#include "convolutional\_layer.h"

#include "utils.h"

#include "batchnorm\_layer.h"

#include "im2col.h"

#include "col2im.h"

#include "blas.h"

#include "gemm.h"

#include "box.h"

#include <stdio.h>

#include <time.h>

#ifdef AI2

#include "xnor\_layer.h"

#endif

#ifdef \_\_cplusplus

#define PUT\_IN\_REGISTER

#else

#define PUT\_IN\_REGISTER register

#endif

#ifndef AI2

#define AI2 0

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

#endif

void swap\_binary(convolutional\_layer \*l)

{

float \*swap = l->weights;

l->weights = l->binary\_weights;

l->binary\_weights = swap;

#ifdef GPU

swap = l->weights\_gpu;

l->weights\_gpu = l->binary\_weights\_gpu;

l->binary\_weights\_gpu = swap;

#endif

}

void binarize\_weights(float \*weights, int n, int size, float \*binary)

{

int i, f;

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

float mean = 0;

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

mean += fabs(weights[f\*size + i]);

}

mean = mean / size;

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

binary[f\*size + i] = (weights[f\*size + i] > 0) ? mean: -mean;

}

}

}

void binarize\_cpu(float \*input, int n, float \*binary)

{

int i;

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

binary[i] = (input[i] > 0) ? 1 : -1;

}

}

void binarize\_input(float \*input, int n, int size, float \*binary)

{

int i, s;

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

float mean = 0;

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

mean += fabs(input[i\*size + s]);

}

mean = mean / n;

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

binary[i\*size + s] = (input[i\*size + s] > 0) ? mean : -mean;

}

}

}

int convolutional\_out\_height(convolutional\_layer l)

{

return (l.h + 2\*l.pad - l.size) / l.stride\_y + 1;

}

int convolutional\_out\_width(convolutional\_layer l)

{

return (l.w + 2\*l.pad - l.size) / l.stride\_x + 1;

}

image get\_convolutional\_image(convolutional\_layer l)

{

int h,w,c;

h = convolutional\_out\_height(l);

w = convolutional\_out\_width(l);

c = l.n;

return float\_to\_image(w,h,c,l.output);

}

image get\_convolutional\_delta(convolutional\_layer l)

{

int h,w,c;

h = convolutional\_out\_height(l);

w = convolutional\_out\_width(l);

c = l.n;

return float\_to\_image(w,h,c,l.delta);

}

size\_t get\_workspace\_size32(layer l){

#ifdef CUDNN

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 && l.train) most = s;

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

l.weightDesc,

l.ddstTensorDesc,

l.convDesc,

l.dsrcTensorDesc,

l.bd\_algo,

&s));

if (s > most && l.train) most = s;

return most;

}

#endif

if (l.xnor) {

size\_t re\_packed\_input\_size = l.c \* l.w \* l.h \* sizeof(float);

size\_t workspace\_size = (size\_t)l.bit\_align\*l.size\*l.size\*l.c \* sizeof(float);

if (workspace\_size < re\_packed\_input\_size) workspace\_size = re\_packed\_input\_size;

return workspace\_size;

}

return (size\_t)l.out\_h\*l.out\_w\*l.size\*l.size\*(l.c / l.groups)\*sizeof(float);

}

size\_t get\_workspace\_size16(layer l) {

#if defined(CUDNN) && defined(CUDNN\_HALF)

if (gpu\_index >= 0) {

size\_t most = 0;

size\_t s = 0;

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

l.srcTensorDesc16,

l.weightDesc16,

l.convDesc,

l.dstTensorDesc16,

l.fw\_algo16,

&s));

if (s > most) most = s;

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

l.srcTensorDesc16,

l.ddstTensorDesc16,

l.convDesc,

l.dweightDesc16,

l.bf\_algo16,

&s));

if (s > most && l.train) most = s;

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

l.weightDesc16,

l.ddstTensorDesc16,

l.convDesc,

l.dsrcTensorDesc16,

l.bd\_algo16,

&s));

if (s > most && l.train) most = s;

return most;

}

#endif

return 0;

//if (l.xnor) return (size\_t)l.bit\_align\*l.size\*l.size\*l.c \* sizeof(float);

//return (size\_t)l.out\_h\*l.out\_w\*l.size\*l.size\*l.c \* sizeof(float);

}

size\_t get\_convolutional\_workspace\_size(layer l) {

size\_t workspace\_size = get\_workspace\_size32(l);

size\_t workspace\_size16 = get\_workspace\_size16(l);

if (workspace\_size16 > workspace\_size) workspace\_size = workspace\_size16;

return workspace\_size;

}

#ifdef GPU

#ifdef CUDNN

void create\_convolutional\_cudnn\_tensors(layer \*l)

{

CHECK\_CUDNN(cudnnCreateTensorDescriptor(&l->normTensorDesc));

CHECK\_CUDNN(cudnnCreateTensorDescriptor(&l->normDstTensorDesc));

CHECK\_CUDNN(cudnnCreateTensorDescriptor(&l->srcTensorDesc));

CHECK\_CUDNN(cudnnCreateTensorDescriptor(&l->dstTensorDesc));

CHECK\_CUDNN(cudnnCreateFilterDescriptor(&l->weightDesc));

CHECK\_CUDNN(cudnnCreateTensorDescriptor(&l->dsrcTensorDesc));

CHECK\_CUDNN(cudnnCreateTensorDescriptor(&l->ddstTensorDesc));

CHECK\_CUDNN(cudnnCreateFilterDescriptor(&l->dweightDesc));

CHECK\_CUDNN(cudnnCreateTensorDescriptor(&l->normDstTensorDescF16));

CHECK\_CUDNN(cudnnCreateTensorDescriptor(&l->srcTensorDesc16));

CHECK\_CUDNN(cudnnCreateTensorDescriptor(&l->dstTensorDesc16));

CHECK\_CUDNN(cudnnCreateFilterDescriptor(&l->weightDesc16));

CHECK\_CUDNN(cudnnCreateTensorDescriptor(&l->dsrcTensorDesc16));

CHECK\_CUDNN(cudnnCreateTensorDescriptor(&l->ddstTensorDesc16));

CHECK\_CUDNN(cudnnCreateFilterDescriptor(&l->dweightDesc16));

CHECK\_CUDNN(cudnnCreateConvolutionDescriptor(&l->convDesc));

}

void cudnn\_convolutional\_setup(layer \*l, int cudnn\_preference, size\_t workspace\_size\_specify)

{

// CUDNN\_HALF

// TRUE\_HALF\_CONFIG is only supported on architectures with true fp16 support (compute capability 5.3 and 6.0):

// Tegra X1, Jetson TX1, DRIVE CX, DRIVE PX, Quadro GP100, Tesla P100

// PSEUDO\_HALF\_CONFIG is required for Tensor Cores - our case!

cudnnDataType\_t data\_type = CUDNN\_DATA\_FLOAT;

#if(CUDNN\_MAJOR >= 7)

// Tensor Core uses CUDNN\_TENSOR\_OP\_MATH instead of CUDNN\_DEFAULT\_MATH

// For \*\_ALGO\_WINOGRAD\_NONFUSED can be used CUDNN\_DATA\_FLOAT

// otherwise Input, Filter and Output descriptors (xDesc, yDesc, wDesc, dxDesc, dyDesc and dwDesc as applicable) have dataType = CUDNN\_DATA\_HALF

// Three techniques for training using Mixed-precision: https://devblogs.nvidia.com/mixed-precision-training-deep-neural-networks/

// 1. Accumulation into FP32

// 2. Loss Scaling - required only for: activation gradients. We do not use.

// 3. FP32 Master Copy of Weights

// More: http://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html#tensor\_ops

if (l->groups < 1) l->groups = 1;

if (l->stride\_x < 1) l->stride\_x = 1;

if (l->stride\_y < 1) l->stride\_y = 1;

CHECK\_CUDNN(cudnnSetConvolutionGroupCount(l->convDesc, l->groups));

CHECK\_CUDNN(cudnnSetConvolutionMathType(l->convDesc, CUDNN\_TENSOR\_OP\_MATH));

#if((CUDNN\_MAJOR\*10 + CUDNN\_MINOR) >= 72) // cuDNN >= 7.2

//CHECK\_CUDNN(cudnnSetConvolutionMathType(l->convDesc, CUDNN\_TENSOR\_OP\_MATH\_ALLOW\_CONVERSION)); // reduces the speed of regular and group convolution

#endif

#else //if(CUDNN\_MAJOR >= 7)

if (l->groups > 1) {

error("CUDNN < 7 doesn't support groups, please upgrade!");

}

#endif

// INT8\_CONFIG, INT8\_EXT\_CONFIG, INT8x4\_CONFIG and INT8x4\_EXT\_CONFIG are only supported

// on architectures with DP4A support (compute capability 6.1 and later).

//cudnnDataType\_t data\_type = CUDNN\_DATA\_INT8;

// backward delta

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(l->dsrcTensorDesc, CUDNN\_TENSOR\_NCHW, data\_type, l->batch, l->c, l->h, l->w));

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(l->ddstTensorDesc, CUDNN\_TENSOR\_NCHW, data\_type, l->batch, l->out\_c, l->out\_h, l->out\_w));

CHECK\_CUDNN(cudnnSetFilter4dDescriptor(l->dweightDesc, data\_type, CUDNN\_TENSOR\_NCHW, l->n, l->c / l->groups, l->size, l->size));

// forward

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(l->srcTensorDesc, CUDNN\_TENSOR\_NCHW, data\_type, l->batch, l->c, l->h, l->w));

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(l->dstTensorDesc, CUDNN\_TENSOR\_NCHW, data\_type, l->batch, l->out\_c, l->out\_h, l->out\_w));

CHECK\_CUDNN(cudnnSetFilter4dDescriptor(l->weightDesc, data\_type, CUDNN\_TENSOR\_NCHW, l->n, l->c / l->groups, l->size, l->size));

// backward delta

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(l->dsrcTensorDesc16, CUDNN\_TENSOR\_NCHW, CUDNN\_DATA\_HALF, l->batch, l->c, l->h, l->w));

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(l->ddstTensorDesc16, CUDNN\_TENSOR\_NCHW, CUDNN\_DATA\_HALF, l->batch, l->out\_c, l->out\_h, l->out\_w));

CHECK\_CUDNN(cudnnSetFilter4dDescriptor(l->dweightDesc16, CUDNN\_DATA\_HALF, CUDNN\_TENSOR\_NCHW, l->n, l->c / l->groups, l->size, l->size));

// forward

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(l->srcTensorDesc16, CUDNN\_TENSOR\_NCHW, CUDNN\_DATA\_HALF, l->batch, l->c, l->h, l->w));

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(l->dstTensorDesc16, CUDNN\_TENSOR\_NCHW, CUDNN\_DATA\_HALF, l->batch, l->out\_c, l->out\_h, l->out\_w));

CHECK\_CUDNN(cudnnSetFilter4dDescriptor(l->weightDesc16, CUDNN\_DATA\_HALF, CUDNN\_TENSOR\_NCHW, l->n, l->c / l->groups, l->size, l->size));

// batch norm

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(l->normDstTensorDescF16, CUDNN\_TENSOR\_NCHW, CUDNN\_DATA\_HALF, l->batch, l->out\_c, l->out\_h, l->out\_w));

// batch norm

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(l->normTensorDesc, CUDNN\_TENSOR\_NCHW, CUDNN\_DATA\_FLOAT, 1, l->out\_c, 1, 1));

CHECK\_CUDNN(cudnnSetTensor4dDescriptor(l->normDstTensorDesc, CUDNN\_TENSOR\_NCHW, CUDNN\_DATA\_FLOAT, l->batch, l->out\_c, l->out\_h, l->out\_w));

//printf("\n l->dilation = %d, l->pad = %d, l->size = %d, l->stride = %d, l->stride\_x = %d, l->stride\_y = %d, l->groups = %d, l->w = %d, l->h = %d, l->c = %d, l->n = %d, l->out\_w = %d, l->out\_h = %d, l->out\_c = %d, l->batch = %d, data\_type = %d \n",

// l->dilation, l->pad, l->size, l->stride, l->stride\_x, l->stride\_y, l->groups, l->w, l->h, l->c, l->n, l->out\_w, l->out\_h, l->out\_c, l->batch, data\_type);

#if(CUDNN\_MAJOR >= 6)

CHECK\_CUDNN(cudnnSetConvolution2dDescriptor(l->convDesc, l->pad \* l->dilation, l->pad \* l->dilation, l->stride\_y, l->stride\_x, l->dilation, l->dilation, CUDNN\_CROSS\_CORRELATION, CUDNN\_DATA\_FLOAT)); // cudnn >= 6.0

#else

CHECK\_CUDNN(cudnnSetConvolution2dDescriptor(l->convDesc, l->pad \* l->dilation, l->pad \* l->dilation, l->stride\_y, l->stride\_x, l->dilation, l->dilation, CUDNN\_CROSS\_CORRELATION)); // cudnn 5.1

#endif

int forward\_algo = CUDNN\_CONVOLUTION\_FWD\_PREFER\_FASTEST;

int backward\_algo = CUDNN\_CONVOLUTION\_BWD\_DATA\_PREFER\_FASTEST;

int backward\_filter = CUDNN\_CONVOLUTION\_BWD\_FILTER\_PREFER\_FASTEST;

if (cudnn\_preference == cudnn\_smallest)

{

forward\_algo = CUDNN\_CONVOLUTION\_FWD\_NO\_WORKSPACE;

backward\_algo = CUDNN\_CONVOLUTION\_BWD\_DATA\_NO\_WORKSPACE;

backward\_filter = CUDNN\_CONVOLUTION\_BWD\_FILTER\_NO\_WORKSPACE;

printf(" CUDNN-slow ");

}

if (cudnn\_preference == cudnn\_specify)

{

forward\_algo = CUDNN\_CONVOLUTION\_FWD\_SPECIFY\_WORKSPACE\_LIMIT;

backward\_algo = CUDNN\_CONVOLUTION\_BWD\_DATA\_SPECIFY\_WORKSPACE\_LIMIT;

backward\_filter = CUDNN\_CONVOLUTION\_BWD\_FILTER\_SPECIFY\_WORKSPACE\_LIMIT;

//printf(" CUDNN-specified %zu ", workspace\_size\_specify);

}

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

l->srcTensorDesc,

l->weightDesc,

l->convDesc,

l->dstTensorDesc,

(cudnnConvolutionFwdPreference\_t)forward\_algo,

workspace\_size\_specify,

&l->fw\_algo));

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

l->weightDesc,

l->ddstTensorDesc,

l->convDesc,

l->dsrcTensorDesc,

(cudnnConvolutionBwdDataPreference\_t)backward\_algo,

workspace\_size\_specify,

&l->bd\_algo));

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

l->srcTensorDesc,

l->ddstTensorDesc,

l->convDesc,

l->dweightDesc,

(cudnnConvolutionBwdFilterPreference\_t)backward\_filter,

workspace\_size\_specify,

&l->bf\_algo));

//if (data\_type == CUDNN\_DATA\_HALF)

{

// HALF-16 if(data\_type == CUDNN\_DATA\_HALF)

l->fw\_algo16 = CUDNN\_CONVOLUTION\_FWD\_ALGO\_IMPLICIT\_PRECOMP\_GEMM;

l->bd\_algo16 = CUDNN\_CONVOLUTION\_BWD\_DATA\_ALGO\_1;

l->bf\_algo16 = CUDNN\_CONVOLUTION\_BWD\_FILTER\_ALGO\_1;

// FLOAT-32 if(data\_type == CUDNN\_DATA\_FLOAT)

//l->fw\_algo16 = CUDNN\_CONVOLUTION\_FWD\_ALGO\_WINOGRAD\_NONFUSED;

//l->bd\_algo16 = CUDNN\_CONVOLUTION\_BWD\_DATA\_ALGO\_WINOGRAD\_NONFUSED;

//l->bf\_algo16 = CUDNN\_CONVOLUTION\_BWD\_FILTER\_ALGO\_WINOGRAD\_NONFUSED;

}

}

#endif

#endif

void free\_convolutional\_batchnorm(convolutional\_layer \*l)

{

if (!l->share\_layer) {

if (l->scales) free(l->scales), l->scales = NULL;

if (l->scale\_updates) free(l->scale\_updates), l->scale\_updates = NULL;

if (l->mean) free(l->mean), l->mean = NULL;

if (l->variance) free(l->variance), l->variance = NULL;

if (l->mean\_delta) free(l->mean\_delta), l->mean\_delta = NULL;

if (l->variance\_delta) free(l->variance\_delta), l->variance\_delta = NULL;

if (l->rolling\_mean) free(l->rolling\_mean), l->rolling\_mean = NULL;

if (l->rolling\_variance) free(l->rolling\_variance), l->rolling\_variance = NULL;

if (l->x) free(l->x), l->x = NULL;

if (l->x\_norm) free(l->x\_norm), l->x\_norm = NULL;

#ifdef GPU

if (l->scales\_gpu) cuda\_free(l->scales\_gpu), l->scales\_gpu = NULL;

if (l->scale\_updates\_gpu) cuda\_free(l->scale\_updates\_gpu), l->scale\_updates\_gpu = NULL;

if (l->mean\_gpu) cuda\_free(l->mean\_gpu), l->mean\_gpu = NULL;

if (l->variance\_gpu) cuda\_free(l->variance\_gpu), l->variance\_gpu = NULL;

if (l->mean\_delta\_gpu) cuda\_free(l->mean\_delta\_gpu), l->mean\_delta\_gpu = NULL;

if (l->variance\_delta\_gpu) cuda\_free(l->variance\_delta\_gpu), l->variance\_delta\_gpu = NULL;

if (l->rolling\_mean\_gpu) cuda\_free(l->rolling\_mean\_gpu), l->rolling\_mean\_gpu = NULL;

if (l->rolling\_variance\_gpu) cuda\_free(l->rolling\_variance\_gpu), l->rolling\_variance\_gpu = NULL;

if (l->x\_gpu) cuda\_free(l->x\_gpu), l->x\_gpu = NULL;

if (l->x\_norm\_gpu) cuda\_free(l->x\_norm\_gpu), l->x\_norm\_gpu = NULL;

#endif

}

}

convolutional\_layer make\_convolutional\_layer(int batch, int steps, int h, int w, int c, int n, int groups, int size, int stride\_x, int stride\_y, int dilation, int padding, ACTIVATION activation, int batch\_normalize, int binary, int xnor, int adam, int use\_bin\_output, int index, int antialiasing, convolutional\_layer \*share\_layer, int assisted\_excitation, int deform, int train)

{

int total\_batch = batch\*steps;

int i;

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

l.type = CONVOLUTIONAL;

l.train = train;

if (xnor) groups = 1; // disable groups for XNOR-net

if (groups < 1) groups = 1;

const int blur\_stride\_x = stride\_x;

const int blur\_stride\_y = stride\_y;

l.antialiasing = antialiasing;

if (antialiasing) {

stride\_x = stride\_y = l.stride = l.stride\_x = l.stride\_y = 1; // use stride=1 in host-layer

}

l.deform = deform;

l.assisted\_excitation = assisted\_excitation;

l.share\_layer = share\_layer;

l.index = index;

l.h = h;

l.w = w;

l.c = c;

l.groups = groups;

l.n = n;

l.binary = binary;

l.xnor = xnor;

l.use\_bin\_output = use\_bin\_output;

l.batch = batch;

l.steps = steps;

l.stride = stride\_x;

l.stride\_x = stride\_x;

l.stride\_y = stride\_y;

l.dilation = dilation;

l.size = size;

l.pad = padding;

l.batch\_normalize = batch\_normalize;

l.learning\_rate\_scale = 1;

l.nweights = (c / groups) \* n \* size \* size;

if (l.share\_layer) {

if (l.size != l.share\_layer->size || l.nweights != l.share\_layer->nweights || l.c != l.share\_layer->c || l.n != l.share\_layer->n) {

printf(" Layer size, nweights, channels or filters don't match for the share\_layer");

getchar();

}

l.weights = l.share\_layer->weights;

l.weight\_updates = l.share\_layer->weight\_updates;

l.biases = l.share\_layer->biases;

l.bias\_updates = l.share\_layer->bias\_updates;

}

else {

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

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

if (train) {

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

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

}

}

// float scale = 1./sqrt(size\*size\*c);

float scale = sqrt(2./(size\*size\*c/groups));

if (l.activation == NORM\_CHAN || l.activation == NORM\_CHAN\_SOFTMAX || l.activation == NORM\_CHAN\_SOFTMAX\_MAXVAL) {

for (i = 0; i < l.nweights; ++i) l.weights[i] = 1; // rand\_normal();

}

else {

for (i = 0; i < l.nweights; ++i) l.weights[i] = scale\*rand\_uniform(-1, 1); // rand\_normal();

}

int out\_h = convolutional\_out\_height(l);

int out\_w = convolutional\_out\_width(l);

l.out\_h = out\_h;

l.out\_w = out\_w;

l.out\_c = n;

l.outputs = l.out\_h \* l.out\_w \* l.out\_c;

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

l.activation = activation;

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

#ifndef GPU

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

#endif // not GPU

l.forward = forward\_convolutional\_layer;

l.backward = backward\_convolutional\_layer;

l.update = update\_convolutional\_layer;

if(binary){

l.binary\_weights = (float\*)xcalloc(l.nweights, sizeof(float));

l.cweights = (char\*)xcalloc(l.nweights, sizeof(char));

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

}

if(xnor){

l.binary\_weights = (float\*)xcalloc(l.nweights, sizeof(float));

l.binary\_input = (float\*)xcalloc(l.inputs \* l.batch, sizeof(float));

int align = 32;// 8;

int src\_align = l.out\_h\*l.out\_w;

l.bit\_align = src\_align + (align - src\_align % align);

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

const size\_t new\_c = l.c / 32;

size\_t in\_re\_packed\_input\_size = new\_c \* l.w \* l.h + 1;

l.bin\_re\_packed\_input = (uint32\_t\*)xcalloc(in\_re\_packed\_input\_size, sizeof(uint32\_t));

l.lda\_align = 256; // AVX2

int k = l.size\*l.size\*l.c;

size\_t k\_aligned = k + (l.lda\_align - k%l.lda\_align);

size\_t t\_bit\_input\_size = k\_aligned \* l.bit\_align / 8;

l.t\_bit\_input = (char\*)xcalloc(t\_bit\_input\_size, sizeof(char));

}

if(batch\_normalize){

if (l.share\_layer) {

l.scales = l.share\_layer->scales;

l.scale\_updates = l.share\_layer->scale\_updates;

l.mean = l.share\_layer->mean;

l.variance = l.share\_layer->variance;

l.mean\_delta = l.share\_layer->mean\_delta;

l.variance\_delta = l.share\_layer->variance\_delta;

l.rolling\_mean = l.share\_layer->rolling\_mean;

l.rolling\_variance = l.share\_layer->rolling\_variance;

}

else {

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

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

l.scales[i] = 1;

}

if (train) {

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

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

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

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

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

}

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

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

}

#ifndef GPU

if (train) {

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

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

}

#endif // not GPU

}

#ifndef GPU

if (l.activation == SWISH || l.activation == MISH) l.activation\_input = (float\*)calloc(total\_batch\*l.outputs, sizeof(float));

#endif // not GPU

if(adam){

l.adam = 1;

l.m = (float\*)xcalloc(l.nweights, sizeof(float));

l.v = (float\*)xcalloc(l.nweights, sizeof(float));

l.bias\_m = (float\*)xcalloc(n, sizeof(float));

l.scale\_m = (float\*)xcalloc(n, sizeof(float));

l.bias\_v = (float\*)xcalloc(n, sizeof(float));

l.scale\_v = (float\*)xcalloc(n, sizeof(float));

}

#ifdef GPU

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

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

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

if(gpu\_index >= 0){

if (l.activation == SWISH || l.activation == MISH) {

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

}

if (l.deform) l.weight\_deform\_gpu = cuda\_make\_array(NULL, l.nweights);

if (adam) {

l.m\_gpu = cuda\_make\_array(l.m, l.nweights);

l.v\_gpu = cuda\_make\_array(l.v, l.nweights);

l.bias\_m\_gpu = cuda\_make\_array(l.bias\_m, n);

l.bias\_v\_gpu = cuda\_make\_array(l.bias\_v, n);

l.scale\_m\_gpu = cuda\_make\_array(l.scale\_m, n);

l.scale\_v\_gpu = cuda\_make\_array(l.scale\_v, n);

}

if (l.share\_layer) {

l.weights\_gpu = l.share\_layer->weights\_gpu;

l.weight\_updates\_gpu = l.share\_layer->weight\_updates\_gpu;

l.weights\_gpu16 = l.share\_layer->weights\_gpu16;

l.weight\_updates\_gpu16 = l.share\_layer->weight\_updates\_gpu16;

l.biases\_gpu = l.share\_layer->biases\_gpu;

l.bias\_updates\_gpu = l.share\_layer->bias\_updates\_gpu;

}

else {

l.weights\_gpu = cuda\_make\_array(l.weights, l.nweights);

if (train) l.weight\_updates\_gpu = cuda\_make\_array(l.weight\_updates, l.nweights);

#ifdef CUDNN\_HALF

l.weights\_gpu16 = cuda\_make\_array(NULL, l.nweights / 2 + 1);

if (train) l.weight\_updates\_gpu16 = cuda\_make\_array(NULL, l.nweights / 2 + 1);

#endif // CUDNN\_HALF

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

if (train) l.bias\_updates\_gpu = cuda\_make\_array(l.bias\_updates, n);

}

l.output\_gpu = cuda\_make\_array(l.output, total\_batch\*out\_h\*out\_w\*n);

if (train) l.delta\_gpu = cuda\_make\_array(l.delta, total\_batch\*out\_h\*out\_w\*n);

if(binary){

l.binary\_weights\_gpu = cuda\_make\_array(l.weights, l.nweights);

}

if(xnor){

l.binary\_weights\_gpu = cuda\_make\_array(l.weights, l.nweights);

l.mean\_arr\_gpu = cuda\_make\_array(0, l.n);

l.binary\_input\_gpu = cuda\_make\_array(0, l.inputs\*l.batch);

}

if(batch\_normalize){

if (l.share\_layer) {

l.scales\_gpu = l.share\_layer->scales\_gpu;

l.scale\_updates\_gpu = l.share\_layer->scale\_updates\_gpu;

l.mean\_gpu = l.share\_layer->mean\_gpu;

l.variance\_gpu = l.share\_layer->variance\_gpu;

l.rolling\_mean\_gpu = l.share\_layer->rolling\_mean\_gpu;

l.rolling\_variance\_gpu = l.share\_layer->rolling\_variance\_gpu;

l.mean\_delta\_gpu = l.share\_layer->mean\_delta\_gpu;

l.variance\_delta\_gpu = l.share\_layer->variance\_delta\_gpu;

}

else {

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

if (train) {

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

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

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

l.m\_cbn\_avg\_gpu = cuda\_make\_array(l.mean, n);

l.v\_cbn\_avg\_gpu = cuda\_make\_array(l.variance, n);

#ifndef CUDNN

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

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

#endif // CUDNN

}

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

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

}

if (train) {

l.x\_gpu = cuda\_make\_array(l.output, total\_batch\*out\_h\*out\_w\*n);

#ifndef CUDNN

l.x\_norm\_gpu = cuda\_make\_array(l.output, total\_batch\*out\_h\*out\_w\*n);

#endif // CUDNN

}

}

if (l.assisted\_excitation)

{

const int size = l.out\_w \* l.out\_h \* l.batch;

l.gt\_gpu = cuda\_make\_array(NULL, size);

l.a\_avg\_gpu = cuda\_make\_array(NULL, size);

}

#ifdef CUDNN

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

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

#endif // CUDNN

}

#endif // GPU

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

//fprintf(stderr, "conv %5d %2d x%2d /%2d %4d x%4d x%4d -> %4d x%4d x%4d\n", n, size, size, stride, w, h, c, l.out\_w, l.out\_h, l.out\_c);

l.bflops = (2.0 \* l.nweights \* l.out\_h\*l.out\_w) / 1000000000.;

if (l.xnor) l.bflops = l.bflops / 32;

if (l.xnor && l.use\_bin\_output) fprintf(stderr, "convXB");

else if (l.xnor) fprintf(stderr, "convX ");

else if (l.share\_layer) fprintf(stderr, "convS ");

else if (l.assisted\_excitation) fprintf(stderr, "convAE");

else fprintf(stderr, "conv ");

if (groups > 1) fprintf(stderr, "%5d/%4d ", n, groups);

else fprintf(stderr, "%5d ", n);

if (stride\_x != stride\_y) fprintf(stderr, "%2dx%2d/%2dx%2d ", size, size, stride\_x, stride\_y);

else {

if (dilation > 1) fprintf(stderr, "%2d x%2d/%2d(%1d)", size, size, stride\_x, dilation);

else fprintf(stderr, "%2d x%2d/%2d ", size, size, stride\_x);

}

fprintf(stderr, "%4d x%4d x%4d -> %4d x%4d x%4d %5.3f BF\n", w, h, c, l.out\_w, l.out\_h, l.out\_c, l.bflops);

//fprintf(stderr, "%5d/%2d %2d x%2d /%2d(%d)%4d x%4d x%4d -> %4d x%4d x%4d %5.3f BF\n", n, groups, size, size, stride, dilation, w, h, c, l.out\_w, l.out\_h, l.out\_c, l.bflops);

if (l.antialiasing) {

printf("AA: ");

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

int blur\_size = 3;

int blur\_pad = blur\_size / 2;

if (l.antialiasing == 2) {

blur\_size = 2;

blur\_pad = 0;

}

\*(l.input\_layer) = make\_convolutional\_layer(batch, steps, out\_h, out\_w, n, n, n, blur\_size, blur\_stride\_x, blur\_stride\_y, 1, blur\_pad, LINEAR, 0, 0, 0, 0, 0, index, 0, NULL, 0, 0, train);

const int blur\_nweights = n \* blur\_size \* blur\_size; // (n / n) \* n \* blur\_size \* blur\_size;

int i;

if (blur\_size == 2) {

for (i = 0; i < blur\_nweights; i += (blur\_size\*blur\_size)) {

l.input\_layer->weights[i + 0] = 1 / 4.f;

l.input\_layer->weights[i + 1] = 1 / 4.f;

l.input\_layer->weights[i + 2] = 1 / 4.f;

l.input\_layer->weights[i + 3] = 1 / 4.f;

}

}

else {

for (i = 0; i < blur\_nweights; i += (blur\_size\*blur\_size)) {

l.input\_layer->weights[i + 0] = 1 / 16.f;

l.input\_layer->weights[i + 1] = 2 / 16.f;

l.input\_layer->weights[i + 2] = 1 / 16.f;

l.input\_layer->weights[i + 3] = 2 / 16.f;

l.input\_layer->weights[i + 4] = 4 / 16.f;

l.input\_layer->weights[i + 5] = 2 / 16.f;

l.input\_layer->weights[i + 6] = 1 / 16.f;

l.input\_layer->weights[i + 7] = 2 / 16.f;

l.input\_layer->weights[i + 8] = 1 / 16.f;

}

}

for (i = 0; i < n; ++i) l.input\_layer->biases[i] = 0;

#ifdef GPU

if (gpu\_index >= 0) {

l.input\_antialiasing\_gpu = cuda\_make\_array(NULL, l.batch\*l.outputs);

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

}

#endif // GPU

}

return l;

}

void denormalize\_convolutional\_layer(convolutional\_layer l)

{

int i, j;

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

float scale = l.scales[i]/sqrt(l.rolling\_variance[i] + .00001);

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

l.weights[i\*l.nweights + 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 test\_convolutional\_layer()

{

convolutional\_layer l = make\_convolutional\_layer(1, 1, 5, 5, 3, 2, 1, 5, 2, 2, 1, 1, LEAKY, 1, 0, 0, 0, 0, 0, 0, NULL, 0, 0, 0);

l.batch\_normalize = 1;

float data[] = {1,1,1,1,1,

1,1,1,1,1,

1,1,1,1,1,

1,1,1,1,1,

1,1,1,1,1,

2,2,2,2,2,

2,2,2,2,2,

2,2,2,2,2,

2,2,2,2,2,

2,2,2,2,2,

3,3,3,3,3,

3,3,3,3,3,

3,3,3,3,3,

3,3,3,3,3,

3,3,3,3,3};

network\_state state = {0};

state.input = data;

forward\_convolutional\_layer(l, state);

}

void resize\_convolutional\_layer(convolutional\_layer \*l, int w, int h)

{

int total\_batch = l->batch\*l->steps;

int old\_w = l->w;

int old\_h = l->h;

l->w = w;

l->h = h;

int out\_w = convolutional\_out\_width(\*l);

int out\_h = convolutional\_out\_height(\*l);

l->out\_w = out\_w;

l->out\_h = out\_h;

l->outputs = l->out\_h \* l->out\_w \* l->out\_c;

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

l->output = (float\*)xrealloc(l->output, total\_batch \* l->outputs \* sizeof(float));

if (l->train) {

l->delta = (float\*)xrealloc(l->delta, total\_batch \* l->outputs \* sizeof(float));

if (l->batch\_normalize) {

l->x = (float\*)xrealloc(l->x, total\_batch \* l->outputs \* sizeof(float));

l->x\_norm = (float\*)xrealloc(l->x\_norm, total\_batch \* l->outputs \* sizeof(float));

}

}

if (l->xnor) {

//l->binary\_input = realloc(l->inputs\*l->batch, sizeof(float));

}

if (l->activation == SWISH || l->activation == MISH) l->activation\_input = (float\*)realloc(l->activation\_input, total\_batch\*l->outputs \* sizeof(float));

#ifdef GPU

if (old\_w < w || old\_h < h || l->dynamic\_minibatch) {

if (l->train) {

cuda\_free(l->delta\_gpu);

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

}

cuda\_free(l->output\_gpu);

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

if (l->batch\_normalize) {

cuda\_free(l->x\_gpu);

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

#ifndef CUDNN

cuda\_free(l->x\_norm\_gpu);

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

#endif // CUDNN

}

if (l->xnor) {

cuda\_free(l->binary\_input\_gpu);

l->binary\_input\_gpu = cuda\_make\_array(0, l->inputs\*l->batch);

}

if (l->activation == SWISH || l->activation == MISH) {

cuda\_free(l->activation\_input\_gpu);

l->activation\_input\_gpu = cuda\_make\_array(l->activation\_input, total\_batch\*l->outputs);

}

if (l->assisted\_excitation)

{

cuda\_free(l->gt\_gpu);

cuda\_free(l->a\_avg\_gpu);

const int size = l->out\_w \* l->out\_h \* l->batch;

l->gt\_gpu = cuda\_make\_array(NULL, size);

l->a\_avg\_gpu = cuda\_make\_array(NULL, size);

}

}

#ifdef CUDNN

cudnn\_convolutional\_setup(l, cudnn\_fastest, 0);

#endif

#endif

l->workspace\_size = get\_convolutional\_workspace\_size(\*l);

#ifdef CUDNN

// check for excessive memory consumption

size\_t free\_byte;

size\_t total\_byte;

CHECK\_CUDA(cudaMemGetInfo(&free\_byte, &total\_byte));

if (l->workspace\_size > free\_byte || l->workspace\_size >= total\_byte / 2) {

printf(" used slow CUDNN algo without Workspace! Need memory: %zu, available: %zu\n", l->workspace\_size, (free\_byte < total\_byte/2) ? free\_byte : total\_byte/2);

cudnn\_convolutional\_setup(l, cudnn\_smallest, 0);

l->workspace\_size = get\_convolutional\_workspace\_size(\*l);

}

#endif

}

void set\_specified\_workspace\_limit(convolutional\_layer \*l, size\_t workspace\_size\_limit)

{

#ifdef CUDNN

size\_t free\_byte;

size\_t total\_byte;

CHECK\_CUDA(cudaMemGetInfo(&free\_byte, &total\_byte));

cudnn\_convolutional\_setup(l, cudnn\_specify, workspace\_size\_limit);

l->workspace\_size = get\_convolutional\_workspace\_size(\*l);

//printf("Set specified workspace limit for cuDNN: %zu, available: %zu, workspace = %zu \n", workspace\_size\_limit, free\_byte, l->workspace\_size);

#endif // CUDNN

}

void add\_bias(float \*output, float \*biases, int batch, int n, int size)

{

int i,j,b;

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

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

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

output[(b\*n + i)\*size + j] += biases[i];

}

}

}

}

void scale\_bias(float \*output, float \*scales, int batch, int n, int size)

{

int i,j,b;

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

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

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

output[(b\*n + i)\*size + j] \*= scales[i];

}

}

}

}

void backward\_bias(float \*bias\_updates, float \*delta, int batch, int n, int size)

{

int i,b;

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

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

bias\_updates[i] += sum\_array(delta+size\*(i+b\*n), size);

}

}

}

void gemm\_nn\_custom(int M, int N, int K, float ALPHA,

float \*A, int lda,

float \*B, int ldb,

float \*C, int ldc)

{

int i, j, k;

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

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

PUT\_IN\_REGISTER float A\_PART = ALPHA \* A[i \* lda + k];

//printf("\n weight = %f \n", A\_PART);

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

C[i\*ldc + j] += A\_PART\*B[k\*ldb + j];

}

}

}

}

void get\_mean\_array(float \*src, size\_t size, size\_t filters, float \*mean\_arr) {

size\_t i, counter;

counter = 0;

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

mean\_arr[counter++] = fabs(src[i]);

}

}

/\*

void float\_to\_bit(float \*src, unsigned char \*dst, size\_t size) {

size\_t dst\_size = size / 8 + 1;

memset(dst, 0, dst\_size);

size\_t i, dst\_i, dst\_shift;

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

if (src[i] > 0) set\_bit(dst, i);

}

}

\*/

void bit\_to\_float(unsigned char \*src, float \*dst, size\_t size, size\_t filters, float \*mean\_arr) {

memset(dst, 0, size \*sizeof(float));

size\_t i;

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

float mean\_val = 1;

if(mean\_arr != NULL) mean\_val = fabs(mean\_arr[i / (size / filters)]);

if(get\_bit(src, i)) dst[i] = mean\_val;

else dst[i] = -mean\_val;

}

}

void binary\_align\_weights(convolutional\_layer \*l)

{

int m = l->n; // (l->n / l->groups)

int k = l->size\*l->size\*l->c; // ->size\*l->size\*(l->c / l->groups)

size\_t new\_lda = k + (l->lda\_align - k % l->lda\_align); // (k / 8 + 1) \* 8;

l->new\_lda = new\_lda;

binarize\_weights(l->weights, m, k, l->binary\_weights);

size\_t align\_weights\_size = new\_lda \* m;

l->align\_bit\_weights\_size = align\_weights\_size / 8 + 1;

float\* align\_weights = (float\*)xcalloc(align\_weights\_size, sizeof(float));

l->align\_bit\_weights = (char\*)xcalloc(l->align\_bit\_weights\_size, sizeof(char));

size\_t i, j;

// align A without transpose

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

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

align\_weights[i\*new\_lda + j] = l->binary\_weights[i\*k + j];

}

}

if (l->c % 32 == 0)

//if(gpu\_index < 0 && l->stride == 1 && l->pad == 1 && l->c % 32 == 0)

//if (l->stride == 1 && l->pad == 1 && l->c % 32 == 0)

{

int fil, chan;

const int items\_per\_filter = l->c \* l->size \* l->size;

//const int dst\_items\_per\_filter = new\_lda;

for (fil = 0; fil < l->n; ++fil)

{

for (chan = 0; chan < l->c; chan += 32)

{

const int items\_per\_channel = l->size\*l->size;

for (i = 0; i < items\_per\_channel; ++i)

{

//uint32\_t val = 0;

int c\_pack;

for (c\_pack = 0; c\_pack < 32; ++c\_pack) {

float src = l->binary\_weights[fil\*items\_per\_filter + (chan + c\_pack)\*items\_per\_channel + i];

//align\_weights[fil\*items\_per\_filter + chan\*items\_per\_channel + i \* 32 + c\_pack] = src;

align\_weights[fil\*new\_lda + chan\*items\_per\_channel + i\*32 + c\_pack] = src;

//val |= (src << c);

}

}

}

}

//printf("\n l.index = %d \t aw[0] = %f, aw[1] = %f, aw[2] = %f, aw[3] = %f \n", l->index, align\_weights[0], align\_weights[1], align\_weights[2], align\_weights[3]);

//memcpy(l->binary\_weights, align\_weights, (l->size \* l->size \* l->c \* l->n) \* sizeof(float));

float\_to\_bit(align\_weights, (unsigned char\*)l->align\_bit\_weights, align\_weights\_size);

//if (l->n >= 32)

if(gpu\_index >= 0)

{

//int M = l->n;

//int N = l->out\_w\*l->out\_h;

//printf("\n M = %d, N = %d, M %% 8 = %d, N %% 8 = %d - weights \n", M, N, M % 8, N % 8);

//printf("\n l.w = %d, l.c = %d, l.n = %d \n", l->w, l->c, l->n);

for (i = 0; i < align\_weights\_size / 8; ++i) l->align\_bit\_weights[i] = ~(l->align\_bit\_weights[i]);

}

get\_mean\_array(l->binary\_weights, m\*k, l->n, l->mean\_arr);

//get\_mean\_array(l->binary\_weights, m\*new\_lda, l->n, l->mean\_arr);

}

else {

float\_to\_bit(align\_weights, (unsigned char\*)l->align\_bit\_weights, align\_weights\_size);

get\_mean\_array(l->binary\_weights, m\*k, l->n, l->mean\_arr);

}

//l->mean\_arr = calloc(l->n, sizeof(float));

//get\_mean\_array(align\_weights, align\_weights\_size, l->n, l->mean\_arr);

#ifdef GPU

cudaError\_t status;

l->align\_workspace\_size = l->bit\_align \* l->size \* l->size \* l->c;

status = cudaMalloc((void \*\*)&l->align\_workspace\_gpu, l->align\_workspace\_size \* sizeof(float));

status = cudaMalloc((void \*\*)&l->transposed\_align\_workspace\_gpu, l->align\_workspace\_size \* sizeof(float));

CHECK\_CUDA(status);

//l->align\_bit\_weights\_gpu = cuda\_make\_array(l->align\_bit\_weights, l->align\_bit\_weights\_size \* sizeof(char)/sizeof(float));

status = cudaMalloc((void \*\*)&l->align\_bit\_weights\_gpu, l->align\_bit\_weights\_size);

CHECK\_CUDA(status);

status = cudaMemcpy(l->align\_bit\_weights\_gpu, l->align\_bit\_weights, l->align\_bit\_weights\_size, cudaMemcpyHostToDevice);

CHECK\_CUDA(status);

status = cudaMemcpy(l->binary\_weights\_gpu, l->binary\_weights, m\*k \* sizeof(float), cudaMemcpyHostToDevice);

CHECK\_CUDA(status);

//l->mean\_arr\_gpu = cuda\_make\_array(l->mean\_arr, l->n);

cuda\_push\_array(l->mean\_arr\_gpu, l->mean\_arr, l->n);

CHECK\_CUDA(cudaDeviceSynchronize());

#endif // GPU

free(align\_weights);

}

// binary transpose

size\_t binary\_transpose\_align\_input(int k, int n, float \*b, char \*\*t\_bit\_input, size\_t ldb\_align, int bit\_align)

{

size\_t new\_ldb = k + (ldb\_align - k%ldb\_align); // (k / 8 + 1) \* 8;

//printf("\n n = %d, bit\_align = %d \n", n, bit\_align);

size\_t t\_intput\_size = new\_ldb \* bit\_align;// n;

size\_t t\_bit\_input\_size = t\_intput\_size / 8;// +1;

memset(\*t\_bit\_input, 0, t\_bit\_input\_size \* sizeof(char));

//int src\_size = k \* bit\_align;

// b - [bit\_align, k] - [l.bit\_align, l.size\*l.size\*l.c] = src\_size

// t\_input - [bit\_align, k] - [n', k]

// t\_bit\_input - [new\_ldb, n] - [k', n]

//transpose\_bin(t\_input, \*t\_bit\_input, k, n, bit\_align, new\_ldb, 8);

transpose\_bin((uint32\_t\*)b, (uint32\_t\*)\*t\_bit\_input, k, n, bit\_align, new\_ldb, 8);

return t\_intput\_size;

}

void forward\_convolutional\_layer(convolutional\_layer l, network\_state state)

{

int out\_h = convolutional\_out\_height(l);

int out\_w = convolutional\_out\_width(l);

int i, j;

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

if (l.xnor && (!l.align\_bit\_weights || state.train)) {

if (!l.align\_bit\_weights || state.train) {

binarize\_weights(l.weights, l.n, l.nweights, l.binary\_weights);

//printf("\n binarize\_weights l.align\_bit\_weights = %p \n", l.align\_bit\_weights);

}

swap\_binary(&l);

binarize\_cpu(state.input, l.c\*l.h\*l.w\*l.batch, l.binary\_input);

state.input = l.binary\_input;

}

int m = l.n / l.groups;

int k = l.size\*l.size\*l.c / l.groups;

int n = out\_h\*out\_w;

static int u = 0;

u++;

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

{

for (j = 0; j < l.groups; ++j)

{

float \*a = l.weights +j\*l.nweights / l.groups;

float \*b = state.workspace;

float \*c = l.output +(i\*l.groups + j)\*n\*m;

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

//gemm\_nn\_custom(m, n, k, 1, a, k, b, n, c, n);

if (l.xnor && l.align\_bit\_weights && !state.train && l.stride\_x == l.stride\_y)

{

memset(b, 0, l.bit\_align\*l.size\*l.size\*l.c \* sizeof(float));

if (l.c % 32 == 0)

{

//printf(" l.index = %d - new XNOR \n", l.index);

int ldb\_align = l.lda\_align;

size\_t new\_ldb = k + (ldb\_align - k%ldb\_align); // (k / 8 + 1) \* 8;

//size\_t t\_intput\_size = new\_ldb \* l.bit\_align;// n;

//size\_t t\_bit\_input\_size = t\_intput\_size / 8;// +1;

int re\_packed\_input\_size = l.c \* l.w \* l.h;

memset(state.workspace, 0, re\_packed\_input\_size \* sizeof(float));

const size\_t new\_c = l.c / 32;

size\_t in\_re\_packed\_input\_size = new\_c \* l.w \* l.h + 1;

memset(l.bin\_re\_packed\_input, 0, in\_re\_packed\_input\_size \* sizeof(uint32\_t));

//float \*re\_packed\_input = calloc(l.c \* l.w \* l.h, sizeof(float));

//uint32\_t \*bin\_re\_packed\_input = calloc(new\_c \* l.w \* l.h + 1, sizeof(uint32\_t));

// float32x4 by channel (as in cuDNN)

repack\_input(state.input, state.workspace, l.w, l.h, l.c);

// 32 x floats -> 1 x uint32\_t

float\_to\_bit(state.workspace, (unsigned char \*)l.bin\_re\_packed\_input, l.c \* l.w \* l.h);

//free(re\_packed\_input);

// slow - convolution the packed inputs and weights: float x 32 by channel (as in cuDNN)

//convolution\_repacked((uint32\_t \*)bin\_re\_packed\_input, (uint32\_t \*)l.align\_bit\_weights, l.output,

// l.w, l.h, l.c, l.n, l.size, l.pad, l.new\_lda, l.mean\_arr);

// // then exit from if()

im2col\_cpu\_custom((float \*)l.bin\_re\_packed\_input, new\_c, l.h, l.w, l.size, l.stride, l.pad, state.workspace);

//im2col\_cpu((float \*)bin\_re\_packed\_input, new\_c, l.h, l.w, l.size, l.stride, l.pad, b);

//free(bin\_re\_packed\_input);

int new\_k = l.size\*l.size\*l.c / 32;

// good for (l.c == 64)

//gemm\_nn\_bin\_32bit\_packed(m, n, new\_k, 1,

// l.align\_bit\_weights, l.new\_lda/32,

// b, n,

// c, n, l.mean\_arr);

// // then exit from if()

transpose\_uint32((uint32\_t \*)state.workspace, (uint32\_t\*)l.t\_bit\_input, new\_k, n, n, new\_ldb);

// the main GEMM function

gemm\_nn\_custom\_bin\_mean\_transposed(m, n, k, 1, (unsigned char\*)l.align\_bit\_weights, new\_ldb, (unsigned char\*)l.t\_bit\_input, new\_ldb, c, n, l.mean\_arr);

// // alternative GEMM

//gemm\_nn\_bin\_transposed\_32bit\_packed(m, n, new\_k, 1,

// l.align\_bit\_weights, l.new\_lda/32,

// t\_bit\_input, new\_ldb / 32,

// c, n, l.mean\_arr);

//free(t\_bit\_input);

}

else

{ // else (l.c % 32 != 0)

//--------------------------------------------------------

//printf(" l.index = %d - old XNOR \n", l.index);

//im2col\_cpu\_custom\_align(state.input, l.c, l.h, l.w, l.size, l.stride, l.pad, b, l.bit\_align);

im2col\_cpu\_custom\_bin(state.input, l.c, l.h, l.w, l.size, l.stride, l.pad, state.workspace, l.bit\_align);

//size\_t output\_size = l.outputs;

//float \*count\_output = calloc(output\_size, sizeof(float));

//size\_t bit\_output\_size = output\_size / 8 + 1;

//char \*bit\_output = calloc(bit\_output\_size, sizeof(char));

//size\_t intput\_size = n \* k; // (out\_h\*out\_w) X (l.size\*l.size\*l.c) : after im2col()

//size\_t bit\_input\_size = intput\_size / 8 + 1;

//char \*bit\_input = calloc(bit\_input\_size, sizeof(char));

//size\_t weights\_size = k \* m; //l.size\*l.size\*l.c\*l.n; // l.nweights

//size\_t bit\_weights\_size = weights\_size / 8 + 1;

//char \*bit\_weights = calloc(bit\_weights\_size, sizeof(char));

//float \*mean\_arr = calloc(l.n, sizeof(float));

// transpose B from NxK to KxN (x-axis (ldb = l.size\*l.size\*l.c) - should be multiple of 8 bits)

{

//size\_t ldb\_align = 256; // 256 bit for AVX2

int ldb\_align = l.lda\_align;

size\_t new\_ldb = k + (ldb\_align - k%ldb\_align);

size\_t t\_intput\_size = binary\_transpose\_align\_input(k, n, state.workspace, &l.t\_bit\_input, ldb\_align, l.bit\_align);

// 5x times faster than gemm()-float32

gemm\_nn\_custom\_bin\_mean\_transposed(m, n, k, 1, (unsigned char\*)l.align\_bit\_weights, new\_ldb, (unsigned char\*)l.t\_bit\_input, new\_ldb, c, n, l.mean\_arr);

//gemm\_nn\_custom\_bin\_mean\_transposed(m, n, k, 1, bit\_weights, k, t\_bit\_input, new\_ldb, c, n, mean\_arr);

//free(t\_input);

//free(t\_bit\_input);

//}

}

}

add\_bias(l.output, l.biases, l.batch, l.n, out\_h\*out\_w);

//activate\_array(l.output, m\*n\*l.batch, l.activation);

if (l.activation == SWISH) activate\_array\_swish(l.output, l.outputs\*l.batch, l.activation\_input, l.output);

else if (l.activation == MISH) activate\_array\_mish(l.output, l.outputs\*l.batch, l.activation\_input, l.output);

else if (l.activation == NORM\_CHAN) activate\_array\_normalize\_channels(l.output, l.outputs\*l.batch, l.batch, l.out\_c, l.out\_w\*l.out\_h, l.output);

else if (l.activation == NORM\_CHAN\_SOFTMAX) activate\_array\_normalize\_channels\_softmax(l.output, l.outputs\*l.batch, l.batch, l.out\_c, l.out\_w\*l.out\_h, l.output, 0);

else if (l.activation == NORM\_CHAN\_SOFTMAX\_MAXVAL) activate\_array\_normalize\_channels\_softmax(l.output, l.outputs\*l.batch, l.batch, l.out\_c, l.out\_w\*l.out\_h, l.output, 1);

else activate\_array\_cpu\_custom(l.output, m\*n\*l.batch, l.activation);

return;

}

else {

//printf(" l.index = %d - FP32 \n", l.index);

float \*im = state.input + (i\*l.groups + j)\*(l.c / l.groups)\*l.h\*l.w;

if (l.size == 1) {

b = im;

}

else {

//im2col\_cpu(im, l.c / l.groups, l.h, l.w, l.size, l.stride, l.pad, b);

im2col\_cpu\_ext(im, // input

l.c / l.groups, // input channels

l.h, l.w, // input size (h, w)

l.size, l.size, // kernel size (h, w)

l.pad \* l.dilation, l.pad \* l.dilation, // padding (h, w)

l.stride\_y, l.stride\_x, // stride (h, w)

l.dilation, l.dilation, // dilation (h, w)

b); // output

}

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

// bit-count to float

}

//c += n\*m;

//state.input += l.c\*l.h\*l.w;

}

}

if(l.batch\_normalize){

forward\_batchnorm\_layer(l, state);

}

else {

add\_bias(l.output, l.biases, l.batch, l.n, out\_h\*out\_w);

}

//activate\_array(l.output, m\*n\*l.batch, l.activation);

if (l.activation == SWISH) activate\_array\_swish(l.output, l.outputs\*l.batch, l.activation\_input, l.output);

else if (l.activation == MISH) activate\_array\_mish(l.output, l.outputs\*l.batch, l.activation\_input, l.output);

else if (l.activation == NORM\_CHAN) activate\_array\_normalize\_channels(l.output, l.outputs\*l.batch, l.batch, l.out\_c, l.out\_w\*l.out\_h, l.output);

else if (l.activation == NORM\_CHAN\_SOFTMAX) activate\_array\_normalize\_channels\_softmax(l.output, l.outputs\*l.batch, l.batch, l.out\_c, l.out\_w\*l.out\_h, l.output, 0);

else if (l.activation == NORM\_CHAN\_SOFTMAX\_MAXVAL) activate\_array\_normalize\_channels\_softmax(l.output, l.outputs\*l.batch, l.batch, l.out\_c, l.out\_w\*l.out\_h, l.output, 1);

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

if(l.binary || l.xnor) swap\_binary(&l);

//visualize\_convolutional\_layer(l, "conv\_visual", NULL);

//wait\_until\_press\_key\_cv();

if(l.assisted\_excitation && state.train) assisted\_excitation\_forward(l, state);

if (l.antialiasing) {

network\_state s = { 0 };

s.train = state.train;

s.workspace = state.workspace;

s.net = state.net;

s.input = l.output;

forward\_convolutional\_layer(\*(l.input\_layer), s);

//simple\_copy\_ongpu(l.outputs\*l.batch, l.output, l.input\_antialiasing);

memcpy(l.output, l.input\_layer->output, l.input\_layer->outputs \* l.input\_layer->batch \* sizeof(float));

}

}

void assisted\_excitation\_forward(convolutional\_layer l, network\_state state)

{

const int iteration\_num = (\*state.net.seen) / (state.net.batch\*state.net.subdivisions);

// epoch

//const float epoch = (float)(\*state.net.seen) / state.net.train\_images\_num;

// calculate alpha

//const float alpha = (1 + cos(3.141592 \* iteration\_num)) / (2 \* state.net.max\_batches);

//const float alpha = (1 + cos(3.141592 \* epoch)) / (2 \* state.net.max\_batches);

float alpha = (1 + cos(3.141592 \* iteration\_num / state.net.max\_batches));

if (l.assisted\_excitation > 1) {

if (iteration\_num > l.assisted\_excitation) alpha = 0;

else alpha = (1 + cos(3.141592 \* iteration\_num / l.assisted\_excitation));

}

//printf("\n epoch = %f, alpha = %f, seen = %d, max\_batches = %d, train\_images\_num = %d \n",

// epoch, alpha, (\*state.net.seen), state.net.max\_batches, state.net.train\_images\_num);

float \*a\_avg = (float \*)xcalloc(l.out\_w \* l.out\_h \* l.batch, sizeof(float));

float \*g = (float \*)xcalloc(l.out\_w \* l.out\_h \* l.batch, sizeof(float));

int b;

int w, h, c;

l.max\_boxes = state.net.num\_boxes;

l.truths = l.max\_boxes\*(4 + 1);

for (b = 0; b < l.batch; ++b)

{

// calculate G

int t;

for (t = 0; t < state.net.num\_boxes; ++t) {

box truth = float\_to\_box\_stride(state.truth + t\*(4 + 1) + b\*l.truths, 1);

if (!truth.x) break; // continue;

int left = floor((truth.x - truth.w / 2) \* l.out\_w);

int right = ceil((truth.x + truth.w / 2) \* l.out\_w);

int top = floor((truth.y - truth.h / 2) \* l.out\_h);

int bottom = ceil((truth.y + truth.h / 2) \* l.out\_h);

for (w = left; w <= right; w++) {

for (h = top; h < bottom; h++) {

g[w + l.out\_w \* h + l.out\_w\*l.out\_h\*b] = 1;

}

}

}

}

for (b = 0; b < l.batch; ++b)

{

// calculate average A

for (w = 0; w < l.out\_w; w++) {

for (h = 0; h < l.out\_h; h++) {

for (c = 0; c < l.out\_c; c++) {

a\_avg[w + l.out\_w\*(h + l.out\_h\*b)] += l.output[w + l.out\_w\*(h + l.out\_h\*(c + l.out\_c\*b))];

}

a\_avg[w + l.out\_w\*(h + l.out\_h\*b)] /= l.out\_c; // a\_avg / d

}

}

}

// change activation

for (b = 0; b < l.batch; ++b)

{

for (w = 0; w < l.out\_w; w++) {

for (h = 0; h < l.out\_h; h++) {

for (c = 0; c < l.out\_c; c++)

{

// a = a + alpha(t) + e(c,i,j) = a + alpha(t) + g(i,j) \* avg\_a(i,j) / channels

l.output[w + l.out\_w\*(h + l.out\_h\*(c + l.out\_c\*b))] +=

alpha \*

g[w + l.out\_w\*(h + l.out\_h\*b)] \*

a\_avg[w + l.out\_w\*(h + l.out\_h\*b)];

//l.output[w + l.out\_w\*(h + l.out\_h\*(c + l.out\_c\*b))] =

// alpha \* g[w + l.out\_w\*(h + l.out\_h\*b)] \* a\_avg[w + l.out\_w\*(h + l.out\_h\*b)];

}

}

}

}

if(0) // visualize ground truth

{

#ifdef OPENCV

for (b = 0; b < l.batch; ++b)

{

image img = float\_to\_image(l.out\_w, l.out\_h, 1, &g[l.out\_w\*l.out\_h\*b]);

char buff[100];

sprintf(buff, "a\_excitation\_%d", b);

show\_image\_cv(img, buff);

image img2 = float\_to\_image(l.out\_w, l.out\_h, 1, &l.output[l.out\_w\*l.out\_h\*l.out\_c\*b]);

char buff2[100];

sprintf(buff2, "a\_excitation\_act\_%d", b);

show\_image\_cv(img2, buff2);

wait\_key\_cv(5);

}

wait\_until\_press\_key\_cv();

#endif // OPENCV

}

free(g);

free(a\_avg);

}

void backward\_convolutional\_layer(convolutional\_layer l, network\_state state)

{

int i, j;

int m = l.n / l.groups;

int n = l.size\*l.size\*l.c / l.groups;

int k = l.out\_w\*l.out\_h;

if (l.activation == SWISH) gradient\_array\_swish(l.output, l.outputs\*l.batch, l.activation\_input, l.delta);

else if (l.activation == MISH) gradient\_array\_mish(l.outputs\*l.batch, l.activation\_input, l.delta);

else if (l.activation == NORM\_CHAN\_SOFTMAX || l.activation == NORM\_CHAN\_SOFTMAX\_MAXVAL) gradient\_array\_normalize\_channels\_softmax(l.output, l.outputs\*l.batch, l.batch, l.out\_c, l.out\_w\*l.out\_h, l.delta);

else if (l.activation == NORM\_CHAN) gradient\_array\_normalize\_channels(l.output, l.outputs\*l.batch, l.batch, l.out\_c, l.out\_w\*l.out\_h, l.delta);

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

if (l.batch\_normalize) {

backward\_batchnorm\_layer(l, state);

}

else {

backward\_bias(l.bias\_updates, l.delta, l.batch, l.n, k);

}

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

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

float \*a = l.delta + (i\*l.groups + j)\*m\*k;

float \*b = state.workspace;

float \*c = l.weight\_updates + j\*l.nweights / l.groups;

float \*im = state.input + (i\*l.groups + j)\* (l.c / l.groups)\*l.h\*l.w;

//im2col\_cpu(im, l.c / l.groups, l.h, l.w, l.size, l.stride, l.pad, b);

im2col\_cpu\_ext(

im, // input

l.c / l.groups, // input channels

l.h, l.w, // input size (h, w)

l.size, l.size, // kernel size (h, w)

l.pad \* l.dilation, l.pad \* l.dilation, // padding (h, w)

l.stride\_y, l.stride\_x, // stride (h, w)

l.dilation, l.dilation, // dilation (h, w)

b); // output

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

if (state.delta) {

a = l.weights + j\*l.nweights / l.groups;

b = l.delta + (i\*l.groups + j)\*m\*k;

c = state.workspace;

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

//col2im\_cpu(state.workspace, l.c / l.groups, l.h, l.w, l.size, l.stride,

// l.pad, state.delta + (i\*l.groups + j)\*l.c / l.groups\*l.h\*l.w);

col2im\_cpu\_ext(

state.workspace, // input

l.c / l.groups, // input channels (h, w)

l.h, l.w, // input size (h, w)

l.size, l.size, // kernel size (h, w)

l.pad \* l.dilation, l.pad \* l.dilation, // padding (h, w)

l.stride\_y, l.stride\_x, // stride (h, w)

l.dilation, l.dilation, // dilation (h, w)

state.delta + (i\*l.groups + j)\* (l.c / l.groups)\*l.h\*l.w); // output (delta)

}

}

}

}

void update\_convolutional\_layer(convolutional\_layer l, int batch, float learning\_rate\_init, float momentum, float decay)

{

float learning\_rate = learning\_rate\_init\*l.learning\_rate\_scale;

//float momentum = a.momentum;

//float decay = a.decay;

//int batch = a.batch;

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

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

scal\_cpu(l.nweights, momentum, l.weight\_updates, 1);

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

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

if (l.scales) {

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

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

}

}

image get\_convolutional\_weight(convolutional\_layer l, int i)

{

int h = l.size;

int w = l.size;

int c = l.c / l.groups;

return float\_to\_image(w, h, c, l.weights + i\*h\*w\*c);

}

void rgbgr\_weights(convolutional\_layer l)

{

int i;

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

image im = get\_convolutional\_weight(l, i);

if (im.c == 3) {

rgbgr\_image(im);

}

}

}

void rescale\_weights(convolutional\_layer l, float scale, float trans)

{

int i;

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

image im = get\_convolutional\_weight(l, i);

if (im.c == 3) {

scale\_image(im, scale);

float sum = sum\_array(im.data, im.w\*im.h\*im.c);

l.biases[i] += sum\*trans;

}

}

}

image \*get\_weights(convolutional\_layer l)

{

image \*weights = (image \*)xcalloc(l.n, sizeof(image));

int i;

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

weights[i] = copy\_image(get\_convolutional\_weight(l, i));

normalize\_image(weights[i]);

/\*

char buff[256];

sprintf(buff, "filter%d", i);

save\_image(weights[i], buff);

\*/

}

//error("hey");

return weights;

}

image \*visualize\_convolutional\_layer(convolutional\_layer l, char \*window, image \*prev\_weights)

{

image \*single\_weights = get\_weights(l);

show\_images(single\_weights, l.n, window);

image delta = get\_convolutional\_image(l);

image dc = collapse\_image\_layers(delta, 1);

char buff[256];

sprintf(buff, "%s: Output", window);

show\_image(dc, buff);

//save\_image(dc, buff);

free\_image(dc);

return single\_weights;

}