FREE eBook

LEARNING caffe

Free unaffiliated eBook created from
Stack Overflow contributors.

#caffe

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: caffe

It is an unofficial and free caffe ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official caffe.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with caffe

Remarks

Caffe is a library written in C++, to facilitate the experimentation with and use of Convolutional Neural Networks (CNN). Caffe has been developed by Berkeley Vision and Learning Center (BVLC).

Caffe is actually an abbreviation referring to "Convolutional Architectures for Fast Feature Extraction". This acronym encapsulates an important scope of the library. Caffe in the form of a library offers a general programming framework/architecture which can be used to perform efficient training and testing of CNNs. "Efficiency" is a major hallmark of caffe, and stands as a major design objective of Caffe.

Caffe is an open-source library released under BSD 2 Clause license.

Caffe is maintained on GitHub

Caffe can be used to :

- Efficiently train and test multiple CNN architectures, specifically any architecture that can be represented as a directed acyclic graph (DAG).
- Utilize multiple GPUs (upto 4) for training and testing. It is recommended that all the GPUs should be of the same type. Otherwise, performance is limited by the limits of the slowest GPU in the system. For example, in case of TitanX and GTX 980, the performance will be limited by the latter. Mixing multiple architectures is not supported, e.g. Kepler and Fermi 3.

Caffe has been written following efficient Object Oriented Programming (OOP) principles.

A good starting point to begin an introduction to caffe is to get a bird's eye view of how caffe works through its fundamental objects.

Versions

Version	Release Date
1.0	2017-04-19

Examples

Installation and setup

Ubuntu

Below are detailed instructions to install Caffe, pycaffe as well as its dependencies, on Ubuntu 14.04 x64 or 14.10 x64.

Execute the following script, e.g. "bash compile_caffe_ubuntu_14.sh" (~30 to 60 minutes on a new Ubuntu).

```
# This script installs Caffe and pycaffe.
# CPU only, multi-threaded Caffe.

# Usage:
# 0. Set up here how many cores you want to use during the installation:
# By default Caffe will use all these cores.
NUMBER_OF_CORES=4

sudo apt-get install -y libprotobuf-dev libleveldb-dev libsnappy-dev
sudo apt-get install -y libopencv-dev libhdf5-serial-dev
sudo apt-get install -y --no-install-recommends libboost-all-dev
sudo apt-get install -y libatlas-base-dev
sudo apt-get install -y python-dev
sudo apt-get install -y python-pip git

# For Ubuntu 14.04
sudo apt-get install -y libgflags-dev libgoogle-glog-dev liblmdb-dev protobuf-compiler

# Install LMDB
git clone https://github.com/LMDB/lmdb.git
cd lmdb/libraries/liblmdb
sudo make
sudo make install

# More pre-requisites
sudo apt-get install -y cmake unzip doxygen
sudo apt-get install -y protobuf-compiler
sudo apt-get install -y libffi-dev python-pip python-dev build-essential
sudo pip install lmdb
sudo pip install numpy
sudo apt-get install -y python-numpy
sudo apt-get install -y gfortran # required by scipy
sudo pip install scipy # required by scikit-image
sudo apt-get install -y python-scipy # in case pip failed
sudo apt-get install -y python-nose
sudo pip install scikit-image # to fix https://github.com/BVLC/caffe/issues/50


# Get caffe (http://caffe.berkeleyvision.org/installation.html#compilation)
cd
mkdir caffe
cd caffe
wget https://github.com/BVLC/caffe/archive/master.zip
unzip -o master.zip
cd caffe-master

# Prepare Python binding (pycaffe)
cd python
for req in $(cat requirements.txt); do sudo pip install $req; done

# to be able to call "import caffe" from Python after reboot:
echo "export PYTHONPATH=$(pwd):$PYTHONPATH " >> ~/.bash_profile
source ~/.bash_profile # Update shell
cd ..
```

```
# Compile caffe and pycaffe
cp Makefile.config.example Makefile.config
sed -i '8s/.*/CPU_ONLY := 1/' Makefile.config # Line 8: CPU only
sudo apt-get install -y libopenblas-dev
sed -i '33s/.*/BLAS := open/' Makefile.config # Line 33: to use OpenBLAS
# Note that if one day the Makefile.config changes and these line numbers may change
echo "export OPENBLAS_NUM_THREADS=($NUMBER_OF_CORES)" >> ~/.bash_profile
mkdir build
cd build
cmake ..
cd ..
make all -j$NUMBER_OF_CORES # 4 is the number of parallel threads for compilation: typically
equal to number of physical cores
make pycaffe -j$NUMBER_OF_CORES
make test
make runtest
#make matcaffe
make distribute

# Afew few more dependencies for pycaffe
sudo pip install pydot
sudo apt-get install -y graphviz
sudo pip install scikit-learn
```

At the end, you need to run "source ~/.bash_profile" manually or start a new shell to be able to do 'python import caffe'.

Enable multithreading with Caffe

Caffe can run on multiple cores. One way is to enable multithreading with Caffe to use OpenBLAS instead of the default ATLAS. To do so, you can follow these three steps:

1. `sudo apt-get install -y libopenblas-dev`
2. Before compiling Caffe, edit `Makefile.config`, replace `BLAS := atlas` by `BLAS := open`
3. After compiling Caffe, running `export OPENBLAS_NUM_THREADS=4` will cause Caffe to use 4 cores.

Regularization loss (weight decay) in Caffe

In the solver file, we can set a global regularization loss using the `weight_decay` and `regularization_type` options.

In many cases we want different weight decay rates for different layers. This can be done by setting the `decay_mult` option for each layer in the network definition file, where `decay_mult` is the multiplier on the global weight decay rate, so the actual weight decay rate applied for one layer is `decay_mult*weight_decay`.

For example, the following defines a convolutional layer with NO weight decay regardless of the options in the solver file.

```
layer {
  name: "Convolution1"
  type: "Convolution"
  bottom: "data"
```

```
  top: "Convolution1"
  param {
    decay_mult: 0
  }
  convolution_param {
    num_output: 32
    pad: 0
    kernel_size: 3
    stride: 1
    weight_filler {
      type: "xavier"
    }
  }
}
```

See this thread for more information.

Read Getting started with caffe online: https://riptutorial.com/caffe/topic/4382/getting-started-with-caffe

Chapter 2: Basic Caffe Objects - Solver, Net, Layer and Blob

Remarks

A caffe user sends instructions to perform specific operations to caffe objects. These objects interact with each other based on their design specifications and carry out the operation(s). This is a basic principle OOP paradigm.

While there are many caffe object types (or C++ classes), for a beginning basic understanding we focus upon 4 important caffe objects. Our objective at this stage is to simply observe the interaction between these objects on a highly abstracted level where specific implementation and design details are hazed out, and instead a bird's eye view of operation is focussed upon.

The 4 basic caffe objects are :

- **Solver**
- **Net**
- **Layer**
- **Blob**

A very basic introduction and a bird's eye view of their role in the working of caffe is presented in concise points in the examples section.

After reading and getting a basic idea of how these caffe objects interact, each object type can be read about in detail in their dedicated topics.

Examples

How these objects interact together.

- A user is looking to use caffe for CNN training and testing. The user decides upon the CNN architecture design (e.g - No. of layers, No. of filters and their details etc). The user also decides the optimization technique for training and learning parameters in case training is to be carried out. If the operation is of plain vanilla testing, a pre-trained model is specified by the user. Using all this information, the user instantiates a Solver object and provides the Solver object with an instruction (which decides operation(s) such as training and testing).

- **Solver** : This object can be looked upon as an entity that oversees the training and testing of a CNN. It is the actual contractor who gets a CNN up on processor and running. It is specialised in carrying out the specific optimizations that lead to a CNN getting trained.

- **Net** : Net can be thought of as a specialist object that represents the actual CNN over which operation(s) are carried out. Net is instructed by Solver to actually allocate memory for the CNN and instantiate it. Net is also responsible for giving instructions which actually lead to

forward or backpropagation being carried out over the CNN.

- **Layer** : It is an object that represents a particular layer of a CNN. Thus a CNN is made up of layers. As far as caffe is concerned, **Net** object instantiates each "**Layer**" type specified in the architecture definition and it also connects different layers together. A specific layer carries out a specific set of operation(s) (e.g - Max-Pooling, Min-Pooling, 2D Convolution etc.)

- **Blob** : Data flows through a CNN during training and testing. This data apart from containing user data, also includes several intermediate computations that are performed over CNN. This data is encapsulated in an object called Blob.

Read Basic Caffe Objects - Solver, Net, Layer and Blob online:
https://riptutorial.com/caffe/topic/5810/basic-caffe-objects---solver--net--layer-and-blob

Chapter 3: Batch normalization

Introduction

From the docs:

"Normalizes the input to have 0-mean and/or unit (1) variance across the batch.

This layer computes Batch Normalization as described in [1].

[...]

[1] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." arXiv preprint arXiv:1502.03167 (2015)."

Parameters

Parameter	Details
use_global_stats	From rohrbach's post from 2nd March 2016 - maybe he knows:
(use_global_stats)	"By default, during training time, the network is computing global mean/variance statistics via a running average, which is then used at test time to allow deterministic outputs for each input. You can manually toggle whether the network is accumulating or using the statistics via the use_global_stats option. IMPORTANT: for this feature to work, you MUST set the learning rate to zero for all three parameter blobs, i.e., param {lr_mult: 0} three times in the layer definition.
(use_global_stats)	This means by default (as the following is set in batch_norm_layer.cpp), you don't have to set use_global_stats at all in the prototxt. use_global_stats_ = this->phase_ == TEST;"

Examples

Prototxt for training

The following is an example definition for training a BatchNorm layer with channel-wise scale and bias. Typically a BatchNorm layer is inserted between convolution and rectification layers. In this example, the convolution would output the blob `layerx` and the rectification would receive the `layerx-bn` blob.

```
layer { bottom: 'layerx' top: 'layerx-bn' name: 'layerx-bn' type: 'BatchNorm'
  batch_norm_param {
    use_global_stats: false  # calculate the mean and variance for each mini-batch
```

```
    moving_average_fraction: .999  # doesn't effect training
  }
  param { lr_mult: 0 }
  param { lr_mult: 0 }
  param { lr_mult: 0 }}
# channel-wise scale and bias are separate
layer { bottom: 'layerx-bn' top: 'layerx-bn' name: 'layerx-bn-scale' type: 'Scale',
  scale_param {
    bias_term: true
    axis: 1      # scale separately for each channel
    num_axes: 1  # ... but not spatially (default)
    filler { type: 'constant' value: 1 }           # initialize scaling to 1
    bias_filler { type: 'constant' value: 0.001 }  # initialize bias
}}
```

More information can be found in this thread.

Prototxt for deployment

The main change needed is to switch `use_global_stats` to `true`. This switches to using the moving average.

```
layer { bottom: 'layerx' top: 'layerx-bn' name: 'layerx-bn' type: 'BatchNorm'
  batch_norm_param {
    use_global_stats: true  # use pre-calculated average and variance
  }
  param { lr_mult: 0 }
  param { lr_mult: 0 }
  param { lr_mult: 0 }}
# channel-wise scale and bias are separate
layer { bottom: 'layerx-bn' top: 'layerx-bn' name: 'layerx-bn-scale' type: 'Scale',
  scale_param {
    bias_term: true
    axis: 1      # scale separately for each channel
    num_axes: 1  # ... but not spatially (default)
}}
```

Read Batch normalization online: https://riptutorial.com/caffe/topic/6575/batch-normalization

Chapter 4: Custom Python Layers

Introduction

This tutorial will guide through the steps to create a simple custom layer for Caffe using python. By the end of it, there are some examples of custom layers. Usually you would create a custom layer to implement a functionality that isn't available in Caffe, tuning it for your requirements.

Creating a python custom layer adds some overhead to your network and probably isn't as efficient as a C++ custom layer. However, this way, you won't have to compile the whole caffe with your new layer.

Parameters

Parameter	Details
top	An array with the top blobs of your layer. Access data passed to it by using *top[i].data*, where **i** is the index of a specific blob
bottom	An array with the bottom blobs of your layer. Access data passed to it by using *bottom[i].data*, where **i** is the index of a specific blob

Remarks

- Caffe build with Python layer

Caffe needs to be compiled with `WITH_PYTHON_LAYER` option:

```
WITH_PYTHON_LAYER=1 make && make pycaffe
```

- Where should I save the class file?

You have two options (at least that I know of). Either you can save the custom layer file in the same folder as you are going to run the caffe command (probably where your prototxt files would be). Another way, also my favorite one, is to save all your custom layers in a folder and adding this folder to your PYTHONPATH.

References

1. Christopher Bourez's blog

2. Caffe Github
3. StackOverflow

Examples

Layer Template

```
import caffe

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

    def forward(self, bottom, top):
        pass

    def reshape(self, bottom, top):
        pass

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

So important things to remember:

- Your custom layer has to inherit from **caffe.Layer** (so don't forget to *import caffe*);
- You must define the four following methods: **setup**, **forward**, **reshape** and **backward**;
- All methods have a *top* and a *bottom* parameters, which are the blobs that store the input and the output passed to your layer. You can access it using *top[i].data* or *bottom[i].data*, where *i* is the index of the blob in case you have more than one upper or lower blob.

- Setup method

The Setup method is called once during the lifetime of the execution, when Caffe is instantiating all layers. This is where you will read parameters, instantiate fixed-size buffers.

- Reshape method

Use the reshape method for initialization/setup that depends on the bottom blob (layer input) size. It is called once when the network is instantiated.

- Forward method

The Forward method is called for each input batch and is where most of your logic will be.

- Backward method

The Backward method is called during the backward pass of the network. For example, in a convolution-like layer, this would be where you would calculate the gradients. This is optional (a layer can be forward-only).

Prototxt Template

Ok, so now you have your layer designed! This is how you define it in your *.prototxt* file:

```
layer {
  name: "LayerName"
  type: "Python"
  top: "TopBlobName"
  bottom: "BottomBlobName"
  python_param {
    module: "My_Custom_Layer_File"
    layer: "My_Custom_Layer_Class"
    param_str: '{"param1": 1,"param2":True, "param3":"some string"}'
  }
  include{
        phase: TRAIN
  }
}
```

Important remarks:

- **type** must be **Python**;
- You must have a **python_param** dictionary with at least the **module** and **layer** parameters;
- **module** refers to the file where you implemented your layer (without the *.py*);
- **layer** refers to the name of your class;
- You can pass parameters to the layer using **param_str** (more on accessing them bellow);
- Just like any other layer, you can define in which phase you want it to be active (see the examples to see how you can check the current phase);

Passing parameters to the layer

You can define the layer parameters in the prototxt by using *param_str*. Once you've done it, here is an example on how you access these paremeters inside the layer class:

```
def setup(self, bottom, top):
    params = eval(self.param_str)
    param1 = params["param1"]
    param2 = params.get('param2', False) #I usually use this when fetching a bool
    param3 = params["param3"]

    #Continue with the setup
    # ...
```

Measure Layer

In this example we will design a "measure" layer, that outputs the accuracy and a confusion matrix for a binary problem during training and the accuracy, false positive rate and false negative rate during test/validation. Although Caffe already has a Accuracy layer, sometimes you want something more, like a F-measure.

This is my *measureLayer.py* with my class definition:

```
#Remark: This class is designed for a binary problem, where the first class would be the
'negative'
# and the second class would be 'positive'

import caffe
TRAIN = 0
TEST = 1

class Measure_Layer(caffe.Layer):
    #Setup method
    def setup(self, bottom, top):
        #We want two bottom blobs, the labels and the predictions
        if len(bottom) != 2:
            raise Exception("Wrong number of bottom blobs (prediction and label)")

        #And some top blobs, depending on the phase
        if self.phase = TEST and len(top) != 3:
            raise Exception("Wrong number of top blobs (acc, FPR, FNR)")
        if self.phase = TRAIN and len(top) != 5:
            raise Exception("Wrong number of top blobs (acc, tp, tn, fp and fn)")

        #Initialize some attributes
        self.TPs = 0.0
        self.TNs = 0.0
        self.FPs = 0.0
        self.FNs = 0.0
        self.totalImgs = 0

    #Forward method
    def forward(self, bottom, top):
        #The order of these depends on the prototxt definition
        predictions = bottom[0].data
        labels = bottom[1].data

        self.totalImgs += len(labels)

        for i in range(len(labels)): #len(labels) is equal to the batch size
                pred = predictions[i]   #pred is a tuple with the normalized probability
                                        #of a sample i.r.t. two classes
                lab = labels[i]

                if pred[0] > pred[1]:
                        if lab == 1.0:
                                self.FNs += 1.0
                        else:
                                self.TNs += 1.0
                else:
                        if lab == 1.0:
                                self.TPs += 1.0
                        else:
                                self.FPs += 1.0

        acc = (self.TPs + self.TNs) / self.totalImgs

        try: #just assuring we don't divide by 0
                fpr = self.FPs / (self.FPs + self.TNs)
        except:
                fpr = -1.0

        try: #just assuring we don't divide by 0
                fnr = self.FNs / (self.FNs + self.TPs)
```

```
        except:
                fnr = -1.0

       #output data to top blob
       top[0].data = acc
       if self.phase == TRAIN:
           top[1].data = self.TPs
           top[2].data = self.TNs
           top[3].data = self.FPs
           top[4].data = self.FNs
       elif self.phase == TEST:
           top[1].data = fpr
           top[2].data = fnr

    def reshape(self, bottom, top):
        """
        We don't need to reshape or instantiate anything that is input-size sensitive
        """
        pass

    def backward(self, bottom, top):
        """
        This layer does not back propagate
        """
        pass
```

And this is an example of a *prototxt* with it:

```
layer {
  name: "metrics"
  type: "Python"
  top: "Acc"
  top: "TPs"
  top: "TNs"
  top: "FPs"
  top: "FNs"

  bottom: "prediction"   #let's supose we have these two bottom blobs
  bottom: "label"

  python_param {
    module: "measureLayer"
    layer: "Measure_Layer"
  }
  include {
    phase: TRAIN
  }
}

layer {
  name: "metrics"
  type: "Python"
  top: "Acc"
  top: "FPR"
  top: "FNR"

  bottom: "prediction"   #let's supose we have these two bottom blobs
  bottom: "label"

  python_param {
```

```
    module: "measureLayer"
    layer: "Measure_Layer"
  }
  include {
    phase: TEST
  }
}
```

Data Layer

This example is a custom data layer, that receives a text file with image paths, loads a batch of images and preprocesses them. Just a quick tip, Caffe already has a big range of data layers and probably a custom layer is not the most efficient way if you just want something simple.

My *dataLayer.py* could be something like:

```
import caffe

class Custom_Data_Layer(caffe.Layer):
    def setup(self, bottom, top):
        # Check top shape
        if len(top) != 2:
                raise Exception("Need to define tops (data and label)")

        #Check bottom shape
        if len(bottom) != 0:
            raise Exception("Do not define a bottom.")

        #Read parameters
        params = eval(self.param_str)
        src_file = params["src_file"]
        self.batch_size = params["batch_size"]
        self.im_shape = params["im_shape"]
        self.crop_size = params.get("crop_size", False)

        #Reshape top
        if self.crop_size:
            top[0].reshape(self.batch_size, 3, self.crop_size, self.crop_size)
        else:
            top[0].reshape(self.batch_size, 3, self.im_shape, self.im_shape)

        top[1].reshape(self.batch_size)

        #Read source file
        #I'm just assuming we have this method that reads the source file
        #and returns a list of tuples in the form of (img, label)
        self.imgTuples = readSrcFile(src_file)

        self._cur = 0 #use this to check if we need to restart the list of imgs

    def forward(self, bottom, top):
        for itt in range(self.batch_size):
            # Use the batch loader to load the next image.
            im, label = self.load_next_image()

            #Here we could preprocess the image
            # ...
```

```
            # Add directly to the top blob
            top[0].data[itt, ...] = im
            top[1].data[itt, ...] = label

    def load_next_img(self):
        #If we have finished forwarding all images, then an epoch has finished
        #and it is time to start a new one
        if self._cur == len(self.imgTuples):
            self._cur = 0
            shuffle(self.imgTuples)

        im, label = self.imgTuples[self._cur]
        self._cur += 1

        return im, label

    def reshape(self, bottom, top):
        """
        There is no need to reshape the data, since the input is of fixed size
        (img shape and batch size)
        """
        pass

    def backward(self, bottom, top):
        """
        This layer does not back propagate
        """
        pass
```

And the *prototxt* would be like:

```
layer {
  name: "Data"
  type: "Python"
  top: "data"
  top: "label"

  python_param {
    module: "dataLayer"
    layer: "Custom_Data_Layer"
    param_str: '{"batch_size": 126,"im_shape":256, "crop_size":224, "src_file":
"path_to_TRAIN_file.txt"}'
  }
}
```

Read Custom Python Layers online: https://riptutorial.com/caffe/topic/10535/custom-python-layers

Chapter 5: Prepare Data for Training

Examples

Prepare image dataset for image classification task

Caffe has a build-in input layer tailored for image classification tasks (i.e., single integer label per input image). This input `"Data"` layer is built upon an lmdb or leveldb data structure. In order to use `"Data"` layer one has to construct the data structure with all training data.

A quick guide to Caffe's `convert_imageset`

Build

First thing you must do is build caffe and caffe's tools (`convert_imageset` is one of these tools).
After installing caffe and `make`ing it make sure you ran `make tools` as well.
Verify that a binary file `convert_imageset` is created in `$CAFFE_ROOT/build/tools`.

Prepare your data

Images: put all images in a folder (I'll call it here `/path/to/jpegs/`).
Labels: create a text file (e.g., `/path/to/labels/train.txt`) with a line per input image <path/to/file> .
For example:

> img_0000.jpeg 1
> img_0001.jpeg 0
> img_0002.jpeg 0

In this example the first image is labeled `1` while the other two are labeled `0`.

Convert the dataset

Run the binary in shell

```
~$ GLOG_logtostderr=1 $CAFFE_ROOT/build/tools/convert_imageset \
    --resize_height=200 --resize_width=200 --shuffle  \
    /path/to/jpegs/ \
    /path/to/labels/train.txt \
    /path/to/lmdb/train_lmdb
```

Command line explained:

- `GLOG_logtostderr` flag is set to 1 *before* calling `convert_imageset` indicates the logging mechanism to redirect log messages to stderr.
- `--resize_height` and `--resize_width` resize **all** input images to same size `200x200`.
- `--shuffle` randomly change the order of images and does not preserve the order in the

`/path/to/labels/train.txt` file.

- Following are the path to the images folder, the labels text file and the output name. Note that the output name should not exist prior to calling `convert_imageset` otherwise you'll get a scary error message.

Other flags that might be useful:

- `--backend` - allows you to choose between an `lmdb` dataset or `levelDB`.
- `--gray` - convert all images to gray scale.
- `--encoded` and `--encoded_type` - keep image data in encoded (jpg/png) compressed form in the database.
- `--help` - shows some help, see all relevant flags under *Flags from tools/convert_imageset.cpp*

You can check out `$CAFFE_ROOT/examples/imagenet/convert_imagenet.sh` for an example how to use `convert_imageset`.

see this thread for more information.

Prepare arbitrary data in HDF5 format

In addition to image classification datasets, Caffe also have `"HDF5Data"` layer for arbitrary inputs. This layer requires all training/validation data to be stored in hdf5 format files.
This example shows how to use python `h5py` module to construct such hdf5 file and how to setup caffe `"HDF5Data"` layer to read that file.

Build the hdf5 binary file

Assuming you have a text file `'train.txt'` with each line with an image file name and a single floating point number to be used as regression target.

```
import h5py, os
import caffe
import numpy as np

SIZE = 224 # fixed size to all images
with open( 'train.txt', 'r' ) as T :
    lines = T.readlines()
# If you do not have enough memory split data into
# multiple batches and generate multiple separate h5 files
X = np.zeros( (len(lines), 3, SIZE, SIZE), dtype='f4' )
y = np.zeros( (1,len(lines)), dtype='f4' )
for i,l in enumerate(lines):
    sp = l.split(' ')
    img = caffe.io.load_image( sp[0] )
    img = caffe.io.resize( img, (SIZE, SIZE, 3) ) # resize to fixed size
    # you may apply other input transformations here...
    # Note that the transformation should take img from size-by-size-by-3 and transpose it to
3-by-size-by-size
    X[i] = img
    y[i] = float(sp[1])
with h5py.File('train.h5','w') as H:
    H.create_dataset( 'X', data=X ) # note the name X given to the dataset!
```

```
    H.create_dataset( 'y', data=y ) # note the name y given to the dataset!
with open('train_h5_list.txt','w') as L:
    L.write( 'train.h5' ) # list all h5 files you are going to use
```

Configuring `"HDF5Data"` layer

Once you have all `h5` files and the corresponding test files listing them you can add an HDF5 input layer to your `train_val.prototxt`:

```
layer {
  type: "HDF5Data"
  top: "X" # same name as given in create_dataset!
  top: "y"
  hdf5_data_param {
    source: "train_h5_list.txt" # do not give the h5 files directly, but the list.
    batch_size: 32
  }
  include { phase:TRAIN }
}
```

You can find more information here and here.

As shown in above, we pass into Caffe a list of HDF5 files. That is because in the current version there's a size limit of 2GB for a single HDF5 data file. So if the training data exceeds 2GB, we'll need to split it into separate files.

If a single HDF5 data file exceeds 2GB we'll get an error message like

```
Check failed: shape[i] <= 2147483647 / count_ (100 vs. 71) blob size exceeds INT_MAX
```

If the total amount of data is less than 2GB, shall we split the data into separate files or not?

According to a piece of comment in Caffe's source code, a single file would be better,

> If shuffle == true, the ordering of the HDF5 files is shuffled, and the ordering of data within any given HDF5 file is shuffled, but data between different files are not interleaved.

Read Prepare Data for Training online: https://riptutorial.com/caffe/topic/5344/prepare-data-for-training

Chapter 6: Training a Caffe model with pycaffe

Examples

Training a network on the Iris dataset

Given below is a simple example to train a Caffe model on the Iris data set in Python, using PyCaffe. It also gives the predicted outputs given some user-defined inputs.

`iris_tuto.py`

```
import subprocess
import platform
import copy

from sklearn.datasets import load_iris
import sklearn.metrics
import numpy as np
from sklearn.cross_validation import StratifiedShuffleSplit
import matplotlib.pyplot as plt
import h5py
import caffe
import caffe.draw


def load_data():
    '''
    Load Iris Data set
    '''
    data = load_iris()
    print(data.data)
    print(data.target)
    targets = np.zeros((len(data.target), 3))
    for count, target in enumerate(data.target):
        targets[count][target]= 1
    print(targets)

    new_data = {}
    #new_data['input'] = data.data
    new_data['input'] = np.reshape(data.data, (150,1,1,4))
    new_data['output'] = targets
    #print(new_data['input'].shape)
    #new_data['input'] = np.random.random((150, 1, 1, 4))
    #print(new_data['input'].shape)
    #new_data['output'] = np.random.random_integers(0, 1, size=(150,3))
    #print(new_data['input'])

    return new_data

def save_data_as_hdf5(hdf5_data_filename, data):
    '''
    HDF5 is one of the data formats Caffe accepts
    '''
```

```
    with h5py.File(hdf5_data_filename, 'w') as f:
        f['data'] = data['input'].astype(np.float32)
        f['label'] = data['output'].astype(np.float32)


def train(solver_prototxt_filename):
    '''
    Train the ANN
    '''
    caffe.set_mode_cpu()
    solver = caffe.get_solver(solver_prototxt_filename)
    solver.solve()


def print_network_parameters(net):
    '''
    Print the parameters of the network
    '''
    print(net)
    print('net.inputs: {0}'.format(net.inputs))
    print('net.outputs: {0}'.format(net.outputs))
    print('net.blobs: {0}'.format(net.blobs))
    print('net.params: {0}'.format(net.params))

def get_predicted_output(deploy_prototxt_filename, caffemodel_filename, input, net = None):
    '''
    Get the predicted output, i.e. perform a forward pass
    '''
    if net is None:
        net = caffe.Net(deploy_prototxt_filename,caffemodel_filename, caffe.TEST)

    #input = np.array([[ 5.1,  3.5,  1.4,  0.2]])
    #input = np.random.random((1, 1, 1))
    #print(input)
    #print(input.shape)
    out = net.forward(data=input)
    #print('out: {0}'.format(out))
    return out[net.outputs[0]]


import google.protobuf
def print_network(prototxt_filename, caffemodel_filename):
    '''
    Draw the ANN architecture
    '''
    _net = caffe.proto.caffe_pb2.NetParameter()
    f = open(prototxt_filename)
    google.protobuf.text_format.Merge(f.read(), _net)
    caffe.draw.draw_net_to_file(_net, prototxt_filename + '.png' )
    print('Draw ANN done!')


def print_network_weights(prototxt_filename, caffemodel_filename):
    '''
    For each ANN layer, print weight heatmap and weight histogram
    '''
    net = caffe.Net(prototxt_filename,caffemodel_filename, caffe.TEST)
    for layer_name in net.params:
        # weights heatmap
        arr = net.params[layer_name][0].data
        plt.clf()
```

```
        fig = plt.figure(figsize=(10,10))
        ax = fig.add_subplot(111)
        cax = ax.matshow(arr, interpolation='none')
        fig.colorbar(cax, orientation="horizontal")
        plt.savefig('{0}_weights_{1}.png'.format(caffemodel_filename, layer_name), dpi=100,
format='png', bbox_inches='tight') # use format='svg' or 'pdf' for vectorial pictures
        plt.close()

        # weights histogram
        plt.clf()
        plt.hist(arr.tolist(), bins=20)
        plt.savefig('{0}_weights_hist_{1}.png'.format(caffemodel_filename, layer_name),
dpi=100, format='png', bbox_inches='tight') # use format='svg' or 'pdf' for vectorial pictures
        plt.close()


def get_predicted_outputs(deploy_prototxt_filename, caffemodel_filename, inputs):
    '''
    Get several predicted outputs
    '''
    outputs = []
    net = caffe.Net(deploy_prototxt_filename,caffemodel_filename, caffe.TEST)
    for input in inputs:
        #print(input)
        outputs.append(copy.deepcopy(get_predicted_output(deploy_prototxt_filename,
caffemodel_filename, input, net)))
    return outputs


def get_accuracy(true_outputs, predicted_outputs):
    '''

    '''
    number_of_samples = true_outputs.shape[0]
    number_of_outputs = true_outputs.shape[1]
    threshold = 0.0 # 0 if SigmoidCrossEntropyLoss ; 0.5 if EuclideanLoss
    for output_number in range(number_of_outputs):
        predicted_output_binary = []
        for sample_number in range(number_of_samples):
            #print(predicted_outputs)
            #print(predicted_outputs[sample_number][output_number])
            if predicted_outputs[sample_number][0][output_number] < threshold:
                predicted_output = 0
            else:
                predicted_output = 1
            predicted_output_binary.append(predicted_output)

        print('accuracy: {0}'.format(sklearn.metrics.accuracy_score(true_outputs[:,
output_number], predicted_output_binary)))
        print(sklearn.metrics.confusion_matrix(true_outputs[:, output_number],
predicted_output_binary))


def main():
    '''
    This is the main function
    '''

    # Set parameters
    solver_prototxt_filename = 'iris_solver.prototxt'
    train_test_prototxt_filename = 'iris_train_test.prototxt'
```

```
    deploy_prototxt_filename  = 'iris_deploy.prototxt'
    deploy_prototxt_filename  = 'iris_deploy.prototxt'
    deploy_prototxt_batch2_filename  = 'iris_deploy_batchsize2.prototxt'
    hdf5_train_data_filename = 'iris_train_data.hdf5'
    hdf5_test_data_filename = 'iris_test_data.hdf5'
    caffemodel_filename = 'iris__iter_5000.caffemodel' # generated by train()

    # Prepare data
    data = load_data()
    print(data)
    train_data = data
    test_data = data
    save_data_as_hdf5(hdf5_train_data_filename, data)
    save_data_as_hdf5(hdf5_test_data_filename, data)

    # Train network
    train(solver_prototxt_filename)

    # Print network
    print_network(deploy_prototxt_filename, caffemodel_filename)
    print_network(train_test_prototxt_filename, caffemodel_filename)
    print_network_weights(train_test_prototxt_filename, caffemodel_filename)

    # Compute performance metrics
    #inputs = input = np.array([[[[ 5.1,  3.5,  1.4,  0.2]]],[[[ 5.9,  3. ,  5.1,  1.8]]]])
    inputs = data['input']
    outputs = get_predicted_outputs(deploy_prototxt_filename, caffemodel_filename, inputs)
    get_accuracy(data['output'], outputs)


if __name__ == "__main__":
    main()
```

It requires the two following `iris_train_test.prototxt` and `iris_deploy.prototxt` to be in the same folder.

`iris_train_test.prototxt`:

```
name: "IrisNet"
layer {
  name: "iris"
  type: "HDF5Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  hdf5_data_param {
    source: "iris_train_data.txt"
    batch_size: 1

  }
}

layer {
  name: "iris"
  type: "HDF5Data"
  top: "data"
  top: "label"
```

```
  include {
    phase: TEST
  }
  hdf5_data_param {
    source: "iris_test_data.txt"
    batch_size: 1

  }
}




layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "data"
  top: "ip1"
  param {
    lr_mult: 1  # the learning rate multiplier for weights
  }
  param {
    lr_mult: 2  # the learning rate multiplier for biases
  }
  inner_product_param {
    num_output: 50
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
layer {
  name: "drop1"
  type: "Dropout"
  bottom: "ip1"
  top: "ip1"
  dropout_param {
    dropout_ratio: 0.5
  }
}


layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
```

```
  }
  inner_product_param {
    num_output: 50
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "drop2"
  type: "Dropout"
  bottom: "ip2"
  top: "ip2"
  dropout_param {
    dropout_ratio: 0.4
  }
}



layer {
  name: "ip3"
  type: "InnerProduct"
  bottom: "ip2"
  top: "ip3"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 3
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
  name: "drop3"
  type: "Dropout"
  bottom: "ip3"
  top: "ip3"
  dropout_param {
    dropout_ratio: 0.3
  }
}

layer {
  name: "loss"
  type: "SigmoidCrossEntropyLoss"
  # type: "EuclideanLoss"
  # type: "HingeLoss"
  bottom: "ip3"
```

```
  bottom: "label"
  top: "loss"
}
```

`iris_deploy.prototxt`:

```
name: "IrisNet"
input: "data"
input_dim: 1 # batch size
input_dim: 1
input_dim: 1
input_dim: 4


layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "data"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 50
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "ip1"
  top: "ip1"
}
layer {
  name: "drop1"
  type: "Dropout"
  bottom: "ip1"
  top: "ip1"
  dropout_param {
    dropout_ratio: 0.5
  }
}


layer {
  name: "ip2"
  type: "InnerProduct"
  bottom: "ip1"
  top: "ip2"
  param {
    lr_mult: 1
  }
```

```
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 50
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
layer {
  name: "drop2"
  type: "Dropout"
  bottom: "ip2"
  top: "ip2"
  dropout_param {
    dropout_ratio: 0.4
  }
}


layer {
  name: "ip3"
  type: "InnerProduct"
  bottom: "ip2"
  top: "ip3"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 3
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}

layer {
  name: "drop3"
  type: "Dropout"
  bottom: "ip3"
  top: "ip3"
  dropout_param {
    dropout_ratio: 0.3
  }
}
```

`iris_solver.prototxt`:

```
# The train/test net protocol buffer definition
net: "iris_train_test.prototxt"
```

```
# test_iter specifies how many forward passes the test should carry out.
test_iter: 1
# Carry out testing every test_interval training iterations.
test_interval: 1000
# The base learning rate, momentum and the weight decay of the network.
base_lr: 0.0001
momentum: 0.001
weight_decay: 0.0005
# The learning rate policy
lr_policy: "inv"
gamma: 0.0001
power: 0.75
# Display every 100 iterations
display: 1000
# The maximum number of iterations
max_iter: 5000
# snapshot intermediate results
snapshot: 5000
snapshot_prefix: "iris_"
# solver mode: CPU or GPU
solver_mode: CPU # GPU
```

Read Training a Caffe model with pycaffe online: https://riptutorial.com/caffe/topic/4618/training-a-caffe-model-with-pycaffe

Credits

S. No	Chapters	Contributors
1	Getting started with caffe	Community, dontloo, Franck Dernoncourt, GoodDeeds, Parag S. Chandakkar, Shai, Tahir Shahzad, Ujjwal Aryan
2	Basic Caffe Objects - Solver, Net, Layer and Blob	Ujjwal Aryan
3	Batch normalization	dasWesen, Jonathan, Shai
4	Custom Python Layers	Fernanda Andalo, rafaspadilha
5	Prepare Data for Training	dontloo, malreddysid, Shai, Stephen Leppik
6	Training a Caffe model with pycaffe	Franck Dernoncourt, Parag S. Chandakkar