Segmentation of Erythematous Lesions using Multispectral Imaging

A project report

By-
Saahil Ognawala

Supervised by –
Alexandru Duliu,
Sebastian Pölsterl,
Loic Peter and
Prof. Dr. Nassir Navab

Introduction

Erythema are common lesions, that are often the primary symptom of ailments such as psoriasis. The evolution of the lesion's size is indicative of its response to medication and driving decisions regarding further treatment.
Multi-spectral imaging enables superior color reproduction compared to RGB cameras. While it is widely employed in histology, it is still rarely used in dermatology. In this work we introduce an approach that does not separate multi-spectral image acquisition from the examination but rather integrates into the normal workflow. We evaluate our performance to segment erythematous lesions, focusing on either pixel-based, image features based or patch based methods.

Segmentation Methods

Random Forest Classification [5][6]
A random forest (RF) classifier is an ensemble method of machine learning , with a base that relies on decision tree learning. RF involves generating multiple decision trees from training data and emits a class that appears most frequently among all trees in the ensemble.
Decision tree learning is a supervised prediction method, commonly used for both, classification and regression problems, in the data science community. At every level of a decision tree, the classification scheme splits the training data set into subsets depending on the split values dictated by the internal nodes. These internal nodes learn the split values based on a single input feature at a time. The leaf nodes of the tree finally emit the class labels.
A random forest generates a number of such decision trees that differ in their internal nodes, the depth sequence of features as they occur in the tree, and their split values. The ensemble of all the trees in the RF are trained with the same training set and the trees learned independently. The class label that occurs most often among the labels emitted by all the decision trees is, then, chosen as the label by the framework.

Logistic Regression
Logistic regression is a statistic classification method based on probability theory. Under this method, we try to fit a classification model based on a set of feature vectors by deriving a weight vector corresponding with the features. Logistic regression is a discriminative model, and the output of the model is a probability value associated with a class. It is given by –

$$p(c=0|x) = \sigma(b+x^T w),$$
$$p(c=1|x) = 1-\sigma(b+x^T w)$$

assuming that [0,1] are the possible class labels for our data.
Here, *x* is the input vector and *w* is the weight vector corresponding to the input vector.
σ is a "squashing function" that scales real number values down to an interval of (0,1). Logit function is usually used as the squashing function here.

$$\sigma(x) = 1/(\ 1+e^{-x})$$

We try to find the best set of weights by maximizing the probability of classifying the training set correctly. The is done by maximizing the likelihood function –

$$p(C \mid b, w, X) = \prod p(c=1|x^n,b,w)^{c^n}(1-p(c=1|x^n,b,w))^{1-c^n}$$

For training the logistic regression, we minimize the negative log of the above function using simple gradient descent.

$$w \leftarrow w - \eta \nabla_W L$$

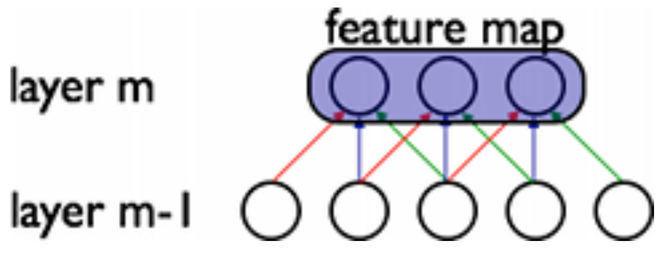
$$b \leftarrow b - \eta \nabla_b L$$

Where, L is the negative log-likelihood of joint probability function of classification.

AdaBoost [4]
Adaptive boosting (AdaBoost) learns a linear combination of weak classifiers, such as decision trees, logistic regression classifiers etc., based on continuously re-weighting mis-classified samples. Predictions of all weak learners are combined by weighted majority voting.

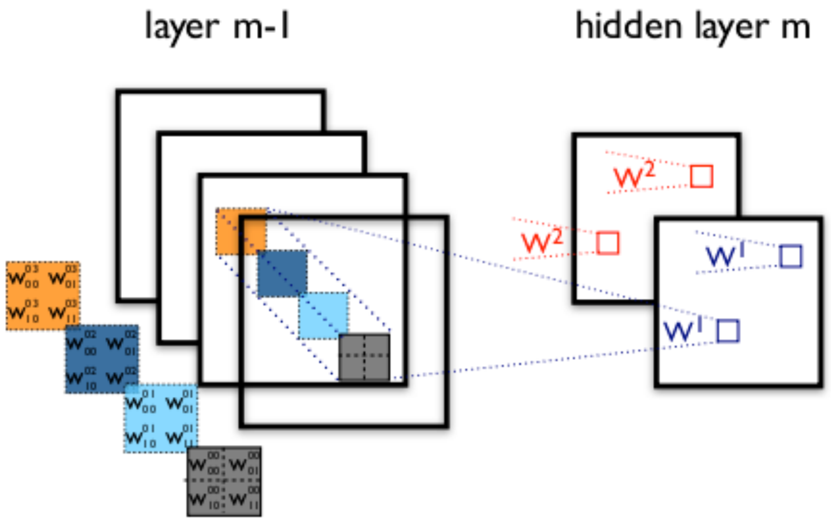
Multilayer Perceptrons [7]
MLPs are graphical models organized in layers with the topology of the network being restricted to directed edges originating from nodes of the previous layer and no connections within a layer or cycles in the graph. Each layer constitutes a function $f(x) = (1 + e^{-b-Wx})^{-1}$, with x being the layer's input, b the bias term and W the weight matrix with respect to the edges between the previous layer and the current layer. In classification tasks, the top most layer usually corresponds to a logistic regression node.

Convolutional Neural Network [8]
Convolutional Neural Networks (CNN) are a specialized form of multilayer perceptron, introduced by LeCun et al. in [Gradient-based learning applied to document recognition]. These networks have the capacity to exploit the spatial local correlation properties present in natural images for classifying objects, much like the complex human visual cortex.
The first characteristic of CNN is the sparse connectivity between adjacent layers of hidden nodes. The strong spatial local correlation is enforced by connecting a hidden unit in the m-th layer with a local (by area) subset of input nodes from (m-1)-th layer.



In the (m-1)-th layer of the image above, input pixels from the image are flattened out in the form of a 1-D vector, and the adjacent nodes form the neighboring pixels. It can be seen that a hidden node in m-th layer is only connected to a subset localization of the input image.

Moreover, the weights of the same color above are shared in a "feature-map" and hidden units are replicated across the entire image input. This is the second defining characteristic of CNNs. The sharing of weights in a feature map ensures that the visual properties of localizations learned in an image are uniform in a single map. Richer features can be learned from the image by using multiple feature maps in the same layer of a CNN. This is illustrated in the image below –



In the image above, (m-1)-th layer applies 4 feature maps of shared weights in the image.
In order to reduce the complexity of computation at the hidden layer, and also to strengthen the spatial invariance property of an image, we sub-sample the activation values from layer (m-1) by MaxPooling. MaxPooling strides over a smaller square area than the feature maps and emits the maximum activation in the square region as the single output value of all the contained pixels.
In order to get the class values for an image, the sub-sampling layer is connected to final logistic regression layer. The whole network is trained using the simple gradient descent method as explained in the logistic regression section.
From the description of the CNN, we can see that the logistic regression layer now gets as input, a much richer set of features learned from the convolutional layer rather than just the pixel values for a point. In particular, using the spatial localization information from he image, we hope to get better segmentation than a naïve pixel based classification.

Learning Process

Training
We train all three methods of segmentation using images from, both, lab and clinical environment. For the first two methods that rely on raw pixel values from all input channels, every row of the dataset contains a vector containing these individual channel values. For the last method (CNN), we extract patches around the pixel that we need to classify as, either, lesion or non-lesion.
We select pixels from the images to be used for training and validation, using a uniform random distribution over all pixels. In order to keep the class populations approximately uniform, we control the number of lesion and non-

lesion points separately from the ground truth data. The dataset of *n* points is represented as follows –

$$X_n \rightarrow Y_n$$

where X_n is the input matrix with n data points and Y_n is the output column vector with class labels for those points.
In the case of random forest and logistic regression methods, the size of every row of input matrix X_n is equal to the number of image channels, *c*. In the case of CNN, the size of every row of X_n is equal to –

$$c * p_x * p_y$$

where (p_x*p_y) is the size of the patch extracted around the pixel of interest.
After extracting the pixel information and corresponding output labels, we shuffle the entire set uniformly.

Validation

Cross validation technique is used to evaluate the learned parameters in all three methods. At the end of every epoch, the learned model is applied to an unseen validation set to ensure that the it doesn't overfit to the training set but, at the same time, performs reasonably well on unseen test data.

Testing

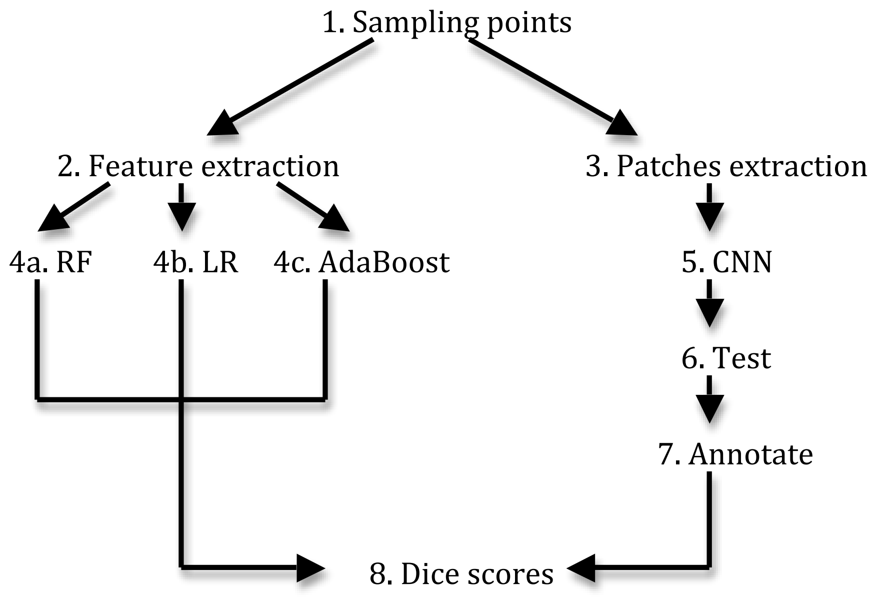
After obtaining a model with trained parameters that does not overfit the training sample we apply it to new unseen data for evaluating our segmentation strategy. For random forest classifier and logistic regression, we extract the feature vector at every pixel in the test image, and emit the class label for it.
For CNN, we extract a patch of size $[p_x,p_y]$ at every pixel located at a distance far enough from the boundaries to not overflow.
We evaluate the performance of all methods by calculation the Dice co-efficient of the segmentation produced against the ground truth images.
The Dice co-efficient is defined as –

$$D = 2\,|G \cap S| \,/\, (|G|+|S|)$$

The numerator denotes the number of pixels classified the same by both, ground truth and segmentation method. The denominator represents the sum of pixels in the ground truth and segmentation. In our experiment, the number of pixels segmented is the same as that in the ground truth.

Code Documentation [1][2]

The segmentation of images can be done by using several machine learning methods. We provide a flowchart of the steps to be taken in order to learn and evaluate the performance of all learning methods on our dataset.



We now describe each of these steps in detail.

[Note: All file paths in Italics *are the examples of path names on the users' machine. Change them to the specific file path as described in the related sections.]*

1. **Sampling points**
   We randomly sample pixel locations in images, to control the number of lesion and non-lesion points used for training. Sampling locations before extracting features/patches was found to be very useful since this reduces the number of calls to the pseudo-random number generator. The sampling script is located at `utils/sampling.py`

   In this script, change the following lines to control the number of marked/lesion points (POPULATION_M) and unmarked/background points (POPULATION_U)
   ```
   POPULATION_M = 3000
   POPULATION_U = 5000
   ```

   Then run the script as –
   ```
   python sampling.py <mask_file_name>.png
   ```

   This will generate a numpy file in the same location as the mask image. The name of this numpy file would be –
   ```
   mask_file_name_<POPULATION_M+POPULATION_U>_sample.npy
   ```

   Run this script on all training images (or write a shell script to automate).

2. **Feature extraction**
   For training learning methods that use feature vectors viz. random forests, logistic regression and adaboost, we need to extract features from

the images that we must train on. For all images, features will be extracted at the pixel locations listed in the sampling files from step 1.

Features can either be raw pixel intensity values from all input channels, or richer features extracted from jfeaturelib. We discuss these both cases below –

a) <u>Pixel intensity vector</u>

   To extract a feature vector with raw pixel intensity values, use the script – `utils/pixels.py`

   In the above script, change the population of lesion and non-lesion points in following lines –

```
                    POPULATION_M = 3000
                    POPULATION_U = 9000
```

   These numbers should be the same as those used during sampling stage.

   Change the number of input channels in the list –

```
       CHANNELS = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

   Then run the script as follows –

```
 python pixels.py <path_to_image> <mask_file_name>.png
```

   Here <path_to_image> should be the entire image name, without the trailing notation for channel number and file extension. Eg. if the first and second channels of the image are */home/ognawala/patient_im-0.png* and */home/ognawala/patient_im-1.png*,
   then the <path_to_image> should be */home/ognawala/patient_im*

   The mask name, however, should be the entire path to the mask image, including the image extension.

   Images corresponding to all the channel numbers listed in CHANNELS will be picked at location <path_to_image>-n.png, where n is a number in CHANNELS.
   The numpy sampling file with POPULATION_M+POPULATION_U points should be located in the same location as the mask file.

   This script will create the following files in the same location as `<path_to_image>-0.png`

```
                   <path_to_image>_X.npy
                   <path_to_image>_Y.npy
```

   Run this script on all training images (or write a shell script to automate).

b) <u>Rich Features</u>

To extract a feature vector, such as Local Binary Pattern, use `long_features.py` instead of `pixels.py` script, also located under `utils/`. This script assumes that jfeaturelib[3] has been run on images from each channel separately and a bzip archive containing the resultant csv has been created.
Run the script as follows –

```
        python long_features.py <path_to_image>
                <mask_file_name>.png
```

As with pixel intensity vector, `<path_to_image>` must have no leading channel number or file extension. Additionally, a bzip file titled `<path_to_image>.csv.bz2` must exist.

The numpy sampling file with POPULATION_M+POPULATION_U points should be located in the same location as the mask file.

This script will create the following files in the same location as `<path_to_image>.csv.bz2`

```
                <path_to_image>_X.npy
                <path_to_image>_Y.npy
```

Run this script on all training images (or write a shell script to automate).

3. **Patch extraction**
For convolutional neural networks that rely on image patches rather than feature vectors, patch extraction must be carried out in lieu of feature extraction. For this use the script `utils/patches.py`

In the above script, change the population of lesion and non-lesion points in following lines –

```
                POPULATION_M = 3000
                POPULATION_U = 9000
```

These numbers should be the same as those used during sampling stage.

Change the number of input channels in the list –

```
        CHANNELS = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Change the size of extracted patches –

```
                PATCH_SIZE = (55, 55)
```

Then run the script as follows –

```
  python patches.py <path_to_image> <mask_file_name>.png
```

Here `<path_to_image>` should be the entire image name, without the trailing notation for channel number and file extension. Eg. if the first and second channels of the image are */home/ognawala/patient_im-0.png* and */home/ognawala/patient_im-1.png*, then the `<path_to_image>` should be */home/ognawala/patient_im*

The mask name, however, should be the entire path to the mask image, including the image extension.

Images corresponding to all the channel numbers listed in CHANNELS will be picked at location `<path_to_image>-n.png`, where n is a number in CHANNELS.
The numpy sampling file with POPULATION_M+POPULATION_U points should be located in the same location as the mask file.

This script will create the following files in the same location as `<path_to_image>-0.png`

```
                    <path_to_image>_X.npy
                    <path_to_image>_Y.npy
```

Run this script on all training images (or write a shell script to automate).

4. **Feature based learning methods**
After extracting feature vectors from images, as described in step 2, next step is the actual training, validation and testing step for the three feature based learning methods - Random forest classifier, Logistic regression and AdaBoost. The respective scripts for these are – `random_forest.py, sgd.py, adaboost.py.`

All three scripts have almost the same structure, other than the variable `clf` that assigns the particular algorithm.

Number of spectrums used for learning can be changed in the line –

```
N_CHANNELS = 10
```

POPULATION_M+POPULATION_U is equal to the sampling size of one image. This should be assigned as N_SAMPLING in the following line –

```
N_SAMPLING = 12000
```

Then the value of N must be set in the following line –

```
N = N_SAMPLING*<k>
```

Where the number of images on which the `pixels.py` was run is *k*.

The images on which a trained model must be applied to, for testing, must be listed under the variable `TEST_IMAGES`. The protocol for specifying a test image path is the same as that of training path, i.e. without the trailing notation for channel number and file extension.

The algorithm assigned to `clf` can be modified by changing the default parameters. The parameter list for the three learning methods can be found at –

http://scikit-learn.org/dev/modules/generated/sklearn.ensemble.RandomForestClassifier.html
http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html
http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

The scripts can by run as follows –

```
    python random_forest.py <path_to_extracted_pixels>
```

`random_forest.py` can be replaced by any of the other two script names mentioned above.

`path_to_extracted_pixels` is the parent folder in which *k* pairs of *_X.npy and *_Y.npy are located. These are the numpy files generated by `pixels.py` or `long_features.py` scripts in step 2.

After the script finishes running, annotated test images are generated in the paths point at by TEST_IMAGES elements.

5. **Convolutional Neural Network (CNN)**

CNN is a complex learning method that uses multilayer perceptron units at it's core. Since MLPs are not supported by scikit-learn, we make use of the Pylearn2 framework for machine learning.

Pylearn2 uses two basic components for learning –

1. Data model
2. YAML configuration

The data model is described in the file `model/msr_dataset.py`. In this Python model, we need to change a few parameters to fit our learning needs.

The variable, SHAPE, must indicate a vector of 3D containing the patch size (matching the patch size used in `patches.py`) script and number of channels used for learning.

```
SHAPE = [55, 55, 10]
```

NSAMPLING must equal POPULATION_M+POPULATION_U used during sampling.

```
NSAMPLING = 8000
```

Depending on the number of pictures `patches.py` was run on, set the NTRAIN and NVALID variables.

The second component of Pylearn2 learning is the YAML configuration file. This configuration file defines the structure of the neural network and learning parameters like learning rate, momentum etc. The YAML file is located under `yaml/`.

The sample YAML file is heavily commented and describes the variables. Some of the most relevant parameters are – `kernel_size`, `pool_shape`, `pool_stride`, `output_channels`, `num_channels` and `path_to_data`.

After preparing the YAML file with training and validation specific configurations, run the training as follows –

```
python pylearn/train.py <yaml_file_name>
```

6. **Testing of CNN model**

A trained and optimized CNN model can be tested on the test dataset by using the script `test.py`. This script can be run as follows –

```
python test.py <model_path> <test_path> <out_path>
```

`model_path` refers to the path of the saved CNN model. This is the name of the file pointed at in the YAML configuration file under the variable `save_path`.

`test_path` is the path to test image, in the same format as specified in training path, i.e. without the trailing notation for channel number and file extension.

[Note: The script will generate a number of numpy files related to this image. If all the numpy files have already been generated in a previous run, please place them under the path of the test image, so as to not generate them again.]

`out_path` will contain the result of testing, in form of numpy files with labels for the corresponding point locations. These label files can be used to annotate the test image, as described in step 7.

7. **Annotating test images**

Using the labels from step 6, we can generate annotated images of the tested samples by using the script `annotate.py`. This script can be run as follows –

```
python  annotate.py <test_path> <out_path>
```

The arguments `test_path` and `out_path` are the same as in step 6.

This script will generate an annotated image called `<test_image_name>annotated.png` at the location `<out_path>`.

8. **Dice score calculation**

We finally analyse the performance of a learned model by evaluating the performance on the test images. This is done by calculating a dice score on the test images compared to the ground truth labels.
The script to calculate dice scores is `dice.py`.
Before running the script, the list of test images has to be given to the script. Edit the following line –

```
pairs = [[‘../data/annotated.png’, ‘../data/truth.png’]]
```

The variable, `pairs`, is a 2D array – every row is a pair of images with the first one being an annotated image by one of the learned models, and the second one the ground truth or the mask image. In the example line above, there is one annotated image called `annotated.png`, for which the mask image is `truth.png`.

References

1. Scikit-learn: Machine Learning in Python, Pedregosa *et al.*, JMLR 12, pp. 2825-2830, 2011.
2. Pylearn2: a machine learning research library, Ian J. Goodfellow, David Warde-Farley, Pascal Lamblin, Vincent Dumoulin, Mehdi Mirza, Razvan Pascanu, James Bergstra, Frédéric Bastien, and Yoshua Bengio.
3. JFeatureLib, Franz Graf, 2012
4. Freund, Y., Schapire, R.: Experiments with a new boosting algorithm. In: Machine Learning. Proceedings of the Thirteenth International Conference (ICML ‘96), Eur. Coordinating Committee for Artificial Intelligence; Italian Assoc. Artificial Intelligence (1996) Proceedings of Thirteenth International Conference on Machine Learning, 1996, Bari, Italy
5. Breiman, L.: Random Forests. Machine Learning 45(1) (2001) 5–32
6. Raileanu, L., Stoffel, K.: Theoretical comparison between the gini index and infor- mation gain criteria. Annals of Mathematics and Artificial Intelligence 41(1) (2004) 77–93
7. McCulloch, W., Walter, P.: A logical calculus of ideas immanent in nervous activity. Bulletin of Mathematical Biophysics 5(4) (1943) 115–133
8. Lecun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE 86(11) (1998) 2278–324