1.Faster-RCNN

rpn_layer.hpp

```
#ifndef CAFFE_RPN_LAYER_HPP_
#define CAFFE_RPN_LAYER_HPP_
#include <vector>
#include "caffe/blob.hpp"
#include "caffe/layer.hpp"
#include "caffe/proto/caffe.pb.h"
#include"opencv2/opencv.hpp"
  #define mymax(a,b) ((a)>(b))?(a):(b)
#define mymin(a,b) ((a)>(b))?(b):(a)
namespace caffe {
        template <typename Dtype>
        class RPNLayer : public Layer<Dtype> {
        public:
                explicit RPNLayer(const LayerParameter& param)
                        : Layer<Dtype>(param) {
                                m_score_.reset(new Blob<Dtype>());
                                m_box_.reset(new Blob<Dtype>());
                                local_anchors_.reset(new Blob<Dtype>());
                        }
                virtual void LayerSetUp(const vector<Blob<Dtype>*>& bottom,
                        const vector<Blob<Dtype>*>& top);
                virtual void Reshape(const vector<Blob<Dtype>*>& bottom,
                        const vector<Blob<Dtype>*>& top){}
                virtual inline const char* type() const { return "RPN"; }

                struct abox{
                        Dtype batch_ind;
                        Dtype x1;
                        Dtype y1;
                        Dtype x2;
                        Dtype y2;
                        Dtype score;
                        bool operator <(const abox&tmp) const{
                                return score < tmp.score;
                        }
                };
```

```
protected:
    virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top);
    //virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
        //const vector<Blob<Dtype>*>& top);
    virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
        const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom){};


    int feat_stride_;
    int base_size_;
    int min_size_;
    int pre_nms_topN_;
    int post_nms_topN_;
    float nms_thresh_;
    vector<int> anchor_scales_;
    vector<float> ratios_;


    vector<vector<float> > gen_anchors_;
    int *anchors_;
    int anchors_nums_;
    int src_height_;
    int src_width_;
    float src_scale_;
    int map_width_;
    int map_height_;

    shared_ptr<Blob<Dtype> > m_score_;
    shared_ptr<Blob<Dtype> > m_box_;
    shared_ptr<Blob<Dtype> >local_anchors_;
    void generate_anchors();
    vector<vector<float> > ratio_enum(vector<float>);
    vector<float> whctrs(vector<float>);
    vector<float> mkanchor(float w,float h,float x_ctr,float y_ctr);
    vector<vector<float> > scale_enum(vector<float>);

    //cv::Mat proposal_local_anchor(int width, int height);
    void proposal_local_anchor();
    void bbox_tranform_inv();
    cv::Mat bbox_tranform_inv(cv::Mat local_anchors, cv::Mat boxs_delta);
    void nms(std::vector<abox> &input_boxes, float nms_thresh);
    void filter_boxs(cv::Mat& pre_box, cv::Mat& score, vector<abox>& aboxes);
    void filter_boxs(vector<abox>& aboxes);
```

```
	};
}	// namespace caffe
```

rpn_layer.cpp

```
#include <algorithm>
#include <vector>

#include "caffe/layers/rpn_layer.hpp"
#include "caffe/util/math_functions.hpp"
#include <opencv2/opencv.hpp>

int debug = 0;
int	tmp[9][4] = {
	{ -83, -39, 100, 56 },
	{ -175, -87, 192, 104 },
	{ -359, -183, 376, 200 },
	{ -55, -55, 72, 72 },
	{ -119, -119, 136, 136 },
	{ -247, -247, 264, 264 },
	{ -35, -79, 52, 96 },
	{ -79, -167, 96, 184 },
	{ -167, -343, 184, 360 }
};
namespace caffe {

	template <typename Dtype>
	void RPNLayer<Dtype>::LayerSetUp(
		const vector<Blob<Dtype>*>& bottom, const vector<Blob<Dtype>*>& top) {
		anchor_scales_.clear();
		ratios_.clear();
		feat_stride_ = this->layer_param_.rpn_param().feat_stride();
		base_size_ = this->layer_param_.rpn_param().basesize();
		min_size_ = this->layer_param_.rpn_param().boxminsize();
		pre_nms_topN_ = this->layer_param_.rpn_param().per_nms_topn();
		post_nms_topN_ = this->layer_param_.rpn_param().post_nms_topn();
		nms_thresh_ = this->layer_param_.rpn_param().nms_thresh();
		int scales_num = this->layer_param_.rpn_param().scale_size();
		for (int i = 0; i < scales_num; ++i)
		{
			anchor_scales_.push_back(this->layer_param_.rpn_param().scale(i));
```

```
    }
    int ratios_num = this->layer_param_.rpn_param().ratio_size();
    for (int i = 0; i < ratios_num; ++i)
    {
        ratios_.push_back(this->layer_param_.rpn_param().ratio(i));
    }


    //anchors_nums_ = 9;
    //anchors_ = new int[anchors_nums_ * 4];
    //memcpy(anchors_, tmp, 9 * 4 * sizeof(int));

    generate_anchors();

    anchors_nums_ = gen_anchors_.size();
    anchors_ = new int[anchors_nums_ * 4];
    for (int i = 0; i<gen_anchors_.size(); ++i)
    {
        for (int j = 0; j<gen_anchors_[i].size(); ++j)
        {
            anchors_[i*4+j] = gen_anchors_[i][j];
        }
    }
    top[0]->Reshape(1, 5, 1, 1);
    if (top.size() > 1)
    {
        top[1]->Reshape(1, 1, 1, 1);
    }
}

template <typename Dtype>
void RPNLayer<Dtype>::generate_anchors(){
    //generate base anchor
    vector<float> base_anchor;
    base_anchor.push_back(0);
    base_anchor.push_back(0);
    base_anchor.push_back(base_size_ - 1);
    base_anchor.push_back(base_size_ - 1);
    //enum ratio anchors
    vector<vector<float> >ratio_anchors = ratio_enum(base_anchor);
    for (int i = 0; i < ratio_anchors.size(); ++i)
    {
        vector<vector<float> > tmp = scale_enum(ratio_anchors[i]);
        gen_anchors_.insert(gen_anchors_.end(), tmp.begin(), tmp.end());
    }
}

template <typename Dtype>
vector<vector<float> > RPNLayer<Dtype>::scale_enum(vector<float> anchor){
    vector<vector<float> > result;
    vector<float> reform_anchor = whctrs(anchor);
    float x_ctr = reform_anchor[2];
    float y_ctr = reform_anchor[3];
    float w = reform_anchor[0];
    float h = reform_anchor[1];
    for (int i = 0; i < anchor_scales_.size(); ++i)
    {
        float ws = w * anchor_scales_[i];
        float hs = h *    anchor_scales_[i];
        vector<float> tmp = mkanchor(ws, hs, x_ctr, y_ctr);
        result.push_back(tmp);
    }
    return result;
}


template <typename Dtype>
vector<vector<float> > RPNLayer<Dtype>::ratio_enum(vector<float> anchor){
    vector<vector<float> > result;
    vector<float> reform_anchor = whctrs(anchor);
    float x_ctr = reform_anchor[2];
    float y_ctr = reform_anchor[3];
    float size = reform_anchor[0] * reform_anchor[1];
    for (int i = 0; i < ratios_.size(); ++i)
    {
        float size_ratios = size / ratios_[i];
        float ws = round(sqrt(size_ratios));
        float hs = round(ws*ratios_[i]);
        vector<float> tmp = mkanchor(ws, hs, x_ctr, y_ctr);
        result.push_back(tmp);
    }
    return result;
}

template <typename Dtype>
vector<float> RPNLayer<Dtype>::mkanchor(float w, float h, float x_ctr, float y_ctr){
    vector<float> tmp;
    tmp.push_back(x_ctr - 0.5*(w - 1));
    tmp.push_back(y_ctr - 0.5*(h - 1));
    tmp.push_back(x_ctr + 0.5*(w - 1));
    tmp.push_back(y_ctr + 0.5*(h - 1));
    return tmp;
}
template <typename Dtype>
vector<float> RPNLayer<Dtype>::whctrs(vector<float> anchor){
    vector<float> result;
    result.push_back(anchor[2] - anchor[0] + 1); //w
    result.push_back(anchor[3] - anchor[1] + 1); //h
    result.push_back((anchor[2] + anchor[0]) / 2); //ctrx
    result.push_back((anchor[3] + anchor[1]) / 2); //ctry
    return result;
}


/*template <typename Dtype>
cv::Mat RPNLayer<Dtype>::proposal_local_anchor(int width, int height)
{
    Blob<float> shift;
    cv::Mat shitf_x(height, width, CV_32SC1);
    cv::Mat shitf_y(height, width, CV_32SC1);
    for (size_t i = 0; i < width; i++)
    {
        for (size_t j = 0; j < height; j++)
        {
            shitf_x.at<int>(j, i) = i * feat_stride_;
            shitf_y.at<int>(j, i) = j * feat_stride_;
        }
    }
    shift.Reshape(anchors_nums_, width*height, 4,    1);
    float *p = shift.mutable_cpu_diff(), *a = shift.mutable_cpu_data();
    for (int i = 0; i < height*width; i++)
    {
        for (int j = 0; j < anchors_nums_; j++)
        {
            size_t num = i * 4 + j * 4 * height*width;
            p[num + 0] = -shitf_x.at<int>(i / shitf_x.cols, i % shitf_x.cols);
            p[num + 2] = -shitf_x.at<int>(i / shitf_x.cols, i % shitf_x.cols);
            p[num + 1] = -shitf_y.at<int>(i / shitf_y.cols, i % shitf_y.cols);
            p[num + 3] = -shitf_y.at<int>(i / shitf_y.cols, i % shitf_y.cols);
            a[num + 0] = anchors_[j * 4 + 0];
            a[num + 1] = anchors_[j * 4 + 1];
            a[num + 2] = anchors_[j * 4 + 2];
```

```
                a[num + 3] = anchors_[j * 4 + 3];
            }
        }
        shift.Update();
        cv::Mat loacl_anchors(anchors_nums_ * height*width, 4, CV_32FC1);
        size_t num = 0;
        for (int i = 0; i < height; ++i)
        {
            for (int j = 0; j < width; ++j)
            {
                for (int c = 0; c < anchors_nums_; ++c)
                {
                    for (int k = 0; k < 4; ++k)
                    {
                        loacl_anchors.at<float>((i*width      +      j)*anchors_nums_+c,      k)=
shift.data_at(c, i*width + j, k, 0);
                    }
                }
            }
        }
        return loacl_anchors;
    }*/

    template <typename Dtype>
    void RPNLayer<Dtype>::proposal_local_anchor(){
        int length = mymax(map_width_, map_height_);
        int step = map_width_*map_height_;
        int *map_m = new int[length];
        for (int i = 0; i < length; ++i)
        {
            map_m[i] = i*feat_stride_;
        }
        Dtype *shift_x = new Dtype[step];
        Dtype *shift_y = new Dtype[step];
        for (int i = 0; i < map_height_; ++i)
        {
            for (int j = 0; j < map_width_; ++j)
            {
                shift_x[i*map_width_ + j] = map_m[j];
                shift_y[i*map_width_ + j] = map_m[i];
            }
        }
        local_anchors_->Reshape(1, anchors_nums_ * 4, map_height_, map_width_);
        Dtype *a = local_anchors_->mutable_cpu_data();
        for (int i = 0; i < anchors_nums_; ++i)
        {
            caffe_set(step, Dtype(anchors_[i * 4 + 0]), a + (i * 4 + 0) *step);
            caffe_set(step, Dtype(anchors_[i * 4 + 1]), a + (i * 4 + 1) *step);
            caffe_set(step, Dtype(anchors_[i * 4 + 2]), a + (i * 4 + 2) *step);
            caffe_set(step, Dtype(anchors_[i * 4 + 3]), a + (i * 4 + 3) *step);
            caffe_axpy(step, Dtype(1), shift_x, a + (i * 4 + 0)*step);
            caffe_axpy(step, Dtype(1), shift_x, a + (i * 4 + 2)*step);
            caffe_axpy(step, Dtype(1), shift_y, a + (i * 4 + 1)*step);
            caffe_axpy(step, Dtype(1), shift_y, a + (i * 4 + 3)*step);
        }
    }

    template<typename Dtype>
    void RPNLayer<Dtype>::filter_boxs(cv::Mat& pre_box, cv::Mat& score, vector<abox>&
aboxes)
    {
        float localMinSize=min_size_*src_scale_;
        aboxes.clear();

        for (int i = 0; i < pre_box.rows; i++)
        {
            int widths = pre_box.at<float>(i, 2) - pre_box.at<float>(i, 0) + 1;
            int heights = pre_box.at<float>(i, 3) - pre_box.at<float>(i, 1) + 1;
            if (widths >= localMinSize || heights >= localMinSize)
            {
                abox tmp;
                tmp.x1 = pre_box.at<float>(i, 0);
                tmp.y1 = pre_box.at<float>(i, 1);
                tmp.x2 = pre_box.at<float>(i, 2);
                tmp.y2 = pre_box.at<float>(i, 3);
                tmp.score = score.at<float>(i, 0);
                aboxes.push_back(tmp);
            }
        }
    }

    template<typename Dtype>
    void RPNLayer<Dtype>::filter_boxs(vector<abox>& aboxes)
    {
        float localMinSize = min_size_*src_scale_;
        aboxes.clear();
        int map_width = m_box_->width();
        int map_height = m_box_->height();
int map_channel = m_box_->channels();
const Dtype *box = m_box_->cpu_data();
const Dtype *score = m_score_->cpu_data();

int step = 4 * map_height*map_width;
int one_step = map_height*map_width;
int offset_w, offset_h, offset_x, offset_y, offset_s;

for (int h = 0; h < map_height; ++h)
{
    for (int w = 0; w < map_width; ++w)
    {
        offset_x = h*map_width + w;
        offset_y = offset_x + one_step;
        offset_w = offset_y + one_step;
        offset_h = offset_w + one_step;
        offset_s = one_step*anchors_nums_+h*map_width + w;
        for (int c = 0; c < map_channel / 4; ++c)
        {
            Dtype width = box[offset_w], height = box[offset_h];
            if (width < localMinSize || height < localMinSize)
            {
            }
            else
            {
                abox tmp;
                tmp.batch_ind = 0;
                tmp.x1 = box[offset_x] - 0.5*width;
                tmp.y1 = box[offset_y] - 0.5*height;
                tmp.x2 = box[offset_x] + 0.5*width;
                tmp.y2 = box[offset_y] + 0.5*height;
                tmp.x1 = mymin(mymax(tmp.x1, 0), src_width_);
                tmp.y1 = mymin(mymax(tmp.y1, 0), src_height_);
                tmp.x2 = mymin(mymax(tmp.x2, 0), src_width_);
                tmp.y2 = mymin(mymax(tmp.y2, 0), src_height_);
                tmp.score = score[offset_s];
                aboxes.push_back(tmp);
            }
            offset_x += step;
            offset_y += step;
            offset_w += step;
            offset_h += step;
            offset_s += one_step;
        }
        }
    }
}

template<typename Dtype>
void RPNLayer<Dtype>::bbox_tranform_inv(){
    int channel = m_box_->channels();
    int height = m_box_->height();
    int width = m_box_->width();
    int step = height*width;
    Dtype * a = m_box_->mutable_cpu_data();
    Dtype * b = local_anchors_->mutable_cpu_data();
    for (int i = 0; i < channel / 4; ++i)
    {
        caffe_axpy(2*step, Dtype(-1), b + (i * 4 + 0)*step, b + (i * 4 + 2)*step);
        caffe_add_scalar(2 * step, Dtype(1), b + (i * 4 + 2)*step);
        caffe_axpy(2*step, Dtype(0.5), b + (i * 4 + 2)*step, b + (i * 4 + 0)*step);

        caffe_mul(2 * step, b + (i * 4 + 2)*step, a + (i * 4 + 0)*step, a + (i * 4 + 0)*step);
        caffe_add(2 * step, b + (i * 4 + 0)*step, a + (i * 4 + 0)*step, a + (i * 4 + 0)*step);

        caffe_exp(2*step, a + (i * 4 + 2)*step, a + (i * 4 + 2)*step);
        caffe_mul(2 * step, b + (i * 4 + 2)*step, a + (i * 4 + 2)*step, a + (i * 4 + 2)*step);
    }
}



template<typename Dtype>
void RPNLayer<Dtype>::nms(std::vector<abox> &input_boxes, float nms_thresh){
    std::vector<float>vArea(input_boxes.size());
    for (int i = 0; i < input_boxes.size(); ++i)
    {
        vArea[i] = (input_boxes.at(i).x2 - input_boxes.at(i).x1 + 1)
            * (input_boxes.at(i).y2 - input_boxes.at(i).y1 + 1);
    }
    for (int i = 0; i < input_boxes.size(); ++i)
    {
        for (int j = i + 1; j < input_boxes.size();)
        {
            float xx1 = std::max(input_boxes[i].x1, input_boxes[j].x1);
            float yy1 = std::max(input_boxes[i].y1, input_boxes[j].y1);
            float xx2 = std::min(input_boxes[i].x2, input_boxes[j].x2);
```

```
                float yy2 = std::min(input_boxes[i].y2, input_boxes[j].y2);
                float w = std::max(float(0), xx2 - xx1 + 1);
                float h = std::max(float(0), yy2 - yy1 + 1);
                float inter = w * h;
                float ovr = inter / (vArea[i] + vArea[j] - inter);
                if (ovr >= nms_thresh)
                {
                    input_boxes.erase(input_boxes.begin() + j);
                    vArea.erase(vArea.begin() + j);
                }
                else
                {
                    j++;
                }
            }
        }
    }

    template <typename Dtype>
    void RPNLayer<Dtype>::Forward_cpu(
        const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top) {

        map_width_ = bottom[1]->width();
        map_height_ = bottom[1]->height();
        //int channels = bottom[1]->channels();


        //get boxs_delta,向右。
        m_box_->CopyFrom(*(bottom[1]), false, true);
        /*cv::Mat boxs_delta(height*width*anchors_nums_, 4, CV_32FC1);
        for (int i = 0; i < height; ++i)
        {
            for (int j = 0; j < width; ++j)
            {
                for (int k = 0; k < anchors_nums_; ++k)
                {
                    for (int c = 0; c < 4; ++c)
                    {
                        boxs_delta.at<float>((i*width + j)*anchors_nums_ + k, c) = bottom[1]->data_at(0, k*4 + c, i, j);
                    }
                }
            }
    }*/



    //get sores  向右，前面 anchors_nums_个位 bg 的得分，后面 anchors_nums_为 fg
得分，我们需要的是后面的。
    m_score_->CopyFrom(*(bottom[0]),false,true);

    /*cv::Mat scores(height*width*anchors_nums_, 1, CV_32FC1);
    for (int i = 0; i < height; ++i)
    {
        for (int j = 0; j < width; ++j)
        {
            for (int k = 0; k < anchors_nums_; ++k)
            {
                scores.at<float>((i*width + j)*anchors_nums_+k, 0) =
bottom[0]->data_at(0, k + anchors_nums_, i, j);
            }
        }
    }*/

    //get im_info

    src_height_ = bottom[2]->data_at(0, 0,0,0);
    src_width_ = bottom[2]->data_at(0, 1,0,0);
    src_scale_ = bottom[2]->data_at(0, 2, 0, 0);

    //gen local anchors  向右

    proposal_local_anchor();
    //cv::Mat local_anchors = proposal_local_anchor(width, height);


    //Convert anchors into proposals via bbox transformations

    bbox_tranform_inv();

    /*for (int i = 0; i < pre_box.rows; ++i)
    {
        if (pre_box.at<float>(i, 0) < 0) pre_box.at<float>(i, 0) = 0;
        if (pre_box.at<float>(i, 0) > (src_width_ - 1)) pre_box.at<float>(i, 0) = src_width_ -
1;
        if (pre_box.at<float>(i, 2) < 0) pre_box.at<float>(i, 2) = 0;
        if (pre_box.at<float>(i, 2) > (src_width_ - 1)) pre_box.at<float>(i, 2) = src_width_ -
```

```
1;
            if (pre_box.at<float>(i, 1) < 0) pre_box.at<float>(i, 1) = 0;
            if (pre_box.at<float>(i, 1) > (src_height_ - 1))pre_box.at<float>(i, 1) = src_height_ -
1;
            if (pre_box.at<float>(i, 3) < 0) pre_box.at<float>(i, 3) = 0;
            if (pre_box.at<float>(i, 3) > (src_height_ - 1))pre_box.at<float>(i, 3) = src_height_ -
1;
        }*/
        vector<abox>aboxes;

        filter_boxs(aboxes);

        //clock_t start, end;
        //start = clock();
        std::sort(aboxes.rbegin(), aboxes.rend()); //降序
        if (pre_nms_topN_ > 0)
        {
            int tmp = mymin(pre_nms_topN_, aboxes.size());
            aboxes.erase(aboxes.begin() + tmp, aboxes.end());
        }

        nms(aboxes,nms_thresh_);
        //end = clock();
        //std::cout << "sort nms:" << (double)(end - start) / CLOCKS_PER_SEC << std::endl;
        if (post_nms_topN_ > 0)
        {
            int tmp = mymin(post_nms_topN_, aboxes.size());
            aboxes.erase(aboxes.begin() + tmp, aboxes.end());
        }
        top[0]->Reshape(aboxes.size(),5,1,1);
        Dtype *top0 = top[0]->mutable_cpu_data();
        for (int i = 0; i < aboxes.size(); ++i)
        {
            //caffe_copy(aboxes.size() * 5, (Dtype*)aboxes.data(), top0);
            top0[0] = aboxes[i].batch_ind;
            top0[1] = aboxes[i].x1;
            top0[2] = aboxes[i].y1;
            top0[3] = aboxes[i].x2;
            top0[4] = aboxes[i].y2;
            top0 += top[0]->offset(1);
        }
        if (top.size()>1)
        {
            top[1]->Reshape(aboxes.size(), 1,1,1);
            Dtype *top1 = top[1]->mutable_cpu_data();
            for (int i = 0; i < aboxes.size(); ++i)
            {
                top1[0] = aboxes[i].score;
                top1 += top[1]->offset(1);
            }
        }
    }

#ifdef CPU_ONLY
        STUB_GPU(RPNLayer);
#endif

    INSTANTIATE_CLASS(RPNLayer);
    REGISTER_LAYER_CLASS(RPN);

}   // namespace caffe
```

roi_pooling_layer.hpp

```
#ifndef CAFFE_ROI_POOLING_LAYER_HPP_
#define CAFFE_ROI_POOLING_LAYER_HPP_

#include <vector>

#include "caffe/blob.hpp"
#include "caffe/common.hpp"
#include "caffe/layer.hpp"
#include "caffe/proto/caffe.pb.h"

namespace caffe {

/**
 * @brief Perform max pooling on regions of interest specified by input, takes
 *          as input N feature maps and a list of R regions of interest.
 *
 *    ROIPoolingLayer takes 2 inputs and produces 1 output. bottom[0] is
 *    [N x C x H x W] feature maps on which pooling is performed. bottom[1] is
 *    [R x 5] containing a list R ROI tuples with batch index and coordinates of
 *    regions of interest. Each row in bottom[1] is a ROI tuple in format
 *    [batch_index x1 y1 x2 y2], where batch_index corresponds to the index of
 *    instance in the first input and x1 y1 x2 y2 are 0-indexed coordinates
 *    of ROI rectangle (including its boundaries).
 *
 *     For each of the R ROIs, max-pooling is performed over pooled_h x pooled_w
 *     output bins (specified in roi_pooling_param). The pooling bin sizes are
 *     adaptively set such that they tile ROI rectangle in the indexed feature
 *     map. The pooling region of vertical bin ph in [0, pooled_h) is computed as
 *
 *       start_ph (included) = y1 + floor(ph * (y2 - y1 + 1) / pooled_h)
 *       end_ph (excluded)      = y1 + ceil((ph + 1) * (y2 - y1 + 1) / pooled_h)
 *
 *     and similar horizontal bins.
 *
 * @param param provides ROIPoolingParameter roi_pooling_param,
 *            with ROIPoolingLayer options:
 *    - pooled_h. The pooled output height.
 *    - pooled_w. The pooled output width
 *    - spatial_scale. Multiplicative spatial scale factor to translate ROI
 *    coordinates from their input scale to the scale used when pooling.
 *
 * Fast R-CNN
 * Written by Ross Girshick
 */

template <typename Dtype>
class ROIPoolingLayer : public Layer<Dtype> {
 public:
  explicit ROIPoolingLayer(const LayerParameter& param)
      : Layer<Dtype>(param) {}
  virtual void LayerSetUp(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top);
  virtual void Reshape(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top);

  virtual inline const char* type() const { return "ROIPooling"; }

  virtual inline int MinBottomBlobs() const { return 2; }
  virtual inline int MaxBottomBlobs() const { return 2; }
  virtual inline int MinTopBlobs() const { return 1; }
  virtual inline int MaxTopBlobs() const { return 1; }

 protected:
  virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top);
  virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top);
  virtual void Backward_cpu(const vector<Blob<Dtype>*>& top,
      const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom);
  virtual void Backward_gpu(const vector<Blob<Dtype>*>& top,
      const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom);

  int channels_;
  int height_;
  int width_;
  int pooled_height_;
  int pooled_width_;
  Dtype spatial_scale_;
  Blob<int> max_idx_;
};

}  // namespace caffe

#endif  // CAFFE_ROI_POOLING_LAYER_HPP_
```

roi_pooling_layer.cpp&roi_pooling_layer.cu

```
#include <algorithm>
#include <cfloat>
#include <vector>

#include "caffe/layers/roi_pooling_layer.hpp"

using std::max;
using std::min;
using std::floor;
using std::ceil;

namespace caffe {

template <typename Dtype>
void ROIPoolingLayer<Dtype>::LayerSetUp(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) {
  ROIPoolingParameter roi_pool_param = this->layer_param_.roi_pooling_param();
  CHECK_GT(roi_pool_param.pooled_h(), 0)
      << "pooled_h must be > 0";
  CHECK_GT(roi_pool_param.pooled_w(), 0)
      << "pooled_w must be > 0";
  pooled_height_ = roi_pool_param.pooled_h();
  pooled_width_ = roi_pool_param.pooled_w();
  spatial_scale_ = roi_pool_param.spatial_scale();
  LOG(INFO) << "Spatial scale: " << spatial_scale_;
}

template <typename Dtype>
void ROIPoolingLayer<Dtype>::Reshape(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) {
  channels_ = bottom[0]->channels();
  height_ = bottom[0]->height();
  width_ = bottom[0]->width();
  top[0]->Reshape(bottom[1]->num(), channels_, pooled_height_,
      pooled_width_);
  max_idx_.Reshape(bottom[1]->num(), channels_, pooled_height_,
      pooled_width_);
}

template <typename Dtype>
void ROIPoolingLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) {
  const Dtype* bottom_data = bottom[0]->cpu_data();
  const Dtype* bottom_rois = bottom[1]->cpu_data();
  // Number of ROIs
  int num_rois = bottom[1]->num();
  int batch_size = bottom[0]->num();
  int top_count = top[0]->count();
  Dtype* top_data = top[0]->mutable_cpu_data();
  caffe_set(top_count, Dtype(-FLT_MAX), top_data);
  int* argmax_data = max_idx_.mutable_cpu_data();
  caffe_set(top_count, -1, argmax_data);

  // For each ROI R = [batch_index x1 y1 x2 y2]: max pool over R
  for (int n = 0; n < num_rois; ++n) {
    int roi_batch_ind = bottom_rois[0];
    int roi_start_w = round(bottom_rois[1] * spatial_scale_);
    int roi_start_h = round(bottom_rois[2] * spatial_scale_);
    int roi_end_w = round(bottom_rois[3] * spatial_scale_);
    int roi_end_h = round(bottom_rois[4] * spatial_scale_);
    CHECK_GE(roi_batch_ind, 0);
    CHECK_LT(roi_batch_ind, batch_size);

    int roi_height = max(roi_end_h - roi_start_h + 1, 1);
    int roi_width = max(roi_end_w - roi_start_w + 1, 1);
    const Dtype bin_size_h = static_cast<Dtype>(roi_height)
                             / static_cast<Dtype>(pooled_height_);
    const Dtype bin_size_w = static_cast<Dtype>(roi_width)
```

```
                         / static_cast<Dtype>(pooled_width_);

const Dtype* batch_data = bottom_data + bottom[0]->offset(roi_batch_ind);

for (int c = 0; c < channels_; ++c) {
  for (int ph = 0; ph < pooled_height_; ++ph) {
    for (int pw = 0; pw < pooled_width_; ++pw) {
      // Compute pooling region for this output unit:
      //    start (included) = floor(ph * roi_height / pooled_height_)
      //    end (excluded) = ceil((ph + 1) * roi_height / pooled_height_)
      int hstart = static_cast<int>(floor(static_cast<Dtype>(ph)
                                          * bin_size_h));
      int wstart = static_cast<int>(floor(static_cast<Dtype>(pw)
                                          * bin_size_w));
      int hend = static_cast<int>(ceil(static_cast<Dtype>(ph + 1)
                                       * bin_size_h));
      int wend = static_cast<int>(ceil(static_cast<Dtype>(pw + 1)
                                       * bin_size_w));

      hstart = min(max(hstart + roi_start_h, 0), height_);
      hend = min(max(hend + roi_start_h, 0), height_);
      wstart = min(max(wstart + roi_start_w, 0), width_);
      wend = min(max(wend + roi_start_w, 0), width_);

      bool is_empty = (hend <= hstart) || (wend <= wstart);

      const int pool_index = ph * pooled_width_ + pw;
      if (is_empty) {
        top_data[pool_index] = 0;
        argmax_data[pool_index] = -1;
      }

      for (int h = hstart; h < hend; ++h) {
        for (int w = wstart; w < wend; ++w) {
          const int index = h * width_ + w;
          if (batch_data[index] > top_data[pool_index]) {
            top_data[pool_index] = batch_data[index];
            argmax_data[pool_index] = index;
          }
        }
      }
    }
  }
  // Increment all data pointers by one channel
```

```
      batch_data += bottom[0]->offset(0, 1);
      top_data += top[0]->offset(0, 1);
      argmax_data += max_idx_.offset(0, 1);
    }
    // Increment ROI data pointer
    bottom_rois += bottom[1]->offset(1);
  }
}

template <typename Dtype>
void ROIPoolingLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*>& top,
      const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom) {
  if (propagate_down[1]) {
    LOG(FATAL) << this->type()
               << " Layer cannot backpropagate to roi inputs.";
  }
  if (!propagate_down[0]) {
    return;
  }
  const Dtype* bottom_rois = bottom[1]->cpu_data();
  const Dtype* top_diff = top[0]->cpu_diff();
  Dtype* bottom_diff = bottom[0]->mutable_cpu_diff();
  caffe_set(bottom[0]->count(), Dtype(0.), bottom_diff);
  const int* argmax_data = max_idx_.cpu_data();
  const int num_rois = top[0]->num();

  // Accumulate gradient over all ROIs
  for (int roi_n = 0; roi_n < num_rois; ++roi_n) {
    int roi_batch_ind = bottom_rois[roi_n * 5];
    // Accumulate gradients over each bin in this ROI
    for (int c = 0; c < channels_; ++c) {
      for (int ph = 0; ph < pooled_height_; ++ph) {
        for (int pw = 0; pw < pooled_width_; ++pw) {
          int offset_top = ((roi_n * channels_ + c) * pooled_height_ + ph)
              * pooled_width_ + pw;
          int argmax_index = argmax_data[offset_top];
          if (argmax_index >= 0) {
            int offset_bottom = (roi_batch_ind * channels_ + c) * height_
                * width_ + argmax_index;
            bottom_diff[offset_bottom] += top_diff[offset_top];
          }
        }
      }
    }
  }
}


#ifdef CPU_ONLY
STUB_GPU(ROIPoolingLayer);
#endif

INSTANTIATE_CLASS(ROIPoolingLayer);
REGISTER_LAYER_CLASS(ROIPooling);

}  // namespace caffe

#include <algorithm>
#include <cfloat>
#include <vector>

#include "caffe/layers/roi_pooling_layer.hpp"


using std::max;
using std::min;

namespace caffe {

template <typename Dtype>
__global__ void ROIPoolForward(const int nthreads, const Dtype* bottom_data,
    const Dtype spatial_scale, const int channels, const int height,
    const int width, const int pooled_height, const int pooled_width,
    const Dtype* bottom_rois, Dtype* top_data, int* argmax_data) {
  CUDA_KERNEL_LOOP(index, nthreads) {
    // (n, c, ph, pw) is an element in the pooled output
    int pw = index % pooled_width;
    int ph = (index / pooled_width) % pooled_height;
    int c = (index / pooled_width / pooled_height) % channels;
    int n = index / pooled_width / pooled_height / channels;

    bottom_rois += n * 5;
    int roi_batch_ind = bottom_rois[0];
    int roi_start_w = round(bottom_rois[1] * spatial_scale);
    int roi_start_h = round(bottom_rois[2] * spatial_scale);
    int roi_end_w = round(bottom_rois[3] * spatial_scale);
    int roi_end_h = round(bottom_rois[4] * spatial_scale);
    // Force malformed ROIs to be 1x1
    int roi_width = max(roi_end_w - roi_start_w + 1, 1);
    int roi_height = max(roi_end_h - roi_start_h + 1, 1);
    Dtype bin_size_h = static_cast<Dtype>(roi_height)
                       / static_cast<Dtype>(pooled_height);
    Dtype bin_size_w = static_cast<Dtype>(roi_width)
                       / static_cast<Dtype>(pooled_width);

    int hstart = static_cast<int>(floor(static_cast<Dtype>(ph)
                                        * bin_size_h));
    int wstart = static_cast<int>(floor(static_cast<Dtype>(pw)
                                        * bin_size_w));
    int hend = static_cast<int>(ceil(static_cast<Dtype>(ph + 1)
                                     * bin_size_h));
    int wend = static_cast<int>(ceil(static_cast<Dtype>(pw + 1)
                                     * bin_size_w));

    // Add roi offsets and clip to input boundaries
    hstart = min(max(hstart + roi_start_h, 0), height);
    hend = min(max(hend + roi_start_h, 0), height);
    wstart = min(max(wstart + roi_start_w, 0), width);
    wend = min(max(wend + roi_start_w, 0), width);
    bool is_empty = (hend <= hstart) || (wend <= wstart);

    // Define an empty pooling region to be zero
    Dtype maxval = is_empty ? 0 : -FLT_MAX;
    // If nothing is pooled, argmax = -1 causes nothing to be backprop'd
    int maxidx = -1;
    bottom_data += (roi_batch_ind * channels + c) * height * width;
    for (int h = hstart; h < hend; ++h) {
      for (int w = wstart; w < wend; ++w) {
        int bottom_index = h * width + w;
        if (bottom_data[bottom_index] > maxval) {
          maxval = bottom_data[bottom_index];
          maxidx = bottom_index;
        }
      }
    }
    top_data[index] = maxval;
    argmax_data[index] = maxidx;
  }
}

template <typename Dtype>
```

```
void ROIPoolingLayer<Dtype>::Forward_gpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) {
  const Dtype* bottom_data = bottom[0]->gpu_data();
  const Dtype* bottom_rois = bottom[1]->gpu_data();
  Dtype* top_data = top[0]->mutable_gpu_data();
  int* argmax_data = max_idx_.mutable_gpu_data();
  int count = top[0]->count();
  // NOLINT_NEXT_LINE(whitespace/operators)
  ROIPoolForward<Dtype><<<CAFFE_GET_BLOCKS(count), CAFFE_CUDA_NUM_THREADS>>>(
      count, bottom_data, spatial_scale_, channels_, height_, width_,
      pooled_height_, pooled_width_, bottom_rois, top_data, argmax_data);
  CUDA_POST_KERNEL_CHECK;
}

template <typename Dtype>
__global__ void ROIPoolBackward(const int nthreads, const Dtype* top_diff,
    const int* argmax_data, const int num_rois, const Dtype spatial_scale,
    const int channels, const int height, const int width,
    const int pooled_height, const int pooled_width, Dtype* bottom_diff,
    const Dtype* bottom_rois) {
  CUDA_KERNEL_LOOP(index, nthreads) {
    // (n, c, h, w) coords in bottom data
    int w = index % width;
    int h = (index / width) % height;
    int c = (index / width / height) % channels;
    int n = index / width / height / channels;

    Dtype gradient = 0;
    // Accumulate gradient over all ROIs that pooled this element
    for (int roi_n = 0; roi_n < num_rois; ++roi_n) {
      const Dtype* offset_bottom_rois = bottom_rois + roi_n * 5;
      int roi_batch_ind = offset_bottom_rois[0];
      // Skip if ROI's batch index doesn't match n
      if (n != roi_batch_ind) {
        continue;
      }

      int roi_start_w = round(offset_bottom_rois[1] * spatial_scale);
      int roi_start_h = round(offset_bottom_rois[2] * spatial_scale);
      int roi_end_w = round(offset_bottom_rois[3] * spatial_scale);
      int roi_end_h = round(offset_bottom_rois[4] * spatial_scale);

      // Skip if ROI doesn't include (h, w)
      const bool in_roi = (w >= roi_start_w && w <= roi_end_w &&
                                               h >= roi_start_h && h <= roi_end_h);
      if (!in_roi) {
        continue;
      }

      int offset = (roi_n * channels + c) * pooled_height * pooled_width;
      const Dtype* offset_top_diff = top_diff + offset;
      const int* offset_argmax_data = argmax_data + offset;

      // Compute feasible set of pooled units that could have pooled
      // this bottom unit

      // Force malformed ROIs to be 1x1
      int roi_width = max(roi_end_w - roi_start_w + 1, 1);
      int roi_height = max(roi_end_h - roi_start_h + 1, 1);

      Dtype bin_size_h = static_cast<Dtype>(roi_height)
                                           / static_cast<Dtype>(pooled_height);
      Dtype bin_size_w = static_cast<Dtype>(roi_width)
                                           / static_cast<Dtype>(pooled_width);

      int phstart = floor(static_cast<Dtype>(h - roi_start_h) / bin_size_h);
      int phend = ceil(static_cast<Dtype>(h - roi_start_h + 1) / bin_size_h);
      int pwstart = floor(static_cast<Dtype>(w - roi_start_w) / bin_size_w);
      int pwend = ceil(static_cast<Dtype>(w - roi_start_w + 1) / bin_size_w);

      phstart = min(max(phstart, 0), pooled_height);
      phend = min(max(phend, 0), pooled_height);
      pwstart = min(max(pwstart, 0), pooled_width);
      pwend = min(max(pwend, 0), pooled_width);

      for (int ph = phstart; ph < phend; ++ph) {
        for (int pw = pwstart; pw < pwend; ++pw) {
            if (offset_argmax_data[ph * pooled_width + pw] == (h * width + w)) {
                gradient += offset_top_diff[ph * pooled_width + pw];
            }
        }
      }
    }
    bottom_diff[index] = gradient;
  }
}

template <typename Dtype>
```

```
void ROIPoolingLayer<Dtype>::Backward_gpu(const vector<Blob<Dtype>*>& top,
      const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom) {
  if (!propagate_down[0]) {
    return;
  }
  const Dtype* bottom_rois = bottom[1]->gpu_data();
  const Dtype* top_diff = top[0]->gpu_diff();
  Dtype* bottom_diff = bottom[0]->mutable_gpu_diff();
  const int count = bottom[0]->count();
  caffe_gpu_set(count, Dtype(0.), bottom_diff);
  const int* argmax_data = max_idx_.gpu_data();
  // NOLINT_NEXT_LINE(whitespace/operators)
  ROIPoolBackward<Dtype><<<CAFFE_GET_BLOCKS(count), CAFFE_CUDA_NUM_THREADS>>>(
      count, top_diff, argmax_data, top[0]->num(), spatial_scale_, channels_,
      height_, width_, pooled_height_, pooled_width_, bottom_diff, bottom_rois);
  CUDA_POST_KERNEL_CHECK;
}

INSTANTIATE_LAYER_GPU_FUNCS(ROIPoolingLayer);

}

message ROIPoolingParameter {
  optional uint32 pooled_h = 1 [default = 0];
  optional uint32 pooled_w = 2 [default = 0];
  optional float spatial_scale = 3 [default = 1];
}
message RPNParameter {
  optional uint32 feat_stride = 1;
  optional uint32 basesize = 2;
  repeated uint32 scale = 3;
  repeated float ratio = 4;
  optional uint32 boxminsize =5;
  optional uint32 per_nms_topn = 9;
  optional uint32 post_nms_topn = 11;
  optional float nms_thresh = 8;
}

namespace RPN{
    struct abox
    {
        float x1;
        float y1;
        float x2;
        float y2;
        float score;
        bool operator <(const abox&tmp) const{
            return score < tmp.score;
        }
   };
    void nms(std::vector<abox>& input_boxes,float nms_thresh);
    cv::Mat bbox_tranform_inv(cv::Mat, cv::Mat);
}
namespace RPN{
    cv::Mat bbox_tranform_inv(cv::Mat local_anchors, cv::Mat boxs_delta){
        cv::Mat pre_box(local_anchors.rows, local_anchors.cols, CV_32FC1);
        for (int i = 0; i < local_anchors.rows; i++)
        {
            double pred_ctr_x, pred_ctr_y, src_ctr_x, src_ctr_y;
            double dst_ctr_x, dst_ctr_y, dst_scl_x, dst_scl_y;
            double src_w, src_h, pred_w, pred_h;
            src_w = local_anchors.at<float>(i, 2) - local_anchors.at<float>(i, 0) + 1;
            src_h = local_anchors.at<float>(i, 3) - local_anchors.at<float>(i, 1) + 1;
            src_ctr_x = local_anchors.at<float>(i, 0) + 0.5 * src_w;
            src_ctr_y = local_anchors.at<float>(i, 1) + 0.5 * src_h;

            dst_ctr_x = boxs_delta.at<float>(i, 0);
            dst_ctr_y = boxs_delta.at<float>(i, 1);
            dst_scl_x = boxs_delta.at<float>(i, 2);
            dst_scl_y = boxs_delta.at<float>(i, 3);
            pred_ctr_x = dst_ctr_x*src_w + src_ctr_x;
            pred_ctr_y = dst_ctr_y*src_h + src_ctr_y;
            pred_w = exp(dst_scl_x) * src_w;
            pred_h = exp(dst_scl_y) * src_h;

            pre_box.at<float>(i, 0) = pred_ctr_x - 0.5*pred_w;
            pre_box.at<float>(i, 1) = pred_ctr_y - 0.5*pred_h;
            pre_box.at<float>(i, 2) = pred_ctr_x + 0.5*pred_w;
            pre_box.at<float>(i, 3) = pred_ctr_y + 0.5*pred_h;
        }
        return pre_box;
    }
    void nms(std::vector<abox> &input_boxes, float nms_thresh){
        std::vector<float>vArea(input_boxes.size());
        for (int i = 0; i < input_boxes.size(); ++i)
        {
            vArea[i] = (input_boxes.at(i).x2 - input_boxes.at(i).x1 + 1)
                * (input_boxes.at(i).y2 - input_boxes.at(i).y1 + 1);
        }
        for (int i = 0; i < input_boxes.size(); ++i)
        {
            for (int j = i + 1; j < input_boxes.size();)
            {
                float xx1 = std::max(input_boxes[i].x1, input_boxes[j].x1);
                float yy1 = std::max(input_boxes[i].y1, input_boxes[j].y1);
                float xx2 = std::min(input_boxes[i].x2, input_boxes[j].x2);
                float yy2 = std::min(input_boxes[i].y2, input_boxes[j].y2);
                float w = std::max(float(0), xx2 - xx1 + 1);
                float      h = std::max(float(0), yy2 - yy1 + 1);
                float      inter = w * h;
                float ovr = inter / (vArea[i] + vArea[j] - inter);
                if (ovr >= nms_thresh)
                {
                    input_boxes.erase(input_boxes.begin() + j);
                    vArea.erase(vArea.begin() + j);
                }
                else
                {
                    j++;
                }
            }
        }
    }
}
```

ObjectDetector.hpp

```
#ifndef OBJECTDETECTOR_H
#define OBJECTDETECTOR_H

#define INPUT_SIZE_NARROW    600
#define INPUT_SIZE_LONG    1000

#include <string>
#include <caffe/net.hpp>
#include <caffe/common.hpp>
#include <opencv2/core/core.hpp>
#include <iostream>
#include <memory>
#include <map>
```

```
using namespace std;

class ObjectDetector
{
public:

        ObjectDetector(const std::string &model_file, const std::string &weights_file);   //构造函数
    //对一张图片，进行检测，将结果保存进 map 数据结构里,分别表示每个类别对应的目标框，如果需要分数信息，则计算分数
        map<int,vector<cv::Rect> > detect(const cv::Mat& image, map<int,vector<float> >* score=NULL);

private:
    boost::shared_ptr< caffe::Net<float> > net_;
    int class_num_;        //类别数+1     ,官方给的 demo  是 20+1 类
};

#endif
```

源文件 ObjectDetector.cpp

```
#include "ObjectDetector.hpp"
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <vector>
#include <fstream>

using std::string;
using std::vector;
using namespace caffe;
using   std::max;
using std::min;


ObjectDetector::ObjectDetector(const std::string &model_file,const std::string &weights_file){
#ifdef CPU_ONLY
    Caffe::set_mode(Caffe::CPU);
#else
    Caffe::set_mode(Caffe::GPU);
#endif
    net_.reset(new Net<float>(model_file, TEST));
    net_->CopyTrainedLayersFrom(weights_file);
    this->class_num_ = net_->blob_by_name("cls_prob")->channels();   //求得类别数+1
}
```

```
//对一张图片，进行检测，将结果保存进 map 数据结构里,分别表示每个类别对应的目标框，
如果需要分数信息，则计算分数
map<int,vector<cv::Rect> > ObjectDetector::detect(const cv::Mat&
image,map<int,vector<float> >* objectScore){

    if(objectScore!=NULL)    //如果需要保存置信度
        objectScore->clear();

    float CONF_THRESH = 0.8;   //置信度阈值
    float NMS_THRESH = 0.3;    //非极大值抑制阈值
    int max_side = max(image.rows, image.cols);    //分别求出图片宽和高的较大者
    int min_side = min(image.rows, image.cols);
    float max_side_scale = float(max_side) / float(INPUT_SIZE_LONG);      //分别求出缩放因
子
    float min_side_scale = float(min_side) / float(INPUT_SIZE_NARROW);
    float max_scale = max(max_side_scale, min_side_scale);

    float img_scale = float(1) / max_scale;
    int height = int(image.rows * img_scale);
    int width = int(image.cols * img_scale);

    int num_out;
    cv::Mat cv_resized;
    image.convertTo(cv_resized, CV_32FC3);
    cv::resize(cv_resized, cv_resized, cv::Size(width, height));
    cv::Mat mean(height, width, cv_resized.type(), cv::Scalar(102.9801, 115.9465, 122.7717));
    cv::Mat normalized;
    subtract(cv_resized, mean, normalized);

    float im_info[3];
    im_info[0] = height;
    im_info[1] = width;
    im_info[2] = img_scale;
    shared_ptr<Blob<float> > input_layer = net_->blob_by_name("data");
    input_layer->Reshape(1, normalized.channels(), height, width);
    net_->Reshape();
    float* input_data = input_layer->mutable_cpu_data();
    vector<cv::Mat> input_channels;
    for (int i = 0; i < input_layer->channels(); ++i) {
        cv::Mat channel(height, width, CV_32FC1, input_data);
        input_channels.push_back(channel);
        input_data += height * width;
    }
    cv::split(normalized, input_channels);
    net_->blob_by_name("im_info")->set_cpu_data(im_info);
    net_->Forward();                                        //进行网络前向传播


    int num = net_->blob_by_name("rois")->num();      //产生的 ROI 个数,比如为 13949 个 ROI
    const float *rois_data = net_->blob_by_name("rois")->cpu_data();      //维度比如为：13949*5*1*1
    int num1 = net_->blob_by_name("bbox_pred")->num();      //预测的矩形框 维度为 13949*84
    cv::Mat rois_box(num, 4, CV_32FC1);
    for (int i = 0; i < num; ++i)
    {
        rois_box.at<float>(i, 0) = rois_data[i * 5 + 1] / img_scale;
        rois_box.at<float>(i, 1) = rois_data[i * 5 + 2] / img_scale;
        rois_box.at<float>(i, 2) = rois_data[i * 5 + 3] / img_scale;
        rois_box.at<float>(i, 3) = rois_data[i * 5 + 4] / img_scale;
    }

    shared_ptr<Blob<float> > bbox_delt_data = net_->blob_by_name("bbox_pred");      // 13949*84
    shared_ptr<Blob<float> > score = net_->blob_by_name("cls_prob");      // 3949*21

    map<int,vector<cv::Rect> > label_objs;      //每个类别，对应的检测目标框
    for (int i = 1; i < class_num_; ++i){      //对每个类，进行遍历
        cv::Mat bbox_delt(num, 4, CV_32FC1);
        for (int j = 0; j < num; ++j){
            bbox_delt.at<float>(j, 0) = bbox_delt_data->data_at(j, i * 4 + 0, 0, 0);
            bbox_delt.at<float>(j, 1) = bbox_delt_data->data_at(j, i * 4 + 1, 0, 0);
            bbox_delt.at<float>(j, 2) = bbox_delt_data->data_at(j, i * 4 + 2, 0, 0);
            bbox_delt.at<float>(j, 3) = bbox_delt_data->data_at(j, i * 4 + 3, 0, 0);
        }
        cv::Mat box_class = RPN::bbox_tranform_inv(rois_box, bbox_delt);

        vector<RPN::abox> aboxes;      //对于 类别 i，检测出的矩形框保存在这
        for (int j = 0; j < box_class.rows; ++j){
            if (box_class.at<float>(j, 0) < 0)    box_class.at<float>(j, 0) = 0;
            if (box_class.at<float>(j, 0) > (image.cols - 1))      box_class.at<float>(j, 0) = image.cols - 1;
            if (box_class.at<float>(j, 2) < 0)    box_class.at<float>(j, 2) = 0;
            if (box_class.at<float>(j, 2) > (image.cols - 1))      box_class.at<float>(j, 2) = image.cols - 1;
            if (box_class.at<float>(j, 1) < 0)    box_class.at<float>(j, 1) = 0;
            if (box_class.at<float>(j, 1) > (image.rows - 1))    box_class.at<float>(j, 1) = image.rows - 1;
            if (box_class.at<float>(j, 3) < 0)    box_class.at<float>(j, 3) = 0;
            if (box_class.at<float>(j, 3) > (image.rows - 1))    box_class.at<float>(j, 3) = image.rows - 1;
            RPN::abox tmp;
            tmp.x1 = box_class.at<float>(j, 0);
            tmp.y1 = box_class.at<float>(j, 1);
            tmp.x2 = box_class.at<float>(j, 2);
            tmp.y2 = box_class.at<float>(j, 3);
            tmp.score = score->data_at(j, i, 0, 0);
            aboxes.push_back(tmp);
        }
        std::sort(aboxes.rbegin(), aboxes.rend());
        RPN::nms(aboxes, NMS_THRESH);    //与非极大值抑制消除对于的矩形框
        for (int k = 0; k < aboxes.size();){
            if (aboxes[k].score < CONF_THRESH)
                aboxes.erase(aboxes.begin() + k);
            else
                k++;
        }
        //################# 将类别 i 的所有检测框，保存
        vector<cv::Rect> rect(aboxes.size());      //对于类别 i，检测出的矩形框
        for(int ii=0;ii<aboxes.size();++ii)

    rect[ii]=cv::Rect(cv::Point(aboxes[ii].x1,aboxes[ii].y1),cv::Point(aboxes[ii].x2,aboxes[ii].y2));
        label_objs[i]=rect;
        //################# 将类别 i 的所有检测框的打分，保存
        if(objectScore!=NULL){                //################# 将类别 i 的所有检测框的打分，保存
            vector<float> tmp(aboxes.size());          //对于  类别 i，检测出的矩形框的得分
            for(int ii=0;ii<aboxes.size();++ii)
                tmp[ii]=aboxes[ii].score;
            objectScore->insert(pair<int,vector<float> >(i,tmp));
        }
    }
    return label_objs;
}
```

//参考博客 https://blog.csdn.net/zxj942405301/article/details/72775463 中的代码

```
//Python 层修改为
layer {
   name: "proposal"
   type: "RPN"
```

```
    bottom: "rpn_cls_prob_reshape"
    bottom: "rpn_bbox_pred"
    bottom: "im_info"
    top: "rois"
    rpn_param {
        feat_stride : 16
        basesize : 16
        scale : 8
        scale : 16
        scale : 32
        ratio : 0.5
        ratio : 1
        ratio : 2
        boxminsize :16
        per_nms_topn : 0;
        post_nms_topn : 0;
        nms_thresh : 0.3
    }
}
```

主函数

```
#include "ObjectDetector.hpp"
#include<opencv2/opencv.hpp>
#include<iostream>
#include<sstream>
using namespace cv;
using namespace std;
string num2str(float i){
    stringstream ss;
    ss<<i;
    return ss.str();
}

int main(int argc,char **argv){
  ::google::InitGoogleLogging(argv[0]);
#ifdef CPU_ONLY
  cout<<"Use CPU\n";
#else
  cout<<"Use GPU\n";
#endif

  ObjectDetector detect("test.prototxt","1.caffemodel");
```

```
    Mat img=imread("1.jpg");
    map<int,vector<float> > score;
    map<int,vector<Rect> > label_objs=detect.detect(img,NULL);    //目标检测
    for(map<int,vector<Rect> >::iterator it=label_objs.begin();it!=label_objs.end();it++){
        int label=it->first;    //标签
        vector<Rect> rects=it->second;    //检测框
        for(int j=0;j<rects.size();j++){
            rectangle(img,rects[j],Scalar(0,255,0),2);      //画出矩形框
            string txt=num2str(label)+" : "+num2str(score[label][j]);
        }
    }
    imshow("", img);
    waitKey();
    return 0;
}
```

2. RGB&HSI

```
#include <highgui.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <opencv2/objdetect/objdetect.hpp>
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/opencv.hpp"
#include <opencv2/core/core.hpp>
#include<iostream>
#include<vector>        -

using namespace std;                            //使用 C++的命名空间
using namespace cv;                              //使用 opencv 的命名空间
void DrawFire(Mat &inputImg, Mat foreImg)
{
    vector<vector<Point>> contours_set;
    findContours(foreImg, contours_set, CV_RETR_EXTERNAL, CV_CHAIN_APPROX_NONE);
    Mat result0;
    Scalar holeColor;
    Scalar externalColor;
    vector<vector<Point>>::iterator iter = contours_set.begin();
    for (; iter != contours_set.end(); )
    {
        Rect rect = boundingRect(*iter);
        float radius;
        Point2f center;
        minEnclosingCircle(*iter, center, radius);
        if (rect.area() > 0)
        {
            rectangle(inputImg, rect, Scalar(0, 255, 0));
            ++iter;
        }
        else
            iter = contours_set.erase(iter);
    }
    imshow("showFire", inputImg);
    waitKey(0);
}
Mat CheckColor(Mat &inImg)
{
    Mat fireImg;
    fireImg.create(inImg.size(), CV_8UC1);
    int redThre = 115; //115~135
    int saturationTh = 45; //55~65
    Mat multiRGB[3];
    int a = inImg.channels();
    split(inImg, multiRGB); //将图片拆分成 R,G,B,三通道的颜色，将三个通道的数据分别存入矩阵数组 multiRGB 数组中
    for (int i = 0; i < inImg.rows; i++)
    {
        for (int j = 0; j < inImg.cols; j++)
        {
            float B, G, R;
            B = multiRGB[0].at<uchar>(i, j);
            G = multiRGB[1].at<uchar>(i, j);
            R = multiRGB[2].at<uchar>(i, j);
            int maxValue = max(max(B, G), R);
            int minValue = min(min(B, G), R);
            double S = (1 - 3.0*minValue / (R + G + B));
            if ((R > redThre) && (R >= G) && (G >= B) && (S > 0.20) && (S > ((255 - R)*saturationTh / redThre)))
                /*经验公式*/
                fireImg.at<uchar>(i, j) = 255;
            else
                fireImg.at<uchar>(i, j) = 0;
        }
    }
```

```
	dilate(fireImg, fireImg, Mat(5, 5, CV_8UC1));
	imshow("fire", fireImg);
	waitKey(0);
	DrawFire(inImg, fireImg);
	return fireImg;
}


int main()
{
	string filepath = "E:\\fire_0.jpg";
	Mat inputImg = imread(filepath, 1);
	CheckColor(inputImg);
	return 0;
}
```