TP 1 (a),(b) et (c) : Reconnaissance des formes pour l'analyse et l'interprétation d'images

Réalisé par :

Gaël Marec et Léo Hein

Table des matières

The goal of these first practical works is to prepare the development of an initial image classification model by producing an algorithm that is able to classify those from the dataset 15-Scenes, which contains 4485 images belonging to 15 scenes categories (e.g. kitchen, bedroom, street, etc.). We are first going to encode small patches of an image separately using the SIFT algorithm, creating local visual descriptors. Then, we will build a visual dictionary using visual descriptors representing recurrent patterns thanks to a clustering method. After that, we will aggregate the SIFT of each image with a Bag of Word method expressing in a compact way an image. Finally, we will classify each image represented by its Bag of Word thanks to a Support Vector Machine.

1 Images descriptors with SIFT and Bag of Words

1.1 SIFT

1.1.1 Computing the gradient of an image

In this first section, we would like to compute the gradient at each pixel of an image. We will do so by approximating the derivatives by finite differences thanks to the application of two convolution Sobel kernels M_x and M_y, one for each dimension. We then can compute the norm and the direction of the gradient.

We first write the function $compute - grad$ which computes the gradient of the input image and returns I_x and I_y.

```
def compute_grad(I):
    Ix = conv_separable(I, hy, hx)
    Iy = conv_separable(I, hx, hy)
    return Ix, Iy
```

We then write the function $compute - grad - mod - ori$ which returns the gradient norms G_n and the discretized orientations G_{disc} of the input image.

```
def compute_grad_mod_ori(I):
    Ix, Iy = compute_grad(I)
    Gn = np.sqrt(np.square(Ix) + np.square(Iy))
    Go = compute_grad_ori(Ix, Iy, Gn)
    return Gn, Go
```

(1) The two used kernels are separable and we have $h_x^T = (-1, 0, 1)$ and $h_y^T = (1, 2, 1)$ so that $M_x = h_y h_x^T$ and $M_y = h_x h_y^T$

(2) We now have $I_x = I * M_x = I(h_y h_x^T)$ and $I_y = I * M_y = I(h_x h_y^T)$. This decomposition of the convolution kernel can help reducing the computational complexity of matrix operations used to calculate the gradient of the image. Indeed, instead of doing one convolution with 9 multiplications, we do two convolutions with 3 multiplications each to achieve the same effect.

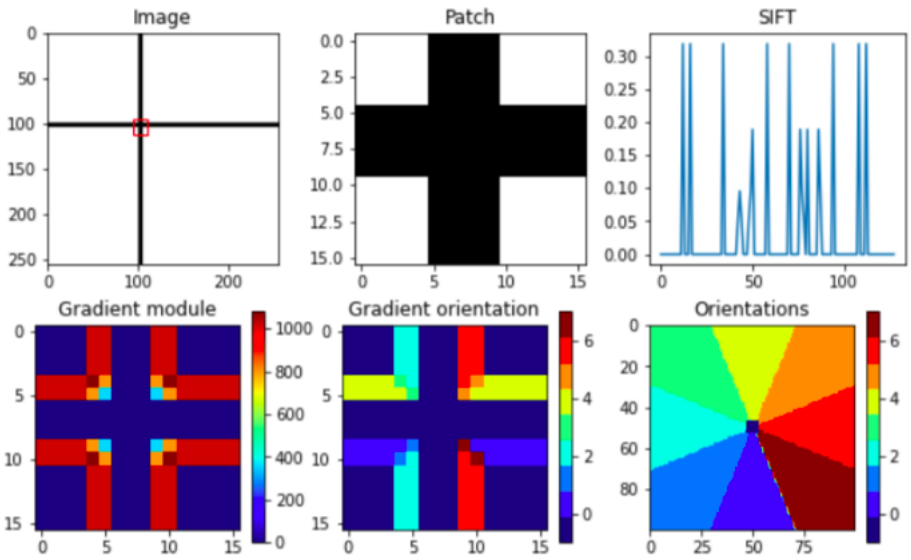
1.1.2 Computing the SIFT representation of a patch

In this section, we compute the algorithm that produces the SIFT representation of a patch P (16x16 pixels) of the image thanks to the previously obtained norm and direction gradient matrix. The patch P is going to be decomposed in 16 sub-regions of 4x4 pixels, each encoded as a vector of size 8. To do so, we'll use the SIFT descriptor which consists in computing a histogram of gradient directions. The complete patch will be described by a vector of size 128 which will be the concatenation of all regions encoding.

We first create the function used to compute the histogram of a patch sub-region. This histogram gathers the gradient norms present in the patch for each discrete direction defined earlier.

```
def compute_histogram(g_n, g_o):
    hist = np.zeros((8))
    n = len(g_o)
    for i in range(len(g_o)):
      for j in range(len(g_o[0])):
        goij = g_o[i,j]
        if goij != -1:
          hist[goij] += g_n[i,j]
    return hist
```

We then complete the given function $compute-sift-region$ so that it can return the SIFT representation of a patch and test it manually and visually thanks to the function display-sift-region.



(3) We weight each patch by a Gaussian mask to give more importance to the centre of the patches. Indeed, as we browse the image, we only space the different 16x16 pixels patches by 8 pixels, meaning that we may have the same information multiple times. To counter this overlap, the Gaussian mask limit the learning of information near the

borders by focusing on the center of the patch. It is also used to smooth the data, which is helpful for edge detection as it helps reducing the level of noise and details in the image, which then improves the result of the following edge-detection algorithm. Indeed, it helps for the invariant scale property.

(4) A discretization of the gradient orientations is necessary. First, if the orientation is not discrete, we may have computational issues, as the vector given back by the histogram is of size the total number of directions. Besides, we want descriptors to get consistent data and features from the image ; by discretizing the gradient orientation, we gather data toward each direction, which is a better description of a feature than one too noisy/specific because of its huge amount of orientations. Even if it may imply a loss of information about the direction (it approximates the direction of the gradient in a way), it helps computing a consistent and global sift vector easily for the image while still gathering the most important part of the data and creating the orientation invariant property.

(5) There are several post-processing steps :

- Returning 0 when $||P_{enc}||$ is too small allow to simplify parts of the image where we observe non significant color variations.
- The two steps of normalization allow to have a better understanding of P_{enc} by having the possibility to compare different P_{enc}. The 0.2 threshold helps reducing the impact of the change in luminosity on our descriptor.

(6) The SIFT method is quite intuitive because it mimics the natural way a human looks at an image. Indeed, the human interprets the image first by colour contrast to understand the points of interest. Besides, SIFT can robustly identify objects even among clutter and under partial occlusion, because the SIFT feature descriptor is invariant to uniform scaling, orientation and illumination changes.

(7) Several examples are given but we will focus on the one we tested our functions on. The first two figures show the entire image and the patch for which we are computing the SIFT representation and the last figure shows the discretization of the orientations. The third figure represents graphically the values of the computed sift. We can see that it has a consistent length of 128 and different values. These are due to the sub-regions chosen by the algorithm that may give back different histograms, because of the different "amount of directions" inside each sub-region. The fourth and fifth graphs are the resulting gradients norm and orientations of our patch. We can see that the colors are consistent with the orientation discretization on the fifth graph (toward the white color) and that the norm of the gradient is uniform excepted in the corners of the cross, which is coherent.

1.1.3 Computing SIFTs on the image dataset

In this section, we compute the SIFT representation of an entire image based on the SIFTs of each one of its patches. To do so, we must first choose center points whose surrounding will be used to compute these SIFTs. One method is to densely

sample them, in our case, we will sample one 16x16 patch every 8 pixels. The function *compute_sift_image* below takes care of this work.

```
def compute_sift_image(I):
    x, y = dense_sampling(I)
    im = auto_padding(I)
    m = gaussian_mask()

    # Here, compute on the global image (norm, gradients)
    Gn, Go = compute_grad_mod_ori(I)
    sifts = np.zeros((len(x), len(y), 128))
    for i, xi in enumerate(x):
        for j, yj in enumerate(y):
            GnR = Gn[xi:xi+16, yj:yj+16]
            GoR = Go[xi:xi+16, yj:yj+16]
            sifts[i, j, :] = compute_sift_region(GnR, GoR, mask=None)
    return sifts
```

1.2 Visual Dictionnary

In this section, we are going to compute a visual dictionary thanks to all SIFTs descriptors extracted from our dataset. Words in this dictionary are SIFT-like descriptor which when combined aims at representing as correctly as possible all the SIFTs but using a limited amount of words. Thus, those words will be frequent patterns in the extracted SIFTs.

We first use the function $compute - load - sift - dataset$ to obtain a list where each element is a list of the SIFTs of a single image. We then complete the function $compute - visual - dict$ so that it computes the visual dictionary. We mainly add a clustering method, K-means in our case, so that we can create a defined number of clusters representing the different words of the dictionary. We arbitrarily choose 1000 clusters, but it is possible to optimize this parameter. Below are the lines of codes added to the function.

```
kmeans = KMeans(n_clusters=n_clusters, n_init=n_init, verbose=verbose).fit(sift)
vdict = kmeans.cluster_centers_
vdict.append(np.zeros((128)))
```

After that, we use the function $compute - or - load - vdict$ to compute the clusters on all the SIFTs. We then analyse the clusters of our dictionary by looking at the 100 closest patches to a particular cluster thanks to the $display - images$ function.

(8) The visual dictionary is made of recurrent features found among the training dataset. It kind of represents different characteristics such as borders, edges, or plane colors present inside an image. With this tool, we can describe an image by a set of features just as the height or weight of a person we want to describe physically, which is necessary if we want to compare and classify the different images. For example, a room or a kitchen may not have the same features and that's what enable us to differentiate them, whereas it doesn't make much sense to compare raw pixels that are too noisy.

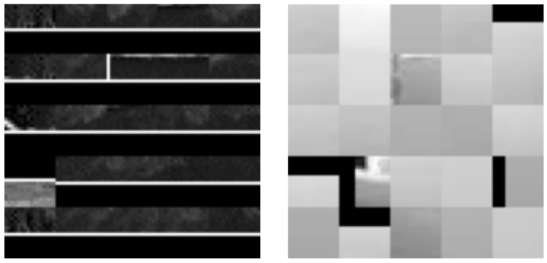
(9) We want to find the set : $\underset{c}{argmin}(\sum_{i=1} ||x_i - c||_2^2 = f(c))$.
$L : (x, x) \mapsto ||x||_2^2 = x^T x$ has for differential $D_x(L).h = 2x^T h$. We deduce that the gradient of f is an argument of the minimum (because f is convex as a sum of convex functions) :$\nabla f(c) = 2\sum_{i=1} (x_i - c) = 0$ and $\boxed{c = \frac{1}{n}\sum_{i=1} x_i}$ which then is the barycenter of the x_i.

(10) Several methods exist such as the elbow method, the average silhouette method or the gap statistic method. For example, the elbow method consists of plotting the explained variation as a function of the number of clusters, and picking the elbow of the curve as the number of clusters to use. We choose the point where diminishing returns are no longer worth the additional cost. Indeed, increasing the number of clusters will naturally improve the fit, but that at some point this is over-fitting, and the elbow reflects this.

(11) We do not directly create a visual dictionary from raw image pixels because they are too sensitive to noise. If there are few blur pixels or reflections that enlighten particular pixels, it may create inconsistent words in the visual dictionary.

(12) We can see on the last figure that by selecting a number of 100 patches, some are very different, but we still see patterns among all of them. However, if we select

now 25 patches, the similarity is striking, which is what we were looking for : the more the descriptors are near the cluster centre, the higher the similarity is.



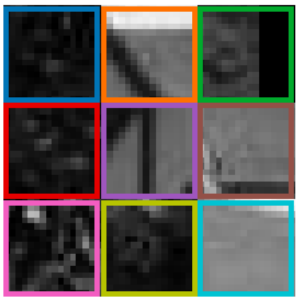
1.3 Bag of Word

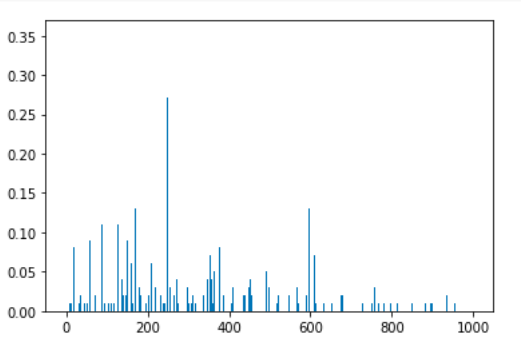
In this section, we want to obtain a numerical representation of each image, which will help us classify other images. We will use a BoW method that will summarize all the local descriptors of an image into a global descriptor with the help of the visual dictionary.

To do so, we wrote the $compute - feats$ function below. For each sift, a one hot encoded vector is created, having a 1 value on the index of its closest word of the visual dictionary. We then sum up all these vectors to create a final vector, characterising the entire image. It is the BoW representation z of the image.

```
def compute_feats(vdict, image_sifts):
    # flatten sifts
    sifts = image_sifts.reshape(-1, 128)  # (N, 128)
    feats = np.zeros(vdict.shape[0])

    for sift in sifts :
      ind_min = np.argmin(distance_matrix(vdict, sift.reshape(1,128)))
      feats[ind_min] += 1

    feats = feats.reshape(1,-1)
    feats = normalize(feats, norm='l2')
    feats = feats[0]

    return feats
```

Finally, we visualize our BoW on the image below.





(13) We have $h[j] = 1$ if the descriptor x_i is the closest to the cluster j. When we sum the h_i we count the number of descriptors affected to each cluster. Concretely, we establish the occurrences histogram of the visual dictionary words composing the picture.

(14) On the picture above, are shown the main 9 features found on the image. There is for example the sky/roof limit feature described by the red squares or the beams on the facade represented by the purple ones. Even if some features are well understood by the algorithm, we can see that it is not perfect as the red squares also consider a part of the grass because of the shadows. Moreover, the algorithm has difficulties separating the trees from the grass as shown by the orange and green squares. The graph under the picture represents our BoW representations. It is normalized but the higher the pic is, the most recurrent the feature is on our picture.

(15) A main advantage of $K - N - N$ is its simplicity of implementation and its rapidity to compute consistent cluster centers, which is what we want in this project. Even if it is one of the simplest learning algorithms, it still give good enough results. We could also use a multilayer percepron to use multiclass classification. The benefits of neural networks being their universality and the possibility to approximate non linear functions. We could also make use of the distance separating the sift to its closest cluster. There are besides a lot of different clustering algorithms, such as DBSCAN for example.

(16) The sum of h_i is a simple function enabling to us to gather the occurencies histogram but it can't distinguish two permutations of distinct $(h_i)_i$. We can then think about weighting the sum $\sum \gamma_i h_i$ ($\gamma_i \in \mathbb{R}$) to give more importance to some descriptors compared to others and so counter this issue.

(17) The normalization L_2 reduces the influence of gradients that are too important (for example a white spot of light on the picture). Its a normalization with respect to the contrast. We could also normalize L_∞. With such normalization, we only observe the direction of high contrast on when there is one (the other directions being nearly cancelled by the normalization).

2 Learning a SVM classifier

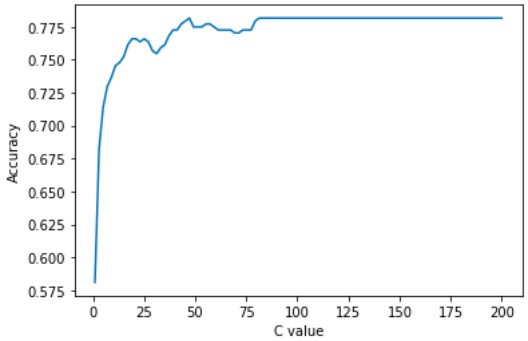
In the previous sessions, we successfully transformed a RGB image into a compact representation that is more suited to image classification. Now, each image is encoded as a BoW vector describing the presence of the recurrent patterns we learned from our train set of images. The last part of this work will be about correctly classifying the images with respect to its category, among the 15 ones of our dataset. To do so, we will use a typical classifier : a support vector machine.

We first Split the dataset into train/validation/test with the following distribution 70/10/20.

We then learn a SVM with scikit-learn on the train set with different values of C. We evaluate our SVMs on the validation set to find the optimal value of C and reevaluate on the test set to obtain the final performance. The script below is calculating the accuracy following this method for all other hyperparameters fixed.

```
C_values = np.linspace(1, 100, 100)
accuracies = []

for c in C_values:
  print(f'Training SVM with C={c}')

  clf = SVC(C=c, kernel='rbf')

  # Fit on sub-train set
  clf.fit(X_train, y_train)

  # Evaluate on val set
  pred = clf.predict(X_validation)
  acc = accuracy_score(y_validation, pred, normalize=True, sample_weight=None)
  accuracies.append(acc)

  print(f'--> accuracy = {acc}')
  print(accuracies)

plt.plot(C_values, accuracies)
plt.show()
```

We first plot the curve of the accuracy according to the C value.

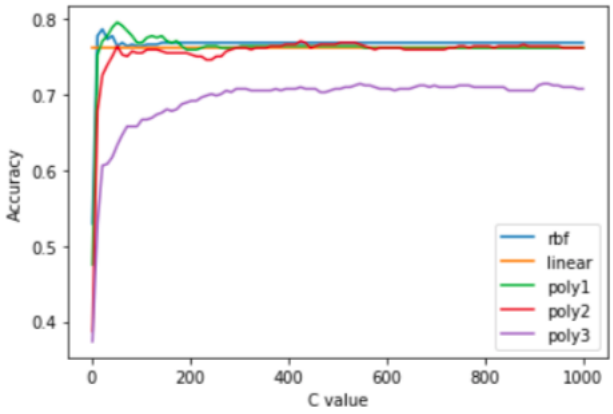


We then find the best C and calculate the accuracy on the test set.

```
bestc = accuracies.index(max(accuracies))
clf = make_pipeline(StandardScaler(), SVC(C=bestc, kernel='rbf'))
clf.fit(X_train, y_train)
predf = clf.predict(X_test)
accr = accuracy_score(y_test, predf, normalize=True, sample_weight=None)
print(acc)
```

We obtain an accuracy of 0.7815 with a C value of 46, a RBF kernel and a scaled gamma.

(18) In order to tune our model,we first grid-searched for each kernel to find the best C value.



Among those five kernel, we can see that the best performances imply either the RBF kernel or the polynomial kernel of degree 1 for "small" C values. Besides, the polyomial kernel of degree 3 seems to be inadapted to this problem compared to the others. The best accuracy in the training set is obtained with the poly1 kernel for a C value of 51,45 and is equal to 0.7950. The accuracy of this SVM is of 0.7604 on the test set.

However, as the computational time of the SVC fitting is quite important, browsing each combination of hyperparameters takes too much time. One of the solutions is to select random combinations of hyperparameters to evaluate them with the same method as before. To do so, we use the *sklearn.RandomizedSearchCV* function. In order to apply the same method as before, we concatenate the validation and train sets as the function will cross-validate the models to get the best hyperparameters.

Eventually, we will test the resulting combination on our test set to get the final accuracy.

```
Cs = np.linspace(10,100,100).tolist()
Degrees = [1, 2, 3, 4]
Kernels = ['linear', 'rbf', 'poly']
param_grid= {'kernel': Kernels, 'C': Cs, 'degree': Degrees,
             'decision_function_shape': ('ovr', 'ovo')}

# Find the Hyperparameters
X_t, y_t = np.concatenate((X_train, X_validation), axis=0),
np.concatenate((y_train, y_validation), axis=0)
clf = SVC(gamma='scale')
sh = RandomizedSearchCV(clf, param_grid, cv=5, n_iter=10).fit(X_t, y_t)
```

```
pd.set_option("display.max_rows", None, "display.max_columns", None)
df = pd.DataFrame(sh.cv_results_)
df = df.sort_values('mean_test_score')
print(df[['param_C', 'param_decision_function_shape', 'param_degree',
          'param_kernel', 'mean_test_score', 'rank_test_score']])
print(sh.best_params_)

# Make the predictions
y_pred = sh.predict(X_validation)
acc = accuracy_score(y_validation, y_pred, normalize=True, sample_weight=None)
print(acc)
```

We show a ranking of some of the best combinations found by the Randomized Search among all the combinations made possible by our parameter Grid. Obviously, it is very likely that a better combination exists, but the one selected here should propose high enough performances.

```
85      84                          ovr             2          rbf         0.727622               15
1       35                          ovo             4          rbf         0.728574               13
93      35                          ovr             3          rbf         0.728574               13
48      43                          ovr             1          rbf         0.728893               12
35      74                          ovr             2          rbf         0.728895               10
51      73                          ovo             1          rbf         0.728895               10
52      33                          ovo             3          rbf         0.729211                9
18      42                          ovo             2          rbf         0.729212                7
79      42                          ovr             1          rbf         0.729212                7
7       40                          ovr             3          rbf         0.729212                6
41      68                          ovo             3          rbf         0.729850                5
83      50                          ovo             1          rbf         0.730167                3
54      50                          ovo             3          rbf         0.730167                3
71      62                          ovo             2          rbf         0.732080                2
60      61                          ovo             4          rbf         0.732080                1
{'kernel': 'rbf', 'degree': 4, 'decision_function_shape': 'ovo', 'C': 61.0}
0.7319819819819819
```

After analyzing the resulting table we can draw some qualitative conclusions about the quality of the different hyperparameters. With this table, it seems that the RBF is unquestionably the best kernel for our problem, which is not exactly what we saw with the last graph. Besides, while the best results present an ovo decision function shape, the difference in performance with the ovr shape is not significant. In addition, the best C value seems to really depend on the combination of the other hyperparameters. This method needs a lot of time too to find a good combination but may be faster than a grid-search. An improvement may be to determine a smaller range of good C values for each kernel ans then apply a randomized-search.

(19) The hyperparameters we decided to focus on are the C value, the kernels, the shape of the decision function, and the gamma value.

The parameter C determines the number and severity of the violations to the margin and the separation hyperplane of our SVM that we will tolerate. If C = 0, then there is no budget for violations to the margin, this is called a hard-margin. Otherwise we talk about soft-margin. As C increases, we become more tolerant of violations to the margin, and so the margin will widen, the classifier may have low bias but high variance, and conversely.

The kernels allow us to go further the linear classifiers, being the standard ones, by generalizing the inner product used to determine the classifier function. We can for example use polynomials kernels in order to get rid of linear separations. There also are the RBF or sigmoid kernels for example. The degree hyperparameter allow us to choose a degree for a polynomial kernel.

There are two shapes of decision functions we can implement. The one vs one decision function and the one vs rest decision function. The OvR strategy involves splitting the multi-class dataset into multiple binary classification problems. A binary classifier is then trained on each binary classification problem and predictions are made using the model that is the most confident. Unlike OvR, the OvO approach splits the dataset into one dataset for each class versus every other class. Although the OvR approach cannot handle multiple datasets, it trains less number of classifiers, making it a faster option and often preferred. On the other hand, the one-vs-one approach is less prone to creating an imbalance in the dataset due to dominance in specific classes.

Finally, the gamma parameter intuitively decides how much curvature we want in a RBF kernel decision boundary. The higher the gamma value it tries to exactly fit the training data set. Technically speaking, large gamma leads to high bias and low variance models, and vice-versa.

(20) We use three different sets to avoid overfitting. It is necessary to have a validation data set in addition to the training and test datasets to tune hyperparameters correctly. Training data is the set of the data on which the actual training takes place. Validation split helps to improve the model performance by fine-tuning the model after each epoch, reducing overfitting compared to a unique train set. Finally the test set is used to test our classifier independently from any training to also limit overfitting.