// Page 4: https://arxiv.org/abs/1506.04214v2

// Page 3: https://arxiv.org/pdf/1705.06368v3.pdf

// https://wikimedia.org/api/rest\_v1/media/math/render/svg/1edbece2559479959fe829e9c6657efb380debe7

#include "conv\_lstm\_layer.h"

#include "connected\_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\_conv\_lstm\_layer(int batch, int h, int w, int c, int output\_filters, int groups, int steps, int size, int stride, int dilation, int pad, ACTIVATION activation, int batch\_normalize, int peephole, int xnor, int train)

{

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

/\*

batch = batch / steps;

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

l.batch = batch;

l.type = LSTM;

l.steps = steps;

l.inputs = inputs;

l.out\_w = 1;

l.out\_h = 1;

l.out\_c = outputs;

\*/

batch = batch / steps;

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

l.train = train;

l.batch = batch;

l.type = CONV\_LSTM;

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.xnor = xnor;

l.peephole = peephole;

// U

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

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

l.uf->batch = batch;

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

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

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

l.ui->batch = batch;

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

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

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

l.ug->batch = batch;

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

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

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

l.uo->batch = batch;

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

// W

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

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

l.wf->batch = batch;

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

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

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

l.wi->batch = batch;

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

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

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

l.wg->batch = batch;

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

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

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

l.wo->batch = batch;

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

// V

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

if (l.peephole) {

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

l.vf->batch = batch;

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

}

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

if (l.peephole) {

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

l.vi->batch = batch;

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

}

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

if (l.peephole) {

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

l.vo->batch = batch;

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

}

l.batch\_normalize = batch\_normalize;

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

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

l.outputs = l.wo->outputs;

int outputs = l.outputs;

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

assert(l.wo->outputs == l.uo->outputs);

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

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

l.forward = forward\_conv\_lstm\_layer;

l.update = update\_conv\_lstm\_layer;

l.backward = backward\_conv\_lstm\_layer;

l.prev\_state\_cpu = (float\*)xcalloc(batch\*outputs, sizeof(float));

l.prev\_cell\_cpu = (float\*)xcalloc(batch\*outputs, sizeof(float));

l.cell\_cpu = (float\*)xcalloc(batch\*outputs\*steps, sizeof(float));

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

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

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

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

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

l.stored\_c\_cpu = (float\*)xcalloc(batch\*outputs, sizeof(float));

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

l.stored\_h\_cpu = (float\*)xcalloc(batch\*outputs, sizeof(float));

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

l.temp2\_cpu = (float\*)xcalloc(batch\*outputs, sizeof(float));

l.temp3\_cpu = (float\*)xcalloc(batch\*outputs, sizeof(float));

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

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

#ifdef GPU

l.forward\_gpu = forward\_conv\_lstm\_layer\_gpu;

l.backward\_gpu = backward\_conv\_lstm\_layer\_gpu;

l.update\_gpu = update\_conv\_lstm\_layer\_gpu;

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

l.output\_gpu = cuda\_make\_array(0, batch\*outputs\*steps);

l.delta\_gpu = cuda\_make\_array(0, batch\*l.outputs\*steps);

l.prev\_state\_gpu = cuda\_make\_array(0, batch\*outputs);

l.prev\_cell\_gpu = cuda\_make\_array(0, batch\*outputs);

l.cell\_gpu = cuda\_make\_array(0, batch\*outputs\*steps);

l.f\_gpu = cuda\_make\_array(0, batch\*outputs);

l.i\_gpu = cuda\_make\_array(0, batch\*outputs);

l.g\_gpu = cuda\_make\_array(0, batch\*outputs);

l.o\_gpu = cuda\_make\_array(0, batch\*outputs);

l.c\_gpu = cuda\_make\_array(0, batch\*outputs);

l.h\_gpu = cuda\_make\_array(0, batch\*outputs);

l.stored\_c\_gpu = cuda\_make\_array(0, batch\*outputs);

l.stored\_h\_gpu = cuda\_make\_array(0, batch\*outputs);

l.temp\_gpu = cuda\_make\_array(0, batch\*outputs);

l.temp2\_gpu = cuda\_make\_array(0, batch\*outputs);

l.temp3\_gpu = cuda\_make\_array(0, batch\*outputs);

l.dc\_gpu = cuda\_make\_array(0, batch\*outputs);

l.dh\_gpu = cuda\_make\_array(0, batch\*outputs);

l.last\_prev\_state\_gpu = cuda\_make\_array(0, l.batch\*l.outputs);

l.last\_prev\_cell\_gpu = cuda\_make\_array(0, l.batch\*l.outputs);

#endif

l.bflops = l.uf->bflops + l.ui->bflops + l.ug->bflops + l.uo->bflops +

l.wf->bflops + l.wi->bflops + l.wg->bflops + l.wo->bflops +

l.vf->bflops + l.vi->bflops + l.vo->bflops;

if(l.peephole) l.bflops += 12 \* l.outputs\*l.batch / 1000000000.;

else l.bflops += 9 \* l.outputs\*l.batch / 1000000000.;

return l;

}

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

{

if (l.peephole) {

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

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

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

}

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

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

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

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

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

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

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

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

}

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

{

if (l->peephole) {

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

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

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

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

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

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

}

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

l->w = w;

l->h = h;

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

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

l->outputs = l->wo->outputs;

int outputs = l->outputs;

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

int steps = l->steps;

int batch = l->batch;

assert(l->wo->outputs == l->uo->outputs);

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

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

l->prev\_state\_cpu = (float\*)xrealloc(l->prev\_state\_cpu, batch\*outputs \* sizeof(float));

l->prev\_cell\_cpu = (float\*)xrealloc(l->prev\_cell\_cpu, batch\*outputs \* sizeof(float));

l->cell\_cpu = (float\*)xrealloc(l->cell\_cpu, batch\*outputs\*steps \* sizeof(float));

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

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

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

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

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

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

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

l->temp2\_cpu = (float\*)xrealloc(l->temp2\_cpu, batch\*outputs \* sizeof(float));

l->temp3\_cpu = (float\*)xrealloc(l->temp3\_cpu, batch\*outputs \* sizeof(float));

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

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

l->stored\_c\_cpu = (float\*)xrealloc(l->stored\_c\_cpu, batch\*outputs \* sizeof(float));

l->stored\_h\_cpu = (float\*)xrealloc(l->stored\_h\_cpu, batch\*outputs \* sizeof(float));

#ifdef GPU

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

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

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

l->output\_gpu = cuda\_make\_array(0, batch\*outputs\*steps);

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

l->delta\_gpu = cuda\_make\_array(0, batch\*outputs\*steps);

if (l->prev\_state\_gpu) cudaFree(l->prev\_state\_gpu);

l->prev\_state\_gpu = cuda\_make\_array(0, batch\*outputs);

if (l->prev\_cell\_gpu) cudaFree(l->prev\_cell\_gpu);

l->prev\_cell\_gpu = cuda\_make\_array(0, batch\*outputs);

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

l->cell\_gpu = cuda\_make\_array(0, batch\*outputs\*steps);

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

l->f\_gpu = cuda\_make\_array(0, batch\*outputs);

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

l->i\_gpu = cuda\_make\_array(0, batch\*outputs);

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

l->g\_gpu = cuda\_make\_array(0, batch\*outputs);

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

l->o\_gpu = cuda\_make\_array(0, batch\*outputs);

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

l->c\_gpu = cuda\_make\_array(0, batch\*outputs);

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

l->h\_gpu = cuda\_make\_array(0, batch\*outputs);

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

l->temp\_gpu = cuda\_make\_array(0, batch\*outputs);

if (l->temp2\_gpu) cudaFree(l->temp2\_gpu);

l->temp2\_gpu = cuda\_make\_array(0, batch\*outputs);

if (l->temp3\_gpu) cudaFree(l->temp3\_gpu);

l->temp3\_gpu = cuda\_make\_array(0, batch\*outputs);

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

l->dc\_gpu = cuda\_make\_array(0, batch\*outputs);

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

l->dh\_gpu = cuda\_make\_array(0, batch\*outputs);

if (l->stored\_c\_gpu) cudaFree(l->stored\_c\_gpu);

l->stored\_c\_gpu = cuda\_make\_array(0, batch\*outputs);

if (l->stored\_h\_gpu) cudaFree(l->stored\_h\_gpu);

l->stored\_h\_gpu = cuda\_make\_array(0, batch\*outputs);

if (l->last\_prev\_state\_gpu) cudaFree(l->last\_prev\_state\_gpu);

l->last\_prev\_state\_gpu = cuda\_make\_array(0, batch\*outputs);

if (l->last\_prev\_cell\_gpu) cudaFree(l->last\_prev\_cell\_gpu);

l->last\_prev\_cell\_gpu = cuda\_make\_array(0, batch\*outputs);

#endif

}

void free\_state\_conv\_lstm(layer l)

{

int i;

for (i = 0; i < l.outputs \* l.batch; ++i) l.h\_cpu[i] = 0;

for (i = 0; i < l.outputs \* l.batch; ++i) l.c\_cpu[i] = 0;

#ifdef GPU

cuda\_push\_array(l.h\_gpu, l.h\_cpu, l.outputs \* l.batch);

cuda\_push\_array(l.c\_gpu, l.c\_cpu, l.outputs \* l.batch);

//fill\_ongpu(l.outputs \* l.batch, 0, l.dc\_gpu, 1); // dont use

//fill\_ongpu(l.outputs \* l.batch, 0, l.dh\_gpu, 1); // dont use

#endif // GPU

}

void randomize\_state\_conv\_lstm(layer l)

{

int i;

for (i = 0; i < l.outputs \* l.batch; ++i) l.h\_cpu[i] = rand\_uniform(-1, 1);

for (i = 0; i < l.outputs \* l.batch; ++i) l.c\_cpu[i] = rand\_uniform(-1, 1);

#ifdef GPU

cuda\_push\_array(l.h\_gpu, l.h\_cpu, l.outputs \* l.batch);

cuda\_push\_array(l.c\_gpu, l.c\_cpu, l.outputs \* l.batch);

#endif // GPU

}

void remember\_state\_conv\_lstm(layer l)

{

memcpy(l.stored\_c\_cpu, l.c\_cpu, l.outputs \* l.batch \* sizeof(float));

memcpy(l.stored\_h\_cpu, l.h\_cpu, l.outputs \* l.batch \* sizeof(float));

#ifdef GPU

copy\_ongpu(l.outputs\*l.batch, l.c\_gpu, 1, l.stored\_c\_gpu, 1);

copy\_ongpu(l.outputs\*l.batch, l.h\_gpu, 1, l.stored\_h\_gpu, 1);

#endif // GPU

}

void restore\_state\_conv\_lstm(layer l)

{

memcpy(l.c\_cpu, l.stored\_c\_cpu, l.outputs \* l.batch \* sizeof(float));

memcpy(l.h\_cpu, l.stored\_h\_cpu, l.outputs \* l.batch \* sizeof(float));

#ifdef GPU

copy\_ongpu(l.outputs\*l.batch, l.stored\_c\_gpu, 1, l.c\_gpu, 1);

copy\_ongpu(l.outputs\*l.batch, l.stored\_h\_gpu, 1, l.h\_gpu, 1);

#endif // GPU

}

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

{

network\_state s = { 0 };

s.train = state.train;

s.workspace = state.workspace;

s.net = state.net;

int i;

layer vf = \*(l.vf);

layer vi = \*(l.vi);

layer vo = \*(l.vo);

layer wf = \*(l.wf);

layer wi = \*(l.wi);

layer wg = \*(l.wg);

layer wo = \*(l.wo);

layer uf = \*(l.uf);

layer ui = \*(l.ui);

layer ug = \*(l.ug);

layer uo = \*(l.uo);

if (state.train) {

if (l.peephole) {

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

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

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

}

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

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

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

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

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

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

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

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

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

}

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

{

if (l.peephole) {

assert(l.outputs == vf.out\_w \* vf.out\_h \* vf.out\_c);

s.input = l.c\_cpu;

forward\_convolutional\_layer(vf, s);

forward\_convolutional\_layer(vi, s);

// vo below

}

assert(l.outputs == wf.out\_w \* wf.out\_h \* wf.out\_c);

assert(wf.c == l.out\_c && wi.c == l.out\_c && wg.c == l.out\_c && wo.c == l.out\_c);

s.input = l.h\_cpu;

forward\_convolutional\_layer(wf, s);

forward\_convolutional\_layer(wi, s);

forward\_convolutional\_layer(wg, s);

forward\_convolutional\_layer(wo, s);

assert(l.inputs == uf.w \* uf.h \* uf.c);

assert(uf.c == l.c && ui.c == l.c && ug.c == l.c && uo.c == l.c);

s.input = state.input;

forward\_convolutional\_layer(uf, s);

forward\_convolutional\_layer(ui, s);

forward\_convolutional\_layer(ug, s);

forward\_convolutional\_layer(uo, s);

// f = wf + uf + vf

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

axpy\_cpu(l.outputs\*l.batch, 1, uf.output, 1, l.f\_cpu, 1);

if (l.peephole) axpy\_cpu(l.outputs\*l.batch, 1, vf.output, 1, l.f\_cpu, 1);

// i = wi + ui + vi

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

axpy\_cpu(l.outputs\*l.batch, 1, ui.output, 1, l.i\_cpu, 1);

if (l.peephole) axpy\_cpu(l.outputs\*l.batch, 1, vi.output, 1, l.i\_cpu, 1);

// g = wg + ug

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

axpy\_cpu(l.outputs\*l.batch, 1, ug.output, 1, l.g\_cpu, 1);

activate\_array(l.f\_cpu, l.outputs\*l.batch, LOGISTIC);

activate\_array(l.i\_cpu, l.outputs\*l.batch, LOGISTIC);

activate\_array(l.g\_cpu, l.outputs\*l.batch, TANH);

// c = f\*c + i\*g

copy\_cpu(l.outputs\*l.batch, l.i\_cpu, 1, l.temp\_cpu, 1);

mul\_cpu(l.outputs\*l.batch, l.g\_cpu, 1, l.temp\_cpu, 1);

mul\_cpu(l.outputs\*l.batch, l.f\_cpu, 1, l.c\_cpu, 1);

axpy\_cpu(l.outputs\*l.batch, 1, l.temp\_cpu, 1, l.c\_cpu, 1);

// o = wo + uo + vo(c\_new)

if (l.peephole) {

s.input = l.c\_cpu;

forward\_convolutional\_layer(vo, s);

}

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

axpy\_cpu(l.outputs\*l.batch, 1, uo.output, 1, l.o\_cpu, 1);

if (l.peephole) axpy\_cpu(l.outputs\*l.batch, 1, vo.output, 1, l.o\_cpu, 1);

activate\_array(l.o\_cpu, l.outputs\*l.batch, LOGISTIC);

// h = o \* tanh(c)

copy\_cpu(l.outputs\*l.batch, l.c\_cpu, 1, l.h\_cpu, 1);

activate\_array(l.h\_cpu, l.outputs\*l.batch, TANH);

mul\_cpu(l.outputs\*l.batch, l.o\_cpu, 1, l.h\_cpu, 1);

if (l.state\_constrain) constrain\_cpu(l.outputs\*l.batch, l.state\_constrain, l.c\_cpu);

fix\_nan\_and\_inf\_cpu(l.c\_cpu, l.outputs\*l.batch);

fix\_nan\_and\_inf\_cpu(l.h\_cpu, l.outputs\*l.batch);

copy\_cpu(l.outputs\*l.batch, l.c\_cpu, 1, l.cell\_cpu, 1);

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

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

l.output += l.outputs\*l.batch;

l.cell\_cpu += l.outputs\*l.batch;

if (l.peephole) {

increment\_layer(&vf, 1);

increment\_layer(&vi, 1);

increment\_layer(&vo, 1);

}

increment\_layer(&wf, 1);

increment\_layer(&wi, 1);

increment\_layer(&wg, 1);

increment\_layer(&wo, 1);

increment\_layer(&uf, 1);

increment\_layer(&ui, 1);

increment\_layer(&ug, 1);

increment\_layer(&uo, 1);

}

}

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

{

network\_state s = { 0 };

s.train = state.train;

s.workspace = state.workspace;

int i;

layer vf = \*(l.vf);

layer vi = \*(l.vi);

layer vo = \*(l.vo);

layer wf = \*(l.wf);

layer wi = \*(l.wi);

layer wg = \*(l.wg);

layer wo = \*(l.wo);

layer uf = \*(l.uf);

layer ui = \*(l.ui);

layer ug = \*(l.ug);

layer uo = \*(l.uo);

if (l.peephole) {

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

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

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

}

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

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

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

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

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

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

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

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

state.input += l.inputs\*l.batch\*(l.steps - 1);

if (state.delta) state.delta += l.inputs\*l.batch\*(l.steps - 1);

l.output += l.outputs\*l.batch\*(l.steps - 1);

l.cell\_cpu += l.outputs\*l.batch\*(l.steps - 1);

l.delta += l.outputs\*l.batch\*(l.steps - 1);

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

if (i != 0) copy\_cpu(l.outputs\*l.batch, l.cell\_cpu - l.outputs\*l.batch, 1, l.prev\_cell\_cpu, 1);

copy\_cpu(l.outputs\*l.batch, l.cell\_cpu, 1, l.c\_cpu, 1);

if (i != 0) copy\_cpu(l.outputs\*l.batch, l.output - l.outputs\*l.batch, 1, l.prev\_state\_cpu, 1);

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

l.dh\_cpu = (i == 0) ? 0 : l.delta - l.outputs\*l.batch;

// f = wf + uf + vf

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

axpy\_cpu(l.outputs\*l.batch, 1, uf.output, 1, l.f\_cpu, 1);

if (l.peephole) axpy\_cpu(l.outputs\*l.batch, 1, vf.output, 1, l.f\_cpu, 1);

// i = wi + ui + vi

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

axpy\_cpu(l.outputs\*l.batch, 1, ui.output, 1, l.i\_cpu, 1);

if (l.peephole) axpy\_cpu(l.outputs\*l.batch, 1, vi.output, 1, l.i\_cpu, 1);

// g = wg + ug

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

axpy\_cpu(l.outputs\*l.batch, 1, ug.output, 1, l.g\_cpu, 1);

// o = wo + uo + vo

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

axpy\_cpu(l.outputs\*l.batch, 1, uo.output, 1, l.o\_cpu, 1);

if (l.peephole) axpy\_cpu(l.outputs\*l.batch, 1, vo.output, 1, l.o\_cpu, 1);

activate\_array(l.f\_cpu, l.outputs\*l.batch, LOGISTIC);

activate\_array(l.i\_cpu, l.outputs\*l.batch, LOGISTIC);

activate\_array(l.g\_cpu, l.outputs\*l.batch, TANH);

activate\_array(l.o\_cpu, l.outputs\*l.batch, LOGISTIC);

copy\_cpu(l.outputs\*l.batch, l.delta, 1, l.temp3\_cpu, 1);

copy\_cpu(l.outputs\*l.batch, l.c\_cpu, 1, l.temp\_cpu, 1);

activate\_array(l.temp\_cpu, l.outputs\*l.batch, TANH);

copy\_cpu(l.outputs\*l.batch, l.temp3\_cpu, 1, l.temp2\_cpu, 1);

mul\_cpu(l.outputs\*l.batch, l.o\_cpu, 1, l.temp2\_cpu, 1);

gradient\_array(l.temp\_cpu, l.outputs\*l.batch, TANH, l.temp2\_cpu);

axpy\_cpu(l.outputs\*l.batch, 1, l.dc\_cpu, 1, l.temp2\_cpu, 1);

// temp = tanh(c)

// temp2 = delta \* o \* grad\_tanh(tanh(c))

// temp3 = delta

copy\_cpu(l.outputs\*l.batch, l.c\_cpu, 1, l.temp\_cpu, 1);

activate\_array(l.temp\_cpu, l.outputs\*l.batch, TANH);

mul\_cpu(l.outputs\*l.batch, l.temp3\_cpu, 1, l.temp\_cpu, 1);

gradient\_array(l.o\_cpu, l.outputs\*l.batch, LOGISTIC, l.temp\_cpu);

// delta for o(w,u,v): temp = delta \* tanh(c) \* grad\_logistic(o)

// delta for c,f,i,g(w,u,v): temp2 = delta \* o \* grad\_tanh(tanh(c)) + delta\_c(???)

// delta for output: temp3 = delta

// o

// delta for O(w,u,v): temp = delta \* tanh(c) \* grad\_logistic(o)

if (l.peephole) {

copy\_cpu(l.outputs\*l.batch, l.temp\_cpu, 1, vo.delta, 1);

s.input = l.cell\_cpu;

//s.delta = l.dc\_cpu;

backward\_convolutional\_layer(vo, s);

}

copy\_cpu(l.outputs\*l.batch, l.temp\_cpu, 1, wo.delta, 1);

s.input = l.prev\_state\_cpu;

//s.delta = l.dh\_cpu;

backward\_convolutional\_layer(wo, s);

copy\_cpu(l.outputs\*l.batch, l.temp\_cpu, 1, uo.delta, 1);

s.input = state.input;

s.delta = state.delta;

backward\_convolutional\_layer(uo, s);

// g

copy\_cpu(l.outputs\*l.batch, l.temp2\_cpu, 1, l.temp\_cpu, 1);

mul\_cpu(l.outputs\*l.batch, l.i\_cpu, 1, l.temp\_cpu, 1);

gradient\_array(l.g\_cpu, l.outputs\*l.batch, TANH, l.temp\_cpu);

// delta for c,f,i,g(w,u,v): temp2 = (delta \* o \* grad\_tanh(tanh(c)) + delta\_c(???)) \* g \* grad\_logistic(i)

copy\_cpu(l.outputs\*l.batch, l.temp\_cpu, 1, wg.delta, 1);

s.input = l.prev\_state\_cpu;

//s.delta = l.dh\_cpu;

backward\_convolutional\_layer(wg, s);

copy\_cpu(l.outputs\*l.batch, l.temp\_cpu, 1, ug.delta, 1);

s.input = state.input;

s.delta = state.delta;

backward\_convolutional\_layer(ug, s);

// i

copy\_cpu(l.outputs\*l.batch, l.temp2\_cpu, 1, l.temp\_cpu, 1);

mul\_cpu(l.outputs\*l.batch, l.g\_cpu, 1, l.temp\_cpu, 1);

gradient\_array(l.i\_cpu, l.outputs\*l.batch, LOGISTIC, l.temp\_cpu);

// delta for c,f,i,g(w,u,v): temp2 = (delta \* o \* grad\_tanh(tanh(c)) + delta\_c(???)) \* g \* grad\_logistic(i)

if (l.peephole) {

copy\_cpu(l.outputs\*l.batch, l.temp\_cpu, 1, vi.delta, 1);

s.input = l.prev\_cell\_cpu;

//s.delta = l.dc\_cpu;

backward\_convolutional\_layer(vi, s);

}

copy\_cpu(l.outputs\*l.batch, l.temp\_cpu, 1, wi.delta, 1);

s.input = l.prev\_state\_cpu;

//s.delta = l.dh\_cpu;

backward\_convolutional\_layer(wi, s);

copy\_cpu(l.outputs\*l.batch, l.temp\_cpu, 1, ui.delta, 1);

s.input = state.input;

s.delta = state.delta;

backward\_convolutional\_layer(ui, s);

// f

copy\_cpu(l.outputs\*l.batch, l.temp2\_cpu, 1, l.temp\_cpu, 1);

mul\_cpu(l.outputs\*l.batch, l.prev\_cell\_cpu, 1, l.temp\_cpu, 1);

gradient\_array(l.f\_cpu, l.outputs\*l.batch, LOGISTIC, l.temp\_cpu);

// delta for c,f,i,g(w,u,v): temp2 = (delta \* o \* grad\_tanh(tanh(c)) + delta\_c(???)) \* c \* grad\_logistic(f)

if (l.peephole) {

copy\_cpu(l.outputs\*l.batch, l.temp\_cpu, 1, vf.delta, 1);

s.input = l.prev\_cell\_cpu;

//s.delta = l.dc\_cpu;

backward\_convolutional\_layer(vf, s);

}

copy\_cpu(l.outputs\*l.batch, l.temp\_cpu, 1, wf.delta, 1);

s.input = l.prev\_state\_cpu;

//s.delta = l.dh\_cpu;

backward\_convolutional\_layer(wf, s);

copy\_cpu(l.outputs\*l.batch, l.temp\_cpu, 1, uf.delta, 1);

s.input = state.input;

s.delta = state.delta;

backward\_convolutional\_layer(uf, s);

copy\_cpu(l.outputs\*l.batch, l.temp2\_cpu, 1, l.temp\_cpu, 1);

mul\_cpu(l.outputs\*l.batch, l.f\_cpu, 1, l.temp\_cpu, 1);

copy\_cpu(l.outputs\*l.batch, l.temp\_cpu, 1, l.dc\_cpu, 1);

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

if (state.delta) state.delta -= l.inputs\*l.batch;

l.output -= l.outputs\*l.batch;

l.cell\_cpu -= l.outputs\*l.batch;

l.delta -= l.outputs\*l.batch;

if (l.peephole) {

increment\_layer(&vf, -1);

increment\_layer(&vi, -1);

increment\_layer(&vo, -1);

}

increment\_layer(&wf, -1);

increment\_layer(&wi, -1);

increment\_layer(&wg, -1);

increment\_layer(&wo, -1);

increment\_layer(&uf, -1);

increment\_layer(&ui, -1);

increment\_layer(&ug, -1);

increment\_layer(&uo, -1);

}

}

#ifdef GPU

void pull\_conv\_lstm\_layer(layer l)

{

if (l.peephole) {

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

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

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

}

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

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

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

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

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

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

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

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

}

void push\_conv\_lstm\_layer(layer l)

{

if (l.peephole) {

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

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

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

}

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

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

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

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

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

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

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

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

}

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

{

if (l.peephole) {

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

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

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

}

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

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

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

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

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

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

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

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

}

void forward\_conv\_lstm\_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 vf = \*(l.vf);

layer vi = \*(l.vi);

layer vo = \*(l.vo);

layer wf = \*(l.wf);

layer wi = \*(l.wi);

layer wg = \*(l.wg);

layer wo = \*(l.wo);

layer uf = \*(l.uf);

layer ui = \*(l.ui);

layer ug = \*(l.ug);

layer uo = \*(l.uo);

if (state.train) {

if (l.peephole) {

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

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

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

}

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

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

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

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

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

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

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

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

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

}

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

{

if (l.peephole) {

assert(l.outputs == vf.out\_w \* vf.out\_h \* vf.out\_c);

s.input = l.c\_gpu;

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

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

// vo below

}

assert(l.outputs == wf.out\_w \* wf.out\_h \* wf.out\_c);

assert(wf.c == l.out\_c && wi.c == l.out\_c && wg.c == l.out\_c && wo.c == l.out\_c);

s.input = l.h\_gpu;

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

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

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

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

assert(l.inputs == uf.w \* uf.h \* uf.c);

assert(uf.c == l.c && ui.c == l.c && ug.c == l.c && uo.c == l.c);

s.input = state.input;

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

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

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

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

// f = wf + uf + vf

add\_3\_arrays\_activate(wf.output\_gpu, uf.output\_gpu, (l.peephole)?vf.output\_gpu:NULL, l.outputs\*l.batch, LOGISTIC, l.f\_gpu);

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

//axpy\_ongpu(l.outputs\*l.batch, 1, uf.output\_gpu, 1, l.f\_gpu, 1);

//if (l.peephole) axpy\_ongpu(l.outputs\*l.batch, 1, vf.output\_gpu, 1, l.f\_gpu, 1);

//activate\_array\_ongpu(l.f\_gpu, l.outputs\*l.batch, LOGISTIC);

// i = wi + ui + vi

add\_3\_arrays\_activate(wi.output\_gpu, ui.output\_gpu, (l.peephole) ? vi.output\_gpu : NULL, l.outputs\*l.batch, LOGISTIC, l.i\_gpu);

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

//axpy\_ongpu(l.outputs\*l.batch, 1, ui.output\_gpu, 1, l.i\_gpu, 1);

//if (l.peephole) axpy\_ongpu(l.outputs\*l.batch, 1, vi.output\_gpu, 1, l.i\_gpu, 1);

//activate\_array\_ongpu(l.i\_gpu, l.outputs\*l.batch, LOGISTIC);

// g = wg + ug

add\_3\_arrays\_activate(wg.output\_gpu, ug.output\_gpu, NULL, l.outputs\*l.batch, TANH, l.g\_gpu);

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

//axpy\_ongpu(l.outputs\*l.batch, 1, ug.output\_gpu, 1, l.g\_gpu, 1);

//activate\_array\_ongpu(l.g\_gpu, l.outputs\*l.batch, TANH);

// c = f\*c + i\*g

sum\_of\_mults(l.f\_gpu, l.c\_gpu, l.i\_gpu, l.g\_gpu, l.outputs\*l.batch, l.c\_gpu); // decreases mAP???

//copy\_ongpu(l.outputs\*l.batch, l.i\_gpu, 1, l.temp\_gpu, 1);

//mul\_ongpu(l.outputs\*l.batch, l.g\_gpu, 1, l.temp\_gpu, 1);

//mul\_ongpu(l.outputs\*l.batch, l.f\_gpu, 1, l.c\_gpu, 1);

//axpy\_ongpu(l.outputs\*l.batch, 1, l.temp\_gpu, 1, l.c\_gpu, 1);

// o = wo + uo + vo(c\_new)

if (l.peephole) {

s.input = l.c\_gpu;

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

}

add\_3\_arrays\_activate(wo.output\_gpu, uo.output\_gpu, (l.peephole) ? vo.output\_gpu : NULL, l.outputs\*l.batch, LOGISTIC, l.o\_gpu);

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

//axpy\_ongpu(l.outputs\*l.batch, 1, uo.output\_gpu, 1, l.o\_gpu, 1);

//if (l.peephole) axpy\_ongpu(l.outputs\*l.batch, 1, vo.output\_gpu, 1, l.o\_gpu, 1);

//activate\_array\_ongpu(l.o\_gpu, l.outputs\*l.batch, LOGISTIC);

// h = o \* tanh(c)

activate\_and\_mult(l.c\_gpu, l.o\_gpu, l.outputs\*l.batch, TANH, l.h\_gpu);

//simple\_copy\_ongpu(l.outputs\*l.batch, l.c\_gpu, l.h\_gpu);

//activate\_array\_ongpu(l.h\_gpu, l.outputs\*l.batch, TANH);

//mul\_ongpu(l.outputs\*l.batch, l.o\_gpu, 1, l.h\_gpu, 1);

fix\_nan\_and\_inf(l.c\_gpu, l.outputs\*l.batch);

fix\_nan\_and\_inf(l.h\_gpu, l.outputs\*l.batch);

if (l.state\_constrain) constrain\_ongpu(l.outputs\*l.batch, l.state\_constrain, l.c\_gpu, 1);

if(state.train) simple\_copy\_ongpu(l.outputs\*l.batch, l.c\_gpu, l.cell\_gpu);

simple\_copy\_ongpu(l.outputs\*l.batch, l.h\_gpu, l.output\_gpu); // is required for both Detection and Training

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

l.output\_gpu += l.outputs\*l.batch;

l.cell\_gpu += l.outputs\*l.batch;

if (l.peephole) {

increment\_layer(&vf, 1);

increment\_layer(&vi, 1);

increment\_layer(&vo, 1);

}

increment\_layer(&wf, 1);

increment\_layer(&wi, 1);

increment\_layer(&wg, 1);

increment\_layer(&wo, 1);

increment\_layer(&uf, 1);

increment\_layer(&ui, 1);

increment\_layer(&ug, 1);

increment\_layer(&uo, 1);

}

}

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

{

float \*last\_output = l.output\_gpu + l.outputs\*l.batch\*(l.steps - 1);

float \*last\_cell = l.cell\_gpu + l.outputs\*l.batch\*(l.steps - 1);

network\_state s = { 0 };

s.train = state.train;

s.workspace = state.workspace;

s.net = state.net;

int i;

layer vf = \*(l.vf);

layer vi = \*(l.vi);

layer vo = \*(l.vo);

layer wf = \*(l.wf);

layer wi = \*(l.wi);

layer wg = \*(l.wg);

layer wo = \*(l.wo);

layer uf = \*(l.uf);

layer ui = \*(l.ui);

layer ug = \*(l.ug);

layer uo = \*(l.uo);

if (l.peephole) {

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

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

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

}

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

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

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

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

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

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

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

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

state.input += l.inputs\*l.batch\*(l.steps - 1);

if (state.delta) state.delta += l.inputs\*l.batch\*(l.steps - 1);

l.output\_gpu += l.outputs\*l.batch\*(l.steps - 1);

l.cell\_gpu += l.outputs\*l.batch\*(l.steps - 1);

l.delta\_gpu += l.outputs\*l.batch\*(l.steps - 1);

//fill\_ongpu(l.outputs \* l.batch, 0, l.dc\_gpu, 1); // dont use

const int sequence = get\_sequence\_value(state.net);

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

if (i != 0) simple\_copy\_ongpu(l.outputs\*l.batch, l.cell\_gpu - l.outputs\*l.batch, l.prev\_cell\_gpu);

//else fill\_ongpu(l.outputs \* l.batch, 0, l.prev\_cell\_gpu, 1); // dont use

else if (state.net.current\_subdivision % sequence != 0) simple\_copy\_ongpu(l.outputs\*l.batch, l.last\_prev\_cell\_gpu, l.prev\_cell\_gpu);

simple\_copy\_ongpu(l.outputs\*l.batch, l.cell\_gpu, l.c\_gpu);

if (i != 0) simple\_copy\_ongpu(l.outputs\*l.batch, l.output\_gpu - l.outputs\*l.batch, l.prev\_state\_gpu);

//else fill\_ongpu(l.outputs \* l.batch, 0, l.prev\_state\_gpu, 1); // dont use

else if(state.net.current\_subdivision % sequence != 0) simple\_copy\_ongpu(l.outputs\*l.batch, l.last\_prev\_state\_gpu, l.prev\_state\_gpu);

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

l.dh\_gpu = (i == 0) ? 0 : l.delta\_gpu - l.outputs\*l.batch;

// f = wf + uf + vf

add\_3\_arrays\_activate(wf.output\_gpu, uf.output\_gpu, (l.peephole) ? vf.output\_gpu : NULL, l.outputs\*l.batch, LOGISTIC, l.f\_gpu);

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

//axpy\_ongpu(l.outputs\*l.batch, 1, uf.output\_gpu, 1, l.f\_gpu, 1);

//if (l.peephole) axpy\_ongpu(l.outputs\*l.batch, 1, vf.output\_gpu, 1, l.f\_gpu, 1);

//activate\_array\_ongpu(l.f\_gpu, l.outputs\*l.batch, LOGISTIC);

// i = wi + ui + vi

add\_3\_arrays\_activate(wi.output\_gpu, ui.output\_gpu, (l.peephole) ? vi.output\_gpu : NULL, l.outputs\*l.batch, LOGISTIC, l.i\_gpu);

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

//axpy\_ongpu(l.outputs\*l.batch, 1, ui.output\_gpu, 1, l.i\_gpu, 1);

//if (l.peephole) axpy\_ongpu(l.outputs\*l.batch, 1, vi.output\_gpu, 1, l.i\_gpu, 1);

//activate\_array\_ongpu(l.i\_gpu, l.outputs\*l.batch, LOGISTIC);

// g = wg + ug

add\_3\_arrays\_activate(wg.output\_gpu, ug.output\_gpu, NULL, l.outputs\*l.batch, TANH, l.g\_gpu);

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

//axpy\_ongpu(l.outputs\*l.batch, 1, ug.output\_gpu, 1, l.g\_gpu, 1);

//activate\_array\_ongpu(l.g\_gpu, l.outputs\*l.batch, TANH);

// o = wo + uo + vo

add\_3\_arrays\_activate(wo.output\_gpu, uo.output\_gpu, (l.peephole) ? vo.output\_gpu : NULL, l.outputs\*l.batch, LOGISTIC, l.o\_gpu);

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

//axpy\_ongpu(l.outputs\*l.batch, 1, uo.output\_gpu, 1, l.o\_gpu, 1);

//if (l.peephole) axpy\_ongpu(l.outputs\*l.batch, 1, vo.output\_gpu, 1, l.o\_gpu, 1);

//activate\_array\_ongpu(l.o\_gpu, l.outputs\*l.batch, LOGISTIC);

simple\_copy\_ongpu(l.outputs\*l.batch, l.delta\_gpu, l.temp3\_gpu); // temp3 = delta

simple\_copy\_ongpu(l.outputs\*l.batch, l.c\_gpu, l.temp\_gpu);

activate\_array\_ongpu(l.temp\_gpu, l.outputs\*l.batch, TANH); // temp = tanh(c)

simple\_copy\_ongpu(l.outputs\*l.batch, l.temp3\_gpu, l.temp2\_gpu);

mul\_ongpu(l.outputs\*l.batch, l.o\_gpu, 1, l.temp2\_gpu, 1); // temp2 = delta \* o

gradient\_array\_ongpu(l.temp\_gpu, l.outputs\*l.batch, TANH, l.temp2\_gpu); // temp2 = delta \* o \* grad\_tanh(tanh(c))

//???

axpy\_ongpu(l.outputs\*l.batch, 1, l.dc\_gpu, 1, l.temp2\_gpu, 1); // temp2 = delta \* o \* grad\_tanh(tanh(c)) + delta\_c(???)

// temp = tanh(c)

// temp2 = delta \* o \* grad\_tanh(tanh(c)) + delta\_c(???)

// temp3 = delta

simple\_copy\_ongpu(l.outputs\*l.batch, l.c\_gpu, l.temp\_gpu);

activate\_array\_ongpu(l.temp\_gpu, l.outputs\*l.batch, TANH); // temp = tanh(c)

mul\_ongpu(l.outputs\*l.batch, l.temp3\_gpu, 1, l.temp\_gpu, 1); // temp = delta \* tanh(c)

gradient\_array\_ongpu(l.o\_gpu, l.outputs\*l.batch, LOGISTIC, l.temp\_gpu); // temp = delta \* tanh(c) \* grad\_logistic(o)

// delta for o(w,u,v): temp = delta \* tanh(c) \* grad\_logistic(o)

// delta for c,f,i,g(w,u,v): temp2 = delta \* o \* grad\_tanh(tanh(c)) + delta\_c(???)

// delta for output: temp3 = delta

// o

// delta for O(w,u,v): temp = delta \* tanh(c) \* grad\_logistic(o)

if (l.peephole) {

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

s.input = l.cell\_gpu;

//s.delta = l.dc\_gpu;

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

}

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

s.input = l.prev\_state\_gpu;

//s.delta = l.dh\_gpu;

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

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

s.input = state.input;

s.delta = state.delta;

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

// g

simple\_copy\_ongpu(l.outputs\*l.batch, l.temp2\_gpu, l.temp\_gpu);

mul\_ongpu(l.outputs\*l.batch, l.i\_gpu, 1, l.temp\_gpu, 1);

gradient\_array\_ongpu(l.g\_gpu, l.outputs\*l.batch, TANH, l.temp\_gpu);

// delta for c,f,i,g(w,u,v): temp = (delta \* o \* grad\_tanh(tanh(c)) + delta\_c(???)) \* i \* grad\_tanh(g)

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

s.input = l.prev\_state\_gpu;

//s.delta = l.dh\_gpu;

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

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

s.input = state.input;

s.delta = state.delta;

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

// i

simple\_copy\_ongpu(l.outputs\*l.batch, l.temp2\_gpu, l.temp\_gpu);

mul\_ongpu(l.outputs\*l.batch, l.g\_gpu, 1, l.temp\_gpu, 1);

gradient\_array\_ongpu(l.i\_gpu, l.outputs\*l.batch, LOGISTIC, l.temp\_gpu);

// delta for c,f,i,g(w,u,v): temp = (delta \* o \* grad\_tanh(tanh(c)) + delta\_c(???)) \* g \* grad\_logistic(i)

if (l.peephole) {

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

s.input = l.prev\_cell\_gpu;

//s.delta = l.dc\_gpu;

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

}

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

s.input = l.prev\_state\_gpu;

//s.delta = l.dh\_gpu;

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

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

s.input = state.input;

s.delta = state.delta;

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

// f

simple\_copy\_ongpu(l.outputs\*l.batch, l.temp2\_gpu, l.temp\_gpu);

mul\_ongpu(l.outputs\*l.batch, l.prev\_cell\_gpu, 1, l.temp\_gpu, 1);

gradient\_array\_ongpu(l.f\_gpu, l.outputs\*l.batch, LOGISTIC, l.temp\_gpu);

// delta for c,f,i,g(w,u,v): temp = (delta \* o \* grad\_tanh(tanh(c)) + delta\_c(???)) \* c \* grad\_logistic(f)

if (l.peephole) {

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

s.input = l.prev\_cell\_gpu;

//s.delta = l.dc\_gpu;

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

}

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

s.input = l.prev\_state\_gpu;

//s.delta = l.dh\_gpu;

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

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

s.input = state.input;

s.delta = state.delta;

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

// c

simple\_copy\_ongpu(l.outputs\*l.batch, l.temp2\_gpu, l.temp\_gpu);

mul\_ongpu(l.outputs\*l.batch, l.f\_gpu, 1, l.temp\_gpu, 1);

simple\_copy\_ongpu(l.outputs\*l.batch, l.temp\_gpu, l.dc\_gpu);

fix\_nan\_and\_inf(l.dc\_gpu, l.outputs\*l.batch);

// delta for c,f,i,g(w,u,v): delta\_c = temp = (delta \* o \* grad\_tanh(tanh(c)) + delta\_c(???)) \* f // (grad\_linear(c)==1)

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

if (state.delta) state.delta -= l.inputs\*l.batch; // new delta: state.delta = prev\_layer.delta\_gpu;

l.output\_gpu -= l.outputs\*l.batch;

l.cell\_gpu -= l.outputs\*l.batch;

l.delta\_gpu -= l.outputs\*l.batch;

if (l.peephole) {

increment\_layer(&vf, -1);

increment\_layer(&vi, -1);

increment\_layer(&vo, -1);

}

increment\_layer(&wf, -1);

increment\_layer(&wi, -1);

increment\_layer(&wg, -1);

increment\_layer(&wo, -1);

increment\_layer(&uf, -1);

increment\_layer(&ui, -1);

increment\_layer(&ug, -1);

increment\_layer(&uo, -1);

}

simple\_copy\_ongpu(l.outputs\*l.batch, last\_output, l.last\_prev\_state\_gpu);

simple\_copy\_ongpu(l.outputs\*l.batch, last\_cell, l.last\_prev\_cell\_gpu);

// free state after each 100 iterations

//if (get\_current\_batch(state.net) % 100) free\_state\_conv\_lstm(l); // dont use

}

#endif