#include "crnn\_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>

static void increment\_layer(layer \*l, int steps)

{

int num = l->outputs\*l->batch\*steps;

l->output += num;

l->delta += num;

l->x += num;

l->x\_norm += num;

#ifdef GPU

l->output\_gpu += num;

l->delta\_gpu += num;

l->x\_gpu += num;

l->x\_norm\_gpu += num;

#endif

}

layer make\_crnn\_layer(int batch, int h, int w, int c, int hidden\_filters, int output\_filters, int groups, int steps, int size, int stride, int dilation, int pad, ACTIVATION activation, int batch\_normalize, int xnor, int train)

{

fprintf(stderr, "CRNN Layer: %d x %d x %d image, %d filters\n", h,w,c,output\_filters);

batch = batch / steps;

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

l.train = train;

l.batch = batch;

l.type = CRNN;

l.steps = steps;

l.size = size;

l.stride = stride;

l.dilation = dilation;

l.pad = pad;

l.h = h;

l.w = w;

l.c = c;

l.groups = groups;

l.out\_c = output\_filters;

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

l.hidden = h \* w \* hidden\_filters;

l.xnor = xnor;

l.state = (float\*)xcalloc(l.hidden \* l.batch \* (l.steps + 1), sizeof(float));

l.input\_layer = (layer\*)xcalloc(1, sizeof(layer));

\*(l.input\_layer) = make\_convolutional\_layer(batch, steps, h, w, c, hidden\_filters, groups, size, stride, stride, dilation, pad, activation, batch\_normalize, 0, xnor, 0, 0, 0, 0, NULL, 0, 0, train);

l.input\_layer->batch = batch;

if (l.workspace\_size < l.input\_layer->workspace\_size) l.workspace\_size = l.input\_layer->workspace\_size;

l.self\_layer = (layer\*)xcalloc(1, sizeof(layer));

\*(l.self\_layer) = make\_convolutional\_layer(batch, steps, h, w, hidden\_filters, hidden\_filters, groups, size, stride, stride, dilation, pad, activation, batch\_normalize, 0, xnor, 0, 0, 0, 0, NULL, 0, 0, train);

l.self\_layer->batch = batch;

if (l.workspace\_size < l.self\_layer->workspace\_size) l.workspace\_size = l.self\_layer->workspace\_size;

l.output\_layer = (layer\*)xcalloc(1, sizeof(layer));

\*(l.output\_layer) = make\_convolutional\_layer(batch, steps, h, w, hidden\_filters, output\_filters, groups, size, stride, stride, dilation, pad, activation, batch\_normalize, 0, xnor, 0, 0, 0, 0, NULL, 0, 0, train);

l.output\_layer->batch = batch;

if (l.workspace\_size < l.output\_layer->workspace\_size) l.workspace\_size = l.output\_layer->workspace\_size;

l.out\_h = l.output\_layer->out\_h;

l.out\_w = l.output\_layer->out\_w;

l.outputs = l.output\_layer->outputs;

assert(l.input\_layer->outputs == l.self\_layer->outputs);

assert(l.input\_layer->outputs == l.output\_layer->inputs);

l.output = l.output\_layer->output;

l.delta = l.output\_layer->delta;

l.forward = forward\_crnn\_layer;

l.backward = backward\_crnn\_layer;

l.update = update\_crnn\_layer;

#ifdef GPU

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

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

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

l.state\_gpu = cuda\_make\_array(l.state, l.batch\*l.hidden\*(l.steps + 1));

l.output\_gpu = l.output\_layer->output\_gpu;

l.delta\_gpu = l.output\_layer->delta\_gpu;

#endif

l.bflops = l.input\_layer->bflops + l.self\_layer->bflops + l.output\_layer->bflops;

return l;

}

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

{

resize\_convolutional\_layer(l->input\_layer, w, h);

if (l->workspace\_size < l->input\_layer->workspace\_size) l->workspace\_size = l->input\_layer->workspace\_size;

resize\_convolutional\_layer(l->self\_layer, w, h);

if (l->workspace\_size < l->self\_layer->workspace\_size) l->workspace\_size = l->self\_layer->workspace\_size;

resize\_convolutional\_layer(l->output\_layer, w, h);

if (l->workspace\_size < l->output\_layer->workspace\_size) l->workspace\_size = l->output\_layer->workspace\_size;

l->output = l->output\_layer->output;

l->delta = l->output\_layer->delta;

int hidden\_filters = l->self\_layer->c;

l->w = w;

l->h = h;

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

l->hidden = h \* w \* hidden\_filters;

l->out\_h = l->output\_layer->out\_h;

l->out\_w = l->output\_layer->out\_w;

l->outputs = l->output\_layer->outputs;

assert(l->input\_layer->inputs == l->inputs);

assert(l->self\_layer->inputs == l->hidden);

assert(l->input\_layer->outputs == l->self\_layer->outputs);

assert(l->input\_layer->outputs == l->output\_layer->inputs);

l->state = (float\*)xrealloc(l->state, l->batch\*l->hidden\*(l->steps + 1)\*sizeof(float));

#ifdef GPU

if (l->state\_gpu) cudaFree(l->state\_gpu);

l->state\_gpu = cuda\_make\_array(l->state, l->batch\*l->hidden\*(l->steps + 1));

l->output\_gpu = l->output\_layer->output\_gpu;

l->delta\_gpu = l->output\_layer->delta\_gpu;

#endif

}

void free\_state\_crnn(layer l)

{

int i;

for (i = 0; i < l.outputs \* l.batch; ++i) l.self\_layer->output[i] = rand\_uniform(-1, 1);

#ifdef GPU

cuda\_push\_array(l.self\_layer->output\_gpu, l.self\_layer->output, l.outputs \* l.batch);

#endif // GPU

}

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

{

update\_convolutional\_layer(\*(l.input\_layer), batch, learning\_rate, momentum, decay);

update\_convolutional\_layer(\*(l.self\_layer), batch, learning\_rate, momentum, decay);

update\_convolutional\_layer(\*(l.output\_layer), batch, learning\_rate, momentum, decay);

}

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

{

network\_state s = {0};

s.train = state.train;

s.workspace = state.workspace;

s.net = state.net;

//s.index = state.index;

int i;

layer input\_layer = \*(l.input\_layer);

layer self\_layer = \*(l.self\_layer);

layer output\_layer = \*(l.output\_layer);

if (state.train) {

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

fill\_cpu(l.hidden \* l.batch \* l.steps, 0, self\_layer.delta, 1);

fill\_cpu(l.hidden \* l.batch \* l.steps, 0, input\_layer.delta, 1);

fill\_cpu(l.hidden \* l.batch, 0, l.state, 1);

}

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

s.input = state.input;

forward\_convolutional\_layer(input\_layer, s);

s.input = l.state;

forward\_convolutional\_layer(self\_layer, s);

float \*old\_state = l.state;

if(state.train) l.state += l.hidden\*l.batch;

if(l.shortcut){

copy\_cpu(l.hidden \* l.batch, old\_state, 1, l.state, 1);

}else{

fill\_cpu(l.hidden \* l.batch, 0, l.state, 1);

}

axpy\_cpu(l.hidden \* l.batch, 1, input\_layer.output, 1, l.state, 1);

axpy\_cpu(l.hidden \* l.batch, 1, self\_layer.output, 1, l.state, 1);

s.input = l.state;

forward\_convolutional\_layer(output\_layer, s);

state.input += l.inputs\*l.batch;

increment\_layer(&input\_layer, 1);

increment\_layer(&self\_layer, 1);

increment\_layer(&output\_layer, 1);

}

}

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

{

network\_state s = {0};

s.train = state.train;

s.workspace = state.workspace;

s.net = state.net;

//s.index = state.index;

int i;

layer input\_layer = \*(l.input\_layer);

layer self\_layer = \*(l.self\_layer);

layer output\_layer = \*(l.output\_layer);

increment\_layer(&input\_layer, l.steps-1);

increment\_layer(&self\_layer, l.steps-1);

increment\_layer(&output\_layer, l.steps-1);

l.state += l.hidden\*l.batch\*l.steps;

for (i = l.steps-1; i >= 0; --i) {

copy\_cpu(l.hidden \* l.batch, input\_layer.output, 1, l.state, 1);

axpy\_cpu(l.hidden \* l.batch, 1, self\_layer.output, 1, l.state, 1);

s.input = l.state;

s.delta = self\_layer.delta;

backward\_convolutional\_layer(output\_layer, s);

l.state -= l.hidden\*l.batch;

/\*

if(i > 0){

copy\_cpu(l.hidden \* l.batch, input\_layer.output - l.hidden\*l.batch, 1, l.state, 1);

axpy\_cpu(l.hidden \* l.batch, 1, self\_layer.output - l.hidden\*l.batch, 1, l.state, 1);

}else{

fill\_cpu(l.hidden \* l.batch, 0, l.state, 1);

}

\*/

s.input = l.state;

s.delta = self\_layer.delta - l.hidden\*l.batch;

if (i == 0) s.delta = 0;

backward\_convolutional\_layer(self\_layer, s);

copy\_cpu(l.hidden\*l.batch, self\_layer.delta, 1, input\_layer.delta, 1);

if (i > 0 && l.shortcut) axpy\_cpu(l.hidden\*l.batch, 1, self\_layer.delta, 1, self\_layer.delta - l.hidden\*l.batch, 1);

s.input = state.input + i\*l.inputs\*l.batch;

if(state.delta) s.delta = state.delta + i\*l.inputs\*l.batch;

else s.delta = 0;

backward\_convolutional\_layer(input\_layer, s);

increment\_layer(&input\_layer, -1);

increment\_layer(&self\_layer, -1);

increment\_layer(&output\_layer, -1);

}

}

#ifdef GPU

void pull\_crnn\_layer(layer l)

{

pull\_convolutional\_layer(\*(l.input\_layer));

pull\_convolutional\_layer(\*(l.self\_layer));

pull\_convolutional\_layer(\*(l.output\_layer));

}

void push\_crnn\_layer(layer l)

{

push\_convolutional\_layer(\*(l.input\_layer));

push\_convolutional\_layer(\*(l.self\_layer));

push\_convolutional\_layer(\*(l.output\_layer));

}

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

{

update\_convolutional\_layer\_gpu(\*(l.input\_layer), batch, learning\_rate, momentum, decay, loss\_scale);

update\_convolutional\_layer\_gpu(\*(l.self\_layer), batch, learning\_rate, momentum, decay, loss\_scale);

update\_convolutional\_layer\_gpu(\*(l.output\_layer), batch, learning\_rate, momentum, decay, loss\_scale);

}

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

{

network\_state s = {0};

s.train = state.train;

s.workspace = state.workspace;

s.net = state.net;

if(!state.train) s.index = state.index; // don't use TC for training (especially without cuda\_convert\_f32\_to\_f16() )

int i;

layer input\_layer = \*(l.input\_layer);

layer self\_layer = \*(l.self\_layer);

layer output\_layer = \*(l.output\_layer);

/\*

#ifdef CUDNN\_HALF // slow and bad for training

if (!state.train && state.net.cudnn\_half) {

s.index = state.index;

cuda\_convert\_f32\_to\_f16(input\_layer.weights\_gpu, input\_layer.c\*input\_layer.n\*input\_layer.size\*input\_layer.size, input\_layer.weights\_gpu16);

cuda\_convert\_f32\_to\_f16(self\_layer.weights\_gpu, self\_layer.c\*self\_layer.n\*self\_layer.size\*self\_layer.size, self\_layer.weights\_gpu16);

cuda\_convert\_f32\_to\_f16(output\_layer.weights\_gpu, output\_layer.c\*output\_layer.n\*output\_layer.size\*output\_layer.size, output\_layer.weights\_gpu16);

}

#endif //CUDNN\_HALF

\*/

if (state.train) {

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

fill\_ongpu(l.hidden \* l.batch \* l.steps, 0, self\_layer.delta\_gpu, 1);

fill\_ongpu(l.hidden \* l.batch \* l.steps, 0, input\_layer.delta\_gpu, 1);

fill\_ongpu(l.hidden \* l.batch, 0, l.state\_gpu, 1);

}

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

s.input = state.input;

forward\_convolutional\_layer\_gpu(input\_layer, s);

s.input = l.state\_gpu;

forward\_convolutional\_layer\_gpu(self\_layer, s);

float \*old\_state = l.state\_gpu;

if(state.train) l.state\_gpu += l.hidden\*l.batch;

if(l.shortcut){

copy\_ongpu(l.hidden \* l.batch, old\_state, 1, l.state\_gpu, 1);

}else{

fill\_ongpu(l.hidden \* l.batch, 0, l.state\_gpu, 1);

}

axpy\_ongpu(l.hidden \* l.batch, 1, input\_layer.output\_gpu, 1, l.state\_gpu, 1);

axpy\_ongpu(l.hidden \* l.batch, 1, self\_layer.output\_gpu, 1, l.state\_gpu, 1);

s.input = l.state\_gpu;

forward\_convolutional\_layer\_gpu(output\_layer, s);

state.input += l.inputs\*l.batch;

increment\_layer(&input\_layer, 1);

increment\_layer(&self\_layer, 1);

increment\_layer(&output\_layer, 1);

}

}

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

{

network\_state s = {0};

s.train = state.train;

s.workspace = state.workspace;

s.net = state.net;

//s.index = state.index;

int i;

layer input\_layer = \*(l.input\_layer);

layer self\_layer = \*(l.self\_layer);

layer output\_layer = \*(l.output\_layer);

increment\_layer(&input\_layer, l.steps - 1);

increment\_layer(&self\_layer, l.steps - 1);

increment\_layer(&output\_layer, l.steps - 1);

float \*init\_state\_gpu = l.state\_gpu;

l.state\_gpu += l.hidden\*l.batch\*l.steps;

for (i = l.steps-1; i >= 0; --i) {

//copy\_ongpu(l.hidden \* l.batch, input\_layer.output\_gpu, 1, l.state\_gpu, 1); // commented in RNN

//axpy\_ongpu(l.hidden \* l.batch, 1, self\_layer.output\_gpu, 1, l.state\_gpu, 1); // commented in RNN

s.input = l.state\_gpu;

s.delta = self\_layer.delta\_gpu;

backward\_convolutional\_layer\_gpu(output\_layer, s);

l.state\_gpu -= l.hidden\*l.batch;

copy\_ongpu(l.hidden\*l.batch, self\_layer.delta\_gpu, 1, input\_layer.delta\_gpu, 1);

s.input = l.state\_gpu;

s.delta = self\_layer.delta\_gpu - l.hidden\*l.batch;

if (i == 0) s.delta = 0;

backward\_convolutional\_layer\_gpu(self\_layer, s);

if (i > 0 && l.shortcut) axpy\_ongpu(l.hidden\*l.batch, 1, self\_layer.delta\_gpu, 1, self\_layer.delta\_gpu - l.hidden\*l.batch, 1);

s.input = state.input + i\*l.inputs\*l.batch;

if(state.delta) s.delta = state.delta + i\*l.inputs\*l.batch;

else s.delta = 0;

backward\_convolutional\_layer\_gpu(input\_layer, s);

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

fix\_nan\_and\_inf(output\_layer.delta\_gpu, output\_layer.inputs \* output\_layer.batch);

fix\_nan\_and\_inf(self\_layer.delta\_gpu, self\_layer.inputs \* self\_layer.batch);

fix\_nan\_and\_inf(input\_layer.delta\_gpu, input\_layer.inputs \* input\_layer.batch);

}

increment\_layer(&input\_layer, -1);

increment\_layer(&self\_layer, -1);

increment\_layer(&output\_layer, -1);

}

fill\_ongpu(l.hidden \* l.batch, 0, init\_state\_gpu, 1); //clean l.state\_gpu

}

#endif