#include "connected\_layer.h"

#include "batchnorm\_layer.h"

#include "convolutional\_layer.h"

#include "utils.h"

#include "dark\_cuda.h"

#include "blas.h"

#include "gemm.h"

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

size\_t get\_connected\_workspace\_size(layer l)

{

#ifdef CUDNN

return get\_convolutional\_workspace\_size(l);

/\*

if (gpu\_index >= 0) {

size\_t most = 0;

size\_t s = 0;

CHECK\_CUDNN(cudnnGetConvolutionForwardWorkspaceSize(cudnn\_handle(),

l.srcTensorDesc,

l.weightDesc,

l.convDesc,

l.dstTensorDesc,

l.fw\_algo,

&s));

if (s > most) most = s;

CHECK\_CUDNN(cudnnGetConvolutionBackwardFilterWorkspaceSize(cudnn\_handle(),

l.srcTensorDesc,

l.ddstTensorDesc,

l.convDesc,

l.dweightDesc,

l.bf\_algo,

&s));

if (s > most) most = s;

CHECK\_CUDNN(cudnnGetConvolutionBackwardDataWorkspaceSize(cudnn\_handle(),

l.weightDesc,

l.ddstTensorDesc,

l.convDesc,

l.dsrcTensorDesc,

l.bd\_algo,

&s));

if (s > most) most = s;

return most;

}

\*/

#endif

return 0;

}

connected\_layer make\_connected\_layer(int batch, int steps, int inputs, int outputs, ACTIVATION activation, int batch\_normalize)

{

int total\_batch = batch\*steps;

int i;

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

l.type = CONNECTED;

l.inputs = inputs;

l.outputs = outputs;

l.batch= batch;

l.batch\_normalize = batch\_normalize;

l.h = 1;

l.w = 1;

l.c = inputs;

l.out\_h = 1;

l.out\_w = 1;

l.out\_c = outputs;

l.n = l.out\_c;

l.size = 1;

l.stride = l.stride\_x = l.stride\_y = 1;

l.pad = 0;

l.activation = activation;

l.learning\_rate\_scale = 1;

l.groups = 1;

l.dilation = 1;

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

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

l.weight\_updates = (float\*)xcalloc(inputs \* outputs, sizeof(float));

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

l.weights = (float\*)xcalloc(outputs \* inputs, sizeof(float));

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

l.forward = forward\_connected\_layer;

l.backward = backward\_connected\_layer;

l.update = update\_connected\_layer;

//float scale = 1./sqrt(inputs);

float scale = sqrt(2.f/inputs);

for(i = 0; i < outputs\*inputs; ++i){

l.weights[i] = scale\*rand\_uniform(-1, 1);

}

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

l.biases[i] = 0;

}

if(batch\_normalize){

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

l.scale\_updates = (float\*)xcalloc(outputs, sizeof(float));

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

l.scales[i] = 1;

}

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

l.mean\_delta = (float\*)xcalloc(outputs, sizeof(float));

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

l.variance\_delta = (float\*)xcalloc(outputs, sizeof(float));

l.rolling\_mean = (float\*)xcalloc(outputs, sizeof(float));

l.rolling\_variance = (float\*)xcalloc(outputs, sizeof(float));

l.x = (float\*)xcalloc(total\_batch \* outputs, sizeof(float));

l.x\_norm = (float\*)xcalloc(total\_batch \* outputs, sizeof(float));

}

#ifdef GPU

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

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

l.update\_gpu = update\_connected\_layer\_gpu;

l.weights\_gpu = cuda\_make\_array(l.weights, outputs\*inputs);

l.biases\_gpu = cuda\_make\_array(l.biases, outputs);

l.weight\_updates\_gpu = cuda\_make\_array(l.weight\_updates, outputs\*inputs);

l.bias\_updates\_gpu = cuda\_make\_array(l.bias\_updates, outputs);

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

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

if (batch\_normalize) {

l.scales\_gpu = cuda\_make\_array(l.scales, outputs);

l.scale\_updates\_gpu = cuda\_make\_array(l.scale\_updates, outputs);

l.mean\_gpu = cuda\_make\_array(l.mean, outputs);

l.variance\_gpu = cuda\_make\_array(l.variance, outputs);

l.rolling\_mean\_gpu = cuda\_make\_array(l.mean, outputs);

l.rolling\_variance\_gpu = cuda\_make\_array(l.variance, outputs);

l.mean\_delta\_gpu = cuda\_make\_array(l.mean, outputs);

l.variance\_delta\_gpu = cuda\_make\_array(l.variance, outputs);

l.x\_gpu = cuda\_make\_array(l.output, total\_batch\*outputs);

l.x\_norm\_gpu = cuda\_make\_array(l.output, total\_batch\*outputs);

}

#ifdef CUDNN

create\_convolutional\_cudnn\_tensors(&l);

cudnn\_convolutional\_setup(&l, cudnn\_fastest, 0); // cudnn\_fastest, cudnn\_smallest

l.workspace\_size = get\_connected\_workspace\_size(l);

#endif // CUDNN

#endif // GPU

fprintf(stderr, "connected %4d -> %4d\n", inputs, outputs);

return l;

}

void update\_connected\_layer(connected\_layer l, int batch, float learning\_rate, float momentum, float decay)

{

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

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

if(l.batch\_normalize){

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

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

}

axpy\_cpu(l.inputs\*l.outputs, -decay\*batch, l.weights, 1, l.weight\_updates, 1);

axpy\_cpu(l.inputs\*l.outputs, learning\_rate/batch, l.weight\_updates, 1, l.weights, 1);

scal\_cpu(l.inputs\*l.outputs, momentum, l.weight\_updates, 1);

}

void forward\_connected\_layer(connected\_layer l, network\_state state)

{

int i;

fill\_cpu(l.outputs\*l.batch, 0, l.output, 1);

int m = l.batch;

int k = l.inputs;

int n = l.outputs;

float \*a = state.input;

float \*b = l.weights;

float \*c = l.output;

gemm(0,1,m,n,k,1,a,k,b,k,1,c,n);

if(l.batch\_normalize){

if(state.train){

mean\_cpu(l.output, l.batch, l.outputs, 1, l.mean);

variance\_cpu(l.output, l.mean, l.batch, l.outputs, 1, l.variance);

scal\_cpu(l.outputs, .95f, l.rolling\_mean, 1);

axpy\_cpu(l.outputs, .05f, l.mean, 1, l.rolling\_mean, 1);

scal\_cpu(l.outputs, .95f, l.rolling\_variance, 1);

axpy\_cpu(l.outputs, .05f, 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.outputs, 1);

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.outputs, 1);

}

scale\_bias(l.output, l.scales, l.batch, l.outputs, 1);

}

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

axpy\_cpu(l.outputs, 1, l.biases, 1, l.output + i\*l.outputs, 1);

}

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

}

void backward\_connected\_layer(connected\_layer l, network\_state state)

{

int i;

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

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

axpy\_cpu(l.outputs, 1, l.delta + i\*l.outputs, 1, l.bias\_updates, 1);

}

if(l.batch\_normalize){

backward\_scale\_cpu(l.x\_norm, l.delta, l.batch, l.outputs, 1, l.scale\_updates);

scale\_bias(l.delta, l.scales, l.batch, l.outputs, 1);

mean\_delta\_cpu(l.delta, l.variance, l.batch, l.outputs, 1, l.mean\_delta);

variance\_delta\_cpu(l.x, l.delta, l.mean, l.variance, l.batch, l.outputs, 1, l.variance\_delta);

normalize\_delta\_cpu(l.x, l.mean, l.variance, l.mean\_delta, l.variance\_delta, l.batch, l.outputs, 1, l.delta);

}

int m = l.outputs;

int k = l.batch;

int n = l.inputs;

float \*a = l.delta;

float \*b = state.input;

float \*c = l.weight\_updates;

gemm(1,0,m,n,k,1,a,m,b,n,1,c,n);

m = l.batch;

k = l.outputs;

n = l.inputs;

a = l.delta;

b = l.weights;

c = state.delta;

if(c) gemm(0,0,m,n,k,1,a,k,b,n,1,c,n);

}

void denormalize\_connected\_layer(layer l)

{

int i, j;

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

float scale = l.scales[i]/sqrt(l.rolling\_variance[i] + .000001f);

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

l.weights[i\*l.inputs + j] \*= scale;

}

l.biases[i] -= l.rolling\_mean[i] \* scale;

l.scales[i] = 1;

l.rolling\_mean[i] = 0;

l.rolling\_variance[i] = 1;

}

}

void statistics\_connected\_layer(layer l)

{

if(l.batch\_normalize){

printf("Scales ");

print\_statistics(l.scales, l.outputs);

/\*

printf("Rolling Mean ");

print\_statistics(l.rolling\_mean, l.outputs);

printf("Rolling Variance ");

print\_statistics(l.rolling\_variance, l.outputs);

\*/

}

printf("Biases ");

print\_statistics(l.biases, l.outputs);

printf("Weights ");

print\_statistics(l.weights, l.outputs);

}

#ifdef GPU

void pull\_connected\_layer(connected\_layer l)

{

cuda\_pull\_array(l.weights\_gpu, l.weights, l.inputs\*l.outputs);

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

cuda\_pull\_array(l.weight\_updates\_gpu, l.weight\_updates, l.inputs\*l.outputs);

cuda\_pull\_array(l.bias\_updates\_gpu, l.bias\_updates, l.outputs);

if (l.batch\_normalize){

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

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

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

}

CHECK\_CUDA(cudaPeekAtLastError());

}

void push\_connected\_layer(connected\_layer l)

{

cuda\_push\_array(l.weights\_gpu, l.weights, l.inputs\*l.outputs);

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

cuda\_push\_array(l.weight\_updates\_gpu, l.weight\_updates, l.inputs\*l.outputs);

cuda\_push\_array(l.bias\_updates\_gpu, l.bias\_updates, l.outputs);

if (l.batch\_normalize){

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

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

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

}

CHECK\_CUDA(cudaPeekAtLastError());

}

void update\_connected\_layer\_gpu(connected\_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 for Mixed-Precision on Tensor-Cores

if (loss\_scale != 1.0) {

scal\_ongpu(l.inputs\*l.outputs, 1.0 / loss\_scale, l.weight\_updates\_gpu, 1);

scal\_ongpu(l.outputs, 1.0 / loss\_scale, l.bias\_updates\_gpu, 1);

scal\_ongpu(l.outputs, 1.0 / loss\_scale, l.scale\_updates\_gpu, 1);

}

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

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

if(l.batch\_normalize){

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

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

}

axpy\_ongpu(l.inputs\*l.outputs, -decay\*batch, l.weights\_gpu, 1, l.weight\_updates\_gpu, 1);

axpy\_ongpu(l.inputs\*l.outputs, learning\_rate/batch, l.weight\_updates\_gpu, 1, l.weights\_gpu, 1);

scal\_ongpu(l.inputs\*l.outputs, momentum, l.weight\_updates\_gpu, 1);

}

void forward\_connected\_layer\_gpu(connected\_layer l, network\_state state)

{

fill\_ongpu(l.outputs\*l.batch, 0, l.output\_gpu, 1);

int m = l.batch;

int k = l.inputs;

int n = l.outputs;

float \* a = state.input;

float \* b = l.weights\_gpu;

float \* c = l.output\_gpu;

#ifdef CUDNN

float one = 1; // alpha[0], beta[0]

float alpha = 1, beta = 0;

CHECK\_CUDNN(cudnnConvolutionForward(cudnn\_handle(),

&alpha, //&one,

l.srcTensorDesc,

state.input,

l.weightDesc,

l.weights\_gpu,

l.convDesc,

l.fw\_algo,

state.workspace,

l.workspace\_size,

&beta, //&one,

l.dstTensorDesc,

l.output\_gpu));

#else // CUDNN

gemm\_ongpu(0,1,m,n,k,1,a,k,b,k,1,c,n);

#endif // CUDNN

if (l.batch\_normalize) {

forward\_batchnorm\_layer\_gpu(l, state);

}

else {

add\_bias\_gpu(l.output\_gpu, l.biases\_gpu, l.batch, l.outputs, 1);

}

//for(i = 0; i < l.batch; ++i) axpy\_ongpu(l.outputs, 1, l.biases\_gpu, 1, l.output\_gpu + i\*l.outputs, 1);

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

}

void backward\_connected\_layer\_gpu(connected\_layer l, network\_state state)

{

int i;

constrain\_ongpu(l.outputs\*l.batch, 1, l.delta\_gpu, 1);

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

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

axpy\_ongpu(l.outputs, 1, l.delta\_gpu + i\*l.outputs, 1, l.bias\_updates\_gpu, 1);

}

if(l.batch\_normalize){

backward\_batchnorm\_layer\_gpu(l, state);

}

#ifdef CUDNN\_DISABLED

float one = 1;

// calculate conv weight updates

// if used: beta=1 then loss decreases faster

CHECK\_CUDNN(cudnnConvolutionBackwardFilter(cudnn\_handle(),

&one,

l.srcTensorDesc,

state.input,

l.ddstTensorDesc,

l.delta\_gpu,

l.convDesc,

l.bf\_algo,

state.workspace,

l.workspace\_size,

&one,

l.dweightDesc,

l.weight\_updates\_gpu));

if (state.delta) {

// http://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html#cudnnConvolutionBackwardData

// calculate delta for the next layer

CHECK\_CUDNN(cudnnConvolutionBackwardData(cudnn\_handle(),

&one,

l.weightDesc,

l.weights\_gpu,

l.ddstTensorDesc,

l.delta\_gpu,

l.convDesc,

l.bd\_algo,

state.workspace,

l.workspace\_size,

&one,

l.dsrcTensorDesc,

state.delta));

}

#else // CUDNN

int m = l.outputs;

int k = l.batch;

int n = l.inputs;

float \* a = l.delta\_gpu;

float \* b = state.input;

float \* c = l.weight\_updates\_gpu;

gemm\_ongpu(1,0,m,n,k,1,a,m,b,n,1,c,n);

m = l.batch;

k = l.outputs;

n = l.inputs;

a = l.delta\_gpu;

b = l.weights\_gpu;

c = state.delta;

if(c) gemm\_ongpu(0,0,m,n,k,1,a,k,b,n,1,c,n);

#endif // CUDNN

}

#endif