#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <stdint.h>

#include "activation\_layer.h"

#include "activations.h"

#include "assert.h"

#include "avgpool\_layer.h"

#include "batchnorm\_layer.h"

#include "blas.h"

#include "connected\_layer.h"

#include "convolutional\_layer.h"

#include "cost\_layer.h"

#include "crnn\_layer.h"

#include "crop\_layer.h"

#include "detection\_layer.h"

#include "dropout\_layer.h"

#include "gru\_layer.h"

#include "list.h"

#include "local\_layer.h"

#include "lstm\_layer.h"

#include "conv\_lstm\_layer.h"

#include "maxpool\_layer.h"

#include "normalization\_layer.h"

#include "option\_list.h"

#include "parser.h"

#include "region\_layer.h"

#include "reorg\_layer.h"

#include "reorg\_old\_layer.h"

#include "rnn\_layer.h"

#include "route\_layer.h"

#include "shortcut\_layer.h"

#include "scale\_channels\_layer.h"

#include "sam\_layer.h"

#include "softmax\_layer.h"

#include "utils.h"

#include "upsample\_layer.h"

#include "version.h"

#include "yolo\_layer.h"

#include "gaussian\_yolo\_layer.h"

typedef struct{

char \*type;

list \*options;

}section;

list \*read\_cfg(char \*filename);

LAYER\_TYPE string\_to\_layer\_type(char \* type)

{

if (strcmp(type, "[shortcut]")==0) return SHORTCUT;

if (strcmp(type, "[scale\_channels]") == 0) return SCALE\_CHANNELS;

if (strcmp(type, "[sam]") == 0) return SAM;

if (strcmp(type, "[crop]")==0) return CROP;

if (strcmp(type, "[cost]")==0) return COST;

if (strcmp(type, "[detection]")==0) return DETECTION;

if (strcmp(type, "[region]")==0) return REGION;

if (strcmp(type, "[yolo]") == 0) return YOLO;

if (strcmp(type, "[Gaussian\_yolo]") == 0) return GAUSSIAN\_YOLO;

if (strcmp(type, "[local]")==0) return LOCAL;

if (strcmp(type, "[conv]")==0

|| strcmp(type, "[convolutional]")==0) return CONVOLUTIONAL;

if (strcmp(type, "[activation]")==0) return ACTIVE;

if (strcmp(type, "[net]")==0

|| strcmp(type, "[network]")==0) return NETWORK;

if (strcmp(type, "[crnn]")==0) return CRNN;

if (strcmp(type, "[gru]")==0) return GRU;

if (strcmp(type, "[lstm]")==0) return LSTM;

if (strcmp(type, "[conv\_lstm]") == 0) return CONV\_LSTM;

if (strcmp(type, "[rnn]")==0) return RNN;

if (strcmp(type, "[conn]")==0

|| strcmp(type, "[connected]")==0) return CONNECTED;

if (strcmp(type, "[max]")==0

|| strcmp(type, "[maxpool]")==0) return MAXPOOL;

if (strcmp(type, "[local\_avg]") == 0

|| strcmp(type, "[local\_avgpool]") == 0) return LOCAL\_AVGPOOL;

if (strcmp(type, "[reorg3d]")==0) return REORG;

if (strcmp(type, "[reorg]") == 0) return REORG\_OLD;

if (strcmp(type, "[avg]")==0

|| strcmp(type, "[avgpool]")==0) return AVGPOOL;

if (strcmp(type, "[dropout]")==0) return DROPOUT;

if (strcmp(type, "[lrn]")==0

|| strcmp(type, "[normalization]")==0) return NORMALIZATION;

if (strcmp(type, "[batchnorm]")==0) return BATCHNORM;

if (strcmp(type, "[soft]")==0

|| strcmp(type, "[softmax]")==0) return SOFTMAX;

if (strcmp(type, "[route]")==0) return ROUTE;

if (strcmp(type, "[upsample]") == 0) return UPSAMPLE;

if (strcmp(type, "[empty]") == 0) return EMPTY;

return BLANK;

}

void free\_section(section \*s)

{

free(s->type);

node \*n = s->options->front;

while(n){

kvp \*pair = (kvp \*)n->val;

free(pair->key);

free(pair);

node \*next = n->next;

free(n);

n = next;

}

free(s->options);

free(s);

}

void parse\_data(char \*data, float \*a, int n)

{

int i;

if(!data) return;

char \*curr = data;

char \*next = data;

int done = 0;

for(i = 0; i < n && !done; ++i){

while(\*++next !='\0' && \*next != ',');

if(\*next == '\0') done = 1;

\*next = '\0';

sscanf(curr, "%g", &a[i]);

curr = next+1;

}

}

typedef struct size\_params{

int batch;

int inputs;

int h;

int w;

int c;

int index;

int time\_steps;

int train;

network net;

} size\_params;

local\_layer parse\_local(list \*options, size\_params params)

{

int n = option\_find\_int(options, "filters",1);

int size = option\_find\_int(options, "size",1);

int stride = option\_find\_int(options, "stride",1);

int pad = option\_find\_int(options, "pad",0);

char \*activation\_s = option\_find\_str(options, "activation", "logistic");

ACTIVATION activation = get\_activation(activation\_s);

int batch,h,w,c;

h = params.h;

w = params.w;

c = params.c;

batch=params.batch;

if(!(h && w && c)) error("Layer before local layer must output image.");

local\_layer layer = make\_local\_layer(batch,h,w,c,n,size,stride,pad,activation);

return layer;

}

convolutional\_layer parse\_convolutional(list \*options, size\_params params)

{

int n = option\_find\_int(options, "filters",1);

int groups = option\_find\_int\_quiet(options, "groups", 1);

int size = option\_find\_int(options, "size",1);

int stride = -1;

//int stride = option\_find\_int(options, "stride",1);

int stride\_x = option\_find\_int\_quiet(options, "stride\_x", -1);

int stride\_y = option\_find\_int\_quiet(options, "stride\_y", -1);

if (stride\_x < 1 || stride\_y < 1) {

stride = option\_find\_int(options, "stride", 1);

if (stride\_x < 1) stride\_x = stride;

if (stride\_y < 1) stride\_y = stride;

}

else {

stride = option\_find\_int\_quiet(options, "stride", 1);

}

int dilation = option\_find\_int\_quiet(options, "dilation", 1);

int antialiasing = option\_find\_int\_quiet(options, "antialiasing", 0);

if (size == 1) dilation = 1;

int pad = option\_find\_int\_quiet(options, "pad",0);

int padding = option\_find\_int\_quiet(options, "padding",0);

if(pad) padding = size/2;

char \*activation\_s = option\_find\_str(options, "activation", "logistic");

ACTIVATION activation = get\_activation(activation\_s);

int assisted\_excitation = option\_find\_float\_quiet(options, "assisted\_excitation", 0);

int share\_index = option\_find\_int\_quiet(options, "share\_index", -1000000000);

convolutional\_layer \*share\_layer = NULL;

if(share\_index >= 0) share\_layer = &params.net.layers[share\_index];

else if(share\_index != -1000000000) share\_layer = &params.net.layers[params.index + share\_index];

int batch,h,w,c;

h = params.h;

w = params.w;

c = params.c;

batch=params.batch;

if(!(h && w && c)) error("Layer before convolutional layer must output image.");

int batch\_normalize = option\_find\_int\_quiet(options, "batch\_normalize", 0);

int cbn = option\_find\_int\_quiet(options, "cbn", 0);

if (cbn) batch\_normalize = 2;

int binary = option\_find\_int\_quiet(options, "binary", 0);

int xnor = option\_find\_int\_quiet(options, "xnor", 0);

int use\_bin\_output = option\_find\_int\_quiet(options, "bin\_output", 0);

int sway = option\_find\_int\_quiet(options, "sway", 0);

int rotate = option\_find\_int\_quiet(options, "rotate", 0);

int stretch = option\_find\_int\_quiet(options, "stretch", 0);

int stretch\_sway = option\_find\_int\_quiet(options, "stretch\_sway", 0);

if ((sway + rotate + stretch + stretch\_sway) > 1) {

printf(" Error: should be used only 1 param: sway=1, rotate=1 or stretch=1 in the [convolutional] layer \n");

exit(0);

}

int deform = sway || rotate || stretch || stretch\_sway;

if (deform && size == 1) {

printf(" Error: params (sway=1, rotate=1 or stretch=1) should be used only with size >=3 in the [convolutional] layer \n");

exit(0);

}

convolutional\_layer layer = make\_convolutional\_layer(batch,1,h,w,c,n,groups,size,stride\_x,stride\_y,dilation,padding,activation, batch\_normalize, binary, xnor, params.net.adam, use\_bin\_output, params.index, antialiasing, share\_layer, assisted\_excitation, deform, params.train);

layer.flipped = option\_find\_int\_quiet(options, "flipped", 0);

layer.dot = option\_find\_float\_quiet(options, "dot", 0);

layer.sway = sway;

layer.rotate = rotate;

layer.stretch = stretch;

layer.stretch\_sway = stretch\_sway;

layer.angle = option\_find\_float\_quiet(options, "angle", 15);

if(params.net.adam){

layer.B1 = params.net.B1;

layer.B2 = params.net.B2;

layer.eps = params.net.eps;

}

return layer;

}

layer parse\_crnn(list \*options, size\_params params)

{

int size = option\_find\_int\_quiet(options, "size", 3);

int stride = option\_find\_int\_quiet(options, "stride", 1);

int dilation = option\_find\_int\_quiet(options, "dilation", 1);

int pad = option\_find\_int\_quiet(options, "pad", 0);

int padding = option\_find\_int\_quiet(options, "padding", 0);

if (pad) padding = size / 2;

int output\_filters = option\_find\_int(options, "output",1);

int hidden\_filters = option\_find\_int(options, "hidden",1);

int groups = option\_find\_int\_quiet(options, "groups", 1);

char \*activation\_s = option\_find\_str(options, "activation", "logistic");

ACTIVATION activation = get\_activation(activation\_s);

int batch\_normalize = option\_find\_int\_quiet(options, "batch\_normalize", 0);

int xnor = option\_find\_int\_quiet(options, "xnor", 0);

layer l = make\_crnn\_layer(params.batch, params.h, params.w, params.c, hidden\_filters, output\_filters, groups, params.time\_steps, size, stride, dilation, padding, activation, batch\_normalize, xnor, params.train);

l.shortcut = option\_find\_int\_quiet(options, "shortcut", 0);

return l;

}

layer parse\_rnn(list \*options, size\_params params)

{

int output = option\_find\_int(options, "output",1);

int hidden = option\_find\_int(options, "hidden",1);

char \*activation\_s = option\_find\_str(options, "activation", "logistic");

ACTIVATION activation = get\_activation(activation\_s);

int batch\_normalize = option\_find\_int\_quiet(options, "batch\_normalize", 0);

int logistic = option\_find\_int\_quiet(options, "logistic", 0);

layer l = make\_rnn\_layer(params.batch, params.inputs, hidden, output, params.time\_steps, activation, batch\_normalize, logistic);

l.shortcut = option\_find\_int\_quiet(options, "shortcut", 0);

return l;

}

layer parse\_gru(list \*options, size\_params params)

{

int output = option\_find\_int(options, "output",1);

int batch\_normalize = option\_find\_int\_quiet(options, "batch\_normalize", 0);

layer l = make\_gru\_layer(params.batch, params.inputs, output, params.time\_steps, batch\_normalize);

return l;

}

layer parse\_lstm(list \*options, size\_params params)

{

int output = option\_find\_int(options, "output",1);

int batch\_normalize = option\_find\_int\_quiet(options, "batch\_normalize", 0);

layer l = make\_lstm\_layer(params.batch, params.inputs, output, params.time\_steps, batch\_normalize);

return l;

}

layer parse\_conv\_lstm(list \*options, size\_params params)

{

// a ConvLSTM with a larger transitional kernel should be able to capture faster motions

int size = option\_find\_int\_quiet(options, "size", 3);

int stride = option\_find\_int\_quiet(options, "stride", 1);

int dilation = option\_find\_int\_quiet(options, "dilation", 1);

int pad = option\_find\_int\_quiet(options, "pad", 0);

int padding = option\_find\_int\_quiet(options, "padding", 0);

if (pad) padding = size / 2;

int output\_filters = option\_find\_int(options, "output", 1);

int groups = option\_find\_int\_quiet(options, "groups", 1);

char \*activation\_s = option\_find\_str(options, "activation", "LINEAR");

ACTIVATION activation = get\_activation(activation\_s);

int batch\_normalize = option\_find\_int\_quiet(options, "batch\_normalize", 0);

int xnor = option\_find\_int\_quiet(options, "xnor", 0);

int peephole = option\_find\_int\_quiet(options, "peephole", 0);

layer l = make\_conv\_lstm\_layer(params.batch, params.h, params.w, params.c, output\_filters, groups, params.time\_steps, size, stride, dilation, padding, activation, batch\_normalize, peephole, xnor, params.train);

l.state\_constrain = option\_find\_int\_quiet(options, "state\_constrain", params.time\_steps \* 32);

l.shortcut = option\_find\_int\_quiet(options, "shortcut", 0);

return l;

}

connected\_layer parse\_connected(list \*options, size\_params params)

{

int output = option\_find\_int(options, "output",1);

char \*activation\_s = option\_find\_str(options, "activation", "logistic");

ACTIVATION activation = get\_activation(activation\_s);

int batch\_normalize = option\_find\_int\_quiet(options, "batch\_normalize", 0);

connected\_layer layer = make\_connected\_layer(params.batch, 1, params.inputs, output, activation, batch\_normalize);

return layer;

}

softmax\_layer parse\_softmax(list \*options, size\_params params)

{

int groups = option\_find\_int\_quiet(options, "groups", 1);

softmax\_layer layer = make\_softmax\_layer(params.batch, params.inputs, groups);

layer.temperature = option\_find\_float\_quiet(options, "temperature", 1);

char \*tree\_file = option\_find\_str(options, "tree", 0);

if (tree\_file) layer.softmax\_tree = read\_tree(tree\_file);

layer.w = params.w;

layer.h = params.h;

layer.c = params.c;

layer.spatial = option\_find\_float\_quiet(options, "spatial", 0);

layer.noloss = option\_find\_int\_quiet(options, "noloss", 0);

return layer;

}

int \*parse\_yolo\_mask(char \*a, int \*num)

{

int \*mask = 0;

if (a) {

int len = strlen(a);

int n = 1;

int i;

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

if (a[i] == ',') ++n;

}

mask = (int\*)xcalloc(n, sizeof(int));

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

int val = atoi(a);

mask[i] = val;

a = strchr(a, ',') + 1;

}

\*num = n;

}

return mask;

}

float \*get\_classes\_multipliers(char \*cpc, const int classes)

{

float \*classes\_multipliers = NULL;

if (cpc) {

int classes\_counters = classes;

int \*counters\_per\_class = parse\_yolo\_mask(cpc, &classes\_counters);

if (classes\_counters != classes) {

printf(" number of values in counters\_per\_class = %d doesn't match with classes = %d \n", classes\_counters, classes);

exit(0);

}

float max\_counter = 0;

int i;

for (i = 0; i < classes\_counters; ++i) if (max\_counter < counters\_per\_class[i]) max\_counter = counters\_per\_class[i];

classes\_multipliers = (float \*)calloc(classes\_counters, sizeof(float));

for (i = 0; i < classes\_counters; ++i) classes\_multipliers[i] = max\_counter / counters\_per\_class[i];

free(counters\_per\_class);

printf(" classes\_multipliers: ");

for (i = 0; i < classes\_counters; ++i) printf("%.1f, ", classes\_multipliers[i]);

printf("\n");

}

return classes\_multipliers;

}

layer parse\_yolo(list \*options, size\_params params)

{

int classes = option\_find\_int(options, "classes", 20);

int total = option\_find\_int(options, "num", 1);

int num = total;

char \*a = option\_find\_str(options, "mask", 0);

int \*mask = parse\_yolo\_mask(a, &num);

int max\_boxes = option\_find\_int\_quiet(options, "max", 90);

layer l = make\_yolo\_layer(params.batch, params.w, params.h, num, total, mask, classes, max\_boxes);

if (l.outputs != params.inputs) {

printf("Error: l.outputs == params.inputs \n");

printf("filters= in the [convolutional]-layer doesn't correspond to classes= or mask= in [yolo]-layer \n");

exit(EXIT\_FAILURE);

}

//assert(l.outputs == params.inputs);

char \*cpc = option\_find\_str(options, "counters\_per\_class", 0);

l.classes\_multipliers = get\_classes\_multipliers(cpc, classes);

l.label\_smooth\_eps = option\_find\_float\_quiet(options, "label\_smooth\_eps", 0.0f);

l.scale\_x\_y = option\_find\_float\_quiet(options, "scale\_x\_y", 1);

l.max\_delta = option\_find\_float\_quiet(options, "max\_delta", FLT\_MAX); // set 10

l.iou\_normalizer = option\_find\_float\_quiet(options, "iou\_normalizer", 0.75);

l.cls\_normalizer = option\_find\_float\_quiet(options, "cls\_normalizer", 1);

char \*iou\_loss = option\_find\_str\_quiet(options, "iou\_loss", "mse"); // "iou");

if (strcmp(iou\_loss, "mse") == 0) l.iou\_loss = MSE;

else if (strcmp(iou\_loss, "giou") == 0) l.iou\_loss = GIOU;

else if (strcmp(iou\_loss, "diou") == 0) l.iou\_loss = DIOU;

else if (strcmp(iou\_loss, "ciou") == 0) l.iou\_loss = CIOU;

else l.iou\_loss = IOU;

fprintf(stderr, "[yolo] params: iou loss: %s (%d), iou\_norm: %2.2f, cls\_norm: %2.2f, scale\_x\_y: %2.2f\n",

iou\_loss, l.iou\_loss, l.iou\_normalizer, l.cls\_normalizer, l.scale\_x\_y);

char \*iou\_thresh\_kind\_str = option\_find\_str\_quiet(options, "iou\_thresh\_kind", "iou");

if (strcmp(iou\_thresh\_kind\_str, "iou") == 0) l.iou\_thresh\_kind = IOU;

else if (strcmp(iou\_thresh\_kind\_str, "giou") == 0) l.iou\_thresh\_kind = GIOU;

else if (strcmp(iou\_thresh\_kind\_str, "diou") == 0) l.iou\_thresh\_kind = DIOU;

else if (strcmp(iou\_thresh\_kind\_str, "ciou") == 0) l.iou\_thresh\_kind = CIOU;

else {

fprintf(stderr, " Wrong iou\_thresh\_kind = %s \n", iou\_thresh\_kind\_str);

l.iou\_thresh\_kind = IOU;

}

l.beta\_nms = option\_find\_float\_quiet(options, "beta\_nms", 0.6);

char \*nms\_kind = option\_find\_str\_quiet(options, "nms\_kind", "default");

if (strcmp(nms\_kind, "default") == 0) l.nms\_kind = DEFAULT\_NMS;

else {

if (strcmp(nms\_kind, "greedynms") == 0) l.nms\_kind = GREEDY\_NMS;

else if (strcmp(nms\_kind, "diounms") == 0) l.nms\_kind = DIOU\_NMS;

else l.nms\_kind = DEFAULT\_NMS;

printf("nms\_kind: %s (%d), beta = %f \n", nms\_kind, l.nms\_kind, l.beta\_nms);

}

l.jitter = option\_find\_float(options, "jitter", .2);

l.focal\_loss = option\_find\_int\_quiet(options, "focal\_loss", 0);

l.ignore\_thresh = option\_find\_float(options, "ignore\_thresh", .5);

l.truth\_thresh = option\_find\_float(options, "truth\_thresh", 1);

l.iou\_thresh = option\_find\_float\_quiet(options, "iou\_thresh", 1); // recommended to use iou\_thresh=0.213 in [yolo]

l.random = option\_find\_float\_quiet(options, "random", 0);

char \*map\_file = option\_find\_str(options, "map", 0);

if (map\_file) l.map = read\_map(map\_file);

a = option\_find\_str(options, "anchors", 0);

if (a) {

int len = strlen(a);

int n = 1;

int i;

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

if (a[i] == ',') ++n;

}

for (i = 0; i < n && i < total\*2; ++i) {

float bias = atof(a);

l.biases[i] = bias;

a = strchr(a, ',') + 1;

}

}

return l;

}

int \*parse\_gaussian\_yolo\_mask(char \*a, int \*num) // Gaussian\_YOLOv3

{

int \*mask = 0;

if (a) {

int len = strlen(a);

int n = 1;

int i;

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

if (a[i] == ',') ++n;

}

mask = (int \*)calloc(n, sizeof(int));

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

int val = atoi(a);

mask[i] = val;

a = strchr(a, ',') + 1;

}

\*num = n;

}

return mask;

}

layer parse\_gaussian\_yolo(list \*options, size\_params params) // Gaussian\_YOLOv3

{

int classes = option\_find\_int(options, "classes", 20);

int max\_boxes = option\_find\_int\_quiet(options, "max", 90);

int total = option\_find\_int(options, "num", 1);

int num = total;

char \*a = option\_find\_str(options, "mask", 0);

int \*mask = parse\_gaussian\_yolo\_mask(a, &num);

layer l = make\_gaussian\_yolo\_layer(params.batch, params.w, params.h, num, total, mask, classes, max\_boxes);

if (l.outputs != params.inputs) {

printf("Error: l.outputs == params.inputs \n");

printf("filters= in the [convolutional]-layer doesn't correspond to classes= or mask= in [Gaussian\_yolo]-layer \n");

exit(EXIT\_FAILURE);

}

//assert(l.outputs == params.inputs);

char \*cpc = option\_find\_str(options, "counters\_per\_class", 0);

l.classes\_multipliers = get\_classes\_multipliers(cpc, classes);

l.label\_smooth\_eps = option\_find\_float\_quiet(options, "label\_smooth\_eps", 0.0f);

l.scale\_x\_y = option\_find\_float\_quiet(options, "scale\_x\_y", 1);

l.max\_delta = option\_find\_float\_quiet(options, "max\_delta", FLT\_MAX); // set 10

l.uc\_normalizer = option\_find\_float\_quiet(options, "uc\_normalizer", 1.0);

l.iou\_normalizer = option\_find\_float\_quiet(options, "iou\_normalizer", 0.75);

l.cls\_normalizer = option\_find\_float\_quiet(options, "cls\_normalizer", 1.0);

char \*iou\_loss = option\_find\_str\_quiet(options, "iou\_loss", "mse"); // "iou");

if (strcmp(iou\_loss, "mse") == 0) l.iou\_loss = MSE;

else if (strcmp(iou\_loss, "giou") == 0) l.iou\_loss = GIOU;

else if (strcmp(iou\_loss, "diou") == 0) l.iou\_loss = DIOU;

else if (strcmp(iou\_loss, "ciou") == 0) l.iou\_loss = CIOU;

else l.iou\_loss = IOU;

char \*iou\_thresh\_kind\_str = option\_find\_str\_quiet(options, "iou\_thresh\_kind", "iou");

if (strcmp(iou\_thresh\_kind\_str, "iou") == 0) l.iou\_thresh\_kind = IOU;

else if (strcmp(iou\_thresh\_kind\_str, "giou") == 0) l.iou\_thresh\_kind = GIOU;

else if (strcmp(iou\_thresh\_kind\_str, "diou") == 0) l.iou\_thresh\_kind = DIOU;

else if (strcmp(iou\_thresh\_kind\_str, "ciou") == 0) l.iou\_thresh\_kind = CIOU;

else {

fprintf(stderr, " Wrong iou\_thresh\_kind = %s \n", iou\_thresh\_kind\_str);

l.iou\_thresh\_kind = IOU;

}

l.beta\_nms = option\_find\_float\_quiet(options, "beta\_nms", 0.6);

char \*nms\_kind = option\_find\_str\_quiet(options, "nms\_kind", "default");

if (strcmp(nms\_kind, "default") == 0) l.nms\_kind = DEFAULT\_NMS;

else {

if (strcmp(nms\_kind, "greedynms") == 0) l.nms\_kind = GREEDY\_NMS;

else if (strcmp(nms\_kind, "diounms") == 0) l.nms\_kind = DIOU\_NMS;

else if (strcmp(nms\_kind, "cornersnms") == 0) l.nms\_kind = CORNERS\_NMS;

else l.nms\_kind = DEFAULT\_NMS;

printf("nms\_kind: %s (%d), beta = %f \n", nms\_kind, l.nms\_kind, l.beta\_nms);

}

char \*yolo\_point = option\_find\_str\_quiet(options, "yolo\_point", "center");

if (strcmp(yolo\_point, "left\_top") == 0) l.yolo\_point = YOLO\_LEFT\_TOP;

else if (strcmp(yolo\_point, "right\_bottom") == 0) l.yolo\_point = YOLO\_RIGHT\_BOTTOM;

else l.yolo\_point = YOLO\_CENTER;

fprintf(stderr, "[Gaussian\_yolo] iou loss: %s (%d), iou\_norm: %2.2f, cls\_norm: %2.2f, scale: %2.2f, point: %d\n",

iou\_loss, l.iou\_loss, l.iou\_normalizer, l.cls\_normalizer, l.scale\_x\_y, l.yolo\_point);

l.jitter = option\_find\_float(options, "jitter", .2);

l.ignore\_thresh = option\_find\_float(options, "ignore\_thresh", .5);

l.truth\_thresh = option\_find\_float(options, "truth\_thresh", 1);

l.iou\_thresh = option\_find\_float\_quiet(options, "iou\_thresh", 1); // recommended to use iou\_thresh=0.213 in [yolo]

l.random = option\_find\_float\_quiet(options, "random", 0);

char \*map\_file = option\_find\_str(options, "map", 0);

if (map\_file) l.map = read\_map(map\_file);

a = option\_find\_str(options, "anchors", 0);

if (a) {

int len = strlen(a);

int n = 1;

int i;

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

if (a[i] == ',') ++n;

}

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

float bias = atof(a);

l.biases[i] = bias;

a = strchr(a, ',') + 1;

}

}

return l;

}

layer parse\_region(list \*options, size\_params params)

{

int coords = option\_find\_int(options, "coords", 4);

int classes = option\_find\_int(options, "classes", 20);

int num = option\_find\_int(options, "num", 1);

int max\_boxes = option\_find\_int\_quiet(options, "max", 90);

layer l = make\_region\_layer(params.batch, params.w, params.h, num, classes, coords, max\_boxes);

if (l.outputs != params.inputs) {

printf("Error: l.outputs == params.inputs \n");

printf("filters= in the [convolutional]-layer doesn't correspond to classes= or num= in [region]-layer \n");

exit(EXIT\_FAILURE);

}

//assert(l.outputs == params.inputs);

l.log = option\_find\_int\_quiet(options, "log", 0);

l.sqrt = option\_find\_int\_quiet(options, "sqrt", 0);

l.softmax = option\_find\_int(options, "softmax", 0);

l.focal\_loss = option\_find\_int\_quiet(options, "focal\_loss", 0);

//l.max\_boxes = option\_find\_int\_quiet(options, "max",30);

l.jitter = option\_find\_float(options, "jitter", .2);

l.rescore = option\_find\_int\_quiet(options, "rescore",0);

l.thresh = option\_find\_float(options, "thresh", .5);

l.classfix = option\_find\_int\_quiet(options, "classfix", 0);

l.absolute = option\_find\_int\_quiet(options, "absolute", 0);

l.random = option\_find\_float\_quiet(options, "random", 0);

l.coord\_scale = option\_find\_float(options, "coord\_scale", 1);

l.object\_scale = option\_find\_float(options, "object\_scale", 1);

l.noobject\_scale = option\_find\_float(options, "noobject\_scale", 1);

l.mask\_scale = option\_find\_float(options, "mask\_scale", 1);

l.class\_scale = option\_find\_float(options, "class\_scale", 1);

l.bias\_match = option\_find\_int\_quiet(options, "bias\_match",0);

char \*tree\_file = option\_find\_str(options, "tree", 0);

if (tree\_file) l.softmax\_tree = read\_tree(tree\_file);

char \*map\_file = option\_find\_str(options, "map", 0);

if (map\_file) l.map = read\_map(map\_file);

char \*a = option\_find\_str(options, "anchors", 0);

if(a){

int len = strlen(a);

int n = 1;

int i;

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

if (a[i] == ',') ++n;

}

for(i = 0; i < n && i < num\*2; ++i){

float bias = atof(a);

l.biases[i] = bias;

a = strchr(a, ',')+1;

}

}

return l;

}

detection\_layer parse\_detection(list \*options, size\_params params)

{

int coords = option\_find\_int(options, "coords", 1);

int classes = option\_find\_int(options, "classes", 1);

int rescore = option\_find\_int(options, "rescore", 0);

int num = option\_find\_int(options, "num", 1);

int side = option\_find\_int(options, "side", 7);

detection\_layer layer = make\_detection\_layer(params.batch, params.inputs, num, side, classes, coords, rescore);

layer.softmax = option\_find\_int(options, "softmax", 0);

layer.sqrt = option\_find\_int(options, "sqrt", 0);

layer.max\_boxes = option\_find\_int\_quiet(options, "max",30);

layer.coord\_scale = option\_find\_float(options, "coord\_scale", 1);

layer.forced = option\_find\_int(options, "forced", 0);

layer.object\_scale = option\_find\_float(options, "object\_scale", 1);

layer.noobject\_scale = option\_find\_float(options, "noobject\_scale", 1);

layer.class\_scale = option\_find\_float(options, "class\_scale", 1);

layer.jitter = option\_find\_float(options, "jitter", .2);

layer.random = option\_find\_float\_quiet(options, "random", 0);

layer.reorg = option\_find\_int\_quiet(options, "reorg", 0);

return layer;

}

cost\_layer parse\_cost(list \*options, size\_params params)

{

char \*type\_s = option\_find\_str(options, "type", "sse");

COST\_TYPE type = get\_cost\_type(type\_s);

float scale = option\_find\_float\_quiet(options, "scale",1);

cost\_layer layer = make\_cost\_layer(params.batch, params.inputs, type, scale);

layer.ratio = option\_find\_float\_quiet(options, "ratio",0);

return layer;

}

crop\_layer parse\_crop(list \*options, size\_params params)

{

int crop\_height = option\_find\_int(options, "crop\_height",1);

int crop\_width = option\_find\_int(options, "crop\_width",1);

int flip = option\_find\_int(options, "flip",0);

float angle = option\_find\_float(options, "angle",0);

float saturation = option\_find\_float(options, "saturation",1);

float exposure = option\_find\_float(options, "exposure",1);

int batch,h,w,c;

h = params.h;

w = params.w;

c = params.c;

batch=params.batch;

if(!(h && w && c)) error("Layer before crop layer must output image.");

int noadjust = option\_find\_int\_quiet(options, "noadjust",0);

crop\_layer l = make\_crop\_layer(batch,h,w,c,crop\_height,crop\_width,flip, angle, saturation, exposure);

l.shift = option\_find\_float(options, "shift", 0);

l.noadjust = noadjust;

return l;

}

layer parse\_reorg(list \*options, size\_params params)

{

int stride = option\_find\_int(options, "stride",1);

int reverse = option\_find\_int\_quiet(options, "reverse",0);

int batch,h,w,c;

h = params.h;

w = params.w;

c = params.c;

batch=params.batch;

if(!(h && w && c)) error("Layer before reorg layer must output image.");

layer layer = make\_reorg\_layer(batch,w,h,c,stride,reverse);

return layer;

}

layer parse\_reorg\_old(list \*options, size\_params params)

{

printf("\n reorg\_old \n");

int stride = option\_find\_int(options, "stride", 1);

int reverse = option\_find\_int\_quiet(options, "reverse", 0);

int batch, h, w, c;

h = params.h;

w = params.w;

c = params.c;

batch = params.batch;

if (!(h && w && c)) error("Layer before reorg layer must output image.");

layer layer = make\_reorg\_old\_layer(batch, w, h, c, stride, reverse);

return layer;

}

maxpool\_layer parse\_local\_avgpool(list \*options, size\_params params)

{

int stride = option\_find\_int(options, "stride", 1);

int stride\_x = option\_find\_int\_quiet(options, "stride\_x", stride);

int stride\_y = option\_find\_int\_quiet(options, "stride\_y", stride);

int size = option\_find\_int(options, "size", stride);

int padding = option\_find\_int\_quiet(options, "padding", size - 1);

int maxpool\_depth = 0;

int out\_channels = 1;

int antialiasing = 0;

const int avgpool = 1;

int batch, h, w, c;

h = params.h;

w = params.w;

c = params.c;

batch = params.batch;

if (!(h && w && c)) error("Layer before [local\_avgpool] layer must output image.");

maxpool\_layer layer = make\_maxpool\_layer(batch, h, w, c, size, stride\_x, stride\_y, padding, maxpool\_depth, out\_channels, antialiasing, avgpool, params.train);

return layer;

}

maxpool\_layer parse\_maxpool(list \*options, size\_params params)

{

int stride = option\_find\_int(options, "stride",1);

int stride\_x = option\_find\_int\_quiet(options, "stride\_x", stride);

int stride\_y = option\_find\_int\_quiet(options, "stride\_y", stride);

int size = option\_find\_int(options, "size",stride);

int padding = option\_find\_int\_quiet(options, "padding", size-1);

int maxpool\_depth = option\_find\_int\_quiet(options, "maxpool\_depth", 0);

int out\_channels = option\_find\_int\_quiet(options, "out\_channels", 1);

int antialiasing = option\_find\_int\_quiet(options, "antialiasing", 0);

const int avgpool = 0;

int batch,h,w,c;

h = params.h;

w = params.w;

c = params.c;

batch=params.batch;

if(!(h && w && c)) error("Layer before [maxpool] layer must output image.");

maxpool\_layer layer = make\_maxpool\_layer(batch, h, w, c, size, stride\_x, stride\_y, padding, maxpool\_depth, out\_channels, antialiasing, avgpool, params.train);

return layer;

}

avgpool\_layer parse\_avgpool(list \*options, size\_params params)

{

int batch,w,h,c;

w = params.w;

h = params.h;

c = params.c;

batch=params.batch;

if(!(h && w && c)) error("Layer before avgpool layer must output image.");

avgpool\_layer layer = make\_avgpool\_layer(batch,w,h,c);

return layer;

}

dropout\_layer parse\_dropout(list \*options, size\_params params)

{

float probability = option\_find\_float(options, "probability", .2);

int dropblock = option\_find\_int\_quiet(options, "dropblock", 0);

float dropblock\_size\_rel = option\_find\_float\_quiet(options, "dropblock\_size\_rel", 0);

int dropblock\_size\_abs = option\_find\_float\_quiet(options, "dropblock\_size\_abs", 0);

if (dropblock\_size\_abs > params.w || dropblock\_size\_abs > params.h) {

printf(" [dropout] - dropblock\_size\_abs = %d that is bigger than layer size %d x %d \n", dropblock\_size\_abs, params.w, params.h);

dropblock\_size\_abs = min\_val\_cmp(params.w, params.h);

}

if (dropblock && !dropblock\_size\_rel && !dropblock\_size\_abs) {

printf(" [dropout] - None of the parameters (dropblock\_size\_rel or dropblock\_size\_abs) are set, will be used: dropblock\_size\_abs = 7 \n");

dropblock\_size\_abs = 7;

}

if (dropblock\_size\_rel && dropblock\_size\_abs) {

printf(" [dropout] - Both parameters are set, only the parameter will be used: dropblock\_size\_abs = %d \n", dropblock\_size\_abs);

dropblock\_size\_rel = 0;

}

dropout\_layer layer = make\_dropout\_layer(params.batch, params.inputs, probability, dropblock, dropblock\_size\_rel, dropblock\_size\_abs, params.w, params.h, params.c);

layer.out\_w = params.w;

layer.out\_h = params.h;

layer.out\_c = params.c;

return layer;

}

layer parse\_normalization(list \*options, size\_params params)

{

float alpha = option\_find\_float(options, "alpha", .0001);

float beta = option\_find\_float(options, "beta" , .75);

float kappa = option\_find\_float(options, "kappa", 1);

int size = option\_find\_int(options, "size", 5);

layer l = make\_normalization\_layer(params.batch, params.w, params.h, params.c, size, alpha, beta, kappa);

return l;

}

layer parse\_batchnorm(list \*options, size\_params params)

{

layer l = make\_batchnorm\_layer(params.batch, params.w, params.h, params.c, params.train);

return l;

}

layer parse\_shortcut(list \*options, size\_params params, network net)

{

char \*activation\_s = option\_find\_str(options, "activation", "linear");

ACTIVATION activation = get\_activation(activation\_s);

char \*weights\_type\_str = option\_find\_str\_quiet(options, "weights\_type", "none");

WEIGHTS\_TYPE\_T weights\_type = NO\_WEIGHTS;

if(strcmp(weights\_type\_str, "per\_feature") == 0 || strcmp(weights\_type\_str, "per\_layer") == 0) weights\_type = PER\_FEATURE;

else if (strcmp(weights\_type\_str, "per\_channel") == 0) weights\_type = PER\_CHANNEL;

else if (strcmp(weights\_type\_str, "none") != 0) {

printf("Error: Incorrect weights\_type = %s \n Use one of: none, per\_feature, per\_channel \n", weights\_type\_str);

getchar();

exit(0);

}

char \*weights\_normalizion\_str = option\_find\_str\_quiet(options, "weights\_normalizion", "none");

WEIGHTS\_NORMALIZATION\_T weights\_normalizion = NO\_NORMALIZATION;

if (strcmp(weights\_normalizion\_str, "relu") == 0 || strcmp(weights\_normalizion\_str, "avg\_relu") == 0) weights\_normalizion = RELU\_NORMALIZATION;

else if (strcmp(weights\_normalizion\_str, "softmax") == 0) weights\_normalizion = SOFTMAX\_NORMALIZATION;

else if (strcmp(weights\_type\_str, "none") != 0) {

printf("Error: Incorrect weights\_normalizion = %s \n Use one of: none, relu, softmax \n", weights\_normalizion\_str);

getchar();

exit(0);

}

char \*l = option\_find(options, "from");

int len = strlen(l);

if (!l) error("Route Layer must specify input layers: from = ...");

int n = 1;

int i;

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

if (l[i] == ',') ++n;

}

int\* layers = (int\*)calloc(n, sizeof(int));

int\* sizes = (int\*)calloc(n, sizeof(int));

float \*\*layers\_output = (float \*\*)calloc(n, sizeof(float \*));

float \*\*layers\_delta = (float \*\*)calloc(n, sizeof(float \*));

float \*\*layers\_output\_gpu = (float \*\*)calloc(n, sizeof(float \*));

float \*\*layers\_delta\_gpu = (float \*\*)calloc(n, sizeof(float \*));

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

int index = atoi(l);

l = strchr(l, ',') + 1;

if (index < 0) index = params.index + index;

layers[i] = index;

sizes[i] = params.net.layers[index].outputs;

layers\_output[i] = params.net.layers[index].output;

layers\_delta[i] = params.net.layers[index].delta;

}

#ifdef GPU

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

layers\_output\_gpu[i] = params.net.layers[layers[i]].output\_gpu;

layers\_delta\_gpu[i] = params.net.layers[layers[i]].delta\_gpu;

}

#endif// GPU

layer s = make\_shortcut\_layer(params.batch, n, layers, sizes, params.w, params.h, params.c, layers\_output, layers\_delta,

layers\_output\_gpu, layers\_delta\_gpu, weights\_type, weights\_normalizion, activation, params.train);

free(layers\_output\_gpu);

free(layers\_delta\_gpu);

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

int index = layers[i];

assert(params.w == net.layers[index].out\_w && params.h == net.layers[index].out\_h);

if (params.w != net.layers[index].out\_w || params.h != net.layers[index].out\_h || params.c != net.layers[index].out\_c)

fprintf(stderr, " (%4d x%4d x%4d) + (%4d x%4d x%4d) \n",

params.w, params.h, params.c, net.layers[index].out\_w, net.layers[index].out\_h, params.net.layers[index].out\_c);

}

return s;

}

layer parse\_scale\_channels(list \*options, size\_params params, network net)

{

char \*l = option\_find(options, "from");

int index = atoi(l);

if (index < 0) index = params.index + index;

int scale\_wh = option\_find\_int\_quiet(options, "scale\_wh", 0);

int batch = params.batch;

layer from = net.layers[index];

layer s = make\_scale\_channels\_layer(batch, index, params.w, params.h, params.c, from.out\_w, from.out\_h, from.out\_c, scale\_wh);

char \*activation\_s = option\_find\_str\_quiet(options, "activation", "linear");

ACTIVATION activation = get\_activation(activation\_s);

s.activation = activation;

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

printf(" [scale\_channels] layer doesn't support SWISH or MISH activations \n");

}

return s;

}

layer parse\_sam(list \*options, size\_params params, network net)

{

char \*l = option\_find(options, "from");

int index = atoi(l);

if (index < 0) index = params.index + index;

int batch = params.batch;

layer from = net.layers[index];

layer s = make\_sam\_layer(batch, index, params.w, params.h, params.c, from.out\_w, from.out\_h, from.out\_c);

char \*activation\_s = option\_find\_str\_quiet(options, "activation", "linear");

ACTIVATION activation = get\_activation(activation\_s);

s.activation = activation;

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

printf(" [sam] layer doesn't support SWISH or MISH activations \n");

}

return s;

}

layer parse\_activation(list \*options, size\_params params)

{

char \*activation\_s = option\_find\_str(options, "activation", "linear");

ACTIVATION activation = get\_activation(activation\_s);

layer l = make\_activation\_layer(params.batch, params.inputs, activation);

l.out\_h = params.h;

l.out\_w = params.w;

l.out\_c = params.c;

l.h = params.h;

l.w = params.w;

l.c = params.c;

return l;

}

layer parse\_upsample(list \*options, size\_params params, network net)

{

int stride = option\_find\_int(options, "stride", 2);

layer l = make\_upsample\_layer(params.batch, params.w, params.h, params.c, stride);

l.scale = option\_find\_float\_quiet(options, "scale", 1);

return l;

}

route\_layer parse\_route(list \*options, size\_params params)

{

char \*l = option\_find(options, "layers");

if(!l) error("Route Layer must specify input layers");

int len = strlen(l);

int n = 1;

int i;

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

if (l[i] == ',') ++n;

}

int\* layers = (int\*)xcalloc(n, sizeof(int));

int\* sizes = (int\*)xcalloc(n, sizeof(int));

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

int index = atoi(l);

l = strchr(l, ',')+1;

if(index < 0) index = params.index + index;

layers[i] = index;

sizes[i] = params.net.layers[index].outputs;

}

int batch = params.batch;

int groups = option\_find\_int\_quiet(options, "groups", 1);

int group\_id = option\_find\_int\_quiet(options, "group\_id", 0);

route\_layer layer = make\_route\_layer(batch, n, layers, sizes, groups, group\_id);

convolutional\_layer first = params.net.layers[layers[0]];

layer.out\_w = first.out\_w;

layer.out\_h = first.out\_h;

layer.out\_c = first.out\_c;

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

int index = layers[i];

convolutional\_layer next = params.net.layers[index];

if(next.out\_w == first.out\_w && next.out\_h == first.out\_h){

layer.out\_c += next.out\_c;

}else{

fprintf(stderr, " The width and height of the input layers are different. \n");

layer.out\_h = layer.out\_w = layer.out\_c = 0;

}

}

layer.out\_c = layer.out\_c / layer.groups;

layer.w = first.w;

layer.h = first.h;

layer.c = layer.out\_c;

if (n > 3) fprintf(stderr, " \t ");

else if (n > 1) fprintf(stderr, " \t ");

else fprintf(stderr, " \t\t ");

fprintf(stderr, " ");

if (layer.groups > 1) fprintf(stderr, "%d/%d", layer.group\_id, layer.groups);

else fprintf(stderr, " ");

fprintf(stderr, " -> %4d x%4d x%4d \n", layer.out\_w, layer.out\_h, layer.out\_c);

return layer;

}

learning\_rate\_policy get\_policy(char \*s)

{

if (strcmp(s, "random")==0) return RANDOM;

if (strcmp(s, "poly")==0) return POLY;

if (strcmp(s, "constant")==0) return CONSTANT;

if (strcmp(s, "step")==0) return STEP;

if (strcmp(s, "exp")==0) return EXP;

if (strcmp(s, "sigmoid")==0) return SIG;

if (strcmp(s, "steps")==0) return STEPS;

if (strcmp(s, "sgdr")==0) return SGDR;

fprintf(stderr, "Couldn't find policy %s, going with constant\n", s);

return CONSTANT;

}

void parse\_net\_options(list \*options, network \*net)

{

net->max\_batches = option\_find\_int(options, "max\_batches", 0);

net->batch = option\_find\_int(options, "batch",1);

net->learning\_rate = option\_find\_float(options, "learning\_rate", .001);

net->learning\_rate\_min = option\_find\_float\_quiet(options, "learning\_rate\_min", .00001);

net->batches\_per\_cycle = option\_find\_int\_quiet(options, "sgdr\_cycle", net->max\_batches);

net->batches\_cycle\_mult = option\_find\_int\_quiet(options, "sgdr\_mult", 2);

net->momentum = option\_find\_float(options, "momentum", .9);

net->decay = option\_find\_float(options, "decay", .0001);

int subdivs = option\_find\_int(options, "subdivisions",1);

net->time\_steps = option\_find\_int\_quiet(options, "time\_steps",1);

net->track = option\_find\_int\_quiet(options, "track", 0);

net->augment\_speed = option\_find\_int\_quiet(options, "augment\_speed", 2);

net->init\_sequential\_subdivisions = net->sequential\_subdivisions = option\_find\_int\_quiet(options, "sequential\_subdivisions", subdivs);

if (net->sequential\_subdivisions > subdivs) net->init\_sequential\_subdivisions = net->sequential\_subdivisions = subdivs;

net->try\_fix\_nan = option\_find\_int\_quiet(options, "try\_fix\_nan", 0);

net->batch /= subdivs;

net->batch \*= net->time\_steps;

net->subdivisions = subdivs;

\*net->seen = 0;

\*net->cur\_iteration = 0;

net->loss\_scale = option\_find\_float\_quiet(options, "loss\_scale", 1);

net->dynamic\_minibatch = option\_find\_int\_quiet(options, "dynamic\_minibatch", 0);

net->optimized\_memory = option\_find\_int\_quiet(options, "optimized\_memory", 0);

net->workspace\_size\_limit = (size\_t)1024\*1024 \* option\_find\_float\_quiet(options, "workspace\_size\_limit\_MB", 1024); // 1024 MB by default

net->adam = option\_find\_int\_quiet(options, "adam", 0);

if(net->adam){

net->B1 = option\_find\_float(options, "B1", .9);

net->B2 = option\_find\_float(options, "B2", .999);

net->eps = option\_find\_float(options, "eps", .000001);

}

net->h = option\_find\_int\_quiet(options, "height",0);

net->w = option\_find\_int\_quiet(options, "width",0);

net->c = option\_find\_int\_quiet(options, "channels",0);

net->inputs = option\_find\_int\_quiet(options, "inputs", net->h \* net->w \* net->c);

net->max\_crop = option\_find\_int\_quiet(options, "max\_crop",net->w\*2);

net->min\_crop = option\_find\_int\_quiet(options, "min\_crop",net->w);

net->flip = option\_find\_int\_quiet(options, "flip", 1);

net->blur = option\_find\_int\_quiet(options, "blur", 0);

net->gaussian\_noise = option\_find\_int\_quiet(options, "gaussian\_noise", 0);

net->mixup = option\_find\_int\_quiet(options, "mixup", 0);

int cutmix = option\_find\_int\_quiet(options, "cutmix", 0);

int mosaic = option\_find\_int\_quiet(options, "mosaic", 0);

if (mosaic && cutmix) net->mixup = 4;

else if (cutmix) net->mixup = 2;

else if (mosaic) net->mixup = 3;

net->letter\_box = option\_find\_int\_quiet(options, "letter\_box", 0);

net->label\_smooth\_eps = option\_find\_float\_quiet(options, "label\_smooth\_eps", 0.0f);

net->resize\_step = option\_find\_float\_quiet(options, "resize\_step", 32);

net->attention = option\_find\_int\_quiet(options, "attention", 0);

net->adversarial\_lr = option\_find\_float\_quiet(options, "adversarial\_lr", 0);

net->angle = option\_find\_float\_quiet(options, "angle", 0);

net->aspect = option\_find\_float\_quiet(options, "aspect", 1);

net->saturation = option\_find\_float\_quiet(options, "saturation", 1);

net->exposure = option\_find\_float\_quiet(options, "exposure", 1);

net->hue = option\_find\_float\_quiet(options, "hue", 0);

net->power = option\_find\_float\_quiet(options, "power", 4);

if(!net->inputs && !(net->h && net->w && net->c)) error("No input parameters supplied");

char \*policy\_s = option\_find\_str(options, "policy", "constant");

net->policy = get\_policy(policy\_s);

net->burn\_in = option\_find\_int\_quiet(options, "burn\_in", 0);

#ifdef GPU

if (net->gpu\_index >= 0) {

int compute\_capability = get\_gpu\_compute\_capability(net->gpu\_index);

#ifdef CUDNN\_HALF

if (compute\_capability >= 700) net->cudnn\_half = 1;

else net->cudnn\_half = 0;

#endif// CUDNN\_HALF

fprintf(stderr, " compute\_capability = %d, cudnn\_half = %d \n", compute\_capability, net->cudnn\_half);

}

else fprintf(stderr, " GPU isn't used \n");

#endif// GPU

if(net->policy == STEP){

net->step = option\_find\_int(options, "step", 1);

net->scale = option\_find\_float(options, "scale", 1);

} else if (net->policy == STEPS || net->policy == SGDR){

char \*l = option\_find(options, "steps");

char \*p = option\_find(options, "scales");

char \*s = option\_find(options, "seq\_scales");

if(net->policy == STEPS && (!l || !p)) error("STEPS policy must have steps and scales in cfg file");

if (l) {

int len = strlen(l);

int n = 1;

int i;

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

if (l[i] == ',') ++n;

}

int\* steps = (int\*)xcalloc(n, sizeof(int));

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

float\* seq\_scales = (float\*)xcalloc(n, sizeof(float));

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

float scale = 1.0;

if (p) {

scale = atof(p);

p = strchr(p, ',') + 1;

}

float sequence\_scale = 1.0;

if (s) {

sequence\_scale = atof(s);

s = strchr(s, ',') + 1;

}

int step = atoi(l);

l = strchr(l, ',') + 1;

steps[i] = step;

scales[i] = scale;

seq\_scales[i] = sequence\_scale;

}

net->scales = scales;

net->steps = steps;

net->seq\_scales = seq\_scales;

net->num\_steps = n;

}

} else if (net->policy == EXP){

net->gamma = option\_find\_float(options, "gamma", 1);

} else if (net->policy == SIG){

net->gamma = option\_find\_float(options, "gamma", 1);

net->step = option\_find\_int(options, "step", 1);

} else if (net->policy == POLY || net->policy == RANDOM){

//net->power = option\_find\_float(options, "power", 1);

}

}

int is\_network(section \*s)

{

return (strcmp(s->type, "[net]")==0

|| strcmp(s->type, "[network]")==0);

}

void set\_train\_only\_bn(network net)

{

int train\_only\_bn = 0;

int i;

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

if (net.layers[i].train\_only\_bn) train\_only\_bn = net.layers[i].train\_only\_bn; // set l.train\_only\_bn for all previous layers

if (train\_only\_bn) {

net.layers[i].train\_only\_bn = train\_only\_bn;

if (net.layers[i].type == CONV\_LSTM) {

net.layers[i].wf->train\_only\_bn = train\_only\_bn;

net.layers[i].wi->train\_only\_bn = train\_only\_bn;

net.layers[i].wg->train\_only\_bn = train\_only\_bn;

net.layers[i].wo->train\_only\_bn = train\_only\_bn;

net.layers[i].uf->train\_only\_bn = train\_only\_bn;

net.layers[i].ui->train\_only\_bn = train\_only\_bn;

net.layers[i].ug->train\_only\_bn = train\_only\_bn;

net.layers[i].uo->train\_only\_bn = train\_only\_bn;

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

net.layers[i].vf->train\_only\_bn = train\_only\_bn;

net.layers[i].vi->train\_only\_bn = train\_only\_bn;

net.layers[i].vo->train\_only\_bn = train\_only\_bn;

}

}

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

net.layers[i].input\_layer->train\_only\_bn = train\_only\_bn;

net.layers[i].self\_layer->train\_only\_bn = train\_only\_bn;

net.layers[i].output\_layer->train\_only\_bn = train\_only\_bn;

}

}

}

}

network parse\_network\_cfg(char \*filename)

{

return parse\_network\_cfg\_custom(filename, 0, 0);

}

network parse\_network\_cfg\_custom(char \*filename, int batch, int time\_steps)

{

list \*sections = read\_cfg(filename);

node \*n = sections->front;

if(!n) error("Config file has no sections");

network net = make\_network(sections->size - 1);

net.gpu\_index = gpu\_index;

size\_params params;

if (batch > 0) params.train = 0; // allocates memory for Detection only

else params.train = 1; // allocates memory for Detection & Training

section \*s = (section \*)n->val;

list \*options = s->options;

if(!is\_network(s)) error("First section must be [net] or [network]");

parse\_net\_options(options, &net);

#ifdef GPU

printf("net.optimized\_memory = %d \n", net.optimized\_memory);

if (net.optimized\_memory >= 2 && params.train) {

pre\_allocate\_pinned\_memory((size\_t)1024 \* 1024 \* 1024 \* 8); // pre-allocate 8 GB CPU-RAM for pinned memory

}

#endif // GPU

params.h = net.h;

params.w = net.w;

params.c = net.c;

params.inputs = net.inputs;

if (batch > 0) net.batch = batch;

if (time\_steps > 0) net.time\_steps = time\_steps;

if (net.batch < 1) net.batch = 1;

if (net.time\_steps < 1) net.time\_steps = 1;

if (net.batch < net.time\_steps) net.batch = net.time\_steps;

params.batch = net.batch;

params.time\_steps = net.time\_steps;

params.net = net;

printf("mini\_batch = %d, batch = %d, time\_steps = %d, train = %d \n", net.batch, net.batch \* net.subdivisions, net.time\_steps, params.train);

int avg\_outputs = 0;

int avg\_counter = 0;

float bflops = 0;

size\_t workspace\_size = 0;

size\_t max\_inputs = 0;

size\_t max\_outputs = 0;

int receptive\_w = 1, receptive\_h = 1;

int receptive\_w\_scale = 1, receptive\_h\_scale = 1;

const int show\_receptive\_field = option\_find\_float\_quiet(options, "show\_receptive\_field", 0);

n = n->next;

int count = 0;

free\_section(s);

fprintf(stderr, " layer filters size/strd(dil) input output\n");

while(n){

params.index = count;

fprintf(stderr, "%4d ", count);

s = (section \*)n->val;

options = s->options;

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

LAYER\_TYPE lt = string\_to\_layer\_type(s->type);

if(lt == CONVOLUTIONAL){

l = parse\_convolutional(options, params);

}else if(lt == LOCAL){

l = parse\_local(options, params);

}else if(lt == ACTIVE){

l = parse\_activation(options, params);

}else if(lt == RNN){

l = parse\_rnn(options, params);

}else if(lt == GRU){

l = parse\_gru(options, params);

}else if(lt == LSTM){

l = parse\_lstm(options, params);

}else if (lt == CONV\_LSTM) {

l = parse\_conv\_lstm(options, params);

}else if(lt == CRNN){

l = parse\_crnn(options, params);

}else if(lt == CONNECTED){

l = parse\_connected(options, params);

}else if(lt == CROP){

l = parse\_crop(options, params);

}else if(lt == COST){

l = parse\_cost(options, params);

l.keep\_delta\_gpu = 1;

}else if(lt == REGION){

l = parse\_region(options, params);

l.keep\_delta\_gpu = 1;

}else if (lt == YOLO) {

l = parse\_yolo(options, params);

l.keep\_delta\_gpu = 1;

}else if (lt == GAUSSIAN\_YOLO) {

l = parse\_gaussian\_yolo(options, params);

l.keep\_delta\_gpu = 1;

}else if(lt == DETECTION){

l = parse\_detection(options, params);

}else if(lt == SOFTMAX){

l = parse\_softmax(options, params);

net.hierarchy = l.softmax\_tree;

l.keep\_delta\_gpu = 1;

}else if(lt == NORMALIZATION){

l = parse\_normalization(options, params);

}else if(lt == BATCHNORM){

l = parse\_batchnorm(options, params);

}else if(lt == MAXPOOL){

l = parse\_maxpool(options, params);

}else if (lt == LOCAL\_AVGPOOL) {

l = parse\_local\_avgpool(options, params);

}else if(lt == REORG){

l = parse\_reorg(options, params); }

else if (lt == REORG\_OLD) {

l = parse\_reorg\_old(options, params);

}else if(lt == AVGPOOL){

l = parse\_avgpool(options, params);

}else if(lt == ROUTE){

l = parse\_route(options, params);

int k;

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

net.layers[l.input\_layers[k]].use\_bin\_output = 0;

net.layers[l.input\_layers[k]].keep\_delta\_gpu = 1;

}

}else if (lt == UPSAMPLE) {

l = parse\_upsample(options, params, net);

}else if(lt == SHORTCUT){

l = parse\_shortcut(options, params, net);

net.layers[count - 1].use\_bin\_output = 0;

net.layers[l.index].use\_bin\_output = 0;

net.layers[l.index].keep\_delta\_gpu = 1;

}else if (lt == SCALE\_CHANNELS) {

l = parse\_scale\_channels(options, params, net);

net.layers[count - 1].use\_bin\_output = 0;

net.layers[l.index].use\_bin\_output = 0;

net.layers[l.index].keep\_delta\_gpu = 1;

}

else if (lt == SAM) {

l = parse\_sam(options, params, net);

net.layers[count - 1].use\_bin\_output = 0;

net.layers[l.index].use\_bin\_output = 0;

net.layers[l.index].keep\_delta\_gpu = 1;

}else if(lt == DROPOUT){

l = parse\_dropout(options, params);

l.output = net.layers[count-1].output;

l.delta = net.layers[count-1].delta;

#ifdef GPU

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

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

l.keep\_delta\_gpu = 1;

#endif

}

else if (lt == EMPTY) {

layer empty\_layer = {(LAYER\_TYPE)0};

empty\_layer.out\_w = params.w;

empty\_layer.out\_h = params.h;

empty\_layer.out\_c = params.c;

l = empty\_layer;

l.output = net.layers[count - 1].output;

l.delta = net.layers[count - 1].delta;

#ifdef GPU

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

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

#endif

}else{

fprintf(stderr, "Type not recognized: %s\n", s->type);

}

// calculate receptive field

if(show\_receptive\_field)

{

int dilation = max\_val\_cmp(1, l.dilation);

int stride = max\_val\_cmp(1, l.stride);

int size = max\_val\_cmp(1, l.size);

if (l.type == UPSAMPLE || (l.type == REORG))

{

l.receptive\_w = receptive\_w;

l.receptive\_h = receptive\_h;

l.receptive\_w\_scale = receptive\_w\_scale = receptive\_w\_scale / stride;

l.receptive\_h\_scale = receptive\_h\_scale = receptive\_h\_scale / stride;

}

else {

if (l.type == ROUTE) {

receptive\_w = receptive\_h = receptive\_w\_scale = receptive\_h\_scale = 0;

int k;

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

layer route\_l = net.layers[l.input\_layers[k]];

receptive\_w = max\_val\_cmp(receptive\_w, route\_l.receptive\_w);

receptive\_h = max\_val\_cmp(receptive\_h, route\_l.receptive\_h);

receptive\_w\_scale = max\_val\_cmp(receptive\_w\_scale, route\_l.receptive\_w\_scale);

receptive\_h\_scale = max\_val\_cmp(receptive\_h\_scale, route\_l.receptive\_h\_scale);

}

}

else

{

int increase\_receptive = size + (dilation - 1) \* 2 - 1;// stride;

increase\_receptive = max\_val\_cmp(0, increase\_receptive);

receptive\_w += increase\_receptive \* receptive\_w\_scale;

receptive\_h += increase\_receptive \* receptive\_h\_scale;

receptive\_w\_scale \*= stride;

receptive\_h\_scale \*= stride;

}

l.receptive\_w = receptive\_w;

l.receptive\_h = receptive\_h;

l.receptive\_w\_scale = receptive\_w\_scale;

l.receptive\_h\_scale = receptive\_h\_scale;

}

//printf(" size = %d, dilation = %d, stride = %d, receptive\_w = %d, receptive\_w\_scale = %d - ", size, dilation, stride, receptive\_w, receptive\_w\_scale);

int cur\_receptive\_w = receptive\_w;

int cur\_receptive\_h = receptive\_h;

fprintf(stderr, "%4d - receptive field: %d x %d \n", count, cur\_receptive\_w, cur\_receptive\_h);

}

#ifdef GPU

// futher GPU-memory optimization: net.optimized\_memory == 2

if (net.optimized\_memory >= 2 && params.train && l.type != DROPOUT)

{

l.optimized\_memory = net.optimized\_memory;

if (l.output\_gpu) {

cuda\_free(l.output\_gpu);

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

l.output\_gpu = cuda\_make\_array\_pinned\_preallocated(NULL, l.batch\*l.outputs); // l.steps

}

if (l.activation\_input\_gpu) {

cuda\_free(l.activation\_input\_gpu);

l.activation\_input\_gpu = cuda\_make\_array\_pinned\_preallocated(NULL, l.batch\*l.outputs); // l.steps

}

if (l.x\_gpu) {

cuda\_free(l.x\_gpu);

l.x\_gpu = cuda\_make\_array\_pinned\_preallocated(NULL, l.batch\*l.outputs); // l.steps

}

// maximum optimization

if (net.optimized\_memory >= 3 && l.type != DROPOUT) {

if (l.delta\_gpu) {

cuda\_free(l.delta\_gpu);

//l.delta\_gpu = cuda\_make\_array\_pinned\_preallocated(NULL, l.batch\*l.outputs); // l.steps

//printf("\n\n PINNED DELTA GPU = %d \n", l.batch\*l.outputs);

}

}

if (l.type == CONVOLUTIONAL) {

set\_specified\_workspace\_limit(&l, net.workspace\_size\_limit); // workspace size limit 1 GB

}

}

#endif // GPU

l.clip = option\_find\_float\_quiet(options, "clip", 0);

l.dynamic\_minibatch = net.dynamic\_minibatch;

l.onlyforward = option\_find\_int\_quiet(options, "onlyforward", 0);

l.dont\_update = option\_find\_int\_quiet(options, "dont\_update", 0);

l.burnin\_update = option\_find\_int\_quiet(options, "burnin\_update", 0);

l.stopbackward = option\_find\_int\_quiet(options, "stopbackward", 0);

l.train\_only\_bn = option\_find\_int\_quiet(options, "train\_only\_bn", 0);

l.dontload = option\_find\_int\_quiet(options, "dontload", 0);

l.dontloadscales = option\_find\_int\_quiet(options, "dontloadscales", 0);

l.learning\_rate\_scale = option\_find\_float\_quiet(options, "learning\_rate", 1);

option\_unused(options);

net.layers[count] = l;

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

if (l.inputs > max\_inputs) max\_inputs = l.inputs;

if (l.outputs > max\_outputs) max\_outputs = l.outputs;

free\_section(s);

n = n->next;

++count;

if(n){

if (l.antialiasing) {

params.h = l.input\_layer->out\_h;

params.w = l.input\_layer->out\_w;

params.c = l.input\_layer->out\_c;

params.inputs = l.input\_layer->outputs;

}

else {

params.h = l.out\_h;

params.w = l.out\_w;

params.c = l.out\_c;

params.inputs = l.outputs;

}

}

if (l.bflops > 0) bflops += l.bflops;

if (l.w > 1 && l.h > 1) {

avg\_outputs += l.outputs;

avg\_counter++;

}

}

free\_list(sections);

#ifdef GPU

if (net.optimized\_memory && params.train)

{

int k;

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

layer l = net.layers[k];

// delta GPU-memory optimization: net.optimized\_memory == 1

if (!l.keep\_delta\_gpu) {

const size\_t delta\_size = l.outputs\*l.batch; // l.steps

if (net.max\_delta\_gpu\_size < delta\_size) {

net.max\_delta\_gpu\_size = delta\_size;

if (net.global\_delta\_gpu) cuda\_free(net.global\_delta\_gpu);

if (net.state\_delta\_gpu) cuda\_free(net.state\_delta\_gpu);

assert(net.max\_delta\_gpu\_size > 0);

net.global\_delta\_gpu = (float \*)cuda\_make\_array(NULL, net.max\_delta\_gpu\_size);

net.state\_delta\_gpu = (float \*)cuda\_make\_array(NULL, net.max\_delta\_gpu\_size);

}

if (l.delta\_gpu) {

if (net.optimized\_memory >= 3) {}

else cuda\_free(l.delta\_gpu);

}

l.delta\_gpu = net.global\_delta\_gpu;

}

// maximum optimization

if (net.optimized\_memory >= 3 && l.type != DROPOUT) {

if (l.delta\_gpu && l.keep\_delta\_gpu) {

//cuda\_free(l.delta\_gpu); // already called above

l.delta\_gpu = cuda\_make\_array\_pinned\_preallocated(NULL, l.batch\*l.outputs); // l.steps

//printf("\n\n PINNED DELTA GPU = %d \n", l.batch\*l.outputs);

}

}

net.layers[k] = l;

}

}

#endif

set\_train\_only\_bn(net); // set l.train\_only\_bn for all required layers

net.outputs = get\_network\_output\_size(net);

net.output = get\_network\_output(net);

avg\_outputs = avg\_outputs / avg\_counter;

fprintf(stderr, "Total BFLOPS %5.3f \n", bflops);

fprintf(stderr, "avg\_outputs = %d \n", avg\_outputs);

#ifdef GPU

get\_cuda\_stream();

get\_cuda\_memcpy\_stream();

if (gpu\_index >= 0)

{

int size = get\_network\_input\_size(net) \* net.batch;

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));

}

// pre-allocate memory for inference on Tensor Cores (fp16)

if (net.cudnn\_half) {

\*net.max\_input16\_size = max\_inputs;

CHECK\_CUDA(cudaMalloc((void \*\*)net.input16\_gpu, \*net.max\_input16\_size \* sizeof(short))); //sizeof(half)

\*net.max\_output16\_size = max\_outputs;

CHECK\_CUDA(cudaMalloc((void \*\*)net.output16\_gpu, \*net.max\_output16\_size \* sizeof(short))); //sizeof(half)

}

if (workspace\_size) {

fprintf(stderr, " Allocate additional workspace\_size = %1.2f MB \n", (float)workspace\_size/1000000);

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

}

else {

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

}

}

#else

if (workspace\_size) {

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

}

#endif

LAYER\_TYPE lt = net.layers[net.n - 1].type;

if ((net.w % 32 != 0 || net.h % 32 != 0) && (lt == YOLO || lt == REGION || lt == DETECTION)) {

printf("\n Warning: width=%d and height=%d in cfg-file must be divisible by 32 for default networks Yolo v1/v2/v3!!! \n\n",

net.w, net.h);

}

return net;

}

list \*read\_cfg(char \*filename)

{

FILE \*file = fopen(filename, "r");

if(file == 0) file\_error(filename);

char \*line;

int nu = 0;

list \*sections = make\_list();

section \*current = 0;

while((line=fgetl(file)) != 0){

++ nu;

strip(line);

switch(line[0]){

case '[':

current = (section\*)xmalloc(sizeof(section));

list\_insert(sections, current);

current->options = make\_list();

current->type = line;

break;

case '\0':

case '#':

case ';':

free(line);

break;

default:

if(!read\_option(line, current->options)){

fprintf(stderr, "Config file error line %d, could parse: %s\n", nu, line);

free(line);

}

break;

}

}

fclose(file);

return sections;

}

void save\_convolutional\_weights\_binary(layer l, FILE \*fp)

{

#ifdef GPU

if(gpu\_index >= 0){

pull\_convolutional\_layer(l);

}

#endif

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

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

int i, j, k;

fwrite(l.biases, sizeof(float), l.n, fp);

if (l.batch\_normalize){

fwrite(l.scales, sizeof(float), l.n, fp);

fwrite(l.rolling\_mean, sizeof(float), l.n, fp);

fwrite(l.rolling\_variance, sizeof(float), l.n, fp);

}

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

float mean = l.binary\_weights[i\*size];

if(mean < 0) mean = -mean;

fwrite(&mean, sizeof(float), 1, fp);

for(j = 0; j < size/8; ++j){

int index = i\*size + j\*8;

unsigned char c = 0;

for(k = 0; k < 8; ++k){

if (j\*8 + k >= size) break;

if (l.binary\_weights[index + k] > 0) c = (c | 1<<k);

}

fwrite(&c, sizeof(char), 1, fp);

}

}

}

void save\_shortcut\_weights(layer l, FILE \*fp)

{

#ifdef GPU

if (gpu\_index >= 0) {

pull\_shortcut\_layer(l);

printf("\n pull\_shortcut\_layer \n");

}

#endif

int i;

for (i = 0; i < l.nweights; ++i) printf(" %f, ", l.weight\_updates[i]);

printf(" l.nweights = %d - update \n", l.nweights);

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

printf(" l.nweights = %d \n\n", l.nweights);

int num = l.nweights;

fwrite(l.weights, sizeof(float), num, fp);

}

void save\_convolutional\_weights(layer l, FILE \*fp)

{

if(l.binary){

//save\_convolutional\_weights\_binary(l, fp);

//return;

}

#ifdef GPU

if(gpu\_index >= 0){

pull\_convolutional\_layer(l);

}

#endif

int num = l.nweights;

fwrite(l.biases, sizeof(float), l.n, fp);

if (l.batch\_normalize){

fwrite(l.scales, sizeof(float), l.n, fp);

fwrite(l.rolling\_mean, sizeof(float), l.n, fp);

fwrite(l.rolling\_variance, sizeof(float), l.n, fp);

}

fwrite(l.weights, sizeof(float), num, fp);

//if(l.adam){

// fwrite(l.m, sizeof(float), num, fp);

// fwrite(l.v, sizeof(float), num, fp);

//}

}

void save\_batchnorm\_weights(layer l, FILE \*fp)

{

#ifdef GPU

if(gpu\_index >= 0){

pull\_batchnorm\_layer(l);

}

#endif

fwrite(l.biases, sizeof(float), l.c, fp);

fwrite(l.scales, sizeof(float), l.c, fp);

fwrite(l.rolling\_mean, sizeof(float), l.c, fp);

fwrite(l.rolling\_variance, sizeof(float), l.c, fp);

}

void save\_connected\_weights(layer l, FILE \*fp)

{

#ifdef GPU

if(gpu\_index >= 0){

pull\_connected\_layer(l);

}

#endif

fwrite(l.biases, sizeof(float), l.outputs, fp);

fwrite(l.weights, sizeof(float), l.outputs\*l.inputs, fp);

if (l.batch\_normalize){

fwrite(l.scales, sizeof(float), l.outputs, fp);

fwrite(l.rolling\_mean, sizeof(float), l.outputs, fp);

fwrite(l.rolling\_variance, sizeof(float), l.outputs, fp);

}

}

void save\_weights\_upto(network net, char \*filename, int cutoff)

{

#ifdef GPU

if(net.gpu\_index >= 0){

cuda\_set\_device(net.gpu\_index);

}

#endif

fprintf(stderr, "Saving weights to %s\n", filename);

FILE \*fp = fopen(filename, "wb");

if(!fp) file\_error(filename);

int major = MAJOR\_VERSION;

int minor = MINOR\_VERSION;

int revision = PATCH\_VERSION;

fwrite(&major, sizeof(int), 1, fp);

fwrite(&minor, sizeof(int), 1, fp);

fwrite(&revision, sizeof(int), 1, fp);

(\*net.seen) = get\_current\_iteration(net) \* net.batch \* net.subdivisions; // remove this line, when you will save to weights-file both: seen & cur\_iteration

fwrite(net.seen, sizeof(uint64\_t), 1, fp);

int i;

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

layer l = net.layers[i];

if (l.type == CONVOLUTIONAL && l.share\_layer == NULL) {

save\_convolutional\_weights(l, fp);

} if (l.type == SHORTCUT && l.nweights > 0) {

save\_shortcut\_weights(l, fp);

} if(l.type == CONNECTED){

save\_connected\_weights(l, fp);

} if(l.type == BATCHNORM){

save\_batchnorm\_weights(l, fp);

} if(l.type == RNN){

save\_connected\_weights(\*(l.input\_layer), fp);

save\_connected\_weights(\*(l.self\_layer), fp);

save\_connected\_weights(\*(l.output\_layer), fp);

} if(l.type == GRU){

save\_connected\_weights(\*(l.input\_z\_layer), fp);

save\_connected\_weights(\*(l.input\_r\_layer), fp);

save\_connected\_weights(\*(l.input\_h\_layer), fp);

save\_connected\_weights(\*(l.state\_z\_layer), fp);

save\_connected\_weights(\*(l.state\_r\_layer), fp);

save\_connected\_weights(\*(l.state\_h\_layer), fp);

} if(l.type == LSTM){

save\_connected\_weights(\*(l.wf), fp);

save\_connected\_weights(\*(l.wi), fp);

save\_connected\_weights(\*(l.wg), fp);

save\_connected\_weights(\*(l.wo), fp);

save\_connected\_weights(\*(l.uf), fp);

save\_connected\_weights(\*(l.ui), fp);

save\_connected\_weights(\*(l.ug), fp);

save\_connected\_weights(\*(l.uo), fp);

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

if (l.peephole) {

save\_convolutional\_weights(\*(l.vf), fp);

save\_convolutional\_weights(\*(l.vi), fp);

save\_convolutional\_weights(\*(l.vo), fp);

}

save\_convolutional\_weights(\*(l.wf), fp);

save\_convolutional\_weights(\*(l.wi), fp);

save\_convolutional\_weights(\*(l.wg), fp);

save\_convolutional\_weights(\*(l.wo), fp);

save\_convolutional\_weights(\*(l.uf), fp);

save\_convolutional\_weights(\*(l.ui), fp);

save\_convolutional\_weights(\*(l.ug), fp);

save\_convolutional\_weights(\*(l.uo), fp);

} if(l.type == CRNN){

save\_convolutional\_weights(\*(l.input\_layer), fp);

save\_convolutional\_weights(\*(l.self\_layer), fp);

save\_convolutional\_weights(\*(l.output\_layer), fp);

} if(l.type == LOCAL){

#ifdef GPU

if(gpu\_index >= 0){

pull\_local\_layer(l);

}

#endif

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

int size = l.size\*l.size\*l.c\*l.n\*locations;

fwrite(l.biases, sizeof(float), l.outputs, fp);

fwrite(l.weights, sizeof(float), size, fp);

}

}

fclose(fp);

}

void save\_weights(network net, char \*filename)

{

save\_weights\_upto(net, filename, net.n);

}

void transpose\_matrix(float \*a, int rows, int cols)

{

float\* transpose = (float\*)xcalloc(rows \* cols, sizeof(float));

int x, y;

for(x = 0; x < rows; ++x){

for(y = 0; y < cols; ++y){

transpose[y\*rows + x] = a[x\*cols + y];

}

}

memcpy(a, transpose, rows\*cols\*sizeof(float));

free(transpose);

}

void load\_connected\_weights(layer l, FILE \*fp, int transpose)

{

fread(l.biases, sizeof(float), l.outputs, fp);

fread(l.weights, sizeof(float), l.outputs\*l.inputs, fp);

if(transpose){

transpose\_matrix(l.weights, l.inputs, l.outputs);

}

//printf("Biases: %f mean %f variance\n", mean\_array(l.biases, l.outputs), variance\_array(l.biases, l.outputs));

//printf("Weights: %f mean %f variance\n", mean\_array(l.weights, l.outputs\*l.inputs), variance\_array(l.weights, l.outputs\*l.inputs));

if (l.batch\_normalize && (!l.dontloadscales)){

fread(l.scales, sizeof(float), l.outputs, fp);

fread(l.rolling\_mean, sizeof(float), l.outputs, fp);

fread(l.rolling\_variance, sizeof(float), l.outputs, fp);

//printf("Scales: %f mean %f variance\n", mean\_array(l.scales, l.outputs), variance\_array(l.scales, l.outputs));

//printf("rolling\_mean: %f mean %f variance\n", mean\_array(l.rolling\_mean, l.outputs), variance\_array(l.rolling\_mean, l.outputs));

//printf("rolling\_variance: %f mean %f variance\n", mean\_array(l.rolling\_variance, l.outputs), variance\_array(l.rolling\_variance, l.outputs));

}

#ifdef GPU

if(gpu\_index >= 0){

push\_connected\_layer(l);

}

#endif

}

void load\_batchnorm\_weights(layer l, FILE \*fp)

{

fread(l.biases, sizeof(float), l.c, fp);

fread(l.scales, sizeof(float), l.c, fp);

fread(l.rolling\_mean, sizeof(float), l.c, fp);

fread(l.rolling\_variance, sizeof(float), l.c, fp);

#ifdef GPU

if(gpu\_index >= 0){

push\_batchnorm\_layer(l);

}

#endif

}

void load\_convolutional\_weights\_binary(layer l, FILE \*fp)

{

fread(l.biases, sizeof(float), l.n, fp);

if (l.batch\_normalize && (!l.dontloadscales)){

fread(l.scales, sizeof(float), l.n, fp);

fread(l.rolling\_mean, sizeof(float), l.n, fp);

fread(l.rolling\_variance, sizeof(float), l.n, fp);

}

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

int i, j, k;

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

float mean = 0;

fread(&mean, sizeof(float), 1, fp);

for(j = 0; j < size/8; ++j){

int index = i\*size + j\*8;

unsigned char c = 0;

fread(&c, sizeof(char), 1, fp);

for(k = 0; k < 8; ++k){

if (j\*8 + k >= size) break;

l.weights[index + k] = (c & 1<<k) ? mean : -mean;

}

}

}

#ifdef GPU

if(gpu\_index >= 0){

push\_convolutional\_layer(l);

}

#endif

}

void load\_convolutional\_weights(layer l, FILE \*fp)

{

if(l.binary){

//load\_convolutional\_weights\_binary(l, fp);

//return;

}

int num = l.nweights;

int read\_bytes;

read\_bytes = fread(l.biases, sizeof(float), l.n, fp);

if (read\_bytes > 0 && read\_bytes < l.n) printf("\n Warning: Unexpected end of wights-file! l.biases - l.index = %d \n", l.index);

//fread(l.weights, sizeof(float), num, fp); // as in connected layer

if (l.batch\_normalize && (!l.dontloadscales)){

read\_bytes = fread(l.scales, sizeof(float), l.n, fp);

if (read\_bytes > 0 && read\_bytes < l.n) printf("\n Warning: Unexpected end of wights-file! l.scales - l.index = %d \n", l.index);

read\_bytes = fread(l.rolling\_mean, sizeof(float), l.n, fp);

if (read\_bytes > 0 && read\_bytes < l.n) printf("\n Warning: Unexpected end of wights-file! l.rolling\_mean - l.index = %d \n", l.index);

read\_bytes = fread(l.rolling\_variance, sizeof(float), l.n, fp);

if (read\_bytes > 0 && read\_bytes < l.n) printf("\n Warning: Unexpected end of wights-file! l.rolling\_variance - l.index = %d \n", l.index);

if(0){

int i;

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

printf("%g, ", l.rolling\_mean[i]);

}

printf("\n");

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

printf("%g, ", l.rolling\_variance[i]);

}

printf("\n");

}

if(0){

fill\_cpu(l.n, 0, l.rolling\_mean, 1);

fill\_cpu(l.n, 0, l.rolling\_variance, 1);

}

}

read\_bytes = fread(l.weights, sizeof(float), num, fp);

if (read\_bytes > 0 && read\_bytes < l.n) printf("\n Warning: Unexpected end of wights-file! l.weights - l.index = %d \n", l.index);

//if(l.adam){

// fread(l.m, sizeof(float), num, fp);

// fread(l.v, sizeof(float), num, fp);

//}

//if(l.c == 3) scal\_cpu(num, 1./256, l.weights, 1);

if (l.flipped) {

transpose\_matrix(l.weights, (l.c/l.groups)\*l.size\*l.size, l.n);

}

//if (l.binary) binarize\_weights(l.weights, l.n, (l.c/l.groups)\*l.size\*l.size, l.weights);

#ifdef GPU

if(gpu\_index >= 0){

push\_convolutional\_layer(l);

}

#endif

}

void load\_shortcut\_weights(layer l, FILE \*fp)

{

int num = l.nweights;

int read\_bytes;

read\_bytes = fread(l.weights, sizeof(float), num, fp);

if (read\_bytes > 0 && read\_bytes < num) printf("\n Warning: Unexpected end of wights-file! l.weights - l.index = %d \n", l.index);

//for (int i = 0; i < l.nweights; ++i) printf(" %f, ", l.weights[i]);

//printf(" read\_bytes = %d \n\n", read\_bytes);

#ifdef GPU

if (gpu\_index >= 0) {

push\_shortcut\_layer(l);

}

#endif

}

void load\_weights\_upto(network \*net, char \*filename, int cutoff)

{

#ifdef GPU

if(net->gpu\_index >= 0){

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

}

#endif

fprintf(stderr, "Loading weights from %s...", filename);

fflush(stdout);

FILE \*fp = fopen(filename, "rb");

if(!fp) file\_error(filename);

int major;

int minor;

int revision;

fread(&major, sizeof(int), 1, fp);

fread(&minor, sizeof(int), 1, fp);

fread(&revision, sizeof(int), 1, fp);

if ((major \* 10 + minor) >= 2) {

printf("\n seen 64");

uint64\_t iseen = 0;

fread(&iseen, sizeof(uint64\_t), 1, fp);

\*net->seen = iseen;

}

else {

printf("\n seen 32");

uint32\_t iseen = 0;

fread(&iseen, sizeof(uint32\_t), 1, fp);

\*net->seen = iseen;

}

\*net->cur\_iteration = get\_current\_batch(\*net);

printf(", trained: %.0f K-images (%.0f Kilo-batches\_64) \n", (float)(\*net->seen / 1000), (float)(\*net->seen / 64000));

int transpose = (major > 1000) || (minor > 1000);

int i;

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

layer l = net->layers[i];

if (l.dontload) continue;

if(l.type == CONVOLUTIONAL && l.share\_layer == NULL){

load\_convolutional\_weights(l, fp);

}

if (l.type == SHORTCUT && l.nweights > 0) {

load\_shortcut\_weights(l, fp);

}

if(l.type == CONNECTED){

load\_connected\_weights(l, fp, transpose);

}

if(l.type == BATCHNORM){

load\_batchnorm\_weights(l, fp);

}

if(l.type == CRNN){

load\_convolutional\_weights(\*(l.input\_layer), fp);

load\_convolutional\_weights(\*(l.self\_layer), fp);

load\_convolutional\_weights(\*(l.output\_layer), fp);

}

if(l.type == RNN){

load\_connected\_weights(\*(l.input\_layer), fp, transpose);

load\_connected\_weights(\*(l.self\_layer), fp, transpose);

load\_connected\_weights(\*(l.output\_layer), fp, transpose);

}

if(l.type == GRU){

load\_connected\_weights(\*(l.input\_z\_layer), fp, transpose);

load\_connected\_weights(\*(l.input\_r\_layer), fp, transpose);

load\_connected\_weights(\*(l.input\_h\_layer), fp, transpose);

load\_connected\_weights(\*(l.state\_z\_layer), fp, transpose);

load\_connected\_weights(\*(l.state\_r\_layer), fp, transpose);

load\_connected\_weights(\*(l.state\_h\_layer), fp, transpose);

}

if(l.type == LSTM){

load\_connected\_weights(\*(l.wf), fp, transpose);

load\_connected\_weights(\*(l.wi), fp, transpose);

load\_connected\_weights(\*(l.wg), fp, transpose);

load\_connected\_weights(\*(l.wo), fp, transpose);

load\_connected\_weights(\*(l.uf), fp, transpose);

load\_connected\_weights(\*(l.ui), fp, transpose);

load\_connected\_weights(\*(l.ug), fp, transpose);

load\_connected\_weights(\*(l.uo), fp, transpose);

}

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

if (l.peephole) {

load\_convolutional\_weights(\*(l.vf), fp);

load\_convolutional\_weights(\*(l.vi), fp);

load\_convolutional\_weights(\*(l.vo), fp);

}

load\_convolutional\_weights(\*(l.wf), fp);

load\_convolutional\_weights(\*(l.wi), fp);

load\_convolutional\_weights(\*(l.wg), fp);

load\_convolutional\_weights(\*(l.wo), fp);

load\_convolutional\_weights(\*(l.uf), fp);

load\_convolutional\_weights(\*(l.ui), fp);

load\_convolutional\_weights(\*(l.ug), fp);

load\_convolutional\_weights(\*(l.uo), fp);

}

if(l.type == LOCAL){

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

int size = l.size\*l.size\*l.c\*l.n\*locations;

fread(l.biases, sizeof(float), l.outputs, fp);

fread(l.weights, sizeof(float), size, fp);

#ifdef GPU

if(gpu\_index >= 0){

push\_local\_layer(l);

}

#endif

}

if (feof(fp)) break;

}

fprintf(stderr, "Done! Loaded %d layers from weights-file \n", i);

fclose(fp);

}

void load\_weights(network \*net, char \*filename)

{

load\_weights\_upto(net, filename, net->n);

}

// load network & force - set batch size

network \*load\_network\_custom(char \*cfg, char \*weights, int clear, int batch)

{

printf(" Try to load cfg: %s, weights: %s, clear = %d \n", cfg, weights, clear);

network\* net = (network\*)xcalloc(1, sizeof(network));

\*net = parse\_network\_cfg\_custom(cfg, batch, 1);

if (weights && weights[0] != 0) {

printf(" Try to load weights: %s \n", weights);

load\_weights(net, weights);

}

fuse\_conv\_batchnorm(\*net);

if (clear) {

(\*net->seen) = 0;

(\*net->cur\_iteration) = 0;

}

return net;

}

// load network & get batch size from cfg-file

network \*load\_network(char \*cfg, char \*weights, int clear)

{

printf(" Try to load cfg: %s, clear = %d \n", cfg, clear);

network\* net = (network\*)xcalloc(1, sizeof(network));

\*net = parse\_network\_cfg(cfg);

if (weights && weights[0] != 0) {

printf(" Try to load weights: %s \n", weights);

load\_weights(net, weights);

}

if (clear) {

(\*net->seen) = 0;

(\*net->cur\_iteration) = 0;

}

return net;

}