Caffe In Depth Documentation

Release 0.0.1

Alpesis

Dec 21, 2017

Contents

Table of Contents:

CHAPTER 1

Data

1.1 Proto

1.2 Blob

CHAPTER 2

Network

2.1 Net

2.1.1 Overview

Functions

2.1.2 Net

(net) Init

Steps:

1. get **root_net_**
2. get **phase_**
3. FilterNet()
4. filter layers

2.2 Solver

2.2.1 Overview

Functions

2.2.2 Factory

SolverRegistry

SolverRegisterer

2.2.3 Solver

Solver

WorkerSolver

2.2.4 Stochastic Gradient Decents

SGDSolver

(public) Interfaces

(sgdsolver) SGDSolver

(sgdsolver) type

(sgdsolver) history

(protected) Solve

(sgdsolver) PreSolve

- history_
- update_
- temp_

(sgdsolver) ApplyUpdate

Steps:

```
1. rate = GetLearningRate()

2. if (display && iter % display == 0): logging

3. ClipGradients()
```

```
4. for params_ids:
    - Normalize(param_id)
    - Regularize(param_id)
    - ComputeUpdateValue(param_id, rate)

5. net_->Update()
```

Source Codes:

```
CHECK(Caffe::root_solver());

Dtype rate = GetLearningRate();

if (this->param_.display() && this->iter_ % this->param_.display() == 0)
{
    LOG(INFO) << "Iteration " << this->iter_ << ", lr = " << rate;
}

ClipGradients();

for (int param_id = 0; param_id < this->net_->learnable_params().size(); ++param_id)
{
    Normalize(param_id);
    Regularize(param_id);
    ComputeUpdateValue(param_id, rate);
}

this->net_->Update();
```

(sgdsolver) GetLearningRate

Learning Rate Decay Policy:

- fixed: `fixed = base_lr`
- step: `step = base_lr * gamma ^ (floor( iter/step ))`
- exp: `exp = base_lr * gamma ^ iter`
- inv: `inv = base_lr * (1 + gamma * iter) ^ (- power)`
- multistep: similar to `step`, but it allows non uniform steps defined by stepvalue
- poly: `poly = base_lr * (1 - iter/max_iter) ^ (power)`, a polynomial decay
- sigmoid: `sigmoid = base_lr * ( 1 / ( 1 + exp(-gamma * (iter - stepsize))))`, a sigmoid decay

Parameters:

- `base_lr`: solver param
- `max_iter`: solver param
- `gamma`: solver param
- `step`: solver param
- `stepvalue`: solver param
- `power`: solver param

- `iter`: the current iteration

(rate) fixed

- input: `base_lr`
- output: rate
- **calculations:**
  - rate = base_lr

```
rate = this->param_.base_lr()
```

(rate) step

- **inputs:**
  - base_lr
  - gamma
  - iter
  - step
- **output:**
  - rate
- **calculations:**
  - rate = base_lr * gamma ^ (floor (iter / step))

```
this->current_step_ = this->iter_ / this->param_.stepsize();
rate = this->param_.base_lr() * pow(this->param_.gamma(), this->current_step_);
```

(rate) exp

- **inputs:**
  - base_lr
  - gamma
  - iter
- **output:**
  - rate
- **calculations:**
  - rate = base_lr * gamma ^ iter

```
rate = this->param_.base_lr() * pow(this->param_.gamma(), this->iter_);
```

(rate) inv

- **inputs:**
  - base_lr
  - gamma
  - power
  - iter
- **output:**
  - rate
- **calculations:**
  - rate = base_lr * (1 + gamma * iter) ^ (- power)

For example:

```
- inputs:
    - base_lr: 0.01
    - gamma: 0.0001
    - power: 0.75
    - iter: 0
- output:
    - rate
- calculations:
    - rate = 0.01 * (1 + 0.0001 * 0) ^ (- .075) = 0.01
```

Source codes:

```
rate = this->param_.base_lr() * pow(Dtype(1) + this->param_.gamma() * this->iter_, -␣
↪this->param_.power());
```

(rate) multistep

- **inputs:**
  - base_lr
  - gamma
  - stepvalue
  - iter
  - current_step
- **output:**
  - rate
- **calculations:**
  - if (current_step < stepvalue && iter >= stepvalue(current_step)): current_step++
  - rate = base_lr * gamma ^ (floor(current_step))

```
if (this->current_step_ < this->param_.stepvalue_size() &&
    this->iter_ >= this->param_.stepvalue(this->current_step_))
{
  this->current_step_++;
  LOG(INFO) << "MultiStep Status: Iteration " <<
  this->iter_ << ", step = " << this->current_step_;
}
rate = this->param_.base_lr() * pow(this->param_.gamma(), this->current_step_);
```

(rate) poly

- **inputs:**
  - base_lr
  - power
  - max_iter
  - iter
- **output:**
  - rate
- **calculations:**
  - rate = base_lr * (1 - iter/max_iter) ^ (power)

```
rate = this->param_.base_lr() * pow(Dtype(1.) - (Dtype(this->iter_) / Dtype(this->
↪param_.max_iter())), this->param_.power());
```

(rate) sigmoid

- **inputs:**
  - base_lr
  - gamma
  - stepsize
  - iter
- **output:**
  - rate
- **calculations:**
  - rate = base_lr * (1 / (1 + exp(-gamma * (iter - stepsize))))

```
rate = this->param_.base_lr() * (Dtype(1.) / (Dtype(1.) + exp(-this->param_.gamma() *␣
↪(Dtype(this->iter_) - Dtype(this->param_.stepsize())))));
```

(sgdsolver) ClipGradients

- **inputs:**
  - clip_graidents
  - net_params
- **output:**
  - net_params[i]->scale_diff(net_params)
- **calculations:**
  - **clip_gradients:**
    - clip_gradients = param.clip_gradients()
    - if clip_gradients < 0, return
  - **net_params & sumsq_diff**
    - net_params = **net_**->learnable_params()
    - for net_params: sumsq_diff += net_params[i]->sumsq_diff()
  - **l2norm_diff**
    - l2norm_diff = sqrt(sumsq_diff)
    - **if l2norm_diff > clip_gradients:**
      - scale_factor = clip_gradients / l2norm_diff
      - for net_params: net_params[i]->scale_diff(scale_factor)

```
std::cout << "(SGDSolver) ClipGradients: " << std::endl;

const Dtype clip_gradients = this->param_.clip_gradients();
if (clip_gradients < 0) { return; }

const vector<Blob<Dtype>*>& net_params = this->net_->learnable_params();
Dtype sumsq_diff = 0;
for (int i = 0; i < net_params.size(); ++i)
{
   sumsq_diff += net_params[i]->sumsq_diff();
}

const Dtype l2norm_diff = std::sqrt(sumsq_diff);
if (l2norm_diff > clip_gradients)
{
    Dtype scale_factor = clip_gradients / l2norm_diff;
    LOG(INFO) << "Gradient clipping: scaling down gradients (L2 norm "
              << l2norm_diff << " > " << clip_gradients << ") "
              << "by scale factor " << scale_factor;
    for (int i = 0; i < net_params.size(); ++i)
    {
       net_params[i]->scale_diff(scale_factor);
    }
}
```

(sgdsolver) Normalize

(sgdsolver) Regularize

- **inputs:**
  - regularization_type

L2:

- inputs:
- output:
- **calculations:**
  - weight_decay = **param_**.weight_decay()
  - net_params_weight_decay = this->net_->params_weight_decay()
  - local_decay = weight_decay * net_params_weight_decay[param_id]
  - diff = local_decay * data + diff

L1:

- inputs:
- output:
- **calculations:**
  -

(sgdsolver) ComputeUpdateValue

- **inputs:**
  - net_params_lr
  - momentum
- **output:**
  - net_params_diff
- **calculations:**
  - local_rate = rate * net_params_lr[param_id]
  - **history_** = local_rate * diff + momentum * **history_**
  - net_params_diff = **history_**

```
std::cout << "(SGDSolver) ComputeUpdateValue: " << std::endl;

const vector<Blob<Dtype>*>& net_params = this->net_->learnable_params();
const vector<float>& net_params_lr = this->net_->params_lr();

Dtype momentum = this->param_.momentum();
Dtype local_rate = rate * net_params_lr[param_id];


   case Caffe::CPU:
```

```
  {
    caffe_cpu_axpby(net_params[param_id]->count(),
                    local_rate,
                    net_params[param_id]->cpu_diff(),
                    momentum,
                    history_[param_id]->mutable_cpu_data());
    caffe_copy(net_params[param_id]->count(),
               history_[param_id]->cpu_data(),
               net_params[param_id]->mutable_cpu_diff());
    break;
  }
```

(protected) Snapshot and Restore

(sgdsolver) SnapshotSolverState

(sgdsolver) SnapshotSolverStateToBinaryProto

(sgdsolver) SnapshotSolverStateToHDF5

(sgdsolver) RestoreStateFromBinaryProto

(sgdsolver) RestoreStateFromHDF5

NesterovSolver

AdaGradSolver

RMSPropSolver

```
31      231   // RMSProp decay value
32  232   // MeanSquare(t) = rms_decay*MeanSquare(t-1) + (1-rms_
↪decay)*SquareGradient(t)
33  233   optional float rms_decay = 38;
```

AdaDeltaSolver

AdamSolver

2.3 Layers

2.3.1 Convolutions

Overview

Demo

1. Forward

transformation

- (im2col) data_im -> (im2col) -> col_buff

formula

(gemm) C := alpha * A * B + beta * C

- (weights: C = A * B) weights * col_buff = top_data
- (bias: C = A * B + C) bias * **bias_multiplier_** + top_data = top_data

shapes

- data_im: (C, H, W)
- col_buff: (kernel_channels * kernel_h * kernel_w, output_h * output_w)
- weights: (**output_channels_** / **group_**, kernel_channels * kernel_h * kernel_w)
- bias: (**num_output_**, 1)
- **bias_multiplier_**: (1, output_h * output_w)
- top_data: (**num_output_**, output_h * output_w)

1.1. (forward) im2col

- **inputs:**
  - data_im: (C, H, W)
  - pad: (h, w)
  - kernel: (h, w)
  - stride: (h, w)
  - dilation: (h, w)
  - data_col: (**kernel_dim_**, output_h * output_w)

- **output:**
  - col_buff: (**kernel_dim_**, output_h * output_w)
- **calculations:**
  - **output_shape: data_col(kernel_dim_, output_h * output_w)**
    - **kernel_dim_** = channels * kernel_h * kernel_w
    - output_h = (height + 2 * pad_h - (dilation_h * (kernel_h - 1) + 1)) / stride_h + 1
    - output_w = (width + 2 * pad_w - (dilation_w * (kernel_w - 1) + 1)) / stride_w + 1
  - **data_im -> data_col**
    - loop (TODO)

1.2. (forward) weights

- **inputs:**
  - (weights) weights + **weight_offset_** * g (**conv_out_channels_** / **group_**, **kernel_dim_**)
  - (col_buff) col_buff + **col_offset_** * g (**kernel_dim_**, output_h * output_w)
- **output:**
  - (output) output + **output_offset_** * g (**conv_out_channels_** / **group_**, **conv_out_spatial_dim_**)
- **calculations:**
  - **shapes:**
    - weights: (**conv_out_channels_** / **group_**, **kernel_dim_**)
    - col_buff: (**kernel_dim_**, output_h * output_w)
    - output: (**conv_out_channels_** / **group_**, **conv_out_spatial_dim_**)
    - **calculations:**
      - **offsets:**

        weight_offset_ = **conv_out_channels_** / **group_** * **kernel_dim_**

        col_offset_ = **kernel_dim_** * output_h * output_w

        output_offset_ = **conv_out_channels_** /**group_** * **conv_out_spatial_dim_**
      - **dimensions:**

        conv_out_channels_:

        kernel_dim_ = channels * kernel_h * kernel_w

        output_h = (height + 2 * pad_h - (dilation_h * (kernel_h - 1) + 1)) / stride_h + 1

        output_w = (width + 2 * pad_w - (dilation_w * (kernel_w - 1) + 1)) / stride_w + 1

        conv_out_spatial_dim_ = output_h * output_w
  - **gemm:**
    - output + **output_offset_** * g = (weights + **weight_offset_** * g) * (col_buff + **col_offset_** * g)

1.3. (forward) bias

- **inputs:**
  - biases: (**num_output_**, 1)
  - **bias_multiplier_**: (1, output_h * output_w)
  - output: (**num_output_**, output_h * output_w)
- **output:**
  - output: (**num_output_**, output_h * output_w)
- **calculations:**
  - (gemm) output = bias * **bias_multiplier_** + output

2. Backward

Data

Blob

blobs

- blobs_[0]: weights
- blobs_[1]: bias
- bottom
- top

blobs: size

- bottom.size()
- top.size()

blobs: shape, dimensions, spatial dimensions

- bottom/**top_shape_**: (channels, height, width)
- bottom/**top_dim_**: channels * height * width
- bottom/top/**conv_spatial_dim_**: height * width

(proto) ConvolutionParameter

1. Outputs:

Constraint	Type	Variable	No.	Default	Description
optional	uint32	num_output	1		the number of outputs for the layer

2. Forward/Backward:

2.1. Weights:

Constraint	Type	Variable	No.	Default	Description
optional	FillerParameter	weight_filler	7		
optional	FillerParameter	bias_filler	8		
optional	bool	bias_term	2	True	whether to have bias terms

2.2. Convolution:

Constraint	Type	Variable	No.	Default	Description
repeated	uint32	pad	3	0	
repeated	uint32	kernel_size	4		
repeated	uint32	stride	6	1	
repeated	uint32	dilation	18	1	
optional	uint32	pad_h	9	0	for 2D only
optional	uint32	pad_w	10	0	for 2D only
optional	uint32	kernel_h	11		
optional	uint32	kernel_w	12		
optional	uint32	stride_h	13		
optional	uint32	stride_w	14		
optional	uint32	group	5	1	the group size for group conv

2.3. im2col/col2im:

Constraint	Type	Variable	No.	Default	Description
optional	int32	axis	16	1	
optional	bool	force_nd_im2col	17	False	

3. Engine:

Constraint	Type	Variable	No.	Default
optional	Engine	engine	15	default, caffe, cudnn

(layer) ConvolutionParameters

Constraint	Type	Variable	Default
public	virtual inline const char*	type()	Convolution
protected	virtual inline bool	reverse_dimensions()	False

(layer) BaseConvolutionParameters

1. public blobs

Constraint	Type	Variable	Default
public	virtual inline int	MinBottomBlobs() const	1
public	virtual inline int	MinTopBlobs() const	1
public	virtual inline bool	EqualNumBottomTopBlobs() const	True

2. config params for LayerSetUp

Constraint	Type	Variable	Default	Remark
protected	Blob<int>	**kernel_shape_**		conv_param
protected	Blob<int>	**stride_**		conv_param
protected	Blob<int>	**pad_**		conv_param
protected	Blob<int>	**dilation_**		conv_param
protected	int	**channel_axis_**		conv_param
protected	int	**group_**	1	conv_param / gemm
protected	int	**num_output_**		conv_param / bias
protected	bool	**force_nd_im2col_**		conv_param
protected	bool	**is_1x1_**		
protected	bool	**bias_term_**		
protected	int	**num_**		
protected	int	**channels_**		output_channels
protected	int	**num_spatial_axes_**	2	
protected	int	**weight_offset_**		gemm
private	int	**conv_out_channels_**		
private	int	**conv_in_channels_**		
private	int	**kernel_dim_**		

3. interim params for Reshape

Constraint	Type	Variable	Default	Remark
protected	Blob<int>	**conv_input_shape_**		
protected	vector<int>	**col_buffer_shape_**		
protected	const vector<int>*	**bottom_shape_**		
protected	int	**bottom_dim_**		
protected	int	**top_dim_**		
protected	int	**out_spatial_dim_**		bias
private	int	**num_kernels_im2col_**		
private	int	**num_kernels_col2im_**		
private	int	**conv_out_spatial_dim_**		
private	int	**col_offset_**		gemm
private	int	**output_offset_**		
private	Blob<Dtype>	**col_buffer_**		
private	Blob<Dtype>	**bias_multiplier_**		

4. input/output params

Constraint	Type	Variable	Default	Remark
protected	inline int	input_shape		
protected	virtual bool	reverse_dimensions	0	
protected	vector<int>	**output_shape_**		compute_output_shape

Functions

(layer) Convolution

(conv) compute_output_shape

Calculations:

- **inputs:**
  - input_dim: (height, width)
  - kernel: (height, width)
  - pad: (height, width)
  - stride: (height, width)
  - dilation: (height, width)
- **outputs:**
  - output_dim: (height, width)
- **calculations:**
  - kernel_extent = dilation * (kernel - 1) + 1
  - output_dim = (input_dim + 2 * pad - kernel_extent) / stride + 1

Extensions:

- kernel_extent_h = dilation_h * (kernel_h - 1) + 1
- output_h = (input_h + 2 * pad_h - kernel_extent_h) / stride_h + 1
- kernel_extent_w = dilation_w * (kernel_w + 1) + 1
- output_w = (input_w + 2 * pad_w - kernel_extent_w) / stride_w + 1

For example:

```
- inputs:
  - input_dim: (224, 224)
  - kernel: (3, 3)
  - pad: (1, 1)
  - stride: (1, 1)
  - dilation: (1, 1)

- output: output_shape_: (224, 224)
```

```
- calculations:
  - kernel_extent = 1 * (3 - 1) + 1 = 3
  - output_shape_ = (224 + 2 * 1 - 3) / 1 + 1 = 224
```

(conv) Forward_cpu

(conv) Backward_cpu

(conv) Forward_gpu

(conv) Backward_gpu

(layer) BaseConvolution

(base_conv) LayerSetUp

1. (conv_param) force_nd_im2col_

```
force_nd_im2col_ = conv_param.force_nd_im2col();
```

2. (conv_param) channel_axis_

```
channel_axis_ = bottom[0]->CanonicalAxisIndex(conv_param.axis());
```

3. (bottom[0]) num_spatial_axes_

```
first_spatial_axis = channel_axis_ + 1
num_axes = bottom[0]->num_axes()
num_spatial_axes_ = num_axes - first_spatial_axis
CHECK: num_spatial_axes_ >= 0

i.e. bottom[0].shape: (1, 3, 600, 800)

- first_spatial_axis = 1 + 1 = 2
- num_spatial_axes_ = 4 - 2 = 2
```

4. (bottom[0]) bottom_dim_blob_shape

```
bottom_dim_blob_shape(1, num_spatial_axes_ + 1)
```

5. (bottom[0]) spatial_dim_blob_shape

```
spatial_dim_blob_shape(1, max(num_spatial_axes_, 1))
```

6. (conv_param) kernel

```
kernel_shape_.Reshape(spatial_dim_blob_shape)
kernel_h, kernel_w
```

7. (conv_param) stride

```
stride_shape_.Reshape(spatial_dim_blob_shape)
stride_h, stride_w
```

8. (conv_param) pad

```
pad_shape_.Reshape(spatial_dim_blob_shape)
pad_h, pad_w
```

9. (conv_param) dilation

```
dilation_shape_.Reshape(spatial_dim_blob_shape)
dilation_h, dilation_w
```

10. (im2col) is_1x1_

```
// Special case: im2col is the identity for 1x1 convolution with stride 1
// and no padding, so flag for skipping the buffer and transformation.
is_1x1_ = true;
for (int i = 0; i < num_spatial_axes_; ++i)
{
  is_1x1_ &= kernel_shape_data[i] == 1 && stride_data[i] == 1 && pad_data[i] == 0;
  if (!is_1x1_) { break; }
}
```

11. (bottom[0]) channels_

```
channels_ = bottom[0]->shape(channel_axis_)
```

12. (conv_param) num_output_

```
num_output_ = this->layer_param_.convolution_param().num_output()
CHECK: num_output_ > 0
```

13. (conv_param) group_

```
group_ = this->layer_param_.convolution_param().group();
CHECK_EQ(channels_ % group_, 0);
CHECK_EQ(num_output_ % group_, 0) << "Number of output should be multiples of group.";
```

14. (conv) conv_out_channels_, conv_in_channels_

```
if (reverse_dimensions())
{
  conv_out_channels_ = channels_;
  conv_in_channels_ = num_output_;
}
else
{
  conv_out_channels_ = num_output_;
  conv_in_channels_ = channels_;
}
```

15. (conv) blobs_, weights, bias

blobs_:

- blobs_[0]: weights
- blobs_[1]: biases

weights:

- weight_shape[0] = **conv_out_channels_**
- weight_shape[1] = **conv_in_channels_** / **group_**
- data: (conv_param) weight_filler
- **kernel_dim_** = weights.count(1)
- **weight_offset_** = **conv_out_channels_** * **kernel_dim_** / **group_**

biases:

- **bias_term_**: (conv_param)
- bias_shape: (**bias_term_**, **num_output_**)
- data: (conv_param) bias_filler

```
// Handle the parameters: weights and biases.
// - blobs_[0] holds the filter weights
// - blobs_[1] holds the biases (optional)
vector<int> weight_shape(2);
weight_shape[0] = conv_out_channels_;
weight_shape[1] = conv_in_channels_ / group_;
for (int i = 0; i < num_spatial_axes_; ++i)
{
  weight_shape.push_back(kernel_shape_data[i]);
}
```

```
bias_term_ = this->layer_param_.convolution_param().bias_term();
vector<int> bias_shape(bias_term_, num_output_);

if (this->blobs_.size() > 0)
{
  CHECK_EQ(1 + bias_term_, this->blobs_.size()) << "Incorrect number of weight blobs.
↪";
  if (weight_shape != this->blobs_[0]->shape())
  {
    Blob<Dtype> weight_shaped_blob(weight_shape);
    LOG(FATAL) << "Incorrect weight shape: expected shape "
               << weight_shaped_blob.shape_string() << "; instead, shape was "
               << this->blobs_[0]->shape_string();
  }

  if (bias_term_ && bias_shape != this->blobs_[1]->shape())
  {
    Blob<Dtype> bias_shaped_blob(bias_shape);
    LOG(FATAL) << "Incorrect bias shape: expected shape "
               << bias_shaped_blob.shape_string() << "; instead, shape was "
               << this->blobs_[1]->shape_string();
  }
  LOG(INFO) << "Skipping parameter initialization";
}
else
{
  if (bias_term_)
  {
    this->blobs_.resize(2);
  }
  else
  {
    this->blobs_.resize(1);
  }

  // Initialize and fill the weights:
  // output channels x input channels per-group x kernel height x kernel width
  this->blobs_[0].reset(new Blob<Dtype>(weight_shape));
  shared_ptr<Filler<Dtype> > weight_filler(GetFiller<Dtype>(
      this->layer_param_.convolution_param().weight_filler()));
  weight_filler->Fill(this->blobs_[0].get());

  // If necessary, initialize and fill the biases.
  if (bias_term_)
  {
    this->blobs_[1].reset(new Blob<Dtype>(bias_shape));
    shared_ptr<Filler<Dtype> > bias_filler(GetFiller<Dtype>(
        this->layer_param_.convolution_param().bias_filler()));
    bias_filler->Fill(this->blobs_[1].get());
  }
}
kernel_dim_ = this->blobs_[0]->count(1);
weight_offset_ = conv_out_channels_ * kernel_dim_ / group_;
// Propagate gradients to the parameters (as directed by backward pass).
```

16. (conv) param_propagate_down_

```
this->param_propagate_down_.resize(this->blobs_.size(), true);
```

(base_conv) Reshape

1. (bottom[0]) num_axes

Calculations:

- **inputs:**
  - **channel_axis_**
  - **num_spatial_axes_**
- **output:**
  - num_axes
- **calulations:**
  - first_spatial_axis = **channel_axis_** + 1
  - num_axes = first_spatial_axis + **num_spatial_axes_**

For example:

```
i.e. bottom[0].shape: (1, 3, 600, 800)

- inputs:
  - channel_axis_ = 1
  - num_spatial_axes_ = 2
- output:
  - num_axes = 4
- calculations:
  - first_spatial_axis = channel_axis_ + 1 = 1 + 1 = 2
  - num_axes = first_spatial_axis + num_spatial_axes_ = 2 + 2 = 4
```

2. (bottom[0]) num_

Calculations:

- input: **channel_axis_**
- output: **num_**
- calculation: **num_** = bottom[0]->count(0, **channel_axis_**)

For example:

```
i.e. bottom[0].shape: (1, 3, 600, 800)

- input: channel_axis_ = 1
- output: num_ = bottom[0]->count(0, 1) = 1
```

3. (bottom[0]) channels_

Calculations:

- **inputs:**
  - **channels_**
  - **channel_axis_**
- CHECK: bottom[0]->shape(**channel_axis_**) == **channels_**

For example:

```
i.e. bottom[0].shape: (1, 3, 600, 800)

- inputs
  - channels: 3
  - channel_axis_: 1
- CHECK:
  - bottom[0]->shape(channel_axis_) = bottom[0]->shape(1) = 3 == 3
```

4. (bottom[0]) shape

Purpose: Check all inputs with the same shape.

Calculations:

- **inputs:**
  - bottom.size()
  - bottom[bottom_id]->shape()
- CHECK: bottom[0]->shape() == bottom[bottom_id]->shape()

For example:

```
i.e. bottom: bottom[0]

- bottom.size(): 1
- bottom[0].shape == bottom[0].shape

i.e. bottom: bottom[0], bottom[1]

- bottom.size(): 2
- bottom[0].shape == bottom[0].shape
- bottom[0].shape == bottom[1].shape
```

5. (bottom) bottom_shape_

```
bottom_shape_ = &bottom[0]->shape()
```

6. (top) top_shape, output_shape_

Calculations:

- **inputs:**
  - input_shape: (height, width)
  - kernel: (height, width)
  - pad: (height, width)
  - stride: (height, width)
  - dilation: (height, width)
- **output:**
  - **top_shape_**: (output_h, output_w)
- **calculations:**
  - kernel_extent_h = dilation_h * (kernel_h - 1) + 1
  - output_h = (input_h + 2 * pad_h - kernel_extent_h) / stride_h + 1
  - kernel_extent_w = dilation_w * (kernel_w + 1) + 1
  - output_w = (input_w + 2 * pad_w - kernel_extent_w) / stride_w + 1

For example:

```
- inputs:
 - input_dim: (224, 224)
  - kernel: (3, 3)
  - pad: (1, 1)
  - stride: (1, 1)
  - dilation: (1, 1)

- output: output_shape_: (224, 224)

- calculations:
  - kernel_extent = 1 * (3 - 1) + 1 = 3
  - output_shape_ = (224 + 2 * 1 - 3) / 1 + 1 = 224
```

7. (conv) conv_out_spatial_dim_

Calculations:

- inputs:
- output:
- calculation:

For example:

```
if (reverse_dimensions()): conv_out_spatial_dim_ = bottom[0]->count(first_spatial_
↪axis)
else                     : conv_out_spatial_dim_ = top[0]->count(first_spatial_axis)

bottom[0].shape: (1, 3, 224, 244)
top[0].shape: (1, 3, 244, 244)

- conv_out_spatial_dim_ = height * width = 224 * 224 = 50176
```

8. (conv) col_offset

Calculations:

- inputs:
- output:
- calculations:

For example:

```
col_offset_ = kernel_dim_ * conv_out_spatial_dim_;

- kernel_dim_: channels * kernel_h * kernel_w = 3 * 3 * 3 = 27
- conv_out_spatial_dim_ = height * width = 224 * 224 = 50176
- col_offset_ = 27 * 50176 = 27 * 50176 = 1354752
```

9. (conv) output_offset_

Calculations:

- **inputs:**
  - conv_out_channels
  - **conv_out_spatial_dim_**
  - **group_**
- **output:**
  - **output_offset_**
- **calculations:**
  - **output_offset_** = **conv_out_channels_** * **conv_out_spatial_dim_** / **group_**

For example:

```
output_offset_ = conv_out_channels_ * conv_out_spatial_dim_ / group_

- conv_out_channels_: 64
- conv_out_spatial_dim_: 50176
- group_: 1
- output_offset_ = 64 * 50176 / 1 = 3211264
```

10. (conv) conv_input_shape_

Calculations:

- inputs:
- **output:**
  - **conv_input_shape_**: (channels, height, width)
  - conv_input_shape_data: (**num_spatial_axes_**, data[channel * height * width])
- calculations:

For example:

```
- inputs:
  - bottom[0]->shape(channel_axis_, num_spatial_axes_ + 1)
  - top[0]->shape(channel_axis_, num_spatial_axes_ + 1)
- output:
  - conv_input_shape_: (3, 244, 244)
- calculations:
  - conv_input_shape_:
    - bottom_dim_blob_shape: (1, num_spatial_axes_ + 1)
    - conv_input_shape_: (1, bottom_dim_blob_shape)
  - conv_input_shape_data:
    - if reverse_dimensions(): conv_input_shape_data[i] = top[0]->shape(channel_axis_␣
↪+ i)
    - else                   : conv_input_shape_data[i] = bottom[0]->shape(channel_
↪axis_ + i)
```

11. (conv) col_buffer_, col_buffer_shape_

Calculations:

- inputs:
- output:
- calculations:

For example:

```
col_buffer_shape_: (channels, height, width) = (kernel_dim_ * group_, in/out_h, in/
↪out_w)

- col_buffer_shape_: channels = kernel_dim_ * group_
- col_buffer_shape_: height = in/out_h = input/output_shape_h
- col_buffer_shape_: width = in/out_w = input/output_shape_w

for num_spatial_axes_:
    if (reverse_dimensions): input_shape(i+1)
    else                   : output_shape[i]


col_buffer_shape_: (27, 244, 244)
```

12. (conv) bottom_dim_, top_dim_

Calculations:

- inputs:
- outputs:
- calculations: - **bottom_dim_**: channels * height * width - **top_dim_**: channels * height * width

For example:

```
bottom_dim_ = bottom[0]->count(channel_axis_);
top_dim_ = top[0]->count(channel_axis_);
```

```
bottom_dim_: (3, 224, 224) = 3 * 224 * 224 = 150528
top_dim_: (64, 224, 224) = 64 * 224 * 224 = 3211264
```

13. (conv) num_kernels_im2col_, num_kernels_col2im_

```
num_kernels_im2col_ = conv_in_channels_ * conv_out_spatial_dim_;
num_kernels_col2im_ = reverse_dimensions() ? top_dim_ : bottom_dim_;

- conv_in_channels_: 3
- conv_out_spatial_dim_: bottom/top[0].height * bottom/top[0].width = 224 * 224 =␣
↪50176
- num_kernels_im2col_ = 3 * 50176 = 150528
- num_kernels_col2im_ = reverse_dimensions() ? top_dim_ : bottom_dim_ = c * h * w = 3␣
↪* 224 * 224 = 150528
```

14. (conv) bias_multiplier_

Calculations:

- inputs:
- outputs:
- calculations:

For example:

```
out_spatial_dim_ = (top[0]) height * width
bias_multiplier_: (shape) (1, top.height * top.width) = 224 * 224 = 50176

bias_multiplier_: [1, top.height * top.width]
  [1, 1, 1, ..., 1]
```

(base_conv) forward_cpu_gemm

(base_conv) forward_cpu_bias

(base_conv) backward_cpu_gemm

(base_conv) backward_cpu_bias

(base_conv) weight_cpu_gemm

(base_conv) conv_im2col_cpu

Data

- **inputs:**
  - (data_im) data:

  - (data_col) col_buff:
  - (channels) **conv_in_channels_**
  - (height) **conv_input_shape_**.cpu_data()[1]
  - (width) **conv_input_shape_**.cpu_data()[2]
  - (kernel_h) **kernel_shape_**.cpu_data()[0]
  - (kernel_w) **kernel_shape_**.cpu_data()[1]
  - (pad_h) **pad_**.cpu_data()[0]
  - (pad_w) **pad_**.cpu_data()[1]
  - (stride_h) **stride_**.cpu_data()[0]
  - (stride_w) **stride_**.cpu_data()[1]
  - (dilation_h) **dilation_**.cpu_data()[0]
  - (diliation_w) **dilation_**.cpu_data()[1]
- **output:**
  - (data_col) col_buff:
- **conditional params:**
  - **force_nd_im2col_**:
  - **num_spatial_axes_**:

(input) data:

- shape: (C, H, W)

(input) col_buff:

- shape: (C, H, W)

```
shape: (C, H, W)
- shape_C: kernel_dim_ * group_ = kernel_channels * kernel_height * kernel_width *␣
↪group_
- shape_H: input/output_shape_height
- shape_W: input_output_shape_width

- input_shape: (height, width)


- output_shape: (height, width) <- compute_output_shape()
- output_h:
  - kernel_extent_h = dilation_h * (kernel_h - 1) + 1
  - output_h = (input_h + 2 * pad_h - kernel_extent_h) / stride_h + 1
- output_w:
  - kernel_extent_w = dilation_w * (kernel_w + 1) + 1
  - output_w = (input_w + 2 * pad_w - kernel_extent_w) / stride_w + 1

i.e. shape (C, H, W)

inputs:
- image: (224, 224)
- pad: (1, 1)
- kernel: (3, 3, 3)
- dilation: (1, 1)
```

```
- stride: (1, 1)
- group_: 1

output: shape (27, 224, 224)
- shape_C = 3 * 3 * 3 * 1 = 27
- shape_H = 224
  - kernel_extent_h = 1 * (3 - 1) + 1 = 3
  - output_h = (224 + 2 * 1 - 3) / 1 + 1 = 224
- shape_W = 224
```

(output) data_col:

- shape: kernel_channels * kernel_h * kernel_w * num_kernels_conv_out

(base_conv) conv_col2im_cpu

(utils) IM2COL/COL2IM

(utils) im2col_cpu

(utils) im2col_nd_cpu

(utils) col2im_cpu

(utils) col2im_nd_cpu

(utils) GEMM/GEMV

(utils) caffe_cpu_gemm

(utils) caffe_cpu_gemv

2.3.2 Pooling

Overview

Data

(proto) PoolingParameter

Conditional Params

Constraint	Type	Variable	No.	Default	Description
optional	PoolMethod	pool	1	MAX	
optional	bool	global_pooling	12	False	
optional	Engine	engine	11		

Pooling Params

Constraint	Type	Variable	No.	Default	Description
optional	uint32	pad	4	0	
optional	uint32	pad_h	9	0	
optional	uint32	pad_w	10	0	
optional	uint32	kernel_size	2		
optional	uint32	kernel_h	5		
optional	uint32	kernel_w	6		
optional	uint32	stride	3	1	
optional	uint32	stride_h	7		
optional	uint32	stride_w	8		

Functions

(layer) Pooling

(pooling) LayerSetUp

1. get pooling_params
2. get **kernel_h_** and **kernel_w_**:

```
if global_pooling:
    kernel_h_ = bottom[0].height
    kernel_w_ = bottom[0].width
elif kernel_size:
    kernel_h_ = kernel_size.h
    kernel_w_ = kernel_size.w
elif kernel_h and kernel_w:
    kernel_h_ = kernel_h
    kernel_w_ = kernel_w
```

3. get **pad_h_** and **pad_w_**

```
if pad:
    pad_h_ = pad.h
    pad_w_ = pad_w
elif pad_h, pad_w:
    pad_h_ = pad_h
    pad_w_ = pad_w
```

4. get **stride_h_** and stride_w

```
if stride:
    stride_h_ = stride.h
    stride_w_ = stride.w
elif stride_h, stride_w:
    stride_h_ = stride_h
    stride_w_ = stride_w
```

5. check global_pooling

```
if global_pooling:
    pad == 0 && stride == 1
```

6. check pad

```
if pad_h_ != 0 and pad_w_ != 0:
    PoolMethod == MAX or AVE
```

(pooling) Reshape

1. get image channels, height, and width

```
channels_ = bottom[0]->channels();
height_ = bottom[0]->height();
width_ = bottom[0]->width();
```

2. get **kernel_h_** and **kernel_w_**

```
if global_pooling:
    kernel_h_ = bottom[0]->height
    kernel_w_ = bottom[0]->width
```

3. get **pooled_height_** and **pooled_width_**

```
pooled_height_ = static_cast<int>(ceil(static_cast<float>(height_ + 2 * pad_h_ -␣
↪kernel_h_) / stride_h_)) + 1;
pooled_width_ = static_cast<int>(ceil(static_cast<float>(width_ + 2 * pad_w_ - kernel_
↪w_) / stride_w_)) + 1;

if (pad_h_ || pad_w_)
{
   // If we have padding, ensure that the last pooling starts strictly
   // inside the image (instead of at the padding); otherwise clip the last.
   if ((pooled_height_ - 1) * stride_h_ >= height_ + pad_h_)
   {
     --pooled_height_;
   }

   if ((pooled_width_ - 1) * stride_w_ >= width_ + pad_w_)
   {
     --pooled_width_;
   }
   CHECK_LT((pooled_height_ - 1) * stride_h_, height_ + pad_h_);
   CHECK_LT((pooled_width_ - 1) * stride_w_, width_ + pad_w_);
}
```

4. get top.shape

```
top[0]->Reshape(bottom[0]->num(),
                channels_,
                pooled_height_,
                pooled_width_);

 if (top.size() > 1)
 {
   top[1]->ReshapeLike(*top[0]);
 }
```

5. if max pooling, get **max_idx_**

```
// If max pooling, we will initialize the vector index part.
if (this->layer_param_.pooling_param().pool() == PoolingParameter_PoolMethod_MAX &&
↪top.size() == 1)
{
    max_idx_.Reshape(bottom[0]->num(),
                     channels_,
                     pooled_height_,
                     pooled_width_);
}
```

6. if stochastic pooling, get **rand_idx_**

```
// If stochastic pooling, we will initialize the random index part.
if (this->layer_param_.pooling_param().pool() == PoolingParameter_PoolMethod_
↪STOCHASTIC)
{
   rand_idx_.Reshape(bottom[0]->num(),
                     channels_,
                     pooled_height_,
                     pooled_width_);
}
```

(pooling) Forward_cpu

1. get bottom and top data

```
const Dtype* bottom_data = bottom[0]->cpu_data();
Dtype* top_data = top[0]->mutable_cpu_data();
```

2. get top_count

```
const int top_count = top[0]->count();
```

3. process mask

```
// We'll output the mask to top[1] if it's of size >1.
const bool use_top_mask = top.size() > 1;
int* mask = NULL;  // suppress warnings about uninitalized variables
Dtype* top_mask = NULL;
```

4. process max pooling

```
// initialization
if use_top_mask:
    top_mask = top[1]->mutable_cpu_data();
    caffe_set(top_count, Dtype(-1), top_mask);
else:
    mask = max_idx_.mutable_cpu_data();
    caffe_set(top_count, -1, mask);

caffe_set(top_count, Dtype(-FLT_MAX), top_data);
```

5. process average pooling

2.3.3 ReLU

Overview

Algorithms

References

- Rectifier Nonlinearities Improve Neural Network Acoustic Models

Data

Parameters

Conditional Params

Constraint	Type	Variable	No.	Default	Description
optional	Engine	engine	2	DEFAULT	

ReLU Params

Constraint	Type	Variable	No.	Default	Description
optional	float	negative_slope	0		

Functions

(layer) ReLU

(layer) Forward_cpu

(layer) Backward_cpu

(layer) Forward_gpu

(layer) Backward_gpu

2.3.4 Inner Product

Overview

Algorithms

Data

Constraint	Type	Variable	No.	Default	Remark
optional	uint32	num_output	1		
optional	bool	bias_term	2	TRUE	
optional	FillerParameter	weight_filler	3		
optional	FillerParameter	bias_filler	4		
optional	int32	axis	5	1	

Functions

(layer) Inner Product

(layer) LayerSetUp

1. get bias_term

```
bias_term_ = bias_term
```

2. get num_output

```
N_ = num_output
```

3. get axis

```
axis = bottom[0]->CanonicalAxisIndex(this->layer_param_.inner_product_param().axis())
```

4. process **K_**

```
K_ = bottom[0]->count(axis)
```

5. process weights and bias

```
if blobs_.size > 0: not initialized
else:
    if bias_term_: blobs_.size == 2
    else:          blobs_.size == 1

    // initialize weights
    get shape: (N_, K_) = (num_output, bottom(axis))
    fill weights: by weight_filler

    // initialize bias
    if bias_term_:
        get shape: (1, N_) = (1, num_output)
        fill bias: by bias_filler
```

6. process **param_propagate_down_**

```
this->param_propagate_down_.resize(this->blobs_.size(), true)
```

(layer) Reshape

1. check dimensions:

```
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.";
```

2. get **M_**

```
M_ = bottom[0]->count(0, axis);
```

3. process top.shape

```
vector<int> top_shape = bottom[0]->shape();
top_shape.resize(axis + 1);
top_shape[axis] = N_;
top[0]->Reshape(top_shape);
```

4. process bias_multiplier

```
vector<int> bias_shape(1, M_);
bias_multiplier_.Reshape(bias_shape);
caffe_set(M_, Dtype(1), bias_multiplier_.mutable_cpu_data());
```

(layer) Forward_cpu

1. get bottom and top
2. get weights
3. (weights) caffe_cpu_gemm()

```
shapes:
- bottom: (M, K)
- top: (M, N)
```

```
- weights: (N, K)

top = bottom * weights.T
top[M, N] = bottom[M, K] * weight.T [K, N]

- M: bottom[0]->count(0, axis)
- K: bottom[0]->count(axis)
- N: num_output

For example:
- bottom: (N, C, H, W)
  -> axis: 1
  -> M_: N
  -> K_: C * H * W)
  -> N_: num_output

top[N, num_output] = bottom[N, C * H * W] * weights.T [C * H * W, num_output]
```

4. (bias) caffe_cpu_gemm()

```
top[M, N] = bias_multiplier_[M, 1] * bias[1, N] + top[M, N]
top[N, num_output] = bias_multiplier_[N, 1] * bias[1, num_output] + top[N, num_output]
```

(layer) Backward_cpu

(layer) Forward_gpu

(layer) Backward_gpu

2.3.5 Softmax

Overview

Algorithms

Data

Constraint	Type	Variable	No.	Default	Remark
optional	Engine	engine	1	Default	
optional	int32	axis	2	1	

Functions

(layer) Softmax

(layer) Reshape

1. **softmax_axis_** = axis
2. top.shape <- bottom.shape

3. get sum_multiplier.shape: (1, axis->num_axis) with value 1

```
sum_multiplier_: shape with (1, C)
```

4. get **outer_num_** and **inner_num_**

```
outer_num_ = bottom[0]->count(0, softmax_axis_);
inner_num_ = bottom[0]->count(softmax_axis_ + 1);
```

5. get **scale_**.shape

```
scale_.dims = bottom.dims
scale_dims[softmax_axis_] = 1
scale_.reshape(scale_dims)
```

(layer) Forward_cpu

1. get bottom and top data
2. get **scale_**
3. channels

```
channels = bottom[0]->shape(softmax_axis_)
```

4. dim

```
dim = bottom[0]->count() / outer_num_
```

5. top_data <- bottom_data

```
caffe_copy(bottom[0]->count(), bottom_data, top_data)
```

6. calculate softmax

```
for outer_num_: 0 -> N*C

    // initialize the scale_data
    caffe_copy(inner_num_, bottom_data + i * dim, scale_data)

    for channels:
        for inner_num_:
            scale_data[k] = std::max(scale_data[k], bottom_data[i * dim + j * inner_
↪num_ + k])

    // subtraction
    top_data[C, H*W] = -1 * sum_multiplier[C, 1] * scale_data[1, H*W] + top_data[C,␣
↪H*W]

    // expoentiation
    caffe_exp<Dtype>(dim, top_data, top_data)

    // sum after exp
    scale_data[C, H*W] = top_data[C, 1] * sum_multiplier_[1, H*W]

    // division
    for channels:
```

```
        caffe_div(inner_num_, top_data, scale_data, top_data);
        top_data += inner_num_;
```

(layer) Backward_cpu

(layer) Forward_gpu

(layer) Backward_gpu

2.4 Utils

2.4.1 Maths

Unary/Binary Functions

Unary Functions

(maths) caffe_sqr

(maths) caffe_exp

(maths) caffe_log

(maths) caffe_abs

(maths) caffe_powx

Binary Functions

(maths) caffe_add

(maths) caffe_sub

(maths) caffe_mul

(maths) caffe_div

BLAS

1. scalar-vector operations

- sscal, dscal
- scopy, dcopy
- isamax, idamax
- saxpy, daxpy

- sdot, ddot
- dasum
- dnrm2
- drot

2. matrix-vector operations

- sgemv, dgemv
- strmv, dtrmv
- strsv, dtrsv
- dgbmv
- sger, dger
- dsymv
- dtbmv
- dsyr

3. matrix-matrix operations

- sgemm, dgemm
- ssyrk, dsyrk
- strsm, dtrsm
- strmm, dtrmm
- ssymm, dsymm
- ssyr2k, dsyr2k

Precisions:

- S: real single precision
- D: real double precision
- C: complex single precision
- Z: complex double precision

Scalar-vector operations

(maths) caffe_axpy

saxpy: Y[n] <- alpha * X[n] + Y[n]

(maths) caffe_scal

scal: X[n] <- alpha * X[n]

- X:
- alpha:

(maths) caffe_asum

asum: result <- sum(xi)

Vector-vector operations

(maths) caffe_dot

dot: result[n] <- X[n] * Y[n]

(maths) caffe_axpby

axpby: Y[n] <- alpha * X[n] + beta * Y[n]

Matrix-vector operations

(maths) caffe_gemv

gemv: Y[m] <- alpha * A[m, n] * X[n] + Y[m]

Matrix-matrix operations

(maths) caffe_gemm

gemm: C <- alpha * A[m, k] * B[k, n] + beta * C[m, n]

References

- Basic Linear Algebra Subprograms Library Programmer's Guide and API Reference

Random

(maths) caffe_rng_rand

(maths) caffe_rng_uniform

(maths) caffe_rng_gaussian

(maths) caffe_rng_bernoulli