#include "blas.h"

#include "utils.h"

#include <math.h>

#include <assert.h>

#include <float.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

void reorg\_cpu(float \*x, int out\_w, int out\_h, int out\_c, int batch, int stride, int forward, float \*out)

{

int b,i,j,k;

int in\_c = out\_c/(stride\*stride);

//printf("\n out\_c = %d, out\_w = %d, out\_h = %d, stride = %d, forward = %d \n", out\_c, out\_w, out\_h, stride, forward);

//printf(" in\_c = %d, in\_w = %d, in\_h = %d \n", in\_c, out\_w\*stride, out\_h\*stride);

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

for(k = 0; k < out\_c; ++k){

for(j = 0; j < out\_h; ++j){

for(i = 0; i < out\_w; ++i){

int in\_index = i + out\_w\*(j + out\_h\*(k + out\_c\*b));

int c2 = k % in\_c;

int offset = k / in\_c;

int w2 = i\*stride + offset % stride;

int h2 = j\*stride + offset / stride;

int out\_index = w2 + out\_w\*stride\*(h2 + out\_h\*stride\*(c2 + in\_c\*b));

if(forward) out[out\_index] = x[in\_index]; // used by default for forward (i.e. forward = 0)

else out[in\_index] = x[out\_index];

}

}

}

}

}

void flatten(float \*x, int size, int layers, int batch, int forward)

{

float\* swap = (float\*)xcalloc(size \* layers \* batch, sizeof(float));

int i,c,b;

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

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

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

int i1 = b\*layers\*size + c\*size + i;

int i2 = b\*layers\*size + i\*layers + c;

if (forward) swap[i2] = x[i1];

else swap[i1] = x[i2];

}

}

}

memcpy(x, swap, size\*layers\*batch\*sizeof(float));

free(swap);

}

void weighted\_sum\_cpu(float \*a, float \*b, float \*s, int n, float \*c)

{

int i;

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

c[i] = s[i]\*a[i] + (1-s[i])\*(b ? b[i] : 0);

}

}

void weighted\_delta\_cpu(float \*a, float \*b, float \*s, float \*da, float \*db, float \*ds, int n, float \*dc)

{

int i;

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

if(da) da[i] += dc[i] \* s[i];

if(db) db[i] += dc[i] \* (1-s[i]);

ds[i] += dc[i] \* (a[i] - b[i]);

}

}

static float relu(float src) {

if (src > 0) return src;

return 0;

}

void shortcut\_multilayer\_cpu(int size, int src\_outputs, int batch, int n, int \*outputs\_of\_layers, float \*\*layers\_output, float \*out, float \*in, float \*weights, int nweights, WEIGHTS\_NORMALIZATION\_T weights\_normalizion)

{

// nweights - l.n or l.n\*l.c or (l.n\*l.c\*l.h\*l.w)

const int layer\_step = nweights / (n + 1); // 1 or l.c or (l.c \* l.h \* l.w)

int step = 0;

if (nweights > 0) step = src\_outputs / layer\_step; // (l.c \* l.h \* l.w) or (l.w\*l.h) or 1

int id;

#pragma omp parallel for

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

int src\_id = id;

const int src\_i = src\_id % src\_outputs;

src\_id /= src\_outputs;

int src\_b = src\_id;

float sum = 1, max\_val = -FLT\_MAX;

int i;

if (weights && weights\_normalizion) {

if (weights\_normalizion == SOFTMAX\_NORMALIZATION) {

for (i = 0; i < (n + 1); ++i) {

const int weights\_index = src\_i / step + i\*layer\_step; // [0 or c or (c, h ,w)]

float w = weights[weights\_index];

if (max\_val < w) max\_val = w;

}

}

const float eps = 0.0001;

sum = eps;

for (i = 0; i < (n + 1); ++i) {

const int weights\_index = src\_i / step + i\*layer\_step; // [0 or c or (c, h ,w)]

const float w = weights[weights\_index];

if (weights\_normalizion == RELU\_NORMALIZATION) sum += relu(w);

else if (weights\_normalizion == SOFTMAX\_NORMALIZATION) sum += expf(w - max\_val);

}

}

if (weights) {

float w = weights[src\_i / step];

if (weights\_normalizion == RELU\_NORMALIZATION) w = relu(w) / sum;

else if (weights\_normalizion == SOFTMAX\_NORMALIZATION) w = expf(w - max\_val) / sum;

out[id] = in[id] \* w; // [0 or c or (c, h ,w)]

}

else out[id] = in[id];

// layers

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

int add\_outputs = outputs\_of\_layers[i];

if (src\_i < add\_outputs) {

int add\_index = add\_outputs\*src\_b + src\_i;

int out\_index = id;

float \*add = layers\_output[i];

if (weights) {

const int weights\_index = src\_i / step + (i + 1)\*layer\_step; // [0 or c or (c, h ,w)]

float w = weights[weights\_index];

if (weights\_normalizion == RELU\_NORMALIZATION) w = relu(w) / sum;

else if (weights\_normalizion == SOFTMAX\_NORMALIZATION) w = expf(w - max\_val) / sum;

out[out\_index] += add[add\_index] \* w; // [0 or c or (c, h ,w)]

}

else out[out\_index] += add[add\_index];

}

}

}

}

void backward\_shortcut\_multilayer\_cpu(int size, int src\_outputs, int batch, int n, int \*outputs\_of\_layers,

float \*\*layers\_delta, float \*delta\_out, float \*delta\_in, float \*weights, float \*weight\_updates, int nweights, float \*in, float \*\*layers\_output, WEIGHTS\_NORMALIZATION\_T weights\_normalizion)

{

// nweights - l.n or l.n\*l.c or (l.n\*l.c\*l.h\*l.w)

const int layer\_step = nweights / (n + 1); // 1 or l.c or (l.c \* l.h \* l.w)

int step = 0;

if (nweights > 0) step = src\_outputs / layer\_step; // (l.c \* l.h \* l.w) or (l.w\*l.h) or 1

int id;

#pragma omp parallel for

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

int src\_id = id;

int src\_i = src\_id % src\_outputs;

src\_id /= src\_outputs;

int src\_b = src\_id;

float grad = 1, sum = 1, max\_val = -FLT\_MAX;;

int i;

if (weights && weights\_normalizion) {

if (weights\_normalizion == SOFTMAX\_NORMALIZATION) {

for (i = 0; i < (n + 1); ++i) {

const int weights\_index = src\_i / step + i\*layer\_step; // [0 or c or (c, h ,w)]

float w = weights[weights\_index];

if (max\_val < w) max\_val = w;

}

}

const float eps = 0.0001;

sum = eps;

for (i = 0; i < (n + 1); ++i) {

const int weights\_index = src\_i / step + i\*layer\_step; // [0 or c or (c, h ,w)]

const float w = weights[weights\_index];

if (weights\_normalizion == RELU\_NORMALIZATION) sum += relu(w);

else if (weights\_normalizion == SOFTMAX\_NORMALIZATION) sum += expf(w - max\_val);

}

/\*

grad = 0;

for (i = 0; i < (n + 1); ++i) {

const int weights\_index = src\_i / step + i\*layer\_step; // [0 or c or (c, h ,w)]

const float delta\_w = delta\_in[id] \* in[id];

const float w = weights[weights\_index];

if (weights\_normalizion == RELU\_NORMALIZATION) grad += delta\_w \* relu(w) / sum;

else if (weights\_normalizion == SOFTMAX\_NORMALIZATION) grad += delta\_w \* expf(w - max\_val) / sum;

}

\*/

}

if (weights) {

float w = weights[src\_i / step];

if (weights\_normalizion == RELU\_NORMALIZATION) w = relu(w) / sum;

else if (weights\_normalizion == SOFTMAX\_NORMALIZATION) w = expf(w - max\_val) / sum;

delta\_out[id] += delta\_in[id] \* w; // [0 or c or (c, h ,w)]

weight\_updates[src\_i / step] += delta\_in[id] \* in[id] \* grad;

}

else delta\_out[id] += delta\_in[id];

// layers

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

int add\_outputs = outputs\_of\_layers[i];

if (src\_i < add\_outputs) {

int add\_index = add\_outputs\*src\_b + src\_i;

int out\_index = id;

float \*layer\_delta = layers\_delta[i];

if (weights) {

float \*add = layers\_output[i];

const int weights\_index = src\_i / step + (i + 1)\*layer\_step; // [0 or c or (c, h ,w)]

float w = weights[weights\_index];

if (weights\_normalizion == RELU\_NORMALIZATION) w = relu(w) / sum;

else if (weights\_normalizion == SOFTMAX\_NORMALIZATION) w = expf(w - max\_val) / sum;

layer\_delta[add\_index] += delta\_in[id] \* w; // [0 or c or (c, h ,w)]

weight\_updates[weights\_index] += delta\_in[id] \* add[add\_index] \* grad;

}

else layer\_delta[add\_index] += delta\_in[id];

}

}

}

}

void shortcut\_cpu(int batch, int w1, int h1, int c1, float \*add, int w2, int h2, int c2, float \*out)

{

int stride = w1/w2;

int sample = w2/w1;

assert(stride == h1/h2);

assert(sample == h2/h1);

if(stride < 1) stride = 1;

if(sample < 1) sample = 1;

int minw = (w1 < w2) ? w1 : w2;

int minh = (h1 < h2) ? h1 : h2;

int minc = (c1 < c2) ? c1 : c2;

int i,j,k,b;

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

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

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

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

int out\_index = i\*sample + w2\*(j\*sample + h2\*(k + c2\*b));

int add\_index = i\*stride + w1\*(j\*stride + h1\*(k + c1\*b));

out[out\_index] += add[add\_index];

}

}

}

}

}

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

{

float scale = 1./(batch \* spatial);

int i,j,k;

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

mean[i] = 0;

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

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

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

mean[i] += x[index];

}

}

mean[i] \*= scale;

}

}

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

{

float scale = 1./(batch \* spatial - 1);

int i,j,k;

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

variance[i] = 0;

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

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

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

variance[i] += pow((x[index] - mean[i]), 2);

}

}

variance[i] \*= scale;

}

}

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

{

int b, f, i;

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

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

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

int index = b\*filters\*spatial + f\*spatial + i;

x[index] = (x[index] - mean[f])/(sqrt(variance[f] + .000001f));

}

}

}

}

void const\_cpu(int N, float ALPHA, float \*X, int INCX)

{

int i;

for(i = 0; i < N; ++i) X[i\*INCX] = ALPHA;

}

void mul\_cpu(int N, float \*X, int INCX, float \*Y, int INCY)

{

int i;

for(i = 0; i < N; ++i) Y[i\*INCY] \*= X[i\*INCX];

}

void pow\_cpu(int N, float ALPHA, float \*X, int INCX, float \*Y, int INCY)

{

int i;

for(i = 0; i < N; ++i) Y[i\*INCY] = pow(X[i\*INCX], ALPHA);

}

void axpy\_cpu(int N, float ALPHA, float \*X, int INCX, float \*Y, int INCY)

{

int i;

for(i = 0; i < N; ++i) Y[i\*INCY] += ALPHA\*X[i\*INCX];

}

void scal\_cpu(int N, float ALPHA, float \*X, int INCX)

{

int i;

for(i = 0; i < N; ++i) X[i\*INCX] \*= ALPHA;

}

void scal\_add\_cpu(int N, float ALPHA, float BETA, float \*X, int INCX)

{

int i;

for (i = 0; i < N; ++i) X[i\*INCX] = X[i\*INCX] \* ALPHA + BETA;

}

void fill\_cpu(int N, float ALPHA, float \*X, int INCX)

{

int i;

if (INCX == 1 && ALPHA == 0) {

memset(X, 0, N \* sizeof(float));

}

else {

for (i = 0; i < N; ++i) X[i\*INCX] = ALPHA;

}

}

void deinter\_cpu(int NX, float \*X, int NY, float \*Y, int B, float \*OUT)

{

int i, j;

int index = 0;

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

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

if(X) X[j\*NX + i] += OUT[index];

++index;

}

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

if(Y) Y[j\*NY + i] += OUT[index];

++index;

}

}

}

void inter\_cpu(int NX, float \*X, int NY, float \*Y, int B, float \*OUT)

{

int i, j;

int index = 0;

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

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

OUT[index++] = X[j\*NX + i];

}

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

OUT[index++] = Y[j\*NY + i];

}

}

}

void copy\_cpu(int N, float \*X, int INCX, float \*Y, int INCY)

{

int i;

for(i = 0; i < N; ++i) Y[i\*INCY] = X[i\*INCX];

}

void mult\_add\_into\_cpu(int N, float \*X, float \*Y, float \*Z)

{

int i;

for(i = 0; i < N; ++i) Z[i] += X[i]\*Y[i];

}

void smooth\_l1\_cpu(int n, float \*pred, float \*truth, float \*delta, float \*error)

{

int i;

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

float diff = truth[i] - pred[i];

float abs\_val = fabs(diff);

if(abs\_val < 1) {

error[i] = diff \* diff;

delta[i] = diff;

}

else {

error[i] = 2\*abs\_val - 1;

delta[i] = (diff > 0) ? 1 : -1;

}

}

}

void l1\_cpu(int n, float \*pred, float \*truth, float \*delta, float \*error)

{

int i;

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

float diff = truth[i] - pred[i];

error[i] = fabs(diff);

delta[i] = diff > 0 ? 1 : -1;

}

}

void softmax\_x\_ent\_cpu(int n, float \*pred, float \*truth, float \*delta, float \*error)

{

int i;

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

float t = truth[i];

float p = pred[i];

error[i] = (t) ? -log(p) : 0;

delta[i] = t-p;

}

}

void logistic\_x\_ent\_cpu(int n, float \*pred, float \*truth, float \*delta, float \*error)

{

int i;

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

float t = truth[i];

float p = pred[i];

error[i] = -t\*log(p) - (1-t)\*log(1-p);

delta[i] = t-p;

}

}

void l2\_cpu(int n, float \*pred, float \*truth, float \*delta, float \*error)

{

int i;

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

float diff = truth[i] - pred[i];

error[i] = diff \* diff;

delta[i] = diff;

}

}

float dot\_cpu(int N, float \*X, int INCX, float \*Y, int INCY)

{

int i;

float dot = 0;

for(i = 0; i < N; ++i) dot += X[i\*INCX] \* Y[i\*INCY];

return dot;

}

void softmax(float \*input, int n, float temp, float \*output, int stride)

{

int i;

float sum = 0;

float largest = -FLT\_MAX;

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

if(input[i\*stride] > largest) largest = input[i\*stride];

}

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

float e = exp(input[i\*stride]/temp - largest/temp);

sum += e;

output[i\*stride] = e;

}

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

output[i\*stride] /= sum;

}

}

void softmax\_cpu(float \*input, int n, int batch, int batch\_offset, int groups, int group\_offset, int stride, float temp, float \*output)

{

int g, b;

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

for(g = 0; g < groups; ++g){

softmax(input + b\*batch\_offset + g\*group\_offset, n, temp, output + b\*batch\_offset + g\*group\_offset, stride);

}

}

}

void upsample\_cpu(float \*in, int w, int h, int c, int batch, int stride, int forward, float scale, float \*out)

{

int i, j, k, b;

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

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

for (j = 0; j < h\*stride; ++j) {

for (i = 0; i < w\*stride; ++i) {

int in\_index = b\*w\*h\*c + k\*w\*h + (j / stride)\*w + i / stride;

int out\_index = b\*w\*h\*c\*stride\*stride + k\*w\*h\*stride\*stride + j\*w\*stride + i;

if (forward) out[out\_index] = scale\*in[in\_index];

else in[in\_index] += scale\*out[out\_index];

}

}

}

}

}

void constrain\_cpu(int size, float ALPHA, float \*X)

{

int i;

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

X[i] = fminf(ALPHA, fmaxf(-ALPHA, X[i]));

}

}

void fix\_nan\_and\_inf\_cpu(float \*input, size\_t size)

{

int i;

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

float val = input[i];

if (isnan(val) || isinf(val))

input[i] = 1.0f / i; // pseudo random value

}

}