#include "softmax\_layer.h"

#include "blas.h"

#include "dark\_cuda.h"

#include "utils.h"

#include "blas.h"

#include <float.h>

#include <math.h>

#include <stdlib.h>

#include <stdio.h>

#include <assert.h>

#define SECRET\_NUM -1234

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

{

int b;

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

int i;

int count = 0;

for (i = 0; i < hierarchy->groups; ++i) {

int group\_size = hierarchy->group\_size[i];

softmax(input + b\*inputs + count, group\_size, temp, output + b\*inputs + count, 1);

count += group\_size;

}

}

}

softmax\_layer make\_softmax\_layer(int batch, int inputs, int groups)

{

assert(inputs%groups == 0);

fprintf(stderr, "softmax %4d\n", inputs);

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

l.type = SOFTMAX;

l.batch = batch;

l.groups = groups;

l.inputs = inputs;

l.outputs = inputs;

l.loss = (float\*)xcalloc(inputs \* batch, sizeof(float));

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

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

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

l.forward = forward\_softmax\_layer;

l.backward = backward\_softmax\_layer;

#ifdef GPU

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

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

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

l.loss\_gpu = cuda\_make\_array(l.loss, inputs\*batch);

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

#endif

return l;

}

void forward\_softmax\_layer(const softmax\_layer l, network\_state net)

{

if(l.softmax\_tree){

int i;

int count = 0;

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

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

softmax\_cpu(net.input + count, group\_size, l.batch, l.inputs, 1, 0, 1, l.temperature, l.output + count);

count += group\_size;

}

} else {

softmax\_cpu(net.input, l.inputs/l.groups, l.batch, l.inputs, l.groups, l.inputs/l.groups, 1, l.temperature, l.output);

}

if(net.truth && !l.noloss){

softmax\_x\_ent\_cpu(l.batch\*l.inputs, l.output, net.truth, l.delta, l.loss);

l.cost[0] = sum\_array(l.loss, l.batch\*l.inputs);

}

}

void backward\_softmax\_layer(const softmax\_layer l, network\_state net)

{

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

}

#ifdef GPU

void pull\_softmax\_layer\_output(const softmax\_layer layer)

{

cuda\_pull\_array(layer.output\_gpu, layer.output, layer.inputs\*layer.batch);

}

void forward\_softmax\_layer\_gpu(const softmax\_layer l, network\_state net)

{

if(l.softmax\_tree){

softmax\_tree\_gpu(net.input, 1, l.batch, l.inputs, l.temperature, l.output\_gpu, \*l.softmax\_tree);

/\*

int i;

int count = 0;

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

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

softmax\_gpu(net.input\_gpu + count, group\_size, l.batch, l.inputs, 1, 0, 1, l.temperature, l.output\_gpu + count);

count += group\_size;

}

\*/

} else {

if(l.spatial){

softmax\_gpu\_new\_api(net.input, l.c, l.batch\*l.c, l.inputs/l.c, l.w\*l.h, 1, l.w\*l.h, 1, l.output\_gpu);

}else{

softmax\_gpu\_new\_api(net.input, l.inputs/l.groups, l.batch, l.inputs, l.groups, l.inputs/l.groups, 1, l.temperature, l.output\_gpu);

}

}

if(net.truth && !l.noloss){

softmax\_x\_ent\_gpu(l.batch\*l.inputs, l.output\_gpu, net.truth, l.delta\_gpu, l.loss\_gpu);

if(l.softmax\_tree){

mask\_gpu\_new\_api(l.batch\*l.inputs, l.delta\_gpu, SECRET\_NUM, net.truth, 0);

mask\_gpu\_new\_api(l.batch\*l.inputs, l.loss\_gpu, SECRET\_NUM, net.truth, 0);

}

cuda\_pull\_array(l.loss\_gpu, l.loss, l.batch\*l.inputs);

l.cost[0] = sum\_array(l.loss, l.batch\*l.inputs);

}

}

void backward\_softmax\_layer\_gpu(const softmax\_layer layer, network\_state net)

{

axpy\_ongpu(layer.batch\*layer.inputs, 1, layer.delta\_gpu, 1, net.delta, 1);

}

#endif