#include "activations.h"

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <float.h>

char \*get\_activation\_string(ACTIVATION a)

{

switch(a){

case LOGISTIC:

return "logistic";

case LOGGY:

return "loggy";

case RELU:

return "relu";

case ELU:

return "elu";

case SELU:

return "selu";

case GELU:

return "gelu";

case RELIE:

return "relie";

case RAMP:

return "ramp";

case LINEAR:

return "linear";

case TANH:

return "tanh";

case PLSE:

return "plse";

case LEAKY:

return "leaky";

case STAIR:

return "stair";

case HARDTAN:

return "hardtan";

case LHTAN:

return "lhtan";

default:

break;

}

return "relu";

}

ACTIVATION get\_activation(char \*s)

{

if (strcmp(s, "logistic")==0) return LOGISTIC;

if (strcmp(s, "swish") == 0) return SWISH;

if (strcmp(s, "mish") == 0) return MISH;

if (strcmp(s, "normalize\_channels") == 0) return NORM\_CHAN;

if (strcmp(s, "normalize\_channels\_softmax") == 0) return NORM\_CHAN\_SOFTMAX;

if (strcmp(s, "normalize\_channels\_softmax\_maxval") == 0) return NORM\_CHAN\_SOFTMAX\_MAXVAL;

if (strcmp(s, "loggy")==0) return LOGGY;

if (strcmp(s, "relu")==0) return RELU;

if (strcmp(s, "relu6") == 0) return RELU6;

if (strcmp(s, "elu")==0) return ELU;

if (strcmp(s, "selu") == 0) return SELU;

if (strcmp(s, "gelu") == 0) return GELU;

if (strcmp(s, "relie")==0) return RELIE;

if (strcmp(s, "plse")==0) return PLSE;

if (strcmp(s, "hardtan")==0) return HARDTAN;

if (strcmp(s, "lhtan")==0) return LHTAN;

if (strcmp(s, "linear")==0) return LINEAR;

if (strcmp(s, "ramp")==0) return RAMP;

if (strcmp(s, "leaky")==0) return LEAKY;

if (strcmp(s, "tanh")==0) return TANH;

if (strcmp(s, "stair")==0) return STAIR;

fprintf(stderr, "Couldn't find activation function %s, going with ReLU\n", s);

return RELU;

}

float activate(float x, ACTIVATION a)

{

switch(a){

case LINEAR:

return linear\_activate(x);

case LOGISTIC:

return logistic\_activate(x);

case LOGGY:

return loggy\_activate(x);

case RELU:

return relu\_activate(x);

case ELU:

return elu\_activate(x);

case SELU:

return selu\_activate(x);

case GELU:

return gelu\_activate(x);

case RELIE:

return relie\_activate(x);

case RAMP:

return ramp\_activate(x);

case LEAKY:

return leaky\_activate(x);

case TANH:

return tanh\_activate(x);

case PLSE:

return plse\_activate(x);

case STAIR:

return stair\_activate(x);

case HARDTAN:

return hardtan\_activate(x);

case LHTAN:

return lhtan\_activate(x);

}

return 0;

}

void activate\_array(float \*x, const int n, const ACTIVATION a)

{

int i;

if (a == LINEAR) {}

else if (a == LEAKY) {

#pragma omp parallel for

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

x[i] = leaky\_activate(x[i]);

}

}

else if (a == LOGISTIC) {

#pragma omp parallel for

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

x[i] = logistic\_activate(x[i]);

}

}

else {

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

x[i] = activate(x[i], a);

}

}

}

void activate\_array\_swish(float \*x, const int n, float \* output\_sigmoid, float \* output)

{

int i;

#pragma omp parallel for

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

float x\_val = x[i];

float sigmoid = logistic\_activate(x\_val);

output\_sigmoid[i] = sigmoid;

output[i] = x\_val \* sigmoid;

}

}

// https://github.com/digantamisra98/Mish

void activate\_array\_mish(float \*x, const int n, float \* activation\_input, float \* output)

{

const float MISH\_THRESHOLD = 20;

int i;

#pragma omp parallel for

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

float x\_val = x[i];

activation\_input[i] = x\_val; // store value before activation

output[i] = x\_val \* tanh\_activate( softplus\_activate(x\_val, MISH\_THRESHOLD) );

}

}

void activate\_array\_normalize\_channels(float \*x, const int n, int batch, int channels, int wh\_step, float \*output)

{

int size = n / channels;

int i;

#pragma omp parallel for

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

int wh\_i = i % wh\_step;

int b = i / wh\_step;

const float eps = 0.0001;

if (i < size) {

float sum = eps;

int k;

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

float val = x[wh\_i + k \* wh\_step + b\*wh\_step\*channels];

if (val > 0) sum += val;

}

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

float val = x[wh\_i + k \* wh\_step + b\*wh\_step\*channels];

if (val > 0) val = val / sum;

else val = 0;

output[wh\_i + k \* wh\_step + b\*wh\_step\*channels] = val;

}

}

}

}

void activate\_array\_normalize\_channels\_softmax(float \*x, const int n, int batch, int channels, int wh\_step, float \*output, int use\_max\_val)

{

int size = n / channels;

int i;

#pragma omp parallel for

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

int wh\_i = i % wh\_step;

int b = i / wh\_step;

const float eps = 0.0001;

if (i < size) {

float sum = eps;

float max\_val = -FLT\_MAX;

int k;

if (use\_max\_val) {

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

float val = x[wh\_i + k \* wh\_step + b\*wh\_step\*channels];

if (val > max\_val || k == 0) max\_val = val;

}

}

else

max\_val = 0;

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

float val = x[wh\_i + k \* wh\_step + b\*wh\_step\*channels];

sum += expf(val - max\_val);

}

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

float val = x[wh\_i + k \* wh\_step + b\*wh\_step\*channels];

val = expf(val - max\_val) / sum;

output[wh\_i + k \* wh\_step + b\*wh\_step\*channels] = val;

}

}

}

}

void gradient\_array\_normalize\_channels\_softmax(float \*x, const int n, int batch, int channels, int wh\_step, float \*delta)

{

int size = n / channels;

int i;

#pragma omp parallel for

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

int wh\_i = i % wh\_step;

int b = i / wh\_step;

if (i < size) {

float grad = 0;

int k;

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

const int index = wh\_i + k \* wh\_step + b\*wh\_step\*channels;

float out = x[index];

float d = delta[index];

grad += out\*d;

}

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

const int index = wh\_i + k \* wh\_step + b\*wh\_step\*channels;

float d = delta[index];

d = d \* grad;

delta[index] = d;

}

}

}

}

void gradient\_array\_normalize\_channels(float \*x, const int n, int batch, int channels, int wh\_step, float \*delta)

{

int size = n / channels;

int i;

#pragma omp parallel for

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

int wh\_i = i % wh\_step;

int b = i / wh\_step;

if (i < size) {

float grad = 0;

int k;

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

const int index = wh\_i + k \* wh\_step + b\*wh\_step\*channels;

float out = x[index];

float d = delta[index];

grad += out\*d;

}

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

const int index = wh\_i + k \* wh\_step + b\*wh\_step\*channels;

if (x[index] > 0) {

float d = delta[index];

d = d \* grad;

delta[index] = d;

}

}

}

}

}

float gradient(float x, ACTIVATION a)

{

switch(a){

case LINEAR:

return linear\_gradient(x);

case LOGISTIC:

return logistic\_gradient(x);

case LOGGY:

return loggy\_gradient(x);

case RELU:

return relu\_gradient(x);

case NORM\_CHAN:

//return relu\_gradient(x);

case NORM\_CHAN\_SOFTMAX\_MAXVAL:

//...

case NORM\_CHAN\_SOFTMAX:

printf(" Error: should be used custom NORM\_CHAN or NORM\_CHAN\_SOFTMAX-function for gradient \n");

exit(0);

return 0;

case ELU:

return elu\_gradient(x);

case SELU:

return selu\_gradient(x);

case GELU:

return gelu\_gradient(x);

case RELIE:

return relie\_gradient(x);

case RAMP:

return ramp\_gradient(x);

case LEAKY:

return leaky\_gradient(x);

case TANH:

return tanh\_gradient(x);

case PLSE:

return plse\_gradient(x);

case STAIR:

return stair\_gradient(x);

case HARDTAN:

return hardtan\_gradient(x);

case LHTAN:

return lhtan\_gradient(x);

}

return 0;

}

void gradient\_array(const float \*x, const int n, const ACTIVATION a, float \*delta)

{

int i;

#pragma omp parallel for

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

delta[i] \*= gradient(x[i], a);

}

}

// https://github.com/BVLC/caffe/blob/04ab089db018a292ae48d51732dd6c66766b36b6/src/caffe/layers/swish\_layer.cpp#L54-L56

void gradient\_array\_swish(const float \*x, const int n, const float \* sigmoid, float \* delta)

{

int i;

#pragma omp parallel for

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

float swish = x[i];

delta[i] \*= swish + sigmoid[i]\*(1 - swish);

}

}

// https://github.com/digantamisra98/Mish

void gradient\_array\_mish(const int n, const float \* activation\_input, float \* delta)

{

int i;

#pragma omp parallel for

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

const float MISH\_THRESHOLD = 20.0f;

// implementation from TensorFlow: https://github.com/tensorflow/addons/commit/093cdfa85d334cbe19a37624c33198f3140109ed

// implementation from Pytorch: https://github.com/thomasbrandon/mish-cuda/blob/master/csrc/mish.h#L26-L31

float inp = activation\_input[i];

const float sp = softplus\_activate(inp, MISH\_THRESHOLD);

const float grad\_sp = 1 - exp(-sp);

const float tsp = tanh(sp);

const float grad\_tsp = (1 - tsp\*tsp) \* grad\_sp;

const float grad = inp \* grad\_tsp + tsp;

delta[i] \*= grad;

//float x = activation\_input[i];

//float d = 2 \* expf(x) + expf(2 \* x) + 2;

//float w = 4 \* (x + 1) + 4 \* expf(2 \* x) + expf(3 \* x) + expf(x)\*(4 \* x + 6);

//float derivative = expf(x) \* w / (d \* d);

//delta[i] \*= derivative;

}

}