#include "darknet.h"

#include <stdio.h>

#include <time.h>

#include <assert.h>

#include "network.h"

#include "image.h"

#include "data.h"

#include "utils.h"

#include "blas.h"

#include "crop\_layer.h"

#include "connected\_layer.h"

#include "gru\_layer.h"

#include "rnn\_layer.h"

#include "crnn\_layer.h"

#include "conv\_lstm\_layer.h"

#include "local\_layer.h"

#include "convolutional\_layer.h"

#include "activation\_layer.h"

#include "detection\_layer.h"

#include "region\_layer.h"

#include "normalization\_layer.h"

#include "batchnorm\_layer.h"

#include "maxpool\_layer.h"

#include "reorg\_layer.h"

#include "reorg\_old\_layer.h"

#include "avgpool\_layer.h"

#include "cost\_layer.h"

#include "softmax\_layer.h"

#include "dropout\_layer.h"

#include "route\_layer.h"

#include "shortcut\_layer.h"

#include "scale\_channels\_layer.h"

#include "sam\_layer.h"

#include "yolo\_layer.h"

#include "gaussian\_yolo\_layer.h"

#include "upsample\_layer.h"

#include "parser.h"

load\_args get\_base\_args(network \*net)

{

load\_args args = { 0 };

args.w = net->w;

args.h = net->h;

args.size = net->w;

args.min = net->min\_crop;

args.max = net->max\_crop;

args.angle = net->angle;

args.aspect = net->aspect;

args.exposure = net->exposure;

args.center = net->center;

args.saturation = net->saturation;

args.hue = net->hue;

return args;

}

int64\_t get\_current\_iteration(network net)

{

return \*net.cur\_iteration;

}

int get\_current\_batch(network net)

{

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

return batch\_num;

}

/\*

void reset\_momentum(network net)

{

if (net.momentum == 0) return;

net.learning\_rate = 0;

net.momentum = 0;

net.decay = 0;

#ifdef GPU

//if(net.gpu\_index >= 0) update\_network\_gpu(net);

#endif

}

\*/

void reset\_network\_state(network \*net, int b)

{

int i;

for (i = 0; i < net->n; ++i) {

#ifdef GPU

layer l = net->layers[i];

if (l.state\_gpu) {

fill\_ongpu(l.outputs, 0, l.state\_gpu + l.outputs\*b, 1);

}

if (l.h\_gpu) {

fill\_ongpu(l.outputs, 0, l.h\_gpu + l.outputs\*b, 1);

}

#endif

}

}

void reset\_rnn(network \*net)

{

reset\_network\_state(net, 0);

}

float get\_current\_seq\_subdivisions(network net)

{

int sequence\_subdivisions = net.init\_sequential\_subdivisions;

if (net.num\_steps > 0)

{

int batch\_num = get\_current\_batch(net);

int i;

for (i = 0; i < net.num\_steps; ++i) {

if (net.steps[i] > batch\_num) break;

sequence\_subdivisions \*= net.seq\_scales[i];

}

}

if (sequence\_subdivisions < 1) sequence\_subdivisions = 1;

if (sequence\_subdivisions > net.subdivisions) sequence\_subdivisions = net.subdivisions;

return sequence\_subdivisions;

}

int get\_sequence\_value(network net)

{

int sequence = 1;

if (net.sequential\_subdivisions != 0) sequence = net.subdivisions / net.sequential\_subdivisions;

if (sequence < 1) sequence = 1;

return sequence;

}

float get\_current\_rate(network net)

{

int batch\_num = get\_current\_batch(net);

int i;

float rate;

if (batch\_num < net.burn\_in) return net.learning\_rate \* pow((float)batch\_num / net.burn\_in, net.power);

switch (net.policy) {

case CONSTANT:

return net.learning\_rate;

case STEP:

return net.learning\_rate \* pow(net.scale, batch\_num/net.step);

case STEPS:

rate = net.learning\_rate;

for(i = 0; i < net.num\_steps; ++i){

if(net.steps[i] > batch\_num) return rate;

rate \*= net.scales[i];

//if(net.steps[i] > batch\_num - 1 && net.scales[i] > 1) reset\_momentum(net);

}

return rate;

case EXP:

return net.learning\_rate \* pow(net.gamma, batch\_num);

case POLY:

return net.learning\_rate \* pow(1 - (float)batch\_num / net.max\_batches, net.power);

//if (batch\_num < net.burn\_in) return net.learning\_rate \* pow((float)batch\_num / net.burn\_in, net.power);

//return net.learning\_rate \* pow(1 - (float)batch\_num / net.max\_batches, net.power);

case RANDOM:

return net.learning\_rate \* pow(rand\_uniform(0,1), net.power);

case SIG:

return net.learning\_rate \* (1./(1.+exp(net.gamma\*(batch\_num - net.step))));

case SGDR:

{

int last\_iteration\_start = 0;

int cycle\_size = net.batches\_per\_cycle;

while ((last\_iteration\_start + cycle\_size) < batch\_num)

{

last\_iteration\_start += cycle\_size;

cycle\_size \*= net.batches\_cycle\_mult;

}

rate = net.learning\_rate\_min +

0.5\*(net.learning\_rate - net.learning\_rate\_min)

\* (1. + cos((float)(batch\_num - last\_iteration\_start)\*3.14159265 / cycle\_size));

return rate;

}

default:

fprintf(stderr, "Policy is weird!\n");

return net.learning\_rate;

}

}

char \*get\_layer\_string(LAYER\_TYPE a)

{

switch(a){

case CONVOLUTIONAL:

return "convolutional";

case ACTIVE:

return "activation";

case LOCAL:

return "local";

case DECONVOLUTIONAL:

return "deconvolutional";

case CONNECTED:

return "connected";

case RNN:

return "rnn";

case GRU:

return "gru";

case LSTM:

return "lstm";

case CRNN:

return "crnn";

case MAXPOOL:

return "maxpool";

case REORG:

return "reorg";

case AVGPOOL:

return "avgpool";

case SOFTMAX:

return "softmax";

case DETECTION:

return "detection";

case REGION:

return "region";

case YOLO:

return "yolo";

case GAUSSIAN\_YOLO:

return "Gaussian\_yolo";

case DROPOUT:

return "dropout";

case CROP:

return "crop";

case COST:

return "cost";

case ROUTE:

return "route";

case SHORTCUT:

return "shortcut";

case SCALE\_CHANNELS:

return "scale\_channels";

case SAM:

return "sam";

case NORMALIZATION:

return "normalization";

case BATCHNORM:

return "batchnorm";

default:

break;

}

return "none";

}

network make\_network(int n)

{

network net = {0};

net.n = n;

net.layers = (layer\*)xcalloc(net.n, sizeof(layer));

net.seen = (uint64\_t\*)xcalloc(1, sizeof(uint64\_t));

net.cur\_iteration = (int\*)xcalloc(1, sizeof(int));

#ifdef GPU

net.input\_gpu = (float\*\*)xcalloc(1, sizeof(float\*));

net.truth\_gpu = (float\*\*)xcalloc(1, sizeof(float\*));

net.input16\_gpu = (float\*\*)xcalloc(1, sizeof(float\*));

net.output16\_gpu = (float\*\*)xcalloc(1, sizeof(float\*));

net.max\_input16\_size = (size\_t\*)xcalloc(1, sizeof(size\_t));

net.max\_output16\_size = (size\_t\*)xcalloc(1, sizeof(size\_t));

#endif

return net;

}

void forward\_network(network net, network\_state state)

{

state.workspace = net.workspace;

int i;

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

state.index = i;

layer l = net.layers[i];

if(l.delta && state.train){

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

}

//double time = get\_time\_point();

l.forward(l, state);

//printf("%d - Predicted in %lf milli-seconds.\n", i, ((double)get\_time\_point() - time) / 1000);

state.input = l.output;

/\*

float avg\_val = 0;

int k;

for (k = 0; k < l.outputs; ++k) avg\_val += l.output[k];

printf(" i: %d - avg\_val = %f \n", i, avg\_val / l.outputs);

\*/

}

}

void update\_network(network net)

{

int i;

int update\_batch = net.batch\*net.subdivisions;

float rate = get\_current\_rate(net);

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

layer l = net.layers[i];

if(l.update){

l.update(l, update\_batch, rate, net.momentum, net.decay);

}

}

}

float \*get\_network\_output(network net)

{

#ifdef GPU

if (gpu\_index >= 0) return get\_network\_output\_gpu(net);

#endif

int i;

for(i = net.n-1; i > 0; --i) if(net.layers[i].type != COST) break;

return net.layers[i].output;

}

float get\_network\_cost(network net)

{

int i;

float sum = 0;

int count = 0;

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

if(net.layers[i].cost){

sum += net.layers[i].cost[0];

++count;

}

}

return sum/count;

}

int get\_predicted\_class\_network(network net)

{

float \*out = get\_network\_output(net);

int k = get\_network\_output\_size(net);

return max\_index(out, k);

}

void backward\_network(network net, network\_state state)

{

int i;

float \*original\_input = state.input;

float \*original\_delta = state.delta;

state.workspace = net.workspace;

for(i = net.n-1; i >= 0; --i){

state.index = i;

if(i == 0){

state.input = original\_input;

state.delta = original\_delta;

}else{

layer prev = net.layers[i-1];

state.input = prev.output;

state.delta = prev.delta;

}

layer l = net.layers[i];

if (l.stopbackward) break;

if (l.onlyforward) continue;

l.backward(l, state);

}

}

float train\_network\_datum(network net, float \*x, float \*y)

{

#ifdef GPU

if(gpu\_index >= 0) return train\_network\_datum\_gpu(net, x, y);

#endif

network\_state state={0};

\*net.seen += net.batch;

state.index = 0;

state.net = net;

state.input = x;

state.delta = 0;

state.truth = y;

state.train = 1;

forward\_network(net, state);

backward\_network(net, state);

float error = get\_network\_cost(net);

//if(((\*net.seen)/net.batch)%net.subdivisions == 0) update\_network(net);

return error;

}

float train\_network\_sgd(network net, data d, int n)

{

int batch = net.batch;

float\* X = (float\*)xcalloc(batch \* d.X.cols, sizeof(float));

float\* y = (float\*)xcalloc(batch \* d.y.cols, sizeof(float));

int i;

float sum = 0;

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

get\_random\_batch(d, batch, X, y);

net.current\_subdivision = i;

float err = train\_network\_datum(net, X, y);

sum += err;

}

free(X);

free(y);

return (float)sum/(n\*batch);

}

float train\_network(network net, data d)

{

return train\_network\_waitkey(net, d, 0);

}

float train\_network\_waitkey(network net, data d, int wait\_key)

{

assert(d.X.rows % net.batch == 0);

int batch = net.batch;

int n = d.X.rows / batch;

float\* X = (float\*)xcalloc(batch \* d.X.cols, sizeof(float));

float\* y = (float\*)xcalloc(batch \* d.y.cols, sizeof(float));

int i;

float sum = 0;

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

get\_next\_batch(d, batch, i\*batch, X, y);

net.current\_subdivision = i;

float err = train\_network\_datum(net, X, y);

sum += err;

if(wait\_key) wait\_key\_cv(5);

}

(\*net.cur\_iteration) += 1;

#ifdef GPU

update\_network\_gpu(net);

#else // GPU

update\_network(net);

#endif // GPU

free(X);

free(y);

return (float)sum/(n\*batch);

}

float train\_network\_batch(network net, data d, int n)

{

int i,j;

network\_state state={0};

state.index = 0;

state.net = net;

state.train = 1;

state.delta = 0;

float sum = 0;

int batch = 2;

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

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

int index = random\_gen()%d.X.rows;

state.input = d.X.vals[index];

state.truth = d.y.vals[index];

forward\_network(net, state);

backward\_network(net, state);

sum += get\_network\_cost(net);

}

update\_network(net);

}

return (float)sum/(n\*batch);

}

int recalculate\_workspace\_size(network \*net)

{

#ifdef GPU

cuda\_set\_device(net->gpu\_index);

if (gpu\_index >= 0) cuda\_free(net->workspace);

#endif

int i;

size\_t workspace\_size = 0;

for (i = 0; i < net->n; ++i) {

layer l = net->layers[i];

//printf(" %d: layer = %d,", i, l.type);

if (l.type == CONVOLUTIONAL) {

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

}

else if (l.type == CONNECTED) {

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

}

if (l.workspace\_size > workspace\_size) workspace\_size = l.workspace\_size;

net->layers[i] = l;

}

#ifdef GPU

if (gpu\_index >= 0) {

printf("\n try to allocate additional workspace\_size = %1.2f MB \n", (float)workspace\_size / 1000000);

net->workspace = cuda\_make\_array(0, workspace\_size / sizeof(float) + 1);

printf(" CUDA allocate done! \n");

}

else {

free(net->workspace);

net->workspace = (float\*)xcalloc(1, workspace\_size);

}

#else

free(net->workspace);

net->workspace = (float\*)xcalloc(1, workspace\_size);

#endif

//fprintf(stderr, " Done!\n");

return 0;

}

void set\_batch\_network(network \*net, int b)

{

net->batch = b;

int i;

for(i = 0; i < net->n; ++i){

net->layers[i].batch = b;

#ifdef CUDNN

if(net->layers[i].type == CONVOLUTIONAL){

cudnn\_convolutional\_setup(net->layers + i, cudnn\_fastest, 0);

}

else if (net->layers[i].type == MAXPOOL) {

cudnn\_maxpool\_setup(net->layers + i);

}

#endif

}

recalculate\_workspace\_size(net); // recalculate workspace size

}

int resize\_network(network \*net, int w, int h)

{

#ifdef GPU

cuda\_set\_device(net->gpu\_index);

if(gpu\_index >= 0){

cuda\_free(net->workspace);

if (net->input\_gpu) {

cuda\_free(\*net->input\_gpu);

\*net->input\_gpu = 0;

cuda\_free(\*net->truth\_gpu);

\*net->truth\_gpu = 0;

}

if (net->input\_state\_gpu) cuda\_free(net->input\_state\_gpu);

if (net->input\_pinned\_cpu) {

if (net->input\_pinned\_cpu\_flag) cudaFreeHost(net->input\_pinned\_cpu);

else free(net->input\_pinned\_cpu);

}

}

#endif

int i;

//if(w == net->w && h == net->h) return 0;

net->w = w;

net->h = h;

int inputs = 0;

size\_t workspace\_size = 0;

//fprintf(stderr, "Resizing to %d x %d...\n", w, h);

//fflush(stderr);

for (i = 0; i < net->n; ++i){

layer l = net->layers[i];

//printf(" (resize %d: layer = %d) , ", i, l.type);

if(l.type == CONVOLUTIONAL){

resize\_convolutional\_layer(&l, w, h);

}

else if (l.type == CRNN) {

resize\_crnn\_layer(&l, w, h);

}else if (l.type == CONV\_LSTM) {

resize\_conv\_lstm\_layer(&l, w, h);

}else if(l.type == CROP){

resize\_crop\_layer(&l, w, h);

}else if(l.type == MAXPOOL){

resize\_maxpool\_layer(&l, w, h);

}else if (l.type == LOCAL\_AVGPOOL) {

resize\_maxpool\_layer(&l, w, h);

}else if (l.type == BATCHNORM) {

resize\_batchnorm\_layer(&l, w, h);

}else if(l.type == REGION){

resize\_region\_layer(&l, w, h);

}else if (l.type == YOLO) {

resize\_yolo\_layer(&l, w, h);

}else if (l.type == GAUSSIAN\_YOLO) {

resize\_gaussian\_yolo\_layer(&l, w, h);

}else if(l.type == ROUTE){

resize\_route\_layer(&l, net);

}else if (l.type == SHORTCUT) {

resize\_shortcut\_layer(&l, w, h, net);

}else if (l.type == SCALE\_CHANNELS) {

resize\_scale\_channels\_layer(&l, net);

}else if (l.type == SAM) {

resize\_sam\_layer(&l, w, h);

}else if (l.type == DROPOUT) {

resize\_dropout\_layer(&l, inputs);

l.out\_w = l.w = w;

l.out\_h = l.h = h;

l.output = net->layers[i - 1].output;

l.delta = net->layers[i - 1].delta;

#ifdef GPU

l.output\_gpu = net->layers[i-1].output\_gpu;

l.delta\_gpu = net->layers[i-1].delta\_gpu;

#endif

}else if (l.type == UPSAMPLE) {

resize\_upsample\_layer(&l, w, h);

}else if(l.type == REORG){

resize\_reorg\_layer(&l, w, h);

} else if (l.type == REORG\_OLD) {

resize\_reorg\_old\_layer(&l, w, h);

}else if(l.type == AVGPOOL){

resize\_avgpool\_layer(&l, w, h);

}else if(l.type == NORMALIZATION){

resize\_normalization\_layer(&l, w, h);

}else if(l.type == COST){

resize\_cost\_layer(&l, inputs);

}else{

fprintf(stderr, "Resizing type %d \n", (int)l.type);

error("Cannot resize this type of layer");

}

if(l.workspace\_size > workspace\_size) workspace\_size = l.workspace\_size;

inputs = l.outputs;

net->layers[i] = l;

//if(l.type != DROPOUT)

{

w = l.out\_w;

h = l.out\_h;

}

//if(l.type == AVGPOOL) break;

}

#ifdef GPU

const int size = get\_network\_input\_size(\*net) \* net->batch;

if(gpu\_index >= 0){

printf(" try to allocate additional workspace\_size = %1.2f MB \n", (float)workspace\_size / 1000000);

net->workspace = cuda\_make\_array(0, workspace\_size/sizeof(float) + 1);

net->input\_state\_gpu = cuda\_make\_array(0, size);

if (cudaSuccess == cudaHostAlloc(&net->input\_pinned\_cpu, size \* sizeof(float), cudaHostRegisterMapped))

net->input\_pinned\_cpu\_flag = 1;

else {

cudaGetLastError(); // reset CUDA-error

net->input\_pinned\_cpu = (float\*)xcalloc(size, sizeof(float));

net->input\_pinned\_cpu\_flag = 0;

}

printf(" CUDA allocate done! \n");

}else {

free(net->workspace);

net->workspace = (float\*)xcalloc(1, workspace\_size);

if(!net->input\_pinned\_cpu\_flag)

net->input\_pinned\_cpu = (float\*)xrealloc(net->input\_pinned\_cpu, size \* sizeof(float));

}

#else

free(net->workspace);

net->workspace = (float\*)xcalloc(1, workspace\_size);

#endif

//fprintf(stderr, " Done!\n");

return 0;

}

int get\_network\_output\_size(network net)

{

int i;

for(i = net.n-1; i > 0; --i) if(net.layers[i].type != COST) break;

return net.layers[i].outputs;

}

int get\_network\_input\_size(network net)

{

return net.layers[0].inputs;

}

detection\_layer get\_network\_detection\_layer(network net)

{

int i;

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

if(net.layers[i].type == DETECTION){

return net.layers[i];

}

}

fprintf(stderr, "Detection layer not found!!\n");

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

return l;

}

image get\_network\_image\_layer(network net, int i)

{

layer l = net.layers[i];

if (l.out\_w && l.out\_h && l.out\_c){

return float\_to\_image(l.out\_w, l.out\_h, l.out\_c, l.output);

}

image def = {0};

return def;

}

layer\* get\_network\_layer(network\* net, int i)

{

return net->layers + i;

}

image get\_network\_image(network net)

{

int i;

for(i = net.n-1; i >= 0; --i){

image m = get\_network\_image\_layer(net, i);

if(m.h != 0) return m;

}

image def = {0};

return def;

}

void visualize\_network(network net)

{

image \*prev = 0;

int i;

char buff[256];

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

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

layer l = net.layers[i];

if(l.type == CONVOLUTIONAL){

prev = visualize\_convolutional\_layer(l, buff, prev);

}

}

}

void top\_predictions(network net, int k, int \*index)

{

int size = get\_network\_output\_size(net);

float \*out = get\_network\_output(net);

top\_k(out, size, k, index);

}

// A version of network\_predict that uses a pointer for the network

// struct to make the python binding work properly.

float \*network\_predict\_ptr(network \*net, float \*input)

{

return network\_predict(\*net, input);

}

float \*network\_predict(network net, float \*input)

{

#ifdef GPU

if(gpu\_index >= 0) return network\_predict\_gpu(net, input);

#endif

network\_state state = {0};

state.net = net;

state.index = 0;

state.input = input;

state.truth = 0;

state.train = 0;

state.delta = 0;

forward\_network(net, state);

float \*out = get\_network\_output(net);

return out;

}

int num\_detections(network \*net, float thresh)

{

int i;

int s = 0;

for (i = 0; i < net->n; ++i) {

layer l = net->layers[i];

if (l.type == YOLO) {

s += yolo\_num\_detections(l, thresh);

}

if (l.type == GAUSSIAN\_YOLO) {

s += gaussian\_yolo\_num\_detections(l, thresh);

}

if (l.type == DETECTION || l.type == REGION) {

s += l.w\*l.h\*l.n;

}

}

return s;

}

int num\_detections\_batch(network \*net, float thresh, int batch)

{

int i;

int s = 0;

for (i = 0; i < net->n; ++i) {

layer l = net->layers[i];

if (l.type == YOLO) {

s += yolo\_num\_detections\_batch(l, thresh, batch);

}

if (l.type == DETECTION || l.type == REGION) {

s += l.w\*l.h\*l.n;

}

}

return s;

}

detection \*make\_network\_boxes(network \*net, float thresh, int \*num)

{

layer l = net->layers[net->n - 1];

int i;

int nboxes = num\_detections(net, thresh);

if (num) \*num = nboxes;

detection\* dets = (detection\*)xcalloc(nboxes, sizeof(detection));

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

dets[i].prob = (float\*)xcalloc(l.classes, sizeof(float));

// tx,ty,tw,th uncertainty

dets[i].uc = (float\*)xcalloc(4, sizeof(float)); // Gaussian\_YOLOv3

if (l.coords > 4) {

dets[i].mask = (float\*)xcalloc(l.coords - 4, sizeof(float));

}

}

return dets;

}

detection \*make\_network\_boxes\_batch(network \*net, float thresh, int \*num, int batch)

{

int i;

layer l = net->layers[net->n - 1];

int nboxes = num\_detections\_batch(net, thresh, batch);

assert(num != NULL);

\*num = nboxes;

detection\* dets = (detection\*)calloc(nboxes, sizeof(detection));

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

dets[i].prob = (float\*)calloc(l.classes, sizeof(float));

if (l.coords > 4) {

dets[i].mask = (float\*)calloc(l.coords - 4, sizeof(float));

}

}

return dets;

}

void custom\_get\_region\_detections(layer l, int w, int h, int net\_w, int net\_h, float thresh, int \*map, float hier, int relative, detection \*dets, int letter)

{

box\* boxes = (box\*)xcalloc(l.w \* l.h \* l.n, sizeof(box));

float\*\* probs = (float\*\*)xcalloc(l.w \* l.h \* l.n, sizeof(float\*));

int i, j;

for (j = 0; j < l.w\*l.h\*l.n; ++j) probs[j] = (float\*)xcalloc(l.classes, sizeof(float));

get\_region\_boxes(l, 1, 1, thresh, probs, boxes, 0, map);

for (j = 0; j < l.w\*l.h\*l.n; ++j) {

dets[j].classes = l.classes;

dets[j].bbox = boxes[j];

dets[j].objectness = 1;

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

dets[j].prob[i] = probs[j][i];

}

}

free(boxes);

free\_ptrs((void \*\*)probs, l.w\*l.h\*l.n);

//correct\_region\_boxes(dets, l.w\*l.h\*l.n, w, h, net\_w, net\_h, relative);

correct\_yolo\_boxes(dets, l.w\*l.h\*l.n, w, h, net\_w, net\_h, relative, letter);

}

void fill\_network\_boxes(network \*net, int w, int h, float thresh, float hier, int \*map, int relative, detection \*dets, int letter)

{

int prev\_classes = -1;

int j;

for (j = 0; j < net->n; ++j) {

layer l = net->layers[j];

if (l.type == YOLO) {

int count = get\_yolo\_detections(l, w, h, net->w, net->h, thresh, map, relative, dets, letter);

dets += count;

if (prev\_classes < 0) prev\_classes = l.classes;

else if (prev\_classes != l.classes) {

printf(" Error: Different [yolo] layers have different number of classes = %d and %d - check your cfg-file! \n",

prev\_classes, l.classes);

}

}

if (l.type == GAUSSIAN\_YOLO) {

int count = get\_gaussian\_yolo\_detections(l, w, h, net->w, net->h, thresh, map, relative, dets, letter);

dets += count;

}

if (l.type == REGION) {

custom\_get\_region\_detections(l, w, h, net->w, net->h, thresh, map, hier, relative, dets, letter);

//get\_region\_detections(l, w, h, net->w, net->h, thresh, map, hier, relative, dets);

dets += l.w\*l.h\*l.n;

}

if (l.type == DETECTION) {

get\_detection\_detections(l, w, h, thresh, dets);

dets += l.w\*l.h\*l.n;

}

}

}

void fill\_network\_boxes\_batch(network \*net, int w, int h, float thresh, float hier, int \*map, int relative, detection \*dets, int letter, int batch)

{

int prev\_classes = -1;

int j;

for (j = 0; j < net->n; ++j) {

layer l = net->layers[j];

if (l.type == YOLO) {

int count = get\_yolo\_detections\_batch(l, w, h, net->w, net->h, thresh, map, relative, dets, letter, batch);

dets += count;

if (prev\_classes < 0) prev\_classes = l.classes;

else if (prev\_classes != l.classes) {

printf(" Error: Different [yolo] layers have different number of classes = %d and %d - check your cfg-file! \n",

prev\_classes, l.classes);

}

}

if (l.type == REGION) {

custom\_get\_region\_detections(l, w, h, net->w, net->h, thresh, map, hier, relative, dets, letter);

//get\_region\_detections(l, w, h, net->w, net->h, thresh, map, hier, relative, dets);

dets += l.w\*l.h\*l.n;

}

if (l.type == DETECTION) {

get\_detection\_detections(l, w, h, thresh, dets);

dets += l.w\*l.h\*l.n;

}

}

}

detection \*get\_network\_boxes(network \*net, int w, int h, float thresh, float hier, int \*map, int relative, int \*num, int letter)

{

detection \*dets = make\_network\_boxes(net, thresh, num);

fill\_network\_boxes(net, w, h, thresh, hier, map, relative, dets, letter);

return dets;

}

void free\_detections(detection \*dets, int n)

{

int i;

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

free(dets[i].prob);

if (dets[i].uc) free(dets[i].uc);

if (dets[i].mask) free(dets[i].mask);

}

free(dets);

}

void free\_batch\_detections(det\_num\_pair \*det\_num\_pairs, int n)

{

int i;

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

free\_detections(det\_num\_pairs[i].dets, det\_num\_pairs[i].num);

free(det\_num\_pairs);

}

// JSON format:

//{

// "frame\_id":8990,

// "objects":[

// {"class\_id":4, "name":"aeroplane", "relative coordinates":{"center\_x":0.398831, "center\_y":0.630203, "width":0.057455, "height":0.020396}, "confidence":0.793070},

// {"class\_id":14, "name":"bird", "relative coordinates":{"center\_x":0.398831, "center\_y":0.630203, "width":0.057455, "height":0.020396}, "confidence":0.265497}

// ]

//},

char \*detection\_to\_json(detection \*dets, int nboxes, int classes, char \*\*names, long long int frame\_id, char \*filename)

{

const float thresh = 0.005; // function get\_network\_boxes() has already filtred dets by actual threshold

char \*send\_buf = (char \*)calloc(1024, sizeof(char));

if (!send\_buf) return 0;

if (filename) {

sprintf(send\_buf, "{\n \"frame\_id\":%lld, \n \"filename\":\"%s\", \n \"objects\": [ \n", frame\_id, filename);

}

else {

sprintf(send\_buf, "{\n \"frame\_id\":%lld, \n \"objects\": [ \n", frame\_id);

}

int i, j;

int class\_id = -1;

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

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

int show = strncmp(names[j], "dont\_show", 9);

if (dets[i].prob[j] > thresh && show)

{

if (class\_id != -1) strcat(send\_buf, ", \n");

class\_id = j;

char \*buf = (char \*)calloc(2048, sizeof(char));

if (!buf) return 0;

//sprintf(buf, "{\"image\_id\":%d, \"category\_id\":%d, \"bbox\":[%f, %f, %f, %f], \"score\":%f}",

// image\_id, j, dets[i].bbox.x, dets[i].bbox.y, dets[i].bbox.w, dets[i].bbox.h, dets[i].prob[j]);

sprintf(buf, " {\"class\_id\":%d, \"name\":\"%s\", \"relative\_coordinates\":{\"center\_x\":%f, \"center\_y\":%f, \"width\":%f, \"height\":%f}, \"confidence\":%f}",

j, names[j], dets[i].bbox.x, dets[i].bbox.y, dets[i].bbox.w, dets[i].bbox.h, dets[i].prob[j]);

int send\_buf\_len = strlen(send\_buf);

int buf\_len = strlen(buf);

int total\_len = send\_buf\_len + buf\_len + 100;

send\_buf = (char \*)realloc(send\_buf, total\_len \* sizeof(char));

if (!send\_buf) {

if (buf) free(buf);

return 0;// exit(-1);

}

strcat(send\_buf, buf);

free(buf);

}

}

}

strcat(send\_buf, "\n ] \n}");

return send\_buf;

}

float \*network\_predict\_image(network \*net, image im)

{

//image imr = letterbox\_image(im, net->w, net->h);

float \*p;

if(net->batch != 1) set\_batch\_network(net, 1);

if (im.w == net->w && im.h == net->h) {

// Input image is the same size as our net, predict on that image

p = network\_predict(\*net, im.data);

}

else {

// Need to resize image to the desired size for the net

image imr = resize\_image(im, net->w, net->h);

p = network\_predict(\*net, imr.data);

free\_image(imr);

}

return p;

}

det\_num\_pair\* network\_predict\_batch(network \*net, image im, int batch\_size, int w, int h, float thresh, float hier, int \*map, int relative, int letter)

{

network\_predict(\*net, im.data);

det\_num\_pair \*pdets = (struct det\_num\_pair \*)calloc(batch\_size, sizeof(det\_num\_pair));

int num;

int batch;

for(batch=0; batch < batch\_size; batch++){

detection \*dets = make\_network\_boxes\_batch(net, thresh, &num, batch);

fill\_network\_boxes\_batch(net, w, h, thresh, hier, map, relative, dets, letter, batch);

pdets[batch].num = num;

pdets[batch].dets = dets;

}

return pdets;

}

float \*network\_predict\_image\_letterbox(network \*net, image im)

{

//image imr = letterbox\_image(im, net->w, net->h);

float \*p;

if (net->batch != 1) set\_batch\_network(net, 1);

if (im.w == net->w && im.h == net->h) {

// Input image is the same size as our net, predict on that image

p = network\_predict(\*net, im.data);

}

else {

// Need to resize image to the desired size for the net

image imr = letterbox\_image(im, net->w, net->h);

p = network\_predict(\*net, imr.data);

free\_image(imr);

}

return p;

}

int network\_width(network \*net) { return net->w; }

int network\_height(network \*net) { return net->h; }

matrix network\_predict\_data\_multi(network net, data test, int n)

{

int i,j,b,m;

int k = get\_network\_output\_size(net);

matrix pred = make\_matrix(test.X.rows, k);

float\* X = (float\*)xcalloc(net.batch \* test.X.rows, sizeof(float));

for(i = 0; i < test.X.rows; i += net.batch){

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

if(i+b == test.X.rows) break;

memcpy(X+b\*test.X.cols, test.X.vals[i+b], test.X.cols\*sizeof(float));

}

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

float \*out = network\_predict(net, X);

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

if(i+b == test.X.rows) break;

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

pred.vals[i+b][j] += out[j+b\*k]/n;

}

}

}

}

free(X);

return pred;

}

matrix network\_predict\_data(network net, data test)

{

int i,j,b;

int k = get\_network\_output\_size(net);

matrix pred = make\_matrix(test.X.rows, k);

float\* X = (float\*)xcalloc(net.batch \* test.X.cols, sizeof(float));

for(i = 0; i < test.X.rows; i += net.batch){

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

if(i+b == test.X.rows) break;

memcpy(X+b\*test.X.cols, test.X.vals[i+b], test.X.cols\*sizeof(float));

}

float \*out = network\_predict(net, X);

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

if(i+b == test.X.rows) break;

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

pred.vals[i+b][j] = out[j+b\*k];

}

}

}

free(X);

return pred;

}

void print\_network(network net)

{

int i,j;

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

layer l = net.layers[i];

float \*output = l.output;

int n = l.outputs;

float mean = mean\_array(output, n);

float vari = variance\_array(output, n);

fprintf(stderr, "Layer %d - Mean: %f, Variance: %f\n",i,mean, vari);

if(n > 100) n = 100;

for(j = 0; j < n; ++j) fprintf(stderr, "%f, ", output[j]);

if(n == 100)fprintf(stderr,".....\n");

fprintf(stderr, "\n");

}

}

void compare\_networks(network n1, network n2, data test)

{

matrix g1 = network\_predict\_data(n1, test);

matrix g2 = network\_predict\_data(n2, test);

int i;

int a,b,c,d;

a = b = c = d = 0;

for(i = 0; i < g1.rows; ++i){

int truth = max\_index(test.y.vals[i], test.y.cols);

int p1 = max\_index(g1.vals[i], g1.cols);

int p2 = max\_index(g2.vals[i], g2.cols);

if(p1 == truth){

if(p2 == truth) ++d;

else ++c;

}else{

if(p2 == truth) ++b;

else ++a;

}

}

printf("%5d %5d\n%5d %5d\n", a, b, c, d);

float num = pow((abs(b - c) - 1.), 2.);

float den = b + c;

printf("%f\n", num/den);

}

float network\_accuracy(network net, data d)

{

matrix guess = network\_predict\_data(net, d);

float acc = matrix\_topk\_accuracy(d.y, guess,1);

free\_matrix(guess);

return acc;

}

float \*network\_accuracies(network net, data d, int n)

{

static float acc[2];

matrix guess = network\_predict\_data(net, d);

acc[0] = matrix\_topk\_accuracy(d.y, guess, 1);

acc[1] = matrix\_topk\_accuracy(d.y, guess, n);

free\_matrix(guess);

return acc;

}

float network\_accuracy\_multi(network net, data d, int n)

{

matrix guess = network\_predict\_data\_multi(net, d, n);

float acc = matrix\_topk\_accuracy(d.y, guess,1);

free\_matrix(guess);

return acc;

}

void free\_network(network net)

{

int i;

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

free\_layer(net.layers[i]);

}

free(net.layers);

free(net.seq\_scales);

free(net.scales);

free(net.steps);

free(net.seen);

free(net.cur\_iteration);

#ifdef GPU

if (gpu\_index >= 0) cuda\_free(net.workspace);

else free(net.workspace);

free\_pinned\_memory();

if (net.input\_state\_gpu) cuda\_free(net.input\_state\_gpu);

if (net.input\_pinned\_cpu) { // CPU

if (net.input\_pinned\_cpu\_flag) cudaFreeHost(net.input\_pinned\_cpu);

else free(net.input\_pinned\_cpu);

}

if (\*net.input\_gpu) cuda\_free(\*net.input\_gpu);

if (\*net.truth\_gpu) cuda\_free(\*net.truth\_gpu);

if (net.input\_gpu) free(net.input\_gpu);

if (net.truth\_gpu) free(net.truth\_gpu);

if (\*net.input16\_gpu) cuda\_free(\*net.input16\_gpu);

if (\*net.output16\_gpu) cuda\_free(\*net.output16\_gpu);

if (net.input16\_gpu) free(net.input16\_gpu);

if (net.output16\_gpu) free(net.output16\_gpu);

if (net.max\_input16\_size) free(net.max\_input16\_size);

if (net.max\_output16\_size) free(net.max\_output16\_size);

#else

free(net.workspace);

#endif

}

static float relu(float src) {

if (src > 0) return src;

return 0;

}

static float lrelu(float src) {

const float eps = 0.001;

if (src > eps) return src;

return eps;

}

void fuse\_conv\_batchnorm(network net)

{

int j;

for (j = 0; j < net.n; ++j) {

layer \*l = &net.layers[j];

if (l->type == CONVOLUTIONAL) {

//printf(" Merges Convolutional-%d and batch\_norm \n", j);

if (l->share\_layer != NULL) {

l->batch\_normalize = 0;

}

if (l->batch\_normalize) {

int f;

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

{

l->biases[f] = l->biases[f] - (double)l->scales[f] \* l->rolling\_mean[f] / (sqrt((double)l->rolling\_variance[f] + .00001));

const size\_t filter\_size = l->size\*l->size\*l->c / l->groups;

int i;

for (i = 0; i < filter\_size; ++i) {

int w\_index = f\*filter\_size + i;

l->weights[w\_index] = (double)l->weights[w\_index] \* l->scales[f] / (sqrt((double)l->rolling\_variance[f] + .00001));

}

}

free\_convolutional\_batchnorm(l);

l->batch\_normalize = 0;

#ifdef GPU

if (gpu\_index >= 0) {

push\_convolutional\_layer(\*l);

}

#endif

}

}

else if (l->type == SHORTCUT && l->weights && l->weights\_normalizion)

{

if (l->nweights > 0) {

//cuda\_pull\_array(l.weights\_gpu, l.weights, l.nweights);

int i;

for (i = 0; i < l->nweights; ++i) printf(" w = %f,", l->weights[i]);

printf(" l->nweights = %d, j = %d \n", l->nweights, j);

}

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

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

int chan, i;

for (chan = 0; chan < layer\_step; ++chan)

{

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

if (l->weights\_normalizion == SOFTMAX\_NORMALIZATION) {

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

int w\_index = chan + i \* layer\_step;

float w = l->weights[w\_index];

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

}

}

const float eps = 0.0001;

sum = eps;

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

int w\_index = chan + i \* layer\_step;

float w = l->weights[w\_index];

if (l->weights\_normalizion == RELU\_NORMALIZATION) sum += lrelu(w);

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

}

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

int w\_index = chan + i \* layer\_step;

float w = l->weights[w\_index];

if (l->weights\_normalizion == RELU\_NORMALIZATION) w = lrelu(w) / sum;

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

l->weights[w\_index] = w;

}

}

l->weights\_normalizion = NO\_NORMALIZATION;

#ifdef GPU

if (gpu\_index >= 0) {

push\_shortcut\_layer(\*l);

}

#endif

}

else {

//printf(" Fusion skip layer type: %d \n", l->type);

}

}

}

void forward\_blank\_layer(layer l, network\_state state) {}

void calculate\_binary\_weights(network net)

{

int j;

for (j = 0; j < net.n; ++j) {

layer \*l = &net.layers[j];

if (l->type == CONVOLUTIONAL) {

//printf(" Merges Convolutional-%d and batch\_norm \n", j);

if (l->xnor) {

//printf("\n %d \n", j);

//l->lda\_align = 256; // 256bit for AVX2 // set in make\_convolutional\_layer()

//if (l->size\*l->size\*l->c >= 2048) l->lda\_align = 512;

binary\_align\_weights(l);

if (net.layers[j].use\_bin\_output) {

l->activation = LINEAR;

}

#ifdef GPU

// fuse conv\_xnor + shortcut -> conv\_xnor

if ((j + 1) < net.n && net.layers[j].type == CONVOLUTIONAL) {

layer \*sc = &net.layers[j + 1];

if (sc->type == SHORTCUT && sc->w == sc->out\_w && sc->h == sc->out\_h && sc->c == sc->out\_c)

{

l->bin\_conv\_shortcut\_in\_gpu = net.layers[net.layers[j + 1].index].output\_gpu;

l->bin\_conv\_shortcut\_out\_gpu = net.layers[j + 1].output\_gpu;

net.layers[j + 1].type = BLANK;

net.layers[j + 1].forward\_gpu = forward\_blank\_layer;

}

}

#endif // GPU

}

}

}

//printf("\n calculate\_binary\_weights Done! \n");

}

void copy\_cudnn\_descriptors(layer src, layer \*dst)

{

#ifdef CUDNN

dst->normTensorDesc = src.normTensorDesc;

dst->normDstTensorDesc = src.normDstTensorDesc;

dst->normDstTensorDescF16 = src.normDstTensorDescF16;

dst->srcTensorDesc = src.srcTensorDesc;

dst->dstTensorDesc = src.dstTensorDesc;

dst->srcTensorDesc16 = src.srcTensorDesc16;

dst->dstTensorDesc16 = src.dstTensorDesc16;

#endif // CUDNN

}

void copy\_weights\_net(network net\_train, network \*net\_map)

{

int k;

for (k = 0; k < net\_train.n; ++k) {

layer \*l = &(net\_train.layers[k]);

layer tmp\_layer;

copy\_cudnn\_descriptors(net\_map->layers[k], &tmp\_layer);

net\_map->layers[k] = net\_train.layers[k];

copy\_cudnn\_descriptors(tmp\_layer, &net\_map->layers[k]);

if (l->type == CRNN) {

layer tmp\_input\_layer, tmp\_self\_layer, tmp\_output\_layer;

copy\_cudnn\_descriptors(\*net\_map->layers[k].input\_layer, &tmp\_input\_layer);

copy\_cudnn\_descriptors(\*net\_map->layers[k].self\_layer, &tmp\_self\_layer);

copy\_cudnn\_descriptors(\*net\_map->layers[k].output\_layer, &tmp\_output\_layer);

net\_map->layers[k].input\_layer = net\_train.layers[k].input\_layer;

net\_map->layers[k].self\_layer = net\_train.layers[k].self\_layer;

net\_map->layers[k].output\_layer = net\_train.layers[k].output\_layer;

//net\_map->layers[k].output\_gpu = net\_map->layers[k].output\_layer->output\_gpu; // already copied out of if()

copy\_cudnn\_descriptors(tmp\_input\_layer, net\_map->layers[k].input\_layer);

copy\_cudnn\_descriptors(tmp\_self\_layer, net\_map->layers[k].self\_layer);

copy\_cudnn\_descriptors(tmp\_output\_layer, net\_map->layers[k].output\_layer);

}

else if(l->input\_layer) // for AntiAliasing

{

layer tmp\_input\_layer;

copy\_cudnn\_descriptors(\*net\_map->layers[k].input\_layer, &tmp\_input\_layer);

net\_map->layers[k].input\_layer = net\_train.layers[k].input\_layer;

copy\_cudnn\_descriptors(tmp\_input\_layer, net\_map->layers[k].input\_layer);

}

net\_map->layers[k].batch = 1;

net\_map->layers[k].steps = 1;

}

}

// combine Training and Validation networks

network combine\_train\_valid\_networks(network net\_train, network net\_map)

{

network net\_combined = make\_network(net\_train.n);

layer \*old\_layers = net\_combined.layers;

net\_combined = net\_train;

net\_combined.layers = old\_layers;

net\_combined.batch = 1;

int k;

for (k = 0; k < net\_train.n; ++k) {

layer \*l = &(net\_train.layers[k]);

net\_combined.layers[k] = net\_train.layers[k];

net\_combined.layers[k].batch = 1;

if (l->type == CONVOLUTIONAL) {

#ifdef CUDNN

net\_combined.layers[k].normTensorDesc = net\_map.layers[k].normTensorDesc;

net\_combined.layers[k].normDstTensorDesc = net\_map.layers[k].normDstTensorDesc;

net\_combined.layers[k].normDstTensorDescF16 = net\_map.layers[k].normDstTensorDescF16;

net\_combined.layers[k].srcTensorDesc = net\_map.layers[k].srcTensorDesc;

net\_combined.layers[k].dstTensorDesc = net\_map.layers[k].dstTensorDesc;

net\_combined.layers[k].srcTensorDesc16 = net\_map.layers[k].srcTensorDesc16;

net\_combined.layers[k].dstTensorDesc16 = net\_map.layers[k].dstTensorDesc16;

#endif // CUDNN

}

}

return net\_combined;

}

void free\_network\_recurrent\_state(network net)

{

int k;

for (k = 0; k < net.n; ++k) {

if (net.layers[k].type == CONV\_LSTM) free\_state\_conv\_lstm(net.layers[k]);

if (net.layers[k].type == CRNN) free\_state\_crnn(net.layers[k]);

}

}

void randomize\_network\_recurrent\_state(network net)

{

int k;

for (k = 0; k < net.n; ++k) {

if (net.layers[k].type == CONV\_LSTM) randomize\_state\_conv\_lstm(net.layers[k]);

if (net.layers[k].type == CRNN) free\_state\_crnn(net.layers[k]);

}

}

void remember\_network\_recurrent\_state(network net)

{

int k;

for (k = 0; k < net.n; ++k) {

if (net.layers[k].type == CONV\_LSTM) remember\_state\_conv\_lstm(net.layers[k]);

//if (net.layers[k].type == CRNN) free\_state\_crnn(net.layers[k]);

}

}

void restore\_network\_recurrent\_state(network net)

{

int k;

for (k = 0; k < net.n; ++k) {

if (net.layers[k].type == CONV\_LSTM) restore\_state\_conv\_lstm(net.layers[k]);

if (net.layers[k].type == CRNN) free\_state\_crnn(net.layers[k]);

}

}