# Caffe自定义神经层

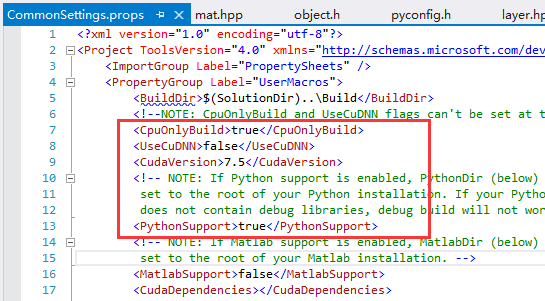
## 1.准备

本实验是在Windows下实现的，对于Windows下编译Caffe，配置Python接口，参考网上有关教程，在这里只说明几点：

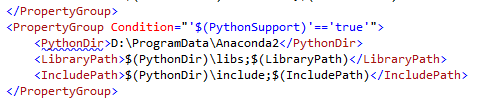
### 1.1下载

下载caffe的windows版本，地址：<https://github.com/Microsoft/caffe>

### 1.2属性配置



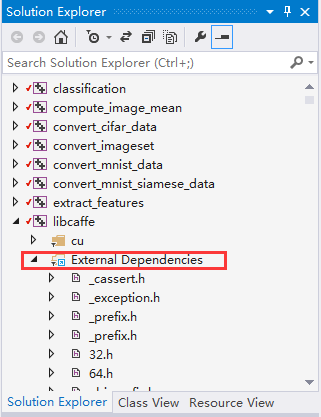
填写Python目录，我这里安装的是anaconda2:



### 1.3报错，打不开文件python27\_d.lib

在编译时，可能会报链接错误，找不到python27\_d.lib，采用如下方法解决：

在解决方案管理器中的External Depandencies中找到pyconfig.h和object.h：

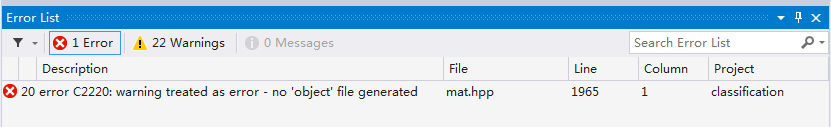


作如下修改：

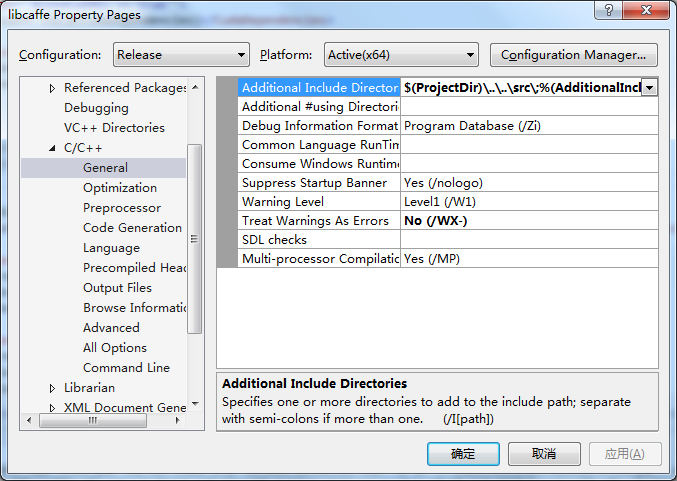
|  |
| --- |
| 1.1 将  #ifdef \_DEBUG  # define Py\_DEBUG  #endif  改为  #ifdef \_DEBUG  //# define Py\_DEBUG  #endif  1.2 将  # ifdef \_DEBUG  # pragma comment(lib,"python27\_d.lib")  # else  # pragma comment(lib,"python27.lib")  # endif /\* \_DEBUG \*/  改为  # ifdef \_DEBUG  # pragma comment(lib,"python27.lib")  # else  # pragma comment(lib,"python27.lib")  # endif /\* \_DEBUG \*/  2. 修改object.h  将  #if defined(Py\_DEBUG) && !defined(Py\_TRACE\_REFS)  #define Py\_TRACE\_REFS  #endif  改为  #if defined(Py\_DEBUG) && !defined(Py\_TRACE\_REFS)  // #define Py\_TRACE\_REFS  #endif |

### 1.4 no ‘object’ file generated

如果报以下错误：



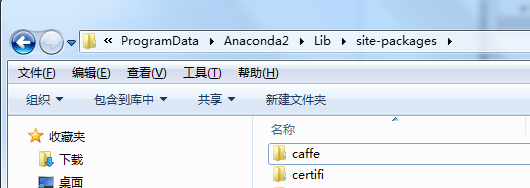
在相应的工程的属性页上，将Treat Waring As Errors改成No



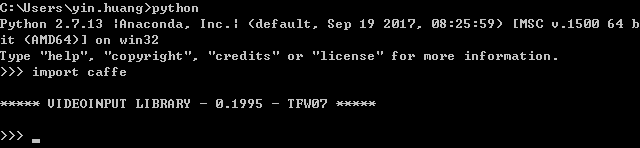
### 1.5 为python添加caffe包

编译完成后，会在目录C:\caffe\Build\x64\Release\pycaffe下生成一个caffe文件夹，将其拷贝到以下目录：

D:\ProgramData\Anaconda2\Lib\site-packages



### 1.6 测试



Caffe成功导入，配置成功。

## 2.自定义神经层

### 2.1 修改proto文件

本实例是自定义一个SquareLayer层，实现层的幂操作，默认做平方操作，该层只有一个设置参数power，默认为2。

修改C:\caffe\src\caffe\proto目录下的caffe.proto文件，添加一个SquareLayerParameter参数，其中只包括一个power参数，默认为2.0：

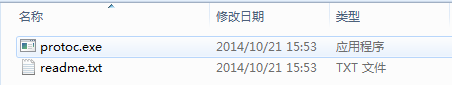
|  |
| --- |
| //新加入的SquareLayer参数  message SquareLayerParameter {  optional float power = 1 [default = 2.0];  } |

在对LayerParmeter添加一个该层参数

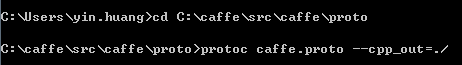
|  |
| --- |
| //新加入的SquareLayer层  optional SquareLayerParameter square\_param = 200; |

### 2.2 编译caffe.proto文件

在<https://developers.google.com/protocol-buffers/> 上下载protoc-2.6.1-win32，解压得：



添加该目录到环境变量，生成caffe.pb.cc和caffe.pb.h





把caffe.pb.h移动到C:\caffe\include\caffe\proto目录

### 2.3 编写SquareLayer类

SquareLayer继承自NeuronLayer类，完成层与层之间的计算，可参考Caffe的PowerLayer层的实现。

主要实现几个函数type,LayerSetUp，Forward\_cpu，Backward\_cpu三个函数。

Type函数，返回层的类型：

|  |
| --- |
| virtual inline const char\* type() const { return "Square"; } |

LayerSetUp，设置层：

|  |
| --- |
| template <typename Dtype>  void SquareLayer<Dtype>::LayerSetUp(const vector<Blob<Dtype>\*>& bottom,  const vector<Blob<Dtype>\*>& top) {  NeuronLayer<Dtype>::LayerSetUp(bottom, top);  CHECK\_NE(top[0], bottom[0]) << this->type() << " Layer does not "  "allow in-place computation.";  power = this->layer\_param\_.square\_param().power();  } |

读取层参数，this->layer\_param\_是LayerParameter层参数，注意参数值的获取方法。

Forward\_cpu，前向传播：

|  |
| --- |
| template <typename Dtype>  void SquareLayer<Dtype>::Forward\_cpu(  const vector<Blob<Dtype>\*>& bottom, const vector<Blob<Dtype>\*>& top) {  const int count = top[0]->count();  const Dtype\* bottom\_data = bottom[0]->cpu\_data();  Dtype\* top\_data = top[0]->mutable\_cpu\_data();  //正向传播，就是从bottom\_data计算top\_data，y = x^a， 这里的x即是bottom\_data,y是top\_data  caffe\_powx(count, bottom\_data, Dtype(power), top\_data);  } |

前向传播函数，即是从bottom\_data计算top\_data，这里完成的计算方法是y=x^a，其中x即是bottom\_data，y是top\_data，a是层的参数，即power。

这里因为计算过程中bottom值不改变，使用一个const Dtype\* bottom\_data指针，指向bottom[0]的数据，使用cpu\_data()返回bottom[0]指针。因为计算过程中top值不改变，使用一个Dtype\* top\_data指针，指向top[0]的数据，使用mutable\_cpu\_data()返回top[0]指针。

Backward\_cpu，反向传播：

|  |
| --- |
| template <typename Dtype>  void SquareLayer<Dtype>::Backward\_cpu(const vector<Blob<Dtype>\*>& top,  const vector<bool>& propagate\_down, const vector<Blob<Dtype>\*>& bottom) {  const int count = top[0]->count();  const Dtype\* top\_diff = top[0]->cpu\_diff();  if (propagate\_down[0]) {  const Dtype\* bottom\_data = bottom[0]->cpu\_data();  Dtype\* bottom\_diff = bottom[0]->mutable\_cpu\_diff();  //反向传播，就是计算bottom\_diff  //1.先计算本层的梯度dy/dx, dy/dx = ax^(a-1)，这里的x即是bottom\_data  caffe\_powx(count, bottom\_data, Dtype(power - 1.0), bottom\_diff);  caffe\_scal(count, Dtype(power), bottom\_diff);  //2.再将本层的梯度与传上来的误差相乘，给bottom\_diff  caffe\_mul(count, bottom\_diff, top\_diff, bottom\_diff);  }  } |

反向传播函数，即是从top\_diff计算bottom\_diff，其中top\_diff是从上层传回来的误差，将top\_diff乘以本层的梯度即得到bottom\_diff。由于本层完成的是y=x^a的计算，其梯度为dy/dx = ax^(a-1)，其中x为bottom\_data。

### 2.4 重新编译Caffe

将SquareLayer的头文件和源文件分别放置到C:\caffe\include\caffe\layers和C:\caffe\src\caffe\layers目录中，并添加到工程中，重新编译Caffe。编译完成后，会在目录C:\caffe\Build\x64\Release\pycaffe下生成一个caffe文件夹，将其拷贝到以下目录：D:\ProgramData\Anaconda2\Lib\site-packages

## 3.测试

这里使用一个简单的网络，输入一张图像，通过打印输入SquareLayer层前后的数据，观察该层是否有效。

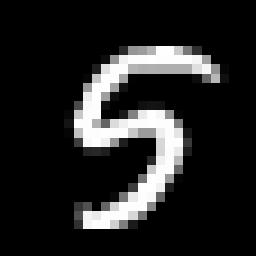
创建一个deploy.prototxt文件：

|  |
| --- |
| name: "CaffeNet" input: "data" input\_shape {  dim: 1 # batchsize  dim: 1 # number of colour channels - rgb  dim: 28 # width  dim: 28 # height }  layer {  name: "square"  type: "Square"  bottom: "data"  top: "data\_out"  square\_param {  power : 2.0  } } |

创建一个SquareLayerTest.py文件：

|  |
| --- |
| **import** numpy **as** np **import** caffe  deploy\_file **=** 'deploy.prototxt' test\_data **=** '5.jpg'  net **=** caffe.Net(deploy\_file, caffe.TEST) transformer **=**caffe.io.Transformer({'data'**:** net.blobs['data'].data.shape}) transformer.set\_transpose('data', (2, 0, 1)) img **=** caffe.io.load\_image(test\_data, color **=** False) net.blobs['data'].data[...] **=** transformer.preprocess('data', img) **print** net.blobs['data'].data[0][0][14] out **=** net.forward() **print** net.blobs['data\_out'].data[0][0][14] |

5.jpg文件：



输出：

|  |
| --- |
| [ 0. 0. 0. 0. 0. 0. 0.  0.01960784 0.76218486 0.9946779 0.99019605 0.99607843 0.99215686  0.99215686 0.78431374 0.61316526 0.60784316 0.93725491 0.98711485  0.90252101 0.09019608 0. 0. 0. 0. 0.  0. 0. ]  [ 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00  0.00000000e+00 0.00000000e+00 0.00000000e+00 3.84467538e-04  5.80925763e-01 9.89384115e-01 9.80488241e-01 9.92172241e-01  9.84375238e-01 9.84375238e-01 6.15148067e-01 3.75971645e-01  3.69473308e-01 8.78446758e-01 9.74395692e-01 8.14544201e-01  8.13533273e-03 0.00000000e+00 0.00000000e+00 0.00000000e+00  0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00] |

可以看到，相应元素被做了平方运算。