#include "yolo\_layer.h"

#include "activations.h"

#include "blas.h"

#include "box.h"

#include "dark\_cuda.h"

#include "utils.h"

#include <math.h>

#include <stdio.h>

#include <assert.h>

#include <string.h>

#include <stdlib.h>

extern int check\_mistakes;

layer make\_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 = YOLO;

l.n = n;

l.total = total;

l.batch = batch;

l.h = h;

l.w = w;

l.c = n\*(classes + 4 + 1);

l.out\_w = l.w;

l.out\_h = l.h;

l.out\_c = l.c;

l.classes = classes;

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

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

if(mask) l.mask = mask;

else{

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

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

l.mask[i] = i;

}

}

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

l.outputs = h\*w\*n\*(classes + 4 + 1);

l.inputs = l.outputs;

l.max\_boxes = max\_boxes;

l.truths = l.max\_boxes\*(4 + 1); // 90\*(4 + 1);

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

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

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

l.biases[i] = .5;

}

l.forward = forward\_yolo\_layer;

l.backward = backward\_yolo\_layer;

#ifdef GPU

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

l.backward\_gpu = backward\_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\*)xcalloc(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\*)xcalloc(batch \* l.outputs, sizeof(float));

}

#endif

fprintf(stderr, "yolo\n");

srand(time(0));

return l;

}

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

{

l->w = w;

l->h = h;

l->outputs = h\*w\*l->n\*(l->classes + 4 + 1);

l->inputs = l->outputs;

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

if (!l->delta\_pinned) l->delta = (float\*)xrealloc(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\*)xcalloc(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\*)xcalloc(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\_yolo\_box(float \*x, float \*biases, int n, int index, int i, int j, int lw, int lh, int w, int h, int stride)

{

box b;

// ln - natural logarithm (base = e)

// x` = t.x \* lw - i; // x = ln(x`/(1-x`)) // x - output of previous conv-layer

// y` = t.y \* lh - i; // y = ln(y`/(1-y`)) // y - output of previous conv-layer

// w = ln(t.w \* net.w / anchors\_w); // w - output of previous conv-layer

// h = ln(t.h \* net.h / anchors\_h); // h - output of previous conv-layer

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

b.y = (j + x[index + 1\*stride]) / lh;

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

b.h = exp(x[index + 3\*stride]) \* biases[2\*n+1] / h;

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) {

//printf("\n val = %f > max\_val = %f \n", val, max\_val);

val = max\_val;

}

else if (val < -max\_val) {

//printf("\n val = %f < -max\_val = %f \n", val, -max\_val);

val = -max\_val;

}

return val;

}

ious delta\_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, int accumulate, float max\_delta)

{

ious all\_ious = { 0 };

// i - step in layer width

// j - step in layer height

// Returns a box in absolute coordinates

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

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);

// avoid nan in dx\_box\_iou

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

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

if (iou\_loss == MSE) // old loss

{

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

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

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

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

//printf(" tx = %f, ty = %f, tw = %f, th = %f \n", tx, ty, tw, th);

//printf(" x = %f, y = %f, w = %f, h = %f \n", x[index + 0 \* stride], x[index + 1 \* stride], x[index + 2 \* stride], x[index + 3 \* stride]);

// accumulate delta

delta[index + 0 \* stride] += scale \* (tx - x[index + 0 \* stride]) \* iou\_normalizer;

delta[index + 1 \* stride] += scale \* (ty - x[index + 1 \* stride]) \* iou\_normalizer;

delta[index + 2 \* stride] += scale \* (tw - x[index + 2 \* stride]) \* iou\_normalizer;

delta[index + 3 \* stride] += scale \* (th - x[index + 3 \* stride]) \* iou\_normalizer;

}

else {

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

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

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

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

// 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));

// jacobian^t (transpose)

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

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

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

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

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

dw \*= exp(x[index + 2 \* stride]);

dh \*= exp(x[index + 3 \* stride]);

// normalize iou weight

dx \*= iou\_normalizer;

dy \*= iou\_normalizer;

dw \*= iou\_normalizer;

dh \*= iou\_normalizer;

dx = fix\_nan\_inf(dx);

dy = fix\_nan\_inf(dy);

dw = fix\_nan\_inf(dw);

dh = fix\_nan\_inf(dh);

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

dx = clip\_value(dx, max\_delta);

dy = clip\_value(dy, max\_delta);

dw = clip\_value(dw, max\_delta);

dh = clip\_value(dh, max\_delta);

}

if (!accumulate) {

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

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

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

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

}

// accumulate delta

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

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

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

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

}

return all\_ious;

}

void averages\_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;

}

}

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

{

int n;

if (delta[index + stride\*class\_id]){

float y\_true = 1;

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

float result\_delta = y\_true - output[index + stride\*class\_id];

if(!isnan(result\_delta) && !isinf(result\_delta)) delta[index + stride\*class\_id] = result\_delta;

//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;

}

// Focal loss

if (focal\_loss) {

// Focal Loss

float alpha = 0.5; // 0.25 or 0.5

//float gamma = 2; // hardcoded in many places of the grad-formula

int ti = index + stride\*class\_id;

float pt = output[ti] + 0.000000000000001F;

// http://fooplot.com/#W3sidHlwZSI6MCwiZXEiOiItKDEteCkqKDIqeCpsb2coeCkreC0xKSIsImNvbG9yIjoiIzAwMDAwMCJ9LHsidHlwZSI6MTAwMH1d

float grad = -(1 - pt) \* (2 \* pt\*logf(pt) + pt - 1); // http://blog.csdn.net/linmingan/article/details/77885832

//float grad = (1 - pt) \* (2 \* pt\*logf(pt) + pt - 1); // https://github.com/unsky/focal-loss

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

delta[index + stride\*n] = (((n == class\_id) ? 1 : 0) - output[index + stride\*n]);

delta[index + stride\*n] \*= alpha\*grad;

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

}

}

else {

// default

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;

float result\_delta = y\_true - output[index + stride\*n];

if (!isnan(result\_delta) && !isinf(result\_delta)) delta[index + stride\*n] = result\_delta;

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\_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\_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\*(4+l.classes+1) + entry\*l.w\*l.h + loc;

}

void forward\_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) {

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

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

scal\_add\_cpu(2 \* l.w\*l.h, l.scale\_x\_y, -0.5\*(l.scale\_x\_y - 1), l.output + index, 1); // scale x,y

index = entry\_index(l, b, n\*l.w\*l.h, 4);

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

}

}

#endif

// delta is zeroed

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

if (!state.train) return;

//float avg\_iou = 0;

float tot\_iou = 0;

float tot\_giou = 0;

float tot\_diou = 0;

float tot\_ciou = 0;

float tot\_iou\_loss = 0;

float tot\_giou\_loss = 0;

float tot\_diou\_loss = 0;

float tot\_ciou\_loss = 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\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 0);

box pred = get\_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);

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 || class\_id < 0) {

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("\n 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 and class\_id check garbage value

}

if (!truth.x) break; // continue;

int class\_index = entry\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 4 + 1);

int obj\_index = entry\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 4);

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

if (isnan(objectness) || isinf(objectness)) l.output[obj\_index] = 0;

int class\_id\_match = compare\_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\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 4);

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\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 4 + 1);

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\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 4 + 1);

delta\_yolo\_class(l.output, l.delta, class\_index, class\_id, l.classes, l.w\*l.h, 0, l.focal\_loss, 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\_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, 1, 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 < 0 || truth.y < 0 || truth.x > 1 || truth.y > 1 || truth.w < 0 || truth.h < 0) {

char buff[256];

printf(" Wrong label: truth.x = %f, truth.y = %f, truth.w = %f, truth.h = %f \n", truth.x, truth.y, truth.w, truth.h);

sprintf(buff, "echo \"Wrong label: truth.x = %f, truth.y = %f, truth.w = %f, truth.h = %f\" >> bad\_label.list",

truth.x, truth.y, truth.w, truth.h);

system(buff);

}

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

if (class\_id >= l.classes || class\_id < 0) continue; // if label contains class\_id more than number of classes in the cfg-file and class\_id check garbage value

if (!truth.x) break; // continue;

float best\_iou = 0;

int best\_n = 0;

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

j = (truth.y \* 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\_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;

ious all\_ious = delta\_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, 1, l.max\_delta);

// range is 0 <= 1

tot\_iou += all\_ious.iou;

tot\_iou\_loss += 1 - all\_ious.iou;

// range is -1 <= giou <= 1

tot\_giou += all\_ious.giou;

tot\_giou\_loss += 1 - all\_ious.giou;

tot\_diou += all\_ious.diou;

tot\_diou\_loss += 1 - all\_ious.diou;

tot\_ciou += all\_ious.ciou;

tot\_ciou\_loss += 1 - all\_ious.ciou;

int obj\_index = entry\_index(l, b, mask\_n\*l.w\*l.h + j\*l.w + i, 4);

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

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

int class\_index = entry\_index(l, b, mask\_n\*l.w\*l.h + j\*l.w + i, 4 + 1);

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

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

//printf(" mask\_n = %d, l.output[obj\_index] = %f, l.output[class\_index + class\_id] = %f \n\n", mask\_n, l.output[obj\_index], l.output[class\_index + class\_id]);

++count;

++class\_count;

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

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

}

// 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\_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;

ious all\_ious = delta\_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, 1, l.max\_delta);

// range is 0 <= 1

tot\_iou += all\_ious.iou;

tot\_iou\_loss += 1 - all\_ious.iou;

// range is -1 <= giou <= 1

tot\_giou += all\_ious.giou;

tot\_giou\_loss += 1 - all\_ious.giou;

tot\_diou += all\_ious.diou;

tot\_diou\_loss += 1 - all\_ious.diou;

tot\_ciou += all\_ious.ciou;

tot\_ciou\_loss += 1 - all\_ious.ciou;

int obj\_index = entry\_index(l, b, mask\_n\*l.w\*l.h + j\*l.w + i, 4);

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

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

int class\_index = entry\_index(l, b, mask\_n\*l.w\*l.h + j\*l.w + i, 4 + 1);

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

++count;

++class\_count;

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

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

}

}

}

}

// 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\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 0);

int class\_index = entry\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 4 + 1);

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

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

}

}

}

}

if (count == 0) count = 1;

if (class\_count == 0) class\_count = 1;

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

//printf("Region %d Avg IOU: %f, Class: %f, Obj: %f, No Obj: %f, .5R: %f, .75R: %f, count: %d\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);

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

float\* no\_iou\_loss\_delta = (float \*)calloc(l.batch \* l.outputs, sizeof(float));

memcpy(no\_iou\_loss\_delta, 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 index = entry\_index(l, b, n\*l.w\*l.h + j\*l.w + i, 0);

no\_iou\_loss\_delta[index + 0 \* stride] = 0;

no\_iou\_loss\_delta[index + 1 \* stride] = 0;

no\_iou\_loss\_delta[index + 2 \* stride] = 0;

no\_iou\_loss\_delta[index + 3 \* stride] = 0;

}

}

}

}

float classification\_loss = l.cls\_normalizer \* pow(mag\_array(no\_iou\_loss\_delta, l.outputs \* l.batch), 2);

free(no\_iou\_loss\_delta);

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

float iou\_loss = loss - classification\_loss;

float avg\_iou\_loss = 0;

// gIOU loss + MSE (objectness) loss

if (l.iou\_loss == MSE) {

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

}

else {

// Always compute classification loss both for iou + cls loss and for logging with mse loss

// TODO: remove IOU loss fields before computing MSE on class

// probably split into two arrays

if (l.iou\_loss == GIOU) {

avg\_iou\_loss = count > 0 ? l.iou\_normalizer \* (tot\_giou\_loss / count) : 0;

}

else {

avg\_iou\_loss = count > 0 ? l.iou\_normalizer \* (tot\_iou\_loss / count) : 0;

}

\*(l.cost) = avg\_iou\_loss + classification\_loss;

}

loss /= l.batch;

classification\_loss /= l.batch;

iou\_loss /= l.batch;

fprintf(stderr, "v3 (%s loss, Normalizer: (iou: %.2f, cls: %.2f) Region %d Avg (IOU: %f, GIOU: %f), Class: %f, Obj: %f, No Obj: %f, .5R: %f, .75R: %f, count: %d, class\_loss = %f, iou\_loss = %f, total\_loss = %f \n",

(l.iou\_loss == MSE ? "mse" : (l.iou\_loss == GIOU ? "giou" : "iou")), l.iou\_normalizer, l.cls\_normalizer, state.index, tot\_iou / count, tot\_giou / count, avg\_cat / class\_count, avg\_obj / count, avg\_anyobj / (l.w\*l.h\*l.n\*l.batch), recall / count, recall75 / count, count,

classification\_loss, iou\_loss, loss);

}

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

{

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

}

// Converts output of the network to detection boxes

// w,h: image width,height

// netw,neth: network width,height

// relative: 1 (all callers seems to pass TRUE)

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

{

int i;

// network height (or width)

int new\_w = 0;

// network height (or width)

int new\_h = 0;

// Compute scale given image w,h vs network w,h

// I think this "rotates" the image to match network to input image w/h ratio

// new\_h and new\_w are really just network width and height

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;

}

// difference between network width and "rotated" width

float deltaw = netw - new\_w;

// difference between network height and "rotated" height

float deltah = neth - new\_h;

// ratio between rotated network width and network width

float ratiow = (float)new\_w / netw;

// ratio between rotated network width and network width

float ratioh = (float)new\_h / neth;

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

box b = dets[i].bbox;

// x = ( x - (deltaw/2)/netw ) / ratiow;

// x - [(1/2 the difference of the network width and rotated width) / (network width)]

b.x = (b.x - deltaw / 2. / netw) / ratiow;

b.y = (b.y - deltah / 2. / neth) / ratioh;

// scale to match rotation of incoming image

b.w \*= 1 / ratiow;

b.h \*= 1 / ratioh;

// relative seems to always be == 1, I don't think we hit this condition, ever.

if (!relative) {

b.x \*= w;

b.w \*= w;

b.y \*= h;

b.h \*= h;

}

dets[i].bbox = b;

}

}

/\*

void correct\_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;

}

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 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\_index(l, 0, n\*l.w\*l.h + i, 4);

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

++count;

}

}

}

return count;

}

int yolo\_num\_detections\_batch(layer l, float thresh, int batch)

{

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\_index(l, batch, n\*l.w\*l.h + i, 4);

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

++count;

}

}

}

return count;

}

void avg\_flipped\_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 + 4 + 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\_yolo\_detections(layer l, int w, int h, int netw, int neth, float thresh, int \*map, int relative, detection \*dets, int letter)

{

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

int i,j,n;

float \*predictions = l.output;

// This snippet below is not necessary

// Need to comment it in order to batch processing >= 2 images

//if (l.batch == 2) avg\_flipped\_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\_index(l, 0, n\*l.w\*l.h + i, 4);

float objectness = predictions[obj\_index];

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

if (objectness > thresh) {

//printf("\n objectness = %f, thresh = %f, i = %d, n = %d \n", objectness, thresh, i, n);

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

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

dets[count].objectness = objectness;

dets[count].classes = l.classes;

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

int class\_index = entry\_index(l, 0, n\*l.w\*l.h + i, 4 + 1 + j);

float prob = objectness\*predictions[class\_index];

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

}

++count;

}

}

}

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

return count;

}

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

{

int i,j,n;

float \*predictions = l.output;

//if (l.batch == 2) avg\_flipped\_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\_index(l, batch, n\*l.w\*l.h + i, 4);

float objectness = predictions[obj\_index];

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

if (objectness > thresh) {

//printf("\n objectness = %f, thresh = %f, i = %d, n = %d \n", objectness, thresh, i, n);

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

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

dets[count].objectness = objectness;

dets[count].classes = l.classes;

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

int class\_index = entry\_index(l, batch, n\*l.w\*l.h + i, 4 + 1 + j);

float prob = objectness\*predictions[class\_index];

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

}

++count;

}

}

}

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

return count;

}

#ifdef GPU

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

{

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

simple\_copy\_ongpu(l.batch\*l.inputs, state.input, l.output\_gpu);

int b, n;

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

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

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

// y = 1./(1. + exp(-x))

// x = ln(y/(1-y)) // ln - natural logarithm (base = e)

// if(y->1) x -> inf

// if(y->0) x -> -inf

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

if (l.scale\_x\_y != 1) scal\_add\_ongpu(2 \* l.w\*l.h, l.scale\_x\_y, -0.5\*(l.scale\_x\_y - 1), l.output\_gpu + index, 1); // scale x,y

index = entry\_index(l, b, n\*l.w\*l.h, 4);

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

}

}

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 \*)xcalloc(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 \*)xcalloc(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\_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\_yolo\_layer\_gpu(const layer l, network\_state state)

{

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

}

#endif