CAFFE DOCUMENTATION

DEEP LEARNING COURSE

Data Science Program, GWU, USA
School of Electrical and Computer Engineering, OSU, USA

✉: martin.t.hagan@okstate.edu
✉: ajafari@gwu.edu
✉: amir.h.jafari@okstate.edu

August 27, 2017

Contents

List of Figures

List of Tables

1 CAFFE MODEL ARCHITECTURE

A caffe model consists of layers and number of parameters. All parameters are defined in the document file which called caffe.proto (Caffe layers and their parameters are defined in the protocol buffer definitions for the project in caffe.proto. [1]) To use caffe,w e need to learn the configuration file (prototxt) preparation. There are many types of layers, such as Data, Convolution, Pooling, etc. The data flow between the layers called Blobs. To run caffe, we need to create a model, there are commonly used models available such as Lenet, Alex, etc.

1.1 Blob, Layer and NET

Deep neural network (DNN) is composed of many layers interconnected together. Caffe software is a tool to make a deep neural network and train these kind of network architectures. Caffe has a certain strategy, to build a model layer by layer. It is defined as all information and data blobs, so as to facilitate the conduct of operations and communications. Blob is caffe framework of a standard array of a unified memory interface. We will describe storage information and communication between the layers in detailed in section (1.1.1) , (1.1.2) and (1.1.3).

1.1.1 Blob

Blobs encapsulates the data at runtime while the computation is done in CPU or GPU. Mathematically, blob is an N-dimensional array. Basically, blob is the main and basic component of data storage in caffe software (Matlab uses matrix as the main component). Matrix is two-dimensional array, however, blob is an N-dimensional array. The N dimension in blob can be varied (e.g 2, 3, and so on). For image data, blob can be expressed as $n_i \times c_i \times h_i \times w_i$ a 4D array. Where n_i represents the number of pictures, c_i represents the channels (color image has 3 channels for Red, Green and Blue, gray has one channel), h_i and w_i respectively represent the height and width of the image. Of course, in addition to image data, blob can also be used for non-image data. For example, conventional multi layer perceptron (MLP), is relatively simple to connect the entire network with the 2D blob, using the innerProduct layer to calculate network output.

Model parameters are represented in blob with the certain format. For example, in a convolution by having a color image we have to enter the 3 as channel value, and if there are 96 convolution kernels with image size of (11×11), therefore the blob contains $(96 \times 3 \times 11 \times 11)$ value.

1.1.2 Layer

Layer is composed of elements which defines the operation of the layer and it has a certain format called JSON format. JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. In each layer, we have a section and subsection and they can be distinguished by the curly brackets. Each section and subsection has some fields and for each field we can have a selection of options. Lets look at the following example for the convolution layer:

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  # learning rate and decay multipliers for the filters
  param { lr_mult: 1 decay_mult: 1 }
  # learning rate and decay multipliers for the biases
  param { lr_mult: 2 decay_mult: 0 }
  convolution_param {
    num_output: 96
    kernel_size: 11
    stride: 4n
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
```

Now, layer, bias_filler are the section and subsection respectively and name, type, bottom and top are fields. Each fields in section or in a subsection has selection. For example, name in this layer has the selection of conv1 (name is arbitrary to choose for each layer), type has the selection of Convolution which shows that this layer is a convolution layer. We will go into the detailed of each fields and its selections in later sections.

Fig. 1 shows the simple layer: As you can see the in the figure the layer layout structured as bottom to top. In this example, this convolution layer connected to the data with bottom blob and the top blob can be connected to the next layer. The layers are connected together by top and bottoms. The yellow polygons are the implicit connection to other layers and the blue rectangle represents the convolution layer.

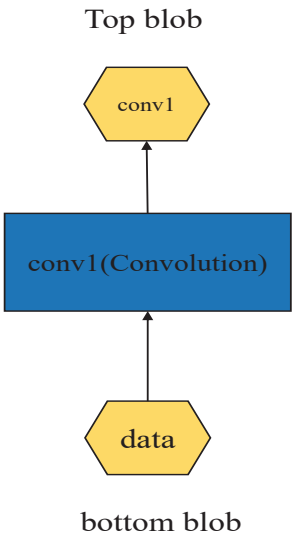

Figure 1: Layer

1.1.3 NET

Like building blocks, a caffe network contains more than one layer, they are connected together from bottom to top. The caffe model has one file which contains all the jason formats for each layer that is called .prototxt file. In this file all sections will define the caffe layers and types of operation and the fields and selections are the options to configure the layer to perform an specific task.

Lets look at simple two-layer model defined in neural networks. Fig. 2 shows the network topology.

The first layer is a data layer which is named as mnist. For the filed of type the data is used for the section. There is no bottom for the data layer however there is two tops which connects the data layer to the innerproduct layer and the loss layer (yellow polygons are the implicit connections). The top second InnerProduct layer is connected to the bottom of the loss layer. The third layer has two bottoms, one for ip and the other one is for data labels. There is top for an output loss which is not drawn. Prototxt corresponding configuration file can be written:

```
name: "LogReg"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  data_param {
```

```
    source: "input_leveldb"
    batch_size: 64
  }
}
layer {
  name: "ip"
  type: "InnerProduct"
  bottom: "data"
  top: "ip"
  inner_product_param {
    num_output: 2
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip"
  bottom: "label"
  top: "loss"
}
```

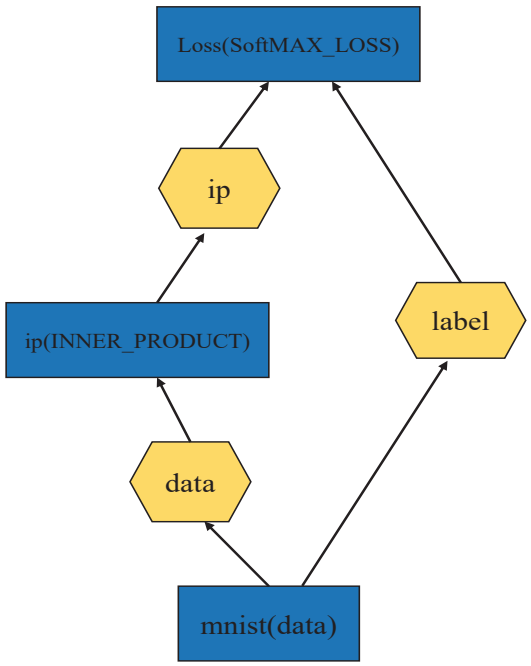

Figure 2: NET

1.2 Data Layer and Parameters

Now in this section, we will first introduce the data layer. Data layer is the bottom of each caffe model and it is the entry model. This layer not only to provide input data, but it provides a data conversion from blobs into another format to save the output. Preprocessing and normalizing data is done in this layer (subtracting the mean, zoom, crop, and mirroring). In this layer we configure and set the desired parameters. There are several format of databases that caffe accepts. These format are efficient ways of reading datas such as LevelDB and LMDB. If the efficiency is not important, the data can also be fetched from the disk file and image formats hdf5 file. Lets look at the common data layer format.

Lets look at the typical example of a data layer prototxt file.

```
name: "CaffeNet"
layer {
  name: "Input"
  type: "Data"
  top: "data"
  top: "Target"
  include {
    phase: TRAIN
  }
  transform_param {
    mirror: true
    crop_size: 227
    mean_file: "data/ilsvrc12/imagenet_mean.binaryproto"
  }
  data_param {
    source: "examples/imagenet/ilsvrc12_train_lmdb"
    batch_size: 256
    backend: LMDB
  }
}
```

In any layer section, usually we have the following fields:

- **name**
- **type**
- **top or bottom**

For the selection of each fields we need to pick the right option to configure the layer. For example for the data layer we can have the following selections for their associated fields.

Fields	Selction
name	An arbitrary name can be used for each layer.
type	LevelDB or LMDB, MemoryData, HDF5Data, ImageData, WindowData
top or bottom	layer names

Table 1: Data layer section fields and selections

type shows the layer functionality. Since this is a data type layer, we need to provide the the source and format of the data. There are different formats of data bases. In general practice the LevelDB or LMDB data is used. **top or bottom**: Each layer has bottom and top parts. Bottom identifies other layers that are inputs to this layer, and top identifies layers that this layer outputs to. Bottom is the input and top would be the output of the layer. Sometimes we have multiple top or more bottom to have more connections to other layers. In the data layer, at least one of the top sections is the data. The second top selection is our target.

As we said before, in each section we have subsections and each subsection has it on fields and selections.

Fields	Selction
include	Train and Test

Table 2: Data layer subsection fields and selections

include: In the time of general training and testing, layer model is not the same. There is a layer with a training phase and another layer with a testing phase, this is done with the include section. If you do not include parameters, it indicates that the layer model both in training and testing.

The **transform_param** subsection has the set fields and selections to configure. This field contains parameters that are used to preprocess or transform the data before it is output to the next layer

Fields	Selction
mirror	True and False
crop_size	Integer value to set the size
mean_file	It is the path to the binaryproto file

Table 3: Data layer subsection preprocessing fields and selections

The configuration of the input data is done in this subsection. The **data_param** subsection has the set fields and selections to configure.

Fields	Selction
source	The directory path that contains the name of the database is in the source section.
batch_size	Integer value to set the size
backend	Choose one of the types from Table 1

Table 4: Data layer subsection data parameter fields and selections

There are 5 selections for the type fields in the data layer section which are as follows

- **LevelDB and LMDB**
- **Data from the memory**
- **Data from HDF5**
- **From the image data**
- **Data from the Windows**

The following examples prototxt files are listed for all these data bases and formats.

```
layer {
  top: " Input "
  top: " Target "
  name: " memory_data "
  type: " MemoryData "
  memory_data_param {
    batch_size: 2
    height: 100
    width: 100
    channels: 1
  }
  transform_param {
    Scale: 0.0078125
    mean_file: " mean.proto "
    mirror: false
  }
}
```

```
layer {
  name: " Input "
  type: " HDF5Data "
```

```
  top: " Data "
  top: " Target "
  hdf5_data_param {
    source: " examples/ Data / train.txt "
    batch_size: 10
  }
}
```

```
layer {
  name: " Input "
  type: " the ImageData "
  top: " Data "
  top: " Target "
  transform_param {
    mirror: false
    crop_size: 227
    mean_file: " Data/ilsvrc12/imagenet_mean.binaryproto "
  }
  image_data_param {
    Source: " examples /_temp/ file_list.txt "
    batch_size: 50
    new_height: 256
    new_width: 256
  }
}
```

```
layer {
  name: " Input "
  type: " WindowData "
  top: " Data "
  top: " Target "
  include {
    phase: TRAIN
  }
  transform_param {
    mirror: true
    crop_size: 227
    mean_file: " Data/ilsvrc12/imagenet_mean.binaryproto "
  }
  window_data_param {
    source:"finetune_pascal_detection/trainval.txt"
    batch_size: 128
    fg_threshold: 0.5
    bg_threshold: 0.5
```

```
    fg_fraction: 0.25
    context_pad: 16
    crop_mode: " Warp "
  }
}
```

Summary:

As you can see from all the following example they share a lot of fields together and they are different in the selections. Also, you can have one universal data layer format which you are able to change the selection to configure it to different data bases.

1.3 Vision Layers, Pooling and LRN

Vision layers usually take images as input and produce other images as output. A typical image in the real-world may have one color channel ($c_i = 1$), as in a grayscale image, or three color channels ($c_i = 3$) as in an RGB (red, green, blue) image. But in this context, the distinguishing characteristic of an image is its spatial structure: usually an image has some non-trivial height $h_i > 1$ and width $w_i > 1$. This 2D geometry naturally lends itself to certain decisions about how to process the input. In particular, most of the vision layers work by applying a particular operation to some region of the input to produce a corresponding region of the output. In contrast, other layers (with few exceptions) ignore the spatial structure of the input, effectively treating it as one big vector with dimension $c_i \times h_i \times w_i$.

1.3.1 Convolution Layer

The Convolution layer convolves the input image with a set of learnable kernels, each producing one feature map in the output image [1].In this layer the input image convolves with a bank of kernels (biases also can be added).

Layer Type: Convolution

lr_mult: It is a layer learning rate. The over all learning rate is base_lr and it is set in solver prototxt.

You must set the parameters:

num_output: the number of feature maps

kernel_size: Specifies height and width of each kernel. If the convolution kernel is unequal length and width then we need to change the kernel_h kernel_w.

Other parameters:

stride: Specifies the intervals at which to apply the kernels to the input, the default is 1. You can also have unequal length stride, we need to set stride_h and stride_w.

pad: Specifies the number of pixels to (implicitly) add to each side of the input, the default is 0. If the kernel size is 5 * 5 and the pad is set to 2, the four edges are expanded two pixels in width and height (four pixels). For unequal padding we can set the pad_h and pad_w respectively.

weight_filler: Weight initialization. The default is ”constant” and set to 0. Most of times used as ”xavier” algorithm to initialize (can be set to ”gaussian”).

bias_filler: Bias initialization. Usually set to ”constant” with values of zero.

bias_term: Specifies whether to learn and apply a set of additive biases to the filter outputs.

group: If $g > 1$, we restrict the connectivity of each filter to a subset of the input. Specifically, the input and output channels are separated into g groups, and the iith output group channels will be only connected to the i-th input group channels.

Input: $n_i \times c_i \times w_i \times h_i$

Output: $n_o \times c_o \times w_o \times h_o$

Where c_o is the parameter num_output, the number of generated feature map,

$w_o = (w_i + 2 \times PAD - kernel\_size)/STRIDE + 1$

$h_o = (h_i + 2 \times PAD - kernel\_size)/STRIDE + 1$

If the stride is set to 1, there is an overlap twice before and after convolution section. If pad = (kernel_size-1) / 2, then after the operation, the width and height unchanged.

```
layer {
  name: " CONV1 "
  type: " Convolution "
  bottom: " Data "
  top: " CONV1 "
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  convolution_param {
    num_output: 20
    kernel_size: 5
    stride: 1
    weight_filler {
      of the type: " Xavier "
    }
    bias_filler {
      type: " Constant "
    }
  }
}
```

1.3.2 Pooling Layer

It is called pooling layer in order to reduce the amount of computation and data dimensions and settings.

Layer Type: Pooling

You must set the parameters: **kernel_size**: Specifies height and width of each kernel. Also it can be set separately with kernel_h and kernel_w. *Other parameters:*

pool: Pooling method, the default is MAX. Other available methods are MAX, AVE, or STOCHASTIC **pad**: Specifies the number of pixels to (implicitly) add to each side of the input. The default value is 0. **stride**: Pooling step size, the default value is set to 1. Generally, we set it to 2 for kernel size 2, to prevent overlapping. For unequal stride we can also set stride_h and stride_w.

```
layer {
  name: " pool1 "
  type: " Pooling "
  bottom: " CONV1 "
  top: " pool1 "
  pooling_param {
    pool: MAX
    kernel_size: 2
    STRIDE: 2
  }
}
```

The method of operation and the pooling layer is substantially the same as the convolution layer.

Input: $n \times c_i \times w_i \times h_i$

Output: $n \times c_o \times w_o \times h_o$

And the difference between the convolution layer is one of the c remains unchanged

$w_o = (w_i + 2 * PADkernel_size)/STRIDE + 1$

$h_o = (h_i + 2 * PADkernel_size)/STRIDE + 1$

If the stride is set to 2 and kernel size is 2, we do not have an overlap. After pooling the feature map size of (100×100) becomes (50).

1.3.3 Local Response Normalization (LRN)

In this layer the partial area of an input can be normalized for ”lateral inhibition” effect. AlexNet or GoogLenet use this feature.

Layer Type: LRN

You must set the parameters:

Parameters: All optional

local_size: The number of channels to sum over (for cross channel LRN) or the side length of the square region to sum over (for within channel LRN) The default is 5. If the cross-channel LRN, then the sum of the number of channels; if it is LRN in the channel, then the length of the square area summation.

alpha: The default is 1. The scaling parameter (see below).

beta: The default is 5. The exponent (see below).

norm_region: The default is across_channels. There are two options, across_channels represents between adjacent channels summed normalized. within_channel expressed in a specific area inside a channel summing normalized. Local_size corresponding to the previous parameter.

The local response normalization layer performs a kind of lateral inhibition by normalizing over local input regions. In across_channels mode, the local regions extend across nearby channels, but have no spatial extent (i.e., they have shape local_size x 1 x 1). In within_channel mode, the local regions extend spatially, but are in separate channels (i.e., they have shape 1 x local_size x local_size). Each input value is divided by $(1+(\frac{\alpha}{n}\sum_i x_i^2))^\beta$, where nn is the size of each local region, and the sum is taken over the region centered at that value (zero padding is added where necessary) [1].

```
layers {
  name: " NORM1 "
  type: LRN
  bottom: " pool1 "
  Top: " NORM1 "
  lrn_param {
    local_size: 5
    Alpha: from 0.0001
    Beta: 0.75
  }
}
```

1.4 Activation Layers and Parameters

ReLU is currently the most used activation function, mainly because of its faster convergence, and to maintain the same effect. In general, activation / Neuron layers are element-wise operators, taking one bottom blob and producing one top blob of the same size. In the layers below, we will ignore the input and out sizes as they are identical:

Input : $n_i \times c_i \times h_i \times w_i$

Output: $n_o \times c_o \times h_o \times w_o$

1.4.1 ReLU / Rectified-Linear and Leaky-ReLU

Layer Type: ReLU

Other parameters:

negative_slope: Specifies whether to leak the negative part by multiplying it with the slope value rather than setting it to 0.

```
layer {
  name: " relu1 "
  type: " Relu "
  bottom: " pool1 "
  top: " pool1 "
}
```

Given an input value x, The ReLU layer computes the output as x if $x > 0$ and negative_slope $*x$ if $x <= 0$. When the negative slope parameter is not set, it is equivalent to the standard ReLU function of taking $max(x, 0)$.

1.4.2 Sigmoid

Layer Type: Sigmoid

The Sigmoid layer computes the output as sigmoid(x) for each input element x.

$$S(x) = \frac{1}{1+e^{-x}} \tag{1}$$

```
layer {
  name: "encode1neuron"
  bottom: "encode1"
  top: "encode1neuron"
  type: "Sigmoid"
}
```

1.4.3 TanH / Hyperbolic Tangent

Layer Type: TanH

The TanH layer computes the output as tanh(x) for each input element x. The hyperbolic tangent function is used for data conversion.

$$tanh(x) = \frac{sinhx}{coshx} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2}$$

```
layer {
  name: "layer"
  bottom: "in"
  top: "out"
  type: "TanH"
}
```

1.4.4 Absolute Value

Layer Type: AbsVal

The AbsVal layer computes the output as abs(x) for each input element x. Each seeking the absolute value of the input data.

```
layer {
  name: "layer"
  bottom: "in"
  top: "out"
  type: "AbsVal"
}
```

1.4.5 Power

Layer Type: Power

Perform power operation on each input data.

$$f(x) = (shift + scale * x)^{power} \tag{3}$$

Other parameters:

power: The default is 1

scale: The default is 1

shift: The default is 0

```
layer {
  name: "layer"
  bottom: "in"
  top: "out"
  type: "Power"
  power_param {
    power: 1
    scale: 1
    shift: 0
  }
}
```

1.4.6 BNLL

Layer Type: BNLL

Binomial normal log likelihood abbreviation. The BNLL (binomial normal log likelihood) layer computes the output as $log(1 + exp(x))$ for each input element x.

```
layer {
  name: " Layer "
  bottom: " in "
  Top: " OUT "
  type: "BNLL"
}
```

1.5 Loss Layers

Loss drives learning by comparing an output to a target and assigning cost to minimize. The loss itself is computed by the forward pass and the gradient w.r.t. to the loss is computed by the backward pass.

1.5.1 Softmax-loss

Layer Type: SoftmaxWithLoss

Softmax-loss and softmax calculation is substantially the same. Softmax is a classifier, calculate the probability category (Likelihood), is an extension of Logistic Regression. Binary Logistic Regression can only be used while softmax can be used for multiple classification.

Softmax Vs. softmax-loss:

Softmax formula:

$$p(z)_j = \frac{e^{zj}}{\sum_{k=1}^{K} e^{zk}} \qquad for j = 1, 2, ..., K \tag{4}$$

The softmax-loss formula:

$$L = -\sum_j y_j log p_j \tag{5}$$

The softmax loss layer computes the multinomial logistic loss of the softmax of its inputs. It's conceptually identical to a softmax layer followed by a multinomial logistic loss layer, but provides a more numerically stable gradient. If the ultimate goal of every user is to obtain the probability of each category of likelihood then the Softmax layer is used.

Softmax-loss layer: output value is loss

```
layer {
  name: " Loss "
  type: " SoftmaxWithLoss "
  bottom: " IP1 "
  bottom: " label "
```

```
  top: " Loss "
}
```

Softmax layer: output likelihood values

```
layers {
  bottom: " cls3_fc "
  Top: " prob "
  name: " prob "
  type: "Softmax"
}
```

1.5.2 Accuracy

Layer Type: Accuracy

Accuracy scores the output as the accuracy of output with respect to target it is not actually a loss and has no backward step.

```
layer {
  name: " Accuracy "
  type: " Accuracy "
  bottom: " IP2 "
  bottom: " label "
  top: " Accuracy "
  include {
    phase: TEST
  }
}
```

1.5.3 Sum-of-Squares / Euclidean

Layer Type: EuclideanLoss

The Euclidean loss layer computes the sum of squares of differences of its two inputs.

$$\frac{1}{2N}\sum_{i=1}^{N}||x_i^1 - x_i^2||_2^2 \tag{6}$$

1.5.4 Hinge / Margin

Layer Type: HingeLoss

Parameters: optional

norm : The norm used. Currently L1, L2

Inputs:

$n * c * h * w$ Predictions

$n * 1 * 1 * 1$ Labels

Output:

$1 * 1 * 1 * 1$ Computed Loss

```
# L1 Norm
layer {
  name: "loss"
  type: "HingeLoss"
  bottom: "pred"
  bottom: "label"
}

# L2 Norm
layer {
  name: "loss"
  type: "HingeLoss"
  bottom: "pred"
  bottom: "label"
  top: "loss"
  hinge_loss_param {
    norm: L2
  }
}
```

The hinge loss layer computes a one-vs-all hinge or squared hinge loss.

1.5.5 Sigmoid Cross-Entropy

Layer Type: SigmoidCrossEntropyLoss

1.5.6 Infogain

Layer Type: InfogainLoss

1.6 Common Layers

1.6.1 Inner Product or Fully Connected layer

The InnerProduct layer (also usually referred to as the fully connected layer) treats the input as a simple vector and produces an output in the form of a single vector (with the blobs height and width set to 1).

Input: $n_i \times c_i \times h_i \times w_i$

Output: $n_o \times c_o \times 1 \times 1$

Layer Type: Inner Product

Full connectivity layer is actually a convolution layer, but its primary data convolution kernel size and the same size. So its basic parameters and parameter convolution layer of the same.

```
layer {
  name: "fc8"
  type: "InnerProduct"
  # learning rate and decay multipliers for the weights
  param { lr_mult: 1 decay_mult: 1 }
  # learning rate and decay multipliers for the biases
  param { lr_mult: 2 decay_mult: 0 }
  inner_product_param {
    num_output: 1000
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
  bottom: "fc7"
  top: "fc8"
}
```

1.6.2 Splitting

Layer Type: Split

The Split layer is a utility layer that splits an input blob to multiple output blobs. This is used when a blob is fed into multiple output layers.

1.6.3 Flattening

Layer Type: Flatten

The Flatten layer is a utility layer that flattens an input of shape $n * c * h * w$ to a simple vector output of shape $n * (c * h * w)$

1.6.4 Reshape

Layer Type: Reshape

Input: A single blob with arbitrary dimensions.

Output: The same blob, with modified dimensions, as specified by reshape_param

```
layer {
   name: "reshape"
   type: "Reshape"
   bottom: "input"
   top: "output"
   reshape_param {
     shape {
       dim: 0  # copy the dimension from below
       dim: 2
       dim: 3
       dim: -1 # infer it from the other dimensions
     }
   }
 }
```

The Reshape layer can be used to change the dimensions of its input, without changing its data. Just like the Flatten layer, only the dimensions are changed; no data is copied in the process.

Output dimensions are specified by the ReshapeParam proto. Positive numbers are

used directly, setting the corresponding dimension of the output blob. In addition, two special values are accepted for any of the target dimension values:

- 0 means copy the respective dimension of the bottom layer. That is, if the bottom has 2 as its 1st dimension, the top will have 2 as its 1st dimension as well, given dim: 0 as the 1st target dimension.
- -1 stands for infer this from the other dimensions. This behavior is similar to that of -1 in numpys or [] for MATLABs reshape: this dimension is calculated to keep the overall element count the same as in the bottom layer. At most one -1 can be used in a reshape operation.

Specifying reshape_param shape dim: 0 dim: -1 makes the layer behave in exactly the same way as the Flatten layer.

As another example, there is an optional parameter set shape, for each dimension of the data value of the specified blob (blob is a four-dimensional data: $n*c*w*h$).

dim: 0 indicates the same dimension, namely the input and output are the same dimensions.

dim: 2 or dim: 3 original dimension becomes 2 or 3

dim: -1 indicates calculated automatically by the system dimensions. The total amount of data the same, the system will automatically calculate the value based on the current dimension dimension other three-dimensional blob data.

Suppose the original data: $64*3*28*28$, 64 color photographs showing the third channel $28*28$

After reshape transformation:

```
reshape_param {
   shape {
     dim: 0
     dim: 0
     Dim: 14
     Dim: -1
   }
 }
```

Output data: $64*14*56*3$

1.6.5 Dropout

Layer Type: Dropout

Dropout is a trick to prevent overfitting. The network can make some random hidden layer weights does not work. Only you need to set a dropout_ratio it.

```
layer {
  name: " Drop7 "
  type: " Dropout "
  bottom: " FC7-CONV "
  top: " FC7-CONV "
  dropout_param {
    dropout_ratio: 0.5
  }
}
```

1.6.6 Concatenation

Layer Type: Concat

The Concat layer is a utility layer that concatenates its multiple input blobs to one single output blob.

axis : 0 for concatenation along num and 1 for channels.

Input:

$n_i * c_i * h * w$ for each input blob i from 1 to K.

Output:

if axis = 0: $(n_1 + n_2 + ... + n_K) * c_1 * h * w$, and all input c_i should be the same.

if axis = 1: $n_1 * (c_1 + c_2 + ... + c_K) * h * w$, and all input n_i should be the same.

```
layer {
  name: "concat"
  bottom: "in1"
  bottom: "in2"
  top: "out"
  type: "Concat"
  concat_param {
    axis: 1
```

```
  }
}
```

1.6.7 Slicing

Layer Type: Slice

The Slice layer is a utility layer that slices an input layer to multiple output layers along a given dimension (currently num or channel only) with given slice indices.

```
layer {
  name: "slicer_label"
  type: "Slice"
  bottom: "label"
  ## Example of label with a shape N x 3 x 1 x 1
  top: "label1"
  top: "label2"
  top: "label3"
  slice_param {
    axis: 1
    slice_point: 1
    slice_point: 2
  }
}
```

axis indicates the target axis; slice_point indicates indexes in the selected dimension (the number of indices must be equal to the number of top blobs minus one).

1.6.8 Elementwise Operations

Layer Type: Eltwise

1.6.9 Argmax

Layer Type: ArgMax

1.6.10 Mean-Variance Normalization

Layer Type: MVN

2 CAFFE SOLVER

2.1 Solver and its configuration file

Solver is the core part of the caffe software, which coordinates the operation of the entire model. All the parameters of the solver in caffe program will run from solver profile. Solver main role is to call to the alternate forward pass algorithm and backward pass algorithm to update the parameters, in order to minimize loss function, This is actually an iterative optimization algorithm.

The current version of caffe offers six optimization algorithm, we can set each optimization algorithm and its configuration in the solver file.

Optimization Algorithms Available in Caffe

Training Algorithm	Type
Gradient Descent Stochastic	”SGD”
AdaDelta	”AdaDelta”
Adaptive Gradient	”AdaGrad”
ADAM	”Adam”
The Accelerated Gradient apos Nesterov	”Nesterov”
RMSprop	”RMSProp”

In every iteration, solver steps perform the following actions:

1. Call forward algorithm to calculate the final output value, and a corresponding loss.
2. Call backward algorithm to calculate each gradient.
3. According to the choice of solver method using gradient parameter update.
4. Record and save each iteration of the learning rate, snapshots, and the corresponding state.

Let’s look at an example:

```
NET: " examples / MNIST / lenet_train_test.prototxt "
test_iter: 100
test_interval: 500
```

```
base_lr: 0.01
Momentum: 0.9
type: SGD
weight_decay: 0.0005
lr_policy: " INV "
Gamma: from 0.0001
Power: 0.75
the display: 100
max_iter: 20000
Snapshot: 5000
snapshot_prefix: " examples / MNIST / lenet "
solver_mode: the CPU
```

NET: Sets the network model. Net needs to be configured in a special configuration file, each model has a number of layers and activation functions with loss layers (prototxt file see the pervious sections).

test_iter: This should be combined with the understanding of the test layer batch_size. For example, total mnist test data has sample of 10,000 data, testing on all sample test data has a low efficiency, so we will be divided into several batches of data. The number of each batch is batch_size. Suppose we set batch_size 100, then we need to iterate 100 times in order to complete the implementation of all test data 10000. Therefore test_iter set to any value less than maximum iteration. After the implementation of data all at once, we called a epoch.

test_interval: The test interval is after 500 iteration we perform testing.

base_lr: As long as we use the gradient descent training algorithm for minimizing the error, there will be a learning rate, also called the step size too. base_lr is the learning rate in caffe. The following adjustments on the basis of the learning rate can be made.

lr_policy: See **lrpolicy**.

Gamma: See **lrpolicy**.

Power: See **lrpolicy**.

lrpolicy: can be set to the following sets, the corresponding learning rate is calculated as follows:

1. Fixed: maintaining base_lr unchanged.
2. Step: If it is set to step, you need to set a step size.
   - $base\_lr * gamma^{(floor(\frac{iter}{stepsize})}$

3. Exp:
    - $base\_lr * gamma^{iter}$
4. Inv: If set to inv, also need to set up a power.
    - $base\_lr * (1 + gamma * iter)^{(-power)}$
5. Multistep: If set to multistep, you also need to set a stepvalue.
6. Poly: learning rate polynomial error.
    - $base\_lr * (1 - \frac{iter}{max_iter})^{(power)}$
7. Sigmoid: Sigmoid attenuation.
    - $base\_lr * (\frac{1}{(1+exp(-gamma*(iter-stepsize))))}$

Momentum:

type: Optimization algorithm selection. This line can be saved, because the default value is SGD. There are six methods to choose from.

weight_decay: Weight decay term, is to prevent over fitting a parameter.

the display: Every training 100 times, once displayed on the screen.

max_iter: The maximum number of iterations.

Snapshot: It is the way to save trained model and solver state. snapshot used for training many times to save the settings, the default is 0, not saved. Snapshot_prefix is the save path.

You can also set snapshot_diff, whether to save the gradient.

You can also set snapshot_format. There are two options: HDF5 and BINARYPROTO, default BINARYPROTO

solver_mode: Set the operation mode to CPU or GPU

We are going to detailed description of each method in the next section.

References

[1] Caffe berkeley vision. [Online]. Available: http://caffe.berkeleyvision.org/tutorial/layers.html