Semester project report

Federico Giraud
Gabriel Hochstrasser
Daniele Rolando

supervision by Prof. Bernard Merialdo

federico.giraud@eurecom.fr
gabriel.hochstrasser@eurecom.fr
daniele.rolando@eurecom.fr
bernard.merialdo@eurecom.fr

Semester project report
24/06/2014

Abstract
In the first part we analyze the intermediate layers outputs of a deep convolutional neural network. The data extracted will be used to perform information retrieval measurements, comparing the performance at different layers. In the second part we perform the same measurements on a similar network, after training the network with sparse filtering. In the third part we propose a way to reduce the number of connections in the supervised layers by including the label into the training process.

Keywords
Deep Learning, Imagenet, Information Retrieval, SVM, Average precision, Sparse Filtering

Contents

Part I
Deep network adaptation for information retrieval

1. Introduction

Deep convolutional neural networks are the current state-of-the-art technique for image recognition. Their ability to recognize images relies on the deep architecture of the network, and especially on the first layers, dedicated to feature extraction. The first layers of the network (the ones after the data input layer) are trained in an unsupervised way and they build different levels of abstraction, starting with the raw input data and ending with a compact representation of the main features of the input image.

That extracted features are then given as input to the last part of the network that acts as a normal classifier and learns how to associate the image features to the correct label.

The purpose of this project is to analyze and classify the output of the different unsupervised layers and to compare their performance in terms of average precision.

2. The network

In this section we will describe the framework and the neural network we used to extract image features.

2.1 Caffe

Caffe [2] is an open source framework for convolutional neural network algorithms, developed by the Berkeley Vision and Learning Center. Caffe provides the state-of-the-art deep learning algorithms and is implemented to run both on CPU and GPU.

The network structure and parameters can be specified in a separate file, giving the possibility to easily train and use different convolutional neural networks without the need to write (almost) any line of code.

2.2 ImageNet network

The network we used is the ImageNet example network deployed with the Caffe framework. The network is an implementation of [3], a deep convolutional neural network that was able to significantly outperform all others techniques in the ILSVRC2012 challenge (in classification and localization tasks). The structure is presented in figure 1.

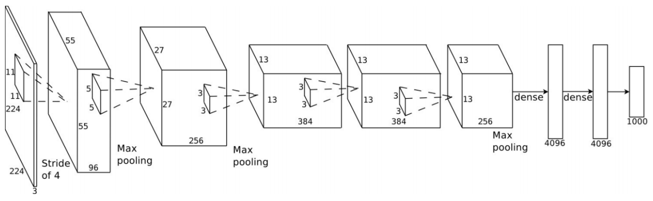


Figure 1. ImageNet network structure

The output of each layer is specified in 3 dimensions: the first two are height and width, and depend on the size of the previous layer and the distance between the patches used for filters; the third dimension is the number of filters specified for that particular layer.

2.2.1 How it works

The input image is cropped and resized to fit in a 224x224x3 (RGB) input layer[1], as shown in figures 2 and 3.

Figure 2. Input image example, 500x290

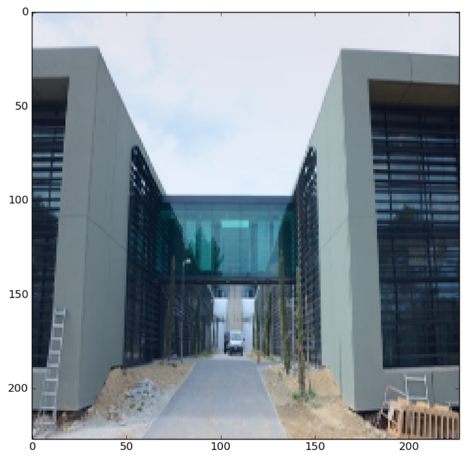


Figure 3. Associated input layer, 227x227

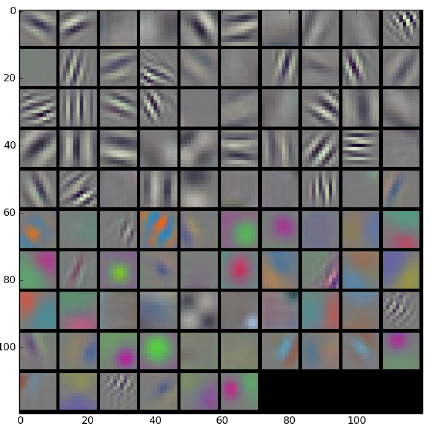
The first layer applies filters to the image, extracting characteristics such as vertical lines, horizontal lines, color gradients and so on. We can see the learned filters of the first layer in figure 4a.

Partial output of the first layer is represented in 4b, where each box represents the output of one of the filters (black means neuron activation). The process is repeated until we reach the last two unsupervised layers, where the image features are represented in the most compact way (figure 5).

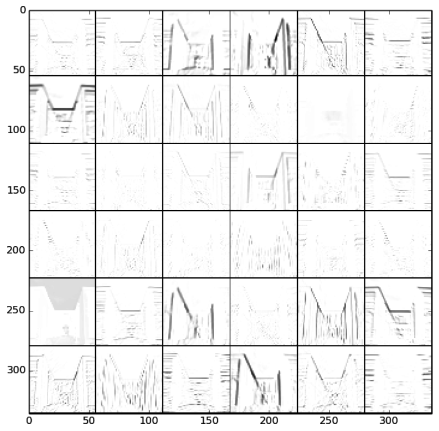
In total, the network has:

- 650,000 neurons

[1]The Caffe implementation uses a 227x227x3 input instead.

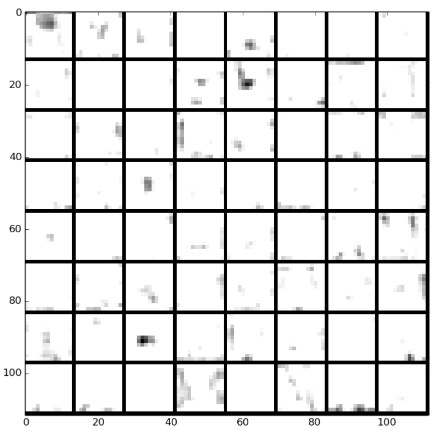


(a) First layer filters

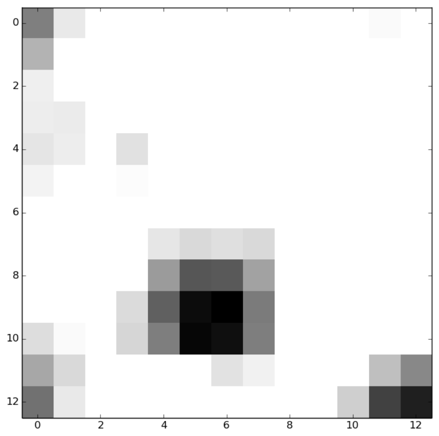


(b) First layer output (one box per filter, first 36 filters)

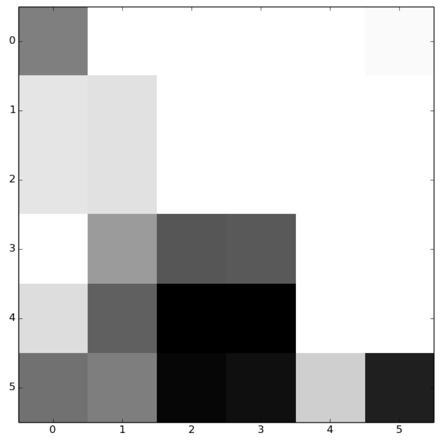
Figure 4. First layer filters and output.



(a) Fifth layer output (one box per filter, first 64 filters, no pooling)



(b) Outout of one of the filters of layer five, without pooling



(c) Same filter after pooling

Figure 5. Last unsupervised layer output.

- 60,000,000 parameters
- 630,000,000 connections

and uses some advanced techniques such as ReLU[2] neurons to speed up computation and dropout [1] to reduce overfit.

2.2.2 Features separation

To show the network feature extraction capabilities in a graphic and intuitive way, we extracted the features of the last unsupervised layer and plotted in two dimensions using the t-SNE [5] technique.

(a) Synset: cat

(b) Synset: dog

Figure 6. Example images from ImageNet.

We downloaded from ImageNet pictures relatives to the synsets cat, dog, bird and lamp (examples in figure 6). The pictures where given to the network and the features of the last unsupervised layer extracted. After this transformation, each image is represented by a vector of 9216 values (being the size of the layer $6x6x256 = 9216$).

This output was then given to a modified t-SNE implementation to plot in 2D. The t-SNE technique reduces the dimensionality of the data while trying to respect the similarity between points. In this way groups of similar vectors in N dimensions will look as separated 2D groups of points, while different vectors will be represented by far away points.

The result for dog and cat images is presented in figure 7. Another experiment was conducted with four synsets: cat, dog, bird and lamp. The resulting plot is in figure 8.

[2]ReLU stands for "rectified linear unit", a neuron with activation function $f(x) = \max(0,x)$.

As we can see from figure 7, even if most of the cat and dog images are similar from a general point of view (most of them represent natural outdoor scenes), the network is able to get the important features of the images that allow to separate ones from the others. As shown by the picture, cat and dog images are clustered in two quite separate groups[3]. A similar result is shown by picture 8, this time with 4 synsets.

2.3 The dataset

The dataset that we are going to use is composed by frames extracted from different videos. Each annotation file contains a list of pictures that are known to contain or not contain the label. For example the following lines are extracted from the Adult.ann annotation file:

```
shot9_19_RKF N
shot9_20_RKF P
shot9_21_RKF N
shot9_22_RKF P
```

and they indicate that there are two images with an adult (those with P) and two where there is no adult (N). Each picture can be present in more than one annotation file and it can contain zero or more annotated objects. Some statistics about the dataset are shown in table 1.

Table 1. Dataset statistics

Train images	119,685	2.2 Gb
Test images	146,788	2.7 Gb
Total images	266,473	4.9 Gb
Labels	500	
Images with annot.	120,660	50%
Average annot. per image	3.6	
No. of labels combinations	25,606	

The ratio between positive and negative images is different for each annotation. For example the annotation "Adult" has about 25% of positive images, while the annotation "Bus" has only 0.1%. The complete statistics for some of the annotations are shown in table 2.

2.4 libSVM

To classify the data coming from the intermediate layers we used the libSVM library, one of the most commonly used libraries for Support Vector Machines. Even if very small, the library supports many features such as:

- Different SVM formulations
- Various kernels (linear, polynomial, radial basis, sigmoid)
- Multi-class classification
- Cross validation

[3]Of course t-SNE, as any other dimensionality reduction technique, cannot perfectly represent the original data disposition.

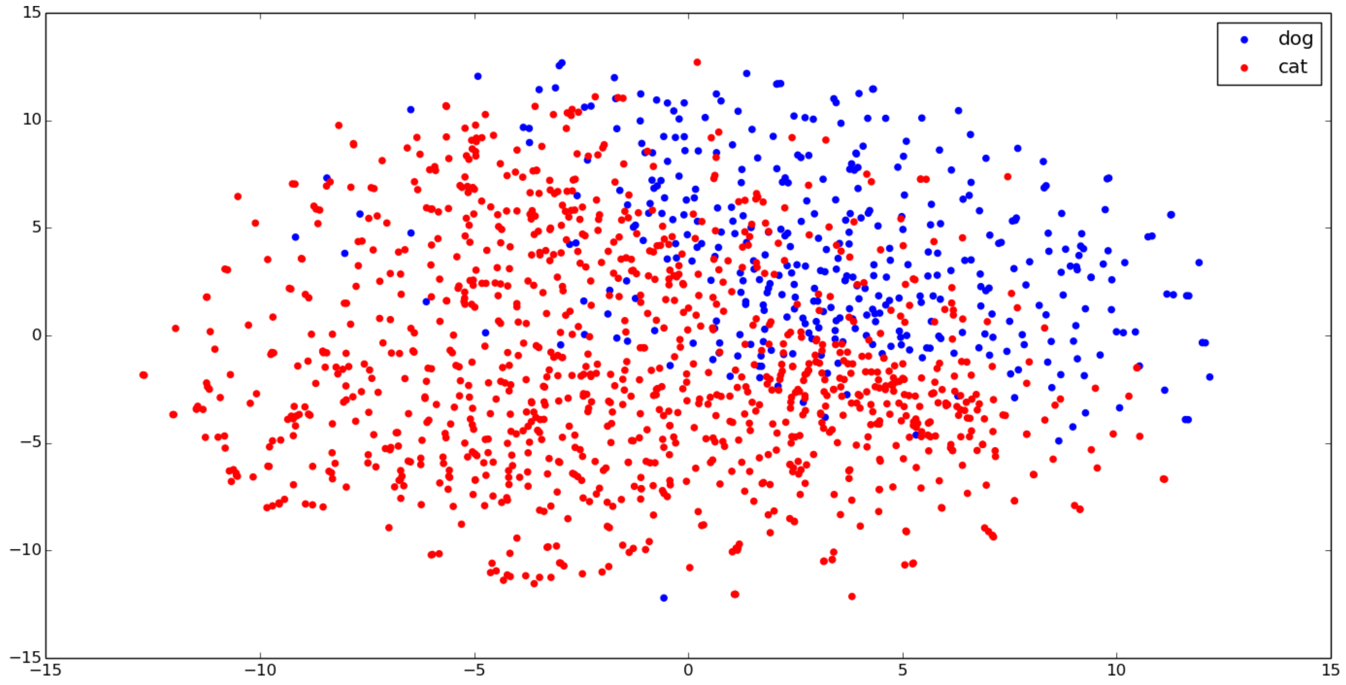


Figure 7. t-SNE representation of layer 5 output (2 synsets)

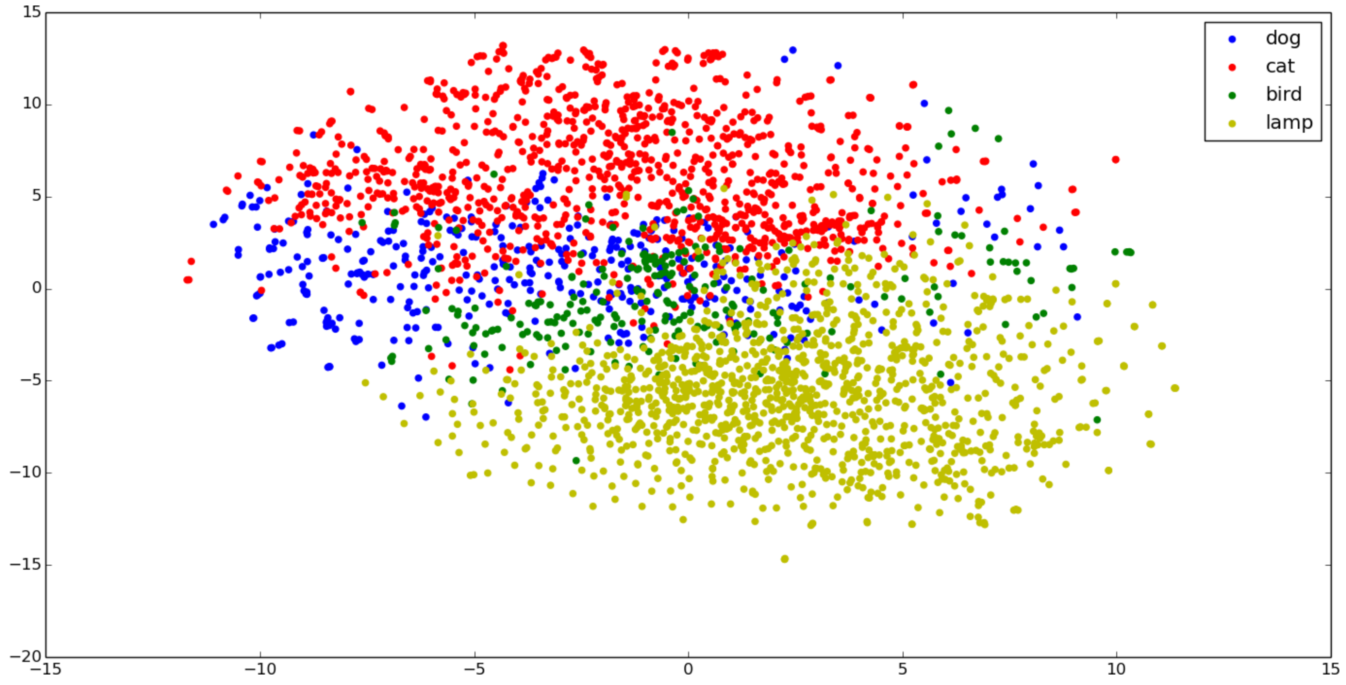


Figure 8. t-SNE representation of layer 5 output (4 synsets)

Table 2. Annotation examples

		Positive			Negative		
Annotation	Images	Total	Train	Test	Total	Train	Test
Adult	50,762	10,798	8,160	2,638	39,964	37,384	2,580
Airplane	71,171	372	171	201	70,799	65,647	5,152
Bus	74,822	71	31	40	74,751	69,370	5,381
Chair	21,576	755	557	198	20,821	17,321	3,500
Gun	6,725	241	76	165	6,484	256	6,228

- Probability estimates

With small modification to the source code is also possible to run the library in parallel to better exploit multi-core CPUs. A small test was performed to estimate the speedup given by the parallel version on a four core machine:

```
$ time ./svm-train dataset
[...]
real     3m22.907s
user     3m21.584s
sys      0m0.536s

$ time ./parallel-svm-train dataset
[...]
real     1m24.002s
user     4m47.576s
sys      0m0.852s
```

As we can see, the computation time went from 202 seconds to 84, giving an approximate speedup of 2.4 times.

2.5 Layers

The unsupervised part of the network is composed by five layers. In Caffe, the functionality of each layer is represented by sub-layers, that are mainly used for convolution, normalization and max pooling. A complete scheme of the unsupervised part is available in figures 9 and 10. The end of each layer is indicated with a blue line.

For example we can see from figure 9 that each of the first two layers is composed by three sub-layers: convolution, max pooling and normalization. As specified in [3], only the first two layers and the fifth one have max-pooling.

For our measurement we decided to extract the data in 6 different points, indicated by red octagons. Their names and sizes are reported in table 3.

From the fifth layer we analyze both the data before and after max-pooling, in order to study the effect of max-pooling on the quality of the features representation.

3. SVM training

In this section we will explain how the SVM model is selected and measurements performed.

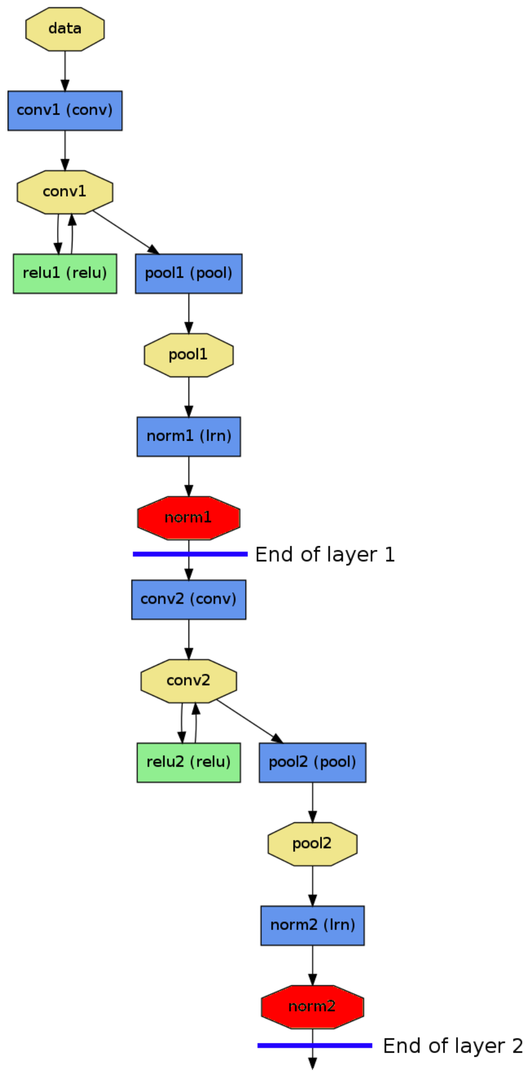


Figure 9. Caffe network structure: layers 1 and 2

3.1 Data extraction

The Caffe library has a Python API that allows to run the network and look neuron weights and values. To extract the data we created a python program that given the layers we are interested in, one annotation file and the picture folder, it directly creates the dataset for libSVM. Example:

```
$ export LD_LIBRARY_PATH=/cuda/lib64:/lib/intel64
```

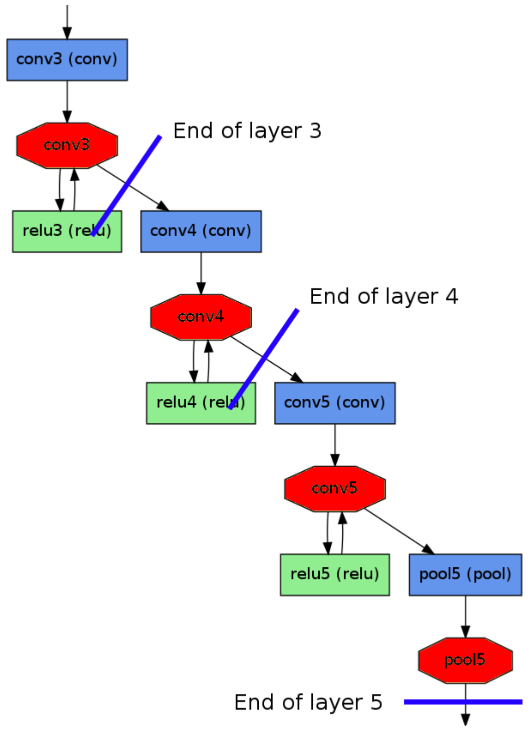


Figure 10. Caffe network structure: layers 3, 4 and 5

Table 3. Extracted sub-layers

Name	Layer	Dimensions	No. of values
norm1	1	96*x*27*x*27	69,984
norm2	2	256*x*13*x*13	43,264
conv3	3	384*x*13*x*13	64,896
conv4	4	384*x*13*x*13	64,896
conv5	5	256*x*13*x*13	43,264
pool5	5	256*x*6*x*6	9,216

```
$ ./create_dataset conv4 conv5 pool5 Adult.ann\
  images/ datasets/
Loading network...
Network loaded in in 1.68s

Images: 9192
Extracting layers ['conv4', 'conv5', 'pool5']:
  Layer conv4: 384, 13, 13 [64896 values]
  Layer conv5: 256, 13, 13 [43264 values]
  Layer pool5: 256, 6, 6 [9216 values]

Starting image elaboration:
  0%    Image: shot1001_20_RKF          0.13s
 10%    Image: shot2101_17_RKF          0.10s
 [...]
DONE

$ ls datasets
  datasets/conv4  datasets/conv5  datasets/pool5
```

As shown, the program reads the annotation file, looks for all the images belonging to that annotation in the specified images folder, and for each image it extracts all the requested layers.

In the usual Caffe implementation the original image originates 10 sub-parts (center and four corners, mirrored and not) and each part is processed by the network, incrementing the computation time by a factor of 10. To speed up the computation, the network description file in Caffe was modified in order to process only the central part of the image.

It will create one dataset for each layer and each dataset will be structured according to the libSVM format:

```
$ cat datasets/pool5
0 110:3.61 112:4.76 ... 9113:4.76 # shot1007_46_RKF
1 1:5.01 12:0.98 ... 8999:4.06 # shot100_17_RKF
0 45:2.34 47:2.05 ... 9110:12.33 # shot100_27_RKF
[...]
```

Each line corresponds to one image. The first value is the label, while the others are the sparse representation of the feature vector, in the form `<dimension_number>:<value>`. The last part is a comment with the name of the image.

These datasets can be directly given to libSVM for cross validation or training. For testing, the same procedure has to be executed on test images in order to create the appropriate datasets.

To reduce the size of the datasets there is a configurable parameter inside the program, called `KEEP_RATIO`, that specifies the probability of an image to be processed (example values: 1 all the images will be considered; 0.1 only 10% of images will be extracted).

To scale the data, libSVM already provides a program that analyze the range of each column in the file and scale it to a value $v \in [0,1]$. Scaling has to be done with the same parameters both on training and test data: libSVM allows to store the scaling factors applied to one dataset, to be later applied to the scaling of another dataset[4].

Data scaling can be a very expensive computation: to scale the norm1 layer data of the complete Adult training set it can take about 20 hours on a Intel i5 3.2GHz machine.

3.2 Model selection

To select the best model three main decisions have to be made:

1. Data scaling
2. Kernel (linear, polynomial, radial basis, sigmoid)
   2.1 Degree in case of polynomial
3. Parameters

First, a reduced dataset was created, to keep the computation time in an acceptable range. We selected one tenth of the Adult annotation and extracted the features from the pool5 layer (the one with the lowest number of dimensions).

We proceeded performing a grid selection for each of the following models:

- Linear kernel

[4]For more information: www.csie.ntu.edu.tw/c̃jlin/papers/guide/guide.pdf

- Polynomial of degree 2
- Polynomial of degree 3
- Polynomial of degree 4
- Radial basis
- Sigmoid

each of them twice, both on scaled and unscaled data, for a total of 12 grid searches.

For every grid search, the performance of the model is evaluated with cross validation, changing the parameters C ($C = 2^{-5,-3...+15}$) and γ ($\gamma = 2^{-15,-12...+3}$), for a total of $1,320$ tests.

The computation was executed in parallel on 20 computers hosted in the EURECOM lab and it took approximately 60 hours to complete.

For every model, the scaled version of the dataset gave approximately 2% better classification performance than the non scaled one. As said in the introduction, the network uses ReLU neurons in the unsupervised part, and they don't have an upper bound for the output signal. Reducing their output range to the interval $[0,1]$ appears to improve the SVM classification score.

In figure 11, page 8, we report the graphical output given by the grid search on scaled dataset.

The top classification scores are:

- 84.95% - Polynomial of degree 3
- 84.80% - Radial basis
- 84.78% - Polynomial of degree 2
- 84.30% - Polynomial of degree 4
- 84.18% - Sigmoid
- 81.97% - Linear kernel

the associated C and γ values are reported in figure 11.

4. Measurements

In this section we perform average precision measurements with different datasets and for each of them we try the best two kernels given by grid search: polynomial of degree 3 and radial basis.

4.1 Balanced dataset - Adult

As first measurement, we decided to use the Adult annotation. To limit the calculation time, we used only 20% of the training data available, while the test was performed on the complete test set. Statistics on the dataset are available in table 4.

Table 4. Balanced dataset - Adult

	Positive	Negative	Total
Train images	1,673	7,518	9,919
Test images	2,638	2,580	5,218

As we can see, the amount of positives and negatives are in the same order of magnitude in this dataset. We will analyze a very unbalanced dataset in the next sections.

4.1.1 Radial basis kernel

We performed the first test using a radial basis kernel with $C = 2$ and $\gamma = 0.0078125 = 2^{-7}$ (values from grid search). We performed the training with libSMV option to get probability estimates. The termination tolerance is the default one of libSVM, $\varepsilon = 0.001$.

Measures of dataset sizes and computation time (on an Intel Core i5-3470 CPU, 3.20 GHz) are shown in table 5, both for radial basis kernel and the polynomial one used in the next section. In this table we also show the average percentage of non zero values for each vector of the dataset. Considering the size of each layer (as reported on table 3, page 6), we can see that the number of non zero values is linearly correlated to the amount of time required by libSVM for the training.

Table 5. Computation statistics

Dataset	Size	Non zero values	RB	Polyn.
pool5	321 MB	$2.5 \cdot 10^3 = 27.7\%$	0.9 h	2.2 h
conv5	545 MB	$4.1 \cdot 10^3 = 9.5\%$	2.5 h	3.3 h
conv4	2.7 GB	$2.0 \cdot 10^4 = 31.5\%$	19.8 h	9.8 h
conv3	2.6 GB	$1.9 \cdot 10^4 = 30.0\%$	16.8 h	12.0 h
norm2	2.7 GB	$2.0 \cdot 10^4 = 47.3\%$	19.1 h	13.4 h
norm1	7.6 GB	$5.8 \cdot 10^4 = 83.6\%$	25.5 h	14.3 h
Total	16.4 GB		3.5 days	2.3 days

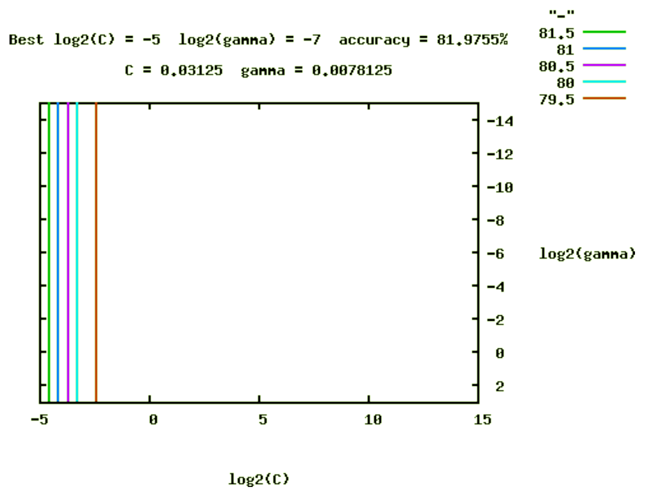
After, the prediction with probability estimates was executed on the test dataset of each layer, using a specific python program that directly extracts average precision given the probability prediction and the test dataset. The output probabilities were used to calculate the average precision with different number of retrieved documents. Results are shown in table 6.

As we can see from the values, the best overall result is obtained in the layer conv5, that is the last unsupervised layer, before performing pooling. As shown in table 5, the pooling reduces the number of non zero values of each vector from 4,100 to 2,500, giving much faster training and prediction time, with very similar results.

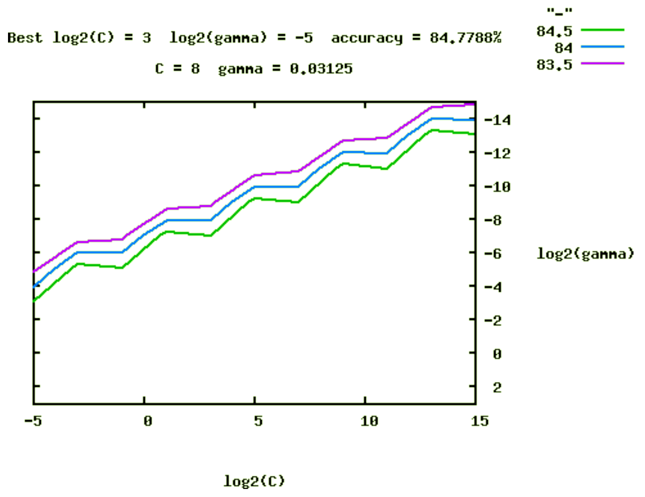
4.1.2 Polynomial degree 3

The same computation was performed using a polynomial kernel of degree 3, with $C = 32$ and $\gamma = 0.0078125 = 2^{-7}$ (values from grid search). As shown by table 5, the learning phase with this kernel takes only about 65% of the time required by the radial basis kernel but it gives comparable results, see table 7 on page 9. This time the layer conv5 gave worse results than the others, and even layers very close to the input data such as norm2 where able to give good results.

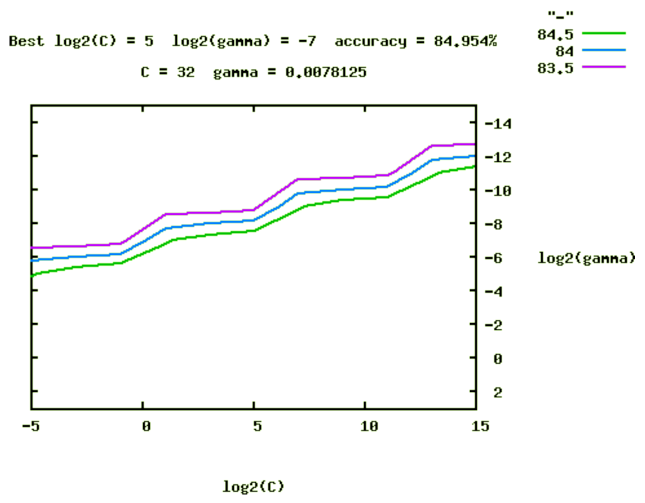
These results may suggest that this class of objects (Adults) is simple enough to allow the SVM to give good probability



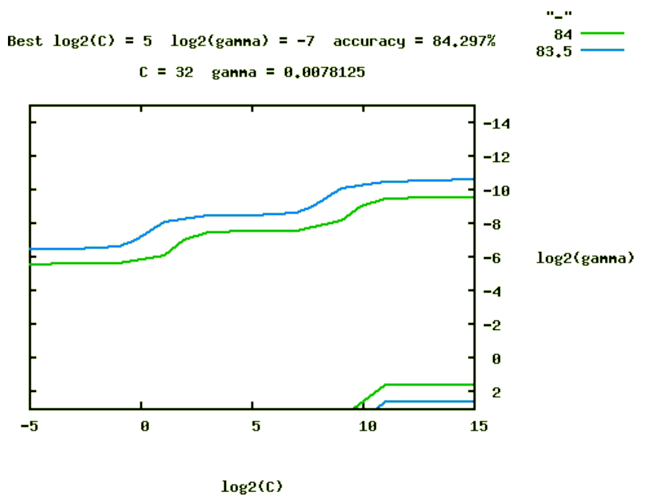
(a) Linear



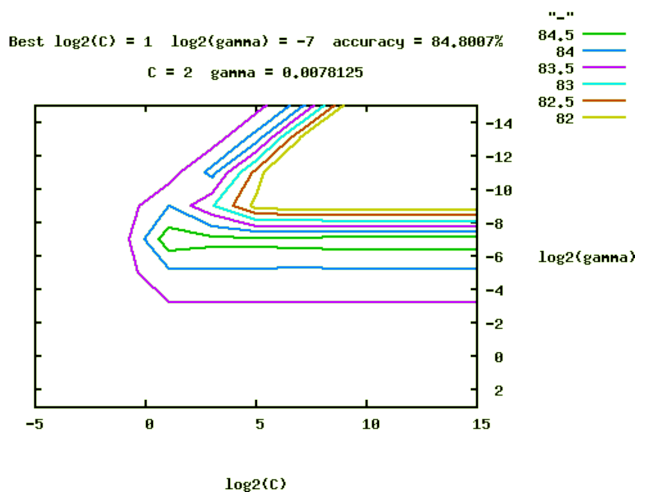
(b) Polynomial of degree 2



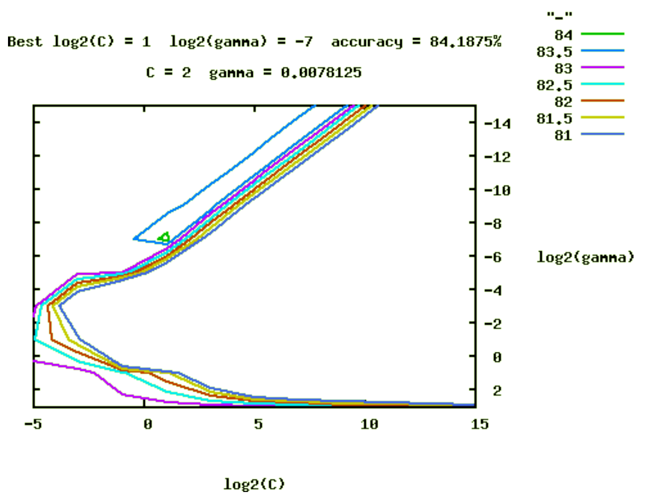
(c) Polynomial of degree 3



(d) Polynomial of degree 4



(e) Radial basis



(f) Sigmoid

Figure 11. Grid search results

Table 6. Average precision on Adult - Radial basis

Layer	$k = 100$	$k = 500$	$k = 1000$	$k = 1500$	$k = 2000$	$k = 2500$
pool5	0.89	0.88	0.87	0.86	0.85	0.83
conv5	0.95	0.91	0.89	0.89	0.86	0.83
conv4	0.96	0.87	0.81	0.79	0.78	0.75
conv3	0.95	0.72	0.70	0.67	0.61	0.57
norm2	0.96	0.75	0.75	0.72	0.72	0.68
norm1	0.88	0.73	0.70	0.55	0.46	0.46

Table 7. Average precision on Adult - Polynomial degree 3

Layer	$k = 100$	$k = 500$	$k = 1000$	$k = 1500$	$k = 2000$	$k = 2500$
pool5	0.83	0.87	0.87	0.86	0.86	0.84
conv5	0.82	0.80	0.81	0.81	0.81	0.81
conv4	0.92	0.91	0.90	0.88	0.86	0.83
conv3	0.93	0.90	0.89	0.87	0.85	0.83
norm2	0.89	0.87	0.88	0.86	0.85	0.81
norm1	0.87	0.80	0.80	0.79	0.78	0.76

estimations even without very abstract data. Notice that the radial basis kernel wasn't able to do that.

4.2 Unbalanced dataset - Flower

To test the same procedure on different data we decided to use the Flower annotation, where the number of positive pictures is one order of magnitude smaller than the negatives. Details in table 8.

Table 8. Unbalanced dataset - Flower

	Positive	Negative	Total
Train images	359	4,013	4,372
Test images	157	6,121	6,278

To compensate for class unbalance, libSVM allows to selectively change the cost C for each class, setting the weights W_i such that for class i the new cost is equal to $W_i \cdot C$. In this way we can specify a big cost for rare classes, giving $W_0 = 1$ and $W_1 = 10$, where 0 are negative samples and 1 are positive ones.

Apart from the class weights, the same kernels and parameters of the previous dataset were used on this dataset.

4.2.1 Radial basis kernel

In this test, the pool5 layer performed significantly better than all the others, even in comparison with conv5, suggesting that even though the data is reduced and simplified by pooling, the SVM can learn a better model on it.

As in the Adult case, this kind of kernel performs much better with abstract data (higher layers). Details in table 9.

4.2.2 Polynomial degree 3

Similarly to what happened with the Adult dataset, also in this test the polynomial kernel was able to give good probability estimation in the lower layers, performing much better in layer norm2 than conv5 for example. Again, this is in contrast to what happened with the radial basis kernel, showing that the type of kernel plays a central role for the performance. Details in table 10.

As we can see comparing these values with the Adult ones, the results are worse. This could indicate that a reduced number of positive elements is not sufficient for the SVM to create a good model, even after setting the weights W_i. A small number of positive elements may not be sufficient to describe the object Flower and together with the complexity of the SVM, this could also cause overfit.

4.3 Few positives dataset - Bus

As last test we tried to build a model based on just 31 positive elements. To do this we took the Bus label and created a small version of the dataset, keeping all the positive images and a small fraction of the negative images, that are much more numerous. More data in table 11.

Table 11. Few positives dataset - Bus

	Positive	Negative	Total
Train images	31	3,456	3,487
Test images	40	5,381	5,421

Like in the previous unbalanced case, we used specific C weights for the classes: $W_0 = 1$ and $W_1 = 100$. The results are reported in tables 12 and 13. The values in this case are very low, even for small recall values. This implies that the positives images are not sufficient to construct a good model of the Bus object. This happens with both radial basis and polynomial kernel.

Table 9. Average precision on Flower - Radial basis

Layer	$k = 10$	$k = 50$	$k = 100$	$k = 500$	$k = 1000$	$k = 1500$	$k = 2000$
pool5	1.00	0.76	0.48	0.46	0.48	0.49	0.49
conv5	1.00	0.37	0.23	0.23	0.27	0.28	0.28
conv4	1.00	0.25	0.14	0.12	0.14	0.16	0.17
conv3	1.00	0.26	0.15	0.18	0.18	0.18	0.18
norm2	1.00	0.24	0.13	0.14	0.15	0.15	0.15
norm1	1.00	0.24	0.12	0.08	0.09	0.09	0.09

Table 10. Average precision on Flower - Polynomial degree 3

Layer	$k = 10$	$k = 50$	$k = 100$	$k = 500$	$k = 1000$	$k = 1500$	$k = 2000$
pool5	0.27	0.46	0.36	0.34	0.38	0.39	0.39
conv5	0.35	0.31	0.19	0.21	0.25	0.26	0.26
conv4	1.00	0.51	0.32	0.34	0.37	0.38	0.38
conv3	0.51	0.46	0.30	0.32	0.35	0.36	0.36
norm2	0.51	0.47	0.35	0.38	0.40	0.40	0.41
norm1	1.00	0.52	0.32	0.30	0.32	0.33	0.34

Table 12. Average precision on Bus - Radial basis

Layer	$k = 10$	$k = 50$	$k = 100$	$k = 500$	$k = 1000$	$k = 1500$	$k = 2000$
pool5	0.000	0.003	0.010	0.019	0.022	0.024	0.025
conv5	0.000	0.001	0.007	0.012	0.018	0.020	0.021
conv4	0.000	0.000	0.003	0.009	0.014	0.017	0.018
conv3	0.000	0.000	0.000	0.004	0.010	0.012	0.014
norm2	0.000	0.000	0.000	0.005	0.009	0.012	0.014
norm1	0.000	0.000	0.003	0.005	0.006	0.008	0.010

Table 13. Average precision on Bus - Polynomial degree 3

Layer	$k = 10$	$k = 50$	$k = 100$	$k = 500$	$k = 1000$	$k = 1500$	$k = 2000$
pool5	0.000	0.002	0.002	0.012	0.022	0.026	0.027
conv5	0.000	0.000	0.001	0.009	0.016	0.020	0.021
conv4	0.000	0.000	0.000	0.000	0.001	0.006	0.009
conv3	0.000	0.000	0.000	0.000	0.000	0.002	0.004
norm2	0.000	0.000	0.000	0.000	0.002	0.003	0.007
norm1	0.000	0.000	0.000	0.000	0.002	0.005	0.005

5. Conclusions

The results show that the quality of recall in relation to the network layer is very dependent on the SVM kernel used. The radial basis kernel gives better results in the last layer of the network (conv5 and pool5), where the data is a more abstract representation of the input image. Instead the polynomial kernel can surprisingly give the same results at very different layers, like pool5 and norm2. Anyway, this can be very dependent on the nature of input image, since recognizing a simple shape can be easily done at the first layers, while a complex structured object may need more layers of abstraction.

In general, the best layer is pool5, the last one of the unsupervised part of the original deep network. Even though pooling reduces the number of dimensions and number of non zero values in the input vectors, the quality of recall does not change. Instead, having a smaller number of values to deal with can help the SVM to generate a better model (and with less computations).

We saw also that is very important to have a sufficient number of positive images in order to create a good model through the SVM.

5.1 Improvements

A lot of improvements can be done. For example, we used grid search based on classification scores, but the best parameters for classification may be sub-optimal for recall. A complete grid search can be performed, to find the best kernel and parameters, using cross validation to directly estimate the average precision on the validation set. This can lead to better results.

Another improvement is to test different datasets based on complexity of the object, shape, color and so on, in order to better understand which kinds of object can be recognized using only raw data and which requires more abstract data.

Part II
Sparse filtering

6. Introduction

In last years unsupervised feature learning has become a good alternative to manually designing feature representations. The biggest problem is that many algorithms are difficult to use because they require to set a great number of hyper-parameters. Sparse filtering is a new feature learning algorithm which requires to set only the number of features to be learned.

We used the sparse filtering algorithm to train the first layers of our network in an unsupervised way. We will show that even with a small dataset it was able to generate filters and results similar to the ones obtained with Caffe.

6.1 Sparse filtering

The key idea of sparse filtering is to avoid explicit modeling of the data distribution: the algorithm works simply by optimizing the sparsity of the feature distribution. The input data is a matrix of floats where each row is a feature and each column represent one example. Sparse filtering wants to obtain 3 properties of the data distribution:

- **Population Sparsity:** each example should be represented by only few active features.
- **Lifetime Sparsity:** each feature should be active only for a few example to be discriminative.
- **High Dispersal:** the distribution of each row should be similar to the one of each other row.

They found that enforcing *population sparsity* and *high dispersal* was sufficient to learn good representations; *lifetime sparsity* is implicitly obtained if you have the other two properties.

6.1.1 Implementation

Let $f_j^{(i)}$ represent the feature value for the i^{th} example where $f_j^{(i)} = w_j^T x^{(i)}$. For a dataset of M examples this gives us the sparse filtering objective function (1):

$$minimize \sum_{i=1}^{M} \|\hat{f}^{(i)}\|_1 = \sum_{i=1}^{M} \left\| \frac{\tilde{f}^{(i)}}{\|\tilde{f}^{(i)}\|_2} \right\|. \tag{1}$$

Our algorithm works multiplying the weight matrix by the input data. The resulting matrix is normalized by row, then by column and finally we sums the absolute values of all cells. This value is used by the solver to check if the solution is improving or not. To minimize the equation (1) we used the scipy method *minimize* with *'L-BFGS-B'* as solver type. It receives as parameter the method described above and calls it several time until it reaches the maximum number of iterations.

The *w* matrix that we obtain represent the input weights that should be assigned to the current convolutional layer to obtain the desired distribution.

Sparse filtering allows us to train each layer separately: using the input images we train the first convolutional layer, then we set its weights and forward all the pictures. The output data of *layer 1* will then be used as input to train the second layer. These operations are repeated for the following layers until the entire network is trained.

6.1.2 Stopping criteria

If any of these three conditions is verified the algorithm is stopped and the current result is returned:

- *Number of iterations:* we set a maximum number of iterations after which the algorithm must terminate even if the convergence condition is not verified.
- *Not improving:* if the improvement between one iteration and the next one is too small the algorithm stops.
- *Convergence:* if the norm of the projected gradient is less than a value provided as a parameter to the *minimize* function the convergence condition is satisfied

In every training attempt we obtained the the first, forth and fifth convolutional layers reached the maximum number of iterations while the second and third layers stopped after about 300 iterations because the norm of projected gradient was less than $1 * 10^{-20}$. This behavior is observed with both 540, 1000 and 3000 input images.

6.1.3 Results normalization

For some layer the weights generated by sparse filtering were of the order of 10^5. This happens because the algorithm minimize only the dot product between the weights and the input data, so the weights can be arbitrarily big as long as the result of the multiplication is small.

Using too big weights has the problem that during the feed-forward phase the intermediate values obtained may be bigger than the maximum value representable using floats on 32 bits. Some of the values of the result matrices were always *nan* which is the value used by *numpy* to show that there was an error computing that value.

Our solution was to normalize the weights in the interval [-1.0 ; +1.0] dividing each value by the maximum number in absolute value present in the matrix. This way the distribution of the weights is not affected and the sparsity characteristics are maintained.

6.2 Patches generation

The examples contained in the feature matrix used as input by the sparse filtering algorithm are patches obtained from the output of the previous convolutional layer (from the input images in the case of the first layer). In the code is possible to specify how many should be taken for each image: they are randomly chosen between all possible patches that can be obtained from that picture.

A patch is a small area of the image which is then flattened to convert it from a 3-dimensional matrix to an array. There are several ways to flatten a matrix: you can for example choose to concatenate the rows or to concatenate the columns. These choices are all valid as long as you use then the corresponding inverse operation to recreate the original matrix when required.

In Decaf the patches extraction and linearization are already implemented in two C++ function *im2col_forward* and *im2col_backward* so we decided to use it. It receives the input data called *padded_data*, the kernel size *_ksize* and the *stride*. The kernel size is the dimension of the patches whose shape will be (ksize, ksize, num. filters). The stride represent how many pixel this area where the patch is extracted should move between one patch and the following one.

Col_data is the output matrix which will contain all the patches flattened and transposed: each column of the matrix is a different patch.

```
wrapper.im2col_forward(padded_data[i:i+1], col_data,
     self._ksize, self._stride)
```

6.3 Decaf

Decaf is a framework that implements convolutional neural networks, with the goal of being efficient and flexible. It was developed by the Berkeley Vision and Learning Center but it is not maintained anymore. Decaf is the predecessor of Caffe: the design pattern is similar and the outputs of the two networks are almost compatible.

The biggest difference is that Decaf is written in python and C++. This means that modify Decaf is easier and faster but the performances of the network are much worse with respect to caffe. Another problem is that there were no complete examples of deep convolutional neural network and the training algorithms were not complete.

One advantage of Decaf is that it is possible to train a network on CPU: you don't need to install the CUDA libraries and have a Nvidia graphic card. The training on CPU is much slower than on GPU but you can execute it on any computer even if it doesn't have a graphic card.

6.3.1 Numpy and Scipy

Decaf relies on the *numpy* and *scipy* python scientific packages: they provide N-dimensional arrays, optimization solvers and optimized linear algebra operations. To obtain the best performances they can be build from source and linked with the *Intel MKL libraries*. Even with this improvements Decaf will still be about 10 times slower then Caffe in the training phase.

7. The network

The network we used has almost the same structure of the one used in the previous part. It is also an implementation of the network developed by Krizhevsky, Sutskevera and Hinton which won the 2012 edition of the ILSVRC-2012 competition.

In the original network they were using two GPUs in parallel (because the memory of one GPU was not sufficient to contain all the data) so they put half of the kernels (or neurons) on each GPU. The two half operations were executed in parallel to increase the network performances and the GPUs communicate only in certain layers to limit the amount of data exchanged between them.

Our code is not executed on GPUs and the RAM memory of our computer was big enough to contain all the data: the maximum amount of memory required is of about 14.5GB using 3000 images and 50 patches per image. Since this parallel execution was not required we substituted the *Group Convolutional Layers* with normal *Convolutional Layers*. This removed the overhead of splitting the data and then merging the results, slightly increasing the performance.

7.1 Dataset

We used the same dataset as the previous part which was extracted by the TrecVid dataset. We executed our experiments on the Adult annotation since it is quite balanced: the ratio of positive image is 25% of all the images with information about the presence of an adult.

Our network requires a long time to train and we were able to use only few thousands of input images. Usually unsupervised learning algorithms are executed on millions of images so the performances of our network would be much worse than Caffe. To compensate this problem we trained the network on images already taken from the Adult dataset. The network will probably overfit, learning human like features, and the precision trying to recognize different types of objects will be much worse. For example we saw that this kind of network was not able to separate images of birds and cats (as will be described in section 8.3)

Unfortunately we were not able to compare the results of a network trained on this dataset and a network trained on generic images since training a network requires almost one week. On the machine that was assigned to us was possible to run only one training at a time because of memory and CPU limitations.

8. Training

8.1 Unsupervised learning

We used sparse filtering to train in an unsupervised way the first 5 layers. One good feature of unsupervised algorithms is that they allow to train the layers separately: this reduces the complexity of the procedure and speeds up the learning phase.

The weight of the first layer are generated using as input the images loaded by the *ImageDataLayer*. Then all the images are feed-forwarded through the first layer: its output is used to generate the patches which will be used to train the second convolutional layer. This operation is repeated for all the other layers until all of them are trained.

The fist layer has smaller filters than the last four: this implies that training the first layer is much faster than the others because the intermediate matrices are much smaller, which implies that the dot product operation is faster. In Table 14 you can see the dimensions of the filters and the training time using 3000 input images and 50 patches per image.

The first two number used to compute the kernel dimension are the dimension of the patches that will be retrieved from the input data and the third value is the number of neurons of the previous convolutional layer (this value is 3 for *conv1* because it represent the RGB components of the image).

Even when the kernel dimension is the same the training times are slightly different: this depends both on the distribution of the data received (if there are many zeros the dot product is faster) and the workload of the computer. We were running several scripts in parallel so the load on the machine (and consequently its performance) was not constant.

We assigned to the forth and fifth layers a different number of maximum iterations: specifically it is one third of the number of iteratins of the first 3 layers. We made this choice since the last two layer have big kernels and would require a even longer time to train. We haven't changed this value for the second and third layer even if they also have big kernels since they converge after 300 iterations.

The feed-forward time represent the total time required to forward the images through the preceding layers and extract the patches. There is a big difference between the times associated with *conv1, conv2* and *conv3* while the last two values are similar to the *conv2* one. This happens because the first two layers are more complex: all layers contain a *ConvolutionLayer* and a *ReluLayer* but the first two contain also a *PoolingLayer* and a *LocalResponseNormalizeLayer*. These two additional layers add some overhead to the feed-forward operation, in particular the normalization phase is computationally expensive.

Table 15 shows the training time for different values of input images and patches extracted. The number of iterations is fixed at 1800 and also the number of patches per image is almost constant at about 50. The time required to train the network with sparse filtering is almost linear in the number of images. It is 2 days with 1000 images and it is 6 days with 3000.

8.2 Supervised learning

The network described in section 7 includes 8 layers: 5 unsupervised and 3 supervised. The last three layers should be trained with the *back-propagation algorithm*. This phase would allow to use this network to recognize objects inside the images.

In the training phase the *ImageDataLayer* outputs both the image data and the associated label. This label is then retrieved by the *KLDivergenceLossLayer* which computes the difference between the predicted label and the correct one.

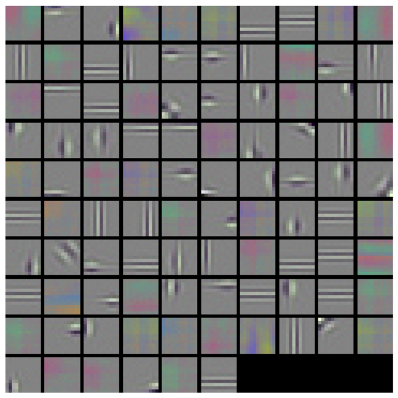


Figure 12. First layer filters

During the execution phase the network outputs an array of float which represent the probability that object_i appears in the image. From this array the 5 highest probabilities are extracted and printed on the screen.

For our experiments we used only the unsupervised part. For each image sent through the network, each layer writes its output in a file with the format required by libSVM (part I section 3.1). When all the images were processed we scaled the output files using the software *svm-scale* (already provided by libSVM) and then we trained the Support Vector Machine using the training scaled data. The model created is then used to evaluate the validation images.

Since this operation must be executed for all input files we created a bash script to automate these operations. The scale operation is really long (takes several hours even for just 3000 images) so the scaling of the train and test data is executed in parallel. In Table 16 you can see the average scaling time for the training and validation data: since they are executed in parallel the total time is just equal to the biggest one which is just about the 65% of the time required executing the two operations sequentially.

For the training phase we used 1000 images randomly chosen inside the train folder and for the test phase we used 5218 images randomly chosen inside the test folder.

8.3 Birds and cats

We downloaded from ImageNet pictures relative to the synsets cat and bird. We trained our network using sparse filtering on 540 images randomly chosen between them. Then we executed the network on 600 images (different from the training ones) writing the output of the fifth layer in a file.

This output file was the processed by a modified t-SNE implementation to plot the values in 2D. As we can see from

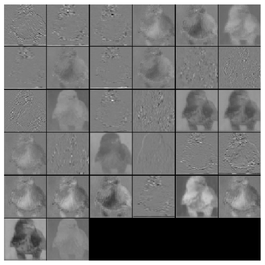
Table 14. Training time per layer

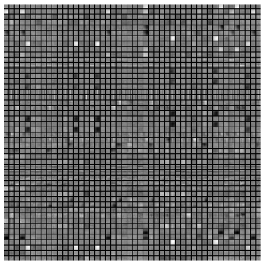
Layer	kernel dimension	Feed-forward time	Iterations	Training time
conv1	11*11*3 = 363	0h 1m 22s	1800	5h 33m 20s
conv2	5*5*96 = 2400	0h 30m 3s	327	15h 39m 20s
conv3	3*3*256 = 2304	1h 31m 54s	328	21h 1m 10s
conv4	3*3*384 = 3456	1h 39m 45s	600	41h 55m 0s
conv5	3*3*384 = 3456	2h 2m 55s	600	34h 51m 40s

Table 15. Total training time (unsupervised only)

Number of images	Number of patches	Max. iterations	Training time
540	60	1800	0d 20h 16m 53s
1000	50	1800	2d 2h 23m 00s
3000	50	1800	6d 2h 30m 00s

(a) Input image


(b) First layer output


(c) Second layer filters

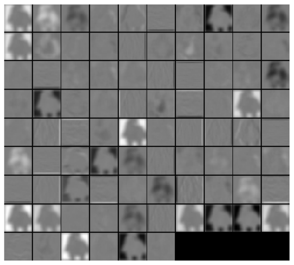

(d) Second layer output

Figure 13. Processing of an image taken from ImageNet

Table 16. Scaling time

Phase	folder	Num. of images	Average scaling time
Training	train	3000	3h 46m 53s
Validation	test	5218	7h 00m 52s

Version	Num. of images	Total time	Improvement
Sequential	7218	11h 10m 45s	0%
Parallel	7218	07h 23m 52s	-35%

Picture 14 even with few training images the network is able to find useful features that allow to correctly separate the images of birds and cats. Obviously the division is not perfect since we trained the network with just 540 pictures and we didn't execute the back-propagation algorithm to fine tuning the weights.

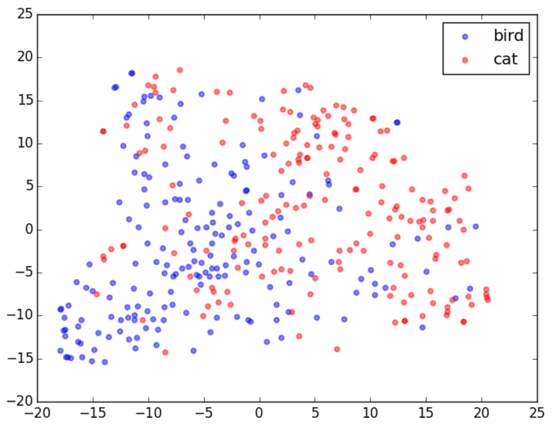


Figure 14. Example images from ImageNet.

9. Measurements

We used libSVM to classify the data generated from the neural network using the Adult annotation. We considered only the images which had an information associated about the presence or absence of an Adult (indicated with a P or a N in the Adult.ann file). We used as model the *polinomial of degree 3*.

In Table 17 are reported the results using the model generated training the network with 1000 images. These values are lower than the one resulting from the same experiment using Caffe but have similar characteristics. Like the previous results also here the precision is almost constant on all layer with the same k. For example with k=2000 the values are all close to 0.60, they are not decreasing in the lower layers.

In Table 18 you can see the results obtained using the network trained on 3000 images. The results are slightly better then the previous ones but still worse than Caffe's results. One strange thing is that the precision here is higher in the lower layers (close to *conv1*) than in the upper ones.

This may be caused by the decision to reduce the number of iterations for the last layers.

10. Considerations

The biggest problem of this network is its performances. It would be impossible to train the network with 100.000 images or millions since it would require months.

One possibility would be to build from source numpy and scipy and integrate the Intel MKL libraries. This can improve the network speed of about 2 times. Another improvement would be to use a profiler to find the bottlenecks in the code and improve those algorithms (or at least check that there are not useless operations).

The results obtained with this network are not particularly good but in our opinion if it would be possible to train it with more images and for a longer time the results will improve greatly. However even from these experiments is possible to see that the sparse filtering algorithm is converging and is able to obtain good features from the input data which can be used to classify them.

The filters of the first layer are also really similar to the one present in Caffe.

Table 17. Results on Adult annotation - 1000 images

Layer	$k = 100$	$k = 500$	$k = 1000$	$k = 1500$	$k = 2000$	$k = 2500$
pool5	0.69	0.71	0.68	0.66	0.62	0.60
conv5_neuron	0.75	0.68	0.63	0.59	0.56	0.54
conv4_neuron	0.61	0.66	0.62	0.59	0.56	0.54
conv3_neuron	0.73	0.73	0.69	0.66	0.63	0.61
norm2	0.72	0.75	0.73	0.72	0.69	0.65
norm1	0.76	0.75	0.74	0.72	0.61	0.68

Table 18. Results on Adult annotation - 3000 images

Layer	$k = 100$	$k = 500$	$k = 1000$	$k = 1500$	$k = 2000$	$k = 2500$
pool5	0.79	0.68	0.71	0.70	0.67	0.69
conv5_neuron	0.77	0.66	0.69	0.69	0.66	0.65
conv4_neuron	0.77	0.68	0.71	0.69	0.67	0.66
conv3_neuron	0.80	0.70	0.73	0.71	0.69	0.69
norm2	0.81	0.72	0.75	0.69	0.72	0.71
norm1	0.83	0.72	0.76	0.71	0.73	0.72

Part III

Modification of the fully-connected layers

11. Introduction

In this section we are looking at the last three layers of the network. This part of the network takes the output from the unsupervised network and feeds it into three fully connected layers. After the last layer there is a classifier, which maps the output to the corresponding labels.
Since the layers are fully connected, the training process is rather time consuming. In this section we show an approach to reduce the number neurons participating in the calculation. This is achieved by grouping similar layers together and include the group information into the training process.

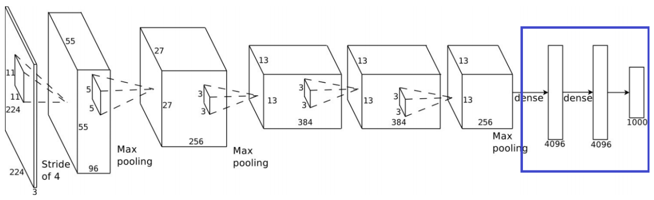


Figure 15. ImageNet network structure

11.1 Reducing the number of involved neurons

As described in the introduction, we to include some additional information about the label into the training process to reduce the number of connections in the last three layers.
Including the labels into the process, produces kind of a chicken and egg problem. To solve this problem, we decided to group the labels. This reduces the problem to finding the corresponding group Id instead of labels. There are different ways of grouping the labels. This is described in the following paragraph in more details.
However we still want to implement a fully connected layer. As a consequence we reduced the number of neurons involved per group. We implement a sliding windows approach to select only certain neurons. In our implementation, we choose to move the windows for each group with 1/4 of the windows size.
The number of fully-connected neurons is reduce to the windows size.

11.2 Grouping of labels

How to group the labels is the choice of the designer of the network. The easiest way to group the labels is in alphabetical order. However this leads to really heterogeneous groups.
The better approach is to use clustering to assigns the labels to different groups, based on the similarity of the output from the unsupervised part. In our simulation we clustered the input data by hand. This was possible, because we used a data set of characters in the alphabet, which follow a specific characteristic. This can be easily detected by humans.
Get Group Id for new Data
We propose two different approaches to determine the group Id for new data:

- **Group selecting by neural network:** Since the Group Id's can be treated as labels, we simply can reuse the existing code with a fixed offset for the sliding window. The advantage of this approach is, that we do not need to store an additional data structure. Instead we only reduce the number of available groups by one.
- **Determine group Id with clustering:** In the case the Group Id's were created by the help of a clustering algorithm, a new data sample, can be assigned by the clustering algorithm.

11.3 Implementation

There are two different specification of the modified fully-connected layers. One can be used on top of the unsupervised part of caffe/ImageNet and the other was created for an artificial character database by the university of Torino [4]. The two specification differ in the number of neurons used per layer. The table below shows the total number of neurons and the neurons by a single group.
The number of used neurons per layer is a tuning parameter and can be set by the designer of the network.

11.4 Data set

Even though there exists an implementation which is running on top of Caffe, we run our calculation on the artificial character database [4]. The reason for this decision, is the really slow convergence of our CPU implementation.The data set we use for the measurement describes ten capital letters of the English alphabet. For each class the data set contains 100

Table 19. Number of neurons for ImageNet and artificial character database implementation
(total , per group)

	Input	Layer 1,2	Layer 3
ImageNet	9096	4048 , 2048	1000 , 1000
Test images	48	120 , 60	400 , 280

learning and 500 testing samples. The grouping for the given data set is listed in table 20

Table 20. Data Set with Group Id

	Group Id	Training Data	Test Data
A	0	100	500
C	30	100	500
D	30	100	500
E	45	100	500
F	45	100	500
G	15	100	500
H	15	100	500
L	30	100	500
P	60	100	500
R	60	100	500

11.5 Measurement

In table 21 you can see the results after training the network and testing the network with 500 testing samples per alphabetic character. This table hold for the assumption, that the group Id is computed correctly.

Table 21. Training Results

	Group Id	Mistakes	Total	Number of group members
A	0	0	500	3
C	15	14	500	2
D	15	0	500	2
E	0	12	500	3
F	30	0	500	2
G	0	2	500	3
H	45	13	500	1
L	30	0	500	2
P	60	59	500	2
R	60	6	500	2

11.6 Conclusion

After working with the adopted version of the fully-connected layers, multiple drawbacks could be determined. A selection is listed below:

- **Grouping process:** The process to assign each label to a group needs a clustering process. Finding an optimal solution is an NP-Problem.
- **New tuning parameters:** The deep learning algorithm already suffer form many tuning parameters, which needs to be set accordingly to the problem. The new introduced parameters, increase this problem.
- **Sliding windows approach:** Since we use a sliding windows to determine which neuron are participation in the calculation, a wrong detected group Id leads to a wrong output.
- **Less clear layer structure:** Because we need to determine from the output of the unsupervised part first the group Id and afterwards run the layers to classify the image, the network structure does not follow any more a straight-line relationship

As a conclusion we want to note, that the modified version of the fully-connected layers is a working solution. However the benefit for training process of deep learning networks has to be put into questions.

References

[1] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012.

[2] Yangqing Jia. Caffe: An open source convolutional architecture for fast feature embedding. `http://caffe.berkeleyvision.org/`, 2013.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[4] Università di Torino Marco Botta, Dipartimento di Informatica. Artificial character database. July, 1992.

[5] L.J.P van der Maaten and G.E. Hinton. Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research*, 9: 2579–2605, Nov 2008.