Project: Image Analysis 2015/2016
Anomaly Detection for Crop Monitoring: Report

Stefano Cerri 849945

June 2, 2016

Contents

List of Figures

Chapter 1

Scope

1.1 Problem

One of the most important tasks to be addressed in intelligent farms is the crop monitoring. Computer Vision and Machine Learning algorithms are typically combined to analyze and classify images acquired by autonomous robots that monitor the growth of crops, and in particular, to discriminate crops (i.e. the good plants) from weeds (any undesirable or unwanted plant growing wild, especially those that takes food or nourishment from crops). Crop/weed discrimination are often tacked as an image-classification problem.

1.2 Dataset

The dataset used in this project is the CWFID dataset[1]. It comprises field images, vegetation segmentation masks and crop/weed plant type annotations. The paper[2] provides details, e.g. on the field setting, acquisition conditions, image and ground truth data format. All images were acquired with the autonomous field robot Bonirob in an organic carrot farm while the carrot plants were in early true leaf growth stage.

1.3 Assignment

The assignment is to create a Convolutional Neural Network, using deep learning, that classificates the images in crop,weed or ground.

[1] https://github.com/cwfid/dataset
[2] see Appendix→Reference document

Chapter 2

Extract patches from the dataset

This work was done by two collegues,Jorge Carpio Lopez de Castro and Andrea Luigi Edoardo Caielli, that work on the same project. The script creates a dataset of patches (with dimension $51 \times 51 \times 3$). A patch is a sub-region of an image that has a particular property. In our case there are three types of patches: crop patches, weed patches and ground patches. The result of the script is a dataset composed of:

- **3000 training patches** : 1000 crop, 1000 weed, 1000 ground
- **3000 testing patches** : 1000 crop, 1000 weed, 1000 ground

Here some examples of patches:

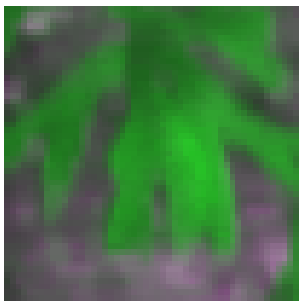

Figure 2.1: crop patch

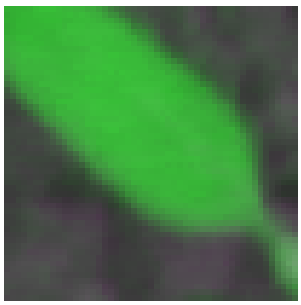

Figure 2.2: weed patch

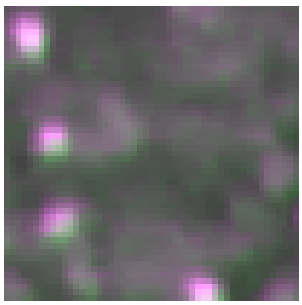

Figure 2.3: ground patch

Chapter 3

Training the network

3.1 MatConvNet library

To perform the given assignment I donloaded and installed MatConvNet library[1]. Note that in order to use some functionalities of the library, such as using GPU acceleration, it has some requirements[2]. There is also a manual[3] that explains all the blocks of a CNN both from the "mathematical" and "code function" view. The configuration that I used was:

- MatConvNet 1.0-beta18
- Matlab 2016a

3.2 What is a Convolutional Neural Network?

CNN is a type of feed-forward artificial neural network in which the connectivity pattern between its neurons is inspired by the organization of the animal visual cortex, whose individual neurons are arranged in such a way that they respond to overlapping regions tiling the visual field.It consist of multiple layers of small neuron collections which process portions of the input image, called receptive fields. The outputs of these collections are then tiled so that their input regions overlap, to obtain a better representation of the original image; this is repeated for every such layer. A more detailed description of the layer of a Convolutional Neural Network can be found in the next section.

[1] http://www.vlfeat.org/matconvnet/
[2] http://www.vlfeat.org/matconvnet/gpu/
[3] http://www.vlfeat.org/matconvnet/matconvnet-manual.pdf

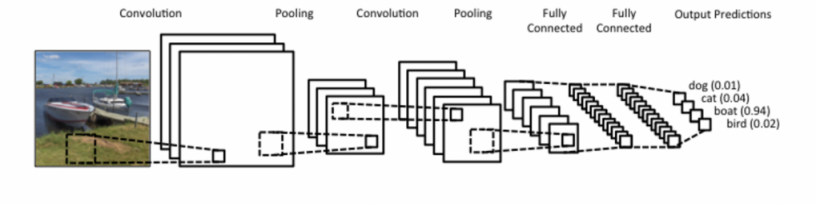

Figure 3.1: Example of a Convolutional Neural Network

3.3 Build the architecture of the network

3.3.1 Typical layer of a CNN

The typical layer of a Convolutional Neural Network are:

- **Convolutional layer**:it is the core building block of a Convolutional Network, and its output volume can be interpreted as holding neurons arranged in a 3D volume.The CONV layer's parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume, producing a 2-dimensional activation map of that filter. As we slide the filter, across the input, we are computing the dot product between the entries of the filter and the input. Intuitively, the network will learn filters that activate when they see some specific type of feature at some spatial position in the input. Stacking these activation maps for all filters along the depth dimension forms the full output volume. Every entry in the output volume can thus also be interpreted as an output of a neuron that looks at only a small region in the input and shares parameters with neurons in the same activation map (since these numbers all result from applying the same filter).Three hyperparameters control the size of the output volume: the **depth**, **stride** and **zero-padding**

- **Pooling Layer**:it is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice). The depth dimension remains unchanged

- **Fully-connected layer**:neurons in a fully connected layer have full connections to all activations in the previous layer. Their activations can hence be computed with a matrix multiplication followed by a bias offset.In other libraries,fully connected blocks or layers are linear functions where each output dimension depends on all the input dimensions.MatConvNet does not distinguish between fully connected layers and convolutional blocks

- **ReLu**:the rectified linear unit will apply an elementwise activation function, such as the $f(x) = max(0, x)$ thresholding at zero. This leaves the size of the volume unchanged

- **SoftMax**: Soft Max is a loss functionakes the score compete through the normalization factor.Softmax can be seen as the combination of an activation function (exponential) and a normalization operator

3.3.2 Typical architecture

The most common form of a ConvNet architecture stacks a few CONV-RELU layers, follows them with POOL layers, and repeats this pattern until the image has been merged spatially to a small size. At some point, it is common to transition to fully-connected layers. The last fully-connected layer holds the output, such as the class scores. In other words, the most common ConvNet architecture follows the pattern:

$INPUT \rightarrow [[CONV \rightarrow RELU] * N \rightarrow POOL?] * M \rightarrow [FC \rightarrow RELU] * K \rightarrow FC \rightarrow LOSS$

where the $*$ indicates repetition, and the POOL? indicates an optional pooling layer. Moreover, $N >= 0$ (and usually $N <= 3$), $M >= 0$,$K >= 0$ (and usually $K < 3$).

3.3.3 Selected architectures

I selected different network architectures for comparing it, after the training, with the testing set. The architectures are:

1. $INPUT \rightarrow [CONV \rightarrow RELU \rightarrow POOL] * 2 \rightarrow FC \rightarrow RELU \rightarrow FC \rightarrow LOSS$
2. $INPUT \rightarrow [CONV \rightarrow RELU \rightarrow POOL] * 3 \rightarrow FC \rightarrow LOSS$
3. $INPUT \rightarrow [CONV \rightarrow RELU \rightarrow POOL] * 4 \rightarrow FC \rightarrow LOSS$
4. $INPUT \rightarrow [CONV \rightarrow RELU \rightarrow POOL] * 3 \rightarrow FC \rightarrow LOSS$
5. $INPUT \rightarrow [CONV \rightarrow RELU \rightarrow POOL] * 3 \rightarrow FC \rightarrow RELU \rightarrow FC \rightarrow LOSS$
6. $INPUT \rightarrow [CONV \rightarrow RELU \rightarrow POOL] * 3 \rightarrow [FC \rightarrow RELU] * 2 \rightarrow FC \rightarrow LOSS$

3.4 Train the network

3.4.1 Create a validation set

In order to avoid overfitting I split the testing set into two sets; a validation test and a test set.The validetion set can be seen as a set of examples used to tune the parameters of the classifier. The final dataset is composed by:

- **training set**: 3000 patches: 1000 crop patches, 1000 weed patches, 1000 ground patches
- **validation set**: 1500 patches: 500 crop patches, 500 weed patches, 500 ground patches
- **testing set**: 1500 patches: 500 crop patches, 500 weed patches, 500 ground patches

3.4.2 Stochastic gradient descent with momentum

To train the network I used stochastich gradient descent with momentum. Stochastic gradient descent (often shortened in SGD) is a stochastic approximation of the gradient descent optimization method for minimizing an objective function that is written as a sum of differentiable functions. The problem of minimizing an objective function can be describe as:

$$Q(w) = \sum_{i=1}^{n} Q_i(w)$$

where the parameter which minimizes $Q(w)$ is to be estimated. Each summand function Q_i is typically associated with the $i-th$ observation in the data set (used for training). Stochastic gradient descent with momentum remembers the update Δw at each iteration, and determines the next update as a convex combination of the gradient and the previous update

$$\Delta w := \eta \bigtriangledown Q_i(w) + \alpha \Delta w$$
$$w := w + \Delta w$$

where η is the learning rate.

3.4.3 Function used

The function cnn_train.m can be used with different datasets and tasks by providing a suitable getBatch function (implemented in cnn_cwfid.m). The function automatically restarts after each training epoch by checkpointing. The function supports training on CPU or on one or more GPUs (specify the list of GPU IDs in the ‘gpus‘ option). Multi-GPU support is relatively primitive but sufficient to obtain a noticable speedup.

3.4.4 Results of the training

Here are the 6 networks training graph. On the left side there is the objective function and on the right side there is the error function.

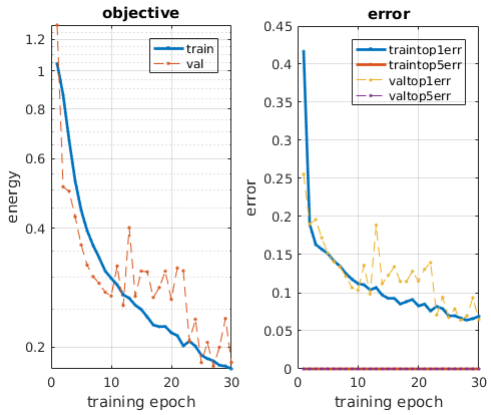

Figure 3.2: network 1 training

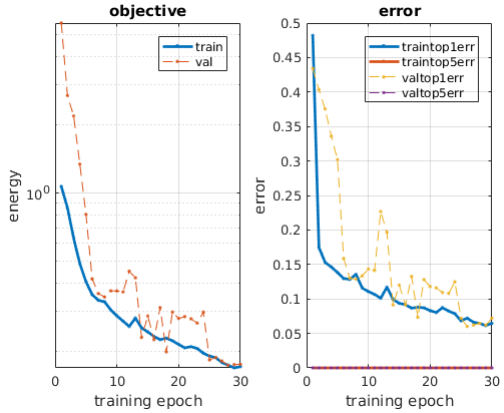

Figure 3.3: network 2 training

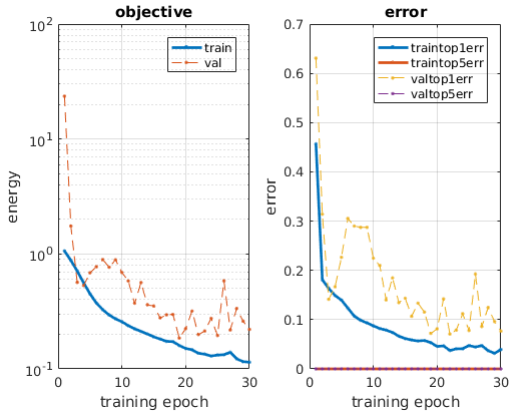

Figure 3.4: network 3 training

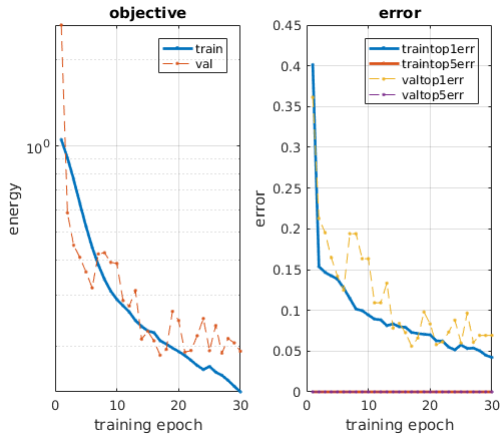

Figure 3.5: network 4 training

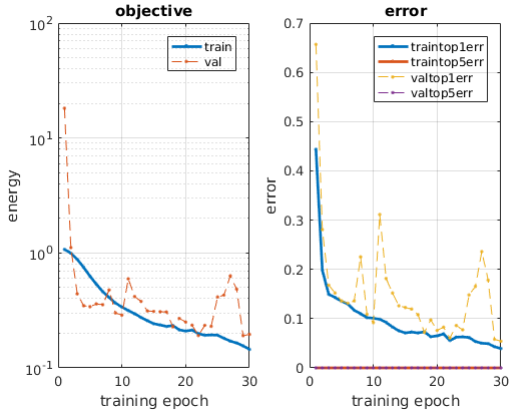

Figure 3.6: network 5 training

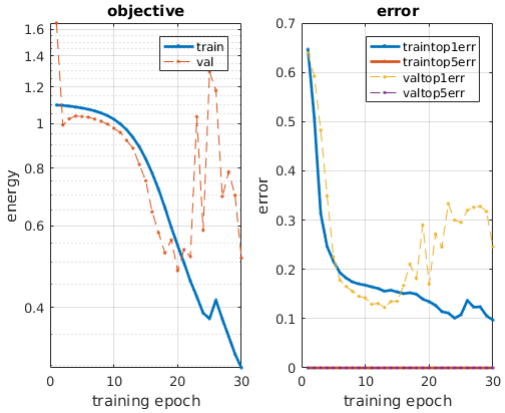

Figure 3.7: network 6 training

Chapter 4

Evaluate the network

4.1 Compute the accuracy of each network

In order to evaluate the network I calculate the accuracy. The accuracy can be computed as:

$$ACC = \frac{TP+TN}{TP+TN+FP+FN}$$

where TP = true positive , TN = true negative , FP = false positive and FN = false negative

Network	**Accuracy**	**Accuracy Crop**	**Accuracy Ground**	**Accuracy Weed**
1	94,00%	92,80%	99,80%	89,40%
2	94,07%	95,60%	99,80%	86,80%
3	92,53%	92,20%	99,80%	85,60%
4	94,13%	93,00%	99,80%	89,60%
5	**94,20%**	**94,00%**	**99,80%**	**88,80%**
6	92,33%	93,60%	99,80%	83,60%

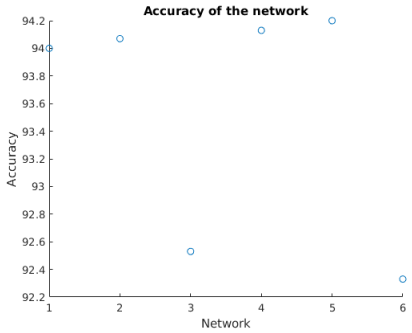

Figure 4.1: accuracy of the networks

The network that has the highest accuracy is **network 5**. From now on I will discuss and talk about network 5 only because it has the best performance and it is the one that I used for the next step.

4.2 Changing the number of maps of each convolutional filter

Once the network has been selected I tried to change the number of maps of each convolutional filter in order to increase the accuracy. Here are some combination of number of maps that I used and the correspondent accuracy.

Filter 1	**Filter 2**	**Filter 3**	**Filter 4**	**Accuracy**
10	20	40	80	94,20%
15	30	60	120	92,60%
8	**16**	**32**	**64**	**94,33%**
7	14	28	56	93,73%
9	18	36	72	93,20%
12	24	48	96	92,53%

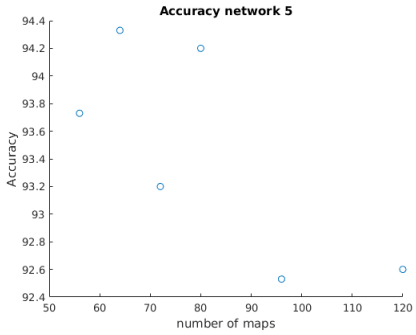

Figure 4.2: accuracy VS number of maps; only the number of maps of the last filter layer is plotted in the graph

This is the training graph with a better combination of number of maps. This configuration is the final one that I used for the given assignment and has a global accuracy of **94,33% 94,20%** for crop **99,80%** for ground and **89,00%** for weed.

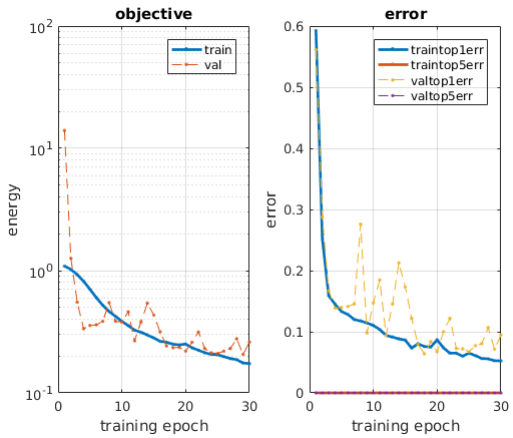

Figure 4.3: final training

4.3 Calculate Confusion Matrix

In order to have a better evaluation of the network I calculated also the confusion matrix, also known as an error matrix, that is a specific table layout that allows visualization of the performance of the classifier.

The Confusion matrix, for the testing set, is:

		Predicted		
		crop	weed	ground
Actual Class	crop	472	28	0
	weed	55	445	0
	ground	1	0	499

4.4 Sliding window approach

In order to have a rough segmentation of the training set I also used to evaluate the network the sliding window approach.It consist of:

- take a testing image. The size of the testing image is $966 \times 1296 \times 3$
- extract patches from that.I used patches of dimension $51 \times 51 \times 3$. I used two kind of stride: 10 and 15. So I obtained respectively 11500 and 5063 patches
- make a new image the same size of the original testing image. On this image, I painted the central pixel of each of the patches with a color depending on the best class that the classifier choosed. The color used are the same of the annotation image of the dataset CWFID so green for crop, red for weed and black for ground
- repeat for all the images of the testing set

What I noticed was that the networks works not as expected. In fact the result images were very different from the annotation image. The accuracy was about 33% a very low accuracy if we compare with a 94% accuracy found with the testing set.
Below I attached the result of the first testing image obtained with the sliding window approach with a stride of 10:

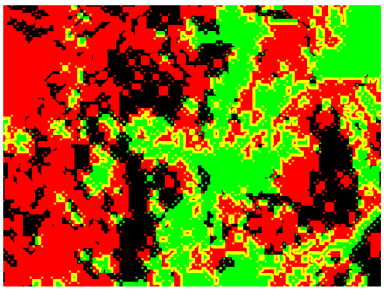

Figure 4.4: sliding window stride=10

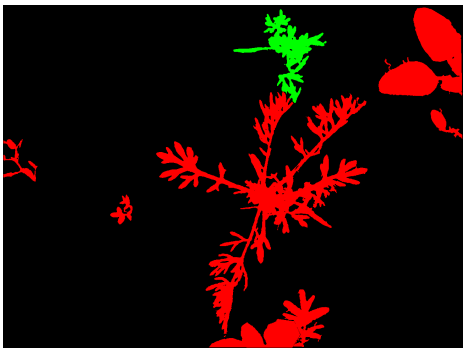

Figure 4.5: annotation

4.5 Umbalanced class problem

The problem was that the network suffers of umbalanced data. Umbalanced data typically refers to a problem of classification where the classes are not represented equally. In fact what I have done was to count the pixels of crop, weed and ground from the annotation images referring to the training set and I discovered that:

- **92,50%** of the pixels are marked as ground
- **5,96%** of the pixels are marked as weed
- **1,54%** of the pixels are marked as crop

I tried to solve this problem by apply a threshold on the probabilities results of the ground patches. The threshold must be a value for which the 92,50% of the ground probability are bigger.In other words:

$$T = groundVector[numberOfPatches \times 0,9250]$$

where $groundVector$ is the vector containing the probabilities of the ground patches in descending order and $numberOfPatches$ is the number of patches apply to the image.
What I get was that the results looks better with an accuracy close to 70% but the networks failed to classify crop form weed. So I decided to apply the same threshold approach also for the crop probabilities and what I get are an overall accuracy from all the test images of 89,62% for a stride of 10 and 91,25% for a stride of 15.
Here are some results. The full results can be view in the gitHub repository[1].

[1] see Appendix Repository

Figure 4.6: sliding window stride=15

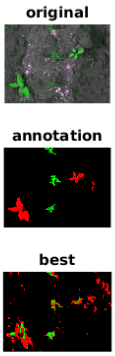

Figure 4.7: sliding window stride=15

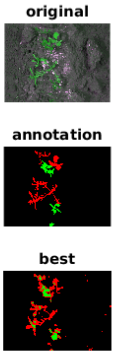

Figure 4.8: sliding window stride=10

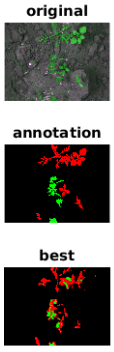

Figure 4.9: sliding window stride=10

Chapter 5

Summary and future extensions

In this chapter I want to make some considerations about how the network works and some future extensions that can be applied in order to have a better performance of the network. If we look at the result of the sliding window we can see that in some images the network classifies some crop pixel as weed. This wrong classifications are due to the threshold approach applied in the section 4.5. In fact, if in the image there are more crop pixels than the one that I calculate in the training set (1,54%) the network classifies only few pixels as crop. Below I attached an image that shows this behaviour of the network.

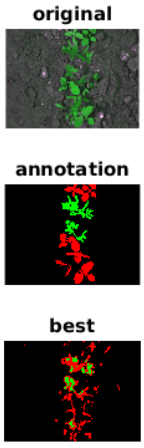

Figure 5.1: bad sliding window result

The result show that a lot of crop pixels are marked as weed and also a lot of weed pixels are marked as ground (the percentage of weed pixels in the training set was 5,96%).

5.1 Possible extensions

In order to have a better performance of the network some solution and extensions can be done:

- **have a larger dataset**: if we have more patches the network could possibly work better. In order to have more patches rotation could be applied
- **resampling the dataset**:
  1. we can add copies of instances from the under-represented class called over- sampling,or more formally sampling with replacement
  2. we can delete instances from the over-represented class, called undersampling
- **try penalized models**: we can use the same algorithms but give them a different perspective on the problem. Penalized classification imposes an additional cost on the model for making classification mistakes on the minority class during training. These penalties can bias the model to pay more attention to the minority class

Chapter 6

Appendix

6.1 Repository

All the code, the dataset, the intermediate results and other documents can be found at this repository https://github.com/ste93ste/cwfid_classification

6.2 Reference document

- http://rd.springer.com/chapter/10.1007%2F978-3-319-16220-1_8

6.3 Software and tool used

- LaTeX (http://www.latex-project.org/) : to redact and to format this document
- Matlab R2016a (http://uk.mathworks.com/products/matlab/) : to compute and evaluate the network
- MatConvNet (http://www.vlfeat.org/matconvnet/) : MATLAB toolbox implementing Convolutional Neural Networks (CNNs)