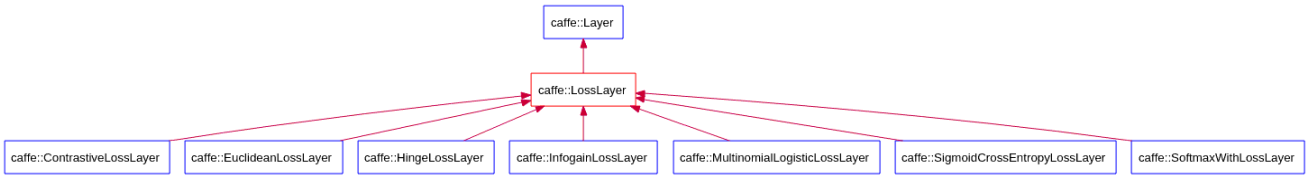
caffe中的损失类

在caffe框架中以将帮我们实现好的损失函数有7个损失函数，嗯，在深度学习领域损失函数的设计也是多种多样，五花八门，可能几个损失函数拼凑在一起再衍生出一个新的损失函数，比如我们可以看到YOLOV3的损失函数，它最终会将定位的回归类损失和识别类别的分类损失加在一起最终得到一个损失。那在这里我们只专注于caffe框架为我们提供的损失函数，关于一些其他的损失函数之后有时间在可以多分析一点。



LossLayer基类

头文件定义

(include/caffe/layers/loss_layer.hpp)

```
22 template <typename Dtype>
23 class LossLayer : public Layer<Dtype> {
24 public:
25     explicit LossLayer(const LayerParameter& param)
26         : Layer<Dtype>(param) {}
27     virtual void LayerSetUp(
28             const vector<Blob<Dtype>*>& bottom, const vector<Blob<Dtype>*>& top);
29     virtual void Reshape(
30             const vector<Blob<Dtype>*>& bottom, const vector<Blob<Dtype>*>& top);
31     //bottom blob的个数就是2，一个是prediction，另外一个是label
32     virtual inline int ExactNumBottomBlobs() const { return 2; }
33
34     /**
35      * @brief For convenience and backwards compatibility, instruct the Net to
36      *        automatically allocate a single top Blob for LossLayers, into which
37      *        they output their singleton loss, (even if the user didn't specify
38      *        one in the prototxt, etc.).
39      */
40     virtual inline bool AutoTopBlobs() const { return true; }
41     virtual inline int ExactNumTopBlobs() const { return 1; }
42     /**
43      * We usually cannot backpropagate to the labels; ignore force_backward for
44      * these inputs.
45      */
46     virtual inline bool AllowForceBackward(const int bottom_index) const {
47         return bottom_index != 1;
48     }
49 };
```

我们可以通过看上述代码知道，输入的Bottom blob的个数为2个，其中一个是我们的预测值，另外一个是label值，另外我们的反向传播只是针对prediction进行反向传播，不针对label进行反向传播。

EuclideanLossLayer

数学公式

欧式距离损失函数的前向传播函数：

$$E = \frac{1}{2N}\sum_{n=1}^{N}||\hat{y}_n - y_n||_2^2 \tag{1}$$

反向传播函数，这里的公式罗列出了对预测值$\hat{y}_n$和目标值y_n的偏导数，那其实我们一般会使用到的是对预测值$\hat{y}_n$的导数。

$$\frac{\partial E}{\partial \hat{y}_n} = \frac{1}{N}\sum_{n=1}^{N}(\hat{y}_n - y_n) \tag{2}$$

$$\frac{\partial E}{\partial y_n} = \frac{1}{N}\sum_{n=1}^{N}(y_n - \hat{y}_n) \tag{3}$$

成员变量

diff_，这个是该Layer自己维护的blob，它主要存放预测值$\hat{y}$和label值y的差值。

Forwad_cpu代码

```
template <typename Dtype>
void EuclideanLossLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
  int count = bottom[0]->count();           //获取每个batch中样本的个数
```

caffe_sub调用封装好的向量减法函数做$\hat{y} - y$的操作。

```
 caffe_sub(
      count,
      bottom[0]->cpu_data(),
      bottom[1]->cpu_data(),
      diff_.mutable_cpu_data());
      //下面调用向量内积函数来求得误差
  Dtype dot = caffe_cpu_dot(count, diff_.cpu_data(), diff_.cpu_data());
  Dtype loss = dot / bottom[0]->num() / Dtype(2);   //获得损失
  top[0]->mutable_cpu_data()[0] = loss;
}
```

Backward_cpu代码

下面的代码需要注意一个地方，就是损失传播的时候为什么是top[0]->cpu_diff()，而不直接是1。因为在实际的使用过程中，我们的各个损失模块还可以构建成一个新的损失，那么这个时候其他的损失就是一些中间层，我们从最终的损失传递误差回来必将通过cpu_diff()来传递误差。

```
template <typename Dtype>
void EuclideanLossLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*>& top,
```

```
    const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom) {
  for (int i = 0; i < 2; ++i) {
    //这里尽管对于prediction和label都进行了反向传播，但是在实际过程中我们对于label值
    //是不进行反向传播的。
    if (propagate_down[i]) {
      const Dtype sign = (i == 0) ? 1 : -1;
      //注意因为损失函数同样也是可以继续当作一个bottom其他层的损失函数的，所以这里反向传播
      //是使用top[0]->cpu_diff()[0]开始反向传播的
      const Dtype alpha = sign * top[0]->cpu_diff()[0] / bottom[i]->num();
      caffe_cpu_axpby(      //该函数的作用:Y = alpha * X + beta * Y
          bottom[i]->count(),              // count
          alpha,                              // alpha
          diff_.cpu_data(),                   // a
          Dtype(0),                           // beta
          bottom[i]->mutable_cpu_diff());  // b
    }
  }
}
```

HingeLossLayer

数学公式

MultinomialLogisticLossLayer