2017

CAFFE MODEL TRAINING

NRUPESH PATEL
nrupesh.patel@sjsu.edu
SJSU ID: 011425271

SAN JOSE STATE UNIVERSITY

REQUIREMENT

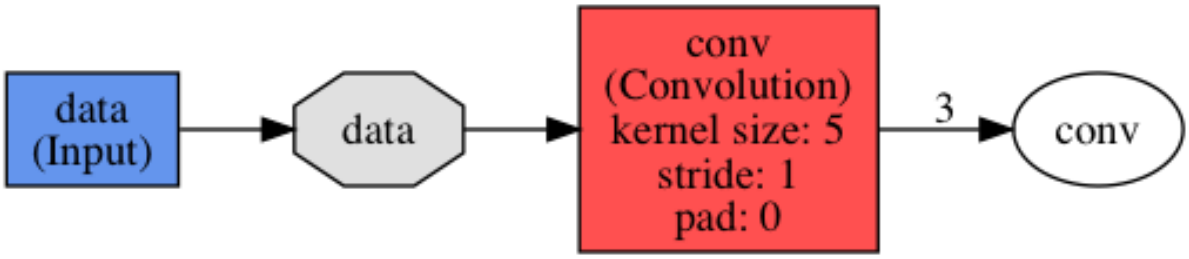
Training a Caffe model was necessary for following reasons:

- No specific model trained for classifying or detecting garbage.
- Model used in current solution is not open-source.

NETWORK MODEL

First let's try simple single-layer network to showcase editing model parameters.

Let's create first a very simple model with a single convolution composed of 3 convolutional neurons, with kernel of size 5x5 and stride of 1:

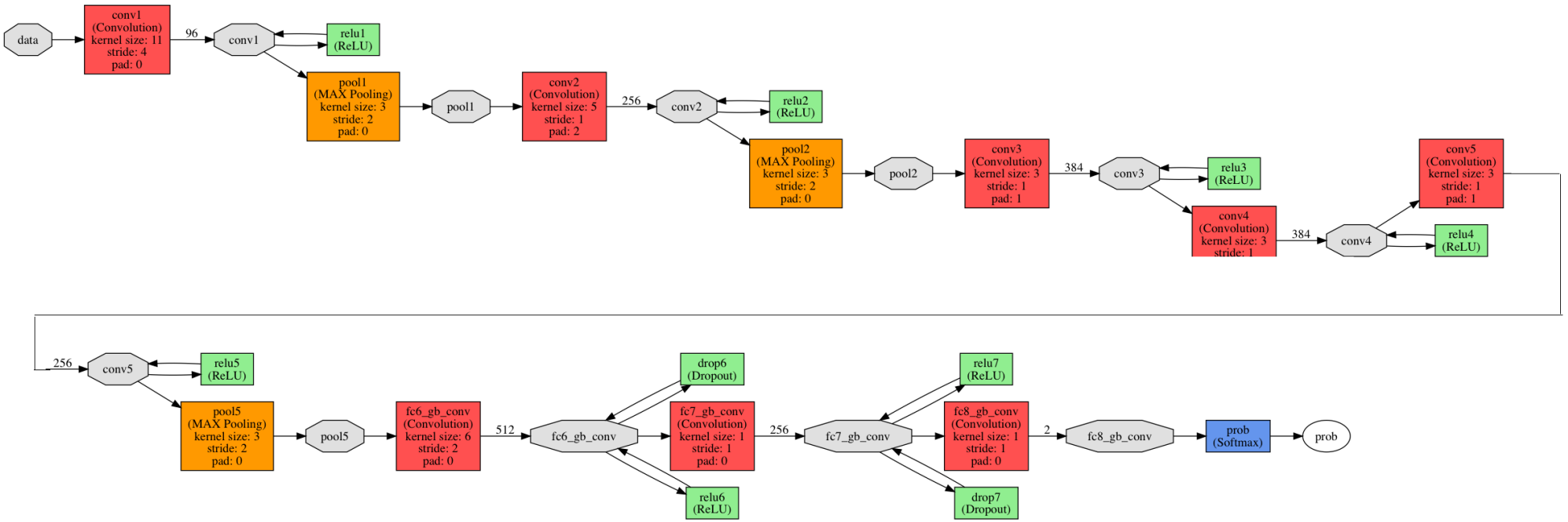


This net will produce 3 output maps from an input map.

CODE (Conv.prototxt)

```
name: "convolution"
layer {
  name: "data"
  type: "Input"
  top: "data"
  input_param { shape: { dim: 1 dim: 1 dim: 100 dim: 100 } }
}
layer {
  name: "conv"
  type: "Convolution"
  bottom: "data"
  top: "conv"
  convolution_param {
    num_output: 3
    kernel_size: 5
    stride: 1
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
```

This is the Network architecture for the model to be trained:



CODE (train_val.prototxt)

```
name: "GarbnetFCN"
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  data_param {
    source: "<path_to_train_lmdb_file>"
    batch_size: 256
    backend: LMDB
  }
}
layer {
  name: "data"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TEST
  }
  data_param {
    source: ""<path_to_test_lmdb_file>"
    batch_size: 50
    backend: LMDB
  }
}
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 96
    kernel_size: 11
    stride: 4
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
```

```
      value: 0
    }
  }
}
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
layer {
  name: "norm1"
  type: "LRN"
  bottom: "conv1"
  top: "norm1"
  lrn_param {
    local_size: 5
    alpha: 0.0001
    beta: 0.75
  }
}
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "norm1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
  }
}
layer {
  name: "conv2"
  type: "Convolution"
  bottom: "pool1"
  top: "conv2"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 256
    pad: 2
    kernel_size: 5
    group: 2
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
```

```
    bias_filler {
      type: "constant"
      value: 0.1
    }
  }
}
layer {
  name: "relu2"
  type: "ReLU"
  bottom: "conv2"
  top: "conv2"
}
layer {
  name: "norm2"
  type: "LRN"
  bottom: "conv2"
  top: "norm2"
  lrn_param {
    local_size: 5
    alpha: 0.0001
    beta: 0.75
  }
}
layer {
  name: "pool2"
  type: "Pooling"
  bottom: "norm2"
  top: "pool2"
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
  }
}
layer {
  name: "conv3"
  type: "Convolution"
  bottom: "pool2"
  top: "conv3"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 384
    pad: 1
    kernel_size: 3
    weight_filler {
      type: "gaussian"
      std: 0.01
```

```
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
layer {
  name: "relu3"
  type: "ReLU"
  bottom: "conv3"
  top: "conv3"
}
layer {
  name: "conv4"
  type: "Convolution"
  bottom: "conv3"
  top: "conv4"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 384
    pad: 1
    kernel_size: 3
    group: 2
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0.1
    }
  }
}
layer {
  name: "relu4"
  type: "ReLU"
  bottom: "conv4"
  top: "conv4"
}
layer {
  name: "conv5"
  type: "Convolution"
  bottom: "conv4"
  top: "conv5"
  param {
    lr_mult: 1
```

```
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  convolution_param {
    num_output: 256
    pad: 1
    kernel_size: 3
    group: 2
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0.1
    }
  }
}
layer {
  name: "relu5"
  type: "ReLU"
  bottom: "conv5"
  top: "conv5"
}
layer {
  name: "pool5"
  type: "Pooling"
  bottom: "conv5"
  top: "pool5"
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
  }
}
layer {
  name: "fc6"
  type: "InnerProduct"
  bottom: "pool5"
  top: "fc6"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  inner_product_param {
    num_output: 4096
    weight_filler {
```

```
      type: "gaussian"
      std: 0.005
    }
    bias_filler {
      type: "constant"
      value: 0.1
    }
  }
}
layer {
  name: "relu6"
  type: "ReLU"
  bottom: "fc6"
  top: "fc6"
}
layer {
  name: "drop6"
  type: "Dropout"
  bottom: "fc6"
  top: "fc6"
  dropout_param {
    dropout_ratio: 0.5
  }
}
layer {
  name: "fc7"
  type: "InnerProduct"
  bottom: "fc6"
  top: "fc7"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  inner_product_param {
    num_output: 4096
    weight_filler {
      type: "gaussian"
      std: 0.005
    }
    bias_filler {
      type: "constant"
      value: 0.1
    }
  }
}
layer {
  name: "relu7"
  type: "ReLU"
  bottom: "fc7"
  top: "fc7"
```

```
}
layer {
  name: "drop7"
  type: "Dropout"
  bottom: "fc7"
  top: "fc7"
  dropout_param {
    dropout_ratio: 0.5
  }
}
layer {
  name: "fc8"
  type: "InnerProduct"
  bottom: "fc7"
  top: "fc8"
  param {
    lr_mult: 1
    decay_mult: 1
  }
  param {
    lr_mult: 2
    decay_mult: 0
  }
  inner_product_param {
    num_output: 1000
    weight_filler {
      type: "gaussian"
      std: 0.01
    }
    bias_filler {
      type: "constant"
      value: 0
    }
  }
}
layer {
  name: "accuracy"
  type: "Accuracy"
  bottom: "fc8"
  bottom: "label"
  top: "accuracy"
  include {
    phase: TEST
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "fc8"
  bottom: "label"
  top: "loss"
}
```

TRAINING

Training requires:

- `train_val.prototxt`: defines the network architecture, initialization parameters, and local learning rates
- `solver.prototxt`: defines optimization/training parameters and serves as the actual file that's called to train a deep network

CODE (solver.prototxt)

```
net: "<path_to_train_val.prototxt>"
test_iter: 1000
test_interval: 1000
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 100000
display: 20
max_iter: 450000
momentum: 0.9
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix: "<path_to_store_snapshot>"
solver_mode: GPU
```

Training Command:

```
./build/tools/caffe train -solver solver.prototxt
```

Training will produce two files:

- *<filename>.caffemodel*: weights of the architecture to be used in testing
- *<filename>.solverstate*: used if training dies (e.g., power outage) to resume training from current iteration

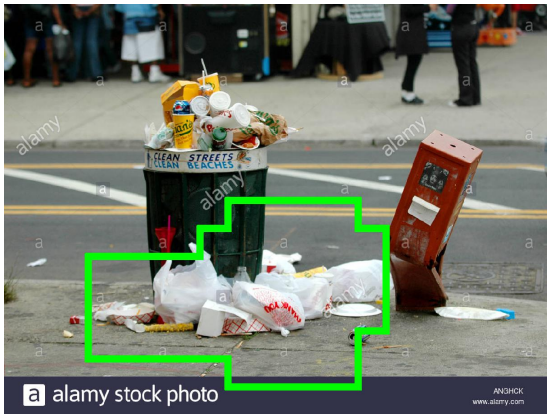
TESTING

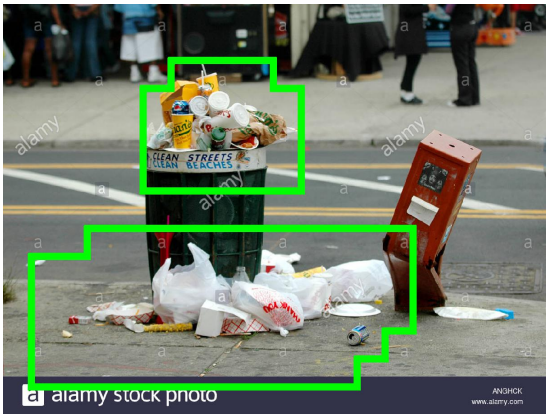
CODE (Python Code to Test)

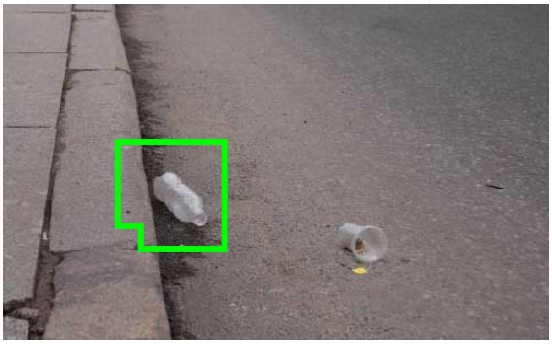
```
import caffe
import os
from pylab import *
import sys
from PIL import Image
import numpy as np
import cv2

def gatherImages(folder,imageNames=None):
    images = []
    names = []
    files = os.listdir(folder)
    total = len(files)
    print 'Total %d images in folder %s' % (total,folder)
    for i in os.listdir(folder):
        try:
            if imageNames is None or i in imageNames:
                example_image = folder+'/'+i
                input_image = Image.open(example_image)
                images.append(input_image)
                names.append(i)
        except:
            pass

    return images,names

def resizeForFCN(image,size):
    w,h = image.size
    if w<h:
        return image.resize((int(227*size),int((227*h*size)/w)))
#227x227 is input for regular CNN
    else:
        return image.resize((int((227*w*size)/h),int(227*size)))

def getSegmentedImage(test_image, probMap,thresh):
    kernel = np.ones((6,6),np.uint8)
    wt,ht = test_image.size
    out_bn = np.zeros((ht,wt),dtype=uint8)

    for h in range(probMap.shape[0]):
                for k in range(probMap.shape[1]):
                    if probMap[h,k] > thresh:
                        x1 = h*62 #stride 2 at fc6_gb_conv
equivalent to 62 pixels stride in input
                        y1 = k*62
                        for hoff in range(x1,227+x1):
                            if hoff < out_bn.shape[0]:
                                for koff in range(y1,227+y1):
```

```
                                    if koff < out_bn.shape[1]:
                                        out_bn[hoff,koff] = 255
    edge = cv2.Canny(out_bn,200,250)
    box = cv2.dilate(edge,kernel,iterations = 3)

    or_im_ar = np.array(test_image)
    or_im_ar[:,:,1] = (or_im_ar[:,:,1] | box)
    or_im_ar[:,:,2] = or_im_ar[:,:,2] * box + or_im_ar[:,:,2]
    or_im_ar[:,:,0] = or_im_ar[:,:,0] * box + or_im_ar[:,:,0]

    return Image.fromarray(or_im_ar)


def getPredictionsFor(images,names,size,thresh,output_folder):
    for i in range(len(images)):
        try:
            test_image = resizeForFCN(images[i],size)

            in_ = np.array(test_image,dtype = np.float32)
            in_ = in_[:,:,::-1]
            in_ -= np.array(mean.mean(1).mean(1))
            in_ = in_.transpose((2,0,1))

            net.blobs['data'].reshape(1,*in_.shape)
            net.blobs['data'].data[...] = in_
            net.forward()

            probMap =net.blobs['prob'].data[0,1]
            print names[i]+'...',
            if len(np.where(probMap>thresh)[0]) > 0:
                print 'Garbage!'
            else:
                print 'Not Garbage!'

            out_ = getSegmentedImage(test_image, probMap,thresh)
            out_.save(output_folder + '/output_' + names[i])
        except:
            pass


mean_filename='<path_to_mean_file>(Optional)'
deploy_filename = '<path_to_deploy.prototxt_file>'
caffemodel_file = '<path_to_caffemodel_file>'

proto_data = open(mean_filename, "rb").read()
a = caffe.io.caffe_pb2.BlobProto.FromString(proto_data)
mean  = caffe.io.blobproto_to_array(a)[0]

net = caffe.Net(deploy_filename,caffemodel_file,caffe.TEST)

#specify 'input' folder containing images for prediction
images,names = gatherImages('input')
#specify 'output' folder to store segmented predictions
getPredictionsFor(images,names,4,0.999,'output')
```

Following are the results when compared the existing solution and newly trained model:








wer East Side


wer East Side

FINE-TUNING

Task for next week is to Fine-Tune the model for more accurate results.

References

[1] Mittal, Gaurav, et al. "SpotGarbage: smartphone app to detect garbage using deep learning." *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 2016.

[2] https://medium.com/@alexrachnog/using-caffe-with-your-own-dataset-b0ade5d71233#.m03nhhn5b

[3] https://github.com/christopher5106/FastAnnotationTool

[4] http://www.pyimagesearch.com/2016/11/28/macos-install-opencv-3-and-python-2-7/

[5] https://www.quora.com/How-do-I-fine-tune-a-Caffe-pre-trained-model-to-do-image-classification-on-my-own-dataset

[6] https://frankzliu.com/experimenting-with-different-penultimate-layers-in-caffe/

[7] http://rodriguezandres.github.io/2016/04/28/caffe/#dataset-preparation