```
void forward_region_layer(const layer l, network net)
```

```
266     for(t = 0; t < 30; ++t){
267         box truth = float_to_box(net.truth + t*(l.coords + 1) + b*l.truths, 1); // get the t-th truth box
268
```

The truth box layout :

Truth box 0
Truth box 1
Truth box 2
Truth box 3
Truth box 29

Each box contains 4 coordinates and 1 class ID

b*l.truths : b is batch size, l.truths is the truth length of one batch
net.truth 是基地址，
問題：truth text 在代碼中是怎麼傳入的?
如果是我會怎麼設計 truth 的排列?
t<30,爲何 30? 一幅訓練圖片的 bbox 固定 30 個，不夠的充 0.
在計算代價函數的時候，根據 0-29 個 bbox_truth，依據每個 bbox_truth 的坐標找到相應的 grid cell，計算 tx,ty,bw,bh,
classID 是由 softmax 得出的，在做反向傳遞的時候，相應 ID truth 爲 1,其他爲 0。
在最後一層的反向傳遞中，delta 函數必須包含代價函數 J 對所以 bbox 參數的偏倒數。
即:

$\frac{\partial J}{\partial w_i}$ $\frac{\partial J}{\partial h_i}$ 對 bbox 大小的偏倒數，i = 0： S×S×B， 所以其維度爲 1× S*S*B，其中只算其有 object 的部分，其他不含 object 的 index 位置偏倒數爲 0.

```
if(!truth.x) break;  // if truth.x = 0 , this class does not exitst in this image
float best_iou = 0;
int best_n = 0;
i = (truth.x * l.w);  // the actual truth x center in pixels
j = (truth.y * l.h);  //
box truth_shift = truth;
truth_shift.x = 0;    // why shift ?
truth_shift.y = 0;    // setting x,y to 0 equals to shifting ? why?
for(n = 0; n < l.n; ++n){  // l.n=5 the number of bounding boxes in one grid cell
    int box_index = entry_index(l, b, n*l.w*l.h + j*l.w + i, 0); // get the 5 predicted boxes in the grid cell where
                                                                 // the ground truth center is located;
                                                                 // need to know the pred box layout
    box pred = get_region_box(l.output, l.biases, n, box_index, i, j, l.w, l.h, l.w*l.h);  // 5 predicted boxes in the ground truth's location
    if(l.bias_match){                                            // what is bias_match ?
        pred.w = l.biases[2*n]/l.w;
        pred.h = l.biases[2*n+1]/l.h;
    }
    pred.x = 0;
    pred.y = 0;
    float iou = box_iou(pred, truth_shift);
    if (iou > best_iou){
        best_iou = iou;
        best_n = n;          // choose the bbox with best iou with the ground truth
    }
}
```

注意下標：j 是 bbox 的 index，不是 bacth index。i 是 gridcell 的 index。

$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$

lost function for box size and

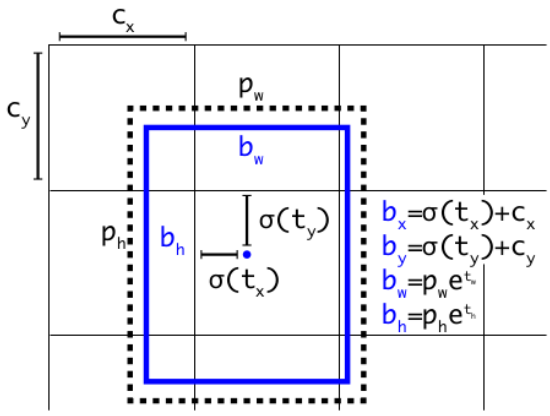
$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]$$

coordinate.
The reason why the author shift the ground truth box and pred box to (0,0) coordinate is that the x,y effect should be removed. It is a separable optimization technique !
Note that this time the author iterate the loop based on the ground truth, t=0 → 29, and calculate the penalty for the cost.
So learn this kind of technique !!! very clever !
Because the cost function is comprised of many parts, coordinate part,class (classification) part and objness part !

For coordinate part of cost function calculation we index from ground truth box : t = 0 → 29 ;

Now calculate the lost value :

```
int box_index = entry_index(l, b, best_n*l.w*l.h + j*l.w + i, 0);  // get the best bbox index from the 5 suggested bbox.
// caculate the coordinate part of lost value
float iou = delta_region_box(truth, l.output, l.biases, best_n, box_index, i, j, l.w, l.h, l.delta, l.coord_scale *  (2 - truth.w*truth.h), l.w*l.h);
//     float delta_region_box(box truth, float *x, float *biases, int n, int index, int i, int j, int w, int h, float *delta, float scale, int stride)
```



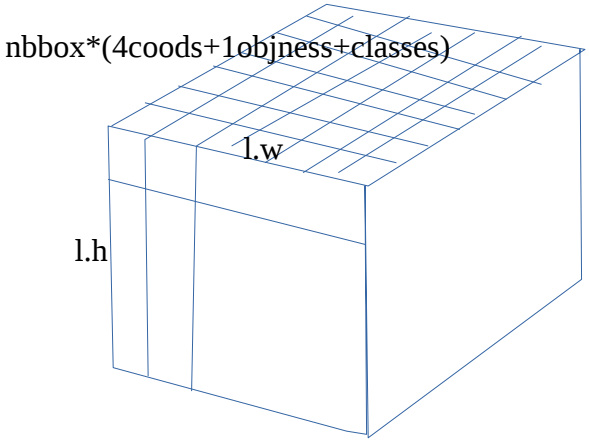

```
box get_region_box(float *x, float *biases, int n, int index, int i, int j, int w, int h, int stride)
{
    box b;
    b.x = (i + x[index + 0*stride]) / w;    // bx = delta(tx) + Cx , so x[index] is the logistic output
                                            // i is Cx the top left x coordinate of the grid cell in pixels
    b.y = (j + x[index + 1*stride]) / h;    // same as bx
    b.w = exp(x[index + 2*stride]) * biases[2*n]    / w;  // bw = Pw * exp(tw),
                                                          // so biases is the anchor box width and height
    b.h = exp(x[index + 3*stride]) * biases[2*n+1] / h;
    return b;
}
```

```
// float iou = delta_region_box(truth, l.output, l.biases, best_n, box_index, i, j, l.w, l.h, l.delta, l.coord_scale *  (2 - truth.w*truth.h), l.w*l.h);
float delta_region_box(box truth, float *x, float *biases, int n, int index, int i, int j, int w, int h, float *delta, float scale, int stride)
{
    // i is the truth center x in pixels, j is center y, top left point
    // w is l.w, the width of the last layer, h is the height
    // x is the output of last layer ?
    // index is the index of the box
    box pred = get_region_box(x, biases, n, index, i, j, w, h, stride);
    float iou = box_iou(pred, truth);
```

```
float tx = (truth.x*w - i);  // i is integer, truth.x* w is float.
float ty = (truth.y*h - j);
```

this one confused me at first glance of it. I thought tx,ty should be zero, it does not make sense for doing this.
Actually that is the beauty of normalized coordinates.

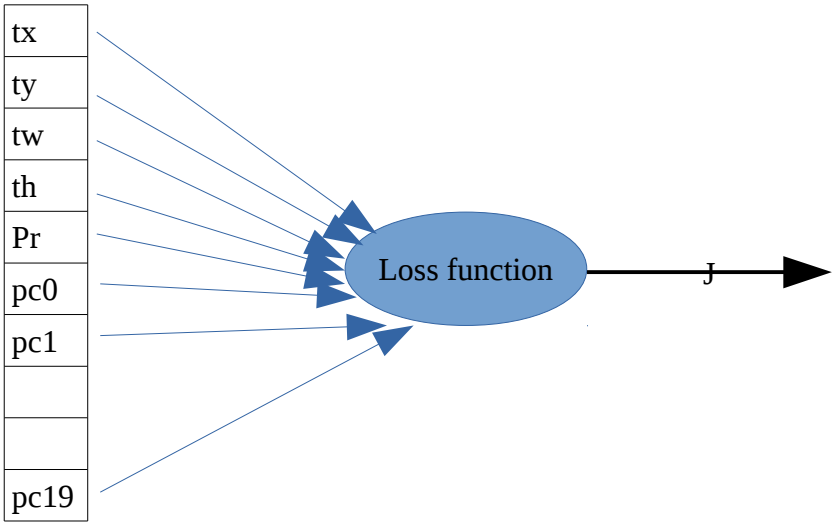
The truth.x is normalized by the original width of the training image, here truth.x * l.w is still the actual width in the current layer image.



The memory layout of the last conv output:
l.n consist of 5 bounding boxes.
Each bounding box consist of :

1. 4 coordinates
2. 1 confidence score,Pr(objness),indicating whether there exist an object.
3. 20 class conditional probability, showing how confident that object belongs to an class if there exist an object.

Q: what is the formula of backpro of regional layer ????



dJ/dxi^ = 2*λ * (xi – xi^) ;

```
float tx = (truth.x*w - i);  // i is integer, truth.x* w is float.
float ty = (truth.y*h - j);
float tw = log(truth.w*w / biases[2*n]);      // log(bw / Pw ), here truth.w * w = bw/orig_w * l.w
float th = log(truth.h*h / biases[2*n + 1]);
```

note that in get_region_box function :

```
b.w = exp(x[index + 2*stride]) * biases[2*n]   / w;  // bw = Pw * exp(tw),
                                                    // so biases is the anchor box width and height
```

x[index + 2 stride] = tw, biase[2n] is in pixels relating to truth object box size derived from k clustering. So b.w here is relative to l.w of the last layer. So that is why the tw of truth.w should be multiply by l.w, tw_truth and tw_pred should compare at the same scale.

Objness cost value :

$$+\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{ij}^{\text{obj}}\left(C_i-\hat{C}_i\right)^2$$

```
// ****** note : only compute the objness and class cost value at the best IOU bbox.
int box_index = entry_index(l, b, best_n*l.w*l.h + j*l.w + i, 0);  // get the best bbox index from the 5 suggested bbox.
// ***** caculate the coordinate part of lost value
float iou = delta_region_box(truth, l.output, l.biases, best_n, box_index, i, j, l.w, l.h, l.delta, l.coord_scale *  (2 - truth.w*truth.h), l.w*l.h);
//     float delta_region_box(box truth, float *x, float *biases, int n, int index, int i, int j, int w, int h, float *delta, float scale, int stride)
if(l.coords > 4){
    int mask_index = entry_index(l, b, best_n*l.w*l.h + j*l.w + i, 4);
    delta_region_mask(net.truth + t*(l.coords + 1) + b*l.truths + 5, l.output, l.coords - 4, mask_index, l.delta, l.w*l.h, l.mask_scale);
}
if(iou > .5) recall += 1;  // TP/(TP+FN)
avg_iou += iou;

int obj_index = entry_index(l, b, best_n*l.w*l.h + j*l.w + i, l.coords);   // get predicted object index at (i,j) grid cell
avg_obj += l.output[obj_index];
l.delta[obj_index] = l.object_scale * (1 - l.output[obj_index]); // the grid cell contains ground truth, so Ci = 1
if (l.rescore) {
    l.delta[obj_index] = l.object_scale * (iou - l.output[obj_index]);
}
if(l.background){
    l.delta[obj_index] = l.object_scale * (0 - l.output[obj_index]);
}
// **** calculate class cost value at the best iou bbox in the ground truth grid cell
int class = net.truth[t*(l.coords + 1) + b*l.truths + l.coords];   // find the class ID at the grid cell
if (l.map) class = l.map[class];
int class_index = entry_index(l, b, best_n*l.w*l.h + j*l.w + i, l.coords + 1); // find the class index
delta_region_class(l.output, l.delta, class_index, class, l.classes, l.softmax_tree, l.class_scale, l.w*l.h, &avg_cat, !l.softmax);
++count;
++class_count;
```

so far the coordinate,bbox size,objness and class portion of the cost function have been solved. Next need to solve noobjness part :

$$+\lambda_{\text{noobj}}\sum_{i=0}^{S^2}\sum_{j=0}^{B}\mathbb{1}_{ij}^{\text{noobj}}\left(C_i-\hat{C}_i\right)^2$$

to do this we need to for loop all the grid cells and bboxes,

so the whole code is show as below:

```
// ***** calculate the noobjness part of cost function
for (j = 0; j < l.h; ++j) {
    for (i = 0; i < l.w; ++i) {     // the final output size is 7*7,8*8,.....depending on the input image size
        for (n = 0; n < l.n; ++n) { // l.n = 5, each grid cell contains 5 bounding boxes
            int box_index = entry_index(l, b, n*l.w*l.h + j*l.w + i, 0);
            box pred = get_region_box(l.output, l.biases, n, box_index, i, j, l.w, l.h, l.w*l.h);
            float best_iou = 0;     // for each bbox, caculate the iou with each ground truth bbox
            for(t = 0; t < 30; ++t){
                box truth = float_to_box(net.truth + t*(l.coords + 1) + b*l.truths, 1);
                if(!truth.x) break;
                float iou = box_iou(pred, truth);
                if (iou > best_iou) {
                    best_iou = iou;  // choose the best IOU as an obj indicator for this bbox
                                     // if best_iou > thresh, it indicates that this bbox cantains an obj
                }
            }
            int obj_index = entry_index(l, b, n*l.w*l.h + j*l.w + i, l.coords); // get Pr(objness) for this bbox
            avg_anyobj += l.output[obj_index];
            l.delta[obj_index] = l.noobject_scale * (0 - l.output[obj_index]);  // by default we assume it contains no obj,
                                                                                // calculate the no objness cost value
            if(l.background) l.delta[obj_index] = l.noobject_scale * (1 - l.output[obj_index]);   // if it is a backgroud,no object here
            if (best_iou > l.thresh) {   // if best_iou > thresh, this bbox contains an obj,set it to 0 and
                l.delta[obj_index] = 0;  // will calculate its value later together with other portion of the cost values
            }

            if(*(net.seen) < 12800){  // why this number 12800 ?
                box truth = {0};
                truth.x = (i + .5)/l.w;
                truth.y = (j + .5)/l.h;
                truth.w = l.biases[2*n]/l.w;
                truth.h = l.biases[2*n+1]/l.h;
                delta_region_box(truth, l.output, l.biases, n, box_index, i, j, l.w, l.h, l.delta, .01, l.w*l.h);
            }
        }
    }
}
```

如果是 background，爲何是 1-objness?

l.delta[]維度?

```
// ******* compute objness,coord,size and class part of cost function
for(t = 0; t < 30; ++t){
    box truth = float_to_box(net.truth + t*(l.coords + 1) + b*l.truths, 1); // get the t-th truth box

    if(!truth.x) break;  // if truth.x = 0 , this class does not exitst in this image
    float best_iou = 0;
    int best_n = 0;
    i = (truth.x * l.w);  // the actual truth x center in pixels
    j = (truth.y * l.h);  //
    box truth_shift = truth;
    truth_shift.x = 0;    // why shift ?
    truth_shift.y = 0;    // setting x,y to 0 equals to shifting ? why?
    for(n = 0; n < l.n; ++n){  // l.n=5 the number of bounding boxes in one grid cell
        int box_index = entry_index(l, b, n*l.w*l.h + j*l.w + i, 0); // get the 5 predicted boxes in the grid cell where
                                                                     // the ground truth center is located;
                                                                     // need to know the pred box layout
        box pred = get_region_box(l.output, l.biases, n, box_index, i, j, l.w, l.h, l.w*l.h);  // 5 predicted boxes in the ground truth's location
        if(l.bias_match){                                            // what is bias_match ?
            pred.w = l.biases[2*n]/l.w;
            pred.h = l.biases[2*n+1]/l.h;
        }
        pred.x = 0;
        pred.y = 0;
        float iou = box_iou(pred, truth_shift);
        if (iou > best_iou){
            best_iou = iou;
            best_n = n;          // choose the bbox with best iou with the ground truth
        }
    }

    // ****** note : only compute the objness and class cost value at the best IOU bbox.
    int box_index = entry_index(l, b, best_n*l.w*l.h + j*l.w + i, 0);  // get the best bbox index from the 5 suggested bbox.
    // ****** caculate the coordinate part of lost value
    float iou = delta_region_box(truth, l.output, l.biases, best_n, box_index, i, j, l.w, l.h, l.delta, l.coord_scale *  (2 - truth.w*truth.h), l.w*l.h);
    //    float delta_region_box(box truth, float *x, float *biases, int n, int index, int i, int j, int w, int h, float *delta, float scale, int stride)
    if(l.coords > 4){
        int mask_index = entry_index(l, b, best_n*l.w*l.h + j*l.w + i, 4);
        delta_region_mask(net.truth + t*(l.coords + 1) + b*l.truths + 5, l.output, l.coords - 4, mask_index, l.delta, l.w*l.h, l.mask_scale);
    }
    if(iou > .5) recall += 1;  // TP/(TP+FN)
    avg_iou += iou;

    int obj_index = entry_index(l, b, best_n*l.w*l.h + j*l.w + i, l.coords);   // get predicted object index at (i,j) grid cell
    avg_obj += l.output[obj_index];
    l.delta[obj_index] = l.object_scale * (1 - l.output[obj_index]); // the grid cell contains ground truth, so Ci = 1
    if (l.rescore) {
        l.delta[obj_index] = l.object_scale * (iou - l.output[obj_index]);
    }
    if(l.background){
        l.delta[obj_index] = l.object_scale * (0 - l.output[obj_index]);
    }
    // **** calculate class cost value at the best iou bbox in the ground truth grid cell
    int class = net.truth[t*(l.coords + 1) + b*l.truths + l.coords];   // find the class ID at the grid cell
    if (l.map) class = l.map[class];
    int class_index = entry_index(l, b, best_n*l.w*l.h + j*l.w + i, l.coords + 1); // find the class index
    delta_region_class(l.output, l.delta, class_index, class, l.classes, l.softmax_tree, l.class_scale, l.w*l.h, &avg_cat, !l.softmax);
    ++count;
    ++class_count;
}
```

Q. in training we set the cost value of bbox with IOU > thresh to be 0 if it is not in the ground truth grid cell. I feel like there must be something wrong, or at least not make sense.

The wrong prediction is not punished in cost function !!!! which leads to unconstrained training in such case, if the DNN was not told how to deal with such condition it does not know how to predict more correctly!
The cost function should do some improvement !!
in such case when neighbouring grid cell has some high IOU with GT bbox,we should give some punishment to that so that the DNN know to move toward the correct grid cell. The cost value should be inversely proportional to the IOU size, which tells the prediction to move forward to the GT, leading to higher overlapping.

In inference we use max suppression method to eliminate the neighbouring grid cells' bbox prediction.

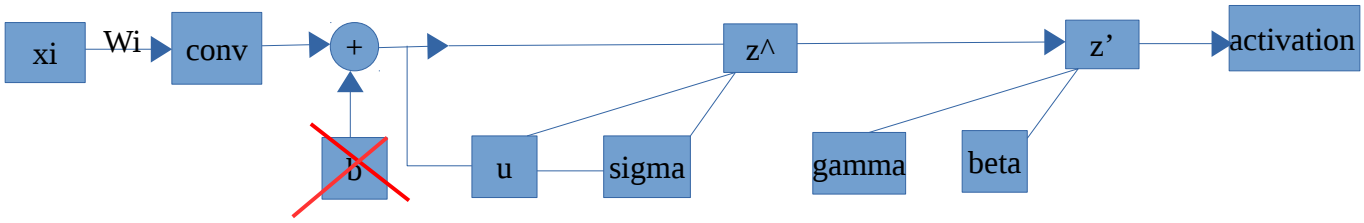
Batch Normalization :

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=0}^{M-1} x_i$$ Batch size = M

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=0}^{M-1} (x_i - \mu)^2$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + e}}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta$$



b is not necessary here,which is contained in beta parameter .

in inference μ_B and σ_B^2 is derived from running average in training phase. β and γ is learned by training.

Starting from forward then backward.

```
if(net.train){
    mean_cpu(l.output, l.batch, l.out_c, l.out_h*l.out_w, l.mean);
```

Compute the mean of the batch data.

```
for(i = 0; i < filters; ++i){
    mean[i] = 0;
    for(j = 0; j < batch; ++j){
        for(k = 0; k < spatial; ++k){
            int index = j*filters*spatial + i*spatial + k;
            mean[i] += x[index];
        }
    }
    mean[i] *= scale;
}
```

Calculate the mean[i] for convolution output channel [i], not for each element l.cout[r][c].

calculate variance

```
variance_cpu(l.output, l.mean, l.batch, l.out_c, l.out_h*l.out_w, l.variance);
```

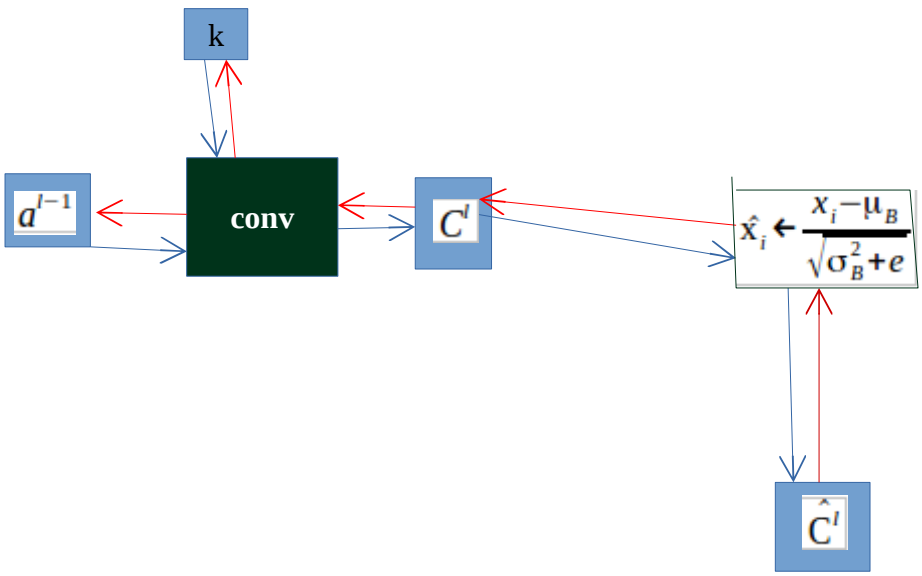
```
scal_cpu(l.out_c, .99, l.rolling_mean, 1);
/*
void scal_cpu(int N, float ALPHA, float *X, int INCX)
{
    int i;
    for(i = 0; i < N; ++i) X[i*INCX] *= ALPHA;
}
*/
axpy_cpu(l.out_c, .01, l.mean, 1, l.rolling_mean, 1);
/*
    void axpy_cpu(int N, float ALPHA, float *X, int INCX, float *Y, int INCY)
    {
        int i;
        for(i = 0; i < N; ++i) Y[i*INCY] += ALPHA*X[i*INCX];
    }
*/
```

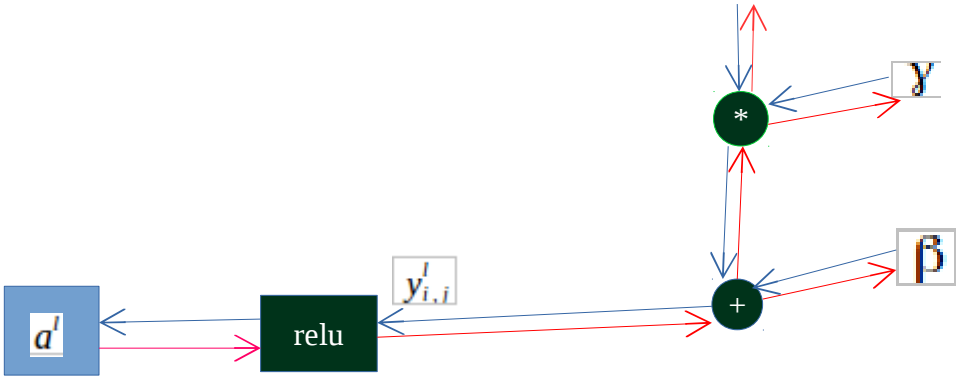
rolling_mean = rolling_mean * 0.99 + new_mean * 0.01 ; a slow change ,averaging process .

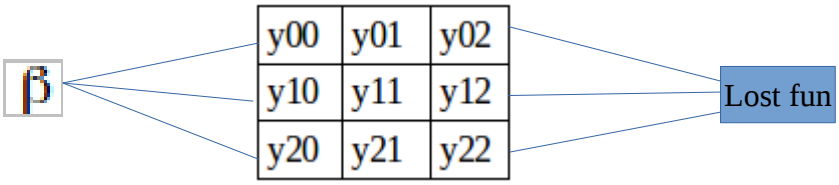
REVIEW ON BP

propagation path :

convolution : $C^l_{i,j} = \sum_{n=-(N-1)/2}^{(N-1)/2} \sum_{m=-(M-1)/2}^{(M-1)/2} a^{l-1}(i+n, j+m) * k(-n,-m)$,i,j is the row and column of input image, k is the kernel, n, m are the kernel coefficients.





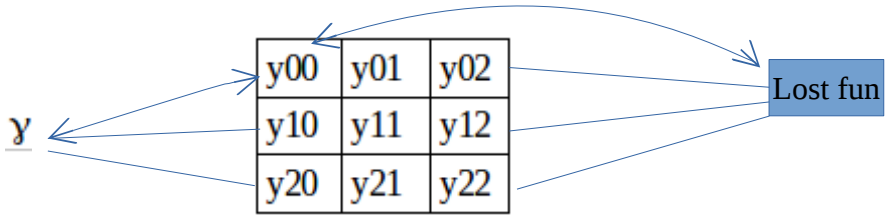


$y_{ch,i,j} \leftarrow \gamma_{ch} \hat{C}^l_{ch,i,j} + \beta_{ch}$ Adding subscript ch to denote that this is for per channel. γ_{ch} and β_{ch} is shared among all pixel elements in the same output channel.

so for output channel [i] :

$\frac{dl}{d\beta_i} = \sum_{b=0}^{B-1} \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} \frac{dl}{y_{i,b,m,n}} * \frac{dy_{i,b,m,n}}{d\beta_i}$,and $\frac{dy_{i,b,m,n}}{d\beta_i} = 1$ B is the batch number, n,m is the row and column of the output channel image, i is the output channel number.

$$\frac{dl}{d\gamma \hat{C}^l_{i,n,m}} = \sum_{b=0}^{b=N-1} \frac{dl}{y_{i,n,m}} * \frac{dy_{i,n,m}}{d\gamma \hat{C}^l_{i,n,m}} = \sum_{b=0}^{b=N-1} \frac{dl}{y_{i,n,m}}$$



$$\frac{dl}{d\gamma_i}=\frac{dl}{d\gamma\hat{C}^l_{n,m}}*\frac{d\gamma\hat{C}^l_{n,m}}{d\gamma}=\frac{dl}{d\gamma\hat{C}^l_{n,m}}*\hat{C}^l_{n,m}$$ **(this one is Wrong!!!!) scale and offset are shared among the whole elements in the same channel. Scale is not element wise.**

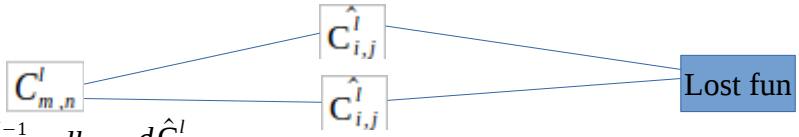
$$y_{ch,i,j}\leftarrow\gamma_{ch}\hat{C}^l_{ch,i,j}+\beta_{ch}$$

$$\frac{dl}{d\gamma_{ch}}=\sum_{b=0}^{B-1}\sum_{n=0}^{N-1}\sum_{m=0}^{M-1}\frac{dl}{dy_{ch,n,m}}*\frac{dy_{ch,n,m}}{d\gamma_{ch}}=\sum_{b=0}^{B-1}\sum_{n=0}^{N-1}\sum_{m=0}^{M-1}\frac{dl}{dy_{ch,n,m}}*\hat{C}^l_{ch,n,m}$$

```
void backward_scale_cpu(float *x_norm, float *delta, int batch, int n, int size, float *scale_updates)
{
    int i,b,f;
    for(f = 0; f < n; ++f){   // for each output channel
        float sum = 0;
        for(b = 0; b < batch; ++b){   // batch
            for(i = 0; i < size; ++i){  // all pixel elements in that channel
                int index = i + size*(f + n*b);
                sum += delta[index] * x_norm[index];  // summing all
            }
        }
        scale_updates[f] += sum;
    }
}
```

$$\frac{dl}{d\hat{C}^l_{n,m}}=\frac{dl}{d\gamma\hat{C}^l_{n,m}}*\frac{d\gamma\hat{C}^l_{n,m}}{d\hat{C}^l_{n,m}}=\frac{dl}{d\gamma\hat{C}^l_{n,m}}*\gamma_i$$

$\hat{C}^l_{i,j}=\frac{C^l_{i,j}-\mu}{\sqrt{\sigma^2+e}}$ so $\hat{C}^l_{i,j}$ is related to every input pixel $C^l_{m,n}$ because of mean and variance,so according to chain rule



$$\frac{dl}{dC^l_{n,m}}=\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{dl}{d\hat{C}^l_{i,j}}*\frac{d\hat{C}^l_{i,j}}{dC^l_{n,m}}$$ chain rule expansion

$\mu_{ch}=\frac{1}{B*N*M}\sum_{b=0}^{B}\sum_{n=0}^{N-1}\sum_{m=0}^{M-1}C^l_{ch,b,n,m}$ ch is the output channel number, don't forget subscript b, for each channel,we have B batch size of inputs. So when calculate the derivatives of loss function with respect to input, you have B*N*M inputs here for each channel !!!!

$$\sigma^2_{ch}=\frac{1}{B*N*M-1}\sum_{b=0}^{B}\sum_{n=0}^{N-1}\sum_{m=0}^{M-1}(C^l_{ch,n,m}-\mu_{ch})^2$$ $\frac{dl}{d\hat{C}^l_{i,j}}=\frac{dl}{dy^l_{i,j}}*\frac{dy^l_{i,j}}{d\hat{C}^l_{i,j}}=\frac{dl}{dy^l_{i,j}}*\gamma$ --eq(2)

$$\frac{dl}{dC^l_{b,n,m}}=\sum_{b'=0}^{B-1}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{dl}{d\hat{C}^l_{b',i,j}}*\frac{d\hat{C}^l_{b',i,j}}{dC^l_{b,n,m}}=\sum_{b'=0}^{B-1}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{dl}{d\hat{C}^l_{b',i,j}}*d\frac{C^l_{b',i,j}-\mu_{ch}}{\sqrt{\sigma^2+e}}}{dC^l_{b,n,m}}$$ ----- eq (1)

derivative of lost function respect to input.

$$\frac{d\frac{C^l_{b',i,j}-\mu_{ch}}{\sqrt{\sigma^2+e}}}{dC^l_{b,n,m}}=\frac{d\frac{C^l_{b',i,j}-\mu_{ch}}{\sqrt{\sigma^2+e}}}{dC^l_{b',i,j}}*\frac{dC^l_{b',i,j}}{dC^l_{b,n,m}}+\frac{d\frac{C^l_{b',i,j}-\mu_{ch}}{\sqrt{\sigma^2+e}}}{d\mu_{ch}}*\frac{d\mu_{ch}}{dC^l_{b,n,m}}+\frac{d\frac{C^l_{b',i,j}-\mu_{ch}}{\sqrt{\sigma^2+e}}}{d\sigma^2}*\frac{d\sigma^2}{dC^l_{b,n,m}}$$ chain rule

so eq(1) consist of three parts. Pay very close attention to subscripts:

$$\frac{dl}{dC^l_{b,n,m}}=\sum_{b'=0}^{B-1}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{dl}{d\hat{C}^l_{b',i,j}}*\frac{d\frac{C^l_{b',i,j}-\mu_{ch}}{\sqrt{\sigma^2+e}}}{dC^l_{b,n,m}}=\sum_{b'=0}^{B-1}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{dl}{dy^l_{b',i,j}}*\frac{dy^l_{b',i,j}}{d\hat{C}^l_{b',i,j}}*\frac{d\hat{C}^l_{b',i,j}}{dC^l_{b,n,m}}$$

$$=\sum_{b'=0}^{B-1}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{dl}{dy^l_{b',i,j}}*\gamma*\frac{d\hat{C}^l_{b',i,j}}{dC^l_{b,n,m}}$$

$$=\sum_{b'=0}^{B-1}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{dl}{dy^l_{b',i,j}}*\gamma*\frac{d\frac{C^l_{b',i,j}-\mu_{ch}}{\sqrt{\sigma^2+e}}}{dC^l_{b,n,m}}$$

$$=\sum_{b'=0}^{B-1}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{dl}{dy^l_{b',i,j}}*\gamma*\left(\frac{d\frac{C^l_{b',i,j}-\mu_{ch}}{\sqrt{\sigma^2+e}}}{dC^l_{b',i,j}}*\frac{dC^l_{b',i,j}}{dC^l_{b,n,m}}+\frac{d\frac{C^l_{b',i,j}-\mu_{ch}}{\sqrt{\sigma^2+e}}}{d\mu_{ch}}*\frac{d\mu_{ch}}{dC^l_{b,n,m}}+\frac{d\frac{C^l_{b',i,j}-\mu_{ch}}{\sqrt{\sigma^2+e}}}{d\sigma^2}*\frac{d\sigma^2}{dC^l_{b,n,m}}\right)$$

----------**eq(3)**

the first term of eq (3) :

$$\sum_{b'=0}^{B-1}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{dl}{dy^l_{b',i,j}}*\gamma*\frac{d\frac{C^l_{b',i,j}-\mu_{ch}}{\sqrt{\sigma^2+e}}}{dC^l_{b,n,m}}=\frac{dl}{dy^l_{b,n,m}}*\gamma*\frac{1}{\sqrt{\sigma^2+e}}$$ only when b=b',n=i,m=j;

the second term of eq (3) :

$$\sum_{b'=0}^{B-1}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{dl}{dy^l_{b',i,j}}*\gamma*\frac{d\frac{C^l_{b',i,j}-\mu_{ch}}{\sqrt{\sigma^2+e}}}{d\mu_{ch}}*\frac{d\mu_{ch}}{dC^l_{b,n,m}}=\sum_{b'=0}^{B-1}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{dl}{dy^l_{b',i,j}}*\gamma*\frac{-1}{\sqrt{\sigma^2+e}}*\frac{1}{B*N*M}\delta_{b',b}\delta_{n,i}\delta_{m,j}$$

$$=\sum_{b'=0}^{B-1}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{dl}{dy^l_{b',i,j}}*\gamma*\frac{-1}{\sqrt{\sigma^2+e}}*\frac{1}{B*N*M}$$

$$=\gamma*\frac{-1}{\sqrt{\sigma^2+e}}*\frac{1}{B*N*M}*\sum_{b'=0}^{B-1}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{dl}{dy^l_{b',i,j}}$$

$$\frac{d\mu_{ch}}{dC^l_{ch,b',n',m'}}=d\frac{\frac{1}{B*N*M}\sum_{b=0}^{B-1}\sum_{n=0}^{N-1}\sum_{m=0}^{M-1}C^l_{ch,b,n,m}}{dC^l_{ch,b',n',m'}}$$

$$=\frac{1}{B*N*M}\delta_{b',b}\delta_{n',n}\delta_{m',m}=\frac{1}{B*N*M}$$

the third term of eq (3) :

$$\sum_{b'=0}^{B-1}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{dl}{dy^l_{b',i,j}}*\gamma*\frac{d\frac{C^l_{b',i,j}-\mu_{ch}}{\sqrt{\sigma^2+e}}}{d\sigma^2}*\frac{d\sigma^2}{dC^l_{b,n,m}}=\sum_{b'=0}^{B-1}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{\frac{dl}{dy^l_{b',i,j}}*\gamma*\frac{1}{2}(C^l_{b',i,j}-\mu_{ch})}{(\sigma^2+e)^{\frac{-3}{2}}}*\frac{d\sigma^2}{dC^l_{b,n,m}}$$

$$\frac{d\sigma^2}{dC^l_{b,n,m}}=\frac{1}{B*N*M}*d\frac{\sum_{b'=0}^{B-1}\sum_{n'=0}^{N-1}\sum_{m'=0}^{M}(C^l_{b',n',m'}-\mu_{ch})^2}{dC^l_{b,n,m}}$$

$$=\frac{1}{B*N*M}*(2*(C^l_{b,n,m}-\mu_{ch})*1+d\frac{\sum_{b'=0}^{B-1}\sum_{n'=0}^{N-1}\sum_{m'=0}^{M-1}(C^l_{b',n',m'}-\mu_{ch})^2}{d\mu_{ch}}*d\frac{\mu_{ch}}{dC^l_{b,n,m}})$$

$$=\frac{1}{B*N*M}*(2*(C^l_{b,n,m}-\mu_{ch})*1-\sum_{b=0}^{B-1}\sum_{n'=0}^{N-1}\sum_{m'=0}^{M-1}2*(C^l_{b',n',m'}-\mu_{ch})*\frac{1}{B*N*M})=\frac{1}{B*N*M}*2*(C^l_{b,n,m}-\mu_{ch})$$

wrong chain rule ? No need to consider mean here . μ_{ch} is considered to be a constant.

So the third term equals to :

$$\sum_{b'=0}^{B-1}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{dl}{dy^l_{b',i,j}}*\gamma*\frac{d\frac{C^l_{b',i,j}-\mu_{ch}}{\sqrt{\sigma^2+e}}}{d\sigma^2}*\frac{d\sigma^2}{dC^l_{b,n,m}}=\sum_{b'=0}^{B-1}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{\frac{dl}{dy^l_{b',i,j}}*\gamma*\frac{1}{2}(C^l_{b',i,j}-\mu_{ch})}{(\sigma^2+e)^{\frac{-3}{2}}}*\frac{d\sigma^2}{dC^l_{b,n,m}}$$

$$=\frac{1}{B*N*M}*2*(C^l_{b,n,m}-\mu_{ch})*\gamma*\frac{0.5}{(\sigma^2+e)^{\frac{-3}{2}}}*\sum_{b'=0}^{B-1}\sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\frac{dl}{dy^l_{b',i,j}}*(C^l_{b',i,j}-\mu_{ch})$$

What is l.bias

```
void backward_batchnorm_layer(layer l, network net)
{
    if(!net.train){
434
435     void backward_bias(float *bias_updates, float *delta, int batch, int n, int size)
436     {
437         int i,b;
438         for(b = 0; b < batch; ++b){
439             for(i = 0; i < n; ++i){  // i is the output channel index
440                 bias_updates[i] += sum_array(delta+size*(i+b*n), size); // summing over all
441                                                                         // image on the channel
442
443                 /*
444                     float sum_array(float *a, int n)
445                     {
446                         int i;
447                         float sum = 0;
448                         for(i = 0; i < n; ++i) sum += a[i];
449                         return sum;
450                     }
451                 */
452             }
453         }
454     }
```

$$\frac{dl}{d\beta_{ch}}=\sum_{b=0}^{B-1}\sum_{n=0}^{N-1}\sum_{m=0}^{M-1}\frac{dl}{y_{ch,b,m,n}}*\frac{dy_{i,b,m,n}}{d\beta_{ch}}=\sum_{b=0}^{B-1}\sum_{n=0}^{N-1}\sum_{m=0}^{M-1}\frac{dl}{y_{ch,b,m,n}}$$ *delta is the sensitive value $\frac{dl}{y_{ch,b,m,n}}$ for back-propagation .

Q: backward_scale_cpu seems not match with my equation, check out what is wrong.
The parameter gamma is shared among all pixel elements in the same output channel.

```
181     scale_bias(l.delta, l.scales, l.batch, l.out_c, l.out_h*l.out_w);
```

```
423 void scale_bias(float *output, float *scales, int batch, int n, int size)
424 {
425     int i,j,b;
426     for(b = 0; b < batch; ++b){
427         for(i = 0; i < n; ++i){
428             for(j = 0; j < size; ++j){
429                 output[(b*n + i)*size + j] *= scales[i];
430             }
431         }
432     }
433 }
```

What is scale_bias ? For what purpose ?
What the above code does is as follows:

$\frac{dl}{y_{ch,b,m,n}}*\gamma_{ch}$ and it actually is derivative of loss function with respect to normalized convolution output C(i,j). $\frac{dl}{d\hat{C}^{l}_{ch,b,m,n}}$ ----this is eq(2),so it is a intermediate value for solving eq (1).

```
87  //   mean_delta_cpu(l.delta, l.variance, l.batch, l.out_c, l.out_w*l.out_h, l.mean_delta);
88  void mean_delta_cpu(float *delta, float *variance, int batch, int filters, int spatial, float *mean_delta)
89  {
90
91      int i,j,k;
92      for(i = 0; i < filters; ++i){  // for each channel
93          mean_delta[i] = 0;
94          for (j = 0; j < batch; ++j) {  // batch
95              for (k = 0; k < spatial; ++k) {  // all pixels in one channel
96                  int index = j*filters*spatial + i*spatial + k;
97                  mean_delta[i] += delta[index];  // mean_delta = sum from 0 to size {dl/dy(ch,n,m)}
98              }
99          }
100         mean_delta[i] *= (-1./sqrt(variance[i] + .00001f));   // mean_delta = sum from 0 to size {dl/dy(ch,n,m)} / -sqrt(var + e)
101     }
102 }
```

what the mean_delta_cpu code does is as follows :

$\frac{-1.0}{\sqrt{\sigma^2+e}}\sum_{b=0}^{B-1}\sum_{n=0}^{N-1}\sum_{m=0}^{M-1}\frac{dl}{dy_{ch,n,m}}$ for what purpose ?

```
// variance_delta_cpu(l.x, l.delta, l.mean, l.variance, l.batch, l.out_c, l.out_w*l.out_h, l.variance_delta);
void  variance_delta_cpu(float *x, float *delta, float *mean, float *variance, int batch, int filters, int spatial, float *variance_delta)
{

    int i,j,k;
    for(i = 0; i < filters; ++i){
        variance_delta[i] = 0;
        for(j = 0; j < batch; ++j){
            for(k = 0; k < spatial; ++k){
                int index = j*filters*spatial + i*spatial + k;
                variance_delta[i] += delta[index]*(x[index] - mean[i]);
            }
        }
        variance_delta[i] *= -.5 * pow(variance[i] + .00001f, (float)(-3./2.));
    }
}
```

$$\frac{-0.5}{\left(\sigma^2+e\right)^{\frac{-3}{2}}}\sum_{b=0}^{B-1}\sum_{n=0}^{N-1}\sum_{m=0}^{M-1}\frac{dl}{dy_{ch,n,m}}*\left(C^l_{ch,n,m}-\mu_{ch}\right)$$

```
//normalize_delta_cpu(l.x, l.mean, l.variance, l.mean_delta, l.variance_delta, l.batch, l.out_c, l.out_w*l.out_h, l.delta);
void normalize_delta_cpu(float *x, float *mean, float *variance, float *mean_delta, float *variance_delta, int batch, int filters, int spatial, float *delta)
{
    int f, j, k;
    for(j = 0; j < batch; ++j){
        for(f = 0; f < filters; ++f){
            for(k = 0; k < spatial; ++k){
                int index = j*filters*spatial + f*spatial + k;
                delta[index] = delta[index] * 1./(sqrt(variance[f] + .00001f)) + variance_delta[f] * 2. * (x[index] - mean[f]) / (spatial * batch) + mean_delta[f]/(spatial*batch);
            }
        }
    }
}
```

write the output expression in paper and take a picture !
This is exactly eq (1) .

Backpro in Conv layer

the maths :

$$C^l_{i,j} = \sum_{n=-(N-1)/2}^{(N-1)/2}\sum_{m=-(M-1)/2}^{(M-1)/2} a^{l-1}(i+n,j+m)*K(-n,-m)$$

if Kernel K is symmetric, usually is not the case in deep learning, then

$$= \sum_{n=-(N-1)/2}^{(N-1)/2}\sum_{m=-(M-1)/2}^{(M-1)/2} a^{l-1}(i+n,j+m)*K(n,m)$$

$$K(-n,-m) = K(n,m)$$

given $\delta^l_{i,j} = \frac{\partial l}{\partial C^l_{i,j}}$ is known, the objective is to calculate

1. the derivative of loss function with respect to the convolution input $a^{l-1}_{i,j}$,ie $\partial l/\partial a^{l-1}_{i,j}$

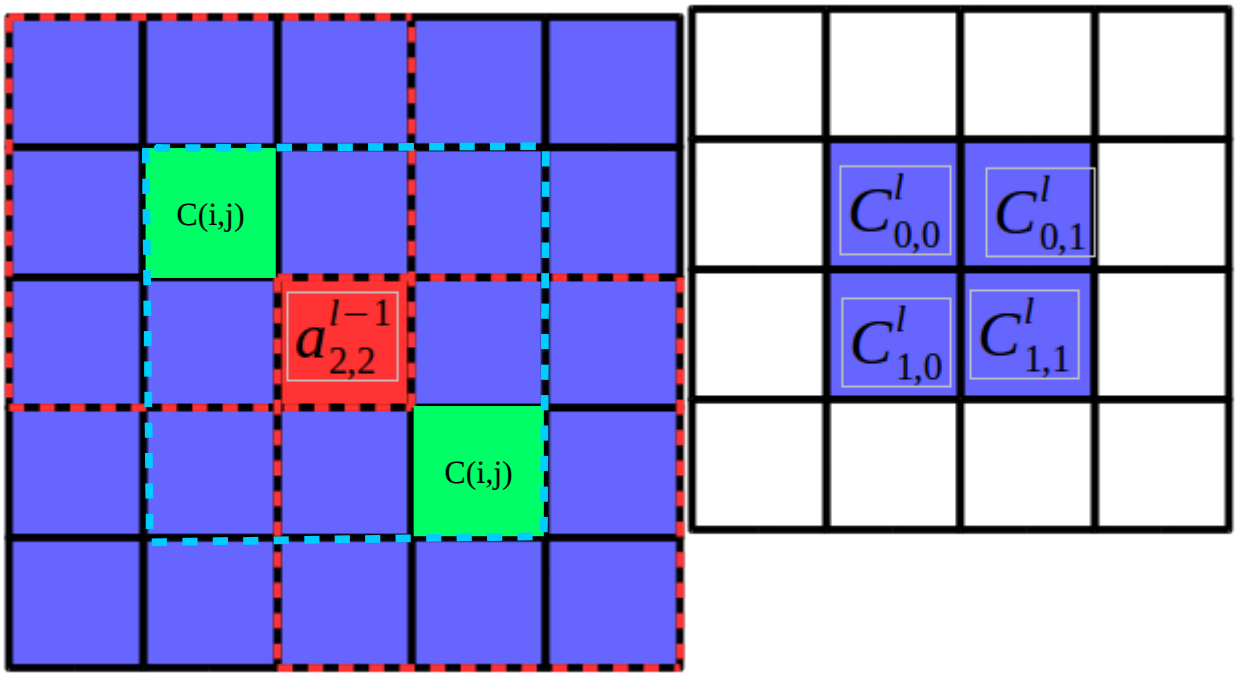
which will be used for calculating backpro in the previous layer. $a^{l-1}_{i,j}$ Is the activation output of the previous condition layer, the input of the current convolution layer.

2. the derivative of loss function with respect to the kernel weights, ie $\partial l / \partial K^{l}_{n,m}$ which is used for weight update.

- **Deriving** $\partial l / \partial a^{l-1}_{i,j}$:

by chain rule we have

$$\frac{\partial l}{\partial a^{l-1}_{i,j}} = \sum_{i'=0}^{I-1} \sum_{j'=0}^{J-1} \frac{\partial l}{\partial C^{l}_{i',j'}} \frac{\partial C^{l}_{i',j'}}{\partial a^{l-1}_{i,j}}$$ ----------**(9)** ,this formula says every convolution output $C^{l}_{i',j'}$ is a function of convolution input $a^{l-1}_{i,j}$, but due to the size of Kernel filter, which is M*N, not every (i',j') is related to (i,j), only these (i',j') in the range of (i+/-(N-1)/2,j+/-(M-1)/2) is related.



Conv output pixels related To input pixel a(2,2), other output pixel C(i,j) is not related to current input pixel
This box size is exactly the filter size

So for any input index (i,j), draw a bounding box of filter size with the center as (i,j), all the C(i+/-(N-1)/2,j+/-(M-1)/2) output pixels is the ones we looking for.

So formula (9) can be rewritten as :
I know this geometric interpretation,but the rigorous formula is hard to derive.
For subscript index convenience we need to make the input indexing and output indexing the same, we pad zeros in the output to make it the output size the same as that of the input image.
For kernel size 3*3 we pad one row or column of zeros along the boundary.

Qestion :
in same convolution. What is the exact process, padding zeros in input or padding zeros at the output ? How does it affect the back propagation ?
If we pad zeros at the input image,

$$C^l_{i,j} = \sum_{n=-(N-1)/2}^{(N-1)/2} \sum_{m=-(M-1)/2}^{(M-1)/2} a^{l-1}(i+n, j+m) * K(-n,-m)$$

$$\frac{(N-1)}{2} \le i \le I + \frac{(N-1)}{2} \quad \frac{(M-1)}{2} \le j \le J + \frac{(M-1)}{2}$$

the padded size of the input image is (I+N-1)*(J+M-1), $a^{l-1}(i+n, j+m)$ is ranging from 0 to I+N-1,and 0 to J+M-1. $C^l_{i,j}$ indexing from i=(N-1)/2 to I+(N-1)/2, j=(M-1)/2 to J+(M-1)/2, the final size of $C^l_{i,j}$ is still I * J.

$$\frac{\partial l}{\partial a^{l-1}_{1,1}} = \sum_{i'=0}^{I-1} \sum_{j'=0}^{J-1} \frac{\partial l}{\partial C^l_{i',j'}} \frac{\partial C^l_{i',j'}}{\partial a^{l-1}_{i,j}}$$

$$= \frac{\partial l}{\partial C^l_{1,1}} \frac{\partial C^l_{1,1}}{\partial a^{l-1}_{1,1}}$$

$$\frac{\partial l}{\partial a_{i,j}^{l-1}} = \sum_{i'=0}^{I-1}\sum_{j'=0}^{J-1} \frac{\partial l}{\partial C_{i',j'}^{l}} \frac{\partial C_{i',j'}^{l}}{\partial a_{i,j}^{l-1}}$$

$$= \sum_{i'=i-(N-1)/2}^{i+(N-1)/2} \sum_{j'=j-(M-1)/2}^{j+(M-1)/2} \frac{\partial l}{\partial C_{i',j'}^{l}} \frac{\partial C_{i',j'}^{l}}{\partial a_{i,j}^{l-1}}$$

$$= \sum_{i'=i-(N-1)/2}^{i+(N-1)/2} \sum_{j'=j-(M-1)/2}^{j+(M-1)/2} \delta_{i',j'}^{l} * \frac{\partial C_{i',j'}^{l}}{\partial a_{i,j}^{l-1}}$$

$$= \sum_{i'=i-(N-1)/2}^{i+(N-1)/2} \sum_{j'=j-(M-1)/2}^{j+(M-1)/2} \delta_{i',j'}^{l} * \frac{\partial \sum_{n=-(N-1)/2}^{(N-1)/2} \sum_{m=-(M-1)/2}^{(M-1)/2} a^{l-1}(i'+n, j'+m) * K(-n,-m)}{\partial a_{i,j}^{l-1}}$$

$$= \sum_{i'=i-(N-1)/2}^{i+(N-1)/2} \sum_{j'=j-(M-1)/2}^{j+(M-1)/2} \delta_{i',j'}^{l} * \delta_{i'+n,i}\delta_{j'+m,j} * K(-n,-m) \qquad i'+n=i, j'+m=j$$

$$= \sum_{i'-i=-(N-1)/2}^{i'-i=(N-1)/2} \sum_{j'-j=-(M-1)/2}^{j'-j=(M-1)/2} \delta^{l}(i',j') * K(i'-i, j'-j) \qquad i'-i=-n, j'-j=-m$$

$$= \sum_{n=-(N-1)/2}^{(N-1)/2} \sum_{m=-(M-1)/2}^{(M-1)/2} \delta^{l}(i+n, j+m) * K(n,m) \qquad i'-i=-n, j'-j=-m, n=-n, m=-m$$

so the final expression turns out to be a correlation. Nice and beautiful !

The interpretation of this equation:
K(n,m) is the rotated version of K(-n,-m) which is the Kernel we used to cross correlate with input a(i,j) to generate the convolution output, and in code we store Kernel in K(-n,-m) order, such that it is very convenient to do forward pass.
$\delta(i,j)$ is the sensitive map of output, i,j is ranging from [(N-1)/2,I-1],and [(M-1)/2,J-1] respectively,because when i,and j are out of this boundary, $a_{i,j}^{l-1}$ is zero, which is the padded value,the derivatives with respect to these values are meaningless.
So the partial derivative of loss function with respect to $a_{i,j}^{l-1}$ is just a cross correlation between $\delta(i,j)$ and K(n,m) which is the rotated version of kernel K(-n,-m) which we stored in memory. This matches the graphical interpretation below,(to be added).

```
for(i = 0; i < l.batch; ++i){
    for(j = 0; j < l.groups; ++j){   // divide output n into groups ?
        float *a = l.delta + (i*l.groups + j)*m*k;  // dl/dC^l_(i,j)
        float *b = net.workspace;  // place to store the whole intermidiate values
        float *c = l.weight_updates + j*l.nweights/l.groups;  // K(n,w), the weights

        float *im  = net.input + (i*l.groups + j)*l.c/l.groups*l.h*l.w;  // a^{l-1}_{i,j}
        float *imd = net.delta + (i*l.groups + j)*l.c/l.groups*l.h*l.w;   // dl/da^{l-1}_{i,j}

        if(l.size == 1){
            b = im;
        } else {
            im2col_cpu(im, l.c/l.groups, l.h, l.w,
                    l.size, l.stride, l.pad, b);   // re-arrange a^{l-1}_{i,j} matrix
        }

        gemm(0,1,m,n,k,1,a,k,b,k,1,c,n);   /* compute dl/dK(n,m) used to update K(n,m), correlate re-arranged,
                                              a^{l-1}_{i,j} matrix whose size is {I,J},with dl/dC^l_{i',j'}
                                              whose size is {I-M+1,J-N+1}, the output size of this correlation
                                              is I-(I-M+1-1)=M,& N, which is the kernel filter size .
                                               dl/dK(n,m) = sum from i'= 0 to I-N+1 sum m=0 to J-M+1
                                                   { a^{l-1}(i'+n,j'+m) * dl/dC^l(i',j') }, this is for
                                                   batch size = 1. for batch size > 1, we have to sum up all
                                                   elements in the batch
                                             */
```

```
if (net.delta) {  // compute dl/da^{l-1}_{i,j},used for previous stage backpropagation
    a = l.weights + j*l.nweights/l.groups;
    b = l.delta + (i*l.groups + j)*m*k;
    c = net.workspace;
    if (l.size == 1) {
        c = imd;
    }

    gemm(1,0,n,k,m,1,a,n,b,k,0,c,k);  /*  sum from n=-(N-1)/2 to (N-1)/2 sum m=-(M-1)/2 to (M-1)/2 {dl/dC^l_{i+n,j+m}
                                            * K(n,m)}

                                      */

    if (l.size != 1) {
        col2im_cpu(net.workspace, l.c/l.groups, l.h, l.w, l.size, l.stride, l.pad, imd); // why this ???
    }
}
```

Feels like the author's code of back propagation for convolution is wrong !!!
as to correlation between input $a_{i,j}^{l-1}$ and $\delta(i,j)$,since both of their width and height are large,there is no need to do im2col operation, which will takes up a lot of memory,it is better just do the correlation as it is.
It really makes no sense passing in parameter l.size,which is 3, here.
But how can yolo make such mistakes ?
Nasty is that how can I verify that my code is right if I rewrite it ?
Got no one to discuss, it is really lonely !

Q: have always been confused where the ground truth labels are passed to the program.
A : in function : **void fill_truth_detection**(char *path, int num_boxes, float *truth, int classes, int flip, float dx, float dy, float sx, float sy)

```
find_replace(path, "images", "labels", labelpath);
find_replace(labelpath, "JPEGImages", "labels", labelpath);

find_replace(labelpath, "raw", "labels", labelpath);
find_replace(labelpath, ".jpg", ".txt", labelpath);
find_replace(labelpath, ".png", ".txt", labelpath);
find_replace(labelpath, ".JPG", ".txt", labelpath);
find_replace(labelpath, ".JPEG", ".txt", labelpath);
```

for now just leave this problem aside,no need to go too deep into this code,actually it is quite a simple problem, I can easily write my own code to read the bbox.
Convert widerface data into yolo data format first. should be not difficult at all .

```
box_label *read_boxes(char *filename, int *n)
{
    FILE *file = fopen(filename, "r");
    if(!file) file_error(filename);
    float x, y, h, w;
    int id;
    int count = 0;
    int size = 64;
    box_label *boxes = calloc(size, sizeof(box_label));
    while(fscanf(file, "%d %f %f %f %f", &id, &x, &y, &w, &h) == 5){
        if(count == size) {
            size = size * 2;
            boxes = realloc(boxes, size*sizeof(box_label)); // double the size if overflowed
        }
        boxes[count].id = id;
        boxes[count].x = x;
        boxes[count].y = y;
        boxes[count].h = h;
        boxes[count].w = w;
        boxes[count].left   = x - w/2;
        boxes[count].right  = x + w/2;
        boxes[count].top    = y - h/2;
        boxes[count].bottom = y + h/2;
        ++count;
    }
    fclose(file);
    *n = count;
    return boxes;
}
```

FINALLY !!!!

```
0--Parade/0_Parade_Parade_0_452.jpg
1
0 0 0 0 0 0 0 0 0 0
```

it is pretty hard to find this bug !!! it is because there is a notation error in widerface datasets, all 0s bbox .
How I debug identify this problem, pretty dummy!!! by printing out the bbox files that before the segmentation fault is met ! Find the repeated one

```
Loaded: 0.000030 seconds
filename = /media/brian/FILE/FD/dataset/WIDER_train/labels/0--Parade/0_Parade_Parade_0_271.txt
filename = /media/brian/FILE/FD/dataset/WIDER_train/labels/0--Parade/0_Parade_Parade_0_452.txt
filename = /media/brian/FILE/FD/dataset/WIDER_train/labels/0--Parade/0_Parade_Parade_0_327.txt
filename = /media/brian/FILE/FD/dataset/WIDER_train/labels/0--Parade/0_Parade_Parade_0_352.txt
filename = /media/brian/FILE/FD/dataset/WIDER_train/labels/0--Parade/0_Parade_Parade_0_55.txt
filename = /media/brian/FILE/FD/dataset/WIDER_train/labels/0--Parade/0_Parade_marchingband_1_278.txt
filename = /media/brian/FILE/FD/dataset/WIDER_train/labels/0--Parade/0_Parade_Parade_0_431.txt
filename = /media/brian/FILE/FD/dataset/WIDER_train/labels/0--Parade/0_Parade_marchingband_1_849.txt
Region Avg IOU: 0.000411, Class: 1.000000, Obj: 0.286350, No Obj: 0.423191, Avg Recall: 0.000000,  count: 22
Region Avg IOU: 0.063117, Class: 1.000000, Obj: 0.195302, No Obj: 0.417910, Avg Recall: 0.000000,  count: 12
Region Avg IOU: 0.126713, Class: 1.000000, Obj: 0.372518, No Obj: 0.420774, Avg Recall: 0.000000,  count: 4
Region Avg IOU: 0.003576, Class: 1.000000, Obj: 0.081934, No Obj: 0.420913, Avg Recall: 0.000000,  count: 1
Region Avg IOU: 0.053003, Class: 1.000000, Obj: 0.415950, No Obj: 0.418532, Avg Recall: 0.000000,  count: 10
Region Avg IOU: 0.094850, Class: 1.000000, Obj: 0.330855, No Obj: 0.418984, Avg Recall: 0.000000,  count: 15
Region Avg IOU: 0.000000, Class: 1.000000, Obj: 0.407818, No Obj: 0.419824, Avg Recall: 0.000000,  count: 1
Region Avg IOU: 0.020587, Class: 1.000000, Obj: 0.228576, No Obj: 0.420924, Avg Recall: 0.000000,  count: 3
28: 340.887665, 490.516357 avg, 0.000000 rate, 0.737354 seconds, 224 images
Loaded: 0.000042 seconds
filename = /media/brian/FILE/FD/dataset/WIDER_train/labels/0--Parade/0_Parade_Parade_0_703.txt
filename = /media/brian/FILE/FD/dataset/WIDER_train/labels/0--Parade/0_Parade_Parade_0_434.txt
filename = /media/brian/FILE/FD/dataset/WIDER_train/labels/0--Parade/0_Parade_marchingband_1_431.txt
filename = /media/brian/FILE/FD/dataset/WIDER_train/labels/0--Parade/0_Parade_Parade_0_204.txt
filename = /media/brian/FILE/FD/dataset/WIDER_train/labels/0--Parade/0_Parade_Parade_0_47.txt
filename = /media/brian/FILE/FD/dataset/WIDER_train/labels/0--Parade/0_Parade_Parade_0_52.txt
filename = /media/brian/FILE/FD/dataset/WIDER_train/labels/0--Parade/0_Parade_marchingband_1_921.txt
filename = /media/brian/FILE/FD/dataset/WIDER_train/labels/0--Parade/0_Parade_marchingband_1_897.txt
Region Avg IOU: 0.098159, Class: 1.000000, Obj: 0.239114, No Obj: 0.420593, Avg Recall: 0.000000,  count: 5
Region Avg IOU: 0.010664, Class: 1.000000, Obj: 0.301002, No Obj: 0.420369, Avg Recall: 0.000000,  count: 5
Region Avg IOU: 0.159404, Class: 1.000000, Obj: 0.236037, No Obj: 0.421181, Avg Recall: 0.000000,  count: 1
Region Avg IOU: 0.035105, Class: 1.000000, Obj: 0.337703, No Obj: 0.416664, Avg Recall: 0.000000,  count: 30
Region Avg IOU: 0.118028, Class: 1.000000, Obj: 0.337164, No Obj: 0.418433, Avg Recall: 0.000000,  count: 21
Region Avg IOU: 0.069586, Class: 1.000000, Obj: 0.234530, No Obj: 0.417606, Avg Recall: 0.000000,  count: 1
Region Avg IOU: 0.049137, Class: 1.000000, Obj: 0.217546, No Obj: 0.417337, Avg Recall: 0.000000,  count: 12
Segmentation fault (core dumped)
```

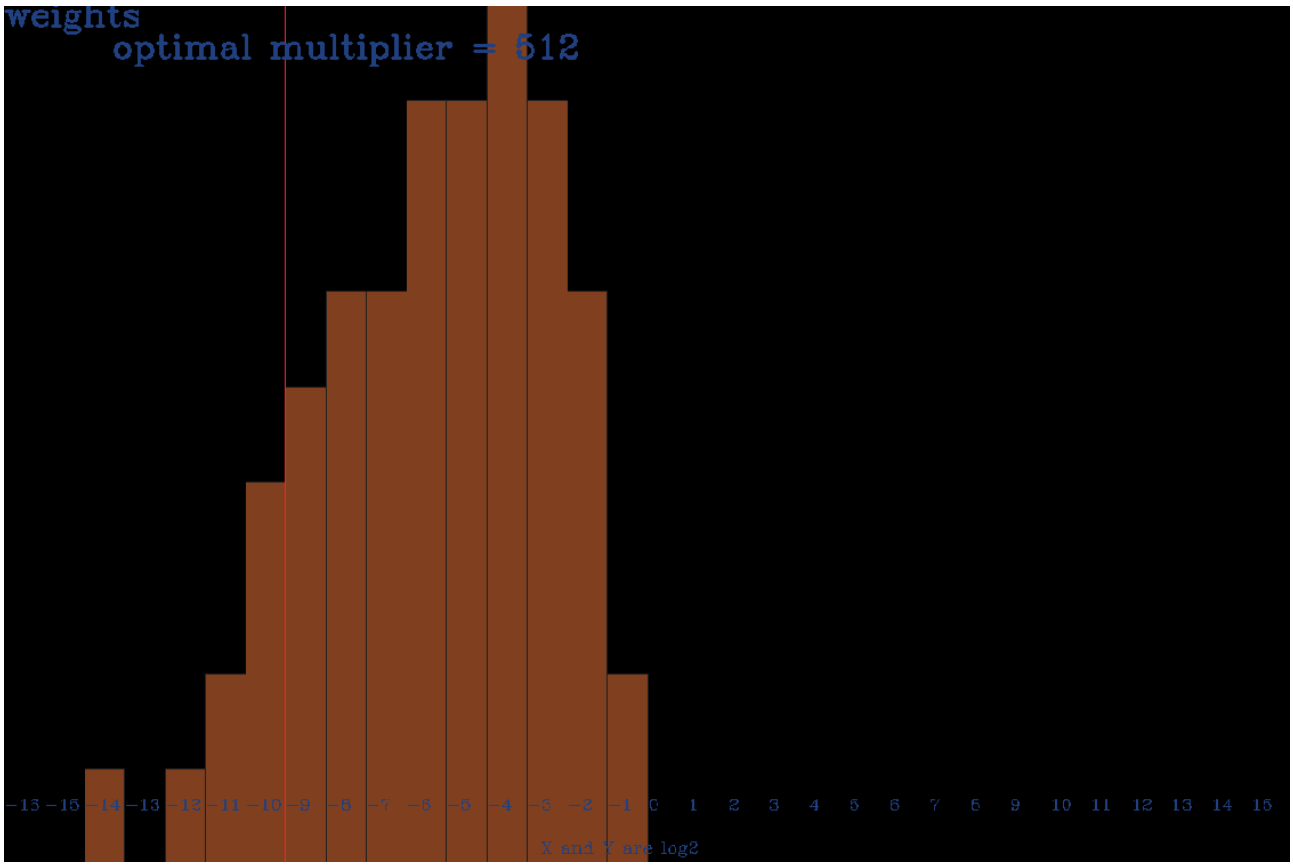
filename = /media/brian/FILE/FD/dataset/WIDER_train/labels/0—Parade/0_Parade_Parade_0_452.txt
this file repeats every time, so call this a human big data analasyse !

Need to change the wider_2_yolo.py file which converts widerface datasets to yolo datasets.

---
should add a visualization option to monitor the loss, it is actually quite easy, just print out the average loss value every several batches. And then plot the char vs time in python. If the loss is not going down with time pass by, there must be something wrong in the code, but since darknet is a good deep learning framework,should this shouldn't happened, but for your own deep learning framework,you actually need to do gradient check and loss value monitoring.
There is actually not much time to analyse for the output terminal parameters, it is quite straight froward, the current loss value and average loss value vs time,
for human interpretation it is quite straight forward to compare your prediction vs ground truth, as listed below

in terms of machine learning language we use loss function and loss value, remember that the lost value here includes several terms :
objectness,class,location and size….
Off course, the learning rate is critical as well, we need to print out that information as well.
What else information do we need ?
IOU !
Recall := TP /(TP+NP)
Precision : TP/all_prediction
too bad I forget the chart again, need go back and check my notes out !

Quantinization!
How should we quantize the weights ???
1. observe the distribution of the weights .



2. approximate_true_value = scale * quantized_value + offset

$Y_{aproxi} = \gamma_{scale} * X_{quantized} + \beta_{offset}$ Totally WRONG !!! what is the point of doing this ? The aim of quantization is to reduce the weights from 32-bits to 8-bits !

What is the intermediate bit width we need to store the multiplication between inputs and weights ?

Will it cause overflow ? How to prevent it or give a alarm when overflow happens ?

How to apply this to FPGA design ?

The first stage of input is color image which is ranging from 0-255, hopefully the weights of all stages are less than 1.0f, otherwise overflow might happen !

So need to examine all the intermediate values, see if the weights **are all less than 1.0f** and **not too small as wel**l, otherwise a scale and shift value will be needed ????

the procedure :

1. get all the weights and intermediate values printed out ! Show their distribution !
2. if all values are less than 1.0f and not too small ?
3. change float to int
4. check the result ! If it works on computer and it should works on FPGA as well !

Question :

1. when deriving distribution and multiplier,why number_of_ranges is 32 ?
2. what is multiplier and what for ?

Whatever, I don't have to strictly follow the author's quantization method which seems to be not quit correct,for instance the W_MAX_VAL value is not quite correct.

I can quantize the weights in my own way, just use his code structure !
Using it as a matlab !
1. for each layer, the normalization factor is not the same, in order to best span the weights among the 8bit bandwidth !
1. so from layer to layer, there should be a different scale factors .

Quantization steps :
1. get distribution of the weights, calculate the mean and deviation .
2. shift the weights to be zero mean
3. normalize the weights with deviation
4. multiplication with 2^8/2 = 128, making it int8
5. store the weights in DDR

Inference process :
1. stream them from DDR
2. undo the scale and shift to get the original weights. In case overflow, should make sure the weights are all less than zero, in most case it should be, other wise something may be potentially wrong, it might cause overflow . What is the data range of intermediate values : less than 1.0 ? bigger than 1.0 ? what is the maximum value it will hit ?
3. convolute weights with input features
4.

Data exploration :
1. print weights in each layer to a txt file .
2. print every intermediate value to a txt file
3. analyze the data in python, get a picture of the mean , deviation and maximum and minimum .

Where should I print the data to file, in the loading weights segment, treat weights and batch_normalization differently !

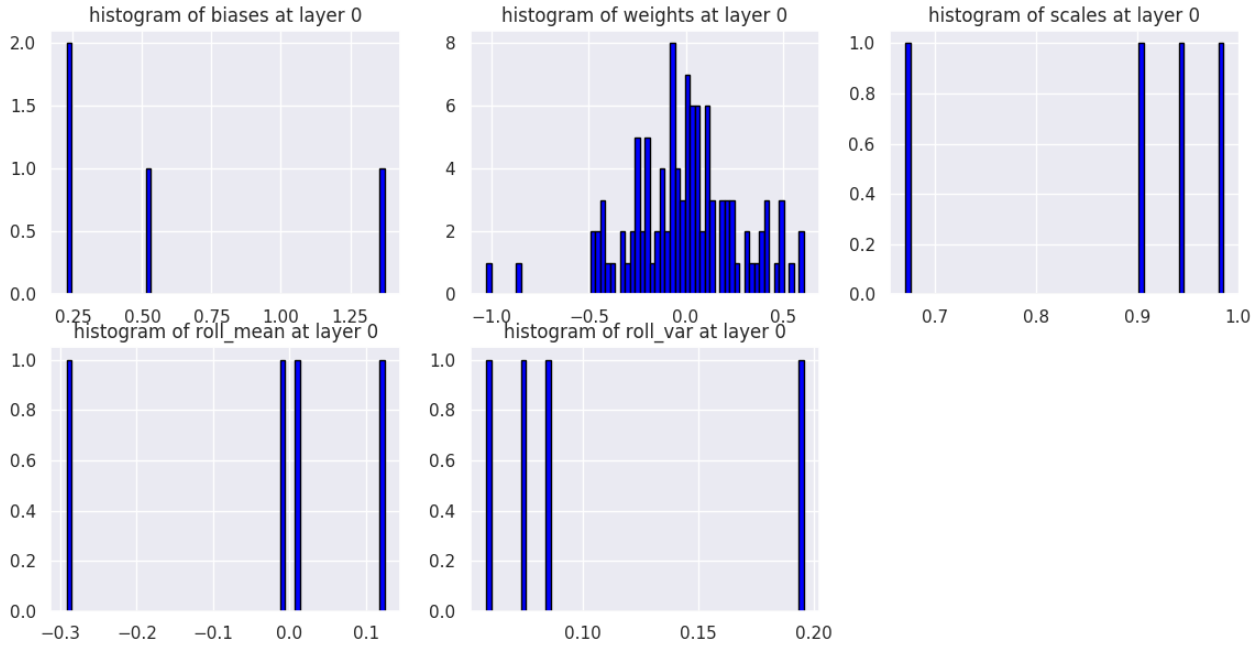
```
3141  // parser.c
3142  void load_weights_upto_cpu(network *net, char *filename, int cutoff)
3143  {
3144  #ifdef GPU
3145      if (net->gpu_index >= 0) {
3146          cuda_set_device(net->gpu_index);
3147      }
3148  #endif
3149      fprintf(stderr, "Loading weights from %s...", filename);
3150      fflush(stdout);
3151      FILE *fp = fopen(filename, "rb");
3152      if (!fp) file_error(filename);
3153
3154      int major;
3155      int minor;
3156      int revision;
3157      fread(&major, sizeof(int), 1, fp);
3158      fread(&minor, sizeof(int), 1, fp);
3159      fread(&revision, sizeof(int), 1, fp);
3160      /*
3161         Brian: which means the first 3 int are major,minor,and revision
3162      */
3163      printf("major = %d\n",major );
3164      printf("minor = %d\n",minor );
3165      printf("revision = %d\n",revision );
3166
3167      /*
3168          Brian: the version compatiblity
3169          *net->seen = the total # of weights going to read ?
3170      */
3171      if ((major * 10 + minor) >= 2) {
3172          fread(net->seen, sizeof(uint64_t), 1, fp);
3173      }
3174      else {
3175          int iseen = 0;
3176          fread(&iseen, sizeof(int), 1, fp);
3177          *net->seen = iseen;
3178      }
3179      //int transpose = (major > 1000) || (minor > 1000);
3180
3181      /*
3182          net->n :  # of layers
3183      */
3184      int i;
3185      for (i = 0; i < net->n && i < cutoff; ++i) {
3186          layer l = net->layers[i];
3187          if (l.dontload) continue;
3188          if (l.type == CONVOLUTIONAL) {
3189              load_convolutional_weights_cpu(l, fp);
3190          }
3191      }
3192      fprintf(stderr, "Done!\n");
3193      fclose(fp);
3194  }
3195
```

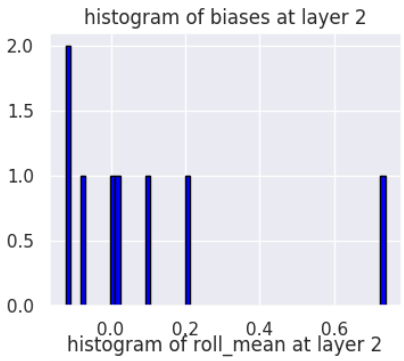
Layout of weights in each layer :

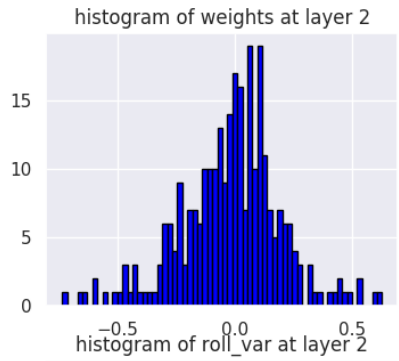
layout of weights :

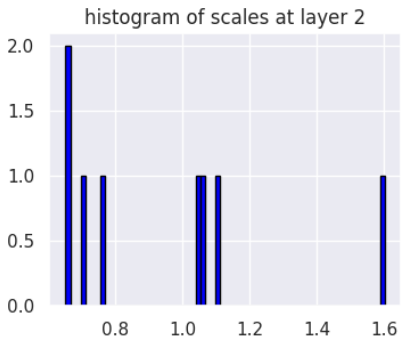
1.major (int)
2.minor (int)
3.revision (int)
4.total # of weights (uint64_t or int depending on version #)
5.biases (l.n(channel #) * sizeof(float))
6.scales (l.n(channel #) * sizeof(float))
7.rolling_mean (l.n(channel #) * sizeof(float))
8.rolling_variance (l.n(channel #) * sizeof(float))
9.conv_weights (num * sizeof(float))

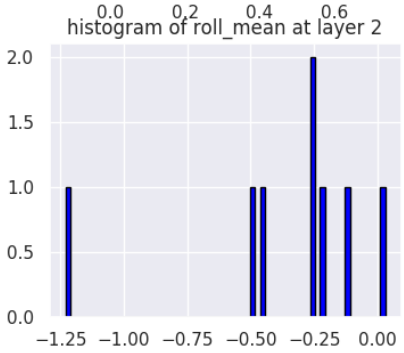
```
3109  /*
3110      layout of weights :
3111          major (int)
3112          minor (int)
3113          revision (int)
3114          total # of weights (uint64_t or int depending on version #)
3115          biases (l.n(channel #) * sizeof(float))
3116          scales (l.n(channel #) * sizeof(float))
3117          rolling_mean (l.n(channel #) * sizeof(float))
3118          rolling_variance (l.n(channel #) * sizeof(float))
3119          conv_weights  (num * sizeof(float))
3120  */
3121
3122  // parser.c
3123  void load_convolutional_weights_cpu(layer l, FILE *fp)
3124  {
3125      int num = l.n*l.c*l.size*l.size;
3126      fread(l.biases, sizeof(float), l.n, fp);
3127      if (l.batch_normalize && (!l.dontloadscales)) {
3128          fread(l.scales, sizeof(float), l.n, fp);
3129          fread(l.rolling_mean, sizeof(float), l.n, fp);
3130          fread(l.rolling_variance, sizeof(float), l.n, fp);
3131      }
3132      fread(l.weights, sizeof(float), num, fp);
3133      /*    if (l.adam) {
3134      fread(l.m, sizeof(float), num, fp);
3135      fread(l.v, sizeof(float), num, fp);
3136      }
```

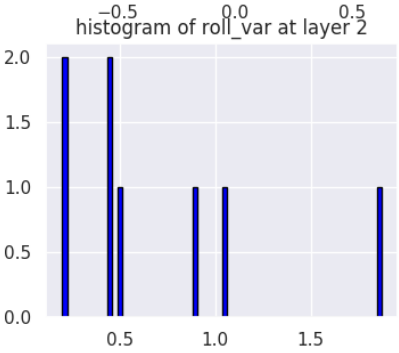

histogram of biases at layer 0
histogram of weights at layer 0
histogram of scales at layer 0
histogram of roll_mean at layer 0
histogram of roll_var at layer 0

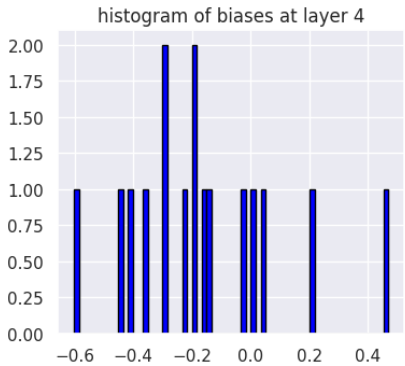

histogram of biases at layer 2

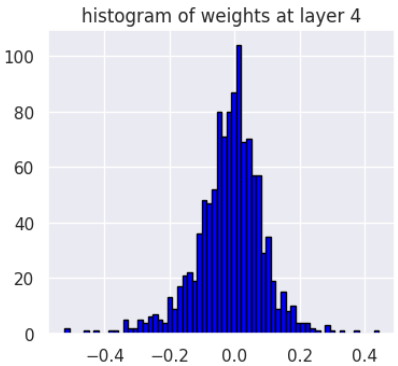

histogram of weights at layer 2

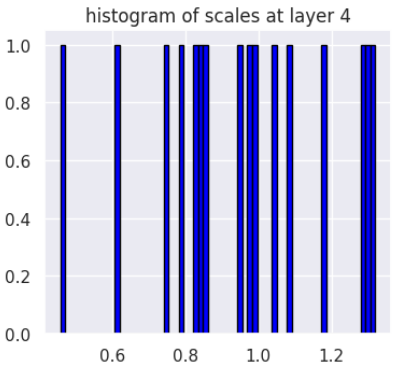

histogram of scales at layer 2

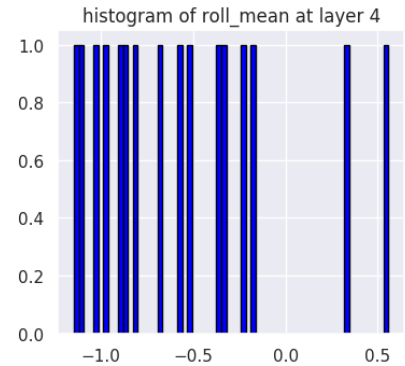

histogram of roll_mean at layer 2

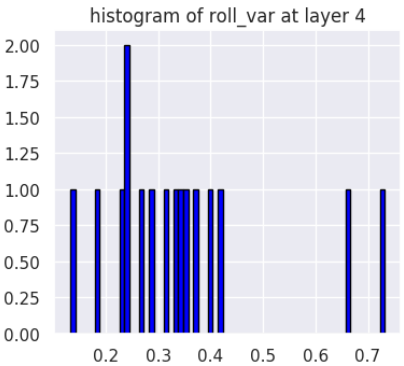

histogram of roll_var at layer 2

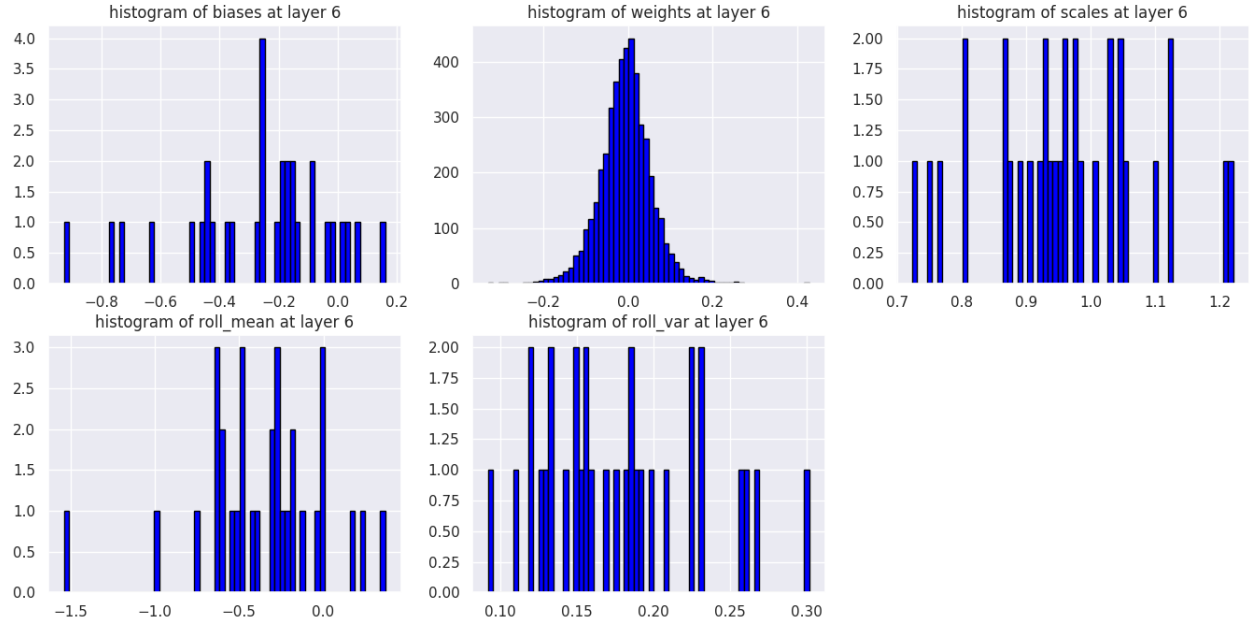

histogram of biases at layer 4

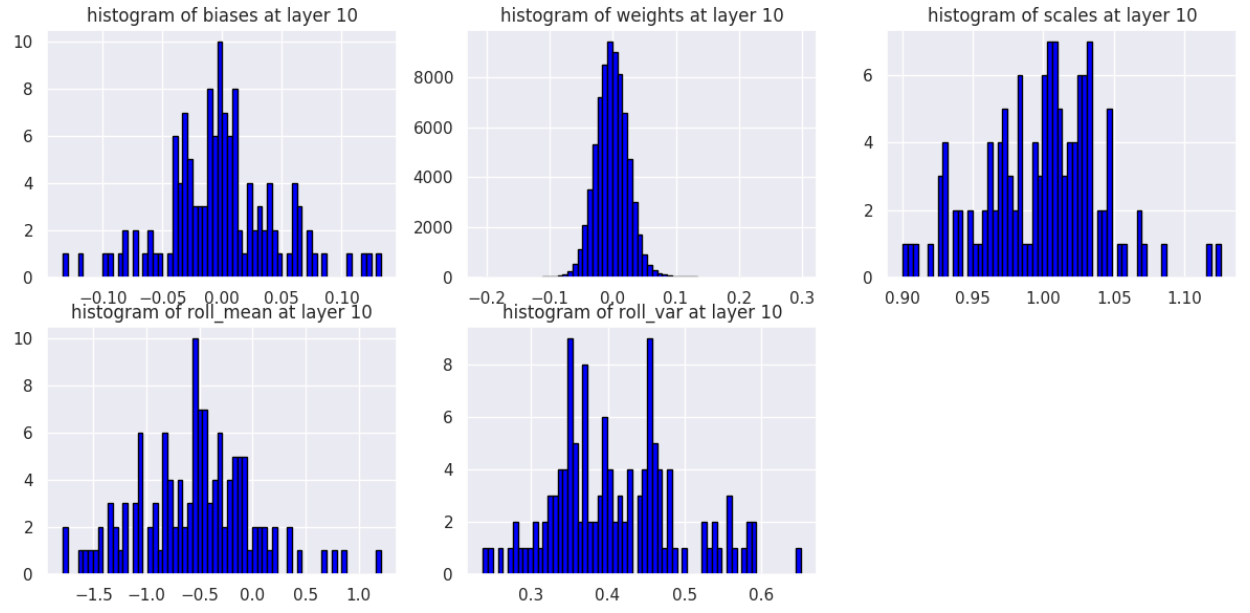

histogram of weights at layer 4

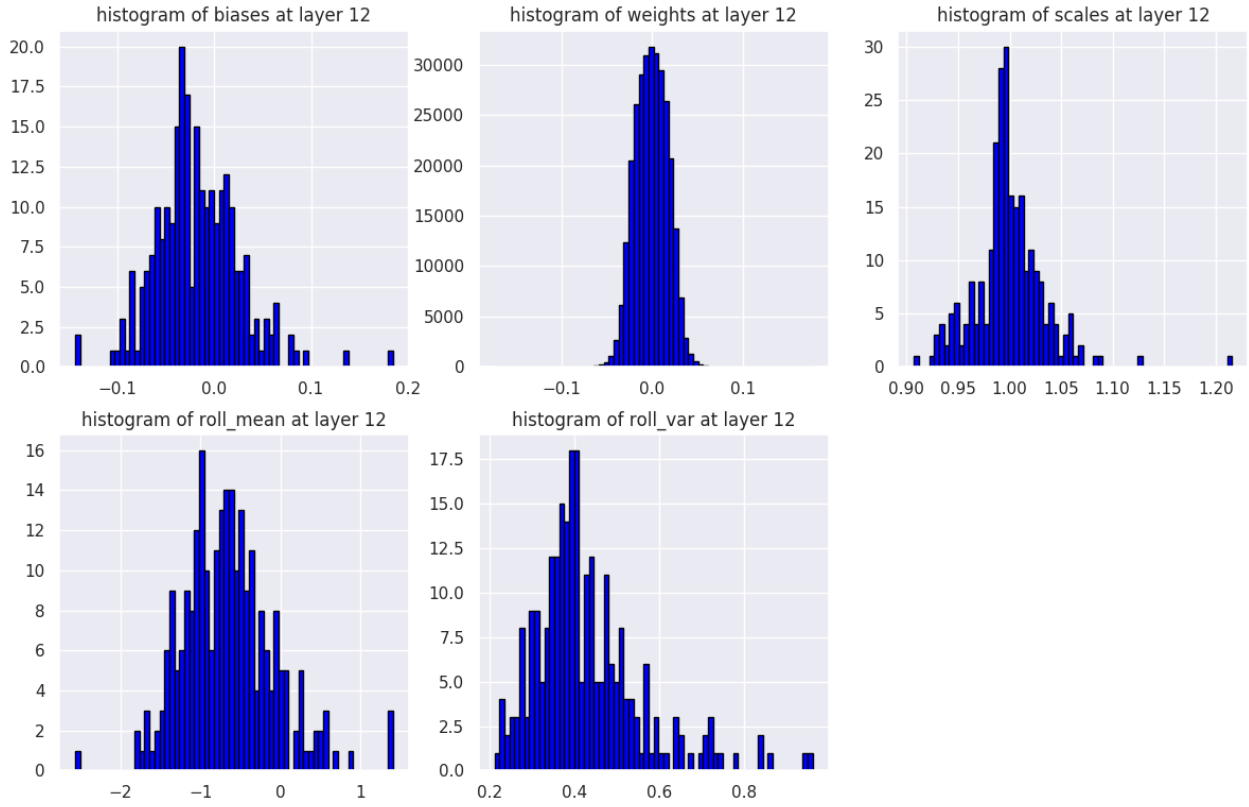

histogram of scales at layer 4

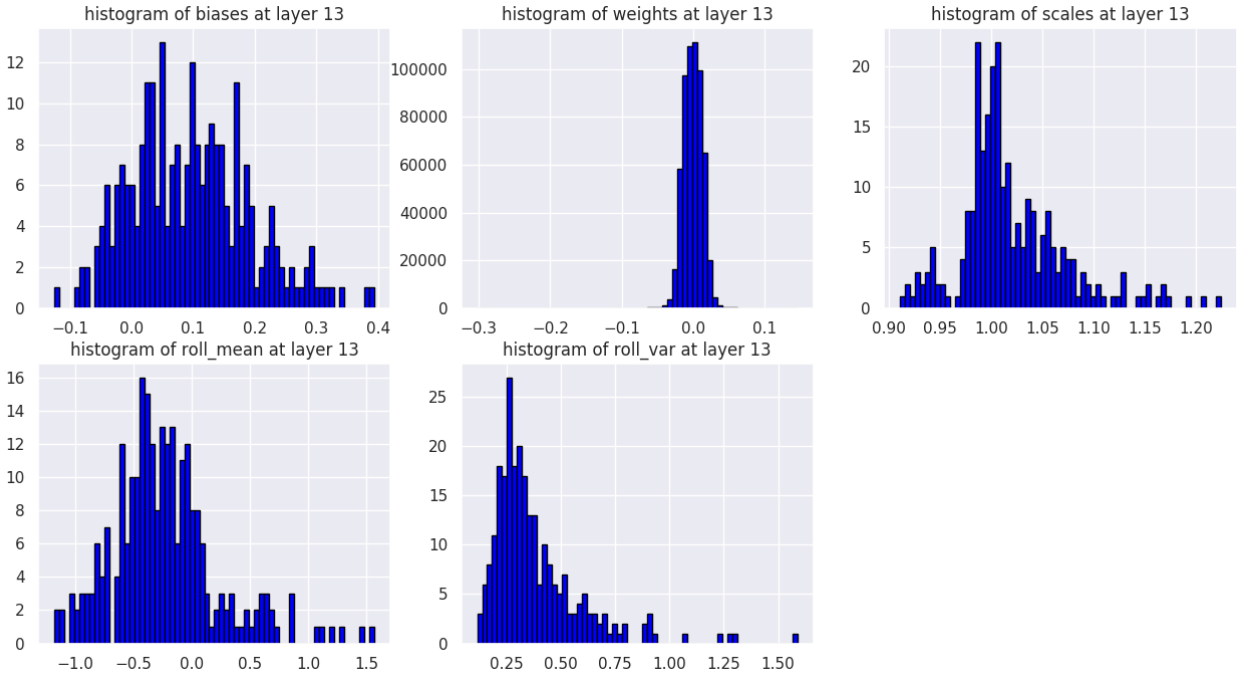

histogram of roll_mean at layer 4

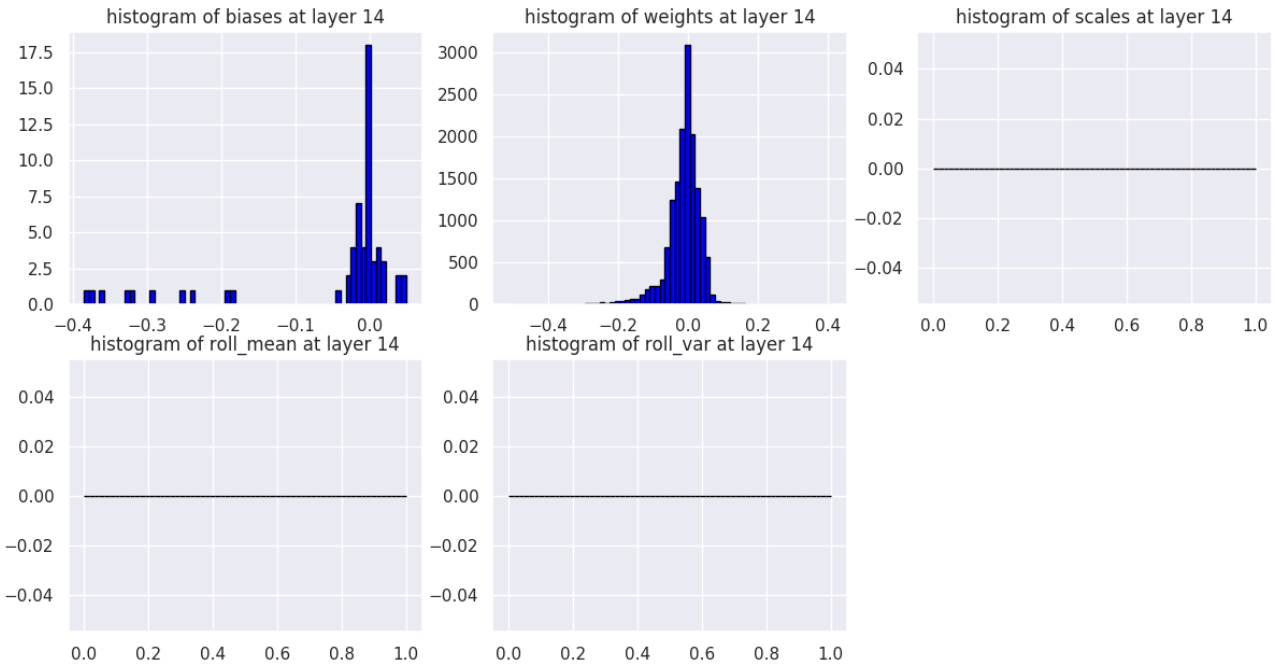

histogram of roll_var at layer 4


histogram of biases at layer 6
histogram of weights at layer 6
histogram of scales at layer 6
histogram of roll_mean at layer 6
histogram of roll_var at layer 6


histogram of biases at layer 10
histogram of weights at layer 10
histogram of scales at layer 10
histogram of roll_mean at layer 10
histogram of roll_var at layer 10


histogram of biases at layer 12
histogram of weights at layer 12
histogram of scales at layer 12
histogram of roll_mean at layer 12
histogram of roll_var at layer 12


histogram of biases at layer 13
histogram of weights at layer 13
histogram of scales at layer 13
histogram of roll_mean at layer 13
histogram of roll_var at layer 13



above are the distributions of floating point biases (beta), weights, scales(gamma), rolling mean and variance
Batch Normalization :

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=0}^{M-1} x_i \qquad \text{Batch size = M}$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=0}^{M-1} (x_i - \mu)^2$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + e}} \qquad y_i \leftarrow \gamma \hat{x}_i + \beta$$

to print all the distribution of each layer's output data is kind of tedious and difficult !
Now I need to change the yolo code in my own way. And verify if the idea is good !

1. $\hat{x}_i \leftarrow \frac{x_i - \mu}{\sqrt{\delta^2 + e}}$ convert the original dataset to zero mean and variance of 1
2. quantization to int8_t
3. reverse back to original data value in int format to approximate original floating value
4. do convolution
5. convert the convolution output to floating value, dividing by normalization factor !

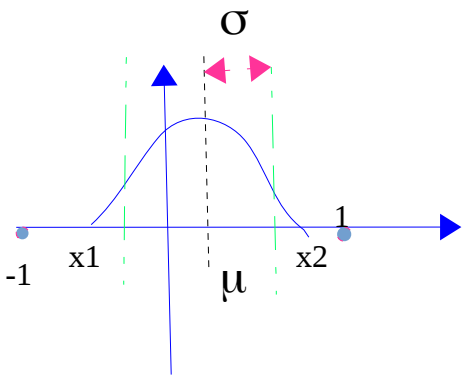
let's try it tomorrow !

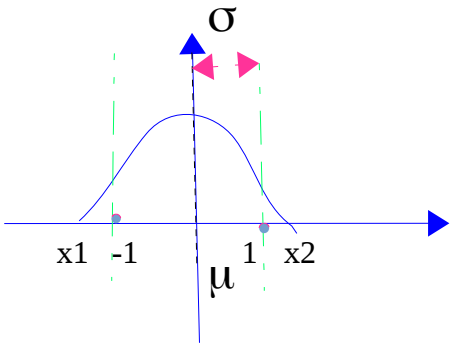
input_quant_multipler ???
why the hell is 40 ? float input * input_quant_multipler = quantized input in int. derive it from the input distribution ? Then each level should have different input_quant_multipler !

Take the feature map as a image,that is why quantization still works when multiplying different scales. Multiplying with different scales does not change the visual effect !

Ok,so I have figured it out that, it is not complicated,but the code is making things complicated, the overflow is simple as follows :
1. scale float input by input_multiplier to int8
2. scale weights by weight_multiplier to int8
3. convolution
4. down scale the convolution output by (input_multiplier * weight_multiplier) to original float value.

How we calculate the quantization multiplier :
a. lazy way :
quantization factor = $|\mu| + \sigma$





b. better way
quantization factor = σ

Problem and observation :
1. too many very small values in convolution weights, using 3*sigma to normalize is wasting too many bits for the small weights that are not important !
2. should set small weights to zero, then recalculate the sigma value
3. normalize with the new sigma value.
4. how to quantize the bias, which tend to be a little bit larger than weights !
5. can not quantize them with their own sigma value separately !
6. how many fractional bits and integer bits for weights
7. how many fractional bits and integer bits for bias, if normalized with weights sigma value .
Weights and biases should be normalized with the same factor !

Trial flow :
1. set the small weights to zero,see if it affects the output ? What defines “small”,
2. calculate the mean and sigma without zeros
4. normalization with 3*sigmas to weights and biases
5.

amazing output : set threshold to 0.01 still get quit good result !
Reducing the size of the weights does not help too much for device like FPGA, since they are streaming. Maybe for arm device,but need to load data layer by layer so better use the cache !
But it can help the normalization and quantization steps ! Reduce the quantization errors, better use the bit width .

So tomorrow can try float normalization first then write your own code for quantization .
yolov2_fuse_conv_batchnorm

from the pruning tests, I get a conclusion : bigger values are more important for small values .

Why my quantization did not work right ?
Write the convolution code myself !
So after writing your own convolution code, and rewriting the quantization code in a separate module,I have found the bug, {} is wrong,so should be very careful with the for loop and {}.
try not to use this code style :

```
if (xxx){

}
for(xxx){

}
```
stick to this style :
```
if (xxx)
{

}
for (xxxxx)
{
	if(xxx)
	{

	}
}
```
so you would not make mistakes that easy !

Observations :
1 . weights and bias normalized by the 3-sigma value does not work. But normalization by their maximum value works. Which means the bigger value in weights and biases are more important than the small values .
2. the output at each layer normalized by their 3-sigma value works, which means even if you truncate the output value it still works somehow . The normalization factor should come from the training stage like rolling mean and rolling variance .
3. I think no need to normalize the output, you can actually replace the relu with an thresholded relu activation function and retrain the network,such that the output is bounded, which is easy for quantization .
4. also the weights and bias can also do quantization training .

Quantization FPGA flow :

……

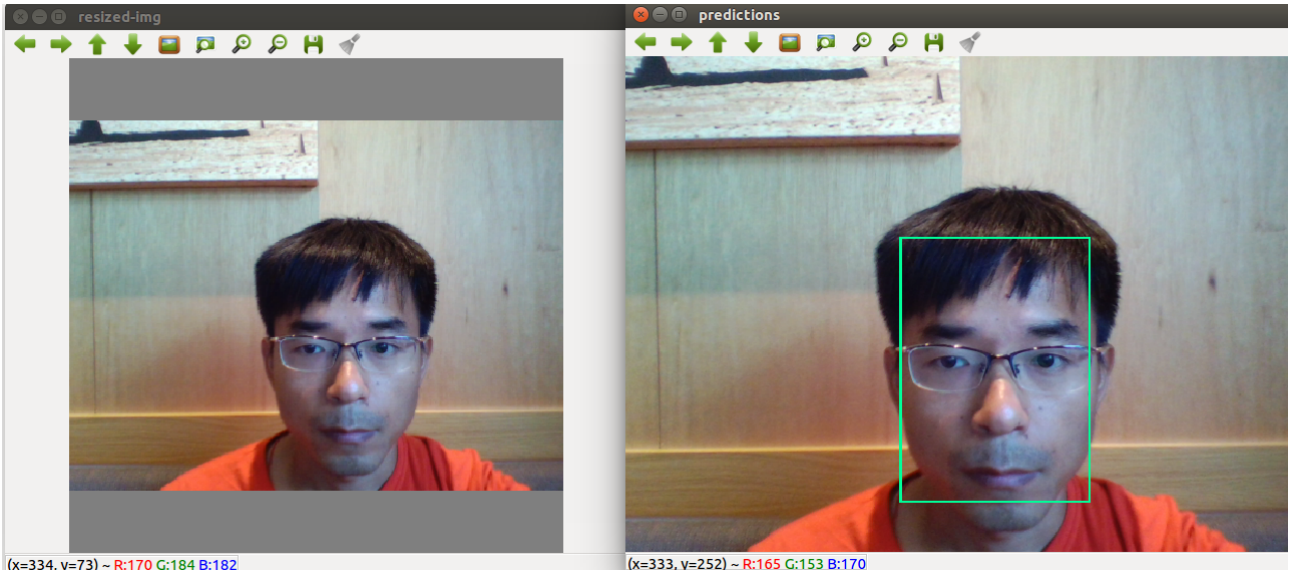
so much for quantization, it does not speed up much for embedded device. Next stage is acceleration via NEON ! Or SSE . Read the nnpack code . Try to rewrite it .
Or maybe rewrite the code in opencl,so that it can run on device apart from nvidia .

---

nnp_convolution_inference this is the API darknet_nnpack uses .
winograd_f6k3 figure out the details of this algorithm then write neon code (try assembly)

- **關於圖像輸入大小**

在配置 cfg 文件中設定大小爲 416×416。代碼
image sized = letterbox_image(im, net->w, net->h);
處理過程：先將圖像等比例縮小，這樣圖像不至於變形，不足部分填充其他顏色，使之成爲 416×416。

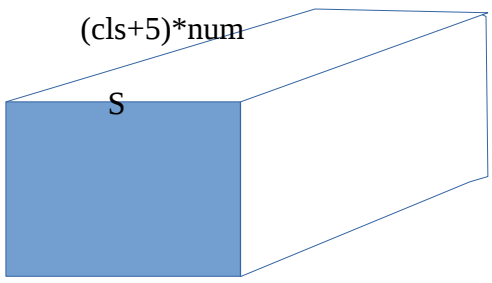


瘦猴子，你一定不能輸！加油！打贏這場不對稱的戰爭！！！好好地活着，健康地活着，快樂地活着，自信地活着！

2019 五月四日
自去年 11 月份，身體一直在走下坡路，中斷了大半年，忘光了。
what is next ?
不要把 darknet 當成黑匣子，這個誰都會。把代碼重新寫一遍。
Backpropogation 那一部分要重新寫麼？還是只寫 forward 部分，然後優化到 FPGA 上去?
要學就學徹底，一步一步來。
1. backpropogation
2. quantization
3. FPGA
我就用 C 和 C++，tensorflow？ what's the point ? 我就把 darknet 吃個爛熟，以後所有的模型都從這裏設計，包括 backpropogation。工具在精不在多。雖然 tensorflow 在設計算法的時候可能是比較方便，但我的目的不止於算法，更重要的是實現。我要是把 darknet 的 backpro 也吃個爛熟，就不愁以後自己設計模型了。 就這麼定了。你要是連這些代碼都沒寫過，別人問你深度學習學的如何？難道就說你只會用黑匣子？這個有什麼難度?
程序員就是要不停地寫代碼。不寫代碼，手生疏!

第一步，最後一層的 backpropogation 代碼的改寫。forward_region_layer
先看原作者的，後再看另外一個人改寫的。

$$\frac{\partial J}{\partial w_i}$$



每個 bbox 包含幾個參數：
1. box 相關的，w,h,x,y,以及 objness.
   l.delta[box_index]
   l.delta[obj_index]

2. class numbers
   l.delta[class_index]
S
所以總共的參數個數爲： S*S*(cls+5)*num
也是最後一層輸出的 output 個數，l.outputs

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]$$

$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2$$

$$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2$$

$$+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2 \qquad (3)$$

注意下標 i is the index of grid cell,j is bbox . 這個代價函數是 yolov1 的，在 yolov2 中，class 部分的代價函數爲 Pij(c),即在第 i 個 gridcell 中的每個 bbox 都有 C 個 class prob（c）

$$\frac{\partial J}{\partial \hat{C}_i} = \lambda_{noobj} * (0 - \hat{C}_i) * 2 \quad \text{for bbox containing no objects}$$

```
l.delta[obj_index] = l.noobject_scale * (0 - l.output[obj_index]);
```

$$\frac{\partial J}{\partial \hat{C}_i} = \lambda_{obj} * (1 - \hat{C}_i) * 2 \quad \text{for bbox containing objects}$$

```
l.delta[obj_index] = l.object_scale * (1 - l.output[obj_index]);
```

$$\frac{\partial J}{\partial \hat{p}_{ij}(c_n)} = \lambda_{obj} * (1 - \hat{p}_{ij}(c_n)) * 2 \quad for \;\; p_{ij}(c_n) = 1, c_n \le numofClasses$$

$$\frac{\partial J}{\partial \hat{p}_{ij}(c_n)} = \lambda_{obj} * (0 - \hat{p}_{ij}(c_n)) * 2 \quad for \;\; p_{ij}(c_n) = 0, c_n \le numofClasses$$

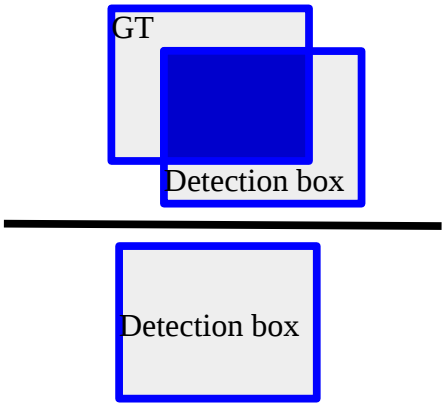
在代價函數的設計上，個人覺得應該把 training 分成兩部分。
1. 先 train 畫 bbox 的位置，大小，objness。detection training。
2. 在這個基礎上再聯合 train，classification，
如果兩個同時 train 會比較難達到收斂。因爲 classification 的 training 會給 detection 的 training 帶來很大的 jitter 噪聲，使得 detection 的他熱愛寧都可能不會收斂。這也是爲啥作者要提供 pretrain 的系數的原因。那個 pretrain 的系數就是 detection 的權重。

所以此處我們只需計算有 objness 的 bbox 的 pij(cn)的反向傳遞函數,總共 n#ofClass 個。計算沒有 objness 的 bbox 的傳遞函數沒有意義。
奇怪爲啥這段代碼裏沒找到對坐標和 bbox 大小的反向傳導函數?
豬，在 delta_region_box 的函數裏，delta_objness 沒有獨立封裝成函數。
delta_region_class 就是應該是對有 objness 的 bbox 的 class prob 的傳導了。

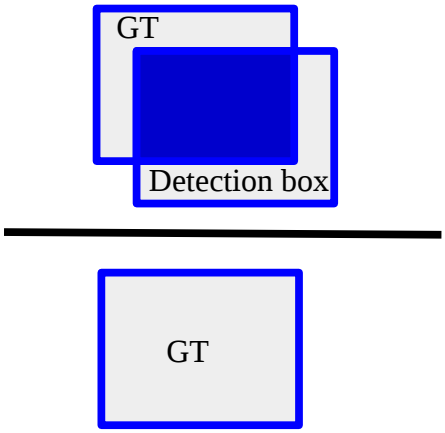
detection 的反向傳導比較簡單，但也是整個論文的很重要的部分，涉及到代價函數的理解！！！
下一步就是 convolution 層和 batchnorm 的反向傳導函數的處理了。公式自己以前推導過，根據公式看代碼，對比下是否推導正確。
然後根據公式和自己對代碼的理解，重新寫一遍代碼，做一個屬於自己的深度學習框架。這樣對深度學習就更加理解透徹了。然後根據自己的平臺，搭建 openpose 等等（後面再說，先實現量化和枝剪，FPGA 實現）。
還有一點，怎麼計算那些曲線呢？怎麼畫 cost 曲線，然後決定 training 停止？看另外一個人的代碼。
另外根據自己的理解先在自己的電腦上從零開始 train 一個 mnist，不要用 pretrain 的 weights，把模型簡化一下，做成輕量的。開始 train 的時候把代價函數中 classification 的部分屏蔽掉，待 objectdetection 訓練成功之後再啓用它，再 train。或者說做個動態的 cost 函數的權重。

五月六日
schedule:
1. AlexeyAB 的 forward_region 部分的代碼，分析如何畫曲線。
2. backpropagation on convolution and batchnorm 代碼分析。
3. 倘若還有時間，開始一個模塊一個模塊地改寫代碼。如量化，枝剪等等。
如何計算 mAP ? Mean Average Precision ?
弄清楚兩個概念:

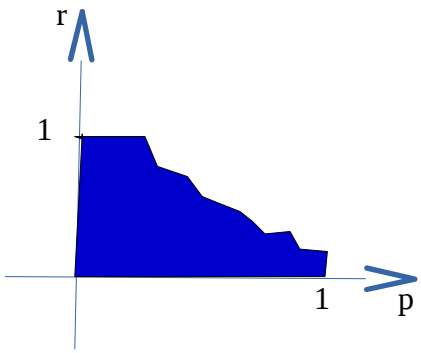
1. $$Precision = \frac{TP}{TP+FP}$$

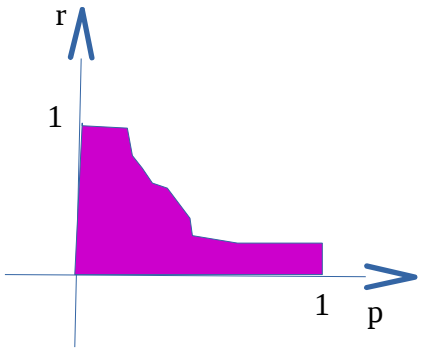


2. $$Recall = \frac{TP}{TP+FN}$$



precision 和 recall 是個 trade off 關系， detection box 大了，recall 高，但 precision 就低了。
如何計算 mAP?
計算每張 training 圖片的各個 class 的 n 個 precisions 和 recalls，即爲 n 個點(precision_value,recall_value),若 training 圖片有 M 張，可得 N 個（p,r)坐標點，將這些點連成一條曲線，曲線下的面積即爲此 class 的 AP，將 K 個 class 的 AP 做個平均，即爲 mAP。
如何計算曲線下的面積？方法很多，業界有統一的計算方法。細節現在先不去深究。





計算這部分的代碼，無需在訓練的 C 代碼中實現，另外寫個 python 的代碼來實現。在 C 代碼中，只需要在 test 的時候輸出每個 class 的(r,p)坐標到 txt 文件。用 python 來計算描點畫圖和計算曲線下的 AP。

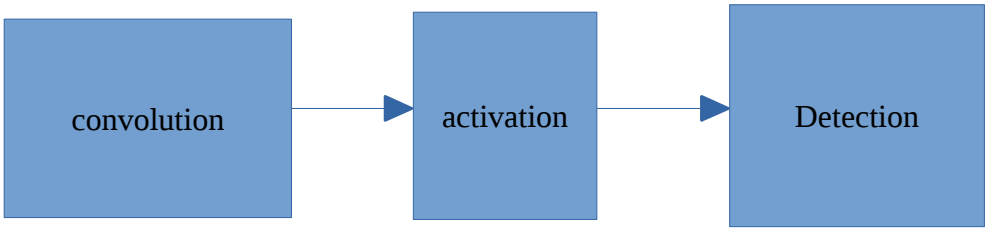
[烏龜]

這是一只
深藏的烏龜
一趴就是千年
從小烏龜
變成
大烏龜
再成
老烏龜
紋絲不動
修煉成
一座道德的山峯

它啊
兩眼總是泛着
善良的淚水
真情地哭着說
對不起
我～
來晚了

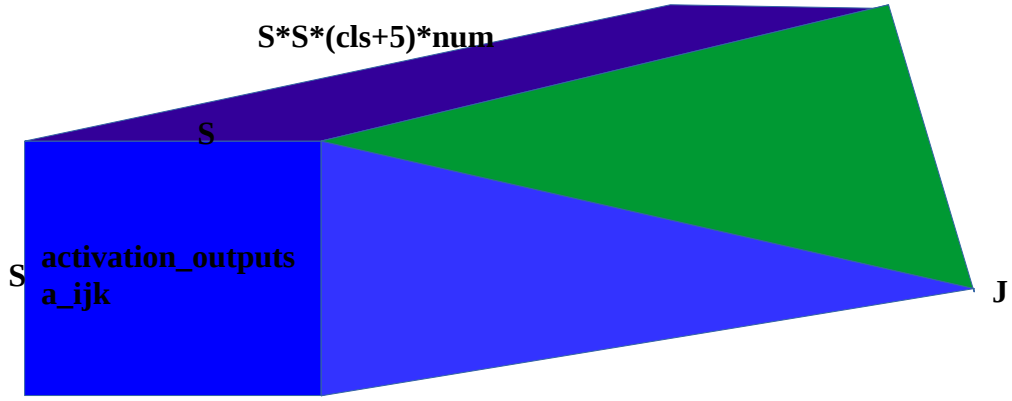
void backward_convolutional_layer(convolutional_layer l, network net)

```
void backward_convolutional_layer(convolutional_layer l, network net)
```




```
/*
    l.output is the output of activation layer
    l.delta is the back-propagation holder
*/
gradient_array(l.output, l.outputs*l.batch, l.activation, l.delta); //  back-propagation on activation layer
```

for the last convolution layer,



$$\delta_{ijk}^{DetectionInput} = \frac{\partial J}{\partial a_{ijk}^{out}} \quad i<S, j<S, k<S*S*(cls+5)*numOfbbox$$

$$\delta_{ijk}^{ActivationInput} = \delta_{ijk}^{DetectiontionInput} * \frac{\partial a_{ijk}^{out}}{\partial BNorm_{ijk}^{out}} \quad i<S, j<S, k<S*S*(cls+5)*numOfbbox$$

所以此處的 gradient_array 是對 activation 的求導，對於 relu 的 activation 函數，activation input >0 時，導數爲 1,detection 的 delta 值直接往前一層傳遞，input <0 的，往前一層傳遞 0 值。

```
void backward_batchnorm_layer(layer l, network net)
```

Back-pro at BN

$$\mu_k = \frac{1}{B*H*W}\sum_{b=0}^{B-1}\sum_{i=0}^{H-1}\sum_{j=0}^{W-1} C_k^l(b,i,j)$$

$$\hat{C}_{k,b,i,j}^l = \frac{C_{k,b,i,j}^l - \mu_k}{\sqrt{\sigma^2+e}}$$

$$\sigma_k^2 = \frac{1}{B*H*W}\sum_{b=0}^{B-1}\sum_{i=0}^{H-1}\sum_{j=0}^{W-1} (C_k^l(b,i,j)-\mu_k)^2$$

$$y_{k,b,i,j} = \gamma_k * \hat{C}_{k,b,i,j}^l + \beta_k$$

Given: $\delta_{k,b,i,j}^l = \frac{\partial Loss}{\partial y_{k,b,i,j}^l}$

```
backward_bias(l.bias_updates, l.delta, l.batch, l.out_c, l.out_w*l.out_h);
```

$$\frac{\partial Loss}{\partial \beta_k^l}$$

$$\frac{\partial J}{\partial \beta_k} = \sum_b \sum_i \sum_j \frac{\partial J}{\partial y_{k,b,i,j}} * \frac{\partial y_{k,b,i,j}}{\beta_k}$$

$$= \sum_b \sum_i \sum_j \frac{\partial J}{\partial y_{k,b,i,j}} * 1$$

下面這段代碼即實現上面的公式。把當前 output channel index n 的所有 delta 值相加，即爲此 output channel 的偏置值的 update。

```
{
    int i,b;
    for(b = 0; b < batch; ++b){
        for(i = 0; i < n; ++i){ // n is the output channel index
            bias_updates[i] += sum_array(delta+size*(i+b*n), size);
        }
    }
}
```

```
backward_scale_cpu(l.x_norm, l.delta, l.batch, l.out_c, l.out_w*l.out_h, l.scale_updates);
```

$$\frac{\partial Loss}{\partial \gamma_k^l}$$

$$\frac{\partial J}{\partial \gamma_k} = \sum_b \sum_i \sum_j \frac{\partial J}{\partial y_{k,b,i,j}} * \frac{\partial y_{k,b,i,j}}{\partial \gamma_k}$$

$$= \sum_b \sum_i \sum_j \frac{\partial J}{\partial y_{k,b,i,j}} * \hat{C}^l_{k,b,i,j}$$

```
void backward_scale_cpu(float *x_norm, float *delta, int batch, int n, int size, float *scale_updates)
{
    int i,b,f;
    for(f = 0; f < n; ++f){   // f is the input channel index
        float sum = 0;
        for(b = 0; b < batch; ++b){
            for(i = 0; i < size; ++i){ // size = w*h
                int index = i + size*(f + n*b);
                sum += delta[index] * x_norm[index];
            }
        }
        scale_updates[f] += sum;
    }
}
```

對 batch-norm 輸入的求導數，用於計算前一級的 delta map . 詳細推導過程見 techshare 的 pdf 文件。

$$\frac{\partial Loss}{\partial C^l_{b,i,j}} = \delta^l(b,i,j)*\gamma*\frac{1}{\sqrt{\sigma^2+e}} + \frac{1}{B*N*M}*\sum_{b'=0}^{B-1}\sum_{i'=0}^{H-1}\sum_{j'=0}^{W-1}\gamma*\delta^l(b',i',j')*\frac{-1}{\sqrt{\sigma^2+e}}$$

$$+\frac{1}{B*N*M}*2*(C^l_{b,i,j}-\mu_{ch})*\frac{0.5}{(\sigma^2+e)^{\frac{-3}{2}}}*\sum_{b'=0}^{B-1}\sum_{i'=0}^{H-1}\sum_{j'=0}^{W-1}\gamma*\delta^l(b',i',j')*(C^l_{b',i',j'}-\mu_k)$$

```
// scale_bias(l.delta, l.scales, l.batch, l.out_c, l.out_h*l.out_w);
void scale_bias(float *output, float *scales, int batch, int n, int size)
{
    int i,j,b;
    for(b = 0; b < batch; ++b){
        for(i = 0; i < n; ++i){  // input channel index
            for(j = 0; j < size; ++j){
                output[(b*n + i)*size + j] *= scales[i];
            }
        }
    }
}
```

這部分代碼計算 每個 delta(b,i,j)*gamma $\delta^l_k(b,i,j)*\gamma_k$ 上面公式省略了下標 k，k 爲 channel 的 index。

執行完此代碼，l.delta_k(b,i,j)包含的值爲 $\delta^l_k(b,i,j)*\gamma_k$

```
// mean_delta_cpu(l.delta, l.variance, l.batch, l.out_c, l.out_w*l.out_h, l.mean_delta);
void mean_delta_cpu(float *delta, float *variance, int batch, int filters, int spatial, float *mean_delta)
{

    int i,j,k;
    for(i = 0; i < filters; ++i){
        mean_delta[i] = 0;
        for (j = 0; j < batch; ++j) {
            for (k = 0; k < spatial; ++k) {
                int index = j*filters*spatial + i*spatial + k;
                mean_delta[i] += delta[index];
            }
        }
        mean_delta[i] *= (-1./sqrt(variance[i] + .00001f));
    }
}
```

此代碼實現的是 mean_delta

$$\sum_{b'=0}^{B-1}\sum_{i'=0}^{H-1}\sum_{j'=0}^{W-1}\gamma*\delta^l(b',i',j')*\frac{-1}{\sqrt{\sigma^2+e}}$$

注意 mean_delta 的維度，1×#_of_filters

```
// variance_delta_cpu(l.x, l.delta, l.mean, l.variance, l.batch, l.out_c, l.out_w*l.out_h, l.variance_delta);
void  variance_delta_cpu(float *x, float *delta, float *mean, float *variance, int batch, int filters, int spatial, float *variance_delta)
{

    int i,j,k;
    for(i = 0; i < filters; ++i){
        variance_delta[i] = 0;
        for(j = 0; j < batch; ++j){
            for(k = 0; k < spatial; ++k){
                int index = j*filters*spatial + i*spatial + k;
                variance_delta[i] += delta[index]*(x[index] - mean[i]);
            }
        }
        variance_delta[i] *= -.5 * pow(variance[i] + .00001f, (float)(-3./2.));
    }
}
```

此代碼實現的是：

$$\frac{0.5}{\left(\sigma^2+e\right)^{\frac{-3}{2}}}*\sum_{b'=0}^{B-1}\sum_{i'=0}^{H-1}\sum_{j'=0}^{W-1}\gamma*\delta^l(b',i',j')*\left(C^l_{b',i',j'}-\mu_k\right)$$

注意 variance_delta 的維度，1×#_of_filters

```
// normalize_delta_cpu(l.x, l.mean, l.variance, l.mean_delta, l.variance_delta, l.batch, l.out_c, l.out_w*l.out_h, l.delta);
void normalize_delta_cpu(float *x, float *mean, float *variance, float *mean_delta, float *variance_delta, int batch, int filters, int spatial, float *delta)
{
    int f, j, k;
    for(j = 0; j < batch; ++j){
        for(f = 0; f < filters; ++f){
            for(k = 0; k < spatial; ++k){
                int index = j*filters*spatial + f*spatial + k;
                delta[index] = delta[index] * 1./(sqrt(variance[f] + .00001f)) \
                + variance_delta[f] * 2. * (x[index] - mean[f]) / (spatial * batch) + mean_delta[f]/(spatial*batch);
            }
        }
    }
}
```

此代碼實現的是整個 $\frac{\partial Loss}{\partial C^l_{b,i,j}}$

```
delta[index] * 1./(sqrt(variance[f] + .00001f))
```

這項即： $= \delta^l(b,i,j)*\gamma*\frac{1}{\sqrt{\sigma^2+e}}$

```
+ variance_delta[f] * 2. * (x[index] - mean[f]) / (spatial * batch)
```

這項即：

$$+\frac{1}{B*N*M}*2*\left(C^l_{b,i,j}-\mu_{ch}\right)*\frac{0.5}{\left(\sigma^2+e\right)^{\frac{-3}{2}}}*\sum_{b'=0}^{B-1}\sum_{i'=0}^{H-1}\sum_{j'=0}^{W-1}\gamma*\delta^l(b',i',j')*\left(C^l_{b',i',j'}-\mu_k\right)$$

```
mean_delta[f]/(spatial*batch)
```

這項即： $+\frac{1}{B*N*M}*\sum_{b'=0}^{B-1}\sum_{i'=0}^{H-1}\sum_{j'=0}^{W-1}\gamma*\delta^l(b',i',j')*\frac{-1}{\sqrt{\sigma^2+e}}$

至此 $\frac{\partial Loss}{\partial C^l_{b,i,j}}$ 準備就緒，可進行下一步的 convolution 層的反向推導了。 $\frac{\partial Loss}{\partial C^l_{b,i,j}}$ 即爲 convolution 層的 delta map: $\delta^l_{k,b,i,j}$,k 爲輸出 channel index。 $C^l_{k,b,i,j}$ 爲 convolution 的輸出，現在要求的是：

1. $\frac{\partial Loss}{\partial weights_{kin,i,j}}$ 這個用於更新系數 (最終還是 correlation)

2. $\frac{\partial Loss}{\partial inputs_{kin,i,j}}$ 這個用於更前一層的反向傳導 (最終還是 correlation)

卷積的公式:

$$C_k^l(i,j) = \sum_{c=0}^{inCHs-1} \sum_{n'=-(N-1)/2}^{(N-1)/2} \sum_{m'=-(M-1)/2}^{(M-1)/2} a^{l-1}(c, i+n', j+m') * K_k(c, -n', -m') + b$$

where $0 \le c < inputChannels$, k *is* k-th *output channel index*,
$(N-1)/2 \le i < I-(N-1)/2$, $(M-1)/2 \le j < J-(M-1)/2$

注意如果輸入沒有 pad 0 輸入和輸出圖像大小是不一樣的，所以，注意下標 i，j。

如果令系數的轉置爲內存的存放形式，即 $K_k^l(-n,-m) = K_k^{'l}(n,m)$

那麼卷積就變成 correlation:

$$C_k^l(i,j) = \sum_{c=0}^{inCHs-1} \sum_{n'=-(N-1)/2}^{(N-1)/2} \sum_{m'=-(M-1)/2}^{(M-1)/2} a^{l-1}(c, i+n', j+m') * K_k^{'}(c, n', m') + b$$

where $0 \le c < inputChannels$, k *is* k-th *output channel index*,
$(N-1)/2 \le i < I-(N-1)/2$, $(M-1)/2 \le j < J-(M-1)/2$

$$\frac{\partial Loss}{\partial K_k^l(cin,n,m)} = \sum_{i=\frac{N-1}{2}}^{I-1-\frac{N-1}{2}} \sum_{j=\frac{M-1}{2}}^{J-1-\frac{M-1}{2}} \frac{\partial Loss}{\partial C_k^l(i,j)} * \frac{\partial C_k^l(i,j)}{\partial K_k^l(cin,n,m)}$$

$$= \sum_{i=\frac{N-1}{2}}^{I-1-\frac{N-1}{2}} \sum_{j=\frac{M-1}{2}}^{J-1-\frac{M-1}{2}} \delta^l(cin,i,j) * a^{l-1}(cin, i+n, j+m)$$

$$-\frac{N-1}{2} \le n \le \frac{N-1}{2} \quad -\frac{M-1}{2} \le m \le \frac{m-1}{2}$$

$$cin = c, n = n', m = m',$$

$$\frac{\partial Loss}{\partial K_k^l(cin,n',m')} \quad let \quad n + \frac{N-1}{2} = n' \quad m + \frac{M-1}{2} = m'$$

$$= \sum_{i=\frac{N-1}{2}}^{I-1-\frac{N-1}{2}} \sum_{j=\frac{M-1}{2}}^{J-1-\frac{M-1}{2}} \delta^{l}(cin,i,j) * a^{l-1}(cin, i+n'-\frac{N-1}{2}, j+m'-\frac{M-1}{2})$$

$0 \le n' \le N-1 \quad 0 \le m' \le m-1$

明天把卷積的反向推導的代碼和自己的公式對比下，徹底弄懂它。咋一看代碼好像跟我的推導結果不一樣，有點奇怪。

幾何意義：

詳細的見更新版的 techshare PDF 文件。

復習下 decay 的由來。

然後寫個自己的 bacthnorm 和 convolution 的反向函數，和原代碼的數據比較下，看是否正確。

$$K = k + \lambda \frac{\partial J}{\partial K} \quad \lambda \, is \, the \, learning \, rate$$

$\hat{J} = J + \frac{\gamma}{2} * K^2$ adding regularization term to the original cost function

$$K = k + \lambda \frac{\partial \hat{J}}{\partial K} = k + \lambda \frac{\partial J}{\partial K} + \lambda * \gamma * K \quad \lambda \, is \, the \, learning \, rate$$

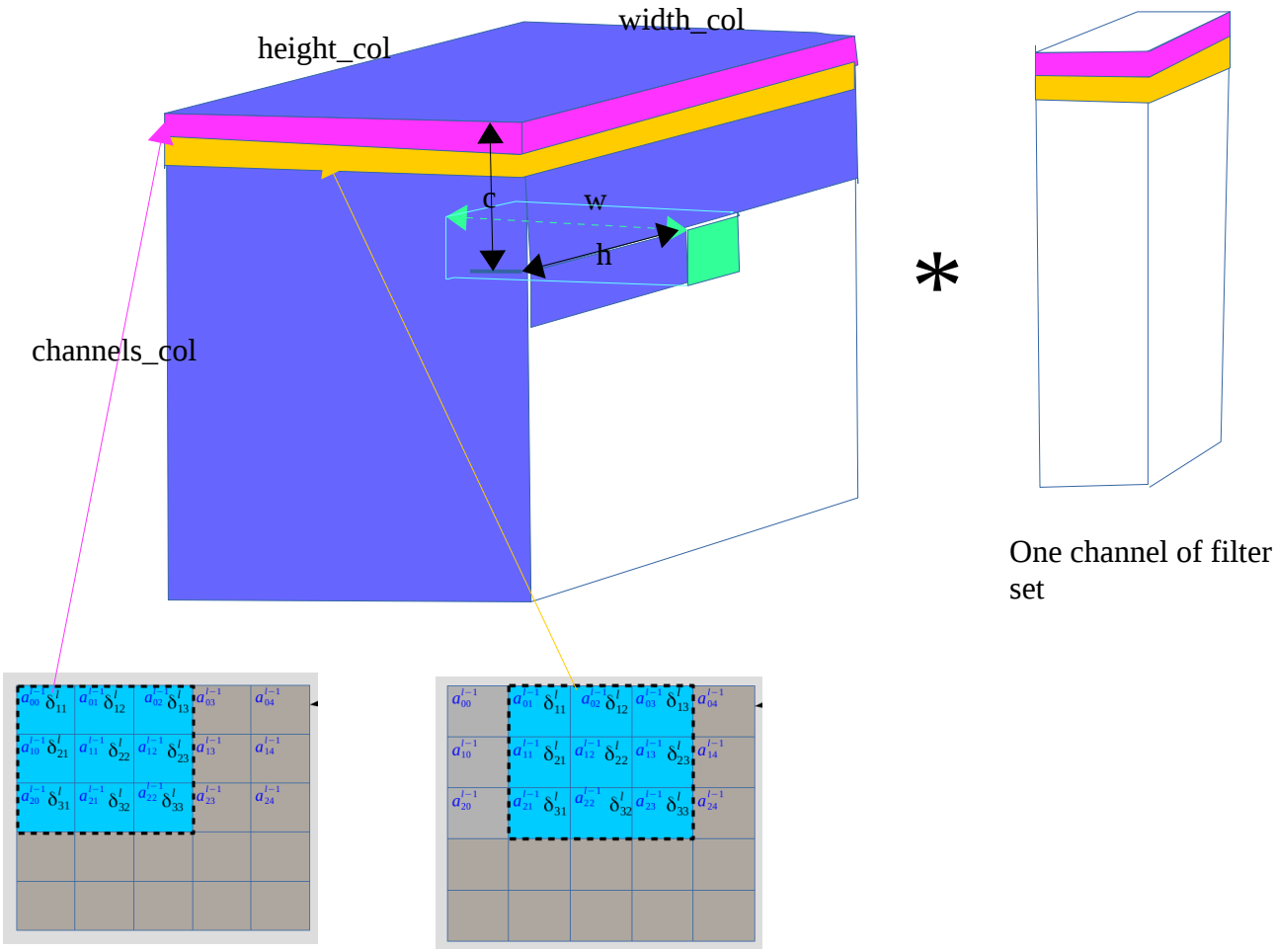
$\gamma * \lambda$ is called decaying factor, why the name decay, because if there is no update from the backpropagation, the third term will eventually lead K to zero, so it is a decaying term.

復 YOLO 的 backpropogation 系數 update 分析。

咋一看跟自己分析的不一樣，尤其是 img2col 讓我有點費解，點解要此變換，因爲這個跟自己的幾何解釋很不一樣。但分析完他的 forward 就明白了，其實跟自己的幾何解釋是一樣的，計算方式不一樣，他套用了矩陣相乘的函數。

```
//From Berkeley Vision's Caffe!
//https://github.com/BVLC/caffe/blob/master/LICENSE
void im2col_cpu(float* data_im,
     int channels,  int height,  int width,
     int ksize,  int stride, int pad, float* data_col)
{
    int c,h,w;
    int height_col = (height + 2*pad - ksize) / stride + 1;
    int width_col = (width + 2*pad - ksize) / stride + 1;

    int channels_col = channels * ksize * ksize;
    for (c = 0; c < channels_col; ++c) {
        int w_offset = c % ksize;
        int h_offset = (c / ksize) % ksize;
        int c_im = c / ksize / ksize;
        for (h = 0; h < height_col; ++h) {
            for (w = 0; w < width_col; ++w) {
                int im_row = h_offset + h * stride;
                int im_col = w_offset + w * stride;
                int col_index = (c * height_col + h) * width_col + w;
                data_col[col_index] = im2col_get_pixel(data_im, height, width, channels,
                        im_row, im_col, c_im, pad);
            }
        }
    }
}
```



One channel of filter set

volume 的每一層就是我那個幾何解釋的虛框移動所得的 input 數據。yolo 這樣做，在數據變換的時候(img2col)把 padding 也考慮進去了。
這樣變換數據排列後，對應顏色的系數 update 就是對應顏色的 volume 的 input 和 delta 相乘，然後累加。
但這麼做數據變換，真的是麻煩了一點。看到 gemn 我就頭暈。
雖然 forward 和 backward 都有 img2col，但最後乘的方式不一樣，一個是和一維向量（weights）縱向相成，另一個是和 2 維向量（delta）橫向相成。