Caffe使用II

高级使用——自定义组件

自定义Layer

- 选择继承的类

可以根据自己的需要，从已有的类中选择自己继承的类，这样可以省却很多麻烦。一般可以继承的类包括

层	描述
Layer	如果你要定义的层和已有的层没有什么重叠，那么可以选择直接继承Layer
DataLayer	自定义网络输入层时，可以考虑继承它，内部的load_batch可能是你要重写的函数
NeuronLayer	自定义神经层，也就是中间的进行运算的层
LossLayer	如果现有的损失函数层不能满足需求，可以继承它

*任何一个层都可以被继承，然后进行重写函数。

*尽量确保要实现的功能是否必须要自己写，不然尽量用已有的层，每一个层在caffe/include/caffe/layers源码中都有详细的介绍。

- 步骤：
- 1. 创建新定义的头文件include/caffe/layers/my_neuron_layer.hpp
    - 重新Layer名的方法：virtual inline const char* type() const { return "MyNeuron"; }
    - 如果只是需要cpu方法的话，可以注释掉forward/backward_gpu()这两个方法
- 2. 创建对应src/caffe/src/my_neuron_layer.cpp的源文件
    - 重写方法LayerSetUp,实现从能从prototxt读取参数
    - 重写方法Reshape，如果对继承类没有修改的话，就不需要重写
    - 重写方法Forward_cpu
    - 重写方法Backward_cpu(非必须)
    - * 如果要GPU支持，则还需要创建src/caffe/src/my_neuron_layer.cu，同理重写方法Forward_gpu/Backward_gpu(非必须)

1.自定义计算层

- 3. proto/caffe.proto注册新的Layer

```
message LayerParameter{
...
++ optional MyNeuronParameter
my_neuron_param = 150;
...
}
...
++ message MyNeuronParameter {
++  optional float power = 1 [default = 2];
++ }
...
message V1LayerParameter{
...
++ MYNEURON = 40;
...
}
```

- 4. my_neuron_layer.cpp添加注册的宏定义

```
INSTANTIATE_CLASS(MyNeuronLayer);
REGISTER_LAYER_CLASS(MyNeuron);
```

如果有my_neuron_layer.cu,则添加

```
INSTANTIATE_LAYER_GPU_FUNCS(MyNeuronLayer);
```

- 5. 重新编译和install

- 定义deploy.prototxt

```
name: "CaffeNet"
input: "data"
input_shape {
  dim: 1 # batchsize
  dim: 1 # number of colour channels - rgb
  dim: 28 # width
  dim: 28 # height
}

layer {
  name: "myneuron"
  type: "MyNeuron"
  bottom: "data"
  top: "data_out"
  my_neuron_param {
    power : 2
  }
}
```

2.自定义数据输入层

- 1. 创建新定义的头文件include/caffe/layers/my_data_layer.hpp
  - 重新Layer名的方法：virtual inline const char* type() const { return "MyData"; }
- 2. 重写LayerSetUp,ShuffleImage,load_batch （见代码my_data_layer.cpp）
- 3. proto/caffe.proto注册新的Layer

```
// My new data parameter
optional MyDataParameter my_data_param = 50;


// My new data layer for infer image data
optional MyDataParameter my_data_param = 150;
```

```
// Message that stores parameters used to apply a new data layer
// to infer image data
message MyDataParameter{
  // Image address
  optional string image_address = 1;
  // Start column for calculation
  // Traning and testing use different columns
  optional int32 start_col = 2;
  // End column for computing
  optional int32 end_col = 3;
  // Height of each image
  optional int32 sample_height = 4 [default = 0];
  // Width of each image
  optional int32 sample_width = 5 [default = 0];
  // Shuffle or not
  optional bool shuffle = 6;
  // Batch size for each computing
  optional int32 batch_size = 7;
  // Gray or color (equals to single channel or 3 channels)
  optional bool is_color = 8 [default = false];
  // Save the image seperated from the image address or not
  optional bool is_save = 9 [default = false];
  // If save, use this save folder
  optional string save_folder = 10;
}
```

- 测试新定义的数据层，使用原来的LeNet网络，把输入层做修改

```
layer {
  name: "mnist"
  type: "MyData"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    scale: 0.00390625
  }
  my_data_param {
    image_address: "/home/luoyun/teach_samples/caffe/6th_class/
digits.png"
    batch_size: 64
    start_col:0
    end_col:70
    shuffle:true
    sample_width:20
    sample_height:20
    is_save:true
    save_folder: "/home/luoyun/teach_samples/caffe/6th_class/img"
  }
}
```

```
layer {
  name: "mnist"
  type: "MyData"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  transform_param {
    scale: 0.00390625
  }
  my_data_param {
    image_address: "/home/luoyun/teach_samples/caffe/6th_class/
digits.png"
    batch_size:32
    start_col:70
    end_col:100
    sample_width:20
    sample_height:20
    shuffle:true
  }
}
```

3.自定义损失函数 Softmax与Softmax_loss

- * Softmax回归的相关文档UFLDL：
  http://ufldl.stanford.edu/wiki/index.php/Softmax%E5%9B%9E%E5%BD%92
- softmax用于多分类问题，比如0-9的数字识别，共有10个输出，而且这10个输出的概率和加起来应该为1，所以可以用一个softmax操作归一化这10个输出。进一步一般化，假如共有k个输出，softmax的假设可以形式化表示为：

$$h_\theta(x^{(i)}) = \begin{bmatrix} p(y^{(i)} = 1 | x^{(i)}; \theta) \\ p(y^{(i)} = 2 | x^{(i)}; \theta) \\ \vdots \\ p(y^{(i)} = k | x^{(i)}; \theta) \end{bmatrix} = \frac{1}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \begin{bmatrix} e^{\theta_1^T x^{(i)}} \\ e^{\theta_2^T x^{(i)}} \\ \vdots \\ e^{\theta_k^T x^{(i)}} \end{bmatrix}$$

- Caffe的实现：（softmax_layer.cpp）

```
template <typename Dtype>
void SoftmaxLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<Dtype>*>& top) {
  const Dtype* bottom_data = bottom[0]->cpu_data();
  Dtype* top_data = top[0]->mutable_cpu_data();
  Dtype* scale_data = scale_.mutable_cpu_data();
  int channels = bottom[0]->shape(softmax_axis_);
  int dim = bottom[0]->count() / outer_num_; // C*W*H
  caffe_copy(bottom[0]->count(), bottom_data, top_data);
  // We need to subtract the max to avoid numerical issues, compute the exp,
  // and then normalize.
  for (int i = 0; i < outer_num_; ++i) {
    // initialize scale_data to the first plane
    caffe_copy(inner_num_, bottom_data + i * dim, scale_data);
    for (int j = 0; j < channels; j++) {
      for (int k = 0; k < inner_num_; k++) {
        scale_data[k] = std::max(scale_data[k],
            bottom_data[i * dim + j * inner_num_ + k]);
      }
    }
    // subtraction
    caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, channels, inner_num_,
        1, -1., sum_multiplier_.cpu_data(), scale_data, 1., top_data);
    // exponentiation
    caffe_exp<Dtype>(dim, top_data, top_data);
    // sum after exp
    caffe_cpu_gemv<Dtype>(CblasTrans, channels, inner_num_, 1.,
        top_data, sum_multiplier_.cpu_data(), 0., scale_data);
    // division
    for (int j = 0; j < channels; j++) {
      caffe_div(inner_num_, top_data, scale_data, top_data);
      top_data += inner_num_;
    }
  }
}
```

- **softmax_loss层(只有在train的时候使用)**
- 假设定义一个loss function，就是softmax回归的loss function，形式化如下:

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{j=1}^{k} 1\left\{y^{(i)} = j\right\} \log \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}}\right]$$

- 对于某个样本i，他对应的label是j，那么对于loss function来说，只需要关心第k路是否是一个概率很大的值，所以就用一个I{·}的示性函数来表示只关心第y(i)y(i)路(即label对应的那一路)，其他路都忽略为0。然后log的部分其实就是第k路的概率值取log。
- softmax可以求梯度，梯度的公式是:

$$\nabla_{\theta_j} J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[x^{(i)}\left(1\{y^{(i)} = j\} - p(y^{(i)} = j|x^{(i)};\theta)\right)\right]$$

- Caffe实现（softmax_loss_layer.cpp）：

```
template <typename Dtype>
void SoftmaxWithLossLayer<Dtype>::Forward_cpu(
    const vector<Blob<Dtype>*>& bottom, const vector<Blob<Dtype>*>& top) {
  // The forward pass computes the softmax prob values.
  softmax_layer_->Forward(softmax_bottom_vec_, softmax_top_vec_);
  const Dtype* prob_data = prob_.cpu_data();
  const Dtype* label = bottom[1]->cpu_data();
  int dim = prob_.count() / outer_num_;
  int count = 0;
  Dtype loss = 0;
  for (int i = 0; i < outer_num_; ++i) {
    for (int j = 0; j < inner_num_; j++) {
      const int label_value = static_cast<int>(label[i * inner_num_ + j]);
      if (has_ignore_label_ && label_value == ignore_label_) {
        continue;
      }
      DCHECK_GE(label_value, 0);
      DCHECK_LT(label_value, prob_.shape(softmax_axis_));
      loss -= log(std::max(prob_data[i * dim + label_value * inner_num_ + j],
                           Dtype(FLT_MIN)));
      ++count;
    }
  }
  top[0]->mutable_cpu_data()[0] = loss / get_normalizer(normalization_, count);
  if (top.size() == 2) {
    top[1]->ShareData(prob_);
  }
}
```

4.自定义Solver

- 1. Train的流程
  - 1. caffe train --solver=…
  - 2. 创建Solver，读入参数
  - 3. Solver::Solve()函数

```
template <typename Dtype>
void Solver<Dtype>::Solve(const char* resume_file) {
  Step(param_.max_iter() - iter_);
  //..
    Snapshot();
  //..

  // some additional display
  // ...
}
```

- 4. Solver::Step()函数：

```
template <typename Dtype>
void Solver<Dtype>::Step(int iters) {

  //10000轮迭代
  while (iter_ < stop_iter) {

    // 每隔500轮进行一次测试
    if (param_.test_interval() && iter_ % param_.test_interval() == 0
        && (iter_ > 0 || param_.test_initialization())
        && Caffe::root_solver()) {
      // 测试网络，实际是执行前向传播计算loss
      TestAll();
    }

    // accumulate the loss and gradient
    Dtype loss = 0;
    for (int i = 0; i < param_.iter_size(); ++i) {
      // 执行反向传播，前向计算损失loss，并计算loss关于权值的偏导
      loss += net_->ForwardBackward(bottom_vec);
    }

    // 平滑loss，计算结果用于输出调试等
    loss /= param_.iter_size();
    // average the loss across iterations for smoothed reporting
    UpdateSmoothedLoss(loss, start_iter, average_loss);

    // 通过反向传播计算的偏导更新权值
    ApplyUpdate();

  }
}
```

- 2. 自定义步骤
- 1. 创建新定义的头文件include/caffe/my_solver.hpp
    - 重新Solver名的方法：virtual inline const char* type() const { return "My"; }
- 2. 创建新定义的源文件src/caffe/solvers/my_solver.cpp
- 3. 不需要注册到caffe.proto，如果需要额外增加参数，直接增加到SolverParameter就可以。

```
message SolverParameter{
...
++ [自定义参数]
...
}
```

- 4. my_solver.cpp底部

```
INSTANTIATE_CLASS(MySolver);
REGISTER_SOLVER_CLASS(My);
```

- *注意：命名必须严格按照[Name]Solver的命名方式，不然编译会无法识别。