caffe中的全连接神经层实现

我们在这一部分主要是分析全连接层的前向传播和反向传播，因为caffe是以Layer为基础进行模块化的计算的，所以激活函数并不是全连接层所需要考虑的范畴。那其实这个全连接层主要的工作就是矩阵的线性变换，那我们具体来看一下全连接层的数学原理和代码实现吧。

数学原理

全连接层主要做的工作为：

$$Z^{[l]} = A^{[l-1]}W^{[l]} + b^{[l]} \tag{1}$$

该公式就是一个矩阵的线性变换，那涉及到矩阵计算我们必须要对公式中每个变量的维度有一个非常清晰的认识，下面简单罗列一下其维度：

$$Z^{[l]} \in (m, n^l) \tag{2}$$

$$W^{[l]} \in (n^{l-1}, n^l) \tag{3}$$

$$A^{[l-1]} \in (m, n^{l-1}) \tag{4}$$

$$b^{[l]} \in (1, n^l) \tag{5}$$

那它的反向传播为：

$$dW^{[l]} = \frac{\partial E}{\partial W[l]} = A^{[l-1]T}dZ^{[l]} \tag{6}$$

$$db^{[l]} = \frac{\partial E}{\partial b^{[l]}} = \sum_{i=1}^{m} dZ^{[l](i)} \tag{7}$$

$$dA^{[l-1]} = \frac{\partial E}{\partial A^{[l-1]}} = dZ^{[l]}W^{[l]T} \tag{8}$$

简单说明一下，根据Layer层次结构的介绍以及Net结构介绍中我们也可以知道在一个Layer类及其派生类中自己维护的blob存放的数据都是权重和偏执数据，而我们的训练数据一般都是在Net中的blob进行相应的维护。所以在这里$W^{[l]}$、$dW^{[l]}$、$b^{[l]}$和$db^{[l]}$都是在Layer层的blob进行维护的。一个Layer的Bottom blob中维护的数据为$A^{[l-1]}$和$dA^{[l-1]}$。Top blob中维护的数据为$Z^{[l]}$和$dZ^{[l]}$。下面具体看一下代码吧。

代码实现

LayerSetUp代码分析

对于LayerSetUp这个函数主要做三件事情：

1、确定上一层Layer和本层Layer的神经元个数

2、确定权重weight和偏置bias的尺寸大小，并创建相应的blob。

3、使用Filler类进行初始化。

后面我将结合前面的数学分析来代码。

```
template <typename Dtype>
void InnerProductLayer<Dtype>::LayerSetUp(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top) {
    //确定本层Layer的神经元个数，这个是需要用户自己指定的
    const int num_output = this->layer_param_.inner_product_param().num_output();
    //是否偏置项，一般我们都是使用有偏置项这一选项的。
    bias_term_ = this->layer_param_.inner_product_param().bias_term();
    //transpose是否为true，这个tranpose主要是针对weight矩阵
    transpose_ = this->layer_param_.inner_product_param().transpose();
```

如果代码中的transpose_为true的话，则weight矩阵对应的维度为(n^{l-1}, n^l)。如果是false的话，则weight矩阵对应的维度为(n^l, n^{l-1})。为了和上面的数学公式所对应，这边的transpose都是设置为true。当然在后面我们可以看到针对transpose是如何处理weight维度的。

```
        N_ = num_output;
        //在全连接层参数InnerProductParameter中axis这个参数的默认值都是设置为1
    const int axis = bottom[0]->CanonicalAxisIndex(
            this->layer_param_.inner_product_param().axis());
        //我们知道我们的上一层的数据的维度为N×C×H×W，我们调用的blob的count(1)函数，
        //相当于获得其C×H×W的值，因此这个K_其实就是上一层Layer的神经元个数，这里注意
        //一下，我们对bottom的数据进行了一个维度变化，将4维数据变成2维了
    K_ = bottom[0]->count(axis);
    if (this->blobs_.size() > 0) {
        LOG(INFO) << "Skipping parameter initialization";
    } else {
        //注意如果有偏置项的话则blobs_数组中应该有两个blob，其中blobs_[0]应该存放着
        //权重weight,而blobs_[1]应该存放着bias
        if (bias_term_) {
            this->blobs_.resize(2);
        } else {
            this->blobs_.resize(1);
        }
        // 我们需要确立weight的形状，如果transpose_为true的话，则weight_shape为(K_, N_),
        //其中K_是上一层的神经元个数，而N_是当前层的神经元个数，所以这就和上面的数学公式对应
            //上了
        vector<int> weight_shape(2);
        if (transpose_) {
            weight_shape[0] = K_;
            weight_shape[1] = N_;
        } else {
            weight_shape[0] = N_;
            weight_shape[1] = K_;
        }
        //创建权重的blob对象，并对其进行初始化动作，初始化相关内容请参考《caffe中的Filler类》这一节
```

```
        this->blobs_[0].reset(new Blob<Dtype>(weight_shape));
        shared_ptr<Filler<Dtype> > weight_filler(GetFiller<Dtype>(
                    this->layer_param_.inner_product_param().weight_filler()));
        weight_filler->Fill(this->blobs_[0].get());
        // 创建bias_的blob对象，一般的向量都是列向量
        if (bias_term_) {
            vector<int> bias_shape(1, N_);
            this->blobs_[1].reset(new Blob<Dtype>(bias_shape));
            shared_ptr<Filler<Dtype> > bias_filler(GetFiller<Dtype>(
                        this->layer_param_.inner_product_param().bias_filler()));
            bias_filler->Fill(this->blobs_[1].get());
        }
    }  // parameter initialization
        //这里的param_propagate_down数组的大小为2，其值都是true，说明weight和bias都是需要反向传播的
    this->param_propagate_down_.resize(this->blobs_.size(), true);
}
```

Reshape代码分析

全连接层的Reshape函数的主要作用是根据输入bottom的blob维度信息，来确定top中的blob维度信息

```
template <typename Dtype>
void InnerProductLayer<Dtype>::Reshape(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top) {
    const int axis = bottom[0]->CanonicalAxisIndex(
            this->layer_param_.inner_product_param().axis());
    const int new_K = bottom[0]->count(axis);
    CHECK_EQ(K_, new_K)
        << "Input size incompatible with inner product parameters.";
    //计算这个batch的大小
    M_ = bottom[0]->count(0, axis);
        //将top的blob的维度设置为(M_, N_)。
    vector<int> top_shape = bottom[0]->shape();
    top_shape.resize(axis + 1);
    top_shape[axis] = N_;
    top[0]->Reshape(top_shape);
    // 这里的bias_multiplier的主要作用是我们计算公式(7)的时候需要用到这么一个bias_multiplier
    if (bias_term_) {
        vector<int> bias_shape(1, M_);
        bias_multiplier_.Reshape(bias_shape);
        caffe_set(M_, Dtype(1), bias_multiplier_.mutable_cpu_data());
    }
}
```

Forward_cpu代码分析

```
void InnerProductLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
        const vector<Blob<Dtype>*>& top) {
    //输入的bottom数据，相当于A^(l-1)的数据
    const Dtype* bottom_data = bottom[0]->cpu_data();
    //需要计算的top数据,相当于Z^(l)的数据
```

```
    Dtype* top_data = top[0]->mutable_cpu_data();
    //权重数据
    const Dtype* weight = this->blobs_[0]->cpu_data();
    //调用矩阵乘法函数来进行计算，从而获得Z^(1)的数据
    caffe_cpu_gemm<Dtype>(CblasNoTrans, transpose_ ? CblasNoTrans : CblasTrans,
            M_, N_, K_, (Dtype)1.,
            bottom_data, weight, (Dtype)0., top_data);
    if (bias_term_) {
        //如果有偏置项的话，则进行下面的操作相当于两个向量相乘，(M_, 1)×(1, N_) =
        //(M_, N_),而这个维度正好是Z^(1)的维度，因此两者一相加相当于加上了对应每个元素加上了偏置项
        caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, M_, N_, 1, (Dtype)1.,
                bias_multiplier_.cpu_data(),
                this->blobs_[1]->cpu_data(), (Dtype)1., top_data);
    }
}
```

Backward_cpu代码分析

```
template <typename Dtype>
void InnerProductLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*>& top,
        const vector<bool>& propagate_down,
        const vector<Blob<Dtype>*>& bottom) {
```

下面的代码就是计算$dW^{[l]} = \frac{\partial E}{\partial W^{[l]}} = A^{[l-1]T} dZ^{[l]}$的梯度信息，因为我们选择了transpose_因此代码和数学公式是可以一一对应上的。

```
 if (this->param_propagate_down_[0]) {
        const Dtype* top_diff = top[0]->cpu_diff();
        const Dtype* bottom_data = bottom[0]->cpu_data();
        // Gradient with respect to weight
        if (transpose_) {
            caffe_cpu_gemm<Dtype>(CblasTrans, CblasNoTrans,
                    K_, N_, M_,
                    (Dtype)1., bottom_data, top_diff,
                    (Dtype)1., this->blobs_[0]->mutable_cpu_diff());
        } else {
            caffe_cpu_gemm<Dtype>(CblasTrans, CblasNoTrans,
                    N_, K_, M_,
                    (Dtype)1., top_diff, bottom_data,
                    (Dtype)1., this->blobs_[0]->mutable_cpu_diff());
        }
    }
```

下面这段代码就是计算$db^{[l]} = \frac{\partial E}{\partial b^{[l]}} = \sum_{i=1}^{m} dZ^{[l](i)}$的信息

```
 if (bias_term_ && this->param_propagate_down_[1]) {
        const Dtype* top_diff = top[0]->cpu_diff();
        caffe_cpu_gemv<Dtype>(CblasTrans, M_, N_, (Dtype)1., top_diff,
                bias_multiplier_.cpu_data(), (Dtype)1.,
                this->blobs_[1]->mutable_cpu_diff());
    }
```

下面这一段代码是计算bottom的$dA^{[l-1]} = \frac{\partial E}{\partial A^{[l-1]}} = dZ^{[l]}W^{[l]T}$的信息

```
if (propagate_down[0]) {
        const Dtype* top_diff = top[0]->cpu_diff();
        if (transpose_) {
            caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasTrans,
                    M_, K_, N_,
                    (Dtype)1., top_diff, this->blobs_[0]->cpu_data(),
                    (Dtype)0., bottom[0]->mutable_cpu_diff());
        } else {
            caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans,
                    M_, K_, N_,
                    (Dtype)1., top_diff, this->blobs_[0]->cpu_data(),
                    (Dtype)0., bottom[0]->mutable_cpu_diff());
        }
    }
}
```

上面就是整个全连接神经网络的前向传播和反向传播的数学公式和代码，上述并没有讲GPU端的代码，因为GPU代码和CPU代码差不多，因为使用GPU计算矩阵乘法的API和CPU端的类似，所以并没有介绍，如果有兴趣可以看看cuda代码。另外关于矩阵乘法的相关API的使用可以参考《caffe的矩阵乘法函数》一节。