data = (unsigned char\*)xcalloc(im.w \* im.h \* im.c, sizeof(unsigned char));

int i,k;

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

for(i = 0; i < im.w\*im.h; ++i){

data[i\*im.c+k] = (unsigned char) (255\*im.data[i + k\*im.w\*im.h]);

}

}

int success = stbi\_write\_png(buff, im.w, im.h, im.c, data, im.w\*im.c);

free(data);

if(!success) fprintf(stderr, "Failed to write image %s\n", buff);

}

void save\_image\_options(image im, const char \*name, IMTYPE f, int quality)

{

char buff[256];

//sprintf(buff, "%s (%d)", name, windows);

if (f == PNG) sprintf(buff, "%s.png", name);

else if (f == BMP) sprintf(buff, "%s.bmp", name);

else if (f == TGA) sprintf(buff, "%s.tga", name);

else if (f == JPG) sprintf(buff, "%s.jpg", name);

else sprintf(buff, "%s.png", name);

unsigned char\* data = (unsigned char\*)xcalloc(im.w \* im.h \* im.c, sizeof(unsigned char));

int i, k;

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

for (i = 0; i < im.w\*im.h; ++i) {

data[i\*im.c + k] = (unsigned char)(255 \* im.data[i + k\*im.w\*im.h]);

}

}

int success = 0;

if (f == PNG) success = stbi\_write\_png(buff, im.w, im.h, im.c, data, im.w\*im.c);

else if (f == BMP) success = stbi\_write\_bmp(buff, im.w, im.h, im.c, data);

else if (f == TGA) success = stbi\_write\_tga(buff, im.w, im.h, im.c, data);

else if (f == JPG) success = stbi\_write\_jpg(buff, im.w, im.h, im.c, data, quality);

free(data);

if (!success) fprintf(stderr, "Failed to write image %s\n", buff);

}

void save\_image(image im, const char \*name)

{

save\_image\_options(im, name, JPG, 80);

}

void save\_image\_jpg(image p, const char \*name)

{

save\_image\_options(p, name, JPG, 80);

}

void show\_image\_layers(image p, char \*name)

{

int i;

char buff[256];

for(i = 0; i < p.c; ++i){

sprintf(buff, "%s - Layer %d", name, i);

image layer = get\_image\_layer(p, i);

show\_image(layer, buff);

free\_image(layer);

}

}

void show\_image\_collapsed(image p, char \*name)

{

image c = collapse\_image\_layers(p, 1);

show\_image(c, name);

free\_image(c);

}

image make\_empty\_image(int w, int h, int c)

{

image out;

out.data = 0;

out.h = h;

out.w = w;

out.c = c;

return out;

}

image make\_image(int w, int h, int c)

{

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

out.data = (float\*)xcalloc(h \* w \* c, sizeof(float));

return out;

}

image make\_random\_image(int w, int h, int c)

{

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

out.data = (float\*)xcalloc(h \* w \* c, sizeof(float));

int i;

for(i = 0; i < w\*h\*c; ++i){

out.data[i] = (rand\_normal() \* .25) + .5;

}

return out;

}

image float\_to\_image\_scaled(int w, int h, int c, float \*data)

{

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

int abs\_max = 0;

int i = 0;

for (i = 0; i < w\*h\*c; ++i) {

if (fabs(data[i]) > abs\_max) abs\_max = fabs(data[i]);

}

for (i = 0; i < w\*h\*c; ++i) {

out.data[i] = data[i] / abs\_max;

}

return out;

}

image float\_to\_image(int w, int h, int c, float \*data)

{

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

out.data = data;

return out;

}

image rotate\_crop\_image(image im, float rad, float s, int w, int h, float dx, float dy, float aspect)

{

int x, y, c;

float cx = im.w/2.;

float cy = im.h/2.;

image rot = make\_image(w, h, im.c);

for(c = 0; c < im.c; ++c){

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

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

float rx = cos(rad)\*((x - w/2.)/s\*aspect + dx/s\*aspect) - sin(rad)\*((y - h/2.)/s + dy/s) + cx;

float ry = sin(rad)\*((x - w/2.)/s\*aspect + dx/s\*aspect) + cos(rad)\*((y - h/2.)/s + dy/s) + cy;

float val = bilinear\_interpolate(im, rx, ry, c);

set\_pixel(rot, x, y, c, val);

}

}

}

return rot;

}

image rotate\_image(image im, float rad)

{

int x, y, c;

float cx = im.w/2.;

float cy = im.h/2.;

image rot = make\_image(im.w, im.h, im.c);

for(c = 0; c < im.c; ++c){

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

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

float rx = cos(rad)\*(x-cx) - sin(rad)\*(y-cy) + cx;

float ry = sin(rad)\*(x-cx) + cos(rad)\*(y-cy) + cy;

float val = bilinear\_interpolate(im, rx, ry, c);

set\_pixel(rot, x, y, c, val);

}

}

}

return rot;

}

void translate\_image(image m, float s)

{

int i;

for(i = 0; i < m.h\*m.w\*m.c; ++i) m.data[i] += s;

}

void scale\_image(image m, float s)

{

int i;

for(i = 0; i < m.h\*m.w\*m.c; ++i) m.data[i] \*= s;

}

image crop\_image(image im, int dx, int dy, int w, int h)

{

image cropped = make\_image(w, h, im.c);

int i, j, k;

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

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

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

int r = j + dy;

int c = i + dx;

float val = 0;

r = constrain\_int(r, 0, im.h-1);

c = constrain\_int(c, 0, im.w-1);

if (r >= 0 && r < im.h && c >= 0 && c < im.w) {

val = get\_pixel(im, c, r, k);

}

set\_pixel(cropped, i, j, k, val);

}

}

}

return cropped;

}

int best\_3d\_shift\_r(image a, image b, int min, int max)

{

if(min == max) return min;

int mid = floor((min + max) / 2.);

image c1 = crop\_image(b, 0, mid, b.w, b.h);

image c2 = crop\_image(b, 0, mid+1, b.w, b.h);

float d1 = dist\_array(c1.data, a.data, a.w\*a.h\*a.c, 10);

float d2 = dist\_array(c2.data, a.data, a.w\*a.h\*a.c, 10);

free\_image(c1);

free\_image(c2);

if(d1 < d2) return best\_3d\_shift\_r(a, b, min, mid);

else return best\_3d\_shift\_r(a, b, mid+1, max);

}

int best\_3d\_shift(image a, image b, int min, int max)

{

int i;

int best = 0;

float best\_distance = FLT\_MAX;

for(i = min; i <= max; i += 2){

image c = crop\_image(b, 0, i, b.w, b.h);

float d = dist\_array(c.data, a.data, a.w\*a.h\*a.c, 100);

if(d < best\_distance){

best\_distance = d;

best = i;

}

printf("%d %f\n", i, d);

free\_image(c);

}

return best;

}

void composite\_3d(char \*f1, char \*f2, char \*out, int delta)

{

if(!out) out = "out";

image a = load\_image(f1, 0,0,0);

image b = load\_image(f2, 0,0,0);

int shift = best\_3d\_shift\_r(a, b, -a.h/100, a.h/100);

image c1 = crop\_image(b, 10, shift, b.w, b.h);

float d1 = dist\_array(c1.data, a.data, a.w\*a.h\*a.c, 100);

image c2 = crop\_image(b, -10, shift, b.w, b.h);

float d2 = dist\_array(c2.data, a.data, a.w\*a.h\*a.c, 100);

if(d2 < d1 && 0){

image swap = a;

a = b;

b = swap;

shift = -shift;

printf("swapped, %d\n", shift);

}

else{

printf("%d\n", shift);

}

image c = crop\_image(b, delta, shift, a.w, a.h);

int i;

for(i = 0; i < c.w\*c.h; ++i){

c.data[i] = a.data[i];

}

#ifdef OPENCV

save\_image\_jpg(c, out);

#else

save\_image(c, out);

#endif

}

void fill\_image(image m, float s)

{

int i;

for (i = 0; i < m.h\*m.w\*m.c; ++i) m.data[i] = s;

}

void letterbox\_image\_into(image im, int w, int h, image boxed)

{

int new\_w = im.w;

int new\_h = im.h;

if (((float)w / im.w) < ((float)h / im.h)) {

new\_w = w;

new\_h = (im.h \* w) / im.w;

}

else {

new\_h = h;

new\_w = (im.w \* h) / im.h;

}

image resized = resize\_image(im, new\_w, new\_h);

embed\_image(resized, boxed, (w - new\_w) / 2, (h - new\_h) / 2);

free\_image(resized);

}

image letterbox\_image(image im, int w, int h)

{

int new\_w = im.w;

int new\_h = im.h;

if (((float)w / im.w) < ((float)h / im.h)) {

new\_w = w;

new\_h = (im.h \* w) / im.w;

}

else {

new\_h = h;

new\_w = (im.w \* h) / im.h;

}

image resized = resize\_image(im, new\_w, new\_h);

image boxed = make\_image(w, h, im.c);

fill\_image(boxed, .5);

//int i;

//for(i = 0; i < boxed.w\*boxed.h\*boxed.c; ++i) boxed.data[i] = 0;

embed\_image(resized, boxed, (w - new\_w) / 2, (h - new\_h) / 2);

free\_image(resized);

return boxed;

}

image resize\_max(image im, int max)

{

int w = im.w;

int h = im.h;

if(w > h){

h = (h \* max) / w;

w = max;

} else {

w = (w \* max) / h;

h = max;

}

if(w == im.w && h == im.h) return im;

image resized = resize\_image(im, w, h);

return resized;

}

image resize\_min(image im, int min)

{

int w = im.w;

int h = im.h;

if(w < h){

h = (h \* min) / w;

w = min;

} else {

w = (w \* min) / h;

h = min;

}

if(w == im.w && h == im.h) return im;

image resized = resize\_image(im, w, h);

return resized;

}

image random\_crop\_image(image im, int w, int h)

{

int dx = rand\_int(0, im.w - w);

int dy = rand\_int(0, im.h - h);

image crop = crop\_image(im, dx, dy, w, h);

return crop;

}

image random\_augment\_image(image im, float angle, float aspect, int low, int high, int size)

{

aspect = rand\_scale(aspect);

int r = rand\_int(low, high);

int min = (im.h < im.w\*aspect) ? im.h : im.w\*aspect;

float scale = (float)r / min;

float rad = rand\_uniform(-angle, angle) \* 2.0 \* M\_PI / 360.;

float dx = (im.w\*scale/aspect - size) / 2.;

float dy = (im.h\*scale - size) / 2.;

if(dx < 0) dx = 0;

if(dy < 0) dy = 0;

dx = rand\_uniform(-dx, dx);

dy = rand\_uniform(-dy, dy);

image crop = rotate\_crop\_image(im, rad, scale, size, size, dx, dy, aspect);

return crop;

}

float three\_way\_max(float a, float b, float c)

{

return (a > b) ? ( (a > c) ? a : c) : ( (b > c) ? b : c) ;

}

float three\_way\_min(float a, float b, float c)

{

return (a < b) ? ( (a < c) ? a : c) : ( (b < c) ? b : c) ;

}

// http://www.cs.rit.edu/~ncs/color/t\_convert.html

void rgb\_to\_hsv(image im)

{

assert(im.c == 3);

int i, j;

float r, g, b;

float h, s, v;

for(j = 0; j < im.h; ++j){

for(i = 0; i < im.w; ++i){

r = get\_pixel(im, i , j, 0);

g = get\_pixel(im, i , j, 1);

b = get\_pixel(im, i , j, 2);

float max = three\_way\_max(r,g,b);

float min = three\_way\_min(r,g,b);

float delta = max - min;

v = max;

if(max == 0){

s = 0;

h = 0;

}else{

s = delta/max;

if(r == max){

h = (g - b) / delta;

} else if (g == max) {

h = 2 + (b - r) / delta;

} else {

h = 4 + (r - g) / delta;

}

if (h < 0) h += 6;

h = h/6.;

}

set\_pixel(im, i, j, 0, h);

set\_pixel(im, i, j, 1, s);

set\_pixel(im, i, j, 2, v);

}

}

}

void hsv\_to\_rgb(image im)

{

assert(im.c == 3);

int i, j;

float r, g, b;

float h, s, v;

float f, p, q, t;

for(j = 0; j < im.h; ++j){

for(i = 0; i < im.w; ++i){

h = 6 \* get\_pixel(im, i , j, 0);

s = get\_pixel(im, i , j, 1);

v = get\_pixel(im, i , j, 2);

if (s == 0) {

r = g = b = v;

} else {

int index = floor(h);

f = h - index;

p = v\*(1-s);

q = v\*(1-s\*f);

t = v\*(1-s\*(1-f));

if(index == 0){

r = v; g = t; b = p;

} else if(index == 1){

r = q; g = v; b = p;

} else if(index == 2){

r = p; g = v; b = t;

} else if(index == 3){

r = p; g = q; b = v;

} else if(index == 4){

r = t; g = p; b = v;

} else {

r = v; g = p; b = q;

}

}

set\_pixel(im, i, j, 0, r);

set\_pixel(im, i, j, 1, g);

set\_pixel(im, i, j, 2, b);

}

}

}

image grayscale\_image(image im)

{

assert(im.c == 3);

int i, j, k;

image gray = make\_image(im.w, im.h, 1);

float scale[] = {0.587, 0.299, 0.114};

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

for(j = 0; j < im.h; ++j){

for(i = 0; i < im.w; ++i){

gray.data[i+im.w\*j] += scale[k]\*get\_pixel(im, i, j, k);

}

}

}

return gray;

}

image threshold\_image(image im, float thresh)

{

int i;

image t = make\_image(im.w, im.h, im.c);

for(i = 0; i < im.w\*im.h\*im.c; ++i){

t.data[i] = im.data[i]>thresh ? 1 : 0;

}

return t;

}

image blend\_image(image fore, image back, float alpha)

{

assert(fore.w == back.w && fore.h == back.h && fore.c == back.c);

image blend = make\_image(fore.w, fore.h, fore.c);

int i, j, k;

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

for(j = 0; j < fore.h; ++j){

for(i = 0; i < fore.w; ++i){

float val = alpha \* get\_pixel(fore, i, j, k) +

(1 - alpha)\* get\_pixel(back, i, j, k);

set\_pixel(blend, i, j, k, val);

}

}

}

return blend;

}

void scale\_image\_channel(image im, int c, float v)

{

int i, j;

for(j = 0; j < im.h; ++j){

for(i = 0; i < im.w; ++i){

float pix = get\_pixel(im, i, j, c);

pix = pix\*v;

set\_pixel(im, i, j, c, pix);

}

}

}

void translate\_image\_channel(image im, int c, float v)

{

int i, j;

for(j = 0; j < im.h; ++j){

for(i = 0; i < im.w; ++i){

float pix = get\_pixel(im, i, j, c);

pix = pix+v;

set\_pixel(im, i, j, c, pix);

}

}

}

image binarize\_image(image im)

{

image c = copy\_image(im);

int i;

for(i = 0; i < im.w \* im.h \* im.c; ++i){

if(c.data[i] > .5) c.data[i] = 1;

else c.data[i] = 0;

}

return c;

}

void saturate\_image(image im, float sat)

{

rgb\_to\_hsv(im);

scale\_image\_channel(im, 1, sat);

hsv\_to\_rgb(im);

constrain\_image(im);

}

void hue\_image(image im, float hue)

{

rgb\_to\_hsv(im);

int i;

for(i = 0; i < im.w\*im.h; ++i){

im.data[i] = im.data[i] + hue;

if (im.data[i] > 1) im.data[i] -= 1;

if (im.data[i] < 0) im.data[i] += 1;

}

hsv\_to\_rgb(im);

constrain\_image(im);

}

void exposure\_image(image im, float sat)

{

rgb\_to\_hsv(im);

scale\_image\_channel(im, 2, sat);

hsv\_to\_rgb(im);

constrain\_image(im);

}

void distort\_image(image im, float hue, float sat, float val)

{

if (im.c >= 3)

{

rgb\_to\_hsv(im);

scale\_image\_channel(im, 1, sat);

scale\_image\_channel(im, 2, val);

int i;

for(i = 0; i < im.w\*im.h; ++i){

im.data[i] = im.data[i] + hue;

if (im.data[i] > 1) im.data[i] -= 1;

if (im.data[i] < 0) im.data[i] += 1;

}

hsv\_to\_rgb(im);

}

else

{

scale\_image\_channel(im, 0, val);

}

constrain\_image(im);

}

void random\_distort\_image(image im, float hue, float saturation, float exposure)

{

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

float dsat = rand\_scale(saturation);

float dexp = rand\_scale(exposure);

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

}

void saturate\_exposure\_image(image im, float sat, float exposure)

{

rgb\_to\_hsv(im);

scale\_image\_channel(im, 1, sat);

scale\_image\_channel(im, 2, exposure);

hsv\_to\_rgb(im);

constrain\_image(im);

}

float bilinear\_interpolate(image im, float x, float y, int c)

{

int ix = (int) floorf(x);

int iy = (int) floorf(y);

float dx = x - ix;

float dy = y - iy;

float val = (1-dy) \* (1-dx) \* get\_pixel\_extend(im, ix, iy, c) +

dy \* (1-dx) \* get\_pixel\_extend(im, ix, iy+1, c) +

(1-dy) \* dx \* get\_pixel\_extend(im, ix+1, iy, c) +

dy \* dx \* get\_pixel\_extend(im, ix+1, iy+1, c);

return val;

}

void quantize\_image(image im)

{

int size = im.c \* im.w \* im.h;

int i;

for (i = 0; i < size; ++i) im.data[i] = (int)(im.data[i] \* 255) / 255. + (0.5/255);

}

void make\_image\_red(image im)

{

int r, c, k;

for (r = 0; r < im.h; ++r) {

for (c = 0; c < im.w; ++c) {

float val = 0;

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

val += get\_pixel(im, c, r, k);

set\_pixel(im, c, r, k, 0);

}

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

//set\_pixel(im, c, r, k, val);

}

set\_pixel(im, c, r, 0, val);

}

}

}

image make\_attention\_image(int img\_size, float \*original\_delta\_cpu, float \*original\_input\_cpu, int w, int h, int c)

{

image attention\_img;

attention\_img.w = w;

attention\_img.h = h;

attention\_img.c = c;

attention\_img.data = original\_delta\_cpu;

make\_image\_red(attention\_img);

int k;

float min\_val = 999999, mean\_val = 0, max\_val = -999999;

for (k = 0; k < img\_size; ++k) {

if (original\_delta\_cpu[k] < min\_val) min\_val = original\_delta\_cpu[k];

if (original\_delta\_cpu[k] > max\_val) max\_val = original\_delta\_cpu[k];

mean\_val += original\_delta\_cpu[k];

}

mean\_val = mean\_val / img\_size;

float range = max\_val - min\_val;

for (k = 0; k < img\_size; ++k) {

float val = original\_delta\_cpu[k];

val = fabs(mean\_val - val) / range;

original\_delta\_cpu[k] = val \* 4;

}

image resized = resize\_image(attention\_img, w / 4, h / 4);

attention\_img = resize\_image(resized, w, h);

free\_image(resized);

for (k = 0; k < img\_size; ++k) attention\_img.data[k] += original\_input\_cpu[k];

//normalize\_image(attention\_img);

//show\_image(attention\_img, "delta");

return attention\_img;

}

image resize\_image(image im, int w, int h)

{

if (im.w == w && im.h == h) return copy\_image(im);

image resized = make\_image(w, h, im.c);

image part = make\_image(w, im.h, im.c);

int r, c, k;

float w\_scale = (float)(im.w - 1) / (w - 1);

float h\_scale = (float)(im.h - 1) / (h - 1);

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

for(r = 0; r < im.h; ++r){

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

float val = 0;

if(c == w-1 || im.w == 1){

val = get\_pixel(im, im.w-1, r, k);

} else {

float sx = c\*w\_scale;

int ix = (int) sx;

float dx = sx - ix;

val = (1 - dx) \* get\_pixel(im, ix, r, k) + dx \* get\_pixel(im, ix+1, r, k);

}

set\_pixel(part, c, r, k, val);

}

}

}

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

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

float sy = r\*h\_scale;

int iy = (int) sy;

float dy = sy - iy;

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

float val = (1-dy) \* get\_pixel(part, c, iy, k);

set\_pixel(resized, c, r, k, val);

}

if(r == h-1 || im.h == 1) continue;

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

float val = dy \* get\_pixel(part, c, iy+1, k);

add\_pixel(resized, c, r, k, val);

}

}

}

free\_image(part);

return resized;

}

void test\_resize(char \*filename)

{

image im = load\_image(filename, 0,0, 3);

float mag = mag\_array(im.data, im.w\*im.h\*im.c);

printf("L2 Norm: %f\n", mag);

image gray = grayscale\_image(im);

image c1 = copy\_image(im);

image c2 = copy\_image(im);

image c3 = copy\_image(im);

image c4 = copy\_image(im);

distort\_image(c1, .1, 1.5, 1.5);

distort\_image(c2, -.1, .66666, .66666);

distort\_image(c3, .1, 1.5, .66666);

distort\_image(c4, .1, .66666, 1.5);

show\_image(im, "Original");

show\_image(gray, "Gray");

show\_image(c1, "C1");

show\_image(c2, "C2");

show\_image(c3, "C3");

show\_image(c4, "C4");

#ifdef OPENCV

while(1){

image aug = random\_augment\_image(im, 0, .75, 320, 448, 320);

show\_image(aug, "aug");

free\_image(aug);

float exposure = 1.15;

float saturation = 1.15;

float hue = .05;

image c = copy\_image(im);

float dexp = rand\_scale(exposure);

float dsat = rand\_scale(saturation);

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

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

show\_image(c, "rand");

printf("%f %f %f\n", dhue, dsat, dexp);

free\_image(c);

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

}

#endif

}

image load\_image\_stb(char \*filename, int channels)

{

int w, h, c;

unsigned char \*data = stbi\_load(filename, &w, &h, &c, channels);

if (!data) {

char shrinked\_filename[1024];

if (strlen(filename) >= 1024) sprintf(shrinked\_filename, "name is too long");

else sprintf(shrinked\_filename, "%s", filename);

fprintf(stderr, "Cannot load image \"%s\"\nSTB Reason: %s\n", shrinked\_filename, stbi\_failure\_reason());

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

fwrite(shrinked\_filename, sizeof(char), strlen(shrinked\_filename), fw);

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

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

fclose(fw);

if (check\_mistakes) {

printf("\n Error in load\_image\_stb() \n");

getchar();

}

return make\_image(10, 10, 3);

//exit(EXIT\_FAILURE);

}

if(channels) c = channels;

int i,j,k;

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

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

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

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

int dst\_index = i + w\*j + w\*h\*k;

int src\_index = k + c\*i + c\*w\*j;

im.data[dst\_index] = (float)data[src\_index]/255.;

}

}

}

free(data);

return im;

}

image load\_image\_stb\_resize(char \*filename, int w, int h, int c)

{

image out = load\_image\_stb(filename, c); // without OpenCV

if ((h && w) && (h != out.h || w != out.w)) {

image resized = resize\_image(out, w, h);

free\_image(out);

out = resized;

}

return out;

}

image load\_image(char \*filename, int w, int h, int c)

{

#ifdef OPENCV

//image out = load\_image\_stb(filename, c);

image out = load\_image\_cv(filename, c);

#else

image out = load\_image\_stb(filename, c); // without OpenCV

#endif // OPENCV

if((h && w) && (h != out.h || w != out.w)){

image resized = resize\_image(out, w, h);

free\_image(out);

out = resized;

}

return out;

}

image load\_image\_color(char \*filename, int w, int h)

{

return load\_image(filename, w, h, 3);

}

image get\_image\_layer(image m, int l)

{

image out = make\_image(m.w, m.h, 1);

int i;

for(i = 0; i < m.h\*m.w; ++i){

out.data[i] = m.data[i+l\*m.h\*m.w];

}

return out;

}

void print\_image(image m)

{

int i, j, k;

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

for(j =0 ; j < m.h; ++j){

for(k = 0; k < m.w; ++k){

printf("%.2lf, ", m.data[i\*m.h\*m.w + j\*m.w + k]);

if(k > 30) break;

}

printf("\n");

if(j > 30) break;

}

printf("\n");

}

printf("\n");

}

image collapse\_images\_vert(image \*ims, int n)

{

int color = 1;

int border = 1;

int h,w,c;

w = ims[0].w;

h = (ims[0].h + border) \* n - border;

c = ims[0].c;

if(c != 3 || !color){

w = (w+border)\*c - border;

c = 1;

}

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

int i,j;

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

int h\_offset = i\*(ims[0].h+border);

image copy = copy\_image(ims[i]);

//normalize\_image(copy);

if(c == 3 && color){

embed\_image(copy, filters, 0, h\_offset);

}

else{

for(j = 0; j < copy.c; ++j){

int w\_offset = j\*(ims[0].w+border);

image layer = get\_image\_layer(copy, j);

embed\_image(layer, filters, w\_offset, h\_offset);

free\_image(layer);

}

}

free\_image(copy);

}

return filters;

}

image collapse\_images\_horz(image \*ims, int n)

{

int color = 1;

int border = 1;

int h,w,c;

int size = ims[0].h;

h = size;

w = (ims[0].w + border) \* n - border;

c = ims[0].c;

if(c != 3 || !color){

h = (h+border)\*c - border;

c = 1;

}

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

int i,j;

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

int w\_offset = i\*(size+border);

image copy = copy\_image(ims[i]);

//normalize\_image(copy);

if(c == 3 && color){

embed\_image(copy, filters, w\_offset, 0);

}

else{

for(j = 0; j < copy.c; ++j){

int h\_offset = j\*(size+border);

image layer = get\_image\_layer(copy, j);

embed\_image(layer, filters, w\_offset, h\_offset);

free\_image(layer);

}

}

free\_image(copy);

}

return filters;

}

void show\_image\_normalized(image im, const char \*name)

{

image c = copy\_image(im);

normalize\_image(c);

show\_image(c, name);

free\_image(c);

}

void show\_images(image \*ims, int n, char \*window)

{

image m = collapse\_images\_vert(ims, n);

/\*

int w = 448;

int h = ((float)m.h/m.w) \* 448;

if(h > 896){

h = 896;

w = ((float)m.w/m.h) \* 896;

}

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

\*/

normalize\_image(m);

save\_image(m, window);

show\_image(m, window);

free\_image(m);

}

void free\_image(image m)

{

if(m.data){

free(m.data);

}

}

// Fast copy data from a contiguous byte array into the image.

LIB\_API void copy\_image\_from\_bytes(image im, char \*pdata)

{

unsigned char \*data = (unsigned char\*)pdata;

int i, k, j;

int w = im.w;

int h = im.h;

int c = im.c;

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

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

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

int dst\_index = i + w \* j + w \* h\*k;

int src\_index = k + c \* i + c \* w\*j;

im.data[dst\_index] = (float)data[src\_index] / 255.;

}

}

}

}