#include "scale\_channels\_layer.h"

#include "utils.h"

#include "dark\_cuda.h"

#include "blas.h"

#include <stdio.h>

#include <assert.h>

layer make\_scale\_channels\_layer(int batch, int index, int w, int h, int c, int w2, int h2, int c2, int scale\_wh)

{

fprintf(stderr,"scale Layer: %d\n", index);

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

l.type = SCALE\_CHANNELS;

l.batch = batch;

l.scale\_wh = scale\_wh;

l.w = w;

l.h = h;

l.c = c;

if (!l.scale\_wh) assert(w == 1 && h == 1);

else assert(c == 1);

l.out\_w = w2;

l.out\_h = h2;

l.out\_c = c2;

if (!l.scale\_wh) assert(l.out\_c == l.c);

else assert(l.out\_w == l.w && l.out\_h == l.h);

l.outputs = l.out\_w\*l.out\_h\*l.out\_c;

l.inputs = l.outputs;

l.index = index;

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

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

l.forward = forward\_scale\_channels\_layer;

l.backward = backward\_scale\_channels\_layer;

#ifdef GPU

l.forward\_gpu = forward\_scale\_channels\_layer\_gpu;

l.backward\_gpu = backward\_scale\_channels\_layer\_gpu;

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

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

#endif

return l;

}

void resize\_scale\_channels\_layer(layer \*l, network \*net)

{

layer first = net->layers[l->index];

l->out\_w = first.out\_w;

l->out\_h = first.out\_h;

l->outputs = l->out\_w\*l->out\_h\*l->out\_c;

l->inputs = l->outputs;

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

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

#ifdef GPU

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

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

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

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

#endif

}

void forward\_scale\_channels\_layer(const layer l, network\_state state)

{

int size = l.batch \* l.out\_c \* l.out\_w \* l.out\_h;

int channel\_size = l.out\_w \* l.out\_h;

int batch\_size = l.out\_c \* l.out\_w \* l.out\_h;

float \*from\_output = state.net.layers[l.index].output;

if (l.scale\_wh) {

int i;

#pragma omp parallel for

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

int input\_index = i % channel\_size + (i / batch\_size)\*channel\_size;

l.output[i] = state.input[input\_index] \* from\_output[i];

}

}

else {

int i;

#pragma omp parallel for

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

l.output[i] = state.input[i / channel\_size] \* from\_output[i];

}

}

activate\_array(l.output, l.outputs\*l.batch, l.activation);

}

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

{

gradient\_array(l.output, l.outputs\*l.batch, l.activation, l.delta);

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

//scale\_cpu(l.batch, l.out\_w, l.out\_h, l.out\_c, l.delta, l.w, l.h, l.c, state.net.layers[l.index].delta);

int size = l.batch \* l.out\_c \* l.out\_w \* l.out\_h;

int channel\_size = l.out\_w \* l.out\_h;

int batch\_size = l.out\_c \* l.out\_w \* l.out\_h;

float \*from\_output = state.net.layers[l.index].output;

float \*from\_delta = state.net.layers[l.index].delta;

if (l.scale\_wh) {

int i;

#pragma omp parallel for

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

int input\_index = i % channel\_size + (i / batch\_size)\*channel\_size;

state.delta[input\_index] += l.delta[i] \* from\_output[i];// / l.out\_c; // l.delta \* from (should be divided by l.out\_c?)

from\_delta[i] += state.input[input\_index] \* l.delta[i]; // input \* l.delta

}

}

else {

int i;

#pragma omp parallel for

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

state.delta[i / channel\_size] += l.delta[i] \* from\_output[i];// / channel\_size; // l.delta \* from (should be divided by channel\_size?)

from\_delta[i] += state.input[i / channel\_size] \* l.delta[i]; // input \* l.delta

}

}

}

#ifdef GPU

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

{

int size = l.batch \* l.out\_c \* l.out\_w \* l.out\_h;

int channel\_size = l.out\_w \* l.out\_h;

int batch\_size = l.out\_c \* l.out\_w \* l.out\_h;

scale\_channels\_gpu(state.net.layers[l.index].output\_gpu, size, channel\_size, batch\_size, l.scale\_wh, state.input, l.output\_gpu);

activate\_array\_ongpu(l.output\_gpu, l.outputs\*l.batch, l.activation);

}

void backward\_scale\_channels\_layer\_gpu(const layer l, network\_state state)

{

gradient\_array\_ongpu(l.output\_gpu, l.outputs\*l.batch, l.activation, l.delta\_gpu);

int size = l.batch \* l.out\_c \* l.out\_w \* l.out\_h;

int channel\_size = l.out\_w \* l.out\_h;

int batch\_size = l.out\_c \* l.out\_w \* l.out\_h;

float \*from\_output = state.net.layers[l.index].output\_gpu;

float \*from\_delta = state.net.layers[l.index].delta\_gpu;

backward\_scale\_channels\_gpu(l.delta\_gpu, size, channel\_size, batch\_size, l.scale\_wh, state.input, from\_delta, from\_output, state.delta);

}

#endif