#include "region\_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>

#define DOABS 1

region\_layer make\_region\_layer(int batch, int w, int h, int n, int classes, int coords, int max\_boxes)

{

region\_layer l = { (LAYER\_TYPE)0 };

l.type = REGION;

l.n = n;

l.batch = batch;

l.h = h;

l.w = w;

l.classes = classes;

l.coords = coords;

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

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

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

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

l.inputs = l.outputs;

l.max\_boxes = max\_boxes;

l.truths = max\_boxes\*(5);

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

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

int i;

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

l.biases[i] = .5;

}

l.forward = forward\_region\_layer;

l.backward = backward\_region\_layer;

#ifdef GPU

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

l.backward\_gpu = backward\_region\_layer\_gpu;

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

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

#endif

fprintf(stderr, "detection\n");

srand(time(0));

return l;

}

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

{

#ifdef GPU

int old\_w = l->w;

int old\_h = l->h;

#endif

l->w = w;

l->h = h;

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

l->inputs = l->outputs;

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

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

#ifdef GPU

//if (old\_w < w || old\_h < h)

{

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\_region\_box(float \*x, float \*biases, int n, int index, int i, int j, int w, int h)

{

box b;

b.x = (i + logistic\_activate(x[index + 0])) / w;

b.y = (j + logistic\_activate(x[index + 1])) / h;

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

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

if(DOABS){

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

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

}

return b;

}

float delta\_region\_box(box truth, float \*x, float \*biases, int n, int index, int i, int j, int w, int h, float \*delta, float scale)

{

box pred = get\_region\_box(x, biases, n, index, i, j, w, h);

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

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

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

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

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

if(DOABS){

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

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

}

delta[index + 0] = scale \* (tx - logistic\_activate(x[index + 0])) \* logistic\_gradient(logistic\_activate(x[index + 0]));

delta[index + 1] = scale \* (ty - logistic\_activate(x[index + 1])) \* logistic\_gradient(logistic\_activate(x[index + 1]));

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

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

return iou;

}

void delta\_region\_class(float \*output, float \*delta, int index, int class\_id, int classes, tree \*hier, float scale, float \*avg\_cat, int focal\_loss)

{

int i, n;

if(hier){

float pred = 1;

while(class\_id >= 0){

pred \*= output[index + class\_id];

int g = hier->group[class\_id];

int offset = hier->group\_offset[g];

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

delta[index + offset + i] = scale \* (0 - output[index + offset + i]);

}

delta[index + class\_id] = scale \* (1 - output[index + class\_id]);

class\_id = hier->parent[class\_id];

}

\*avg\_cat += pred;

} else {

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

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

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

}

}

else {

// default

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

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

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

}

}

}

}

float logit(float x)

{

return log(x/(1.-x));

}

float tisnan(float x)

{

return (x != x);

}

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

}

void softmax\_tree(float \*input, int batch, int inputs, float temp, tree \*hierarchy, float \*output);

void forward\_region\_layer(const region\_layer l, network\_state state)

{

int i,j,b,t,n;

int size = l.coords + l.classes + 1;

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

#ifndef GPU

flatten(l.output, l.w\*l.h, size\*l.n, l.batch, 1);

#endif

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

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

int index = size\*i + b\*l.outputs;

l.output[index + 4] = logistic\_activate(l.output[index + 4]);

}

}

#ifndef GPU

if (l.softmax\_tree){

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

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

int index = size\*i + b\*l.outputs;

softmax\_tree(l.output + index + 5, 1, 0, 1, l.softmax\_tree, l.output + index + 5);

}

}

} else if (l.softmax){

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

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

int index = size\*i + b\*l.outputs;

softmax(l.output + index + 5, l.classes, 1, l.output + index + 5, 1);

}

}

}

#endif

if(!state.train) return;

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

float avg\_iou = 0;

float recall = 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) {

if(l.softmax\_tree){

int onlyclass\_id = 0;

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

box truth = float\_to\_box(state.truth + t\*5 + b\*l.truths);

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

int class\_id = state.truth[t\*5 + b\*l.truths + 4];

float maxp = 0;

int maxi = 0;

if(truth.x > 100000 && truth.y > 100000){

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

int index = size\*n + b\*l.outputs + 5;

float scale = l.output[index-1];

float p = scale\*get\_hierarchy\_probability(l.output + index, l.softmax\_tree, class\_id);

if(p > maxp){

maxp = p;

maxi = n;

}

}

int index = size\*maxi + b\*l.outputs + 5;

delta\_region\_class(l.output, l.delta, index, class\_id, l.classes, l.softmax\_tree, l.class\_scale, &avg\_cat, l.focal\_loss);

++class\_count;

onlyclass\_id = 1;

break;

}

}

if(onlyclass\_id) continue;

}

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

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

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

int index = size\*(j\*l.w\*l.n + i\*l.n + n) + b\*l.outputs;

box pred = get\_region\_box(l.output, l.biases, n, index, i, j, l.w, l.h);

float best\_iou = 0;

int best\_class\_id = -1;

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

box truth = float\_to\_box(state.truth + t\*5 + b\*l.truths);

int class\_id = state.truth[t \* 5 + b\*l.truths + 4];

if (class\_id >= l.classes) continue; // if label contains class\_id more than number of classes in the cfg-file

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

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

if (iou > best\_iou) {

best\_class\_id = state.truth[t\*5 + b\*l.truths + 4];

best\_iou = iou;

}

}

avg\_anyobj += l.output[index + 4];

l.delta[index + 4] = l.noobject\_scale \* ((0 - l.output[index + 4]) \* logistic\_gradient(l.output[index + 4]));

if(l.classfix == -1) l.delta[index + 4] = l.noobject\_scale \* ((best\_iou - l.output[index + 4]) \* logistic\_gradient(l.output[index + 4]));

else{

if (best\_iou > l.thresh) {

l.delta[index + 4] = 0;

if(l.classfix > 0){

delta\_region\_class(l.output, l.delta, index + 5, best\_class\_id, l.classes, l.softmax\_tree, l.class\_scale\*(l.classfix == 2 ? l.output[index + 4] : 1), &avg\_cat, l.focal\_loss);

++class\_count;

}

}

}

if(\*(state.net.seen) < 12800){

box truth = {0};

truth.x = (i + .5)/l.w;

truth.y = (j + .5)/l.h;

truth.w = l.biases[2\*n];

truth.h = l.biases[2\*n+1];

if(DOABS){

truth.w = l.biases[2\*n]/l.w;

truth.h = l.biases[2\*n+1]/l.h;

}

delta\_region\_box(truth, l.output, l.biases, n, index, i, j, l.w, l.h, l.delta, .01);

}

}

}

}

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

box truth = float\_to\_box(state.truth + t\*5 + b\*l.truths);

int class\_id = state.truth[t \* 5 + 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);

getchar();

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

}

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

float best\_iou = 0;

int best\_index = 0;

int best\_n = 0;

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

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

//printf("%d %f %d %f\n", i, truth.x\*l.w, j, truth.y\*l.h);

box truth\_shift = truth;

truth\_shift.x = 0;

truth\_shift.y = 0;

//printf("index %d %d\n",i, j);

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

int index = size\*(j\*l.w\*l.n + i\*l.n + n) + b\*l.outputs;

box pred = get\_region\_box(l.output, l.biases, n, index, i, j, l.w, l.h);

if(l.bias\_match){

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

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

if(DOABS){

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

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

}

}

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

pred.x = 0;

pred.y = 0;

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

if (iou > best\_iou){

best\_index = index;

best\_iou = iou;

best\_n = n;

}

}

//printf("%d %f (%f, %f) %f x %f\n", best\_n, best\_iou, truth.x, truth.y, truth.w, truth.h);

float iou = delta\_region\_box(truth, l.output, l.biases, best\_n, best\_index, i, j, l.w, l.h, l.delta, l.coord\_scale);

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

avg\_iou += iou;

//l.delta[best\_index + 4] = iou - l.output[best\_index + 4];

avg\_obj += l.output[best\_index + 4];

l.delta[best\_index + 4] = l.object\_scale \* (1 - l.output[best\_index + 4]) \* logistic\_gradient(l.output[best\_index + 4]);

if (l.rescore) {

l.delta[best\_index + 4] = l.object\_scale \* (iou - l.output[best\_index + 4]) \* logistic\_gradient(l.output[best\_index + 4]);

}

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

delta\_region\_class(l.output, l.delta, best\_index + 5, class\_id, l.classes, l.softmax\_tree, l.class\_scale, &avg\_cat, l.focal\_loss);

++count;

++class\_count;

}

}

//printf("\n");

#ifndef GPU

flatten(l.delta, l.w\*l.h, size\*l.n, l.batch, 0);

#endif

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

printf("Region Avg IOU: %f, Class: %f, Obj: %f, No Obj: %f, Avg Recall: %f, count: %d\n", avg\_iou/count, avg\_cat/class\_count, avg\_obj/count, avg\_anyobj/(l.w\*l.h\*l.n\*l.batch), recall/count, count);

}

void backward\_region\_layer(const region\_layer l, network\_state state)

{

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

}

void get\_region\_boxes(layer l, int w, int h, float thresh, float \*\*probs, box \*boxes, int only\_objectness, int \*map)

{

int i;

float \*const predictions = l.output;

#pragma omp parallel for

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

int j, n;

int row = i / l.w;

int col = i % l.w;

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

int index = i\*l.n + n;

int p\_index = index \* (l.classes + 5) + 4;

float scale = predictions[p\_index];

if(l.classfix == -1 && scale < .5) scale = 0;

int box\_index = index \* (l.classes + 5);

boxes[index] = get\_region\_box(predictions, l.biases, n, box\_index, col, row, l.w, l.h);

boxes[index].x \*= w;

boxes[index].y \*= h;

boxes[index].w \*= w;

boxes[index].h \*= h;

int class\_index = index \* (l.classes + 5) + 5;

if(l.softmax\_tree){

hierarchy\_predictions(predictions + class\_index, l.classes, l.softmax\_tree, 0);

int found = 0;

if(map){

for(j = 0; j < 200; ++j){

float prob = scale\*predictions[class\_index+map[j]];

probs[index][j] = (prob > thresh) ? prob : 0;

}

} else {

for(j = l.classes - 1; j >= 0; --j){

if(!found && predictions[class\_index + j] > .5){

found = 1;

} else {

predictions[class\_index + j] = 0;

}

float prob = predictions[class\_index+j];

probs[index][j] = (scale > thresh) ? prob : 0;

}

}

} else {

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

float prob = scale\*predictions[class\_index+j];

probs[index][j] = (prob > thresh) ? prob : 0;

}

}

if(only\_objectness){

probs[index][0] = scale;

}

}

}

}

#ifdef GPU

void forward\_region\_layer\_gpu(const region\_layer l, network\_state state)

{

/\*

if(!state.train){

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

return;

}

\*/

flatten\_ongpu(state.input, l.h\*l.w, l.n\*(l.coords + l.classes + 1), l.batch, 1, l.output\_gpu);

if(l.softmax\_tree){

int i;

int count = 5;

for (i = 0; i < l.softmax\_tree->groups; ++i) {

int group\_size = l.softmax\_tree->group\_size[i];

softmax\_gpu(l.output\_gpu+count, group\_size, l.classes + 5, l.w\*l.h\*l.n\*l.batch, 1, l.output\_gpu + count);

count += group\_size;

}

}else if (l.softmax){

softmax\_gpu(l.output\_gpu+5, l.classes, l.classes + 5, l.w\*l.h\*l.n\*l.batch, 1, l.output\_gpu + 5);

}

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

}

cuda\_pull\_array(l.output\_gpu, in\_cpu, l.batch\*l.inputs);

//cudaStreamSynchronize(get\_cuda\_stream());

network\_state cpu\_state = state;

cpu\_state.train = state.train;

cpu\_state.truth = truth\_cpu;

cpu\_state.input = in\_cpu;

forward\_region\_layer(l, cpu\_state);

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

free(cpu\_state.input);

if(!state.train) return;

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

//cudaStreamSynchronize(get\_cuda\_stream());

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

}

void backward\_region\_layer\_gpu(region\_layer l, network\_state state)

{

flatten\_ongpu(l.delta\_gpu, l.h\*l.w, l.n\*(l.coords + l.classes + 1), l.batch, 0, state.delta);

}

#endif

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

{

int i;

int new\_w = 0;

int new\_h = 0;

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;

}

}

void get\_region\_detections(layer l, int w, int h, int netw, int neth, float thresh, int \*map, float tree\_thresh, int relative, detection \*dets)

{

int i, j, n, z;

float \*predictions = l.output;

if (l.batch == 2) {

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 + l.coords + 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.;

}

}

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 index = n\*l.w\*l.h + i;

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

dets[index].prob[j] = 0;

}

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

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

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

float scale = l.background ? 1 : predictions[obj\_index];

dets[index].bbox = get\_region\_box(predictions, l.biases, n, box\_index, col, row, l.w, l.h);// , l.w\*l.h);

dets[index].objectness = scale > thresh ? scale : 0;

if (dets[index].mask) {

for (j = 0; j < l.coords - 4; ++j) {

dets[index].mask[j] = l.output[mask\_index + j\*l.w\*l.h];

}

}

int class\_index = entry\_index(l, 0, n\*l.w\*l.h + i, l.coords + !l.background);

if (l.softmax\_tree) {

hierarchy\_predictions(predictions + class\_index, l.classes, l.softmax\_tree, 0);// , l.w\*l.h);

if (map) {

for (j = 0; j < 200; ++j) {

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

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

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

}

}

else {

int j = hierarchy\_top\_prediction(predictions + class\_index, l.softmax\_tree, tree\_thresh, l.w\*l.h);

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

}

}

else {

if (dets[index].objectness) {

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

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

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

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

}

}

}

}

}

correct\_region\_boxes(dets, l.w\*l.h\*l.n, w, h, netw, neth, relative);

}

void zero\_objectness(layer l)

{

int i, n;

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, l.coords);

l.output[obj\_index] = 0;

}

}

}