// Gaussian YOLOv3 implementation

// Author: Jiwoong Choi

// ICCV 2019 Paper: http://openaccess.thecvf.com/content\_ICCV\_2019/html/Choi\_Gaussian\_YOLOv3\_An\_Accurate\_and\_Fast\_Object\_Detector\_Using\_Localization\_ICCV\_2019\_paper.html

// arxiv.org: https://arxiv.org/abs/1904.04620v2

// source code: https://github.com/jwchoi384/Gaussian\_YOLOv3

#include "gaussian\_yolo\_layer.h"

#include "activations.h"

#include "blas.h"

#include "box.h"

#include "dark\_cuda.h"

#include "utils.h"

#include <stdio.h>

#include <assert.h>

#include <string.h>

#include <stdlib.h>

#ifndef M\_PI

#define M\_PI 3.141592

#endif

extern int check\_mistakes;

layer make\_gaussian\_yolo\_layer(int batch, int w, int h, int n, int total, int \*mask, int classes, int max\_boxes)

{

int i;

layer l = { (LAYER\_TYPE)0 };

l.type = GAUSSIAN\_YOLO;

l.n = n;

l.total = total;

l.batch = batch;

l.h = h;

l.w = w;

l.c = n\*(classes + 8 + 1);

l.out\_w = l.w;

l.out\_h = l.h;

l.out\_c = l.c;

l.classes = classes;

l.cost = (float\*)calloc(1, sizeof(float));

l.biases = (float\*)calloc(total\*2, sizeof(float));

if(mask) l.mask = mask;

else{

l.mask = (int\*)calloc(n, sizeof(int));

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

l.mask[i] = i;

}

}

l.bias\_updates = (float\*)calloc(n\*2, sizeof(float));

l.outputs = h\*w\*n\*(classes + 8 + 1);

l.inputs = l.outputs;

l.max\_boxes = max\_boxes;

l.truths = l.max\_boxes\*(4 + 1);

l.delta = (float\*)calloc(batch\*l.outputs, sizeof(float));

l.output = (float\*)calloc(batch\*l.outputs, sizeof(float));

for(i = 0; i < total\*2; ++i){

l.biases[i] = .5;

}

l.forward = forward\_gaussian\_yolo\_layer;

l.backward = backward\_gaussian\_yolo\_layer;

#ifdef GPU

l.forward\_gpu = forward\_gaussian\_yolo\_layer\_gpu;

l.backward\_gpu = backward\_gaussian\_yolo\_layer\_gpu;

l.output\_gpu = cuda\_make\_array(l.output, batch\*l.outputs);

l.delta\_gpu = cuda\_make\_array(l.delta, batch\*l.outputs);

free(l.output);

if (cudaSuccess == cudaHostAlloc(&l.output, batch\*l.outputs \* sizeof(float), cudaHostRegisterMapped)) l.output\_pinned = 1;

else {

cudaGetLastError(); // reset CUDA-error

l.output = (float\*)calloc(batch \* l.outputs, sizeof(float));

}

free(l.delta);

if (cudaSuccess == cudaHostAlloc(&l.delta, batch\*l.outputs \* sizeof(float), cudaHostRegisterMapped)) l.delta\_pinned = 1;

else {

cudaGetLastError(); // reset CUDA-error

l.delta = (float\*)calloc(batch \* l.outputs, sizeof(float));

}

#endif

//fprintf(stderr, "Gaussian\_yolo\n");

srand(time(0));

return l;

}

void resize\_gaussian\_yolo\_layer(layer \*l, int w, int h)

{

l->w = w;

l->h = h;

l->outputs = h\*w\*l->n\*(l->classes + 8 + 1);

l->inputs = l->outputs;

//l->output = (float \*)realloc(l->output, l->batch\*l->outputs \* sizeof(float));

//l->delta = (float \*)realloc(l->delta, l->batch\*l->outputs \* sizeof(float));

if (!l->output\_pinned) l->output = (float\*)realloc(l->output, l->batch\*l->outputs \* sizeof(float));

if (!l->delta\_pinned) l->delta = (float\*)realloc(l->delta, l->batch\*l->outputs \* sizeof(float));

#ifdef GPU

if (l->output\_pinned) {

CHECK\_CUDA(cudaFreeHost(l->output));

if (cudaSuccess != cudaHostAlloc(&l->output, l->batch\*l->outputs \* sizeof(float), cudaHostRegisterMapped)) {

cudaGetLastError(); // reset CUDA-error

l->output = (float\*)calloc(l->batch \* l->outputs, sizeof(float));

l->output\_pinned = 0;

}

}

if (l->delta\_pinned) {

CHECK\_CUDA(cudaFreeHost(l->delta));

if (cudaSuccess != cudaHostAlloc(&l->delta, l->batch\*l->outputs \* sizeof(float), cudaHostRegisterMapped)) {

cudaGetLastError(); // reset CUDA-error

l->delta = (float\*)calloc(l->batch \* l->outputs, sizeof(float));

l->delta\_pinned = 0;

}

}

cuda\_free(l->delta\_gpu);

cuda\_free(l->output\_gpu);

l->delta\_gpu = cuda\_make\_array(l->delta, l->batch\*l->outputs);

l->output\_gpu = cuda\_make\_array(l->output, l->batch\*l->outputs);

#endif

}

box get\_gaussian\_yolo\_box(float \*x, float \*biases, int n, int index, int i, int j, int lw, int lh, int w, int h, int stride, YOLO\_POINT yolo\_point)

{

box b;

b.w = exp(x[index + 4 \* stride]) \* biases[2 \* n] / w;

b.h = exp(x[index + 6 \* stride]) \* biases[2 \* n + 1] / h;

b.x = (i + x[index + 0 \* stride]) / lw;

b.y = (j + x[index + 2 \* stride]) / lh;

if (yolo\_point == YOLO\_CENTER) {

}

else if (yolo\_point == YOLO\_LEFT\_TOP) {

b.x = (i + x[index + 0 \* stride]) / lw + b.w / 2;

b.y = (j + x[index + 2 \* stride]) / lh + b.h / 2;

}

else if (yolo\_point == YOLO\_RIGHT\_BOTTOM) {

b.x = (i + x[index + 0 \* stride]) / lw - b.w / 2;

b.y = (j + x[index + 2 \* stride]) / lh - b.h / 2;

}

return b;

}

static inline float fix\_nan\_inf(float val)

{

if (isnan(val) || isinf(val)) val = 0;

return val;

}

static inline float clip\_value(float val, const float max\_val)

{

if (val > max\_val) val = max\_val;

else if (val < -max\_val) val = -max\_val;

return val;

}

float delta\_gaussian\_yolo\_box(box truth, float \*x, float \*biases, int n, int index, int i, int j, int lw, int lh, int w, int h, float \*delta,

float scale, int stride, float iou\_normalizer, IOU\_LOSS iou\_loss, float uc\_normalizer, int accumulate, YOLO\_POINT yolo\_point, float max\_delta)

{

box pred = get\_gaussian\_yolo\_box(x, biases, n, index, i, j, lw, lh, w, h, stride, yolo\_point);

float iou;

ious all\_ious = { 0 };

all\_ious.iou = box\_iou(pred, truth);

all\_ious.giou = box\_giou(pred, truth);

all\_ious.diou = box\_diou(pred, truth);

all\_ious.ciou = box\_ciou(pred, truth);

if (pred.w == 0) { pred.w = 1.0; }

if (pred.h == 0) { pred.h = 1.0; }

float sigma\_const = 0.3;

float epsi = pow(10,-9);

float dx, dy, dw, dh;

iou = all\_ious.iou;

float tx, ty, tw, th;

tx = (truth.x\*lw - i);

ty = (truth.y\*lh - j);

tw = log(truth.w\*w / biases[2 \* n]);

th = log(truth.h\*h / biases[2 \* n + 1]);

if (yolo\_point == YOLO\_CENTER) {

}

else if (yolo\_point == YOLO\_LEFT\_TOP) {

tx = ((truth.x - truth.w / 2)\*lw - i);

ty = ((truth.y - truth.h / 2)\*lh - j);

}

else if (yolo\_point == YOLO\_RIGHT\_BOTTOM) {

tx = ((truth.x + truth.w / 2)\*lw - i);

ty = ((truth.y + truth.h / 2)\*lh - j);

}

dx = (tx - x[index + 0 \* stride]);

dy = (ty - x[index + 2 \* stride]);

dw = (tw - x[index + 4 \* stride]);

dh = (th - x[index + 6 \* stride]);

// Gaussian

float in\_exp\_x = dx / x[index+1\*stride];

float in\_exp\_x\_2 = pow(in\_exp\_x, 2);

float normal\_dist\_x = exp(in\_exp\_x\_2\*(-1./2.))/(sqrt(M\_PI \* 2.0)\*(x[index+1\*stride]+sigma\_const));

float in\_exp\_y = dy / x[index+3\*stride];

float in\_exp\_y\_2 = pow(in\_exp\_y, 2);

float normal\_dist\_y = exp(in\_exp\_y\_2\*(-1./2.))/(sqrt(M\_PI \* 2.0)\*(x[index+3\*stride]+sigma\_const));

float in\_exp\_w = dw / x[index+5\*stride];

float in\_exp\_w\_2 = pow(in\_exp\_w, 2);

float normal\_dist\_w = exp(in\_exp\_w\_2\*(-1./2.))/(sqrt(M\_PI \* 2.0)\*(x[index+5\*stride]+sigma\_const));

float in\_exp\_h = dh / x[index+7\*stride];

float in\_exp\_h\_2 = pow(in\_exp\_h, 2);

float normal\_dist\_h = exp(in\_exp\_h\_2\*(-1./2.))/(sqrt(M\_PI \* 2.0)\*(x[index+7\*stride]+sigma\_const));

float temp\_x = (1./2.) \* 1./(normal\_dist\_x+epsi) \* normal\_dist\_x \* scale;

float temp\_y = (1./2.) \* 1./(normal\_dist\_y+epsi) \* normal\_dist\_y \* scale;

float temp\_w = (1./2.) \* 1./(normal\_dist\_w+epsi) \* normal\_dist\_w \* scale;

float temp\_h = (1./2.) \* 1./(normal\_dist\_h+epsi) \* normal\_dist\_h \* scale;

if (!accumulate) {

delta[index + 0 \* stride] = 0;

delta[index + 1 \* stride] = 0;

delta[index + 2 \* stride] = 0;

delta[index + 3 \* stride] = 0;

delta[index + 4 \* stride] = 0;

delta[index + 5 \* stride] = 0;

delta[index + 6 \* stride] = 0;

delta[index + 7 \* stride] = 0;

}

float delta\_x = temp\_x \* in\_exp\_x \* (1. / x[index + 1 \* stride]);

float delta\_y = temp\_y \* in\_exp\_y \* (1. / x[index + 3 \* stride]);

float delta\_w = temp\_w \* in\_exp\_w \* (1. / x[index + 5 \* stride]);

float delta\_h = temp\_h \* in\_exp\_h \* (1. / x[index + 7 \* stride]);

float delta\_ux = temp\_x \* (in\_exp\_x\_2 / x[index + 1 \* stride] - 1. / (x[index + 1 \* stride] + sigma\_const));

float delta\_uy = temp\_y \* (in\_exp\_y\_2 / x[index + 3 \* stride] - 1. / (x[index + 3 \* stride] + sigma\_const));

float delta\_uw = temp\_w \* (in\_exp\_w\_2 / x[index + 5 \* stride] - 1. / (x[index + 5 \* stride] + sigma\_const));

float delta\_uh = temp\_h \* (in\_exp\_h\_2 / x[index + 7 \* stride] - 1. / (x[index + 7 \* stride] + sigma\_const));

if (iou\_loss != MSE) {

// GIoU

iou = all\_ious.giou;

// https://github.com/generalized-iou/g-darknet

// https://arxiv.org/abs/1902.09630v2

// https://giou.stanford.edu/

// https://arxiv.org/abs/1911.08287v1

// https://github.com/Zzh-tju/DIoU-darknet

all\_ious.dx\_iou = dx\_box\_iou(pred, truth, iou\_loss);

float dx, dy, dw, dh;

dx = all\_ious.dx\_iou.dt;

dy = all\_ious.dx\_iou.db;

dw = all\_ious.dx\_iou.dl;

dh = all\_ious.dx\_iou.dr;

if (yolo\_point == YOLO\_CENTER) {

}

else if (yolo\_point == YOLO\_LEFT\_TOP) {

dx = dx - dw/2;

dy = dy - dh/2;

}

else if (yolo\_point == YOLO\_RIGHT\_BOTTOM) {

dx = dx + dw / 2;

dy = dy + dh / 2;

}

// jacobian^t (transpose)

//float dx = (all\_ious.dx\_iou.dl + all\_ious.dx\_iou.dr);

//float dy = (all\_ious.dx\_iou.dt + all\_ious.dx\_iou.db);

//float dw = ((-0.5 \* all\_ious.dx\_iou.dl) + (0.5 \* all\_ious.dx\_iou.dr));

//float dh = ((-0.5 \* all\_ious.dx\_iou.dt) + (0.5 \* all\_ious.dx\_iou.db));

// predict exponential, apply gradient of e^delta\_t ONLY for w,h

dw \*= exp(x[index + 4 \* stride]);

dh \*= exp(x[index + 6 \* stride]);

delta\_x = dx;

delta\_y = dy;

delta\_w = dw;

delta\_h = dh;

}

// normalize iou weight, for GIoU

delta\_x \*= iou\_normalizer;

delta\_y \*= iou\_normalizer;

delta\_w \*= iou\_normalizer;

delta\_h \*= iou\_normalizer;

// normalize Uncertainty weight

delta\_ux \*= uc\_normalizer;

delta\_uy \*= uc\_normalizer;

delta\_uw \*= uc\_normalizer;

delta\_uh \*= uc\_normalizer;

delta\_x = fix\_nan\_inf(delta\_x);

delta\_y = fix\_nan\_inf(delta\_y);

delta\_w = fix\_nan\_inf(delta\_w);

delta\_h = fix\_nan\_inf(delta\_h);

delta\_ux = fix\_nan\_inf(delta\_ux);

delta\_uy = fix\_nan\_inf(delta\_uy);

delta\_uw = fix\_nan\_inf(delta\_uw);

delta\_uh = fix\_nan\_inf(delta\_uh);

if (max\_delta != FLT\_MAX) {

delta\_x = clip\_value(delta\_x, max\_delta);

delta\_y = clip\_value(delta\_y, max\_delta);

delta\_w = clip\_value(delta\_w, max\_delta);

delta\_h = clip\_value(delta\_h, max\_delta);

delta\_ux = clip\_value(delta\_ux, max\_delta);

delta\_uy = clip\_value(delta\_uy, max\_delta);

delta\_uw = clip\_value(delta\_uw, max\_delta);

delta\_uh = clip\_value(delta\_uh, max\_delta);

}

delta[index + 0 \* stride] += delta\_x;

delta[index + 2 \* stride] += delta\_y;

delta[index + 4 \* stride] += delta\_w;

delta[index + 6 \* stride] += delta\_h;

delta[index + 1 \* stride] += delta\_ux;

delta[index + 3 \* stride] += delta\_uy;

delta[index + 5 \* stride] += delta\_uw;

delta[index + 7 \* stride] += delta\_uh;

return iou;

}

void averages\_gaussian\_yolo\_deltas(int class\_index, int box\_index, int stride, int classes, float \*delta)

{

int classes\_in\_one\_box = 0;

int c;

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

if (delta[class\_index + stride\*c] > 0) classes\_in\_one\_box++;

}

if (classes\_in\_one\_box > 0) {

delta[box\_index + 0 \* stride] /= classes\_in\_one\_box;

delta[box\_index + 1 \* stride] /= classes\_in\_one\_box;

delta[box\_index + 2 \* stride] /= classes\_in\_one\_box;

delta[box\_index + 3 \* stride] /= classes\_in\_one\_box;

delta[box\_index + 4 \* stride] /= classes\_in\_one\_box;

delta[box\_index + 5 \* stride] /= classes\_in\_one\_box;

delta[box\_index + 6 \* stride] /= classes\_in\_one\_box;

delta[box\_index + 7 \* stride] /= classes\_in\_one\_box;

}

}

void delta\_gaussian\_yolo\_class(float \*output, float \*delta, int index, int class\_id, int classes, int stride, float \*avg\_cat, float label\_smooth\_eps, float \*classes\_multipliers)

{

int n;

if (delta[index]){

float y\_true = 1;

if (label\_smooth\_eps) y\_true = y\_true \* (1 - label\_smooth\_eps) + 0.5\*label\_smooth\_eps;

delta[index + stride\*class\_id] = y\_true - output[index + stride\*class\_id];

//delta[index + stride\*class\_id] = 1 - output[index + stride\*class\_id];

if (classes\_multipliers) delta[index + stride\*class\_id] \*= classes\_multipliers[class\_id];

if(avg\_cat) \*avg\_cat += output[index + stride\*class\_id];

return;

}

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

float y\_true = ((n == class\_id) ? 1 : 0);

if (label\_smooth\_eps) y\_true = y\_true \* (1 - label\_smooth\_eps) + 0.5\*label\_smooth\_eps;

delta[index + stride\*n] = y\_true - output[index + stride\*n];

if (classes\_multipliers && n == class\_id) delta[index + stride\*class\_id] \*= classes\_multipliers[class\_id];

if(n == class\_id && avg\_cat) \*avg\_cat += output[index + stride\*n];

}

}

int compare\_gaussian\_yolo\_class(float \*output, int classes, int class\_index, int stride, float objectness, int class\_id, float conf\_thresh)

{

int j;

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

//float prob = objectness \* output[class\_index + stride\*j];

float prob = output[class\_index + stride\*j];

if (prob > conf\_thresh) {

return 1;

}

}

return 0;

}

static int entry\_gaussian\_index(layer l, int batch, int location, int entry)

{

int n = location / (l.w\*l.h);

int loc = location % (l.w\*l.h);

return batch\*l.outputs + n\*l.w\*l.h\*(8+l.classes+1) + entry\*l.w\*l.h + loc;

}

void forward\_gaussian\_yolo\_layer(const layer l, network\_state state)

{

int i,j,b,t,n;

memcpy(l.output, state.input, l.outputs\*l.batch\*sizeof(float));

#ifndef GPU

for (b = 0; b < l.batch; ++b){

for(n = 0; n < l.n; ++n){

// x : mu, sigma

int index = entry\_gaussian\_index(l, b, n\*l.w\*l.h, 0);

activate\_array(l.output + index, 2\*l.w\*l.h, LOGISTIC);

scal\_add\_cpu(l.w\*l.h, l.scale\_x\_y, -0.5\*(l.scale\_x\_y - 1), l.output + index, 1); // scale x

// y : mu, sigma

index = entry\_gaussian\_index(l, b, n\*l.w\*l.h, 2);

activate\_array(l.output + index, 2\*l.w\*l.h, LOGISTIC);

scal\_add\_cpu(l.w\*l.h, l.scale\_x\_y, -0.5\*(l.scale\_x\_y - 1), l.output + index, 1); // scale y

// w : sigma

index = entry\_gaussian\_index(l, b, n\*l.w\*l.h, 5);

activate\_array(l.output + index, l.w\*l.h, LOGISTIC);

// h : sigma

index = entry\_gaussian\_index(l, b, n\*l.w\*l.h, 7);

activate\_array(l.output + index, l.w\*l.h, LOGISTIC);

// objectness & class

index = entry\_gaussian\_index(l, b, n\*l.w\*l.h, 8);

activate\_array(l.output + index, (1+l.classes)\*l.w\*l.h, LOGISTIC);

}

}

#endif

memset(l.delta, 0, l.outputs \* l.batch \* sizeof(float));

if (!state.train) return;

float avg\_iou = 0;

float recall = 0;

float recall75 = 0;

float avg\_cat = 0;

float avg\_obj = 0;

float avg\_anyobj = 0;

int count = 0;

int class\_count = 0;

\*(l.cost) = 0;

for (b = 0; b < l.batch; ++b) {

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

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

for (n = 0; n < l.n; ++n) {

int box\_index = entry\_gaussian\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 0);

box pred = get\_gaussian\_yolo\_box(l.output, l.biases, l.mask[n], box\_index, i, j, l.w, l.h, state.net.w, state.net.h, l.w\*l.h, l.yolo\_point);

float best\_match\_iou = 0;

int best\_match\_t = 0;

float best\_iou = 0;

int best\_t = 0;

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

box truth = float\_to\_box\_stride(state.truth + t\*(4 + 1) + b\*l.truths, 1);

int class\_id = state.truth[t\*(4 + 1) + b\*l.truths + 4];

if (class\_id >= l.classes) {

printf("\n Warning: in txt-labels class\_id=%d >= classes=%d in cfg-file. In txt-labels class\_id should be [from 0 to %d] \n", class\_id, l.classes, l.classes - 1);

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

if (check\_mistakes) getchar();

continue; // if label contains class\_id more than number of classes in the cfg-file

}

if(!truth.x) break;

int class\_index = entry\_gaussian\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 9);

int obj\_index = entry\_gaussian\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 8);

float objectness = l.output[obj\_index];

int class\_id\_match = compare\_gaussian\_yolo\_class(l.output, l.classes, class\_index, l.w\*l.h, objectness, class\_id, 0.25f);

float iou = box\_iou(pred, truth);

if (iou > best\_match\_iou && class\_id\_match == 1) {

best\_match\_iou = iou;

best\_match\_t = t;

}

if (iou > best\_iou) {

best\_iou = iou;

best\_t = t;

}

}

int obj\_index = entry\_gaussian\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 8);

avg\_anyobj += l.output[obj\_index];

l.delta[obj\_index] = l.cls\_normalizer \* (0 - l.output[obj\_index]);

if (best\_match\_iou > l.ignore\_thresh) {

l.delta[obj\_index] = 0;

}

else if (state.net.adversarial) {

int class\_index = entry\_gaussian\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 9);

int stride = l.w\*l.h;

float scale = pred.w \* pred.h;

if (scale > 0) scale = sqrt(scale);

l.delta[obj\_index] = scale \* l.cls\_normalizer \* (0 - l.output[obj\_index]);

int cl\_id;

for (cl\_id = 0; cl\_id < l.classes; ++cl\_id) {

if (l.output[class\_index + stride\*cl\_id] \* l.output[obj\_index] > 0.25)

l.delta[class\_index + stride\*cl\_id] = scale \* (0 - l.output[class\_index + stride\*cl\_id]);

}

}

if (best\_iou > l.truth\_thresh) {

l.delta[obj\_index] = l.cls\_normalizer \* (1 - l.output[obj\_index]);

int class\_id = state.truth[best\_t\*(4 + 1) + b\*l.truths + 4];

if (l.map) class\_id = l.map[class\_id];

int class\_index = entry\_gaussian\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 9);

delta\_gaussian\_yolo\_class(l.output, l.delta, class\_index, class\_id, l.classes, l.w\*l.h, 0, l.label\_smooth\_eps, l.classes\_multipliers);

box truth = float\_to\_box\_stride(state.truth + best\_t\*(4 + 1) + b\*l.truths, 1);

const float class\_multiplier = (l.classes\_multipliers) ? l.classes\_multipliers[class\_id] : 1.0f;

delta\_gaussian\_yolo\_box(truth, l.output, l.biases, l.mask[n], box\_index, i, j, l.w, l.h, state.net.w, state.net.h, l.delta, (2-truth.w\*truth.h), l.w\*l.h, l.iou\_normalizer \* class\_multiplier, l.iou\_loss, l.uc\_normalizer, 1, l.yolo\_point, l.max\_delta);

}

}

}

}

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

box truth = float\_to\_box\_stride(state.truth + t\*(4 + 1) + b\*l.truths, 1);

if(!truth.x) break;

float best\_iou = 0;

int best\_n = 0;

i = (truth.x \* l.w);

j = (truth.y \* l.h);

if (l.yolo\_point == YOLO\_CENTER) {

}

else if (l.yolo\_point == YOLO\_LEFT\_TOP) {

i = min\_val\_cmp(l.w-1, max\_val\_cmp(0, ((truth.x - truth.w / 2) \* l.w)));

j = min\_val\_cmp(l.h-1, max\_val\_cmp(0, ((truth.y - truth.h / 2) \* l.h)));

}

else if (l.yolo\_point == YOLO\_RIGHT\_BOTTOM) {

i = min\_val\_cmp(l.w-1, max\_val\_cmp(0, ((truth.x + truth.w / 2) \* l.w)));

j = min\_val\_cmp(l.h-1, max\_val\_cmp(0, ((truth.y + truth.h / 2) \* l.h)));

}

box truth\_shift = truth;

truth\_shift.x = truth\_shift.y = 0;

for(n = 0; n < l.total; ++n){

box pred = {0};

pred.w = l.biases[2\*n]/ state.net.w;

pred.h = l.biases[2\*n+1]/ state.net.h;

float iou = box\_iou(pred, truth\_shift);

if (iou > best\_iou){

best\_iou = iou;

best\_n = n;

}

}

int mask\_n = int\_index(l.mask, best\_n, l.n);

if(mask\_n >= 0){

int class\_id = state.truth[t\*(4 + 1) + b\*l.truths + 4];

if (l.map) class\_id = l.map[class\_id];

int box\_index = entry\_gaussian\_index(l, b, mask\_n\*l.w\*l.h + j\*l.w + i, 0);

const float class\_multiplier = (l.classes\_multipliers) ? l.classes\_multipliers[class\_id] : 1.0f;

float iou = delta\_gaussian\_yolo\_box(truth, l.output, l.biases, best\_n, box\_index, i, j, l.w, l.h, state.net.w, state.net.h, l.delta, (2-truth.w\*truth.h), l.w\*l.h, l.iou\_normalizer \* class\_multiplier, l.iou\_loss, l.uc\_normalizer, 1, l.yolo\_point, l.max\_delta);

int obj\_index = entry\_gaussian\_index(l, b, mask\_n\*l.w\*l.h + j\*l.w + i, 8);

avg\_obj += l.output[obj\_index];

l.delta[obj\_index] = class\_multiplier \* l.cls\_normalizer \* (1 - l.output[obj\_index]);

int class\_index = entry\_gaussian\_index(l, b, mask\_n\*l.w\*l.h + j\*l.w + i, 9);

delta\_gaussian\_yolo\_class(l.output, l.delta, class\_index, class\_id, l.classes, l.w\*l.h, &avg\_cat, l.label\_smooth\_eps, l.classes\_multipliers);

++count;

++class\_count;

if(iou > .5) recall += 1;

if(iou > .75) recall75 += 1;

avg\_iou += iou;

}

// iou\_thresh

for (n = 0; n < l.total; ++n) {

int mask\_n = int\_index(l.mask, n, l.n);

if (mask\_n >= 0 && n != best\_n && l.iou\_thresh < 1.0f) {

box pred = { 0 };

pred.w = l.biases[2 \* n] / state.net.w;

pred.h = l.biases[2 \* n + 1] / state.net.h;

float iou = box\_iou\_kind(pred, truth\_shift, l.iou\_thresh\_kind); // IOU, GIOU, MSE, DIOU, CIOU

// iou, n

if (iou > l.iou\_thresh) {

int class\_id = state.truth[t\*(4 + 1) + b\*l.truths + 4];

if (l.map) class\_id = l.map[class\_id];

int box\_index = entry\_gaussian\_index(l, b, mask\_n\*l.w\*l.h + j\*l.w + i, 0);

const float class\_multiplier = (l.classes\_multipliers) ? l.classes\_multipliers[class\_id] : 1.0f;

float iou = delta\_gaussian\_yolo\_box(truth, l.output, l.biases, n, box\_index, i, j, l.w, l.h, state.net.w, state.net.h, l.delta, (2 - truth.w\*truth.h), l.w\*l.h, l.iou\_normalizer \* class\_multiplier, l.iou\_loss, l.uc\_normalizer, 1, l.yolo\_point, l.max\_delta);

int obj\_index = entry\_gaussian\_index(l, b, mask\_n\*l.w\*l.h + j\*l.w + i, 8);

avg\_obj += l.output[obj\_index];

l.delta[obj\_index] = class\_multiplier \* l.cls\_normalizer \* (1 - l.output[obj\_index]);

int class\_index = entry\_gaussian\_index(l, b, mask\_n\*l.w\*l.h + j\*l.w + i, 9);

delta\_gaussian\_yolo\_class(l.output, l.delta, class\_index, class\_id, l.classes, l.w\*l.h, &avg\_cat, l.label\_smooth\_eps, l.classes\_multipliers);

++count;

++class\_count;

if (iou > .5) recall += 1;

if (iou > .75) recall75 += 1;

avg\_iou += iou;

}

}

}

}

// averages the deltas obtained by the function: delta\_yolo\_box()\_accumulate

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

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

for (n = 0; n < l.n; ++n) {

int box\_index = entry\_gaussian\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 0);

int class\_index = entry\_gaussian\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 9);

const int stride = l.w\*l.h;

averages\_gaussian\_yolo\_deltas(class\_index, box\_index, stride, l.classes, l.delta);

}

}

}

}

// calculate: Classification-loss, IoU-loss and Uncertainty-loss

const int stride = l.w\*l.h;

float\* classification\_lost = (float \*)calloc(l.batch \* l.outputs, sizeof(float));

memcpy(classification\_lost, l.delta, l.batch \* l.outputs \* sizeof(float));

for (b = 0; b < l.batch; ++b) {

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

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

for (n = 0; n < l.n; ++n) {

int box\_index = entry\_gaussian\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 0);

classification\_lost[box\_index + 0 \* stride] = 0;

classification\_lost[box\_index + 1 \* stride] = 0;

classification\_lost[box\_index + 2 \* stride] = 0;

classification\_lost[box\_index + 3 \* stride] = 0;

classification\_lost[box\_index + 4 \* stride] = 0;

classification\_lost[box\_index + 5 \* stride] = 0;

classification\_lost[box\_index + 6 \* stride] = 0;

classification\_lost[box\_index + 7 \* stride] = 0;

}

}

}

}

float class\_loss = pow(mag\_array(classification\_lost, l.outputs \* l.batch), 2);

free(classification\_lost);

float\* except\_uncertainty\_lost = (float \*)calloc(l.batch \* l.outputs, sizeof(float));

memcpy(except\_uncertainty\_lost, l.delta, l.batch \* l.outputs \* sizeof(float));

for (b = 0; b < l.batch; ++b) {

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

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

for (n = 0; n < l.n; ++n) {

int box\_index = entry\_gaussian\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 0);

except\_uncertainty\_lost[box\_index + 4 \* stride] = 0;

except\_uncertainty\_lost[box\_index + 5 \* stride] = 0;

except\_uncertainty\_lost[box\_index + 6 \* stride] = 0;

except\_uncertainty\_lost[box\_index + 7 \* stride] = 0;

}

}

}

}

float except\_uc\_loss = pow(mag\_array(except\_uncertainty\_lost, l.outputs \* l.batch), 2);

free(except\_uncertainty\_lost);

\*(l.cost) = pow(mag\_array(l.delta, l.outputs \* l.batch), 2);

float loss = pow(mag\_array(l.delta, l.outputs \* l.batch), 2);

float uc\_loss = loss - except\_uc\_loss;

float iou\_loss = except\_uc\_loss - class\_loss;

loss /= l.batch;

class\_loss /= l.batch;

uc\_loss /= l.batch;

iou\_loss /= l.batch;

fprintf(stderr, "Region %d Avg IOU: %f, Class: %f, Obj: %f, No Obj: %f, .5R: %f, .75R: %f, count: %d, class\_loss = %.2f, iou\_loss = %.2f, uc\_loss = %.2f, total\_loss = %.2f \n",

state.index, avg\_iou/count, avg\_cat/class\_count, avg\_obj/count, avg\_anyobj/(l.w\*l.h\*l.n\*l.batch), recall/count, recall75/count, count,

class\_loss, iou\_loss, uc\_loss, loss);

}

void backward\_gaussian\_yolo\_layer(const layer l, network\_state state)

{

axpy\_cpu(l.batch\*l.inputs, 1, l.delta, 1, state.delta, 1);

}

void correct\_gaussian\_yolo\_boxes(detection \*dets, int n, int w, int h, int netw, int neth, int relative, int letter)

{

int i;

int new\_w=0;

int new\_h=0;

if (letter) {

if (((float)netw / w) < ((float)neth / h)) {

new\_w = netw;

new\_h = (h \* netw) / w;

}

else {

new\_h = neth;

new\_w = (w \* neth) / h;

}

}

else {

new\_w = netw;

new\_h = neth;

}

/\*

if (((float)netw/w) < ((float)neth/h)) {

new\_w = netw;

new\_h = (h \* netw)/w;

} else {

new\_h = neth;

new\_w = (w \* neth)/h;

}

\*/

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

box b = dets[i].bbox;

b.x = (b.x - (netw - new\_w)/2./netw) / ((float)new\_w/netw);

b.y = (b.y - (neth - new\_h)/2./neth) / ((float)new\_h/neth);

b.w \*= (float)netw/new\_w;

b.h \*= (float)neth/new\_h;

if(!relative){

b.x \*= w;

b.w \*= w;

b.y \*= h;

b.h \*= h;

}

dets[i].bbox = b;

}

}

int gaussian\_yolo\_num\_detections(layer l, float thresh)

{

int i, n;

int count = 0;

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

for(n = 0; n < l.n; ++n){

int obj\_index = entry\_gaussian\_index(l, 0, n\*l.w\*l.h + i, 8);

if(l.output[obj\_index] > thresh){

++count;

}

}

}

return count;

}

/\*

void avg\_flipped\_gaussian\_yolo(layer l)

{

int i,j,n,z;

float \*flip = l.output + l.outputs;

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

for (i = 0; i < l.w/2; ++i) {

for (n = 0; n < l.n; ++n) {

for(z = 0; z < l.classes + 8 + 1; ++z){

int i1 = z\*l.w\*l.h\*l.n + n\*l.w\*l.h + j\*l.w + i;

int i2 = z\*l.w\*l.h\*l.n + n\*l.w\*l.h + j\*l.w + (l.w - i - 1);

float swap = flip[i1];

flip[i1] = flip[i2];

flip[i2] = swap;

if(z == 0){

flip[i1] = -flip[i1];

flip[i2] = -flip[i2];

}

}

}

}

}

for(i = 0; i < l.outputs; ++i){

l.output[i] = (l.output[i] + flip[i])/2.;

}

}

\*/

int get\_gaussian\_yolo\_detections(layer l, int w, int h, int netw, int neth, float thresh, int \*map, int relative, detection \*dets, int letter)

{

int i,j,n;

float \*predictions = l.output;

//if (l.batch == 2) avg\_flipped\_gaussian\_yolo(l);

int count = 0;

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

int row = i / l.w;

int col = i % l.w;

for(n = 0; n < l.n; ++n){

int obj\_index = entry\_gaussian\_index(l, 0, n\*l.w\*l.h + i, 8);

float objectness = predictions[obj\_index];

if (objectness <= thresh) continue; // incorrect behavior for Nan values

if (objectness > thresh) {

int box\_index = entry\_gaussian\_index(l, 0, n\*l.w\*l.h + i, 0);

dets[count].bbox = get\_gaussian\_yolo\_box(predictions, l.biases, l.mask[n], box\_index, col, row, l.w, l.h, netw, neth, l.w\*l.h, l.yolo\_point);

dets[count].objectness = objectness;

dets[count].classes = l.classes;

dets[count].uc[0] = predictions[entry\_gaussian\_index(l, 0, n\*l.w\*l.h + i, 1)]; // tx uncertainty

dets[count].uc[1] = predictions[entry\_gaussian\_index(l, 0, n\*l.w\*l.h + i, 3)]; // ty uncertainty

dets[count].uc[2] = predictions[entry\_gaussian\_index(l, 0, n\*l.w\*l.h + i, 5)]; // tw uncertainty

dets[count].uc[3] = predictions[entry\_gaussian\_index(l, 0, n\*l.w\*l.h + i, 7)]; // th uncertainty

dets[count].points = l.yolo\_point;

//if (l.yolo\_point != YOLO\_CENTER) dets[count].objectness = objectness = 0;

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

int class\_index = entry\_gaussian\_index(l, 0, n\*l.w\*l.h + i, 9 + j);

float uc\_aver = (dets[count].uc[0] + dets[count].uc[1] + dets[count].uc[2] + dets[count].uc[3]) / 4.0;

float prob = objectness\*predictions[class\_index] \* (1.0 - uc\_aver);

dets[count].prob[j] = (prob > thresh) ? prob : 0;

}

++count;

}

}

}

correct\_gaussian\_yolo\_boxes(dets, count, w, h, netw, neth, relative, letter);

return count;

}

#ifdef GPU

void forward\_gaussian\_yolo\_layer\_gpu(const layer l, network\_state state)

{

copy\_ongpu(l.batch\*l.inputs, state.input, 1, l.output\_gpu, 1);

int b, n;

for (b = 0; b < l.batch; ++b)

{

for(n = 0; n < l.n; ++n)

{

// x : mu, sigma

int index = entry\_gaussian\_index(l, b, n\*l.w\*l.h, 0);

activate\_array\_ongpu(l.output\_gpu + index, 2\*l.w\*l.h, LOGISTIC);

scal\_add\_ongpu(l.w\*l.h, l.scale\_x\_y, -0.5\*(l.scale\_x\_y - 1), l.output\_gpu + index, 1); // scale x

// y : mu, sigma

index = entry\_gaussian\_index(l, b, n\*l.w\*l.h, 2);

activate\_array\_ongpu(l.output\_gpu + index, 2\*l.w\*l.h, LOGISTIC);

scal\_add\_ongpu(l.w\*l.h, l.scale\_x\_y, -0.5\*(l.scale\_x\_y - 1), l.output\_gpu + index, 1); // scale y

// w : sigma

index = entry\_gaussian\_index(l, b, n\*l.w\*l.h, 5);

activate\_array\_ongpu(l.output\_gpu + index, l.w\*l.h, LOGISTIC);

// h : sigma

index = entry\_gaussian\_index(l, b, n\*l.w\*l.h, 7);

activate\_array\_ongpu(l.output\_gpu + index, l.w\*l.h, LOGISTIC);

// objectness & class

index = entry\_gaussian\_index(l, b, n\*l.w\*l.h, 8);

activate\_array\_ongpu(l.output\_gpu + index, (1+l.classes)\*l.w\*l.h, LOGISTIC);

}

}

if (!state.train || l.onlyforward) {

//cuda\_pull\_array(l.output\_gpu, l.output, l.batch\*l.outputs);

cuda\_pull\_array\_async(l.output\_gpu, l.output, l.batch\*l.outputs);

CHECK\_CUDA(cudaPeekAtLastError());

return;

}

float \*in\_cpu = (float \*)calloc(l.batch\*l.inputs, sizeof(float));

cuda\_pull\_array(l.output\_gpu, l.output, l.batch\*l.outputs);

memcpy(in\_cpu, l.output, l.batch\*l.outputs \* sizeof(float));

float \*truth\_cpu = 0;

if (state.truth) {

int num\_truth = l.batch\*l.truths;

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

cuda\_pull\_array(state.truth, truth\_cpu, num\_truth);

}

network\_state cpu\_state = state;

cpu\_state.net = state.net;

cpu\_state.index = state.index;

cpu\_state.train = state.train;

cpu\_state.truth = truth\_cpu;

cpu\_state.input = in\_cpu;

forward\_gaussian\_yolo\_layer(l, cpu\_state);

//forward\_yolo\_layer(l, state);

cuda\_push\_array(l.delta\_gpu, l.delta, l.batch\*l.outputs);

free(in\_cpu);

if (cpu\_state.truth) free(cpu\_state.truth);

}

void backward\_gaussian\_yolo\_layer\_gpu(const layer l, network\_state state)

{

axpy\_ongpu(l.batch\*l.inputs, 1, l.delta\_gpu, 1, state.delta, 1);

}

#endif