Deep Learning With Caffe In Python – Part I: Defining A Layer

In this lecture, we will discuss how to get started with Caffe and use its various features. We will then build a convolutional neural network (CNN) that can be used for image classification. Caffe plays very well with the GPU during the training process, hence we can achieve a lot of speed-up. For the purpose of this discussion, it is assumed that you have already installed Caffe on your machine. Let's go ahead and see how to interact with Caffe, shall we?

Prerequisites

Create a python file and add the following lines:

```
import sys
import numpy as np
import matplotlib.pyplot as plt

sys.insert('/path/to/caffe/python')
import caffe
```

If you have a GPU onboard, then we need to tell Caffe that we want it to use the GPU:

```
caffe.set_device(0)
caffe.set_mode_gpu()
```

We are now ready to build a network.

Building a simple layer

As the name suggests, convolutional neural networks (CNNs) rely heavily on convolutions. Big surprise, right? CNNs are still basically neural networks, which means they consist of multiple layers joined together. There are many different types of layers that can be used to build a CNN, convolution layer being one of them. Let's go ahead and see how we can define a simple convolution layer in Caffe. Create a file called "myconvnet.prototxt" and add the following lines to it:

```
name: "myconvolution"
input: "data"
input_dim: 1
input_dim: 1
input_dim: 256
input_dim: 256
layer {
  name: "conv"
  type: "Convolution"
  bottom: "data"
  top: "conv"
  convolution_param {
    num_output: 10
    kernel_size: 3
    stride: 1
    weight_filler {
```

```
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
```

We just defined a single layer CNN consisting of 10 convolutional neurons (as specified by "num_output") with a kernel size of 3×3 (as specified by "kernel_size") and a stride of 1 (as specified by "stride").

Visualizing the network

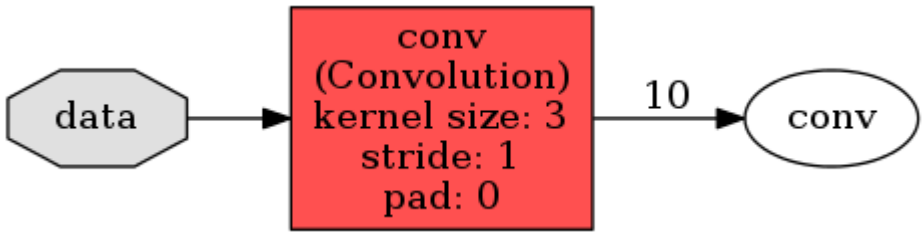
Before we understand how to use the above network, let's see what it actually looks like! The good thing about Caffe is that it provides a way to visualize our network with a simple command. Before that, we need to install pydot and graphviz. Run the following on your terminal:

```
$ pip install pydot
$ sudo apt-get install graphviz libgraphviz-dev
$ pip install pygraphviz
```

We are now ready to visualize the network. Run the following command on your terminal:

```
$ python /path/to/caffe/python/draw_net.py myconvnet.prototxt myconvnet.png
```

Now, if you open "myconvnet.png", you will see something like this:



Pretty sweet, right? In the next lab, we will discuss how to load this network and interact with it.

Deep Learning With Caffe In Python – Part II: Interacting With A Model

In previous lab, we talked about how to define and visualize a single layer convolutional neural network (CNN). In this lab, we will discuss how to interact with a Caffe model. This is a continuation of the previous lab manual.

Interacting with the neural network

Let’s go ahead and load our single layer network. Open the python file designed before and add the following line:

```
import sys
sys.path.insert(0,'/home/ucerd/soft/caffe/caffe/python')
import caffe
import cv2
import numpy as np

net = caffe.Net('myconvnet.prototxt', caffe.TEST)

print "\nnet.inputs =", net.inputs
print "\ndir(net.blobs) =", dir(net.blobs)
print "\ndir(net.params) =", dir(net.params)
print "\nconv shape = ", net.blobs['conv'].data.shape
```

We just created the “net” object to hold our convolutional neural network. You can access the names of input layers using “net.inputs”. You can see them by adding “print net.inputs” to your python file.

This “net” object contains two dictionaries — net.blobs and net.params. Basically, net.blobs is for data in the layers and net.params is for the weights and biases in the network. You can check them out using dir(net.blobs) and dir(net.params). In this case, net.blobs[‘data’] would contain an array of shape (1, 1, 256, 256). Now why does it have 4 dimensions if we are dealing with a simple 2D grayscale image? The first ‘1’ refers to the number of images and the second ‘1’ refers to the number of channels in the image. Caffe uses this format for all data. If you check net.blobs[‘conv’], you’ll see that it contains the output of the ‘conv’ layer of shape (1, 10, 254, 254). If you run a 3×3 kernel over a 256×256 image, the output will be of size 254×254, which is what we get here.

Let’s inspect the parameters:

- net.params[‘conv’][0] contains the weight parameters of our neurons. It’s an array of shape (10, 1, 3, 3) initialized with “weight_filler” parameters. In the prototxt file, we have specified “Gaussian” indicating that the kernel values will have Gaussian distribution.
- net.params[‘conv’][1] contains the bias parameters of our neurons. It’s an array of shape (10,) initialized with “bias_filler” parameters. In the prototxt file, we have specified “constant” indicating that the values will remain constant (0 in this case).

Caffe handles data as “blobs”, which are basically memory abstraction objects. Our data is contained as an array in the field named ‘data’.

Extracting the output of the network

Let’s consider the following image:

and see how to computecthe output for this image. The size of the above image is 960×640, so we need to reshape the data blob from (1, 1, 256, 256) to (1, 1, 960, 640) so that it fits the image.

```
img = cv2.imread('input_image.jpg', 0)
img_blobinp = img[np.newaxis, np.newaxis, :, :]
net.blobs['data'].reshape(*img_blobinp.shape)
net.blobs['data'].data[...] = img_blobinp

net.forward()

for i in range(10):
                    cv2.imwrite('output_image_'     +     str(i)     +     '.jpg',
255*net.blobs['conv'].data[0,i])
```

The “net” object will be populated now. If you check “n”, you will see that it will be filled with data. We can plot the pictures inside each of the 10 neurons in the layer. If you want to plot the image in the nth neuron, you can do access it using net.blobs[‘conv’].data[0,n-1]. We just created 10 output files corresponding to the 10 neurons in the loop. Let’s check what the output from, say, the ninth neuron looks like:

As we can see here, it's an edge detector. You can check out the other files to see the different types of filters generated.

We want the ability to reuse this layer without going through the process again. So let's save it:

```
net.save('myconvmodel.caffemodel')
```

We just created a single layer network in Caffe. You should play around with the "net" object until you get familiar with it. It is used extensively in deep learning applications built using Caffe.

Deep Learning With Caffe In Python – Part III: Training A CNN

In the previous lab, we learn about how to interact with a Caffe model. In this lab, we will learn how to train a proper CNN. Up until now, we were dealing with a single layer network. We just defined it in a prototxt file and visualized it easily. If we want our CNN to perform any meaningful tasks, we should define a multilayer network and allow it to train on a large amount of data. Caffe makes it very easy for us to train a multilayer network. We can specify all the parameters in a prototxt file, create a training database, and just train the network. Let's go ahead and see how to do that, shall we?

Training a deep neural network

We are now ready to create our own model. Make sure you have some labeled training data. If you don't have it, you can any of the datasets listed here. Before we start training a network, we need the following:

- Model definition: A prototxt file containing the model definition (like the one we had earlier)
- Learning algorithm: A prototxt file describing the parameters for the stochastic gradient algorithm. This is called the solver file.
- Mean image: We need to compute the mean image of the training dataset
- Training data: A text file containing the training data images in a specific format
- Testing data: A text file containing the test data images in a specific format

For now, you can take the files named "train_val.prototxt" and "solver.prototxt" located in "/path/to/caffe/models/bvlc_reference_caffenet". Rename them to "my_train_val.prototxt" and "my_solver.prototxt" so that it's clearer for you. Open up "my_solver.prototxt" in a text editor and change the params to make it look like this:

```
net: "my_train_val.prototxt"
test_iter: 1000
test_interval: 1000
base_lr: 0.001
lr_policy: "step"
gamma: 0.1
```

```
stepsize: 10000
display: 20
max_iter: 50000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix: "models/mymodel/train"
solver_mode: GPU
```

Make sure to create the folder “models/mymodel” in the current directory. The solver file “my_solver.prototxt” looks for the file “my_train_val.prototxt” in the same directory and the path “models/mymodel” should be relative to this directory.

We need to create two more files — train_files.txt and test_files.txt. These files should contain images and the coresponding labels in the following format:

```
/path/to/folder/image1.jpg 0
/path/to/folder/image2.jpg 3
/path/to/folder/image3.jpg 1
/path/to/folder/image4.jpg 2
/path/to/folder/image5.jpg 1
...
...
/path/to/folder/imageN.jpg N-1
```

The above file contains images divided into N classes (0-indexed). It’s important that the images are shuffled in both the text files. We want images from random classes to appear in a sequence.

Computing image mean

We need to compute image mean for our dataset in order to use it during training. This is an architectural specification that was derived from experimentation by researchers. This mean image will be subtracted from each image to boost the performance of the network. Caffe provides a way to compute the image mean directly. We need to generate the lmdb database for our training images so that Caffe can use it to generate the mean image. Run the following command to generate the lmdb database:

```
$ GLOG_logtostderr=1 /path/to/caffe/build/tools/convert_imageset
--resize_height=256 --resize_width=256 --shuffle / /path/to/train.txt
/path/to/train_lmdb
```

We are now ready to compute the mean image. Run the following command:

```
$ /path/to/caffe/build/tools/compute_image_mean /path/to/train_lmdb
/path/to/mean_image.binaryproto
```

Training the model

Open up my_train_val.prototxt and change the first few lines as given below:

```
layer {
  name: "data"
  type: "ImageData"
  top: "data"
  top: "label"
```

```
  include {
    phase: TRAIN
  }
  transform_param {
    mirror: true
    crop_size: 227
    mean_file: "my_image_mean.binaryproto"
  }
  image_data_param {
    source: "train_files.txt"
    batch_size: 50
    new_height: 256
    new_width: 256
  }
}
layer {
  name: "data"
  type: "ImageData"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  transform_param {
    mirror: true
    crop_size: 227
    mean_file: "my_image_mean.binaryproto"
  }
  image_data_param {
    source: "test_files.txt"
    batch_size: 50
    new_height: 256
    new_width: 256
  }
}
```

In the “fc8” layer, change the “num_output” parameter to the number of classes you have. We are now ready to train:

```
$ /full/path/to/caffe/build/tools/caffe train --solver
/full/path/to/my_solver.prototxt
```

If everything goes well, it will start printing the log messages on the terminal. You can look at the error values as they start converging with the number of iterations. Once the error is low enough, say 1e-6, you can stop the training. If it reaches the maximum number of iterations as specified in the solver file, it will stop by itself.

Deep Learning With Caffe In Python – Part IV: Classifying An Image

In the previous lab, we understand how to train a convolutional neural network (CNN). One of the most popular use cases for a CNN is to classify images. Once the CNN is trained, we need to know

how to use it to classify an unknown image. The trained model files will be stored as “caffemodel” files, so we need to load those files, preprocess the input images, and then extract the output tags for those images. In this lab work, we will see how to load those trained model files and use it to classify an image. Let’s go ahead see how to do it, shall we?

Loading the model

Training a full network takes time, so we will use an existing trained model to classify an image for now. There are many models available here for tasks such as flower classification, digit classification, scene recognition, and so on. We will be using the caffemodel file available here. Download and save it before you proceed. Open up a new python file and add the following line:

```
net = caffe.Net('/path/to/caffe/models/bvlc_reference_caffenet/deploy.prototxt',
'bvlc_reference_caffenet.caffemodel', caffe.TEST)
```

This will load the model into the “net” object. Make sure you substitute the right path in the input parameters in the above line.

Preprocessing the image

Let’s define the transformer:

```
transformer = caffe.io.Transformer({'data': net.blobs['data'].data.shape})
```

The function of the transformer is to preprocess the input image and transform it into something that Caffe can understand. Let’s set the mean image:

```
transformer.set_mean('data',
np.load('/path/to/caffe/python/caffe/imagenet/ilsvrc_2012_mean.npy').mean(1).mea
n(1))
```

The mean image needs to be subtracted from each input image. A couple of other params:

```
transformer.set_transpose('data', (2,0,1))
transformer.set_channel_swap('data', (2,1,0)) # if using RGB instead of BGR
transformer.set_raw_scale('data', 255.0)
```

The “set_transpose” function here will transform an image from (256,256,3) to (3,256,256). The “set_channel_swap” function will change the channel ordering. Caffe uses BGR image format, so we need to change the image from RGB to BGR. If you are using OpenCV to load the image, then this step is not necessary since OpenCV also uses the BGR format. The “set_raw_scale” function normalizes the values in the image based on the 0-255 range.

We need to reshape the blobs so that they match the image shape. Let’s add the following line to the python file:

```
net.blobs['data'].reshape(1,3,227,227)
```

The first input parameter specifies the batch size. Since we are only classifying one image, it is set to 1. The next three parameters correspond to the size of the cropped image. From each image, a

227×227 sub-image is taken for training in the model file that we loaded. This makes the model more robust. That’s the reason we are using 227 here!

Classifying an image

Let’s load the input image:

```
img = caffe.io.load_image('myimage.jpg')
net.blobs['data'].data[...] = transformer.preprocess('data', img)
```

Let’s compute the output:

```
output = net.forward()
```

The predicted output class can be printed using:

```
print output['prob'].argmax()
```

Download this file before you proceed. It contains the mapping required for the labels.Let’s print all the predicted labels:

```
label_mapping = np.loadtxt("synset_words.txt", str, delimiter='\t')
best_n = net.blobs['prob'].data[0].flatten().argsort()[-1:-6:-1]
print label_mapping[best_n]
```

The above lines will print all the labels predicted by the CNN. You will get the following output:

```
['n03837869 obelisk' 'n03743016 megalith, megalithic structure'
 'n02793495 barn' 'n03028079 church, church building' 'n02980441 castle']
```

During the course of these four labs,

you learnt how to

- Define
- Visualize
- Train
- Run a CNN using Caffe

Keep playing with it and you’ll see that it can be used to perform a wide variety of computer vision tasks!
