Creation of a smart representation of pictures for interactive data exploration

Alexandre Sevin

Alexandre SEVIN

ENSAE 3^{rd} **Year**
End-of-Studies Internship
2016-2017

Creation of a smart representation of pictures for interactive data exploration

LIX
Palaiseau

Supervisor : Yanlei Diao
From May to September 2017

Contents

Introduction

The number of collected data rocketed during the last years. In 2020, we estimate that humankind will produce 40 zettaoctets (10^{21} octets), and it will not stop at this level. During this time, the human capacity to treat data did not increase. We are so facing a big challenge: how to deal with this huge amount of data?

In this context, CEDAR team is currently working on a project of interactive data exploration. This system is designed to help a user to find a region of interest in a database. To do so, it will show some samples to the user and collect his feedback on these samples. It will repeat these two steps until it learns the user interest.

The issue this system is solving is the exploration of a large dataset. For example, let's imagine that someone wants a secondary house, but he has no idea where and which style he desires. To find his perfect house manually, it would take an eternity, even with the good filters. Our system aims to provide him a whole set of announces which correspond to his taste.

The main difficulty is to converge as quickly as possible, e.g. in a minimum of iterations. To achieve a good accuracy on a minimum amount of time, we have to create a good algorithm which can converge fast with only a few training samples. We can easily see how it can find selective factors in a structured database, but it is far more difficult if we consider unstructured data like pictures.

The goal of my internship was to find a way to adapt this system with pictures. First, we will define properly the issue we want to solve. Then, we will look at standard feature extraction techniques and we will see how we can use these methods in our context. Finally, we will compare the different methods we tried.

1 Description of the problem

For a computer, a picture is nothing more than a 3D-vector. Let's take a picture of $n \times m$ pixels. Its associated vector would be of size $n \times m \times 3$. The last dimension corresponds to the three color channels (red, green, blue) where each value is between 0 and 255. For example, $[0, 0, 0]$ corresponds to a black pixel, $[255, 255, 255]$ to a white pixel and $[255, 0, 0]$ to a red pixel.

The difficulty of working with pictures comes from underlying structures. In fact, a pixel by itself does not bring much information. It is the shapes created by many pixels that are really meaningful. We have to teach the computer how to deal with these complex structures. Moreover, we have to create a representation of pictures which could be used by a standard machine learning algorithm, i.e. not a neural network, because we want to use this representation in the active learning framework CEDAR is working on.

First, let's insist on the fact that we do not only want the representation of pictures we have at our disposal. We want to create a function that maps a picture to a vector. If we have a new picture, we have to be able to create its representation without retraining everything. Moreover, this representation must discriminate photos according to different users' interests. This context could be think closed to recommendation systems, but it has several differences.

In our context, we want to find the entire region of interest, whereas a system of recommendation like the one from Netflix ([4]) or Amazon, is a model which aims to give the best recommendation to a user. It means that it will look for one or several items the user will like in its database, but it will not try to define as accurately as possible the decision boundary of the user.

The second difference is on available data. The best approaches for recommendation system are based on collaborative filtering. They give a recommendation based on other users which have a similar behavior. In our case, we suppose that we don't have access to data coming from other users, or not enough to be able to rely on similitudes between users. In practice, we trained our model with 2 different users.

The second point of interest of our context is the dimensionality issue. We want our representation to be low dimensional. A photos of $n \times m$ pixels lives in a space of $n \times m \times 3$ dimensions. However, photos do not cover all this space, they are grouped in a sub space. With the theory of wavelets [1], we can create a photo with only a few thousand dimensions which is so close to the original that a human can not see the difference. However, this representation is not really useful for classification. Our idea is to find a representation with high-level semantic features, to be able to classify pictures with only a few attributes.

Finally, the last issue of our context and probably the harder is the locality property. We want to find the region of interest of the user, so we want the representation to bring closer photos which share same characteristics, without knowing exactly which characteristics we are talking about. Two different users will based their choices on different attributes, so theoretically, we have to take into account every possible attributes.

During this internship, I worked with a housing dataset. It contains 640 property advertisements. For each one, we manually search on the Internet a picture which could be with the advertisement. This leads to 640 pictures from various sources. Moreover, I labeled this dataset according to my interest, and I also ask someone working with me to do the same. We had so two different set of labels available.

First of all, we looked at state of the art methods to perform feature extraction.

2 Literature Review

In this section, we will quickly present the main techniques of feature extraction on pictures. We will first present how these methods work, and then their advantages and drawbacks for our problem.

Let's begin with SIFT detectors.

2.1 SIFT

SIFT stands for Scale Invariant Feature Transform. It was introduced by [8] and fully described in [9], is a method to find keys in a picture. These keys are invariant to scaling, stretching, rotation, contrast, and intensity variations. They are based on color gradients from multiple levels of sampling of the original picture.

Applied to one picture, SIFT will give hundreds of detectors. Each one is described by 128 features. A result of this algorithm is given in figure 1.

Figure 1: SIFT algorithm applied to a picture of our dataset

The computation of these descriptors is done in two steps. First, we find points of interest in our picture. To do so, we will create several pictures from the original one by applying successively Gaussian blurs and down-sampling. Then, we will look for local extrema in the difference of two successive created pictures. This will give us a lot of points of interest in our picture. To remove some, we will apply two filters. The first one removes key points which have not enough contrast, the second one removes those who are on edges but not on corners. These two types of key points are unstable from one picture to another.

These detectors are originally made for object recognition. It means that given a picture of a particular object, we can easily find this object in a second image by searching for a group of detectors similar to the ones extracted from the original picture.

After this paper, a lot of other techniques emerged, like PCA-SIFT ([7]), SURF ([2]), ORB([13]), DTCTH([12]) ... There are all based on the same technique, but each one brings its own improvement (speed, accuracy, resource consumed, ...).

These techniques are very hardly applicable to our context. In fact, the detectors are very useful to find precise features on a given object, but we are interested in the general content if pictures. Moreover, we are looking for features, but also linked between them. It means that we hope to find relations like "if a user like this feature, he will like also this feature". These relations cannot be caught by this technique.

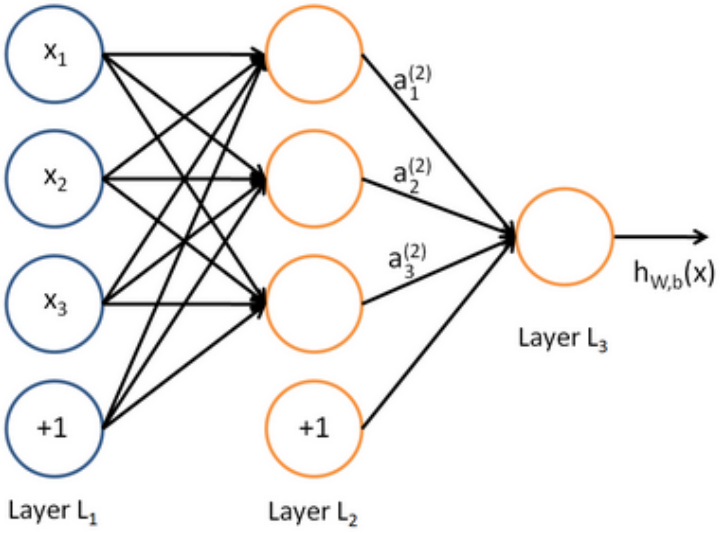


Figure 2: General architecture of a layer of a neural network. Source: Stanford.

Before the apparition of Neural Network and more precisely Convolutional Neural Network (CNN), this was the most widely used technique to perform feature extraction. Now, CNNs are trending. It is a field where there are improvements almost each single day, and there is a lot of different applications, including feature extraction.

2.2 Convolutional Neural Network and Transfer Learning

First of all, let's have a quick reminder about neural networks and CNNs. A neural network is a superposition of several layers. It takes a numerical input which goes through several hidden layers to finally give a numerical output.

Each layer works in the same way. With an input X of size n_I, the output is given by $Y = f(W^T \cdot X + b)$ where f is an activation function, W is a weight matrix and b is a bias. The output size depends on the number of units in the hidden layer. If there is n_h units, the size of W and b are $n_h \times n_I$ and n_h. f is an activation function. It could be a relu ($f(x) = max(0, x)$), a sigmoid ($f(x) = \frac{1}{1+e^{-x}}$) or a hyperbolic tangent ($f(x) = tanh(x)$) for example. The output will be of size n_h.

To train the neural network, we choose a loss function to minimize. It can be l2-loss ($L(y, \tilde{y}) = ||y - \tilde{y}||_2$), or cross entropy($L(y, \tilde{y}) = -y \log(\tilde{y}) - (1 - y) \log(1 - \tilde{y})$) for example. We then feed the neural network with batches of samples and for each weight, we compute and apply the gradient of the loss function. This algorithm requires several passes through the whole dataset. One pass is called an epoch.

In a convolutional neural network, there is a special type of layer which takes a 3D input and gives a 3D output. Given one input X of size $n_{I_1} \times n_{I_2} \times n_{I_3}$, a kernel size $n_{k_1} \times n_{k_2} \times n_{k_3}$, with $n_{I_3} = n_{k_3}$ the output will be: $Y = k \odot X + b$ where k is the kernel and b is a bias. In other words, we will have $Y_{i,j} = \sum_{i_1=-n_{k_1}/2}^{n_{k_1}/2} \sum_{i_2=-n_{k_2}/2}^{n_{k_2}/2} \sum_{i_3=0}^{n_{k_3}} k_{i+i_1,j+i_2,i_3} \times X_{i-i_1,j-i_2,i_3}$. A convolutional layer is composed of several kernel, which will give a 3D output.

Now that this quick reminder is done, let's explain how CNNs can be used via transfer learning to perform feature extraction. We call transfer learning the use of a network which has been trained on a different task. For example, training a neural network to classify dogs

from cats and using it to classify human from animals is a use case of transfer learning.

The most widely used source is Imagenet. It is a competition for object recognition and localization which provides millions of pictures from one hundred different labels. A lot of models are tested on this dataset, and it is supposed to be one of the most complex and complete datasets of pictures.

To show some examples of transfer learning, we can cite the paper [6]. They apply transfer learning in order to recognize breast cancer. Another example is the image style transfer proposed by [3], which uses the last layer of VGG, a CNN trained on Imagenet, as a representation of the content of a picture. This last one is very interesting, because we can think that a good representation of the content of a picture can fit well in our context.

Nowadays, state of the art results on Imagenet are obtained by Resnet [5]. It is a CNN with residual connections between some layers. The structure of this CNN is clearly described in the paper, and we could not do better, so we let the curious reader have a closer look on it if he wants more information.

To perform transfer learning, we remove one (or more) layer at the end of a pre-trained model. It means that we remove at least the classifier, and we replace it with an other classifier. It can be an other neural network layer, but it could also be a standard classifier (SVM, gradient boosting, random forest, ...). We take the weights of the unchanged layers as they are, and we only train the layers and the classifier we added on our task.

Transfer learning has several advantages. The major one is that it gives us the possibility to use a very complex and heavy model without training it, which could take days or weeks with our computation power. However, with this method, we can not customize the representation as much as we want. In fact, we will use the representation given by Resnet and we will build something to customize it a little, but we will never refer to raw pictures. It means that if we are interested in information which is not present in the representation given by Resnet, we will miss it and we could not recover it.

If we want to have a more custom model, we have to create our own neural network and train it. To do so, we have two different options.

2.3 Auto-encoder

The first option would be to use a classic neural network, convolutional or not, on the top of our raw pictures. It would require a labeled dataset and we could train it in a classic way. However, the pictures we have are very high dimensional (we took the same preprocessing than the one for Resnet, which give pictures of size $224 \times 224 \times 3$) and we don't have a lot of samples (640). With a number of features more than 200 times higher than the number of samples, we will have a big problem during the training. A model can learn perfectly the dataset without learning anything generalizable. That's why we did not try this method.

The second option is to train the neural network in an unsupervised way with a help of auto-encoders. Instead of classifying pictures, auto-encoders aims to reconstruct their input. However, in order to make it learn something useful, we impose some constraints on the hidden layers of this network. The most common one is to lower the size of the hidden layers. With this constraint, the neural network has to learn to compress the information. Moreover, if the dimension is too low to contain all the information, it has to select which part of the information it will keep to reconstruct as closely as possible the input.

There are several variants of auto-encoder. We can cite denoising auto-encoder ([14]), or deconvolutional auto-encoder ([11]) for example. However, they all work on the same principle, so we will stick to the standard auto-encoder in this explanation.

With this second method, we overcome the problem of the labels, but we are still facing the issue of dimension. Here again, the auto-encoder could learn the dataset perfectly. This is why we will to not train a model from scratch.

3 Description and tuning of the models

The solution to our lack of data is the use a neural network which has been trained on another dataset, and on another task. We choose the most standard source, Imagenet, and a model which provide state-of-the-art results, Resnet. We removed the last layer, e.g. the classifier and we obtained a representation with 2048 attributes.

We could think that the penultimate layer of Resnet give us the representation we want, but it could contain some information which is not relevant to our context, and the piece of information we want is maybe too small.

First of all, let's challenge a bit the representation given by Resnet.

3.1 Naive Approach

We performed several tests to see if Transfer Learning with Resnet can help us in our context.

The first one is simply to run the entire Resnet model, with its Imagenet classifier, and to look at the results it gave us. Results are given in figure 3. Even if Imagenet has not a "house" or "building" label, it has "mobile_home" and "patio" which cover more than two thirds of our data. The other pictures are split between some building types ("monastery", "boathouse", "church", ...) or objects which could be present nearby a house ("picket fence", "tile roof", ...).

These labels do not fit perfectly the pictures, but they are far from being totally absurd. To confirm that Resnet can recognize a house, we can perform a second test. We fed Resnet with pictures from our dataset and pictures extracted from ImageNet database, and we train a classifier on the top of the penultimate layer to recognize if a picture comes from our dataset or Imagenet dataset, knowing that all pictures received the same preprocessing (resizing and normalization).

In other words, we trained a classifier to recognize if a picture is a house or not. With a simple classifier (a logistic classification), we got an accuracy on the training set of 1. and on the testing set of more than 99 %. These scores mean that the Resnet can recognize almost perfectly a house.

Now that we know that the Resnet is not completely irrelevant in our context, we have to check if it is useful for the classification of user's interest. To do so, we created a classifier trained to recognize the labels we gave to the photos.

To train our model, we split the dataset into a training set (479 pictures) and a testing set (161 pictures). The testing set is not used at all during the whole training phase. We just compute a score on this set to have an idea of the generalization power of our model.

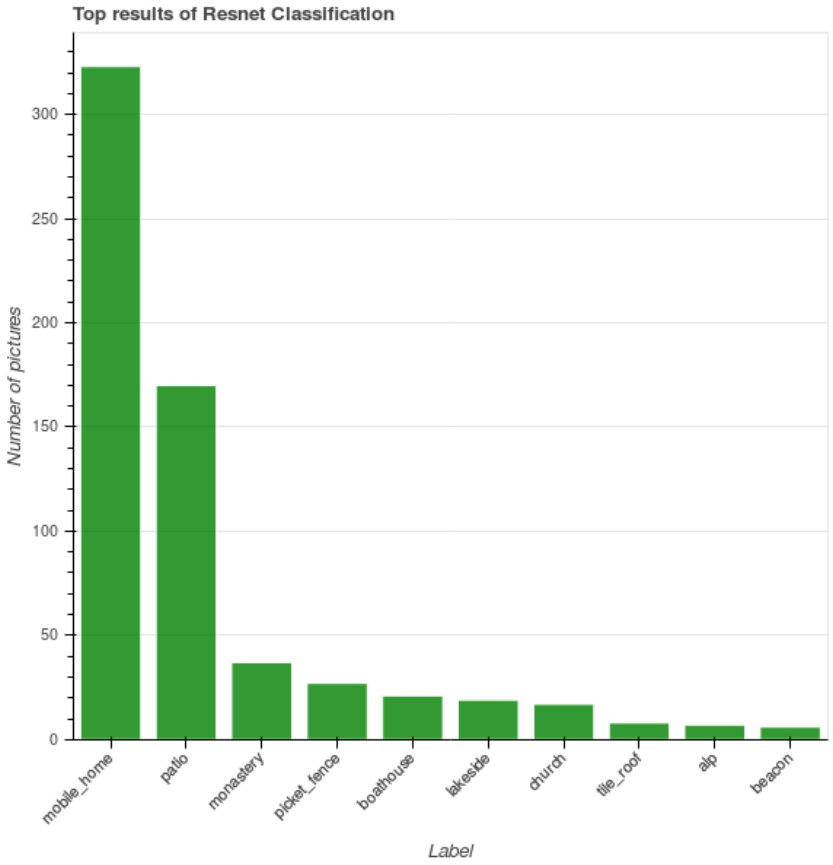


Figure 3: Results of Imagenet classification on our housing dataset.

We kept the same class frequencies on the training and the testing datasets.

We then performed a cross validation. We split again the training set into five buckets, in which we also kept the class frequencies. We then trained our model on four buckets and computed a validation score on the fifth. We did this for each 4-1 split, and we chose the hyperparameters which maximized the average validation score on these five splits. Hyperparameters are parameters whose values are chosen before the beginning of the learning phase (penalization term, learning rate, ...). Then, we trained the model with the best hyperparameters on the whole training set, and we computed a final score on the testing set.

Here, we chose to classify the pictures with a logistic regression with l2 penalization. In this model, if we note $(x_i, y_i)_{i<n}$ the n available observations and their labels, we estimate the value of y with a function $f(x) = w^T x + c$. We have to find the best values for w and c. To find these weights, we minimize the logistic loss $L_{log}(y, x) = log(1 + e^{-yf(x)})$ for each observation, with a penalization term $\frac{1}{2C} w^T w$. C is the inverse of the regularization coefficient, e.g. the lower C is, the higher the penalization is. This gives us the final optimization problem

$$\min_{w,c} \sum_{i=0}^{n} log(1 + e^{-y_i(w^T x_i + c)}) + \frac{1}{2C} w^T w$$

After a quick manual search, we found that a value of 0.008 for C gives the best accuracy during the cross validation (a mean accuracy of 0.88 on the training set, 0.77 on the validation and 0.79 on the testing set). With this classification and a training on the whole training dataset, we get a training score of 0.88, and a testing score of 0.81.

We can draw two conclusions from this result. First, knowing that the naive accuracy,

e.g. the accuracy we can get by always predicting the same label, is 0.69, we can see that this classifier learned useful information. It means that Resnet is pertinent for our context.

The second conclusion is less certain. During the final training phase, the training set is bigger than the one during the cross validation (almost 100 samples more). This rise leads to an improvement of 3% in the accuracy, so we can think we could improve a lot the results will have by adding more data.

Now that the relevance of transfer learning is confirmed, we can think about how to improve the representation obtained with Resnet. The first property we will work on is the dimensionality of this representation. In the next part, we will try to get similar or better results with fewer attributes per sample.

3.2 Classic Approach

A classic approach is to add a new hidden layer at the end of Resnet, before the classifier. In order to lower the dimensionality, we will impose the number of units to be lower than 2048, the dimension of Resnet representation. This hidden layer is trained with a classification task e.g. we train the neural network to find the labels we gave to pictures. With this training, we can hope that the hidden layer will catch useful information for our problem while deleting the noise coming from irrelevant information.

In order to test this representation, the perfect measure would be to run the data exploration algorithm on a bunch of user interests, and to see how good it performs depending on the representation we are working on. However, we only have two sets of labels at our disposal, and it would be very resource consuming to run the entire online algorithm with each configuration.

Instead, we chose to train the neural network to classify a single user's interest in a batch context. It means that it trained on 75% of the database, and we tested its result on the remaining 25%. The score we obtained with this batch classification gave us an approximation of the upper bound in the online context. It is an approximation because the data exploration algorithm is supposed to choose its the training set to maximize its accuracy, and not randomly split the database.

To run this experiment, we took the representation given by Resnet of our dataset. We then trained a hidden layer and a classifier on our dataset with the same labels as the previous experiment. Figure 4 describes this architecture.

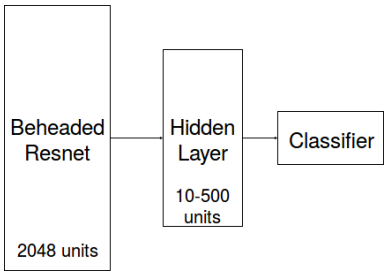


Figure 4: Architecture used for our naive approach

Like before, we split the dataset between a training and a testing dataset, and we performed a cross-validation. However, we did not train the model on the whole dataset to get a final testing score, because we needed a validation set to stop the training of the

neural network. For each model, we trained the neural network and kept the model when the validation score is the highest.The results we present are so averaged on the five folds of the cross validation.

To measure the performance, the score we chose is the accuracy. It is the ratio between the number of good classified samples over the total number of samples.

3.2.1 First experiment with dropouts

First, we trained a neural network with a hidden layer of 128 units with different values for the dropout. Dropout consists in randomly sampling units each time you feed some data to the neural network and shutting them down for this training step. Each time you feed data, you sample the units you shut down. We can add dropout to the input, to the hidden layer(s) or both.

Dropout is a classic technique to reduce the overfitting of a neural network. We say that a model is overfitting when it performs a lot better on the training dataset than on the testing dataset. This can be caused by a lack of data for example. When the training dataset is too small, the model will memorize the dataset instead of learning general information. Like we have a small database, we were strongly thinking that we would face overfitting issues.

The results are presented in figures 15 and 16 in annexes. They are presented in a grid, the second figure being the right part of the first one. When you go down, the dropout on the input is decreasing (the keeping probability is increasing). When you go right, the dropout on the hidden layer is decreasing. The keeping probability take the values 0.1, 0.25, 0.5, 0.7, 0.8, 0.9, 1., where 1. corresponds to the case without dropout.

Each graph shows the accuracy during the training phase on the training, validation and testing set. These accuracies are averaged on the five 4-1 splits of the cross validation. Figure 5 is one these graphs for a dropout of 0.1 on the input and on the hidden layer. The training accuracy is represented by the red curve, the green one stands for the validation, and the blue one for the testing.

With these results, we can do some interesting observations on the effect of the dropout. First, let's look at the first row, and at the red curve representing the training accuracy. More precisely, we will consider the point where the model performs better than a model predicting always yes or always no. This type of models can reach an accuracy of 69 %. We can see that the neural network with the highest dropout begins to learn something useful after about 1500 epochs. We can also see that, when the dropout on the hidden layer is decreasing, this point is reached more quickly (something like 500 epochs for the second model, and almost instantaneously for the others).

Then, let's keep considering the first row and the training accuracy, and let's look at the point where the model reaches a precision of 95 %. For the neural network with the highest dropout, this point is reached between 8000 epochs and 10000 epochs, whereas it is reached in less than 6000 epochs for the second. We can see that this point is reached faster when we decrease the dropout on the hidden layer.

We can complete these two observations by saying that, if we fix the dropout on the input on another value than 0.1, e.g. taking another row, these two remarks are still true. Moreover, if we fix the dropout on the hidden layer and look at the effect of a variation in

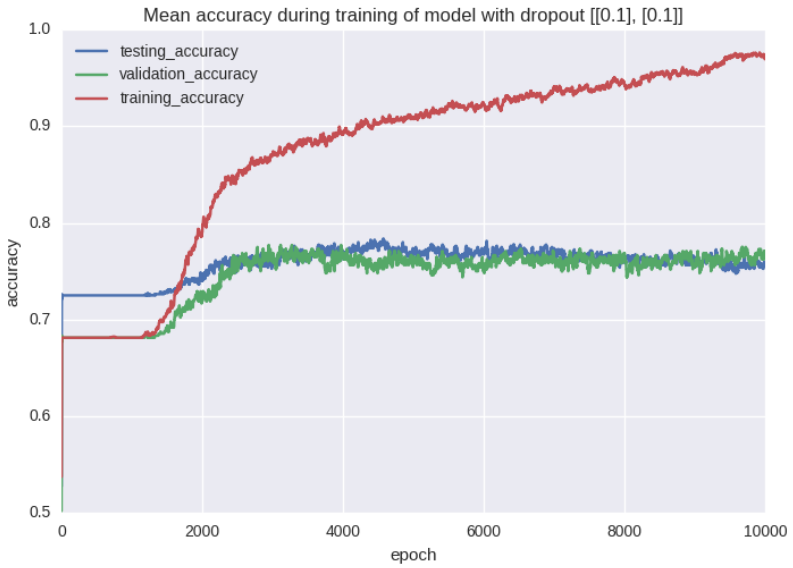


Figure 5: Example of a graph representing the training of one model

the dropout on the input, these two remarks are still true again.

With these observations, we can say that the dropout on the input and the dropout on the hidden layer have the same effect. They both slow down the training of the neural network. However, this is not the effect we are interested in. We want to see the impact of dropout on the generalization power of the neural network. To do so, let's look at the validation and the testing accuracies.

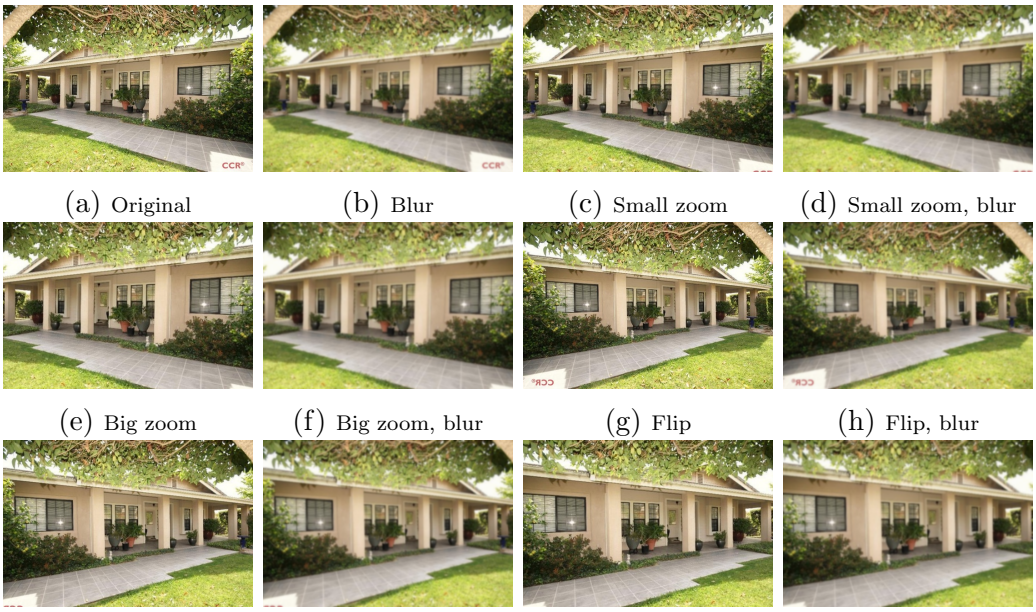
We can observe that there is not a big difference between the different graphs, except when there is a very high dropout, which just slows down the process. In fact, all validation and testing accuracies are reaching the same point and stay more or less constant after. This means that the dropout does not have a big influence here in the training process.

Finally, we can notice that the validation and testing sets are close to the value we found in the first experiment with the whole Resnet representation (around 77% of accuracy). However, our model is strongly overfitting the training dataset, like we anticipated. It means that we could maybe find a solution to prevent this overfitting and so increase the validation and testing accuracies. Dropout was a first option, which did not work. Let's try data augmentation.

3.2.2 Data Augmentation

The failure of the dropout can be interpreted as the lack of information in the training dataset. In fact, we can think that information we want hidden behind some more obvious features, but which are not general enough to discriminate the testing dataset. To overcome this issue, we tried data augmentation.

Data augmentation is a way to increase the number of available samples. From one picture, we created several others (11 in our case) by applying small geometric transformations. We flipped, zoomed a little and blurred the pictures. The results of these transformations are presented in figure 6.



(a) Original (b) Blur (c) Small zoom (d) Small zoom, blur

(e) Big zoom (f) Big zoom, blur (g) Flip (h) Flip, blur

(i) Flip, small zoom (j) Flip,small zoom,blur (k) Flip, big zoom (l) Flip, big zoom, blur

Figure 6: Data augmentation

We use the same neural network architecture as before, i.e. a model with one hidden layer of 128 units. During the training phase, we added to the training set these created pictures. However, we did not add them to the validation set, neither to the testing set, because we wanted to complete the same task as before, and we wanted the results to be comparable.

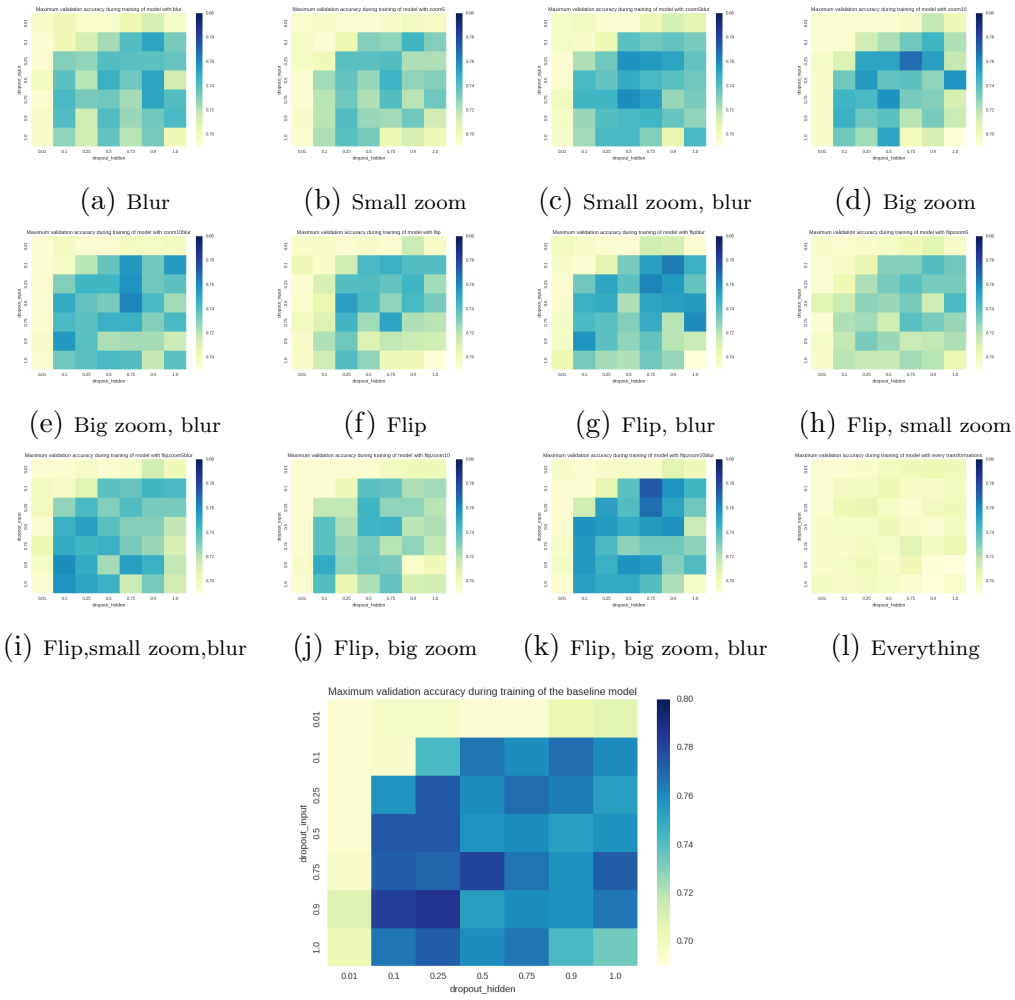
The heatmaps of figure 7 represent the results obtained by training a model on the original images, the original images augmented with one type of transformation and the original images augmented with every transformations we made.

For one heatmap, the vertical axis is the keeping probability on the input and the horizontal axis is the keeping probability on the hidden layer. Each square is representing the validation accuracy achieved by the model with a given dropout. The bluer the square is, the more accurate the model is.

First, we can see that for every transformations used, when the dropout is too high, the model did not learn anything. We can see it because the first row and the first column are always clearer than the rest of the heatmap. The score is around 70% or less in these areas, which is the score of a model predicting always no.

Then, we can see that the middle of the heatmap is often darker than the border. This means that the dropout has a positive effect on the accuracy. However, the difference is very small, and the best keeping probability is changing from one experiment to another. It seems that a keeping probability of 50% on the input and on the hidden layer is good in a majority of cases.

Finally, we can see that the best model is the one without data augmentation, which scores a little less than 80% on the validation set. On the contrary, the model with every transformations is the worst model. If we look at the accuracy curves during the training (figure 20 in annexes), we can see that the training accuracy is increasing, whereas the validation and the testing accuracies are decreasing. This means that the model is learning specificities of the training dataset.



(a) Blur (b) Small zoom (c) Small zoom, blur (d) Big zoom

(e) Big zoom, blur (f) Flip (g) Flip, blur (h) Flip, small zoom

(i) Flip,small zoom,blur (j) Flip, big zoom (k) Flip, big zoom, blur (l) Everything

(m) Original

Figure 7: Results from data augmentation

These observations tell us that the data augmentation is not adapted to our problem, or at least that this augmentation is not adapted to our problem. We can try to explain this by a bad quality of the new pictures. If our created pictures are not homogeneous with the original ones, this technique can not work. To test this, we trained a Support Vector Machine (SVM) to separate the original pictures from the created ones. SVM is an algorithm which finds the best hyperplane to separate the positive from the negative samples.

Here, we trained the SVM without any fine tuning of the hyperparameters, because we just need an approximation of the result. If we get an accuracy close to 0.5, it means that we created good pictures. If the accuracy is close to 1., it means that our created pictures are not relevant. Results are presented in table 1.

We can see that geometric transformations have different results. The flipped pictures are the hardest to discriminate, which seems logical becaus%.e we do not change the picture by flipping it. The zoomed ones are a little easier, and the blurred are the worst. It means that blurring pictures gives pictures which are very different from the original ones whereas flipping pictures could work well.

This impression is confirmed by the results of the neural network. It performs better on flipped images than on blurred images. However, this does not explain why the neural

Transformations used	Training accuracy	Testing accuracy
flip	0.569	0.393
zoom5	0.697	0.65
zoom10	0.786	0.743
blur	0.993	0.989
flip and zoom5	0.697	0.632
flip and zoom10	0.781	0.728
flip and blur	0.992	0.989
zoom5 and blur	0.992	0.989
zoom10 and blur	0.992	0.989
flip and zoom5 and blur	0.992	0.989
flip and zoom10 and blur	0.992	0.989

Table 1: Results of the SVM to separate created pictures from original pictures

network performs better on original pictures than on original and flipped pictures. We are still working on finding an explanation to this point.

To conclude this experiment, we will run several sensitivity tests to see if the results depend on hyperparameters we did not cross-validate.

3.2.3 Sensitivity of the results

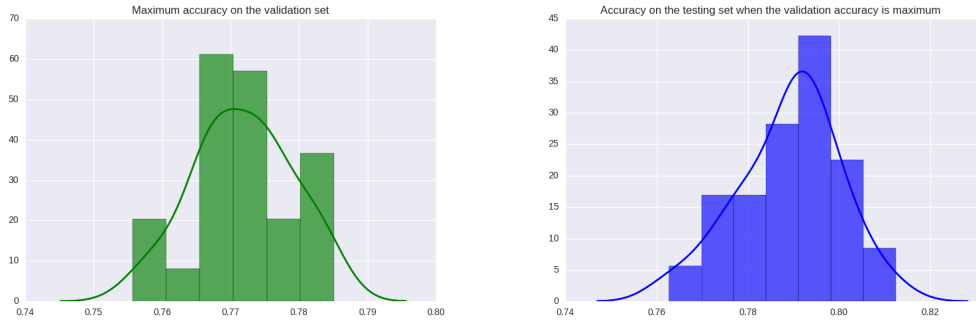
To test the robustness of the results, we ran several times the same experiment. Each time, we either chose a different train-validation-test split, either a different initialization, and we looked at the dispersion of the results.

We chose to train a model without any data augmentation, because it is the less resource consuming, and it gives the best results. We added a dropout of 50% on both the input and the hidden layer.

First, we ran 50 times the training with each time a different initialization, e.g. each weight received a different initial value. The results are presented in figure 8. If we look at the validation accuracy, we can see that the minimum is 75% and the maximum is 78%. Without any more tests, we can say that this model is performing significantly than the naive model predicting always *no*, which scores 69%. However, we can not say that we improved the results of the first experiment (a classifier with the whole Resnet representation). In fact, like we have a standard deviation around 1%, we can not hope to find significant differences of less than 1%.

The last observation we can make is about the number of epochs to reach the maximum validation accuracy. This number varies a lot from one run to another. It justifies the validation set we kept for finding the best number of epochs, even when hyperparameters are set.

We can then look at the stability of the results when the train-validation-test split is changing. As before, we run 50 times the same experiment, but with a different random seed for each run. Results are presented in figure 9. We can see that the results are a little more dispersed than before. The standard deviation of the validation accuracy is around two times as big as the one from the first sensitivity test. However, the results are still significantly superior to the naive model (69%), and not significantly different from the first



(a) Maximum validation accuracy and associated testing accuracy.

Summary of results over 50 runs

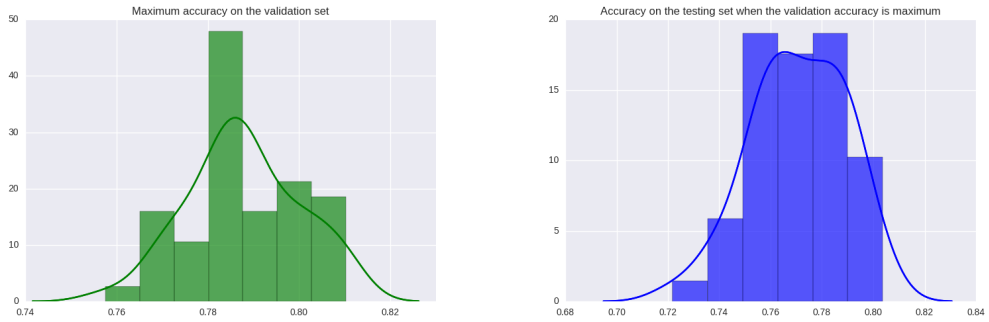
measure	validation max	testing accuracy	number of epochs
mean	0.772	0.789	652.120
standard deviation	0.007	0.011	286.375
minimum	0.756	0.763	210.000
maximum	0.785	0.812	1473.000

(b) Statistics on the validation and testing accuracy.

Figure 8: Results of 50 runs of the same experiment with different initialization.

experiment.

We can also look at the evolution of the results when we change the number of buckets in the cross validation. It means that we will increase the size of the training set and reduce the validation set if we increase the number of buckets, and we will average more results to have the final results. For 3, 5, 7 and 10 buckets, we ran 50 times the training and we look at the distribution of maximum validation accuracy averaged over the buckets. The results



(a) Maximum validation accuracy and associated testing accuracy.

Summary of results over 50 runs

measure	validation max	testing accuracy	number of epochs
mean	0.788	0.771	974.820
standard deviation	0.012	0.018	603.188
minimum	0.758	0.722	271.000
maximum	0.810	0.804	2538.000

(b) Statistics on the validation and testing accuracy.

Figure 9: Results of 50 runs of the same experiment with different train-validation-test split.

are presented in figures 9 and figures 17, 18 and 19 in annexes. We can see that the means of the maximum validation accuracy we obtained are similar, or at least not significantly different.

Finally, we can look for the impact of changing the size of the hidden layer. To do so, we trained 50 models. Each of them has a different number of units in its hidden layer. The less complex has only 6 units, the most has 300, and the others are uniformly distributed between these two bounds. The results are presented in figure 10. We can see that there is not a clear trend in these curves. On the contrary, they look like a white noise, a random signal oscillating around a mean. Our choice of 128 units in the hidden layer is so justified.

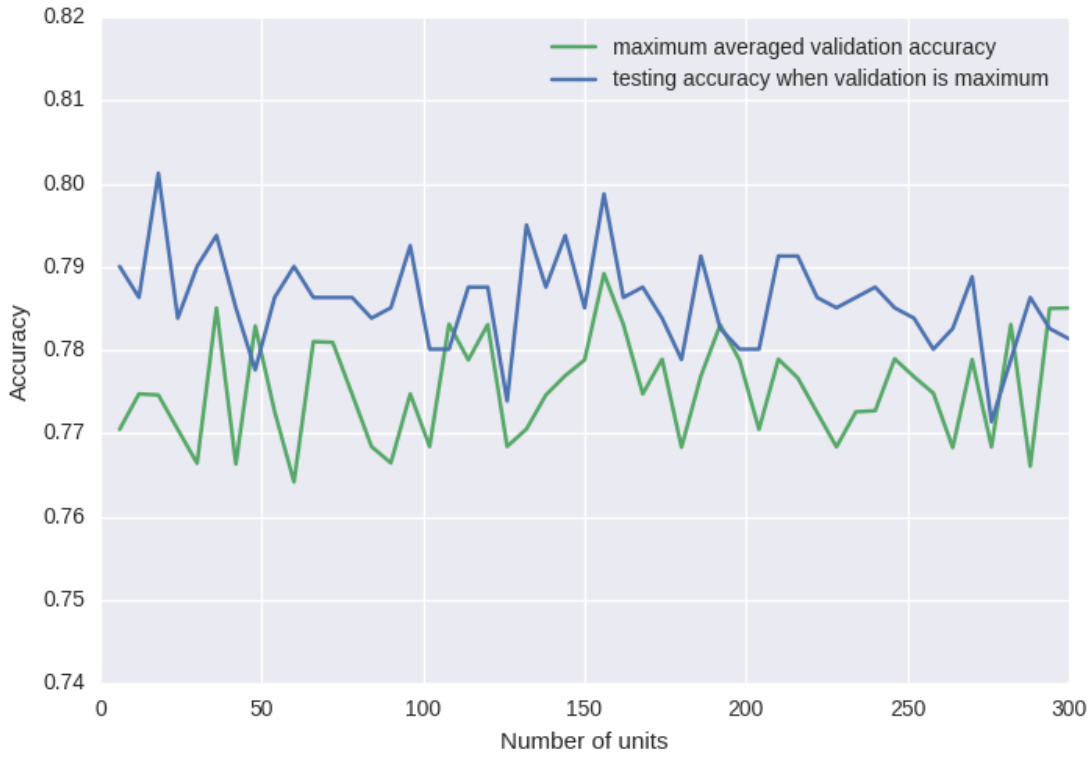


Figure 10: The evolution of maximum averaged validation accuracy and associated testing accuracy with the size of the hidden layer

With these four tests, we can conclude that we have some uncertainty in our results coming from the randomness in the training of the models. We can not control this randomness, because a part of it comes from physical reasons, like a failure of a machine or the time of calculation of one machine compared to another. However, this randomness does not have a big impact on the results and we can still conclude that we have significantly improved the results of the naive classifier, but not the results from the first experiment. We will have to test something else to obtained better results and overcome our problem of overfitting.

3.3 Approach with auto-encoders

3.3.1 Description of the model

Our second approach uses the same architecture as the previous one. However, it differs in the training step. Before, we added a layer and we trained it on the classification of a single user's interest. We do not have any insurance that the representation created with this system can be used for another user. and we are facing a big issue of overfitting. To improve the model on these two points, we created a representation which does not depend on one user's interest, and we only trained a basic classifier like a logistic regression on the top of this representation to see if it contains useful information to classify the pictures.

To train this hidden layer, we used an auto-encoder. It is a neural-network which tries to recreate the input under some constraints. The most classic one is to go through a hidden layer with a low number of units. The neural network will have to compress the information of the input into a small vector, and then to retrieve the full information from this compressed representation. An other constraint is to go through a very large hidden layer, but with a restriction on the sparsity of the representation([11]).

We applied the first constraint for our auto-encoders, and we hope that this representation which is able to reconstruct the pictures is also useful to classify them. To test this hypothesis, we added a logistic classifier which takes as input the smallest hidden layer of the auto-encoder and which aims to classify pictures according to one user's interest.

The full architecture of this neural network is presented in figure 11. This figure represents the standard model. We also tried to use a denoising auto-encoder [14]. It means that we added dropout, as in the first part, on the input, but the auto-encoder tries to recover the input without this dropout. The goal is to teach the auto-encoder to find a hidden structure in the data and to remove the noise on the input by finding the closest element from this hidden structure.

With these models, we have two relevant scores to follow during the training. First, the reconstruction error is important. The auto-encoder aims to reconstruct the input image, so its cost function is

$$C(x, \hat{x}) = ||x - \hat{x}||^2$$

with x the input and $\hat{x}$ the reconstruction given by the auto-encoder.

The second score we will follow is the accuracy of the logistic regression. In fact, the goal is to classify the pictures so this is the most important score.

3.3.2 Results

With this model, we cross-validated the complexity of the auto-encoder (number of layers and number of units in each layer) and the proportion of noise added to the input when we worked with denoising auto-encoder.

To interpret the results, we looked at the training and the validation curves. The testing one is here to give an idea of how well the model performs on data it had not seen before. However, it is not used at all to tune hyperparameters.

First, let's look at the impact of the noise.

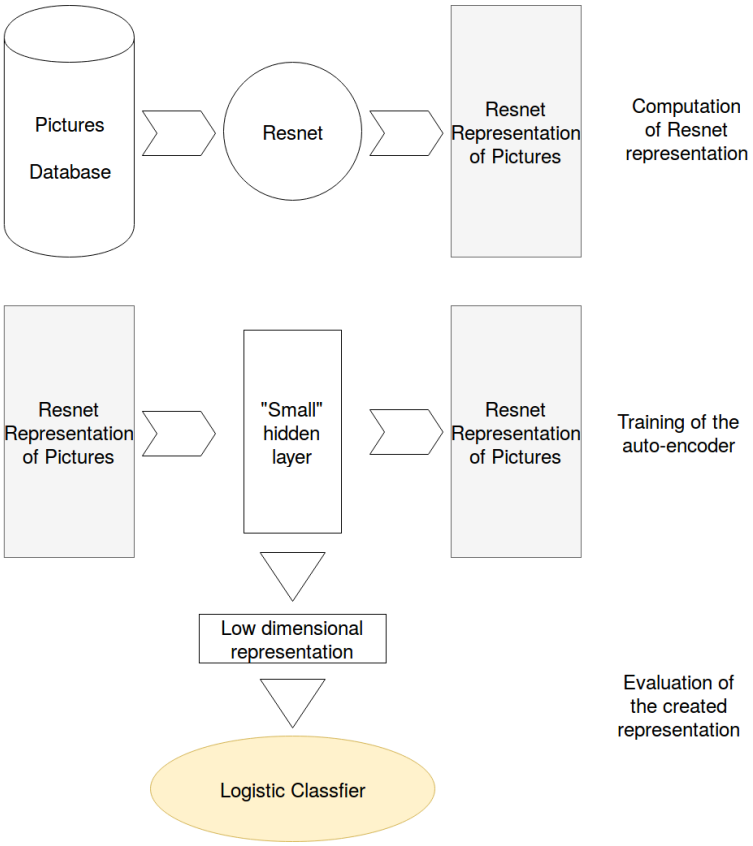


Figure 11: Schema of the approach with auto-encoders.

Dropout

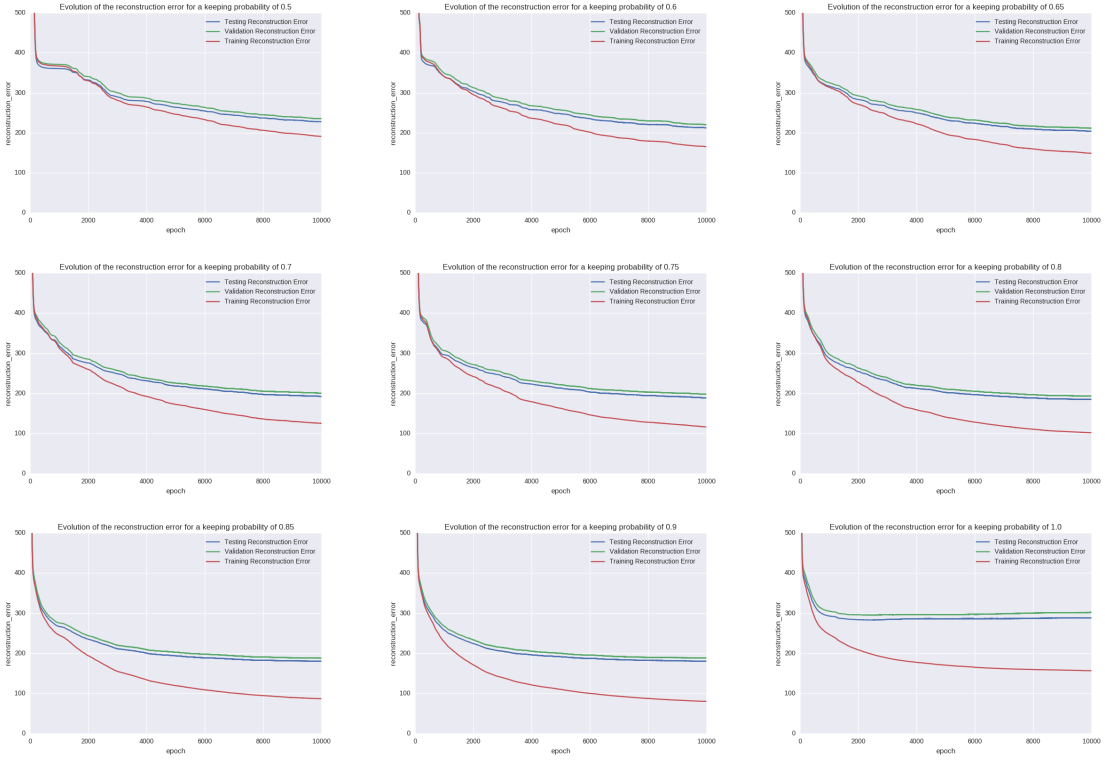
We chose an auto-encoder architecture which gave us the same representation size as the previous approach, e.g. an auto-encoder with one hidden layer of 128 units. We tried several dropout rates between 0 and 0.5. It means that the keeping probability was between 0.5 and 1. Results are presented in figure 12.

Let's first look at the first part of the figure, e.g. the evolution of the reconstruction error during the training. Between the different graphics, the only difference is the keeping probability applied on the dropout on the input of the auto-encoder. This probability takes values between 0.5 (figure on the top left) and 1. (figure on the bottom right). It means that the first figure has a much higher dropout than the last.

First, we can observe a general behavior in these graphics when there is some dropout: the model is slightly overfitting, but the validation and testing accuracies are decreasing during all the training. Moreover, the higher the keeping probability is, the bigger the gap between the training curve and the others is.

These first observations seem very logical. In fact, with a high dropout, the auto-encoder can not learn as fast as it could without dropout. When we lower the dropout, the auto-encoder can fit better the training dataset, but it will also learn some information which is not general enough to encode a picture from the validation or the testing dataset.

If we look at the last graphic, the one without dropout (keeping probability equal to 1.), we can see that the validation and the testing errors seem to reach a stable value. Moreover, this value is higher than the errors reached at the end of the training of the other auto-encoders. We can so conclude that the dropout is useful in this experiment.



(a) Evolution of the reconstruction error for different keeping probabilities.

(b) Evolution of the accuracy for different keeping probabilities.

Figure 12: Evolution of the reconstruction error and the accuracy during the training of our model with different keeping probabilities.

Finally, we can see that the models with a keeping probability between 0.7 and 0.9 are very similar, and it is not easy to say what is the perfect dropout rate for this task.

Then, let's look at the accuracies associated to each of these models. We observe the same behavior as the reconstruction errors, e.g. the model is learning slowly when the dropout is very high, and the model without dropout is the worst.

We can also add a remark on the comparison of validation and testing accuracies. We can see that the former is often much lower than the latter. We can explain this with the random split we made. In fact, the results are averaged on the five buckets of the cross validation. For each bucket, we have a different training and validation sets, but the testing set is always the same. It means that this last set might be easier to classify than the average, which could explain the higher accuracy.

To continue, we will take a dropout of 0.25, and look at the impact of the complexity of the auto-encoder.

Complexity

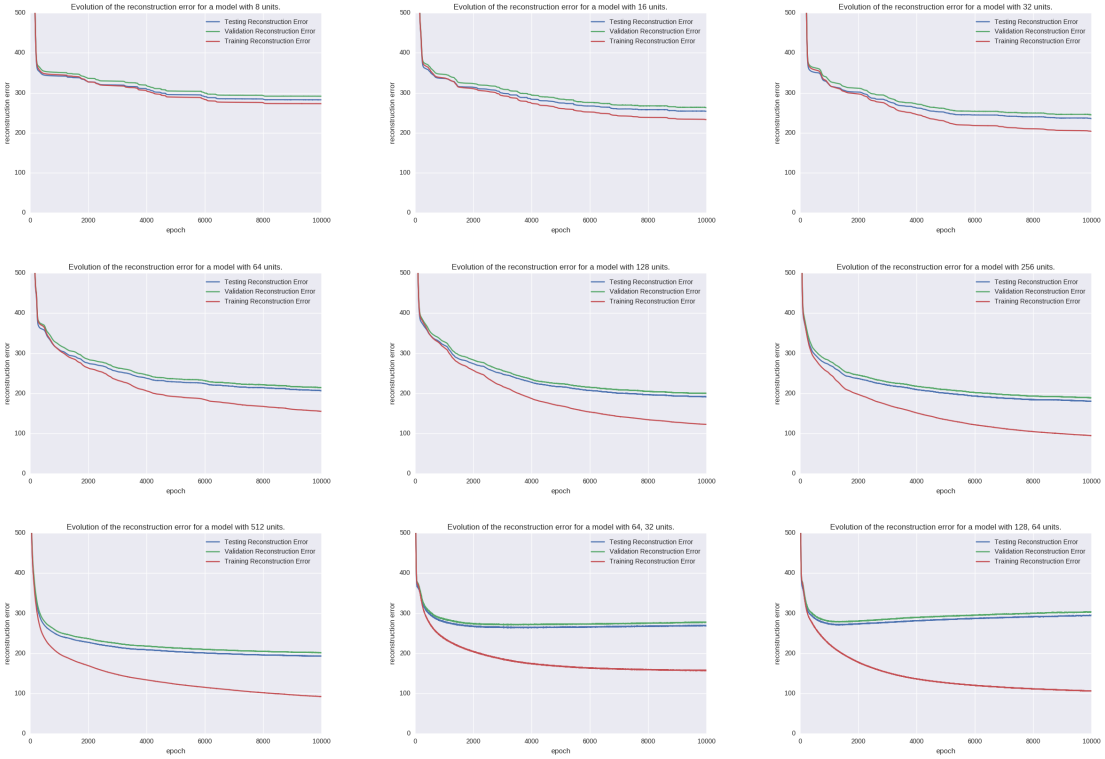
By complexity, we mean how many weights there are in the model, and how complex the function it approximates is. A model with a lot of parameters will be able to approximate better a complex function, but it also has more chance to fit the noise in the training dataset and not the general trend. Results obtained for different number of units in the hidden layer and different numbers of layers are presented in figure 13.

First of all, let's compare the number of weights in each model. If a model has one hidden layer with n units, takes a input of size n_i and returns only one number, it has $(n_i + 1) \times n + n + 1$ parameters. Logically, the number of parameters is increasing with the size of the hidden layer. It means that, amongst our seven first models, the number of weights is increasing.

The two following models have two hidden layers. The number of parameters is given by $(n_i + 1) \times n_1 + (n_1 + 1) \times n_2 + n_2 + 1$ if n_i is the size of the input, n_1 the size of the first hidden layer and n_2 the size of the second one. If we calculate these numbers, we notice that our models with two hidden layers have fewer parameters than the last model with one hidden layer of 512 units. However, the structure in two layers increases the complexity of the model.

The first observation we can make is about the overfitting. Whereas the first models do not fit better the training data than the validation one, the following models start to overfit and the last are completely overfitting. This can be easily seen in the first part of the figure with reconstruction error. In fact, in the first two models, the training curve is following the validation and the testing curves. In the five next models, the training curve goes lower than the two others, but these two others are still decreasing, whereas there are decreasing then increasing for the two last models. These two last graphics are the classic curves of overfitting and they show that either the dropout is not high enough, either the model is too complex for the problem.

In this second part of the figure, we can see the same phenomenon: the first models are underfitting, and the next models are overfitting. However, there is a big difference in the last two models. We can see that the validation and the testing accuracies are increasing then decreasing, but the training curve is not as good as we could expect. This means that



(a) Evolution of the reconstruction error for auto-encoders.

(b) Evolution of the accuracy for auto-encoders.

Figure 13: Evolution of the reconstruction error and the accuracy during the training of our model with different complexities.

the autoencoder is learning very specific details which are not relevant for the classification.

The second observation we can make is about the performance of the different models. We can see that, if we are looking at the maximum accuracy reached on the validation dataset during the training, the models with 128, 256 and 512 units are performing equally (around 76%). We so chose the less complex of these three models to continue, e.g. the model with 128 units.

Now that we created and tuned three different approaches to tackle our problem, we will compare them.

4 Comparison of the three approaches

We now have three different approaches:

- *Resnet* We add a classifier directly on the top of Resnet representation.
- *Feed Forward* We train a hidden layer with a feed forward neural network trained on a classification task.
- *Auto-Encoder* We train a hidden layer with an auto-encoder and we classify the pictures with a classic algorithm.

To compare them, we will compute the best accuracy we can get with these representations first with the labels used during the previous experiments, then with labels the models did not see during their training. It means that we will train with these new labels only the classifiers and not the representation.

To compute the best accuracy we can get, we tried several classifiers including Logistic Regression, Naive Bayes classifier, Nearest Neighbors, Ridge and Lasso classifier, Decision Tree, Random Forest, SVM and Gradient Boosting. We tried several classifiers because we thought that some classifiers which perform well on one representation will not perform as well on the others. For example, we can think that we will need to penalize a lot more the classifier with the *Resnet* approach than the two others because Resnet representation is around twenty times bigger than the two others. Consequently, depending on the applied penalization, we would benefit one approach or another.

For each one of these classifiers, we performed a quick grid search to find good hyperparameters. The entire list is given in the table 4. We kept the classifier and the hyperparameters which maximize the validation accuracy. First, let's compare these algorithms on the set of labels used previously.

Labels used during training

Results are shown in table 2. Here again, results are averaged over the five folds of a cross validation.

First, let's look at the two first approaches. We can see that the *Feed Forward* approach performs better than the first one on the three datasets. This is logical because this representation has been created for this classification task.

Approach	Train score	Validation score	Test score
Baseline	0.69	0.69	0.69
Resnet	0.90	0.77	0.79
Feed Forward	0.97	0.85	0.83
Auto-Encoder	0.84	0.83	0.75
Averaged Auto-Encoder	0.80	0.76	0.75

Table 2: Results of the three approaches with the best classifiers, and the baseline. The accuracies are averaged over the cross validation buckets.

For the *Auto-Encoder* approach, we can observe a big difference between the validation score and the test score. This is not normal. These two sets are playing a similar role during the training phase. The classifier and the auto-encoder did not see these datasets during their training. A possible explication is heterogeneities in the validation and test sets. Like we don't have a very large dataset, it is possible that the train test split we did created a testing set which is harder to classify compared to the training dataset. The auto-encoder may be more sensitive to this phenomenon that the two other approaches because it is trained on a much harder task.

In order to have an idea of how good it performs, we trained the same model with the same classifier on 20 different train-test splits, and we averaged the results over the 20 runs. Results are shown in the last line of table 2. We can see that the validation and the testing scores are much closer. However, they are lower than the other approaches.

We saw that the *Feed Forward* approach seem to work better than the others. However, it has been trained on the same labels, so it benefits from this configuration. Let's see how good they perform on different labels.

Different labels

Here, we used new labels that we did not use before. For the *Feed Forward* approach, we did not retrain the representation with these new labels. However, for these three approaches, we retrained the classifier with a cross validation. Results are shown in the table 3.

Approach	Train score	Validation score	Test score
Baseline	0.63	0.63	0.63
Resnet	0.86	0.67	0.64
Feed Forward	0.89	0.66	0.65
Auto-Encoder	0.77	0.65	0.64

Table 3: Results of the three approaches and the baseline with the best classifiers on a dataset labeled by another user. The accuracies are averaged over the cross validation buckets.

The first observation is about the baseline. With the previous labels, we could get 69% of accuracy by predicting always *no*, whereas we can get only 63% of accuracy with these new labels.

The second observation is about the validation and the testing accuracies. They are similar for the three approaches, which means that we achieve to create a representation with 20 times fewer dimensions than Resnet representation, but which is as useful for the classifi-

cation of one user's interest. However, it also means that we did not create a representation enable to discriminate well our dataset according to any user's interest.

Finally, we can also observe that the *Auto-Encoder* approach, which was created to perform better than the *Feed Forward* approach with a different user, does not perform better. It seems that the later performs better on the user it has been trained on, and equally on a different user.

Now let's compare these three approaches on the locality property.

Locality

Like we said during the description of the problem, we want our representation to keep some localities. It means that it should group together the samples which share the same characteristics. However, the data are in a very high dimensional space (128 dimensions for our "low" dimensional representation).

To look at this property, we have to find a way to reduce the dimension but keeping local and global structures present in the data. The best algorithm we found is called TSNE ([10]).

To explain quickly how it works, TSNE algorithm consists in creating for each couple of points a probability to be neighbors, and to find points of a d-dimensional space which respect these probabilities. Usually, d is set to 2 or 3, in order to be easily visualized. TSNE finds these points by minimizing an appropriate cost function with respect to the coordinates of these new points.

This algorithm is not perfectly adapted to our problem. We would like a supervised algorithm in order to take into account the labels of the data. However, this is the best we found. Another possibility is metric learning ([15]), but we did not find a satisfying algorithm.

The figure 14 shows the points given by this algorithm for the three different representations we have.

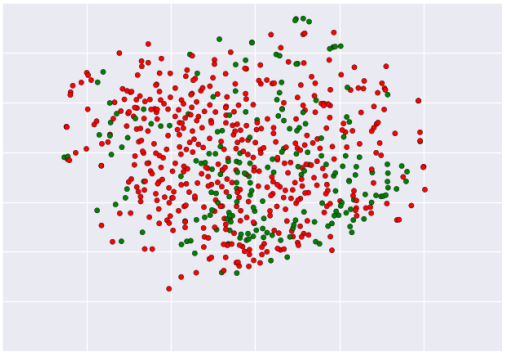
We can see that the *Resnet* and the *Auto-Encoder* approaches do not seem to have a good structure separating the positive from the negative samples. This could come from the TSNE which does not fit enough the goal we want, e.g. representing the data in a way to separate the positives from the negatives, or from our representations. The latter explanation is a lot more likely, because we did not pay any attention to this property.

For the *Feed Forward* approach, we can see that it has a good locality property with labels it saw during its training. However, as soon as we put different labels, the locality disappears. Having a good locality will be the work of the following days/weeks/months/years.

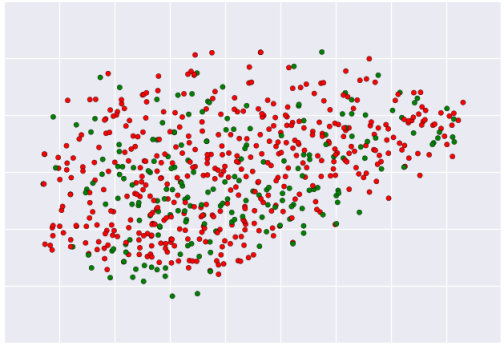
Conclusion

We wanted a representation of pictures which satisfy three properties: dimensionality, customization, and locality.

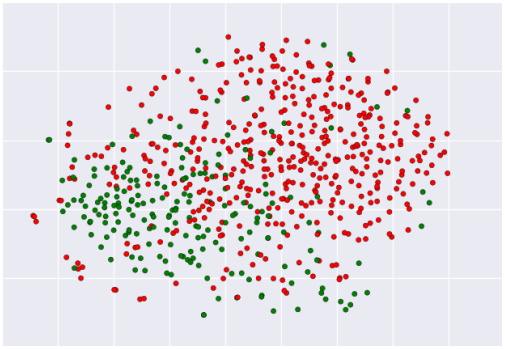
We created a representation which achieves at least as good as Resnet representation with twenty times fewer dimensions. However, we discovered that the representation given



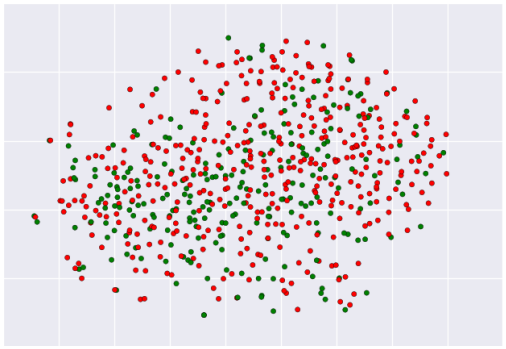
(a) TSNE of the Resnet representation



(b) TSNE of the representation from the auto-encoder approach



(c) TSNE of the representation from the Feed Forward approach



(d) TSNE of the representation from the Feed Forward approach with labels from another user

Figure 14: TSNE on our three representations.

by Resnet is not customized enough for our problem. Moreover, we did not work on the locality property, and this property is not automatically satisfied.

During the following weeks, we have several tasks planned. First, we will change our dataset to have more data. Then, we will look for a more precise representation, maybe with the help of collaborative filtering. We will also look for labels coming from other users in order to have a more precise idea of the generalization power of our representation. Finally, we will work on the locality property, and on an algorithm to visualize this property.

On a more personal perspective, I am really grateful to the CEDAR team, who welcomed me, shared with me these few months and gave me the taste of research.

References

[1] Marc Antonini, Michel Barlaud, Pierre Mathieu, and Ingrid Daubechies. Image coding using wavelet transform. *IEEE Transactions on image processing*, 1(2):205–220, 1992.

[2] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. *Computer vision–ECCV 2006*, pages 404–417, 2006.

[3] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2414–2423, 2016.

[4] Carlos A Gomez-Uribe and Neil Hunt. The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)*, 6(4):13, 2016.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[6] Chetak Kandaswamy, Luís M Silva, Luís A Alexandre, and Jorge M Santos. High-content analysis of breast cancer using single-cell deep transfer learning. *Journal of biomolecular screening*, 21(3):252–259, 2016.

[7] Yan Ke and Rahul Sukthankar. Pca-sift: A more distinctive representation for local image descriptors. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 2, pages II–II. IEEE, 2004.

[8] David G Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.

[9] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[10] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

[11] Andrew Ng. Sparse autoencoder. *CS294A Lecture notes*, 72(2011):1–19, 2011.

[12] Md Mostafijur Rahman, Shanto Rahman, Rayhanur Rahman, BM Mainul Hossain, and Mohammad Shoyaib. Dtcth: a discriminative local pattern descriptor for image classification. *EURASIP Journal on Image and Video Processing*, 2017(1):30, 2017.

[13] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2564–2571. IEEE, 2011.

[14] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

[15] Eric P Xing, Michael I Jordan, Stuart J Russell, and Andrew Y Ng. Distance metric learning with application to clustering with side-information. In *Advances in neural information processing systems*, pages 521–528, 2003.

5 Annexes

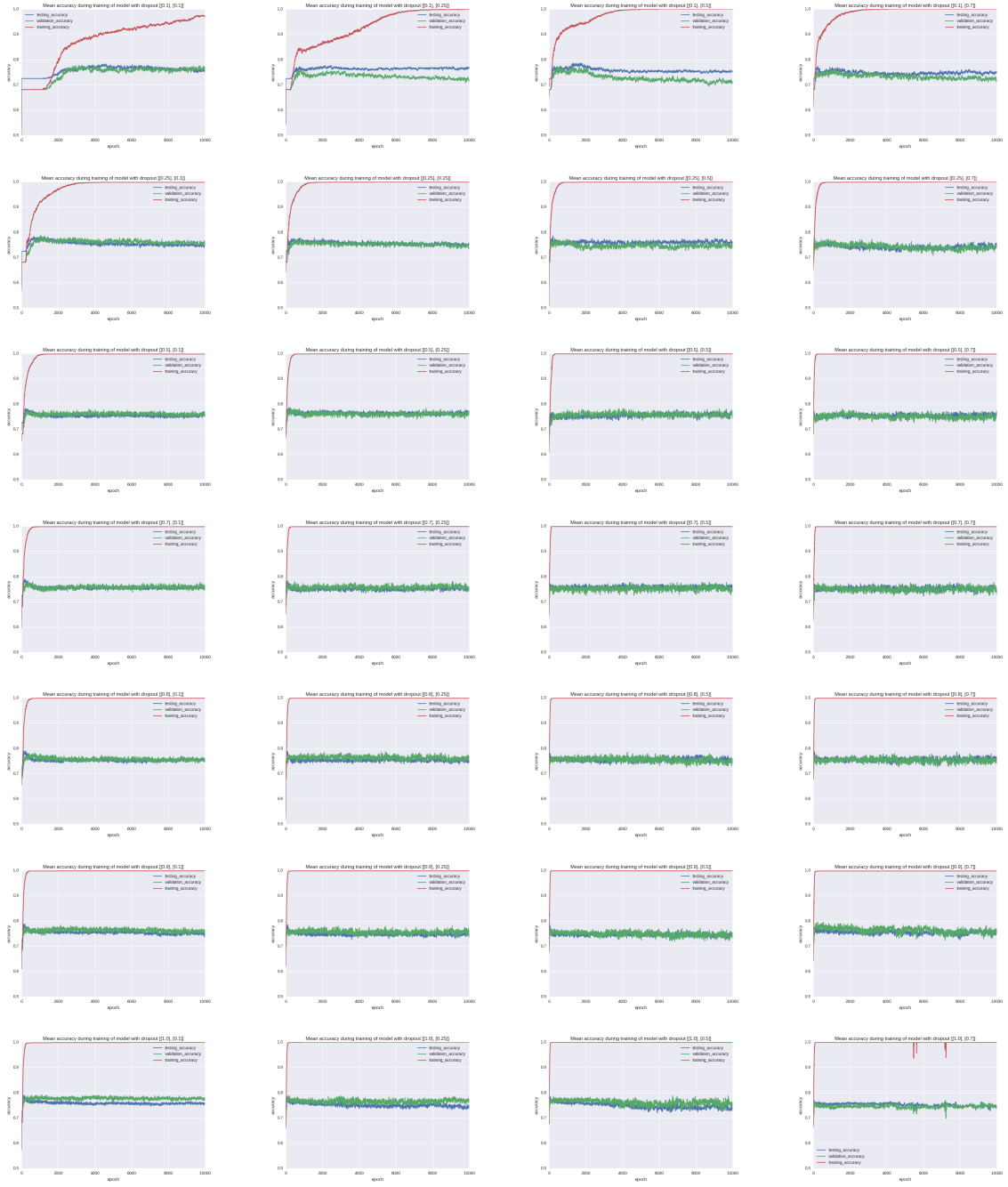


Figure 15: Mean accuracy during training with different dropout of our naive approach part 1.

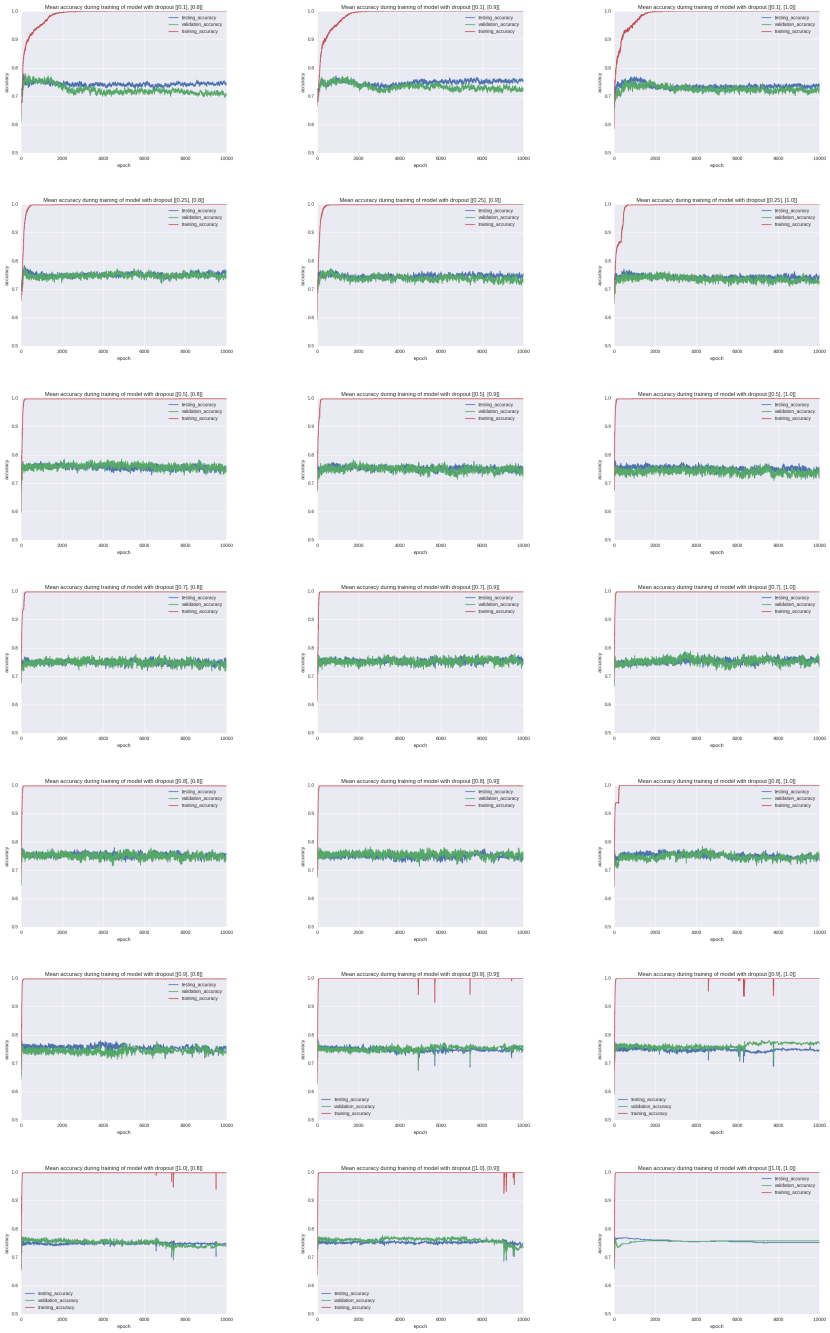
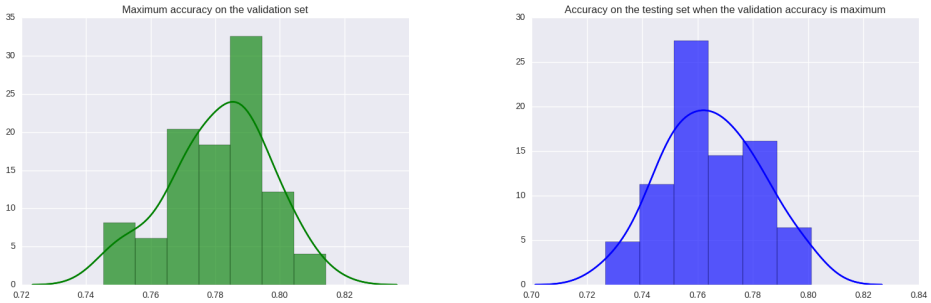


Figure 16: Mean accuracy during training with different dropout of our naive approach part 2.



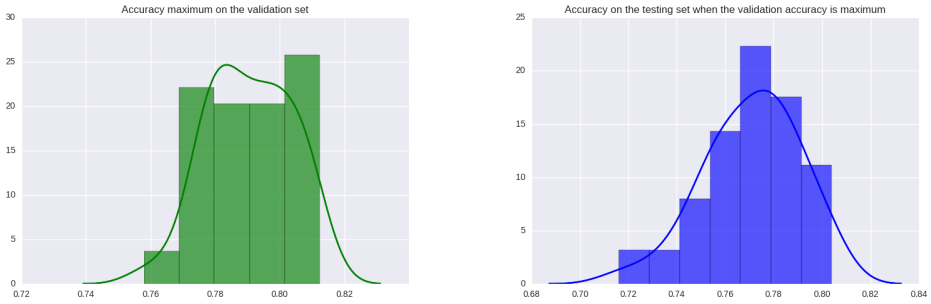
(a) Maximum validation accuracy and associated testing accuracy.

Summary of results over 50 runs

measure	validation max	testing accuracy	number of epochs
mean	0.781	0.765	889.480
standard deviation	0.016	0.017	480.900
minimum	0.745	0.727	255.000
maximum	0.814	0.801	2493.000

(b) Statistics on the validation and testing accuracy.

Figure 17: Results of 50 runs of the same experiment with 3 buckets



(a) Maximum validation accuracy and associated testing accuracy.

Summary of results over 50 runs

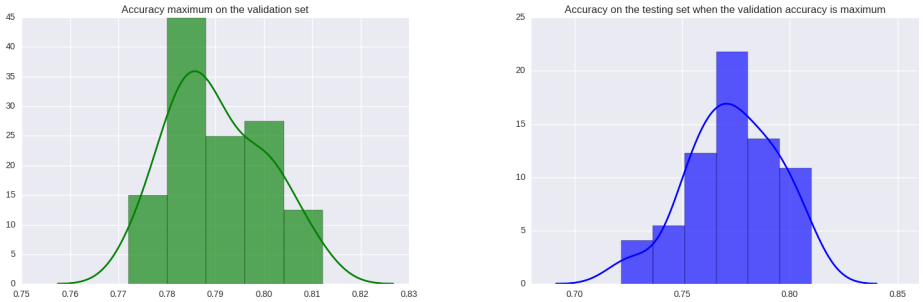
measure	validation max	testing accuracy	number of epochs
mean	0.791	0.771	873.820
standard deviation	0.013	0.020	439.112
minimum	0.758	0.716	216.000
maximum	0.812	0.804	1974.000

(b) Statistics on the validation and testing accuracy.

Figure 18: Results of 50 runs of the same experiment with 7 buckets

Algorithm	Hyperparameters
Decision Tree	Criterion: [”gini”, ”entropy”], splitter: [”best”, ”random”], max_depth: [None, 2, 4, 8, 16], min_samples_split: [2, 5, 10, 20], min_samples_leaf: [1, 5, 10, 20], max_features: [None, ”sqrt”, ”log2”]
Random Forest	n_estimators: [5, 10, 20, 50, 100], criterion: [”gini”, ”entropy”], max_depth: [None, 2, 4, 8, 16], min_samples_split: [2, 5, 10, 20], min_samples_leaf: [1, 5, 10, 20], max_features: [None, ”sqrt”, ”log2”], bootstrap: [True, False]
Perceptron	hidden_layer_sizes: [(100,), (32,), (34,), (200,), (32, 16), (64, 32), (100, 32)], activation: [”identity”, ”logistic”, ”tanh”, ”relu”], solver: [”lbfgs”, ”sgd”, ”adam”], alpha:[0., 0.0001, 0.001, 0.01, 0.1, 1.], learning_rate: [”constant”, ”invscaling”, adaptative”], learning_rate_init: [0.0001, 0.001, 0.01, 0.1], max_iter:[1000]
Adaboost	n_estimators: [10, 25, 50, 100, 200], learning_rate: [0.001, 0.01, 0.1, 1., 10.]
Naive Bayes	None
Ridge	alpha: [0.001, 0.01, 0.1, 1., 10., 100.], fit_intercept: [True, False], normalize: [True, False], solver: [”auto”, ”svd”, ”cholesky”, ”lsqr”, ”sparse-cg”, ”sag”]
Logistic Regression	penalty: [”l1”, ”l2”], C: [0.001, 0.01, 0.1, 1., 10., 100.], fit_intercept: [True, False]
Gradient Boosting	loss: [”deviance”, ”exponential”], learning_rate: [0.001, 0.01, 0.1, 1., 10., 100.], n_estimators: [10, 25, 50], criterion: [”friedman_mse”, ”mse”, ”mae”], max_depth: [1, 3, 4, 8, 16], min_samples_split: [2, 5, 10, 20], min_samples_leaf: [1, 5, 10, 20], max_features: [None, ”sqrt”, ”log2”]
Linear Model	loss: [”hinge”, ”log”, ”modifier_huber”, ”squared_hinge” ”perceptron”, ”huber”, ”epsilon_insensitive”, ”squared_epsilon_insensitive”, ”squared_loss”], penalty”: [”none”, ”l1”, ”l2”, ”elasticnet”], alpha”: [0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1, 1.], l1_ratio”: [0.01, 0.05, 0.1, 0.15, 0.2, 0.3, 0.4, 0.5], fit_intercept”: [True, False], learning_rate”: [”constant”, ”optimal”, ”invscaling”], eta0”: [0.00001, 0.0001, 0.001, 0.01, 0.1]

Table 4: List of the tested algorithms during the grid search and there hyperparameters.



(a) Maximum validation accuracy and associated testing accuracy.

Summary of results over 50 runs

measure	validation max	testing accuracy	number of epochs
mean	0.790	0.773	904.580
standard deviation	0.010	0.021	483.788
minimum	0.772	0.722	288.000
maximum	0.812	0.810	2168.000

(b) Statistics on the validation and testing accuracy.

Figure 19: Results of 50 runs of the same experiment with 10 buckets

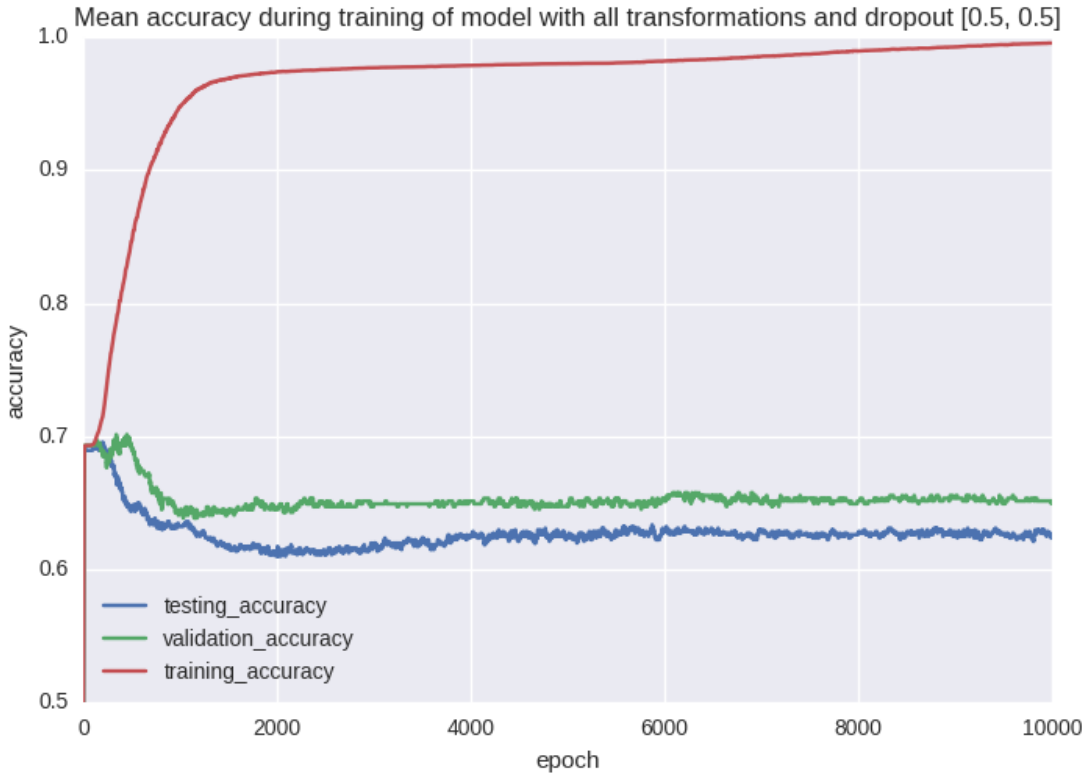


Figure 20: Evolution of the accuracies during the training of a neural network with every augmented pictures.