Caffe Library

4

4.1 Introduction

Implementing ConvNets from scratch is a tedious task. Especially, implementing the backpropagation algorithm correctly requires to calculate the gradient of each layer correctly. Even after implementing the backward pass, it has to be validated by computing the gradient numerically and comparing it with the result of backpropagation. This is called *gradient check*. Moreover, efficient implementation of each layer on GPU is another hard work. For these reasons, it might be more practical to use a library for this purpose.

As we discussed in the previous chapter, there are many libraries and frameworks that can be used for training ConvNets. Among them, there is one library which is suitable for development as well as applied research. This library is called *Caffe*.[1] Figure 4.1 illustrates the structure of Caffe.

The Caffe library is developed in C++ and it utilizes CUDA library for performing computations on GPU.[2] There is a library which is developed by NVIDIA and it is called *cuDNN*. It has implemented common layers found in ConvNets as well as their gradients. Using cuDNN, it is possible to design and train ConvNets which are only executed on GPUs. Caffe makes use of cuDNN for implementing some of layers on GPU. It has also implemented some other layers directly using CUDA. Finally, besides providing interfaces for Python and MATLAB programming languages, it also provides a command tool that can be used for training and testing ConvNets.

One beauty of Caffe is that designing and training a network can be done by employing text files which are later parsed using Protocol Buffers library. But, you are not limited to design and train using only text files. It is possible to also design and

[1] http://caffe.berkeleyvision.org.

[2] There are some branches of Caffe that use OpenCL for communicating with GPU.

H. Habibi Aghdam and E. Jahani Heravi, *Guide to Convolutional Neural Networks*, DOI 10.1007/978-3-319-57550-6_4

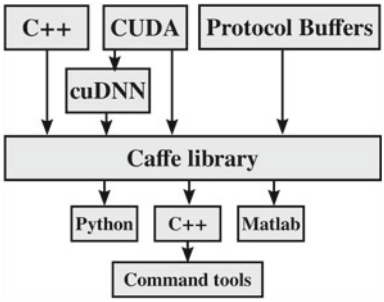

Fig. 4.1 The Caffe library uses different third-party libraries and it provides interfaces for C++, Python, and MATLAB programming languages

train ConvNets by writing a computer program in C++, Python or MATLAB. However, a detailed analysis of ConvNets has to be done by writing compute programs or special softwares.

In this chapter, we will first explain how to use text files and the command tools for designing and training ConvNets. Then, we will explain how to do it in Python. Finally, methods for analyzing ConvNets using Python will be also discussed.

4.2 Installing Caffe

Installation of Caffe requires installing CUDA and some third-party libraries on your system. The list of required libraries can be found in *caffe.berkeleyvision.org*. If you are using Ubuntu, Synaptic Package Manager can be utilized for installing these libraries. Next, CUDA drivers must be installed on the system. Try to download the latest CUDA driver compatible with Caffe from NIVIDIA website. Installing CUDA drivers can be as simple as just running the installation file. In worst case scenario, it may take some time to figure out what are the error messages and to finally install it successfully.

After that, cuDNN library must be downloaded and copied into the CUDA folder which is by default located in */usr/local/cuda*.. You must copy the *cudnn*.h* into the include folder and *libcudnn*.so** into *lib/lib64* folder. Finally, you must follow the instructions provided in the Caffe's website for installing this library.

4.3 Designing Using Text Files

A ConvNet and its training procedure can be defined using two text files. The first text file defines architecture of the neural network including ConvNets and the second file defines the optimization algorithm as well as its parameters. These text files are

usually stored with *.prototxt* extension. This extension shows that the text inside these files follows the syntax defined by the Protocol Buffers (protobuf) protocol.[3]

A protobuf is composed of *messages* where each message can be interpreted as a struct in a programming language such as C++. For example, the following protobuf contains two messages namely *Person* and *Group*.

```
message Person {
    required string name = 1;
    optional int32 age = 2;
    repeated string email = 3;
}

message Group {
    required string name = 1;
    repeated Person member = 3;
}
```

Listing 4.1 A protobuf with two messages.

The field rule *required* shows that specifying this field in the text file is mandatory. In contrast, the rule *optional* shows that specifying this field in the text file is optional. As it turns out, the rule *repeated* states that this filed can be repeated zero or more times in the text file. Finally, numbers after the equal signs are unique tag numbers which are assigned to each field in a message. The number has to be unique inside the message.

From programming perspective, these two messages depict two data structures namely *Person* and *Group*. The Person struct is defined using three fields including one required, one optional and one repeated (array) field. The Group struct also is defined using one required filed and one repeated filed, where each element in this field is an instance of Person.

You can write the above definition in a text editor and save it with *.proto* extension (e.g. sample.proto). Then, you can open the terminal in Ubuntu and execute the following command:

```
protoc -I=SRC_DIR --python_out=DST_DIR SRC_DIR/sample.proto
```

If the command is executed successfully, you should find a file named *sample_pb2.py* in directory *DST_DIR*. Instantiating Group can be done in a programming language. To this end, you should import *sample_pb2.py* to python environment and run the following code:

```
g = sample_pb2.Group()
g.name ='group 1'

m = g.member.add()
m.name = 'Ryan'
m.age=20
m.email.append('mail1@sample.com')
m.email.append('mail1@sample.com')

m = g.member.add()
m.name = 'Harold'
m.age=23
```

[3] Implementations of the methods in this chapter are available at *github.com/pcnn/*.

Using the above code, we create a group called “group 1” with two members. The age of the first member is 20, his name is “Ryan” and he has two email addresses. Moreover, the name of second member is “Harold”. He is 23 years old and he does not have any email.

The appealing property of protobuf is that you can instantiate the Group structure using a plain text file. The following plain text is exactly equivalent to the above Python code:

```
name: "group 1"
member {
    name: "member1"
    age: 20
    email: "mail1@sample.com"
    email: "mail1@sample.com"
}
member {
    name: "member2"
    age: 23
}
```

This method has some advantages over instantiating using programming. First, it is independent of programming language. Second, its readability is higher. Third, it can be easily edited. Fourth, it is more compact. However, there might be some cases that instantiating is much faster when we write a computer program rather than a plain text file.

There is a file called *caffe.proto* inside the source code of the Caffe library which defines several protobuf messages.[4] We will use this file for designing a neural network. In fact, *caffe.proto* is the reference file that you must always refer to it when you have a doubt in your text file. Also, it is constantly updated by developers of the library. Hence, it is a good idea to always keep studying the changes in the newer version so you will have a deeper knowledge about what can be implemented using the Caffe library. There is a message in *caffe.proto* called “NetParameter” and it is currently defined as follows[5]:

```
message NetParameter {
    optional string name = 1;
    optional bool force_backward = 5 [default = false];
    optional NetState state = 6;
    optional bool debug_info = 7 [default = false];
    repeated LayerParameter layer = 100;
}
```

We have excluded deprecated fields marked in the current version from the above message. The architecture of a neural network is defined using this message. It contains a few fields with basic data types (e.g., string, int32, bool). It has also one field of type NetState and an array (repeated) of LayerParameters. Arguably, one can learn Caffe just by throughly studying NetParameter. The reason is illustrated in Fig. 4.2.

[4] All the explanations for the Caffe library in this chapter are valid for the commit number 5a201dd.
[5] This definition may change in next versions.

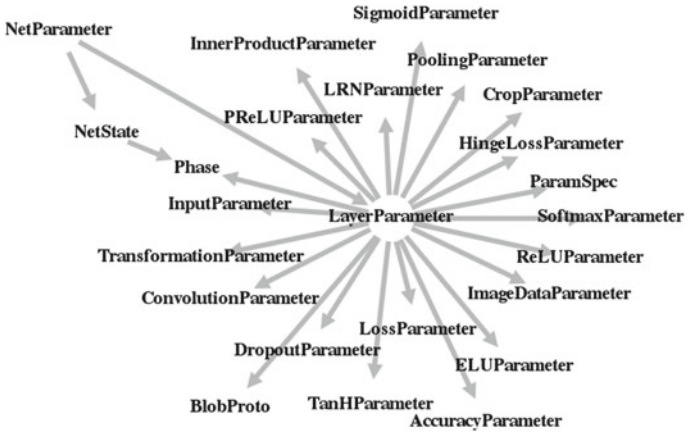

Fig. 4.2 The NetParameter is indirectly connected to many other messages in the Caffe library

It is clear from the figure that NetParameter is indirectly connected to different kinds of layers through LayerParameter. It turns outs that NetParameter is a container to hold layers. Also, there are several other kind of layers in the Caffe library that we have not included in the figure. The message LayerParamter has many fields. Among them, following are the fields that we may need for the purpose of this book:

```
message LayerParameter {
    optional string name = 1;
    optional string type = 2;
    repeated string bottom = 3;
    repeated string top = 4;

    optional ImageDataParameter image_data_param = 115;
    optional TransformationParameter transform_param = 100;

    optional AccuracyParameter accuracy_param = 102;
    optional ConvolutionParameter convolution_param = 106;
    optional CropParameter crop_param = 144;
    optional DropoutParameter dropout_param = 108;
    optional ELUParameter elu_param = 140;
    optional InnerProductParameter inner_product_param = 117;
    optional LRNParameter lrn_param = 118;
    optional PoolingParameter pooling_param = 121;
    optional PReLUParameter prelu_param = 131;
    optional ReLUParameter relu_param = 123;
    optional ReshapeParameter reshape_param = 133;
    optional SigmoidParameter sigmoid_param = 124;
    optional SoftmaxParameter softmax_param = 125;
    optional TanHParameter tanh_param = 127;

    optional HingeLossParameter hinge_loss_param = 114;

    repeated ParamSpec param = 6;
    optional LossParameter loss_param = 101;

    optional Phase phase = 10;
}
```

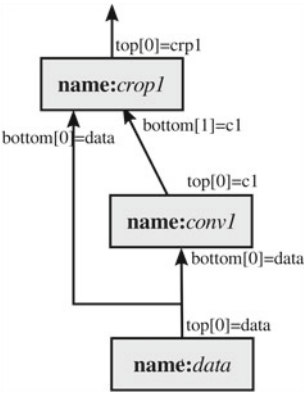

Fig. 4.3 A computational graph (neural network) with three layers

Each layer has a name. Although entering a name for a layer is optional but it is highly recommended to give each layer a *unique* name. This increases readability of your model. It has also another function. Assume you want to have two convolution layers with exactly the same parameters. In other words, these two convolution layers share the same set of weights. This can be easily specified in Caffe by giving these two layers an identical name.

The string filed "type" specifies the type of the layer. For example, by assigning "Convolution" to this field, we tell Caffe that the current layer is a convolution layer. Note that the type of layer is case-sensitive. This means that, assigning "convolution" (small letter c instead of capital letter C) to type will raise an error telling that "convolution" is not a valid type for a layer.

There are two arrays of strings in LayerParameter called "top" and "bottom". If we assume that a layer (an instance of LayerParameter) is represented by a node in computational graphs, the bottom variable shows the tag of incoming nodes to the current node and the top variable shows the tag of outgoing edges. Figure 4.3 illustrates a computational graph with three nodes.

This computational graph is composed of three layers namely $data$, $conv_1$ and $crop_1$. For now, assume that the node $data$ reads images along with their labels from a disk and stores them in memory. Apparently, the node $data$ does not get its information from another node. For this reason, it does not have any bottom (the length of bottom is zero). The node $data$ passes this information to other nodes in the graph. In Caffe, the information produced by a node is recognized by unique tags. The variable top stores the name of these tags. A tag and name of a node could be identical. As we can see in node $data$, it produces only one output. Hence, the length of array top will be equal to 1. The first (and only) element in this array shows the tag of the first output of the node. In the case of $data$, the tag has been also called $data$. Now, any other node can have access to information produced by the node data using its tag.

The second node is a convolution node named $conv_1$. This node receives information from node $data$. The convolution node in this example has only one incoming

node. Therefore, length of bottom array for $conv_1$ will be 1. The first (and only) element in this array refers to the tag, where the information from this tag will come to $conv_1$. In this example, the information comes from $data$. After convolving $bottom[0]$ with filters in $conv_1$ (the value of filter are stored in node itself), it produces only one output. So, length of array top for $conv_1$ will be equal to 1. The tag of output for $conv_1$ has been called $c1$. In this case, the name of node and top of node are not identical.

Finally, the node $crop_1$ receives two inputs. One from $conv_1$ and one from $data$. For this reason, the bottom array in this node has two elements. The first element is connected to $data$ and the second element is connected to $c1$. Then, $crop_1$, crops the first element of bottom (bottom[0]) to make its size identical to the second element of bottom (bottom[1]). This node also generates a single output. The tag of this output is $crp1$.

In general, passing information between computational nodes is done using array of bottoms (incoming) and array of tops (outgoing). Each node stores information about its bottoms and tops as well as its parameters and hyperparameters. There are many other fields in LayerParameter all ending with phrase "Parameter". Based the *type* of a node, we may need to instantiate some of these fields.

4.3.1 Providing Data

The first thing to put in a neural network is at least one layer that provides data for the network. There are a few ways in Caffe to do this. The simplest approach is to provide data using a layer with *type="ImageData"*. This type of layer requires instantiating the field *image_data_param* from LayerParameter. ImageDataParameter is also a message with the following definition:

```
message ImageDataParameter {
    optional string source = 1;

    optional uint32 batch_size = 4 [default = 1];
    optional bool shuffle = 8 [default = false];

    optional uint32 new_height = 9 [default = 0];
    optional uint32 new_width = 10 [default = 0];

    optional bool is_color = 11 [default = true];
    optional string root_folder = 12 [default = ""];
}
```

Again, deprecated fields have been removed from this list. This message is composed of fields with basic data types. An ImageData layer needs a text file with the following structure:

```
ABSOLUTE_PATH_OF_IMAGE1  LABEL1
ABSOLUTE_PATH_OF_IMAGE2  LABEL2
...
ABSOLUTE_PATH_OF_IMAGEN  LABELN
```

Listing 4.2 Structure of train.txt

An ImageData layer assumes that images are stored on the disk using a regular image format such as jpg, bmp, ppm, png, etc. Images could be stored on different locations and different disks on your system. In the above structure, there is one line for each image in the training set. Each line is composed of two parts separated by a *space* character (ASCII code 32). The left part shows the absolute path of the image and the right part shows the class label of that image.

The current implementation of Caffe identifies class label from image using the space character in the line. Consequently, if the path of the image contains space characters, Caffe will not able to decode this line and it may raise an exception. For this reason, avoid space characters in the name of folders and files when you are creating a text file for an ImageData layer.

Moreover, class labels have to be integer numbers and they have to always start from zero. That said, if there are 20 classes in your dataset, the class labels have to be integer numbers between 0 and 19 (19 included). Otherwise, Caffe may raise an exception during training. For example, the following sample shows a small part of a text file that is prepared for an ImageData layer.

```
/home/pc/Desktop/GTSRB/Training_CNN/00019/00000_00006.ppm 19
/home/pc/Desktop/GTSRB/Training_CNN/00029/00003_00021.ppm 29
/home/pc/Desktop/GTSRB/Training_CNN/00010/00054_00008.ppm 10
/home/pc/Desktop/GTSRB/Training_CNN/00023/00010_00027.ppm 23
/home/pc/Desktop/GTSRB/Training_CNN/00033/00022_00008.ppm 33
/home/pc/Desktop/GTSRB/Training_CNN/00021/00000_00005.ppm 21
/home/pc/Desktop/GTSRB/Training_CNN/00005/00020_00022.ppm 5
/home/pc/Desktop/GTSRB/Training_CNN/00025/00026_00018.ppm 25
...
```

Suppose that our dataset contains 3,000,000 images and they are all located in a common folder. In the above sample, all files are stored at */home/pc/Desktop/GTSRB/Training_CNN*. However, this common address is repeated in the text file 3 million times since we have provided absolute path of images. Taking into account the fact that Caffe loads all the paths and their labels into memory once, this means $3{,}000{,}000 \times 35$ characters are repeated in the memory which is equal to about 100 MB memory. If the common path is longer or the number of samples is higher, more memory will be needed to store the information.

To use the memory more efficiently, ImageDataParameter has provided a filed called *root_folder*. This field points to the path of the common folder in the text file. In the above example, this will be equal to */home/pc/Desktop/GTSRB/Training_CNN*. In that case, we can remove the common path from the text file as follows:

```
/00019/00000_00006.ppm 19
/00029/00003_00021.ppm 29
/00010/00054_00008.ppm 10
/00023/00010_00027.ppm 23
/00033/00022_00008.ppm 33
/00021/00000_00005.ppm 21
/00005/00020_00022.ppm 5
/00025/00026_00018.ppm 25
...
```

Caffe will always add the *root_folder* to the beginning of path in each line. This way, redundant information are not stored in the memory.

The variable *batch_size* denotes the size of mini-batch to be forwarded and back-propagated in the network. Common values for this parameter vary between 20 and 256. This also depends on the available memory on your GPU. The Boolean variable *shuffle* shows whether or not Caffe must shuffle the list of files in each epoch or not. Shuffling could be useful for having diverse mini-batches at each epoch. Considering the fact that one epoch refers to processing whole dataset, the list of files is shuffled when the last mini-batch of dataset is processed. In general, setting shuffle to true could be a good practice. Especially, setting this value to true is essential when the text file containing the training samples is ordered based on the class label. In this case, shuffling is an essential step in order to have diverse mini-batches. Finally, as it turns out from their name, if *new_height* and *new_width* have a value greater than zero, the loaded image will be resized to the new size based on the value of these parameters. Finally, the variable *is_color* tells Caffe to load images in color format or grayscale format.

Now, we can define a network containing only an ImageData layer using the protobuf grammar. This is illustrated below.

```
name:"net1"
layer{
    name:"data"
    type:"ImageData"
    top:"data"
    top:"label"
    image_data_param{
        source:"/home/pc/Desktop/train.txt"
        batch_size:30
        root_folder:"/home/pc/Desktop/"
        is_color:true
        shuffle:true
        new_width:32
        new_height:32
    }
}
```

In Caffe, a tensor is a $mini - batch \times Channel \times Height \times Width$ array. Note that an ImageData layer produces two tops. In other words, the length of *top* array for this layer is 2. The first element of the top array stores loaded images. Therefore, the first top of the above layer will be a $30 \times 3 \times 32 \times 32$ tensor. The second element of the top array stores labels of each image in the first top and it will be an array with $mini - batch$ integer elements. Here, it will be a 30-element array of integers.

4.3.2 Convolution Layers

Now, we want to add a convolution layer to the network and connect it to the Image-Data layer. To this end, we must create a layer with *type="Convolution"* and then configure the layer by instantiating *convolution_param*. The type of this variable is ConvolutionParameter which is defined as follows:

```
message ConvolutionParameter {
    optional uint32 num_output = 1;
    optional bool bias_term = 2 [default = true];

    repeated uint32 pad = 3;
    repeated uint32 kernel_size = 4;
    repeated uint32 stride = 6;

    optional FillerParameter weight_filler = 7;
    optional FillerParameter bias_filler = 8;
}
```

The variable *num_output* determines the number of convolution filters. Recall from the previous chapter that the activation of neuron basically is given by $\}(\mathbf{wx} + bias)$. The variable *bias_term* states that whether or not the bias term must be considered in the neuron computation. The variable *pad* denotes the zero-padding size and it is 0 by default. Zero padding is used to handle the borders during convolution. Zero-Padding a $H \times W$ image with *pad=2* can be thought as creating a zero matrix of size $(H + 2pad) \times (W + 2pad)$ and copying the image into this matrix such that is placed exactly in the middle of the zero matrix. Then, if the size of convolution filters is $(2pad + 1) \times (2pad + 1)$, the result of convolution with zero-padded image will be $H \times W$ images which is exactly equal to the size of input image. Padding is usually done for keeping the size of input and output of convolution operations constant. But, it is commonly set to zero.

As it turns out, the variable *kernel_size* determines the spatial size (width and height) of convolution filters. It should be noted that a convolution layer must have the same number of bottoms and tops. It convolves each bottom separately with the filter and passes it to the corresponding top. The third dimension of filters is automatically computed by Caffe based on the number of channels coming from the bottom node. Finally, the variable *stride* illustrates the stride of convolution operation and it is set to 1 by default. Now, we can update the protobuf text and add a convolution layer to the network.

```
name:"net1"
layer{
    name:"data"
    type:"ImageData"
    top:"data"
    top:"label"
    image_data_param{
        source:"/home/hamed/Desktop/train.txt"
        batch_size:30
        root_folder:"/home/hamed/Desktop/"
        is_color:true
        shuffle:true
        new_width:32
        new_height:32
    }
}
layer{
    name:"conv1"
    type:"Convolution"
    bottom:"data"
    top:"conv1"
    convolution_param{
        num_output: 6
        kernel_size:5
    }
}
```

The convolution layer has six filters of size 5×5 and it is connected to a data layer that produces mini-batches of images. Figure 4.4 illustrates the diagram of the neural network created by the above protobuf text.

4.3.3 Initializing Parameters

Any layer with trainable parameters including convolution layers has to be initialized before training. Concretely, convolution filters (weights) and biases of convolution layer have to be initialized. As we explained in the previous chapter, this can be done by setting each weight/bias to a random number. However, generating random number can be done using different distributions and different methods. The *weight_filler* and *bias_filler* parameters in LayerParameter specify the type of initialization method. They are both instances of FillerParameter which are defined as follows:

```
message FillerParameter {
    optional string type = 1 [default = 'constant'];
    optional float value = 2 [default = 0];
    optional float min = 3 [default = 0];
    optional float max = 4 [default = 1];
    optional float mean = 5 [default = 0];
    optional float std = 6 [default = 1];

    enum VarianceNorm {
        FAN_IN = 0;
        FAN_OUT = 1;
        AVERAGE = 2;
    }
    optional VarianceNorm variance_norm = 8 [default = FAN_IN];
}
```

The string variable *type* defines the method that will be used for generating number. Different values can be assigned to this variable. Among them, "constant", "gaussian", "uniform", "xavier" and "mrsa" are commonly used in classification networks. Concretely, a constant filler sets the parameters to a constant value specified by the floating point variable *value*.

Also, a "gaussian" filler assigns random numbers generated by a Gaussian distribution specified by *mean* and *std* variables. Likewise, "uniform" filler assigns random

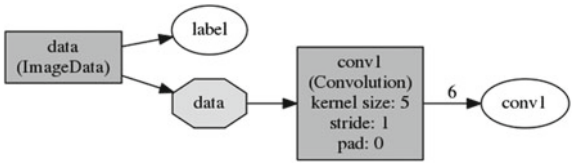

Fig. 4.4 Architecture of the network designed by the protobuf text. *Dark rectangles* show nodes. *Octagon* illustrates the name of the top element. The number of *outgoing arrows* in a node is equal to the length of top array of the node. Similarly, the number of *incoming arrows* to a node shows the length of bottom array of the node. The *ellipses* show the tops that are not connected to another node

number generated by the uniform distribution within a range determined by *min* and *max* variables.

The "xavier" filler generates uniformly distributed random numbers within $[-\sqrt{\frac{3}{n}}, \sqrt{\frac{3}{n}}]$, where depending on the value of *variance_norm* variable n could be the number of inputs (FAN_IN), the number of output (FAN_OUT) or average of them. The "msra" filler is like "xavier" filler. The difference is that it generates Gaussian distributed random number with standard deviation equal to $\sqrt{\frac{2}{n}}$.

As we mentioned in the previous chapter, filters are usually initialized using "xavier" or "mrsa" methods and biases are initialized using constant value zero. Now, we can also define weight and bias initializer for the convolution layer. The updated protobuf text will be:

```
name:"net1"
layer{
    name:"data"
    type:"ImageData"
    top:"data"
    top:"label"
    image_data_param{
        source:"/home/hamed/Desktop/train.txt"
        batch_size:30
        root_folder:"/home/hamed/Desktop/"
        is_color:true
        shuffle:true
        new_width:32
        new_height:32
    }
}
layer{
    name:"conv1"
    type:"Convolution"
    bottom:"data"
    top:"conv1"
    convolution_param{
        num_output: 6
        kernel_size:5
        weight_filler{
            type:"xavier"
        }
        bias_filler{
            type:"constant"
            value:0
        }
    }
}
```

4.3.4 Activation Layer

Each output of convolution layer is given by $\mathbf{wx} + b$. Next, these values must be passed through a nonlinear activation function. In the Caffe library, ReLU, Leaky ReLU, PReLU, ELU, sigmoid, and hyperbolic tangent activations are implemented. Setting *type="ReLU"* will create a Leaky ReLU activation. If we set the leak value to zero, this is equivalent to the ReLU activation. The other activations are created by setting *type="PReLU"*, *type="ELU"*, *type="Sigmoid"* and *type="TanH"*. Then,

depending on the type of activation function, we can also adjust their hyperparameters. The messages for these activations are defined as follows:

```
message ELUParameter {
    optional float alpha = 1 [default = 1];
}
message ReLUParameter {
    optional float negative_slope = 1 [default = 0];
}
message PReLUParameter {
    optional FillerParameter filler = 1;
    optional bool channel_shared = 2 [default = false];
}
```

Clearly, the sigmoid and hyperbolic tangent activation do not have parameters to set. However, as it is mentioned in (2.93) and (2.96) the family of the ReLU activation in Caffe has hyperparameters that should be configured. In the case of Leaky ReLU and ELU activations, we have to determine the value of α in (2.93) and (2.96). In Caffe, α for Leaky ReLU is illustrated by *negative_slope* variable. In the case of PReLU activation, we have to tell Caffe how to initialize the α parameter using the *filler* variable. Also, the Boolean variable *channel_shared* determines whether Caffe should share the same α for all activations (*channel_shared=true*) in the same layer or it must find separate α for each channel in the layer. We can add this activation to the protobuf as follows:

```
name:"net1"
layer{
    name:"data"
    type:"ImageData"
    top:"data"
    top:"label"
    image_data_param{
        source:"/home/hamed/Desktop/train.txt"
        batch_size:30
        root_folder:"/home/hamed/Desktop/"
        is_color:true
        shuffle:true
        new_width:32
        new_height:32
    }
}
layer{
    name:"conv1"
    type:"Convolution"
    bottom:"data"
    top:"conv1"
    convolution_param{
        num_output: 6
        kernel_size:5
        weight_filler{
            type:"xavier"
        }
        bias_filler{
            type:"constant"
            value:0
        }
    }
}
layer{
    type:"ReLU"
    bottom:"conv1"
    top:"relu_c1"
}
```

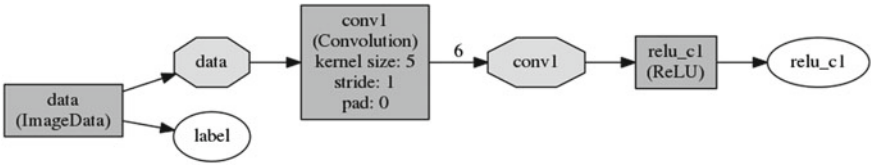

Fig. 4.5 Diagram of the network after adding a ReLU activation

After adding this layer to the network, the architecture will look like Fig. 4.5.

4.3.5 Pooling Layer

A pooling layer is created by setting *type="Pooling"*. Similar to a convolution layer, a pooling layer must have the same number of bottoms and tops. It applies pooling on each bottom separately and passes it to the corresponding top. Parameters of the pooling operation are also determined using an instance of PoolingParameter.

```
message PoolingParameter {
   enum PoolMethod {
   MAX = 0;
   AVE = 1;
   STOCHASTIC = 2;
   }
   optional PoolMethod pool = 1 [default = MAX];
   optional uint32 pad = 4 [default = 0];

   optional uint32 kernel_size = 2;
   optional uint32 stride = 3 [default = 1];
   optional bool global_pooling = 12 [default = false];
}
```

Similar to Convolutionparameter, the variables *pad*, *kernel_size* and *stride* determine the amount of zero padding, size of pooling window, and stride of pooling, respectively. The variable *pool* determines the type of pooling. Currently, Caffe supports max pooling, average pooling, and stochastic pooling. However, we often choose max pooling and it is the default option in Caffe. The variable *global_pooling* pools over the entire spatial region of bottom array. It is equivalent to setting *kernel_size* to the spatial size of the bottom blob. We add a max-pooling layer to our network. The resulting protobuf will be:

```
name:"net1"
layer{
    name:"data"
    type:"ImageData"
    top:"data"
    top:"label"
    image_data_param{
        source:"/home/hamed/Desktop/train.txt"
        batch_size:30
        root_folder:"/home/hamed/Desktop/"
        is_color:true
        shuffle:true
        new_width:32
        new_height:32
```

```
    }
}
layer{
    name:"conv1"
    type:"Convolution"
    bottom:"data"
    top:"conv1"
    convolution_param{
        num_output: 6
        kernel_size:5
        weight_filler{
            type:"xavier"
        }
        bias_filler{
                type:"constant"
        value:0
        }
    }
}
layer{
    name:"relu_c1"
    type:"ReLU"
    bottom:"conv1"
    top:"relu_c1"
    relu_param{
        negative_slope:0.01
    }
}
layer{
    name:"pool1"
    type:"Pooling"
    bottom:"relu_c1"
    top:"pool1"
    pooling_param{
        kernel_size:2
        stride:2
    }
}
```

The pooling will be done over 2×2 regions with stride 2. This will halve the spatial size of the input. Figure 4.6 shows the diagram of the network.

We added another convolution layer with 16 filters of size 5×5, a ReLU activation and a max-pooling with 2×2 region and stride 2 to the network. Figure 4.7 illustrates the diagram of the network.

4.3.6 Fully Connected Layer

A fully connected layer is defined by setting *type="InnerProduct"* in the definition of layer. The number of bottoms and tops must be equal in this type of layer. It computes

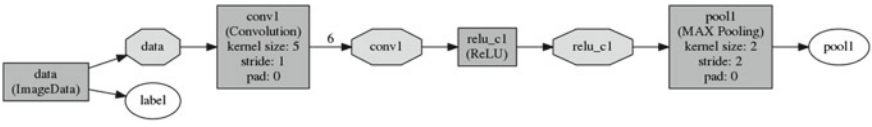

Fig. 4.6 Architecture of network after adding a pooling layer

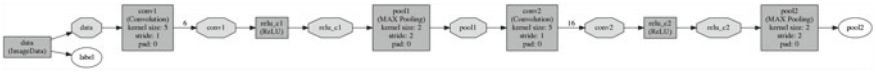

Fig. 4.7 Architecture of network after adding a pooling layer

the top for each bottom separately using the same set of parameters. Hyperparameters of a fully connected layer are specified using an instance of InnerProductParameter which is defined as follows.

```
message InnerProductParameter {
    optional uint32 num_output = 1;
    optional bool bias_term = 2 [default = true];
    optional FillerParameter weight_filler = 3;
    optional FillerParameter bias_filler = 4;
}
```

The variable *num_output* determines the number of neurons in the layer. The variable *bias_term* tells Caffe whether or not to consider the bias term in neuron computations. Also, *weight_filler* and *bias_filler* are used to specify how to initialize the parameters of the fully connected layer.

4.3.7 Dropout Layer

A dropout layer can be placed anywhere in a network. But, it is more common to put it immediately after an activation layer. However, it is mainly placed after activation of fully connected layers. The reason is that fully connected layers increase nonlinearity of a model and they apply final transformations on the extracted features by previous layers. Our model may over fit because of the final transformations. For this reason, we try to regularize the model using dropout layers in fully connected layers. A dropout layer is defined by setting *type="Dropout"*. Then, hyperparameter of a dropout layer is determined using an instance of DropoutParameter which is defined as follows:

```
message DropoutParameter {
    optional float dropout_ratio = 1 [default = 0.5];
}
```

As we can see, a dropout layer only has one hyperparameter which is the ratio of dropout. Since this ratio shows the probability of dropout, it has to be set to a floating point number between 0 and 1. The default value in Caffe is 0.5. We added two fully connected layers to our network and placed a dropout layer after each of these layers. The diagram of network after applying these changes is illustrated in Fig. 4.8.

4.3.8 Classification and Loss Layers

The last layer in a classification network is a fully connected layer, where the number of neurons in this layer is equal to number of classes in the dataset. Training a

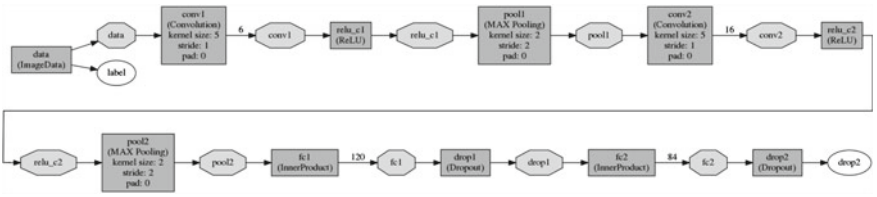

Fig. 4.8 Diagram of network after adding two fully connected layers and two dropout layers

neural network is done by minimizing a loss function. In this book, we explained hinge loss and logistic loss functions for multiclass classification problems. These two loss functions accept at least two bottoms. The first bottom is the output of the classification layer and the second bottom is actual labels produced by the ImageData layer. The loss layer computes the loss based on these two bottoms and returns a scaler in its top.

The hinge loss function is created by setting *type="HingeLoss"* and multiclass logistic loss is created by setting *type="SoftmaxWithLoss"*. Then, we mainly need to enter the bottoms and top of the loss layer. We added a classification layer and a multiclass logistic loss to the protobuf. The final protobuf will be:

```
layer{
    name:"data"
    type:"ImageData"
    top:"data"
    top:"label"
    image_data_param{
        source:"/home/hamed/Desktop/GTSRB/Training_CNN/train.txt"
        batch_size:30
        root_folder:"/home/hamed/Desktop/GTSRB/Training_CNN/"
        is_color:true
        shuffle:true
        new_width:32
        new_height:32
    }
}
layer{
    name:"conv1"
    type:"Convolution"
    bottom:"data"
    top:"conv1"
    convolution_param{
        num_output: 6
        kernel_size:5
        weight_filler{ type:"xavier" }
        bias_filler{ type:"constant" value:0 }
    }
}
layer{
    name:"relu_c1"
    type:"ReLU"
    bottom:"conv1"
    top:"relu_c1"
    relu_param{ negative_slope:0.01 }
}
layer{
    name:"pool1"
    type:"Pooling"
```

```
    bottom:"relu_c1"
    top:"pool1"
    pooling_param{ kernel_size:2 stride:2 }
}
layer{
    name:"conv2"
    type:"Convolution"
    bottom:"pool1"
    top:"conv2"
    convolution_param{
        num_output: 16
        kernel_size:5
        weight_filler{ type:"xavier" }
        bias_filler{ type:"constant" value:0 }
    }
}
layer{
    name:"relu_c2"
    type:"ReLU"
    bottom:"conv2"
    top:"relu_c2"
    relu_param{ negative_slope:0.01 }
}
layer{
    name:"pool2"
    type:"Pooling"
    bottom:"relu_c2"
    top:"pool2"
    pooling_param{ kernel_size:2 stride:2 }
}
layer{
    name:"fc1"
    type:"InnerProduct"
    bottom:"pool2"
    top:"fc1"
    inner_product_param{
        num_output:120
        weight_filler{ type:"xavier" }
        bias_filler{ type:"constant" value:0 }
    }
}
layer{
    name:"relu_fc1"
    type:"ReLU"
    bottom:"fc1"
    top:"relu_fc1"
    relu_param{ negative_slope:0.01 }
}
layer{
    name:"drop1"
    type:"Dropout"
    bottom:"relu_fc1"
    top:"drop1"
    dropout_param{ dropout_ratio:0.4 }
}
layer{
    name:"fc2"
    type:"InnerProduct"
    bottom:"drop1"
    top:"fc2"
    inner_product_param{
        num_output:84
        weight_filler{ type:"xavier" }
        bias_filler{ type:"constant" value:0 }
    }
}
layer{
```

```
    name:"relu_fc2"
    type:"ReLU"
    bottom:"fc2"
    top:"relu_fc2"
    relu_param{ negative_slope:0.01 }
}
layer{
    name:"drop2"
    type:"Dropout"
    bottom:"relu_fc2"
    top:"drop2"
    dropout_param{ dropout_ratio:0.4 }
}
layer{
    name:"fc3_classification"
    type:"InnerProduct"
    bottom:"drop2"
    top:"classifier"
    inner_product_param{
        num_output:43
        weight_filler{type:"xavier"}
        bias_filler{ type:"constant" value:0 }
    }
}
layer{
    name:"loss"
    type:"SoftmaxWithLoss"
    bottom:"classifier"
    bottom:"label"
    top:"loss"
}
```

Considering that there are 43 classes in the GTSRB dataset, the number of neurons in the classification layer must be also equal to 43. The diagram of final network is illustrated in Fig. 4.9.

The above protobuf text is stored in a text file on disk. In this example, we store the above text file in "*/home/pc/cnn.prototxt*". The above definition reads the training samples and feeds them to the network. However, in practice, the network must be evaluated using a *validation set* during training in order to assess how good the network is.

To achieve this goal, the network can be evaluated every K iterations of the training algorithm. As we will see shortly, this can be easily done by setting a parameter. Assume, K iterations have been finished and Caffe wants to evaluate the network. So far, we have only fetched data from a training set. Obviously, we have to tell Caffe where to look for validation samples. To this end, we add another ImageData layer right after the first ImageData layer and specify the location of the validation samples instead of the training samples. In other words, the first layer in the above network definition will be replaced by:

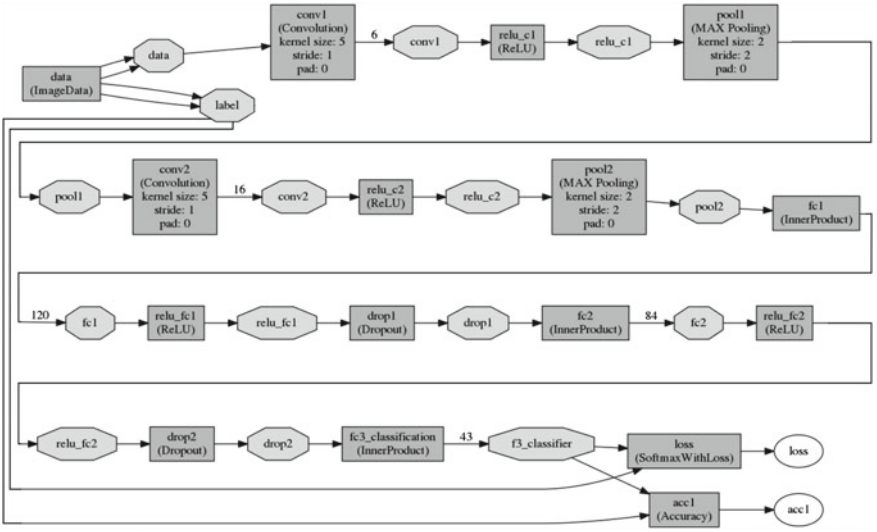

Fig. 4.9 Final architecture of the network. The architecture is similar to the architecture of LeNet-5 in nature. The differences are in activations functions, dropout layer, and connection in middle layers

```
layer{
    name:"data"
    type:"ImageData"
    top:"data"
    top:"label"
    image_data_param{
        source:"/home/hamed/Desktop/GTSRB/Training_CNN/train.txt"
        batch_size:30
        root_folder:"/home/hamed/Desktop/GTSRB/Training_CNN/"
        is_color:true
        shuffle:true
        new_width:32
        new_height:32
    }
}
layer{
    name:"data"
    type:"ImageData"
    top:"data"
    top:"label"
    image_data_param{
        source:"/home/hamed/Desktop/GTSRB/Training_CNN/validation.txt"
        batch_size:10
        root_folder:"/home/hamed/Desktop/GTSRB/Validation_CNN/"
        is_color:true
        shuffle:false
        new_width:32
        new_height:32
    }
}
```

First, the tops of these two layers have to be identical. This is due to the fact the first convolution layer is connected to a top called *data*. If we set top in the second ImageData layer to another name, the convolution layer will not receive any data during validation. Second, the variable *source* in the second layer points to the validation set. Third, the batch sizes of these two layers can be different. Usually, if memory on the GPU device is limited, we usually set the batch size of training set to the appropriate value and then set the batch size of the validation set according to the memory limitations. For instance, we have to set the batch size of validation samples to 10. Fourth, shuffle must be set to false in order to prevent unequal validations sets. In fact, the parameters that we will explain in the next section are adjusted such that the validation set is only scanned once in every test.

However, a user may forget to adjust this parameter properly and some of samples in validation set are fetched more than one time to the network. In that case, if *shuffle* is set to true it is very likely that some samples in two validation steps are not identical. This makes the validation result inaccurate. We alway want to test/validate the different models or same models in different time on exactly identical datasets.

During testing, the data has to **only** come from the first ImageData layer. During validation, the data has to **only** come from the second ImageData layer. One missing piece in the above definition is that how should Caffe understand when to switch from one ImageData layer to another. There is a variable in definition of LayerParameter called *include* which is an instance of NetStateRule.

```
message NetStateRule {
    optional Phase phase = 1;
}
```

When this variable is specified, Caffe will include the layer based on the state of training. This can be explained better in an example. Let us update the above two ImageData layers as follows:

```
layer{
name:"data"
    type:"ImageData"
    top:"data"
    top:"label"
    include{
        phase:TRAIN
    }
    image_data_param{
        source:"/home/hamed/Desktop/GTSRB/Training_CNN/train.txt"
        batch_size:30
        root_folder:"/home/hamed/Desktop/GTSRB/Training_CNN/"
        is_color:true
        shuffle:true
        new_width:32
        new_height:32
    }
    phase:
}
layer{
    name:"data"
    type:"ImageData"
    top:"data"
    top:"label"
    include{
        phase:TRAIN
```

```
    }
    image_data_param{
        source:"/home/hamed/Desktop/GTSRB/Training_CNN/validation.txt"
        batch_size:10
        root_folder:"/home/hamed/Desktop/GTSRB/Validation_CNN/"
        is_color:true
        shuffle:false
        new_width:32
        new_height:32
    }
}
```

During training a network, Caffe alternatively changes its state between TRAIN and TEST based on a parameter called *test_interval* (this parameters will be explain in the next section). In the TRAIN phase, the second ImageData layer will be discarded by Caffe. In contrast, the first layer will be discarded and the second layer will be included in the TEST phase. If the variable *include* is not instantiated in a layer, the layer will be included in both phases. We apply the above changes on the text file and save it

Finally, we add a layer to our network in order to compute the accuracy of the network on test samples. This is simply done by adding the following definition right after the loss layer.

```
layer{
    name:"acc1"
    type:"Accuracy"
    bottom:"classifier"
    bottom:"label"
    top:"acc1"
    include{ phase:TEST }
}
```

4.4 Training a Network

In order to train a neural network in Caffe, we have to design another text file and instantiate a SolverParameter inside this file. All required rules for training a neural network will be specified using SolverParameter.

```
message SolverParameter {
    optional string net = 24;
    optional float base_lr = 5;

    repeated int32 test_iter = 3;
    optional int32 test_interval = 4 [default = 0];
    optional int32 display = 6;

    optional int32 max_iter = 7;
    optional int32 iter_size = 36 [default = 1];

    optional string lr_policy = 8;
    optional float gamma = 9;
    optional float power = 10;
    optional int32 stepsize = 13;

    optional float momentum = 11;
```

```
    optional float weight_decay = 12;
    optional string regularization_type = 29 [default = "L2"];
    optional float clip_gradients = 35 [default = -1];

    optional int32 snapshot = 14 [default = 0];
    optional string snapshot_prefix = 15;

    enum SolverMode {
        CPU = 0;
        GPU = 1;
    }
    optional SolverMode solver_mode = 17 [default = GPU];
    optional int32 device_id = 18 [default = 0];

    optional string type = 40 [default = "SGD"];
}
```

The string variable *net* points to the *.prototxt* file that includes the definition of the network. In our example, this variable is set to *net="/home/pc/cnn.prototxt"*. The variable *base_lr* denotes the base learning rate. The effective learning rate at each iteration is defined based on the value of *lr_policy*, *gamma*, *power*, and *stepsize*. Recall from Sect. 3.6.4 that the learning rate is usually decreased over time. We explained different methods for decreasing the learning rate. In Caffe, setting *lr_policy="exp"* will decrease the learning rate using exponential rule. Likewise, setting this parameter to "step" and "inv" will decrease the learning rate using step method and the inverse method.

The parameter *test_iter* tells Caffe how many mini-batches it should use during test phase. The total number of samples that is used in the test phase will be equal to *test_iter* $\times$ *batch size of test ImageData layer*. The variable *test_iter* is usually set such that the test phase covers all samples of validation set without using a sample twice. Caffe will change its phase from TRAIN to TEST every *test_interval* iterations (mini-batches). Then, it will run the TEST phase for *test_iter* mini-batches and changes its phase to TRAIN again.

While Caffe is training the network, it produces human-readable output. The variable *display* will show this information in the console and write them into a log file for every *display* iterations. Also, the variable *max_iter* shows the maximum number of iterations that must be performed by the optimization algorithm. The log file is accessible in director */tmp* in Ubuntu.

Sometimes, because images are large or memory on GPU device is limited, it is not possible to set mini-batch size of training samples to an appropriate value. On the other hand, if the size of mini-batch is very small, gradient descend is likely to have a very zigzag trajectory and in some cases it may even jump over a (local) minimum. This makes the optimization algorithm more sensitive to the learning rate. Caffe alleviates this problem by first accumulating gradients of *iter_size* mini-batches and updating parameters based on accumulated gradients. This makes it possible to train large networks when memory on the GPU device is not sufficient.

As it turns out, the variable *momentum* determines the value of momentum in the momentum gradient descend. It is usually set to 0.9. The variable *weight_decay* shows the value of λ in the L_1 and L_2 regularizations. The type of regularization is also defined using the string variable *regularization_type*. This variable can be

only set to ”L1” or ”L2”. The variable *clip_gradients* defines the threshold in the max-norm regularization method (Sect. 3.6.3.3).

Caffe stores weights and state of optimization algorithm inside a folder at *snapshot_prefix* for every *snapshot* iteration. Using these files, you can load the parameters of the network after training or resume training from a specific iteration.

The optimization algorithm can be executed on CPU of a GPU. This is specified using the variable *solver_mode*. In the case that you have more than one graphic cards, the variable *device_id* tells Caffe which graphic must be used for computations.

Finally, the string variable *type* determines the type of optimization algorithm. In the rest of this book, we will always use ”SGD” which refers to mini-batch gradient descend. Other optimization algorithms such as Adam, AdaGrad, Nesterov, RMSProp, and AdaDelta are also implemented in the Caffe library. For our example, we write the following protobuf text in a file called *solver.prototxt*.

```
net:'/tmp/cnn.prototxt'
type:"SGD"

base_lr : 0.01

test_iter : 50;
test_interval :500;
display : 50

max_iter : 30000

lr_policy: "step"
stepsize :3000
gamma : 0.98

momentum :0.9
weight_decay :0.00001

snapshot: 1000
snapshot_prefix: 'cnn'
```

After creating the text files for the network architecture and for the optimization algorithm, we can use command tools of the Caffe library to train and evaluate the network. Specifically, running the following command in Terminal of Ubuntu will train the network:

```
./caffe-master/build/tools/caffe train --solver "/PATH_TO_SOLVER/solver.prototxt"
```

4.5 Designing in Python

Assume we have 100 GPUs in which we can train a big neural network on each of them, separately. With these resources available, our aim is to generate 1000 different architectures and train/validate each of them on one of these GPUs. Obviously, it is not tractable for a human to create 1000 different architectures in text files. The situation gets even more impractical if our aim is to generate 1000 significantly different architectures.

The more efficient solution is to generate these files using a computer program. The program may use heuristics to create different architectures or it may generate them randomly. Regardless, the program must generate text files including the definition of network.

The Caffe library provides a Python interface that makes it possible to use Caffe functions in a Python program. The Python interface is located at *caffe-master/python*. If this path is not specified in the *PYTHONPATH* environment variable, importing the Python module of Caffe will cause an error. To solve this problem, you can either set the environment variable or write the following code before importing the module:

```
import sys
sys.path.insert(0, "/home/pc/caffe-master/python")
import caffe
```

In the above script, we have considered that the Caffe library is located at "*/home/pc/caffe-master/*". If you open *__init__.py* from *caffe-master/python/caffe/* you will find the name of functions, classes, objects, and attributes that you can use in your Python script. Alternatively, you can run the following code to obtain the same information:

```
import sys
sys.path.insert(0, "/home/pc/caffe-master/python")
import caffe

print dir(caffe)
```

In order to design a network, we should work with two attributes called *layers* and *params* and a class called *NetSpec*. The following Python script creates a ConvNet identical to the network we created in the previous section.

```
import sys
sys.path.insert(0, "/home/hamed/caffe-master/python")
import caffe

L = caffe.layers
P = caffe.params

def conv_relu(bottom, ks, nout, stride=1, pad=0):
    c = L.Convolution(bottom, kernel_size=ks, num_output=nout,
                      stride=stride, pad=pad,
                      weight_filler={'type':'xavier'},
                      bias_filler={'type':'constant',
                                   'value':0})
    r = L.ReLU(c)
    return c, r

def fc_relu_drop(bottom, nout):
    fc = L.InnerProduct(bottom, num_output=nout,
                        weight_filler={'type':'xavier'},
                        bias_filler={'type':'constant',
                                     'value':0})
    r = L.ReLU(fc)
    d = L.Dropout(r, dropout_ratio=0.4)
    return fc, r, d

net = caffe.net_spec.NetSpec()

net.data, net.label = L.ImageData(source='/home/hamed/Desktop/train.txt',
                                  batch_size=30, is_color=True,
```

```
                                    shuffle=True, new_width=32,
                                    new_height=32, ntop=2)

net.conv1, net.relu1 = conv_relu(net.data, 5, 16)
net.pool1 = L.Pooling(net.relu1, kernel_size=2,
                                    stride=2, pool=P.Pooling.MAX)

net.conv2, net.relu2 = conv_relu(net.pool1, 5, 16)
net.pool2 = L.Pooling(net.relu2, kernel_size=2,
                                    stride=2, pool=P.Pooling.MAX)

net.fc1, net.fc_relu1, net.drop1 = fc_relu_drop(net.pool2, 120)
net.fc2, net.fc_relu2, net.drop2 = fc_relu_drop(net.drop1, 84)
net.f3_classifier = L.InnerProduct(net.drop2, num_output=43,
                                        weight_filler={'type':'xavier'},
                                        bias_filler={'type':'constant',
                                                'value':0})
net.loss = L.SoftmaxWithLoss(net.classifier, net.label)

with open('cnn.prototxt', 'w') as fs:
    fs.write(str(net.to_proto()))
    fs.flush()
```

In general, creating a layer can be done using the following template:

```
net.top1, net.top2,...., net.topN = L.LAYERTYPE(bottom1, bottom2,...,bottomM,
    kwarg1=value, kwarge=value, kwarg=dict(kwarg=value,...), ..., ntop=N)
```

The number of tops in a layer is determined using the argument *ntop*. Using this method, the function will generate *ntop* top(s) in the output. Hence, there have to be *N* variables in the left side assignment operator. The name of tops in the text file will be "top1", "top2" and so on. That said, if the first top of the function is assigned to *net.label*, it is analogous to putting *top="label"* in the text file.

Also, note that the assignments have to be done on *net.**. If you study the source code of NetSpec, you will find that the *__setattr__* of this class is designed in a special way such that executing:

```
    net.DUMMY_NAME = value
```

will actually create an entry in a dictionary with key *DUMMY_NAME*.

The next point is that calling *L.LAYERTYPE* will actually create a layer in the text file where type of the layer will be equal to *type="LAYERTYPE"*. Therefore, if we want to create a convolution layer, we have to call L.Convolution. Likewise, creating pooling, loss and ReLU layers is done by calling L.Pooling, L.SoftmaxWithLoss, and L.ReLU, respectively.

Any argument that is passed to L.LAYERTYPE function will be considered as the bottom of the layer. Also, any keyword argument will be treated as the parameters of the layer. In the case that there is a parameter in a layer such as *weight_filler* with a data type other than basic types, the inner parameters of this parameter can be defined using a dictionary in Python.

After that the architecture of network is defined, it can be simply converted to a string by calling *str(net.to_proto())*. Then, this text can be written into a text file and stored on disk.

4.6 Drawing Architecture of Network

The Python interface provides a function for generating a graph for a given network definition text file. This can be done by calling the following function:

```
import sys
sys.path.insert(0, "/home/hamed/caffe-master/python")
import caffe
import caffe.draw
from caffe.proto import caffe_pb2
from google.protobuf import text_format

def drawcaffe(def_file, save_to, direction ='TB'):
    net = caffe_pb2.NetParameter()
    text_format.Merge(open(def_file).read(), net)

    caffe.draw.draw_net_to_file(net, save_to, direction)
```

This function uses the *GraphViz* Python module to generate the diagram. The parameter *direction* shows the direction of the graph and it might be called by passing 'TB' (top-bottom), 'BT' (bottom-top), 'LR' (left-right), 'RL' (right-left). The diagrams indicated in this chapter are created by calling this function.

4.7 Training Using Python

After creating the *solver.prototxt* file, we can use it for training the network by writing a Python script rather than command tools. The Python script for training a network might look like:

```
caffe.set_mode_gpu()
solver = caffe.get_solver('/tmp/solver.prototxt')
solver.step(25)
```

The first line in this code tells Caffe to use GPU instead of CPU. If this command is not executed, Caffe will use CPU by default. The second line in this code loads the solver definition. Because the path of network is also mentioned inside the solver definition, the network is also automatically loaded. Then, calling the *step(25)* function, runs the optimization algorithm for 25 iterations and stops. Assume that *test_interval=100* and we call *solver.step(150)*. If the network is trained using command tools, Caffe will switch from TRAIN to TEST when immediate after 100^{th} iteration. This will also happen when *solver.step(150)* is called. Hence, if you want that the test phase is not automatically invoked by Caffe, the variable *test_interval* must be set to a large number (larger than the variable *max_iter*).

4.8 Evaluating Using Python

Any neural network must be evaluated in three stages. The first evaluation is done during training using training set. The second evaluation is done during training using validation set and the third evaluation is done using test set after that designing and training the network is completely done.

Recall from Sect. 3.5.3 that a network is usually evaluated using a classification metric. All the classification metrics that we explained in that section are based on actual labels and predicted labels of samples. Actual labels of samples are already available in the dataset. However, predicted labels are obtained using the network. That means in order to evaluate a network using one of the classification metrics, it is necessary to predict labels of samples. These samples may come from the training set, the validation set or the test set.

In the case of neural network, we have to feed the samples to the network and forward them through the network. The output layer shows the score of samples for each class. For example, the output layer of the network in Sect. 4.5 is called f3_classifier. We can access the value of the network computed for a sample using the following command:

```
solver = caffe.get_solver('/tmp/solver.prototxt')
net = solver.net
print net.blobs['classifier'].data
```

In the above script, the first line loads a solver along with the network. The filled *solver.net* returns the network that is used for training. In Caffe, a tensor that retains data is encapsulated in objects of type Blob. The field *net.blobs* is a dictionary where keys of this dictionary are the *tops* of network that we have specified in the network definition and value of each entry in this dictionary is an instance of Blob. For example, the top of the classification layer in Sect. 4.5 is called "classifier". The command *net.blobs['classifier']* returns the blob associated with this layer.

The tensor of a blob is accessible through the field *data*. Hence, *net.blobs['KEY'].data* returns the numerical data in a 4D matrix (tensor). This matrix is in fact a Numpy array. The shape of tensors in Caffe is $N \times C \times H \times W$, where N denotes the number of samples in mini-batch and C illustrates the number of channels. As it turns out, H and W also denote the height and width, respectively.

The batch size of the layer "data" in Sect. 4.5 is equal to 30. Also, this layer loads color images (3 channels) of size 32×32. Therefore, the command *net.blobs['data'].data* returns a 4D matrix of shape $40 \times 3 \times 32 \times 32$. Taking into account the fact that layer "classifier" in this network contains 43 neurons, the command *net.blobs['classifier'].data* will return a matrix of size $40 \times 43 \times 1 \times 1$, where each row in this matrix shows class specific score of the first samples in the mini-batch. Each sample belongs to the class with the highest score.

Assume we want to classify a single image which is stored at */home/sample.ppm*. This means that, the size of mini-batch is equal to 1. To this end, we have to load the image in RGB format and resize it to 32×32 pixels. Then, transpose the axis such that the shape of image becomes $3 \times 32 \times 32$. Finally, this matrix has to be

converted to a $1 \times 3 \times 32 \times 32$ matrix in order to make it compatible with tensors in Caffe. This can be easily done using the following commands:

```
import numpy as np
im = caffe.io.load_image('/home/sample.ppm', color=True)
im = caffe.io.resize(im, (32, 32))
im = np.transpose(im, [2,0,1])
im = im[np.newaxis, ...]
```

Next, this image has to be fed into the network and the output layers must be computed one by one. Technically, this is called forwarding the samples throughout the network. Assuming that *net* is an instance of Caffe.Net, forwarding the above sample can be easily done by calling:

```
net.blobs['data'].data[...] = im[...]
net.forward()
```

It should be noted that [...] in the above code the image into the memory of field *data*. Removing this from the above line will raise an error since it will mean that we are assigning a new memory to the field *data* rather than updating its memory. At this point, *net.blobs[top].data* returns the output of a top in network. In order to classify the above image in our network, we only need to run the following line:

```
label = np.argmax(net.blobs['classifier'].data, axis=1)
```

This will return the index of the class with maximum score. The general procedure for training a ConvNet is illustrated below.

```
Givens:
        X_train: A dataset containing N images of size WxHx3
        Y_train: A vector of length N containing labels of each samples in X_train

        X_valid: A dataset containing K images of size WxHx3
        Y_valid: A vector of length K containing labels of each samples in X_valid

FOR t=1 TO MAX
    TRAIN THE CONVNET FOR m ITERATIONS USING X_train and Y_train

    EVALUATE THE CONVNET USING X_valid and Y_valid
END FOR
```

The training procedure involves constantly updating parameters using the training set and evaluating the network using the validation set. More specifically, the network is trained for m iterations using the training samples. Then, validations samples are fetched into the network and a classification metric such as accuracy is computed for the samples in the validation set. The above procedure is repeated MAX times and the training is finished. One may wonder why the network must be evaluated during training. As we will see in the next chapter, validation is a crucial step in training a classification model such as neural networks. The following code shows how to implement the above procedure in Python:

```
solver = caffe.get_solver('solver.prototxt')

with open('validation.txt','r') as file_id:
    valid_set = csv.reader(file_id, delimiter=' ')
    valid_set = [(row[0], int(row[1])) for row in valid_set]

```

```
net_valid = solver.test_nets[0]                                                    7
data_val = np.zeros(net_valid.blobs['data'].data.shape, dtype='float32')           8
label_actual = np.zeros(net_valid.blobs['label'].data.shape, dtype='int8')         9
for i in xrange(500):                                                              10
    solver.step(1000)                                                              11
                                                                                   12
    print 'Validating ...'                                                         13
    acc_valid = []                                                                 14
    net_valid.share_with(solver.net)                                               15
                                                                                   16
    batch_size = net_valid.blobs['data'].data.shape[0]                             17
    cur_ind = 0                                                                    18
                                                                                   19
    for _ in xrange(800):                                                          20
        for j in xrange(batch_size):                                               21
            rec = valid_set[cur_ind]                                               22
            im = cv2.imread(rec[0], cv2.cv.CV_LOAD_IMAGE_COLOR).astype('float32')  23
            im = im / 255.                                                         24
            im = cv2.resize(im, (32, 32))                                          25
            im = np.transpose(im, [2,0,1])                                         26
                                                                                   27
            data_val[j, ...] = im                                                  28
            label_actual[j, ...] = rec[1]                                          29
            cur_ind = cur_ind + 1 if ((cur_ind+1) < len(valid_set)) else 0         30
                                                                                   31
        net_valid.blobs['data'].data[...] = data_val                               32
        net_valid.blobs['label'].data[...] = label_actual                          33
        net_valid.forward()                                                        34
                                                                                   35
        class_score = net_valid.blobs['classifier'].data.copy()                   36
        label_pred = np.argmax(class_score, axis=1)                                37
        acc = sum(label_actual.ravel() == label_pred) / float(label_pred.size)     38
        acc_valid.append(acc)                                                      39
    mean_acc = np.asarray(acc_valid).mean()                                        40
    print 'Validation accuracy: {}'.format(mean_acc)                               41
```

First line loads the solver together with the train and test networks associated with this solver. Line 3 to Line 5 read the validation dataset into a list. Line 8 and Line 9 create containers for validation samples and their labels. The training loop starts at Line 10 and it will be repeated 500 times. The first statement in this loop (Line 11) is to train the network using training samples for 1000 iterations.

After that, validating the network starts at Line 13. The idea is to load 800 mini-batches of validation samples, where each mini-batch contains *batch_size* samples. The loop from Line 21 to Line 30, loads color images and resize them using OpenCV functions. It also rescales the pixel intensities to [0, 1]. Rescaling is necessary since the training samples are also rescaled by setting *scale:0.0039215* in the definition of the ImageData layer.[6]

The loaded images are transposed and copied to *data_val* tensor. Label of each sample is also copied into *label_actual* tensor. After filling the mini-batch, it is copied into the first layer of the network in Line 32 and Line 33. Then, it is forwarded throughout the network at Line 34.

[6]It is possible to load and scale images using functions in caffe.io module. However, it should be noted that the *imread* function from OpenCV loads color images in BGR order rather than RGB. This is similar to the way the ImageData layer loads images using OpenCV. In the case of using *caffe.io.load_image* function, we must swap R and B channel before feeding them to the network.

Line 36 and Line 37 finds the class of each samples and the accuracy of classification is computed on the mini-batch and it is stored in a list. Finally, the mean accuracy of 800 mini-batches is computed and stored in *mean_acc*. The above code can be used as a basic template for training and validating neural network in Python using Caffe library. It is also possible to keep history of training and validation accuracies in the above code.

However, there are a few points to bear in mind. First, the same transformations must be applied on the validation/test samples as we have used for training samples. Second, the validation samples must be identical every time the network is evaluated. Otherwise, it might not be trivial to assess the network properly. Third, as we discussed earlier, F1-score can be computed over all validation samples rather than accuracy.

4.9 Save and Restore Networks

During training, we might want to save and restore the parameters of the network. In particular, we will need the value of trained parameters in order to load them into the network and use the network in real-world applications. This can be done by writing a customized function to read the value of *net.params* dictionary and save them in a file. Later, we can load the same values to *net.params* dictionary.

Another way is to use the built-in functions in Caffe library. Specifically, the *net.save(string filename)* and the *net.copy_from(string filename)* functions saves the parameters into a binary file and loads them into the network, respectively.

In some cases, we may also want to save information related to the optimizer such as current iteration, current learning rate, current momentum, etc., besides the parameters of network. Later, this information can be loaded into the optimizer as well as the network in order to resume the training from the last stopped point. Caffe provides *solver.snapshot()* and *solver.restore(string filename)* functions for these purposes.

Assume the field *snapshot_prefix* is set to *"/tmp/cnn"* in the solver definition file. Calling *solver.snapshot()* will create two files as follows:

```
/tmp/cnn_iter_X.caffemodel
/tmp/cnn_iter_X.solverstate
```

where *X* is automatically replaced by Caffe with the current iteration of the optimization algorithm. In order to restore the state of the optimization algorithm from a disk, we only need to call *solver.restore(filename)* with a path to a valid *.solverstate* file.

4.10 Python Layer in Caffe

One limitation of the Caffe library is that we are obliged to only utilize the implemented layers of this library. For example, the softplus activation function is not implemented in the current version of the Caffe library. In some cases, we may want to add a layer with a new function that is not implemented in the Caffe library. The obvious solution is to implement this layer directly in C++ by inheriting our classes from classes of the Caffe library. This could be a tedious task especially when the goal is to quickly implement and test our idea.

A more likely scenario in which having a special layer could be advantageous when we work with different datasets. For instance, there are thousands of samples in the GTSRB dataset for the task of traffic sign classification. The bounding box information of each image is provided using a text file. Apparently, these images have to be cropped to exactly fit the bounding box before feeding to a classification network.

This can be done in three ways. The first way is to process whole dataset and crop each image based on their bounding box information and store them on the disk. Then, the processed dataset can be used for training/validation/testing the network. The second solution is to process images on the fly and fill each mini-batch after processing the images. Then these mini-batches can be used for training/validation/testing. However, it should be noted that using this method we will not be longer able to call the *solver.step(int)* function with an argument greater than one or set *iter_size* to a value greater than one. The reason is that, each mini-batch must be filled manually using our code. The third method is to develop a new layer which automatically reads images from the dataset, processes, and passes them to the output (top) of the layer. Using this method, the *solver.step(int)* function can be called with any arbitrary positive number.

The Caffe library provides a special type of layer called PythonLayer. Using this layer, we are able to develop new layers in Python which can be accessed by Caffe. A Python layer is configured using an instance of PythonParameter which is defined as follows:

```
message PythonParameter {
    optional string module = 1;
    optional string layer = 2;
    optional string param_str = 3 [default = ''];
}
```

Based on this definition, a Python layer might look like:

```
layer {
    name: "data"
    type: "Python"
    top: "data"
    python_param {
        module: "python_layer"
        layer: "mypythonlayer"
        param_str: "{\'param1\':1, \'param2\':2.5}"
    }
}
```

The variable *type* of a Python layer must be set to *Python*. Upon reaching to this layer, Caffe will look for *python_layer.py* file next to the *.prototxt* file. Then, it will look for a class called *mypythonlayer* inside this file. Finally, it will pass *"ṕaram1':1, ṕaram2':2.5"* into this class. Caffe will interact with *mypythonlayer* using four methods inside this class. Below is the template that must be followed in designing a new layer in Python.

```
class mypythonlayer(caffe.Layer):
    def setup(self, bottom, top):
        pass

    def reshape(self, bottom, top):
        pass

    def forward(self, bottom, top):
        pass

    def backward(self, top, propagate_down, bottom):
        pass
```

First, the class must be inherited from *Caffe.Layer*. The *setup* method will be called only once when Caffe creates train and test networks. The *backward* method is only called during the backpropagation step. Computing the output of each layer given an input is done by calling *net.forward()* method. Whenever this method is called, the *reshape* and *forward* methods of the layer will be called automatically. The *reshape* method is always called before *forward* method.

It is noteworthy to draw you attention to the prototype of the *backward* method. In contrast to the other three methods, where the first argument is *bottom* and the last argument is *top*, in the *backward* method the places of these two arguments are switched. So, a great care must be taken into account in defining the prototype of the method. Otherwise, you may end up with a layer, where the gradients are not computed correctly. For instance, let us implement the PReLU activation using the Python layer. In this implementation, we consider a distinct PReLU activation for each feature map.

```
class prelu(caffe.Layer):
    def setup(self, bottom, top):
        params = eval(self.param_str)
        shape = [1]*len(bottom[0].data.shape)
        shape[1] = bottom[0].data.shape[1]
        self.axis = range(len(shape))
        del self.axis[1]
        self.axis = tuple(self.axis)

        self.blobs.add_blob(*shape)
        self.blobs[0].data[...] = params['alpha']

    def reshape(self, bottom, top):
        top[0].reshape(*bottom[0].data.shape)

    def forward(self, bottom, top):
        top[0].data[...] = np.where(bottom[0].data > 0,
                                    bottom[0].data,
                                    self.blobs[0].data*bottom[0].data)

    def backward(self, top, propagate_down, bottom):
        self.blobs[0].diff[...] = np.sum(np.where(bottom[0].data > 0,
                                                  np.zeros(bottom[0].data.shape),
```

```
                                        bottom[0].data) * top[0].diff,                24
                                    axis=self.axis, keepdims=True)                    25
        bottom[0].diff[...] = np.where(bottom[0].data > 0,                            26
                                    np.ones(bottom[0].data.shape),                    27
                                    self.blobs[0].data) * top[0].diff                 28
```

The *setup* method converts the *param_str* value specified in the network definition into a dictionary. Then, the shape of parameter vector is determined. Specifically, if the shape of bottom layer is $N \times C \times H \times W$, the shape of parameter vector must be $1 \times C \times 1 \times 1$. The dimensions of array with length 1 will be broadcasted during operations by Numpy. Since there are C feature maps in the bottom layer, there must also be C PReLU activations with different values of α.

In the case of fully connected layers, the bottom layer might be a two-dimensional array instead of four-dimensional array. The *shape* variable in this method ensures that the parameter vector will have a shape consistent with the bottom layer.

The variable *axis* indicates the axis to which the summation of gradient must be performed. Again, this axis also must be consistent with the shape of bottom layer.

Line 10 creates a parameter array in which the shape of this array is determined using the variable *shape*. Note the unpacking operator in this line. Line 11 initializes α of all PReLU activations with a constant number. The setup method is called once and it initializes all parameters of the layer.

The reshape method, determines the shape of top layer in Line 14. The channel-wise PReLU activations are applied on the bottom layer and assigned to the top layer. Note how we have utilized broadcasting of Numpy arrays in order to multiply parameters with the bottom layer. Finally, the backward method computes the gradient with respect to parameters and gradient with respect to the bottom layer.

4.11 Summary

There are various powerful libraries such as Theano, Lasagne, Keras, mxnet, Torch, and TensorFlow that can be used for designing and training neural networks including convolutional neural networks. Among them, Caffe is a library that can be used for both doing research and developing real-world applications. In this chapter, we explained how to design and train neural networks using the Caffe library. Moreover, the Python interface of Caffe was discussed using real examples. Then, we mentioned how to develop new layers in Python and use them in neural networks.

4.12 Exercises

4.1 Suppose the following text files:

```
/sample1.jpg 0                                                                        1
/sample2.jpg 0                                                                        2
/sample3.jpg 0                                                                        3
```

```
/sample4.jpg 0
/sample5.jpg 1
/sample6.jpg 1
/sample7.jpg 1
/sample8.jpg 1
```

```
/sample7.jpg 1
/sample1.jpg 0
/sample3.jpg 0
/sample6.jpg 1
/sample4.jpg 0
/sample5.jpg 1
/sample2.jpg 0
/sample8.jpg 1
```

From optimization algorithm perspective, which one of the above files is appropriate for passing to an ImageData layer? Also, which of these files hast to be shuffled before starting the optimization? Why?

4.2 Shifted ReLU activation is given by Clevert et al. (2015):

$$f(x) = \begin{cases} x - 1 & x > 0 \\ -1 & otherwise \end{cases} \tag{4.1}$$

This activation function is not basically implemented in Caffe. However, you can implement it using current layers in this library. Use a ReLU layer together with Bias layer to implement this activation function in Caffe. A bias layer basically adds a constant to bottom blobs. You can find more information in *caffe.proto* about this layer.

4.3 Why and when shuffle of an ImageData layer in TEST phase must be set to false.

4.4 When setting shuffle to true or false in TEST phase does not matter?

4.5 What happens if we add *include* to the first convolution layer in the network we mentioned in this chapter and set *phase=TEST* for this layer?

4.6 Add codes to the Python script in order to keep the history of training and validation accuracies and plot them using Python.

4.7 How we can check the gradient of the implemented PReLU layer using numerical methods?

4.8 Implement the softplus activation function using a Python layer.

Reference

Clevert DA, Unterthiner T, Hochreiter S (2015) Fast and accurate deep network learning by exponential linear units (ELUs) 1997:1–13. arXiv:1511.07289.pdf