POLITECNICO
DI TORINO

Machine Learning and Artificial Intelligence

Lorenzo Santolini
Politecnico di Torino

January 9, 2019

Contents

1 Homework 1

Requests In this experience we were given a dataset composed by images of dogs, guitars, faces and houses. The parts of the homework are:

1. Choose one image and shows what happens to the image when you re-project it with only first 60 PC, first 6 PC, first 2 PC, last 6 PC.

2. Using scatter-plot, visualize the dataset projected on first 2 PC. Repeat the exercise with only 3 and 4 PC, and with 10 and 11. What do you notice? Justify your answer from theoretical perspective behind PCA.

3. Classify the dataset(divided into training and test set)using a Naive Bayes Classifier in those cases: unmodified images, images projected into first 2PC, and on 3 and 4 PC. Show accuracy and compare results: what are your conclusions?

4. (Optional) Visualize decision boundaries of the classier in the first 2 PC case. Any consideration about those boundaries?

1.1 Decomposition

1.1.1 Theory

When our data are represented by a matrix too large (the number of dimensions is too high), it is difficult to extract the most interesting features and find correlations among them; moreover the space occupied is very high. PCA is a technique that allows to achieve dimensionality reduction while preserving the most important differences among samples. The technique consists of finding orthogonal axes on which the variance is maximized, and project the samples on them.

1.1.2 Manual algorithm

I tried to manually perform the PCA algorithm without the aid of the library sklearn, but the amount of ram requested is much, being our initial samples images (227,227,3). With every called function I used a decorator, to evaluate time performances. Function:

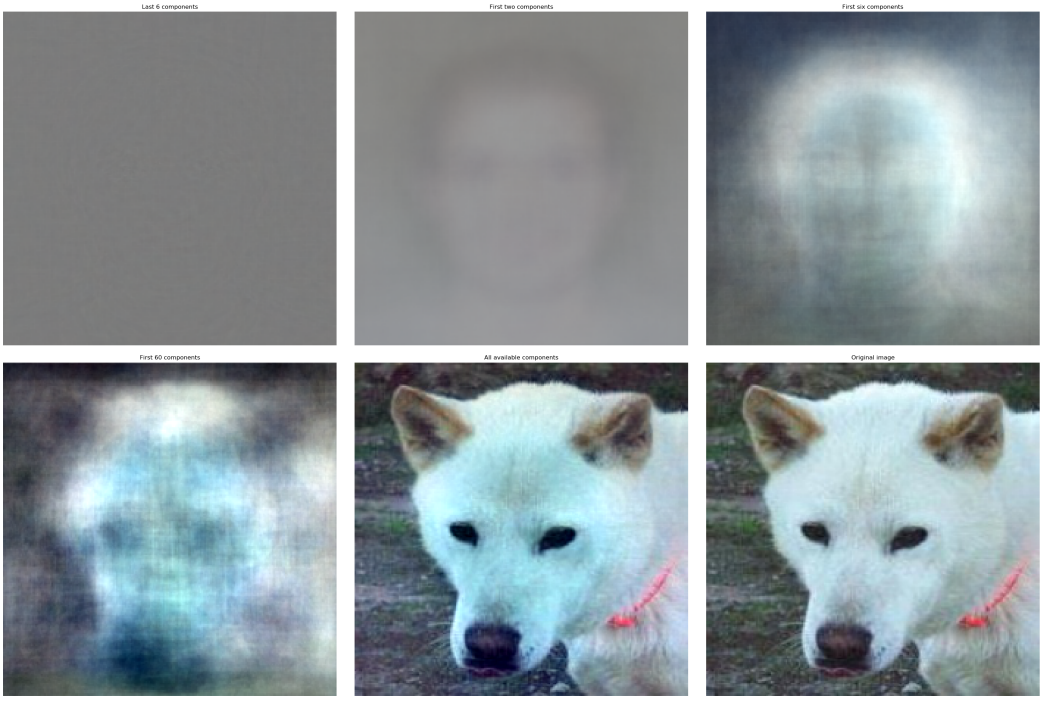
```
1  def classic_pca(X, size, nComp=2)
```

1.1.3 Sklearn algorithm

Due to the problem of before, I re implemented the algorithm with the aid of the sklearn library:

```
def improved_pca(X, size, labels, colors, imgn=700)
```

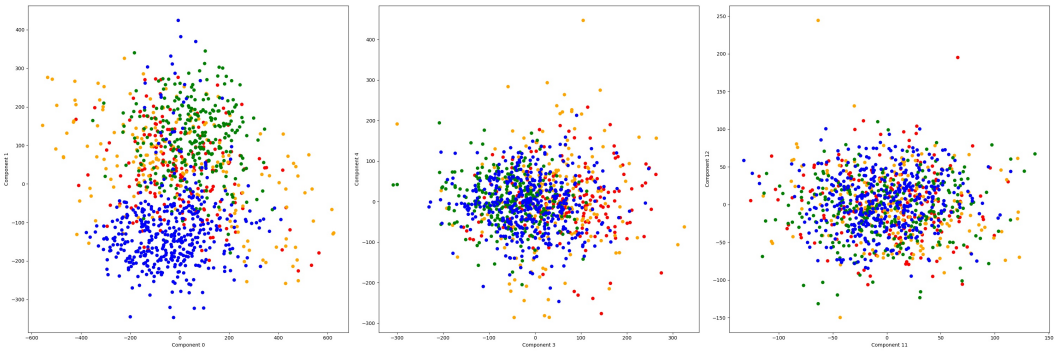
Plotting Once reshaped the images on their original size, we can display them with the different components (respectively Last6, First2, First6, First60, All, Original Image:

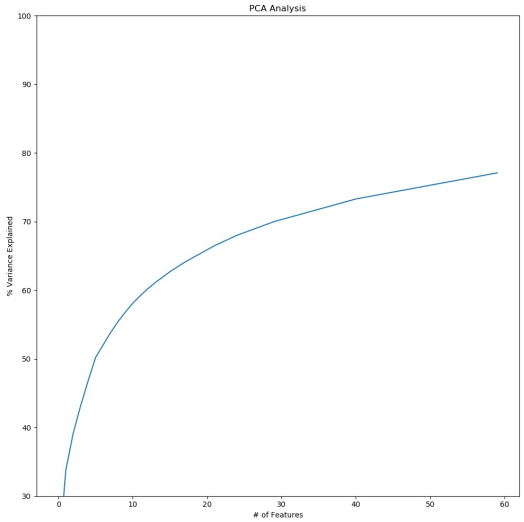


We can notice from these images that in the First2 and First6 ones, a face shape is easily distinguishable. This happened because in the used data set there were a majority of person images with respect to guitars, houses and dogs, and the extracted principal components report this fact in this way.

1.2 Scatter-plot

Here we analyze the scatter-plots of our samples, projected on 0 and 1, 3 and 4, 11 and 12 components respectively. From this plots emerges that in the first one points are better separated with respect to the other two; this because the first two components can better identify a sample of a certain kind.





The cumulative variance graph represents what percentage of the total variance is displayed by a certain number of components. This shows how much the image has lost in features with respect to the original one. In our case we can see that for 60 PC, almost 80 percent of the variance is retained.

1.3 Naive Bayes

Theory Naive Bayes classifier technique is based on the so-called Bayesian theorem and is particularly suited when the dimensionality of the inputs is high. Bayes classifiers are a family of simple "probabilistic classifiers" based on applying Bayes' theorem with strong (naive) independence assumptions between the features.

Abstractly, naive Bayes is a conditional probability model: given a problem instance to be classified, represented by a vector $x = (x_1, \ldots, x_n)$ representing some n features (independent variables), it assigns to this instance probabilities

$$P(C_k|x_1, \ldots, x_n)$$

for each of the possible classes C_k.
The problem with the above formulation is that if the number of features n is large or if a feature can take on a large number of values, then basing such a model on probability tables is unfeasible. To reformulate the problem using

Bayes' theorem, the conditional probability can be decomposed as:

$$P(C_k|x) = \frac{P(C_k)P(x|C_k)}{P(x)}$$

In practice, there is interest only in the numerator of that fraction, because the denominator does not depend on C and the values of the features x_i are given, so that the denominator is effectively constant.
In this experiment we used a Naive Bayes classifier, because we assumed that the distribution would approximate to a Gaussian.

1.3.1 Algorithm

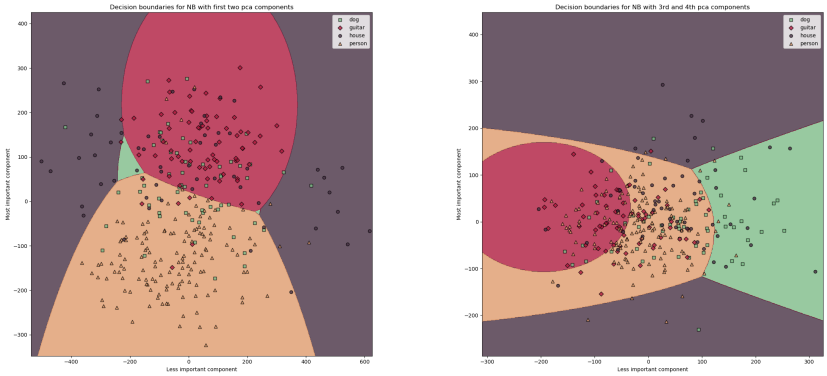
In this part of the homework I split the sample set in the training set and the test set. I trained the classifier first on train set without any re-projection, then on data projected on first two principal components then on third and fourth one.
The scores achieved by testing the models on the test set were:

No Projection	First2 Components	Second2 Components
0.779	0.666	0.428

We can easily notice that the score achieved using just the first two components is close to the one without dimensionality reduction, while with less important components we lose a lot in performances.

1.4 Decision boundaries

I plotted the decision boundaries for our newly created classifiers; In the plots is evident the better accuracy achieved using first two principal components with respect to the other two.



((a)) First two ((b)) Third and fourth

Figure 1: Decision boundaries

2 Homework 2

Requests In this homework we needed to load Iris Dataset in order to perform these steps:

1. Select first two dimensions in the dataset
2. Randomly split data into train, validation, test
3. Over different values of C:
   (a) Train SVM with Linear and RBF kernels
   (b) Plot data and decision boundaries
   (c) Evaluate the method on validation set
4. Show accuracy on validation set varying C
5. Evaluate best values on the test set
6. Perform a grid search on C and Gamma values for the Rbf kernel showing these values scoring on the validation set
7. Evaluate these parameters on the test set
8. Perform a k-fold cross validation on the grid search and show the results

2.1 Data preparation

I loaded the Iris dataset and selected the first components in a few steps:

```
1  iris = datasets.load_iris()
2
```

```
3   X = iris.data[:, :2]
4
5   Y = iris.target
```

To improve performances, it is always better to scale our data. Then i split the dataset in the three parts:

```
1   X = preprocessing.scale(X, with_mean=True, with_std=True)
2
3   X_train, X_test, y_train, y_test = train_test_split(X, Y,
4   train_size=0.5, shuffle=True)
5
6   X_test, X_val, y_test, y_val = train_test_split(X_test, y_test,
7   train_size=0.4, shuffle=True)
```

2.2 Linear search

2.2.1 Theory

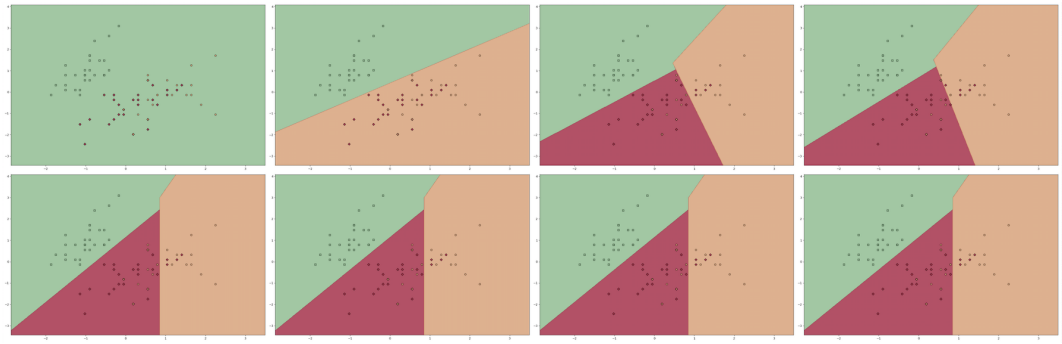
The C parameter tells the SVM optimization how much you want to avoid misclassifying each training example. For large values of C, the optimization will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly. On the other hand, a very small value of C will cause the optimizer to look for a larger-margin separating hyperplane, even if that means misclassifying more points. For very tiny values of C, you should get misclassified samples often, even if our training data would be otherwise linearly separable.

2.2.2 Results

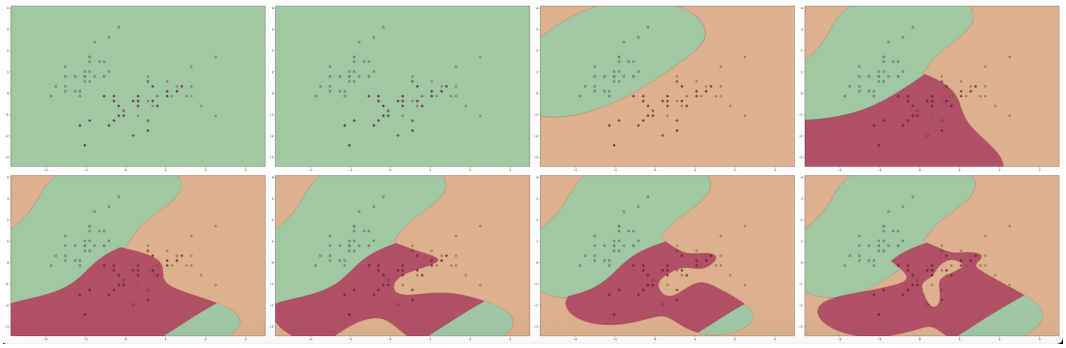
I created a logspace of 8 values for C, ranging from 10^{-3} to 10^{4}. Then i trained both the linear and rbf kernels SVMs on the same test set. After calculating the score for each of them on the validation set, I plotted the decision boundaries and the table of the scores. We can notice from the graph that the decision boundaries are really different:

C values	1.0e-03	1.0e-02	1.0e-01	1.0e+00	1.0e+01	1.0e+02	1.0e+03	1.0e+04
Linear	28.89%	77.78%	84.44%	84.44%	28.89%	64.44%	82.22%	75.56%
Rbf	64.44%	82.22%	84.44%	84.44%	28.89%	84.44%	82.22%	66.67%

Score table



Linear decision boundaries for increasing C



Rbf decision boundaries for increasing C

As we can notice, for low C values there are not even three classification boundaries, because the allowed mistake is really high, and misclassification is not influent. While we increase C, the boundaries change to fit data.

Now we can evaluate or best performing C value on the test set, and the obtained result is pretty good:

```
Best linear score achieved:  80.00%
```

Top linear kernel SVM score on test set

2.3 Grid search

2.3.1 Theory

Technically, the gamma parameter is the inverse of the standard deviation of the RBF kernel (Gaussian function), which is used as similarity measure between two points. Intuitively, a small gamma value define a Gaussian function with a large variance. In this case, two points can be considered similar even if are far from each other. In the other hand, a large gamma value means define a Gaussian function with a small variance and in this case, two points are considered similar just if they are close to each other.

The behavior of the model is very sensitive to the gamma parameter. If gamma is too large, the radius of the area of influence of the support vectors

only includes the support vector itself and no amount of regularization with C will be able to prevent overfitting. When gamma is very small, the model is too constrained and cannot capture the complexity or "shape" of the data. The region of influence of any selected support vector would include the whole training set. The resulting model will behave similarly to a linear model with a set of hyperplanes that separate the centers of high density of any pair of two classes.

2.3.2 Results

For every pair of gamma and C in:

```
C_range = np.logspace(-3, 3, 7)
gamma_range = np.logspace(-9, 3, 13)
```

I built a SVM with rbf kernel using those values, fitted it on the train set and scored it on the validation set. The result obtained is this:

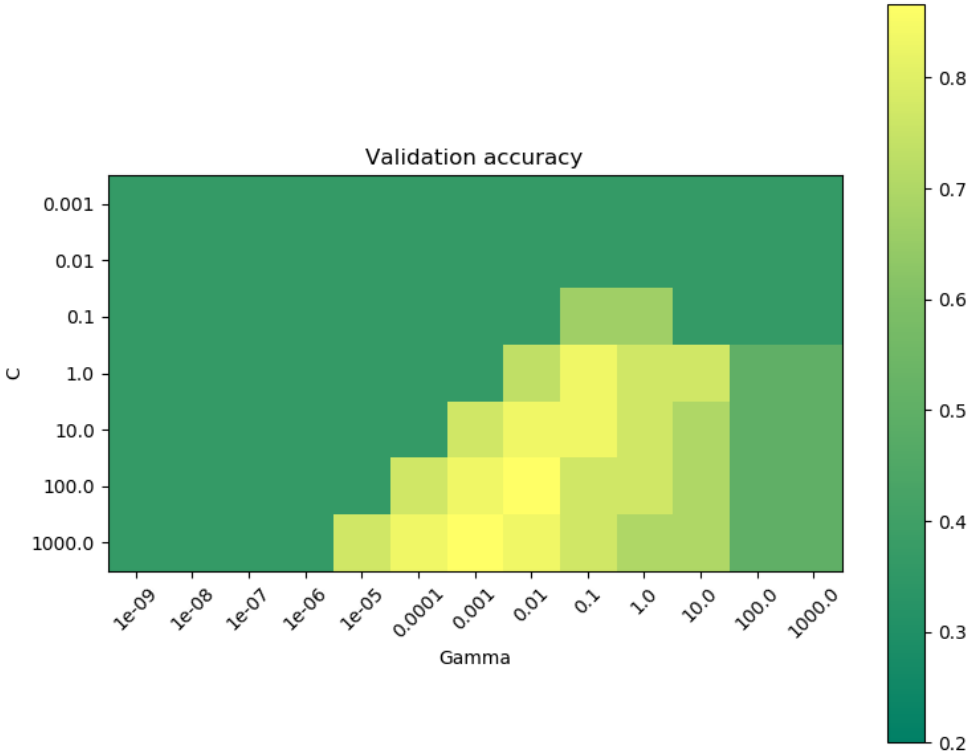
Gamma/C	1.0e−09	1.0e−08	1.0e−07	1.0e−06	1.0e−05	1.0e−04	1.0e−03	1.0e−02	1.0e−01	1.0e+00	1.0e+01	1.0e+02	1.0e+03
1.0e−03	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%
1.0e−02	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%
1.0e−01	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	62.22%	62.22%	24.44%	24.44%	24.44%
1.0e+00	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	84.44%	88.89%	77.78%	77.78%	48.89%	33.33%
1.0e+01	24.44%	24.44%	24.44%	24.44%	24.44%	24.44%	84.44%	88.89%	77.78%	80.00%	73.33%	46.67%	33.33%
1.0e+02	24.44%	24.44%	24.44%	24.44%	24.44%	84.44%	88.89%	82.22%	75.56%	80.00%	75.56%	46.67%	33.33%
1.0e+03	24.44%	24.44%	24.44%	24.44%	84.44%	88.89%	84.44%	77.78%	75.56%	82.22%	75.56%	46.67%	33.33%

Score table

```
Best C =  1.0
Best Gamma =  0.1
Grid search best achieved:  76.67%
```

Score on test set with best values

Heatmap The Table can also be represented in a heatmap, to have a better visual effect of where are the best and the worse values.



Colormap for grid search

Now we can plot the decision boundaries for the model:

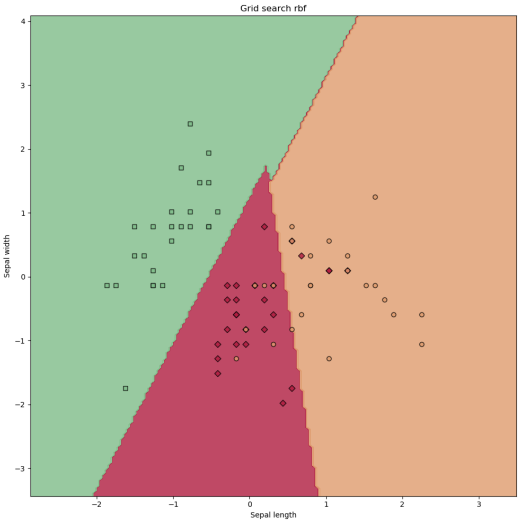


Figure 2: Best rbf classifier decision boundaries

We can notice that the decision boundaries are very similar to a linear classifier.

2.4 K-Fold Cross Validation

2.4.1 Theory

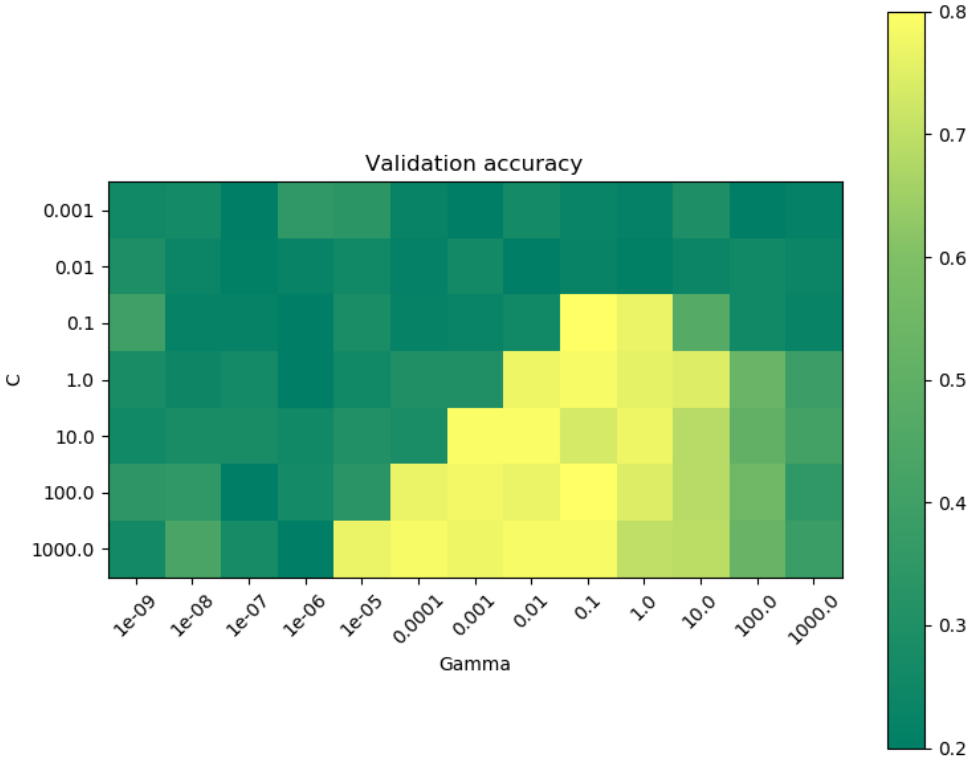
K-fold cross validation Is a resampling procedure used to evaluate machine learning models on a limited data sample. The procedure has a single parameter called k that refers to the number of groups that a given data sample is to be split into. The cross-validation process is then repeated k times, with each of the k subsamples used exactly once as the validation data. The k results can then be averaged to produce a single estimation. The advantage of this method over repeated random sub-sampling (see below) is that all observations are used for both training and validation, and each observation is used for validation exactly once.

2.4.2 Algorithm

I split the dataset in k folds, then I iterated over all of them, while performing a grid search on a logspace of gamma and c values. This procedure is pretty heavy, has a complexity of $O(c * g * k)$. For each combination of these three values, I have trained a SVM with rbf kernel, specifying gamma and c values myself, and then scored it on the test set, saving then the result in a 3d tensor. After this step, I averaged these values along the dimension of the folds, obtaining a 2D matrix with more reliable results. The obtained table and colormap are:

C/Gamma CV	1.0e−09	1.0e−08	1.0e−07	1.0e−06	1.0e−05	1.0e−04	1.0e−03	1.0e−02	1.0e−01	1.0e+00	1.0e+01	1.0e+02	1.0e+03
1.0e−03	25.33%	25.33%	21.33%	28.00%	18.00%	25.33%	19.33%	41.33%	20.00%	21.33%	29.33%	22.00%	23.33%
1.0e−02	23.33%	22.67%	22.00%	22.00%	26.00%	26.67%	20.00%	30.00%	39.33%	20.00%	24.00%	20.00%	23.33%
1.0e−01	32.00%	24.67%	23.33%	21.33%	25.33%	21.33%	24.00%	22.67%	72.67%	76.00%	27.33%	21.33%	30.67%
1.0e+00	21.33%	18.67%	42.00%	24.67%	29.33%	26.00%	21.33%	77.33%	78.00%	78.00%	72.67%	55.33%	42.00%
1.0e+01	20.67%	35.33%	32.67%	28.67%	30.67%	20.00%	72.67%	79.33%	77.33%	76.67%	67.33%	54.00%	40.00%
1.0e+02	17.33%	26.67%	35.33%	30.00%	20.00%	77.33%	79.33%	78.67%	81.33%	76.00%	67.33%	52.00%	38.67%
1.0e+03	24.67%	25.33%	28.00%	28.67%	76.67%	80.00%	77.33%	79.33%	78.00%	72.67%	69.33%	53.33%	44.00%

Table of n-fold cv results.



Colormap of n-fold cv results.

We can easily notice that, with respect to the simple grid search, values differ much more moving along c and gamma values. We can see that the best achieved is **81.33**%, with $C = 100$ and $Gamma = 0.1$

This result is more reliable with respect to the score without cross validation. It is a popular method because it is simple to understand and because it generally results in a less biased or less optimistic estimate of the model skill than other methods.

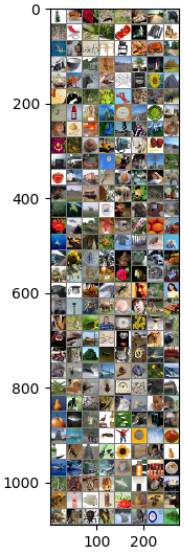
3 Homework 3

Requests In this homework we were requested to train and test different networks trying some deep learning techniques:

1. Train and test a simple Neural Network
2. Train and test a CNN
3. Train and test a CNN with more filters
4. Train and test a CNN with batch normalization and dropout
5. Train and test a CNN applying data augmentation to the dataset
6. Load and fine-tune on CIFAR100 the pre-trained ResNet18

3.1 Dataset

Our dataset is the CIFAR100, consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. Here there is an extract from of our dataset:



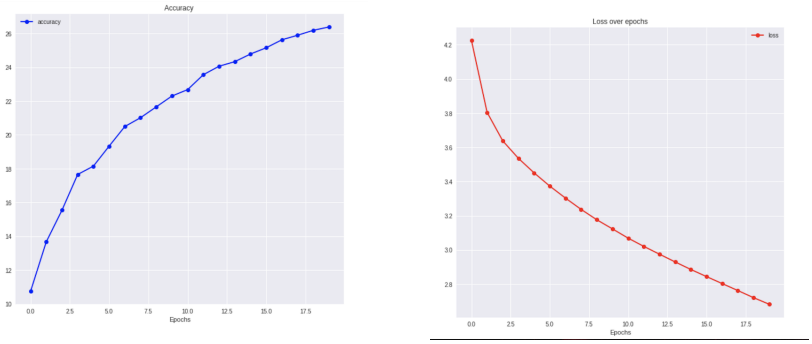
3.2 Simple NN

In this first part, I trained a simple Neural Network, composed by 3 fully connected layers, with these parameters:

1. batch size: 256

2. epochs: 20
3. image resolution: 32x32
4. solver: Adam with lr=0.0001

The obtained result is:



((a)) Accuracy ((b)) Loss

Figure 3: Accuracy and loss for SNN

As we can see, the top accuracy is not very high, achieving 26%.
We obtain a low accuracy compared to the one of previous homeworks. This is because the number of classes to be recognized is much higher.

3.3 Simple CNN

Here I trained a CNN with the same parameters of the previous point. The CNN is composed as:

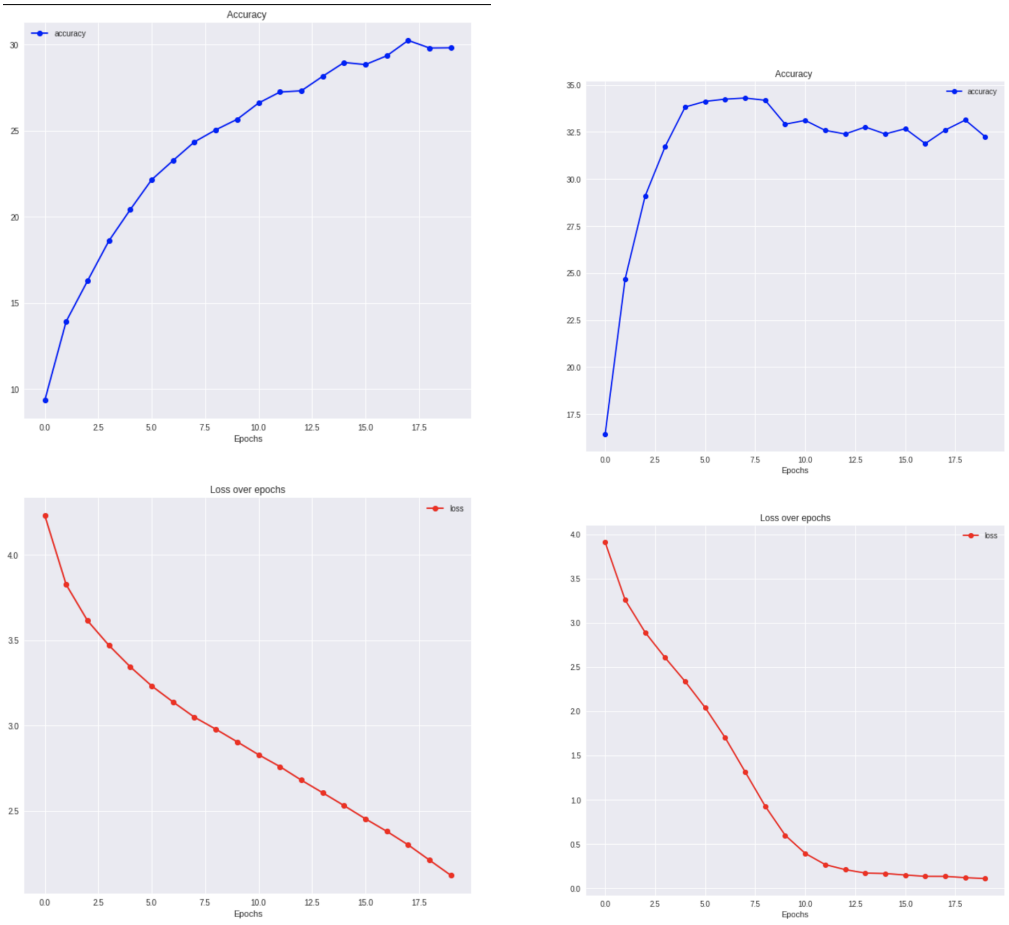
1. Three convolutional layers
2. One last convolutional layer
3. One Max pooling layer
4. Two fully connected layers
5. ReLus every layer to introduce non linearity

```
class CNN(nn.Module):
def __init__(self):
    super(CNN, self).__init__()
    self.conv1 = nn.Conv2d(3, 128, kernel_size=5, stride=2, padding=0)
    self.conv2 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=0)
    self.conv3 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=0)
    self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
```

```
        self.conv_final = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=0)
        self.fc1 = nn.Linear(64 * 4 * 4 * 4, 4096)
        self.fc2 = nn.Linear(4096, n_classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = F.relu(self.pool(self.conv_final(x)))
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

These are the plots of loss and accuracy, with different learning rates:



((a)) Learning rate 0.0001 ((b)) Learning rate 0.001

Figure 4: Accuracy and Loss in simple NN with 32 filters

We can notice from the plot below, that increasing learning rate from 10^{-4}

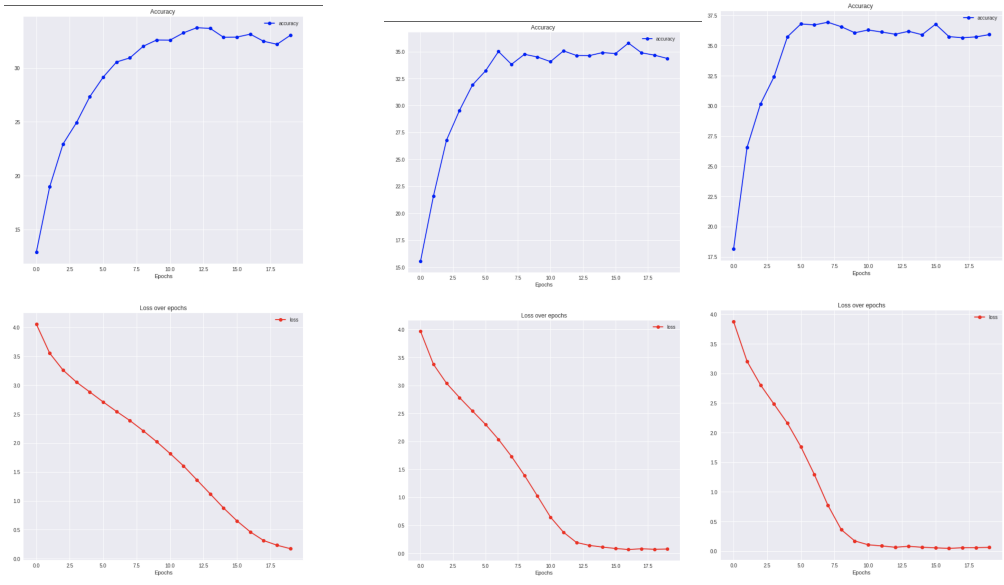
to 10^{-3} we get a better max accuracy in fewer epochs. Then the improvement stops and begins oscillating; this is due to overfitting to the train set.
Increasing the learning rate allows us to obtain better results in fewer epochs, even tough we might risk to miss the minimum if we overdue and we risk to obtain a diverging loss. The result with 3 experiments is 2% better than with lr=0.0001, and it converges in around 5 epochs instead of 15.

3.4 CNN with more filters

In this phase we did again the previous step, but increasing the number of filters used in the convolutional layers, from 32/32/32/64 to:

1. 128/128/128/256
2. 256/256/256/512
3. 512/512/512/1024

We can notice a small improvement in accuracy increasing the number of filters. Another interesting fact to underline is that, going up with the number of filters, the network learns faster, as if we increased the learning rate (more filters, more learning). Here are the plots of the loss-accuracy for the three sizes respectively:



((a)) 128 Filters ((b)) 256 Filters ((c)) 512 filters

Figure 5: Accuracy and Loss of CNN with different amount of filter

Let's make some observations on the obtained results:

- Case 256/256/256/512

  - After some tests I noticed that if we increase the learning rate to 0.00002 and we also increase the number of epochs to 40, it ends up overfitting (accuracy not improving anymore and also decreases) fast.
  - Looking at the curve with more epochs/lower learning rate (40 epochs, 0.0001 lr), we can see that converges more slowly, so it is better or reducing epochs, or increasing lr

- Case 512/512/512/1024
  - Around the epoch 5 the accuracy saturates, while loss keeps decreasing. After a few epochs also the accuracy starts to decrease due to overfitting. We need to introduce some technique to avoid this (like dropout).
  - The training time here gets pretty high due to the number of filters. On Colab it took 45 minutes.

3.5 CNN with some improvement techniques

In this part we need to train the previous neural network (CNN) but introducing some improvements:

1. Introduce batch norm every convolutional layer
2. Introduce batch norm plus make the FC1 layer wider
3. Introduce batch norm + dropout on FC1

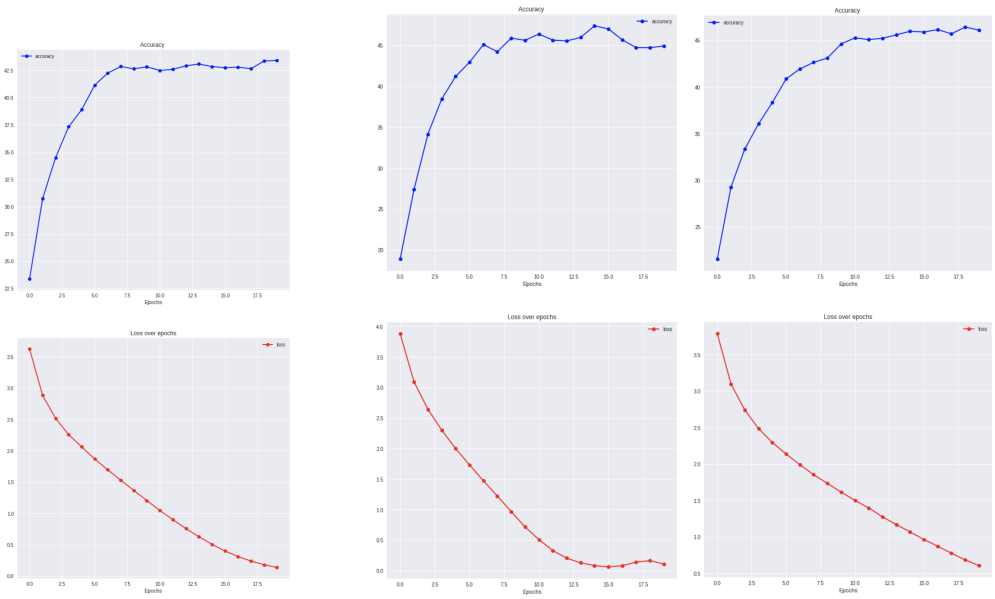
Batch Norm We can see a big improvement in accuracy; this is due to the re-normalization of the data every convolutional layer.
With lr=0.0001 the net accuracy converges after 7 epochs with an accuracy of 43%; Increasing learning rate of an order of magnitude can improve accuracy by a 3-4%, with the same convergence time. If we increase it of another order of magnitude, we can notice a very high loss and a low accuracy (around 10% on epoch 1)

Increasing FC1 size to 8192 Increasing size of the first FC layer does not improve significantly performances (around 46%, like the one before, using best learning rate(0.001).

Adding Dropout Inserting dropout = 0.5 after the first FC layer (FC1) increases training time without big improvements, we still obtain around 46%. We can notice that the loss decreases less, because we switch off some neurons at training time.
Inserting dropout = 0.3 we do not see an improvement of performances.

Inserting dropout = 0.8 improves accuracy of about 2%.
Inserting dropout = 0.8 AND increasing learning rate, we achieve very good performances (53%). Increasing epoch number, we do not have an improvement of performances.

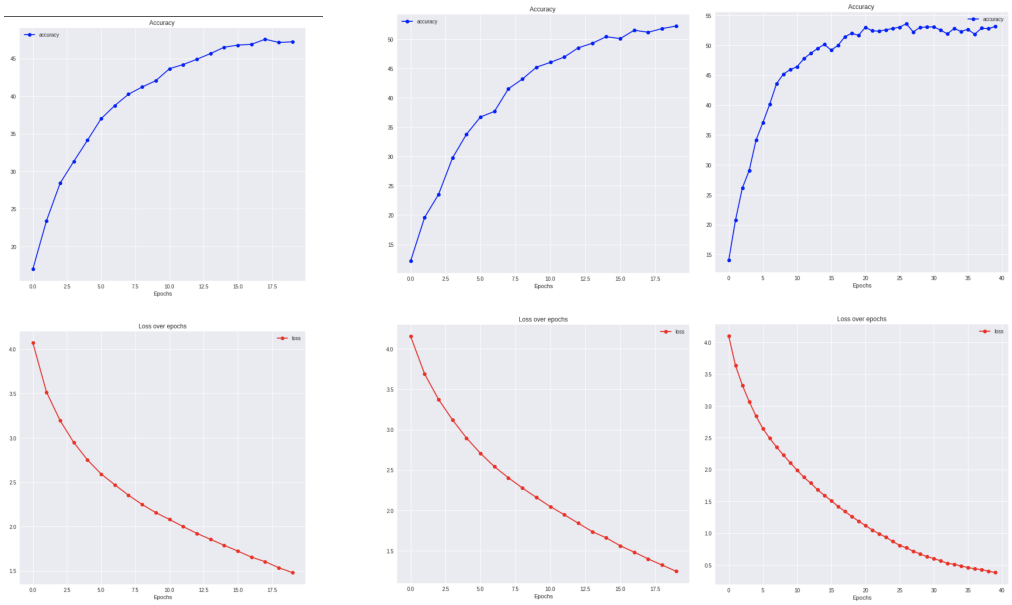
Here are the graphs of the three tasks:



((a)) Batch norm ((b)) 8192 Neurons on FC1 ((c)) Bn + Dropout=0.5

Figure 6: Differences with modifications

I analyzed more in depth the case of dropout=0.8. We can see that the best achieved (54%) is with high learning rate and more epochs (40):



((a)) dropout=0.8, 20 epochs, lr=0.0001 ((b)) dropout=0.8, 20((c)) dropout=0.8, 40 epochs, lr=0.0001 epochs, lr=0.001

Figure 7: Accuracy and Loss of CNN with dropout=0.8 in different cases

3.6 Data augmentation

Here I performed data augmentation in two ways:

1. Random Flips
2. Random Crops

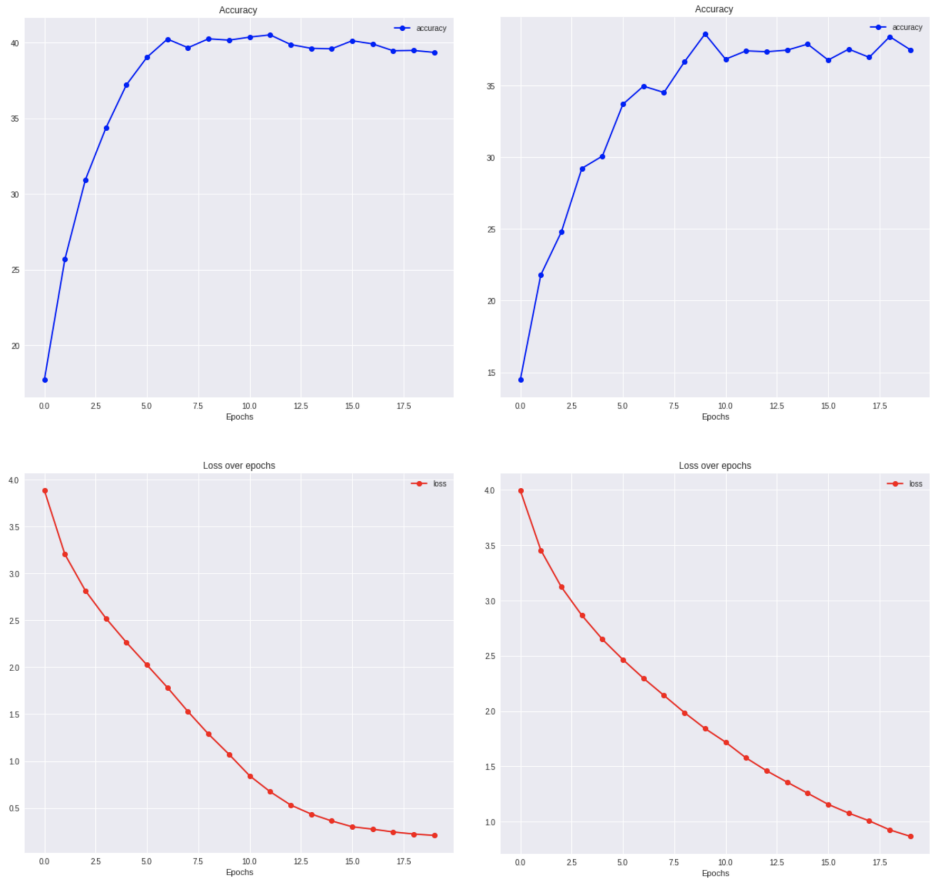
Horizontal flipping Using random horizontal flipping we achieve a good performance increase (we arrive to 40%). This is because we "see" more images, and learn more variations.

Random Crops Did not provide a significant improvement of performances, probably because to crop the images, we need to study better the dataset and we need to crop in different ways depending on image type; random is not enough good, we could learn unwanted features.

For the random crop I used:

```
transforms.Resize((40, 40)),
transforms.RandomCrop(size=[32, 32], padding=0),
```

These are the performances with the two techniques:



((a)) With horizontal flipping ((b)) With random crop

Figure 8: Accuracy and Loss of CNN with data augmentation

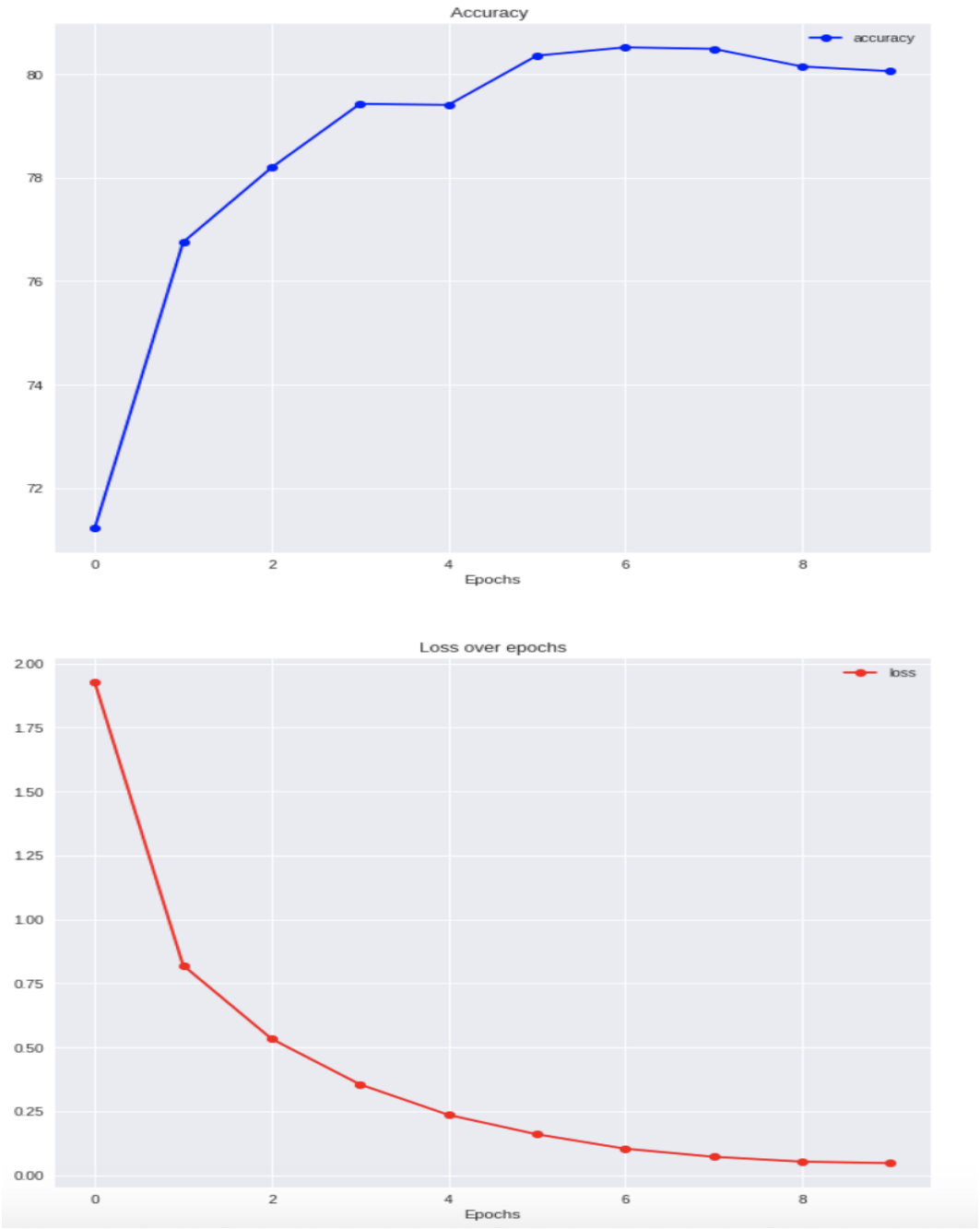
3.7 ResNet18

In this last step, I have imported the pre-trained model of the ResNet18, and I fine-tuned on our dataset with these parameters:

1. batch size: 128
2. resolution: 224x224
3. epochs: 10
4. solver: Adam solver with lr=0.0001
5. data augmentation: Random Horizontal flippings

Has an incredibly high accuracy, after the first epoch achieves 71%. This is because even if not trained specifically on this dataset, it has been trained on a much bigger one, learning more features. Moreover, the architecture is much more complicated, and the net very deep thanks to the technique

to avoid gradient vanishing. The "training" time is very high, even if we are just fine-tuning it (it took 62 minutes).
This is the loss and the accuracy behaviour over 10 epochs:

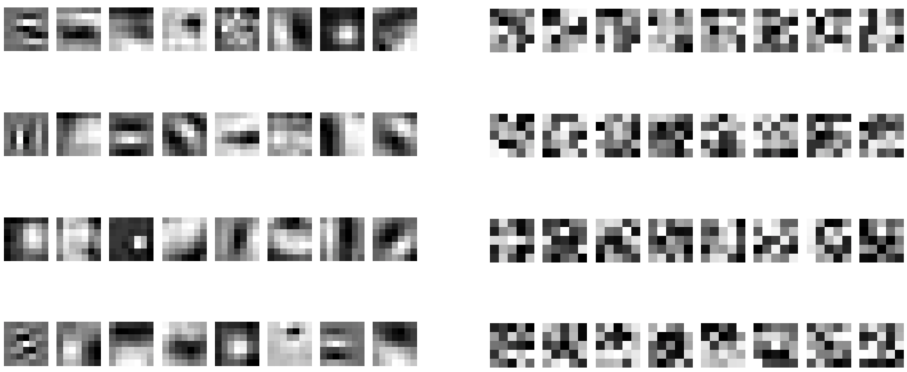


((a)) ResNet

Figure 9: Accuracy and Loss of ResNet18 fine-tuned over 10 epochs

3.8 Feature Maps

I plotted the feature maps after the first convolutional layer of the pre-trained ResNet18 and the ones of out simple NN. The differences are evident, in the first one we can see some patterns and structures, even tough these are low level filters, so are not very explanatory.



((a)) ResNet18's kernels ((b)) CNN's kernels

Figure 10: Feature maps after the first convolutional layer