Deep Learning for Plant Phenotyping

Michael Pound,Andrew French

Abstract

Deep learning is an emerging field that promises unparalleled results on many data analysis problems. We show the success offered by such techniques when applied to the challenging problem of image-based plant phenotyping, and demonstrate state-of-the-art results for root and shoot feature identification and localisation. We use fully automated trait identification using deep learning to identify quantitative trait loci in root architecture datasets. The majority of (12 out of 14) manuallyidentified QTL were also discovered using our automated approach based on the deep learning detection to locate plant features. We predict a paradigm shift in image-based phenotyping bought
about by deep learning approaches.

Citation: Michael Pound,Andrew French Deep Learning for Plant Phenotyping. **protocols.io**
https://www.protocols.io/view/deep-learning-for-plant-phenotyping-jcncive
Published: 11 Aug 2017

Protocol

Install Prerequisite Libraries

Step 1.

This protocol requires the caffe deep learning library, along with python. Detailed instructions exist on the Caffe website. Once installed, the following should produce no errors (using the root CNN architecture as an example):

```
import caffe
import lmdb
import numpy as np

net = caffe.Net('root_model.prototxt',caffe.TEST)
caffe.set_mode_gpu()
```

EXPECTED RESULTS

You should receive no errors when running caffe on the command line, or importing caffe within python.

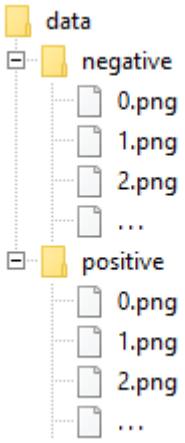
Prepare a classification dataset

Step 2.

In order to scan images and perform localisation, a network must first be trained to classify individual samples of the images. In our work these samples were either 32x32 RGB images for roots, or 64x64

RGB images for shoots.

The makedb.py script will combine multiple images in folders into two LMDB database files containing training and testing images. The required input is one folder of images per class. For example with root tip classification there are two classes (negative, positive) with shoot classification there are five (negative, leaf tip, leaf base, ear tip, ear base).



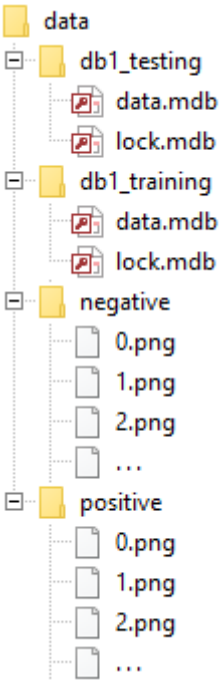
makedb.py can be executed as follows:

```
python makedb.py training-ratio output-prefix class1 class2 class3 ...
```

E.g.

```
python makedb.py 0.8 ./data/db1 ./data/negative ./data/positive
```

Will create db1_training and db1_testing folders, containing training and testing databses with an 80% ratio of training:testing images.



EXPECTED RESULTS

Your directory structure should now contain two folders of lmdb files, holding the training and testing images.

Create a mean image file

Step 3.

The mean image is subtracted from all data during training and testing in order to improve accuracy on small networks. It ensures that the iamge data is centred around 0, rather than 128 (which is standard for 8 bit images). The mean file can be created using the compute mean executable that ships with caffe:

```
/caffe/install/dir/compute_image_mean ./data/db1_training
db1.mean.binaryproto
```

This will compute the average colour of all pixels within the db1_training database, and store this in a binaryproto file.

EXPECTED RESULTS

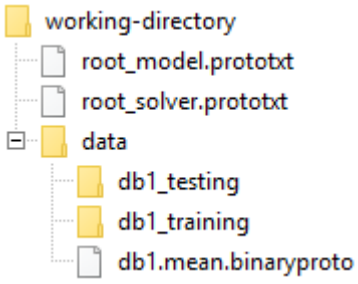
A mean binaryproto file that stores the average colour for all images in the training set.

Train the network

Step 4.

Training in caffe can be achieved using the command prompt, by passing a solver file, which in turn refers to a model file defining the CNN architecture. The model file references the training and testing

databases, as well as the mean file.



Training can be run by calling the caffe executable:

```
/caffe/install/dir/caffe train --solver=./torch_solver.prototxt --gpu 0
```

The training should run, produce a log of the ongoing training and testing accuracy.

```
...
Network initialization done.
Solver scaffolding done.
Starting Optimization
 Solving Torch
 Learning Rate Policy: step
 Iteration 0, Testing net (#0)
     Test net output #0: accuracy = 0.658926
     Test net output #1: class = 1
     Test net output #2: class = 0
     Test net output #3: loss = 0.687874 (* 1 = 0.687874 loss)
 Iteration 0 (-1.4173e-38 iter/s, 23.9718s/100 iters), loss = 0.682398
     Train net output #0: loss = 0.682398 (* 1 = 0.682398 loss)
 Iteration 100 (2.29117 iter/s, 43.6459s/100 iters), loss = 0.638753
     Train net output #0: loss = 0.638753 (* 1 = 0.638753 loss)
 Iteration 200 (2.27664 iter/s, 43.9244s/100 iters), loss = 0.612924
     Train net output #0: loss = 0.612924 (* 1 = 0.612924 loss)
 Iteration 300 (2.26743 iter/s, 44.1027s/100 iters), loss = 0.565217
     Train net output #0: loss = 0.565217 (* 1 = 0.565217 loss)
 Iteration 400 (2.25196 iter/s, 44.4058s/100 iters), loss = 0.620955
     Train net output #0: loss = 0.620955 (* 1 = 0.620955 loss)
 Iteration 500 (2.25102 iter/s, 44.4244s/100 iters), loss = 0.568278
     Train net output #0: loss = 0.568278 (* 1 = 0.568278 loss)
 Iteration 600 (2.25144 iter/s, 44.416s/100 iters), loss = 0.540305
     Train net output #0: loss = 0.540305 (* 1 = 0.540305 loss)
 Iteration 700 (2.25164 iter/s, 44.4122s/100 iters), loss = 0.504978
     Train net output #0: loss = 0.504978 (* 1 = 0.504978 loss)
 Iteration 800 (2.25089 iter/s, 44.4269s/100 iters), loss = 0.497804
     Train net output #0: loss = 0.497804 (* 1 = 0.497804 loss)
 Iteration 900 (2.25158 iter/s, 44.4133s/100 iters), loss = 0.477872
     Train net output #0: loss = 0.477872 (* 1 = 0.477872 loss)
 Iteration 1000, Testing net (#0)
     Test net output #0: accuracy = 0.909238
```

```
    Test net output #1: class = 0.977556
    Test net output #2: class = 0.777178
    Test net output #3: loss = 0.345742 (* 1 = 0.345742 loss)
...
```

Note that the loss (error) is decreasing and the accuracy between the test at iteration 0 and test at iteration 1000 is greatly improved. Caffe will snapshot trained network weights into a caffemodel file.

EXPECTED RESULTS

A .caffemodel file containing the weights of the trained network.

Use the trained weights to scan over whole input images

Step 5.

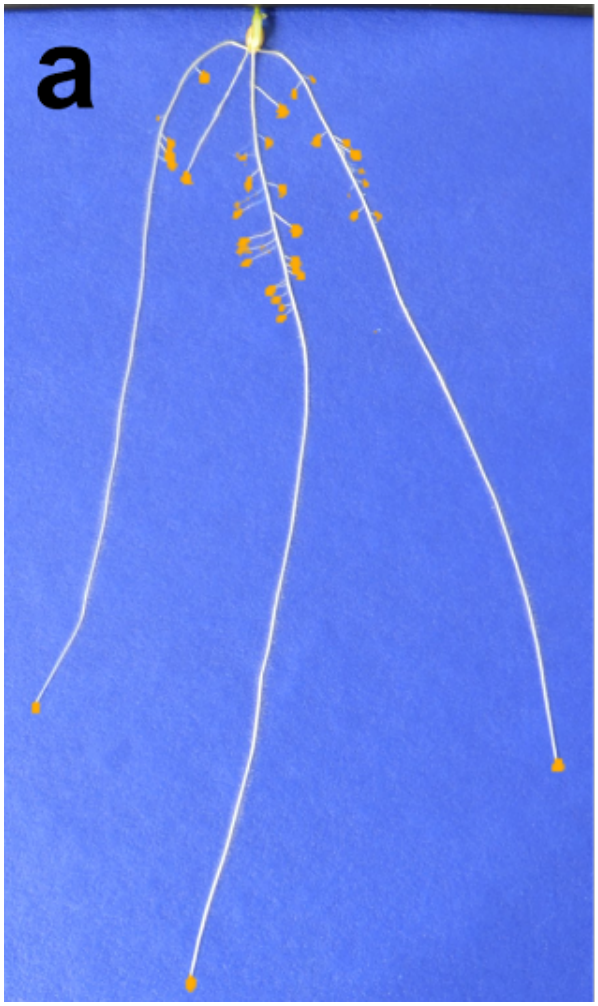
The heatmap.batch.py file will load the network along with pre-trained weights, and scan an input image at regular intervals. This will be used to create a numpy array (.npy file) that containes the likelihood of each class for each pixel. The render_heatmap.py script will take an input image and combine it with this npy file into a coloured output for visualisation. E.g.

```
python heatmap.batch.py ./image.jpg ./image.npy
```

Creates an array containing the class likelihoods, then:

```
python render_heatmap.py ./image.jpg ./image.npy ./output.jpg
```

Will create an output file similar to these:



Note: The .sh files within the testing directories can help scan multiple files using this approach, rather than one at a time.

EXPECTED RESULTS

Output arrays and images locating features of interest.