#include "batchnorm\_layer.h"

#include "blas.h"

#include "utils.h"

#include <stdio.h>

layer make\_batchnorm\_layer(int batch, int w, int h, int c, int train)

{

fprintf(stderr, "Batch Normalization Layer: %d x %d x %d image\n", w,h,c);

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

layer.type = BATCHNORM;

layer.batch = batch;

layer.train = train;

layer.h = layer.out\_h = h;

layer.w = layer.out\_w = w;

layer.c = layer.out\_c = c;

layer.n = layer.c;

layer.output = (float\*)xcalloc(h \* w \* c \* batch, sizeof(float));

layer.delta = (float\*)xcalloc(h \* w \* c \* batch, sizeof(float));

layer.inputs = w\*h\*c;

layer.outputs = layer.inputs;

layer.biases = (float\*)xcalloc(c, sizeof(float));

layer.bias\_updates = (float\*)xcalloc(c, sizeof(float));

layer.scales = (float\*)xcalloc(c, sizeof(float));

layer.scale\_updates = (float\*)xcalloc(c, sizeof(float));

int i;

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

layer.scales[i] = 1;

}

layer.mean = (float\*)xcalloc(c, sizeof(float));

layer.variance = (float\*)xcalloc(c, sizeof(float));

layer.rolling\_mean = (float\*)xcalloc(c, sizeof(float));

layer.rolling\_variance = (float\*)xcalloc(c, sizeof(float));

layer.forward = forward\_batchnorm\_layer;

layer.backward = backward\_batchnorm\_layer;

layer.update = update\_batchnorm\_layer;

#ifdef GPU

layer.forward\_gpu = forward\_batchnorm\_layer\_gpu;

layer.backward\_gpu = backward\_batchnorm\_layer\_gpu;

layer.update\_gpu = update\_batchnorm\_layer\_gpu;

layer.output\_gpu = cuda\_make\_array(layer.output, h \* w \* c \* batch);

layer.biases\_gpu = cuda\_make\_array(layer.biases, c);

layer.scales\_gpu = cuda\_make\_array(layer.scales, c);

if (train) {

layer.delta\_gpu = cuda\_make\_array(layer.delta, h \* w \* c \* batch);

layer.bias\_updates\_gpu = cuda\_make\_array(layer.bias\_updates, c);

layer.scale\_updates\_gpu = cuda\_make\_array(layer.scale\_updates, c);

layer.mean\_delta\_gpu = cuda\_make\_array(layer.mean, c);

layer.variance\_delta\_gpu = cuda\_make\_array(layer.variance, c);

}

layer.mean\_gpu = cuda\_make\_array(layer.mean, c);

layer.variance\_gpu = cuda\_make\_array(layer.variance, c);

layer.rolling\_mean\_gpu = cuda\_make\_array(layer.mean, c);

layer.rolling\_variance\_gpu = cuda\_make\_array(layer.variance, c);

if (train) {

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

#ifndef CUDNN

layer.x\_norm\_gpu = cuda\_make\_array(layer.output, layer.batch\*layer.outputs);

#endif // not CUDNN

}

#ifdef CUDNN

CHECK\_CUDNN(cudnnCreateTensorDescriptor(&layer.normTensorDesc));

CHECK\_CUDNN(cudnnCreateTensorDescriptor(&layer.normDstTensorDesc));

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(layer.normDstTensorDesc, CUDNN\_TENSOR\_NCHW, CUDNN\_DATA\_FLOAT, layer.batch, layer.out\_c, layer.out\_h, layer.out\_w));

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(layer.normTensorDesc, CUDNN\_TENSOR\_NCHW, CUDNN\_DATA\_FLOAT, 1, layer.out\_c, 1, 1));

#endif

#endif

return layer;

}

void backward\_scale\_cpu(float \*x\_norm, float \*delta, int batch, int n, int size, float \*scale\_updates)

{

int i,b,f;

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

float sum = 0;

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

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

int index = i + size\*(f + n\*b);

sum += delta[index] \* x\_norm[index];

}

}

scale\_updates[f] += sum;

}

}

void mean\_delta\_cpu(float \*delta, float \*variance, int batch, int filters, int spatial, float \*mean\_delta)

{

int i,j,k;

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

mean\_delta[i] = 0;

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

for (k = 0; k < spatial; ++k) {

int index = j\*filters\*spatial + i\*spatial + k;

mean\_delta[i] += delta[index];

}

}

mean\_delta[i] \*= (-1./sqrt(variance[i] + .00001f));

}

}

void variance\_delta\_cpu(float \*x, float \*delta, float \*mean, float \*variance, int batch, int filters, int spatial, float \*variance\_delta)

{

int i,j,k;

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

variance\_delta[i] = 0;

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

for(k = 0; k < spatial; ++k){

int index = j\*filters\*spatial + i\*spatial + k;

variance\_delta[i] += delta[index]\*(x[index] - mean[i]);

}

}

variance\_delta[i] \*= -.5 \* pow(variance[i] + .00001f, (float)(-3./2.));

}

}

void normalize\_delta\_cpu(float \*x, float \*mean, float \*variance, float \*mean\_delta, float \*variance\_delta, int batch, int filters, int spatial, float \*delta)

{

int f, j, k;

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

for(f = 0; f < filters; ++f){

for(k = 0; k < spatial; ++k){

int index = j\*filters\*spatial + f\*spatial + k;

delta[index] = delta[index] \* 1./(sqrt(variance[f]) + .00001f) + variance\_delta[f] \* 2. \* (x[index] - mean[f]) / (spatial \* batch) + mean\_delta[f]/(spatial\*batch);

}

}

}

}

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

{

l->out\_h = l->h = h;

l->out\_w = l->w = w;

l->outputs = l->inputs = h\*w\*l->c;

const int output\_size = l->outputs \* l->batch;

l->output = (float\*)realloc(l->output, output\_size \* sizeof(float));

l->delta = (float\*)realloc(l->delta, output\_size \* sizeof(float));

#ifdef GPU

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

l->output\_gpu = cuda\_make\_array(l->output, output\_size);

if (l->train) {

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

l->delta\_gpu = cuda\_make\_array(l->delta, output\_size);

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

l->x\_gpu = cuda\_make\_array(l->output, output\_size);

#ifndef CUDNN

cuda\_free(l->x\_norm\_gpu);

l->x\_norm\_gpu = cuda\_make\_array(l->output, output\_size);

#endif // not CUDNN

}

#ifdef CUDNN

CHECK\_CUDNN(cudnnDestroyTensorDescriptor(l->normDstTensorDesc));

CHECK\_CUDNN(cudnnCreateTensorDescriptor(&l->normDstTensorDesc));

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(l->normDstTensorDesc, CUDNN\_TENSOR\_NCHW, CUDNN\_DATA\_FLOAT, l->batch, l->out\_c, l->out\_h, l->out\_w));

#endif // CUDNN

#endif // GPU

}

void forward\_batchnorm\_layer(layer l, network\_state state)

{

if(l.type == BATCHNORM) copy\_cpu(l.outputs\*l.batch, state.input, 1, l.output, 1);

if(l.type == CONNECTED){

l.out\_c = l.outputs;

l.out\_h = l.out\_w = 1;

}

if(state.train){

mean\_cpu(l.output, l.batch, l.out\_c, l.out\_h\*l.out\_w, l.mean);

variance\_cpu(l.output, l.mean, l.batch, l.out\_c, l.out\_h\*l.out\_w, l.variance);

scal\_cpu(l.out\_c, .9, l.rolling\_mean, 1);

axpy\_cpu(l.out\_c, .1, l.mean, 1, l.rolling\_mean, 1);

scal\_cpu(l.out\_c, .9, l.rolling\_variance, 1);

axpy\_cpu(l.out\_c, .1, l.variance, 1, l.rolling\_variance, 1);

copy\_cpu(l.outputs\*l.batch, l.output, 1, l.x, 1);

normalize\_cpu(l.output, l.mean, l.variance, l.batch, l.out\_c, l.out\_h\*l.out\_w);

copy\_cpu(l.outputs\*l.batch, l.output, 1, l.x\_norm, 1);

} else {

normalize\_cpu(l.output, l.rolling\_mean, l.rolling\_variance, l.batch, l.out\_c, l.out\_h\*l.out\_w);

}

scale\_bias(l.output, l.scales, l.batch, l.out\_c, l.out\_h\*l.out\_w);

add\_bias(l.output, l.biases, l.batch, l.out\_c, l.out\_w\*l.out\_h);

}

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

{

backward\_scale\_cpu(l.x\_norm, l.delta, l.batch, l.out\_c, l.out\_w\*l.out\_h, l.scale\_updates);

scale\_bias(l.delta, l.scales, l.batch, l.out\_c, l.out\_h\*l.out\_w);

mean\_delta\_cpu(l.delta, l.variance, l.batch, l.out\_c, l.out\_w\*l.out\_h, l.mean\_delta);

variance\_delta\_cpu(l.x, l.delta, l.mean, l.variance, l.batch, l.out\_c, l.out\_w\*l.out\_h, l.variance\_delta);

normalize\_delta\_cpu(l.x, l.mean, l.variance, l.mean\_delta, l.variance\_delta, l.batch, l.out\_c, l.out\_w\*l.out\_h, l.delta);

if(l.type == BATCHNORM) copy\_cpu(l.outputs\*l.batch, l.delta, 1, state.delta, 1);

}

void update\_batchnorm\_layer(layer l, int batch, float learning\_rate, float momentum, float decay)

{

//int size = l.nweights;

axpy\_cpu(l.c, learning\_rate / batch, l.bias\_updates, 1, l.biases, 1);

scal\_cpu(l.c, momentum, l.bias\_updates, 1);

axpy\_cpu(l.c, learning\_rate / batch, l.scale\_updates, 1, l.scales, 1);

scal\_cpu(l.c, momentum, l.scale\_updates, 1);

}

#ifdef GPU

void pull\_batchnorm\_layer(layer l)

{

cuda\_pull\_array(l.biases\_gpu, l.biases, l.out\_c);

cuda\_pull\_array(l.scales\_gpu, l.scales, l.out\_c);

cuda\_pull\_array(l.rolling\_mean\_gpu, l.rolling\_mean, l.out\_c);

cuda\_pull\_array(l.rolling\_variance\_gpu, l.rolling\_variance, l.out\_c);

}

void push\_batchnorm\_layer(layer l)

{

cuda\_push\_array(l.biases\_gpu, l.biases, l.out\_c);

cuda\_push\_array(l.scales\_gpu, l.scales, l.out\_c);

cuda\_push\_array(l.rolling\_mean\_gpu, l.rolling\_mean, l.out\_c);

cuda\_push\_array(l.rolling\_variance\_gpu, l.rolling\_variance, l.out\_c);

}

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

{

if (l.type == BATCHNORM) simple\_copy\_ongpu(l.outputs\*l.batch, state.input, l.output\_gpu);

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

if (state.net.adversarial) {

normalize\_gpu(l.output\_gpu, l.rolling\_mean\_gpu, l.rolling\_variance\_gpu, l.batch, l.out\_c, l.out\_h\*l.out\_w);

scale\_bias\_gpu(l.output\_gpu, l.scales\_gpu, l.batch, l.out\_c, l.out\_h\*l.out\_w);

add\_bias\_gpu(l.output\_gpu, l.biases\_gpu, l.batch, l.out\_c, l.out\_w\*l.out\_h);

return;

}

if (state.train) {

simple\_copy\_ongpu(l.outputs\*l.batch, l.output\_gpu, l.x\_gpu);

// cbn

if (l.batch\_normalize == 2) {

fast\_mean\_gpu(l.output\_gpu, l.batch, l.out\_c, l.out\_h\*l.out\_w, l.mean\_gpu);

//fast\_v\_gpu(l.output\_gpu, l.mean\_gpu, l.batch, l.out\_c, l.out\_h\*l.out\_w, l.v\_cbn\_gpu);

const int minibatch\_index = state.net.current\_subdivision + 1;

const int max\_minibatch\_index = state.net.subdivisions;

//printf("\n minibatch\_index = %d, max\_minibatch\_index = %d \n", minibatch\_index, max\_minibatch\_index);

const float alpha = 0.01;

int inverse\_variance = 0;

#ifdef CUDNN

inverse\_variance = 1;

#endif // CUDNN

fast\_v\_cbn\_gpu(l.output\_gpu, l.mean\_gpu, l.batch, l.out\_c, l.out\_h\*l.out\_w, minibatch\_index, max\_minibatch\_index, l.m\_cbn\_avg\_gpu, l.v\_cbn\_avg\_gpu, l.variance\_gpu,

alpha, l.rolling\_mean\_gpu, l.rolling\_variance\_gpu, inverse\_variance, .00001);

normalize\_scale\_bias\_gpu(l.output\_gpu, l.mean\_gpu, l.variance\_gpu, l.scales\_gpu, l.biases\_gpu, l.batch, l.out\_c, l.out\_h\*l.out\_w, inverse\_variance, .00001f);

#ifndef CUDNN

simple\_copy\_ongpu(l.outputs\*l.batch, l.output\_gpu, l.x\_norm\_gpu);

#endif // CUDNN

//printf("\n CBN, minibatch\_index = %d \n", minibatch\_index);

}

else {

#ifdef CUDNN

float one = 1;

float zero = 0;

cudnnBatchNormalizationForwardTraining(cudnn\_handle(),

CUDNN\_BATCHNORM\_SPATIAL,

&one,

&zero,

l.normDstTensorDesc,

l.x\_gpu, // input

l.normDstTensorDesc,

l.output\_gpu, // output

l.normTensorDesc,

l.scales\_gpu,

l.biases\_gpu,

.01,

l.rolling\_mean\_gpu, // output (should be FP32)

l.rolling\_variance\_gpu, // output (should be FP32)

.00001,

l.mean\_gpu, // output (should be FP32)

l.variance\_gpu); // output (should be FP32)

if (state.net.try\_fix\_nan) {

fix\_nan\_and\_inf(l.scales\_gpu, l.n);

fix\_nan\_and\_inf(l.biases\_gpu, l.n);

fix\_nan\_and\_inf(l.mean\_gpu, l.n);

fix\_nan\_and\_inf(l.variance\_gpu, l.n);

fix\_nan\_and\_inf(l.rolling\_mean\_gpu, l.n);

fix\_nan\_and\_inf(l.rolling\_variance\_gpu, l.n);

}

//simple\_copy\_ongpu(l.outputs\*l.batch, l.output\_gpu, l.x\_norm\_gpu);

#else // CUDNN

fast\_mean\_gpu(l.output\_gpu, l.batch, l.out\_c, l.out\_h\*l.out\_w, l.mean\_gpu);

fast\_variance\_gpu(l.output\_gpu, l.mean\_gpu, l.batch, l.out\_c, l.out\_h\*l.out\_w, l.variance\_gpu);

scal\_ongpu(l.out\_c, .99, l.rolling\_mean\_gpu, 1);

axpy\_ongpu(l.out\_c, .01, l.mean\_gpu, 1, l.rolling\_mean\_gpu, 1);

scal\_ongpu(l.out\_c, .99, l.rolling\_variance\_gpu, 1);

axpy\_ongpu(l.out\_c, .01, l.variance\_gpu, 1, l.rolling\_variance\_gpu, 1);

copy\_ongpu(l.outputs\*l.batch, l.output\_gpu, 1, l.x\_gpu, 1);

normalize\_gpu(l.output\_gpu, l.mean\_gpu, l.variance\_gpu, l.batch, l.out\_c, l.out\_h\*l.out\_w);

copy\_ongpu(l.outputs\*l.batch, l.output\_gpu, 1, l.x\_norm\_gpu, 1);

scale\_bias\_gpu(l.output\_gpu, l.scales\_gpu, l.batch, l.out\_c, l.out\_h\*l.out\_w);

add\_bias\_gpu(l.output\_gpu, l.biases\_gpu, l.batch, l.out\_c, l.out\_w\*l.out\_h);

#endif // CUDNN

}

}

else {

normalize\_gpu(l.output\_gpu, l.rolling\_mean\_gpu, l.rolling\_variance\_gpu, l.batch, l.out\_c, l.out\_h\*l.out\_w);

scale\_bias\_gpu(l.output\_gpu, l.scales\_gpu, l.batch, l.out\_c, l.out\_h\*l.out\_w);

add\_bias\_gpu(l.output\_gpu, l.biases\_gpu, l.batch, l.out\_c, l.out\_w\*l.out\_h);

}

}

void backward\_batchnorm\_layer\_gpu(layer l, network\_state state)

{

if (state.net.adversarial) {

inverse\_variance\_ongpu(l.out\_c, l.rolling\_variance\_gpu, l.variance\_gpu, 0.00001);

scale\_bias\_gpu(l.delta\_gpu, l.variance\_gpu, l.batch, l.out\_c, l.out\_h\*l.out\_w);

scale\_bias\_gpu(l.delta\_gpu, l.scales\_gpu, l.batch, l.out\_c, l.out\_h\*l.out\_w);

return;

}

if (!state.train) {

//l.mean\_gpu = l.rolling\_mean\_gpu;

//l.variance\_gpu = l.rolling\_variance\_gpu;

simple\_copy\_ongpu(l.out\_c, l.rolling\_mean\_gpu, l.mean\_gpu);

#ifdef CUDNN

inverse\_variance\_ongpu(l.out\_c, l.rolling\_variance\_gpu, l.variance\_gpu, 0.00001);

#else

simple\_copy\_ongpu(l.out\_c, l.rolling\_variance\_gpu, l.variance\_gpu);

#endif

}

#ifdef CUDNN

float one = 1;

float zero = 0;

cudnnBatchNormalizationBackward(cudnn\_handle(),

CUDNN\_BATCHNORM\_SPATIAL,

&one,

&zero,

&one,

&one,

l.normDstTensorDesc,

l.x\_gpu, // input

l.normDstTensorDesc,

l.delta\_gpu, // input

l.normDstTensorDesc,

l.output\_gpu, //l.x\_norm\_gpu, // output

l.normTensorDesc,

l.scales\_gpu, // input (should be FP32)

l.scale\_updates\_gpu, // output (should be FP32)

l.bias\_updates\_gpu, // output (should be FP32)

.00001,

l.mean\_gpu, // input (should be FP32)

l.variance\_gpu); // input (should be FP32)

simple\_copy\_ongpu(l.outputs\*l.batch, l.output\_gpu, l.delta\_gpu);

//simple\_copy\_ongpu(l.outputs\*l.batch, l.x\_norm\_gpu, l.delta\_gpu);

#else // CUDNN

backward\_bias\_gpu(l.bias\_updates\_gpu, l.delta\_gpu, l.batch, l.out\_c, l.out\_w\*l.out\_h);

backward\_scale\_gpu(l.x\_norm\_gpu, l.delta\_gpu, l.batch, l.out\_c, l.out\_w\*l.out\_h, l.scale\_updates\_gpu);

scale\_bias\_gpu(l.delta\_gpu, l.scales\_gpu, l.batch, l.out\_c, l.out\_h\*l.out\_w);

fast\_mean\_delta\_gpu(l.delta\_gpu, l.variance\_gpu, l.batch, l.out\_c, l.out\_w\*l.out\_h, l.mean\_delta\_gpu);

fast\_variance\_delta\_gpu(l.x\_gpu, l.delta\_gpu, l.mean\_gpu, l.variance\_gpu, l.batch, l.out\_c, l.out\_w\*l.out\_h, l.variance\_delta\_gpu);

normalize\_delta\_gpu(l.x\_gpu, l.mean\_gpu, l.variance\_gpu, l.mean\_delta\_gpu, l.variance\_delta\_gpu, l.batch, l.out\_c, l.out\_w\*l.out\_h, l.delta\_gpu);

#endif // CUDNN

if (l.type == BATCHNORM) simple\_copy\_ongpu(l.outputs\*l.batch, l.delta\_gpu, state.delta);

//copy\_ongpu(l.outputs\*l.batch, l.delta\_gpu, 1, state.delta, 1);

if (state.net.try\_fix\_nan) {

fix\_nan\_and\_inf(l.scale\_updates\_gpu, l.n);

fix\_nan\_and\_inf(l.bias\_updates\_gpu, l.n);

}

}

void update\_batchnorm\_layer\_gpu(layer l, int batch, float learning\_rate\_init, float momentum, float decay, float loss\_scale)

{

float learning\_rate = learning\_rate\_init \* l.learning\_rate\_scale / loss\_scale;

//float momentum = a.momentum;

//float decay = a.decay;

//int batch = a.batch;

axpy\_ongpu(l.c, learning\_rate / batch, l.bias\_updates\_gpu, 1, l.biases\_gpu, 1);

scal\_ongpu(l.c, momentum, l.bias\_updates\_gpu, 1);

axpy\_ongpu(l.c, learning\_rate / batch, l.scale\_updates\_gpu, 1, l.scales\_gpu, 1);

scal\_ongpu(l.c, momentum, l.scale\_updates\_gpu, 1);

}

#endif // GPU