Fine tuning a VGG-16

Motivation

- There are ''famous'' network architectures like AlexNet, GoogLeNet, NetworkInNetwork, VGG, Inception, ResNet, FCN8, SegNet etc.
- because they are top performing in some task on some benchmark dataset like classification of ImageNet, ILSVRC-2012, semantic segmentation of Cityscapes etc.
- Authors have made them publicly available in the framework they used to develop (Keras, Tensorflow, Theano, Lassagne, Caffe... many!)
- What's available is not only the architecture and the source code creating it, but also the weights (layers parameters) after a long and careful training with a possibly huge dataset
- There's even something called a *zoo* : a repository of models and their weights, for instance in Caffe (go here (https://github.com/BVLC/caffe/wiki/Model-Zoo))
- Maybe someone has managed to translate one such model from one framework to yours (Tensorflow), or if not, there are tools to do it like Caffe-to-Tensor (https://github.com/ethereon/caffe-tensorflow)

What's fine tuning a network

You'll like to reuse one such model for

- the **same type of task**, say, image classification, **but**
- on a **different type of images** : instead of ImageNet = 1.4M images of 1000 classes of objects, a dataset of images of cars where classes are make + model

Instead of start training the model from scratch, since the task is the same you want to perform **domain transfer** : adapt some of the net parameters to the new images.

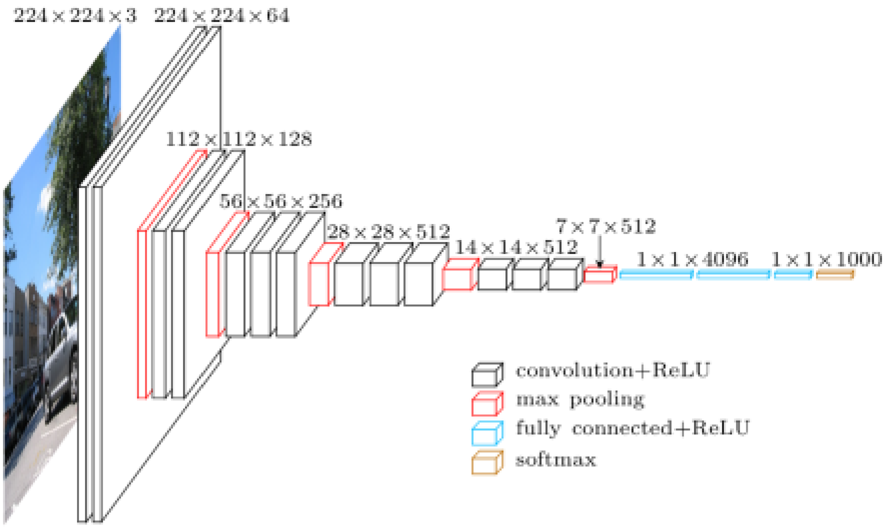
Why ?

- the original net is large $\Rightarrow$ hard to train
- already performs well on another difficult dataset
- filters of lower layers are probably ok
- so we only want to adapt the upper conv layers or just the classifier layers

Fine tuning is taking the parameters of one network as the starting point and retrain (usually just some layers) with new images of a different domain.

VGG-16

VGG achieves 92.7% top-5 test accuracy in ImageNet. We'll retrain VGG-16 to classify cars.



Here (https://github.com/machrisaa/tensorflow-vgg) there's a Python class Vgg16 that instantiates the VGG-16 model and loads all the weights.

We'll change it to

- accept 112 × 112 × 3 images instead of 224 × 224 × 3
- just load the first 5 convolutional layers
- and not the two fully-connected ones that perform the classification (7•7•512 = 25088 to 4096 + 4096 to 1000 classes = 120M params.)
- replace the fully connected layers by 2 smaller fully connected layers (4•4•512 to 512 + 512 to 41 classes = 4M params.)
- train just these 2 new layers

In [ ]:

```
import os
import sys

import numpy as np
import tensorflow as tf

VGG_MEAN = [103.939, 116.779, 123.68]


class My_Vgg16:
    def __init__(self, vgg16_npy_path=None):
        if vgg16_npy_path is None:
            path = sys.modules[self.__class__.__module__].__file__
            # print path
            path = os.path.abspath(os.path.join(path, os.pardir))
            # print path
            path = os.path.join(path, "vgg16.npy")
            print(path)
            vgg16_npy_path = path

        self.data_dict = np.load(vgg16_npy_path, encoding='latin1').item()
        print("npy file loaded")


    def _max_pool(self, bottom, name):
        return tf.nn.max_pool(bottom, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
                              padding='SAME', name=name)


    def _conv_layer(self, bottom, name):
        with tf.variable_scope(name) as scope:
            filt = self.get_conv_filter(name)
            conv = tf.nn.conv2d(bottom, filt, [1, 1, 1, 1], padding='SAME')

            conv_biases = self.get_bias(name)
            bias = tf.nn.bias_add(conv, conv_biases)

            relu = tf.nn.relu(bias)
            return relu


    def get_conv_filter(self, name):
        return tf.Variable(self.data_dict[name][0], name="filter")
        # use tf.constant() to prevent retraining it accidentally


    def get_bias(self, name):
        return tf.Variable(self.data_dict[name][1], name="biases")
        # use tf.constant() to prevent retraining it accidentally


    def build(self, rgb, train=False):
        rgb_scaled = rgb * 255.0

        # Convert RGB to BGR
        red, green, blue = tf.split(3, 3, rgb_scaled)
        assert red.get_shape().as_list()[1:] == [112, 112, 1]
        assert green.get_shape().as_list()[1:] == [112, 112, 1]
        assert blue.get_shape().as_list()[1:] == [112, 112, 1]
```

```
        bgr = tf.concat(3, [
            blue - VGG_MEAN[0],
            green - VGG_MEAN[1],
            red - VGG_MEAN[2],
        ])
        assert bgr.get_shape().as_list()[1:] == [112, 112, 3]

        self.conv1_1 = self._conv_layer(bgr, "conv1_1")
        self.conv1_2 = self._conv_layer(self.conv1_1, "conv1_2")
        self.pool1 = self._max_pool(self.conv1_2, 'pool1')

        self.conv2_1 = self._conv_layer(self.pool1, "conv2_1")
        self.conv2_2 = self._conv_layer(self.conv2_1, "conv2_2")
        self.pool2 = self._max_pool(self.conv2_2, 'pool2')

        self.conv3_1 = self._conv_layer(self.pool2, "conv3_1")
        self.conv3_2 = self._conv_layer(self.conv3_1, "conv3_2")
        self.conv3_3 = self._conv_layer(self.conv3_2, "conv3_3")
        self.pool3 = self._max_pool(self.conv3_3, 'pool3')

        self.conv4_1 = self._conv_layer(self.pool3, "conv4_1")
        self.conv4_2 = self._conv_layer(self.conv4_1, "conv4_2")
        self.conv4_3 = self._conv_layer(self.conv4_2, "conv4_3")
        self.pool4 = self._max_pool(self.conv4_3, 'pool4')

        self.conv5_1 = self._conv_layer(self.pool4, "conv5_1")
        self.conv5_2 = self._conv_layer(self.conv5_1, "conv5_2")
        self.conv5_3 = self._conv_layer(self.conv5_2, "conv5_3")
        self.pool5 = self._max_pool(self.conv5_3, 'pool5')

        shape_pool5 = self.pool5.get_shape().as_list()[1:]
        assert 'pool5', shape_pool5 == [4, 4, 512]

        """
        The new fully connected layers
        """
        dim1 = np.prod(shape_pool5) # 4*4*512 = 8192
        dim2 = 512
        n_output = 41 # number of classes = car makes
        fc_weights  = {
            'wd1': tf.Variable(tf.random_normal([dim1, dim2], stddev=0.1), name=
'wd1'),
            'wd2': tf.Variable(tf.random_normal([dim2, n_output], stddev=0.1), n
ame='wd2')
        }
        fc_biases   = {
            'bd1': tf.Variable(tf.random_normal([dim2], stddev=0.1), name='bd1'
),
            'bd2': tf.Variable(tf.random_normal([n_output], stddev=0.1), name='b
d2')
        }

        x = tf.reshape(self.pool5, [-1, dim1], name="flat_pool5")
        self.fc6 = tf.nn.bias_add(tf.matmul(x, fc_weights['wd1']), fc_biases['bd
1'], name="fc6")
        self.relu6 = tf.nn.relu(self.fc6)
        if train:
            self.relu6 = tf.nn.dropout(self.relu6, 0.75)

        # the two outputs, logits is for training, probs for testing
        self.logits = tf.nn.bias_add(tf.matmul(self.relu6, fc_weights['wd2']), f
```

```
c_biases['bd2'], name="logits")
        self.probs = tf.nn.softmax(self.logits, name="probs")
```

The dataset

It's the same one we used in the Feeding and Queues section.

In [ ]:

```
from feeding_and_queues.dataset import Dataset
```

Run fine tuning

It seems that it's better to reduce the original learning rate, something like

- 1/10th for top layers
- 1/100 for intermediate layers

In [ ]:

```
import numpy as np
import time
import platform
import matplotlib.pyplot as plt

from skimage.io import imread

import tensorflow as tf
from fine_tuning_vgg.my_vgg16 import My_Vgg16
from feeding_and_queues.dataset import Dataset


tf.reset_default_graph()

xs = tf.placeholder(tf.float32, [None, 112, 112, 3])
ys = tf.placeholder(tf.float32, [None])
# the VGG-16 weights are here: https://dl.dropboxusercontent.com/u/50333326/vgg1
6.npy
# Based on this implementation https://github.com/ry/tensorflow-vgg16
if platform.node().upper()=='CVC220':
    vgg16_npy_path = '/home/joans/Documents/recerca/tensorflow/implementacions/t
ensorflow_vgg_master/vgg16.npy'
elif platform.node().upper()=='CVC180':
    vgg16_npy_path = '/home/joans/Documents/classe_tf/notebooks/fine_tuning_vgg/
vgg16.npy'
else:
    raise Exception('Not ready to run on computer '+platform.node())

vgg = My_Vgg16(vgg16_npy_path=vgg16_npy_path)
with tf.name_scope("my_vgg16"):
    vgg.build(xs)


print '\ntrainable variables'
for v in tf.trainable_variables():
    print v.name, v.get_shape().as_list()

"""
We want to train only the weights and biases of the two
fully connected layers.
"""
vars_to_optimize = [v for v in tf.trainable_variables() \
    if v.name.startswith('my_vgg16/wd') \
       or v.name.startswith('my_vgg16/bd')]

print '\nvariables to optimize'
for v in vars_to_optimize:
    print v.name, v.get_shape().as_list()

learning_rate = 0.001/10.
loss = tf.reduce_mean(
    tf.nn.sparse_softmax_cross_entropy_with_logits(vgg.logits, tf.to_int64(ys)))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
"""
minimize with list of variables to update
"""
train_op = optimizer.minimize(loss, var_list=vars_to_optimize)

corr = tf.equal(tf.argmax(vgg.probs, 1),  tf.to_int64(ys)) # count corrects
```

```
accr = tf.reduce_mean(tf.cast(corr, tf.float32)) # accuracy
init = tf.global_variables_initializer()

ds = Dataset()
ds.new_height, ds.new_width = (112, 112)
batch_size = 10
num_epochs = 5

with tf.Session() as sess:
    sess.run(init)
    ds.epochs_completed
    while ds.epochs_completed < num_epochs:
        ds.epochs_completed
        batch_xs, batch_ys = ds.next_batch(batch_size)
        # compute average loss and accuracy for each batch
        _, batch_cost, batch_acc = sess.run([train_op, loss, accr],
                                            feed_dict={xs: batch_xs, ys: batch_y
s})
        print("cost: %.9f, acc %.9f" % (batch_cost, batch_acc))
```