‘Learning convolutional Neural Network”

Beatriz Gómez Ayllón

April 6, 2017

Abstract

Add abstract

Contents

Part I

MOMORY

Chapter 1

Introduction

1.1 Introduction

This document blaaa blaaa

1.1.1 Anti-spoofing

bla bla bla

1.2 Programming language and frameworks

1.2.1 Python

Python is a object-oriented programming language and it is used to develop the experiments necessaries is Python, more specifically, it has been used the 2.7 Python version.

1.2.2 Theano and others frameworks

Theano is the main framework used to develop the deep learning code. But others libraries has been used to, NumPy,Scikit-learn and matplotlib are the most relevant. In addition to this, other Python packages has been used to like Pickle.

Theano

Theano [1] is a Python library that allows users to work with mathematical expressions and work with simbolic variables, moreover, Theano handles multidimensional arrays efficiently. This framework has been used in order to build convolutional neural networks architecture and its training procedure.

There are numerous open-source deep-libraries that have been built on top of Theano, for example Keras, Lasagne and Blocks. Although the usability of using those libraries in stead of Theano is bigger, Theano is more flexible when user wants to develop its own layer, for example.

Theano is not the unique language oriented to deep learning, for example Google, has developed its deep learning language called TensorFlow; Caffe and Torch are others examples.

Theano main page is available in http://deeplearning.net/software/theano/ where the documentation, examples,.. could be found.

NumPy, Scikit-learn and Matplotlib

NumPy is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more. The documentation of this library is available in https://docs.scipy.org/doc/numpy/.

Scikit-learn is an open source Python library and commercially usable that implements a range of machine learning, preprocessing, cross-validation and visualization algorithms which has been build in top of Numpy and matplotlib. Its documentation is available in the following url http://scikit-learn.org/dev/index.html.

Matplotlib is a python library for 2D plotting.

1.3 Databases

Some databases has been used in order to learn, compare results and carry out the project. All the databases are formed by three subsets whose samples are not repeated among subsets: train, validation and test.

- The training subset is used to train the network during epochs. To know how the training behavior, a cost is calculated.
- The validation subset is used to check the behavior of the network while is training, also, the validation subset is usually used to calculate the hyper-parameters of the network, although the hyper-parameters are not calculated until is pointed.

The validation error is calculated for each training epoch. The metric use to the validation is $error(\%) = cost * 100$.

– The test subset, that is used just at the end of the training. The best model is chosen with regard to the best validation error. Different metrics are going to be used for after testing the network (error (%), TP, FP ...).

1.3.1 MNIST digit database

MNIST digit database is a image database of human written digits. This database is commonly used to learn machine learning techniques. Because of that, this database has been used in order to learn Theano and convolutional neural networks. In addition, this database has been used in a implemented convolutional neural network (LeNet).
Some examples of the digit image MNIST database could be seen in 1.1 and the charac-

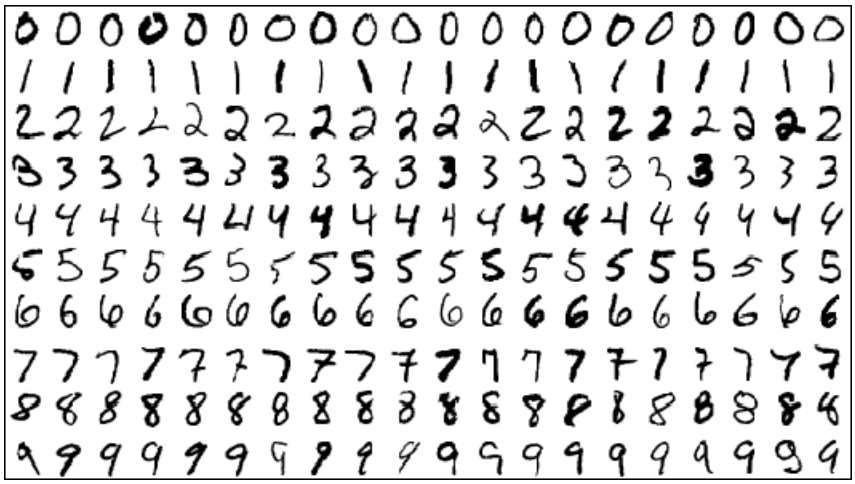


Figure 1.1: digits MNIST images database

teristics of this database are the following ones:

– Number of samples: 70.000 number of unique samples.

– Number of features/ length of each image: 784

– Number of classes: 10, one per digit

– The size of each image is 28x28 pixels.

– The images are in grey scale.

1.3.2 Labeled faces in the wild

The Labeled Faces in the Wild is a well-known and used dataset of faces images that can be found in its official web page http://vis-www.cs.umass.edu/lfw/.

The characteristics of this database are the following ones:

- Number of samples/ Number of images : 13233
- Number of features/ length of each vectorized image: 187500
- Number of classes / Number of people: 5748
- The number of images per person is not the same for each one.
- The size of each image is 250x250 pixels.
- The images are RGB ones.
- The faces in images are in the center of the image.

Figure 1.2: Examples of Labeled faces in the wild imamges

Examples of images of this database could be seen in figure 1.2 in which faces of well-known people are visualized.

This database has been used to learn; to learn how to read a database, how to feed the network with those images.

1.3.3 FRAV dataset

FRAV database is an anti-spoofing face database built in the research group of tu URJC called FRAV. This dataset is formed by five different classes:

- Original images of people.
- Images of people printed (attack).
- Images of people with a mask (attack).
- Images of pleople with a mask with the eyes cropped (attack).
- Images of people in a tablet (attack).

There are the same number of samples in each class. The images of that classes can be found in RGB and NIR (not all RGB images has its corresponding NIR image). Characteristics of FRAV images ere the following ones:

- There are 939 people in each RGB class or 195 in NIR class.
- There is one image per person.
- Each image h as it is own shape.
- The faces in images are in the center of the image.

This images has been used in different ways, in some experiments, just RGB images has been used, so 939 images has been used. When has been used RGB and NIR images at the same time, 195 images has been used (195 RGB and its corresponding NIR). To classify, two different ways has been used; the first one where real people formed one class and the different attacks formed other class, so two classes have been used; and the second way where each attacks correspond with a class, so five classes (4 attacks and 1 real) have been used.

One exaple of RGB images of FRAV database are shown in figure 1.3 and another example could be seen in 1.4. In both images, the four attacks described previously and the real user could be visualized.

Figure 1.3: Four attacks and real user from RGB FRAV database

Figure 1.4: Four attacks and real user from RGB FRAV database

1.3.4 CASIA dataset

Casia is a face anti-spoofing database built in the center for Biometrics and security research.

In the same way as FRAV dataset, this database is formed by real or genuine images of people and three different attacks of the same people:

- Images of people printed (attack).
- Images of people with a mask (attack).
- Images of people with a mask with the eyes cropped (attack).
- There are 939 people in each RGB class or 195 in NIR class.
- There is one image per person.
- The size of each image is 720x1280 pixels.
- The faces in images are in the center of the image.

– Images are RGB.

Two people of Casia database could be seen in figure 1.6 and figure 1.5, where three attacks types could be seen with the real user of both examples.

Figure 1.5: Three attacks and real user from casia database

Figure 1.6: Three attacks and real user from casia database

1.3.5 MSU - MFSD database

The MSU Mobile Face Spoofing Database (MFSD) is a video face anti-spoofing database although for this work just images have been used.

The database consist on users and attacks of the same people:

– Genuine users

–

The characteristics of the database are the following ones:

– 35 images per attack or genuine user.
– Images are RGB.
– Faces are centered in images.
– The size of each image are not equal. Approximately images are 300px heigh and 335px width.

In figure 1.7 and figure 1.8 It is posible to see the three attacks and the real user.

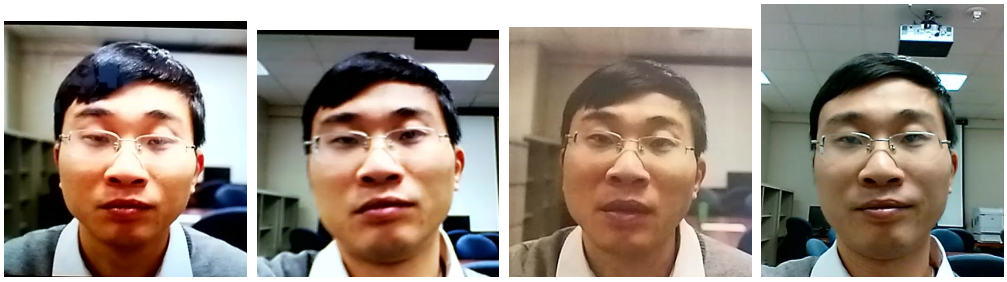


Figure 1.7: Three attacks and real user from a user of MFSD database

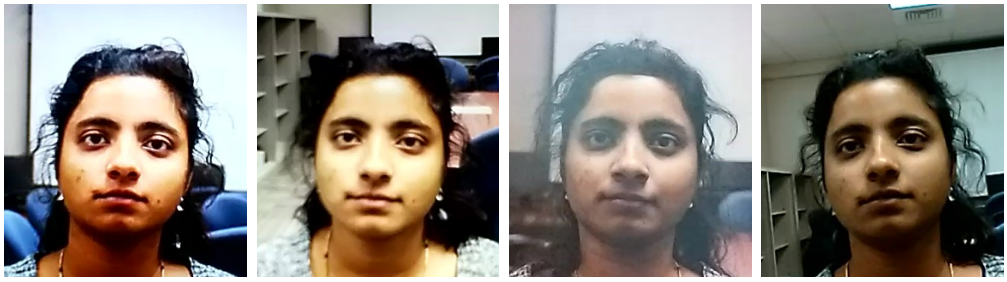


Figure 1.8: Three attacks and real user from a user of MFSD database

1.4 Metrics

In this section, the metrics that have used to express the results are going to be shown and explained.

Real / Classified	Positive	Negative
Positve	TP	FN
Negative	FN	TN

Table 1.1: Confusion Matrix

1.4.1 Cost and Error

The first parameter that is used is the cost. The cost is used while the neural network is training. The lower value, The better performanceof the networok.

More specifically, for training the neural ne

1.4.2 True Positives (TP), True Negatives (TN), False Positives (FP), False Negatives (FN)

True Positives (TP), True Negatives (TN), False Positives (FP), False Negatives (FN) are metrics that are used for bi-classes problems, positive class and negative class. The positive class is real users, genuine user and bonafide and negative class is the attack clas.

Those metrics, aro gotten when a positive or negative samples is well or missclasified [2].

If a positive sample is classified as positive is a true positive (TP), but if it has been classified as negative is a false negative (FN).
If a negative sample is classified as negative is a true negative (TN), but if it has ben classified as positive, is a false positive (FP).

From those four metrics, it could be extracted the confusion matrix for binary classification, explained in 1.1:

1.4.3 ROC curve

From the confusion matrix, it is posible claculate others parameters [2]: precision, recall, specificity, accuracy because its values depend on TP, TN, FP, FN as could be se in it respective equation:

$$precision = \frac{TP}{TP + FP} \tag{1.1}$$

$$recall = \frac{TP}{TP + FN} \tag{1.2}$$

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{1.3}$$

The ROC curve is

1.5 Precision and Recall

1.5.1 APCR and BPCR

The ISO/IEC 30107-3 [3]

Attack Presentation Classification Error Rate (APCER) is defined as the proportion of presentation atacks that has been classified incorrectly (as *bona fide* presentation.)

$$APCER_{PAIS} = \frac{1}{N_{PAIS}} \sum_{i=1}^{N_{PAIS}} (1 - Res_i) \tag{1.4}$$

Bona fide Presentation Classification Error Rate (BPCER) is defined as the proportion of *bina fide* presentations incorrectly classified as presentation attacks.

$$BPCER = \frac{\sum_{i=1}^{N\_BF} RES_i}{N_{BF}} \tag{1.5}$$

where:

- N_BF is the number of *bona fide* presentations
- Res_i is 1 if iˆth presentation is classified as an attack and 0 if is classified as a *bona fie* presentation.
- N_PAIS is the number of attack presentations

Chapter 2

Methodology

2.1 LeNet-5

LeNet-5 [4] is the name of a cercain architecture of a convolutional network designed for handwritten and machine-printed character recognition. The net has been proved with MINST database, a database of handwritten digits that comes with the net.

The basic architecture of Lenet is two convolutional layer, followed each one by a max pooling layer and then a fully-connected layer. This architecture could be visualized in figure X.

LeNet has been used to learn the implementation of convolutional neural networks, and LeNet is the basis of the network developed to carry out this project as it has been modified in order to adapt it to get this project goal. The code of LeNet-5 in Python could be found in www.deeplearning.net

The specifications of the downloaded LeNet code are the architecture of LeNet used for starting to work with this project is formed by two convolutional layers of size 5x5 and with 20 kernels in the first convolutional layer and 50 in the second one, those are followed (each one) by a max pooling-layer of size 2x2. Those four layer are followed by a fully-connected layer with 500 neurons at the output. The classifier which has been used is the logistic regression. The activation function of the convolutional layers and the logistic regression is tanh.

The learning rate is 0.1 and the network runs by 200 epoch. The data is split in thre sets: training, testing and validating. For each subset, when is fed to the network it is fed in batches, whose size is choosen by the user. Using batches is to prevent memory problems.

The training procedure is being realized for too many epochs as users has selected.

While the trianing is being running, the validation is calculated for each epoch.

www.deeplearning.net is the original page of Theano where could be found tutorials, how to install and get it and the documentation of its functions.

2.1.1 Structure of Lenet

LeNet works basically with theano although it uses others libraries like NumPy.

First of all in Lenet the data is loaded. The function that download the data check, firstly, if the data is not downloaded yet, if it downloaded it does not repeat the download. The data is downloaded split into train, test and validation subsets.

After loading the data, the architecture is defined, the layers are called because they has been defined as objects, each kind of layer, one different object. There is a object for convolutional + maxpooling, another object for the hidden layers that is a fully connected layer, and the classifier used that is logistic regression. Each part of the code could be found in www.deeplearning.net

After creating the structure, the functions of training, validating and testing are created as Theano functions.

In a big while loop the training validation and testing is developed. The data is trained. Each mini-batch (that is a small quantity of that given by the user, it is used in order to not train or test all the data together because it would take too much computer resources) is trained and the weights and bias of each layer are updated. It is given a validation frequency, this parameters decides how many validations are produced. So the data is trained and validation, when a best score of validation is produced, the data is tested.

The training would run for a number of epoch that the user has given or by a early-Stopping that has been developed by authors. The early-stopping combats over-fitting by monitoring the model's performance on a validation set.

2.1.2 Results of LeNet

The result of the network using the given database could be seen in figure 2.1, where the least error in validation performance is 0.91 % obtained at iteration 18300, with an error test performance of 0.92 %.

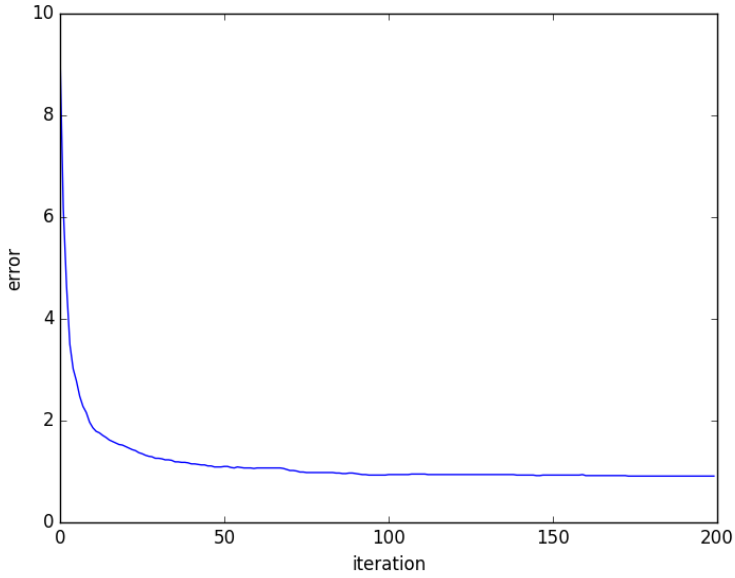


Figure 2.1: Validation error obtained with Lenet and MNIST digits database

It is important optimize the cost function, so it is possible to analyze how the networks works looking the cost function. The cost produced at the training gets reduced with the increasing iterations, as it is seen in figure 2.2.

As it is known, weights are filters, twenty first weights of each convolutional layers are shown in figure 2.3 of the tenth and hundredth epoch.

While the network is learning, weights change and they would not have the same apparence if were represented.

Each time we take a sample and update our weights it is called a mini-batch. Each time we run through the entire database, it's called an epoch. Batch size determines how many examples you look at before making a weight update. The lower it is, the noisier the training signal is going to be, the higher it is, the longer it will take to compute the gradient for each step. http://stats.stackexchange.com/questions/140811/how-large-should-the-batch-size-be-for-stochastic-gradient-descent.

To be sure that the batch size do not affect to the results, it has been changed, in first place into 20 and in second place into 100. In the fist example (when batch size = 20), the while has been break because of early-stopped at epoch 31, from epoch 16 the networks does not improve the validation result.

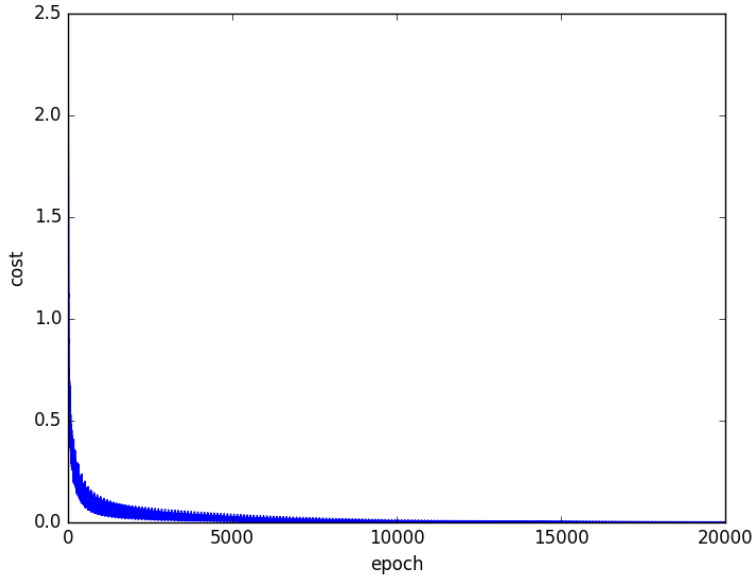


Figure 2.2: Cost function of running Lenet with it is own database.

In figure 2.4 the error in each epoch is represented for 500 batch size, the original size, for a value of 20 and 100. In the original case, the error stars with a value of 9% aprox. with the batch size = 20 the error in the first iteration is about 2.4%, and with a bunch of 100 images, the validation error is 3.5%.

With the original size and size equal to 20, it is possible to get to the same minimum, the difference between those examples is that each one get to that conclusion into different epochs. With a batch size equal to 100, the code stopped because of the early-stop with a patience of 10000; it stop in epoch 33,while those epochs, it has been possible to get to a test error of 1.04% in iteration 8500 when the validation error was 1.01%, it has been running with put getting a better validation score for 17 epoch. With a batch size = 20, the code also has stopped earlier because of the same reason, but in that case it has been possible to get to the same minimum that with the original size; the epoch in which has stopped is 31, it has been running without getting a better validation score for 15 epochs.

If the patience is increased from 10000 to 100000 for a batch size = 20 and equal to 100, results obtained are that for a value of 20, it has been running for 40 epoch, having the same results that the one obtained with a patience of 10000.

For a value of 100 with the patience increased to 100000, the results are not the same as the one obtained the patience equal to 10000. In this occasion, the results are better than in the original one; The code has run for 258,08 minutes and for 200 epochs. The result obtained was better than in the original one has it has said. In epoch 51

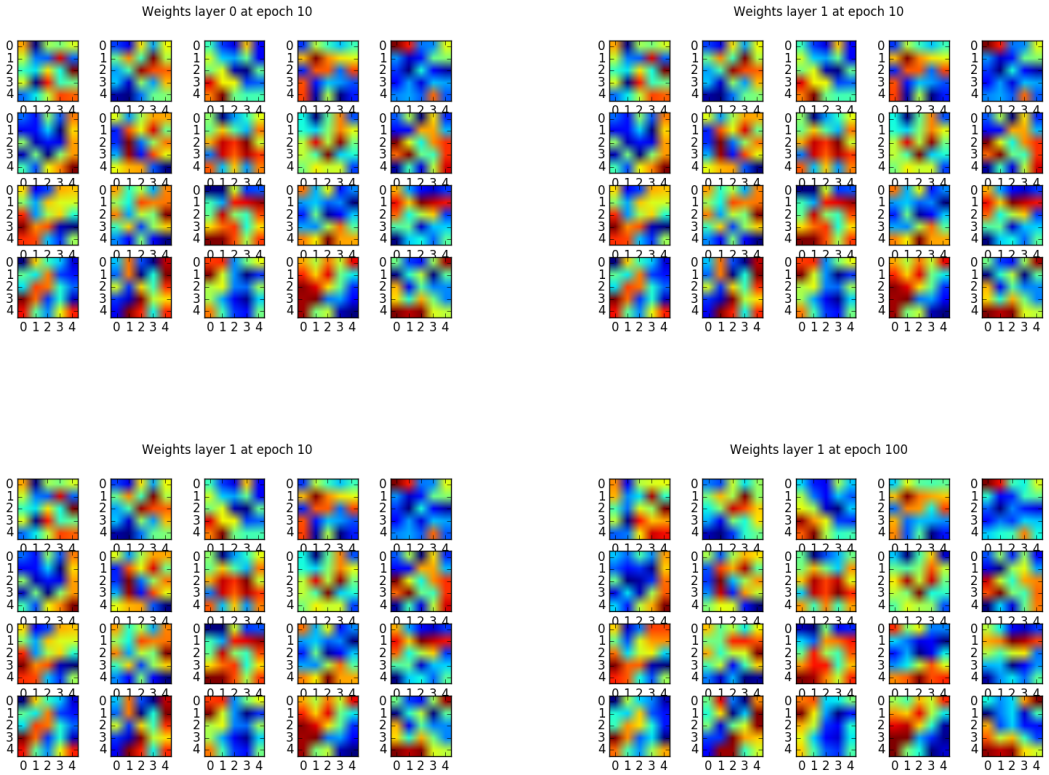
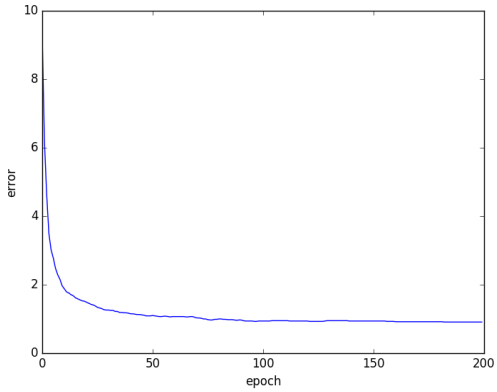


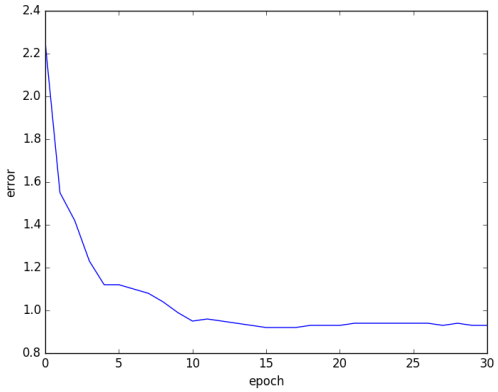
Figure 2.3: Weights at epoch 10 and 100 of the two convolutional layers

has gotten a validation error of 1 % and the error test has been 0,89%, from this point, the test error that has been calculated five more times, has been increasing until get the value 0,85% in iteration 55500, epoch number 111, the validation error obtained has been 0,95%. From this epoch until number 200, the network has not had any better result.

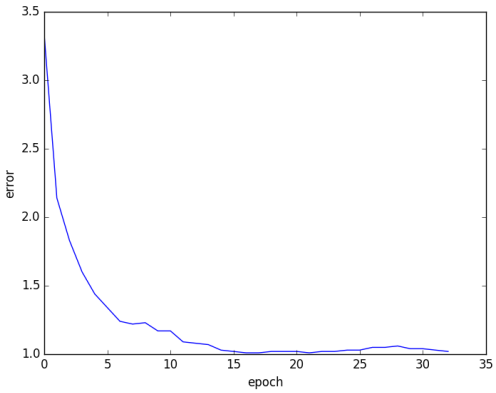
In figure X it is possible to see the error function for those two examples. It could be seen that the example with batch size = 20 has just run for 40 epoch and the good results of the example with batch size = 100, remembering that the patience has been increased from 10000 to 100000.



(a) batch size = 500 (original size)



(b) batch size = 20



(c) batch size = 100

Figure 2.4: Validation error in each epoch for diferentes sizes of batches.

2.1.3 Modifying LeNet

From LeNet, with the database gray scale digit database, the one that LeNet uses in the original example (grey scale images whose size is 28x28), some modification has been realized:

- Using Local Response Normalization (LRN) In which a normalization has been carried out in the convolutional-max pooling layers.
- Using ReLu as activation function: The activation function tanh has been substituted by ReLu activation function in convolutional and fully connected layers.
- Using Relu and LRN: The activation function used is ReLu and LRN has been used as normalization layer.
- Change weights initialization: Weights initialization has been changed by Gaussian. In which mean value that has been used is 0 and std is 0.01. Weights initialization has been changed in convolutional and fully connected layers. Also, bias initialization has been changed by ones.

First, the cost of training process are going to be visualized with the original cost train, without modifying LeNet). In figure 2.5 is represented. Also the validation error (validation cost*100) could be visualized in figure 2.5. The network has been running for 200 epochs.

Visualizing the loss during the training, could be affirmed that the network with Gaussian initialization is in a local minimum because the loss has converged as could be seen in figure 2.5(e). The loss of LeNet without being modified (figure 2.5(a)) is the one whose oscillation at training is less than others.

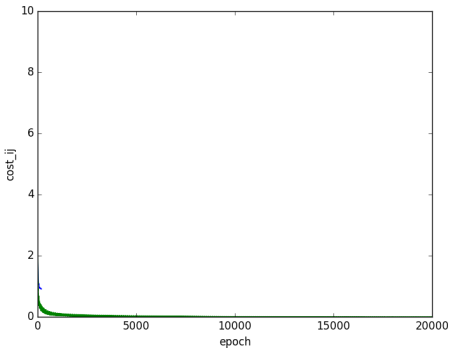
About the error at validation, visualizing the graphs in figure 2.5, it is very similar the curve for original lenet, lenet with ReLu, lenet with LRN and lenet with LRN and ReLu.

At testing, the results obtained have been the following ones:

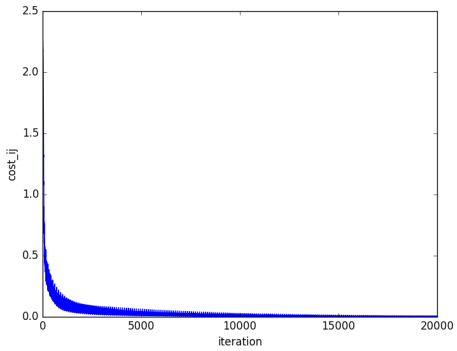
- Original LeNet: Best validation score of 0.91 % obtained at iteration 17400, with test performance 0.92%.
- Using Local Response Normalization: Best validation score of 0.99 % obtained at iteration 13400, with test performance 1.6 %.
- Using ReLu as activation function:Best validation score of 1.04 % obtained at iteration 11900, with test performance 2.4%.

- Using Relu and LRN: Best validation score of 1.18 % obtained at iteration 19500, with test performance 1.08 %.
- Gaussian weight initialization: Best validation score of 81.22% obtained at iteration 100, with test performance 80.90%.

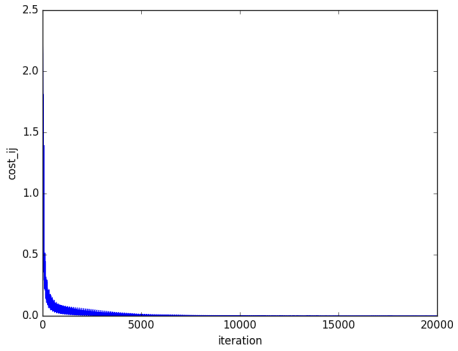
The best configuration for the network is the original one. With Gaussian initalization, the network does not find a local minimum in such a sort of time. Using LRN and Relu, test result is closer to the obtained with LeNet original, but not as good as the last one. Changing the activation function has not been a good change. Not taking into account original LeNet, the best test performance has been obtained with 1,08% using ReLu and LRN, but the best validation error is 0,99% obtained using just LRN. The values are close of the modifications, but the modification of Gaussian weight initialization.



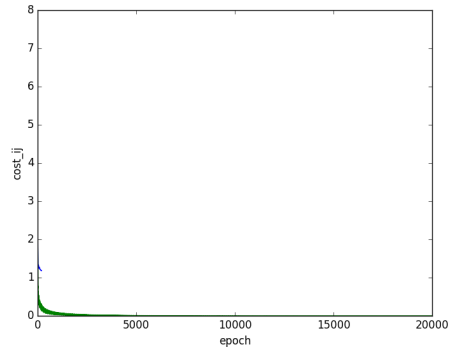
(a) Original LeNet



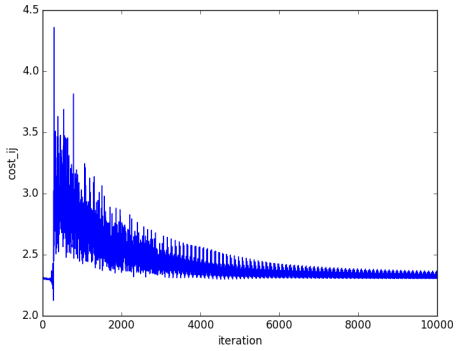
(b) Using LRN



(c) Using ReLu

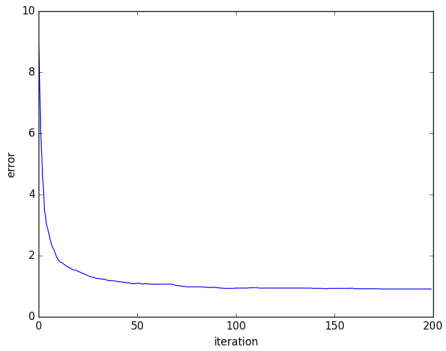


(d) Using ReLu and LRN

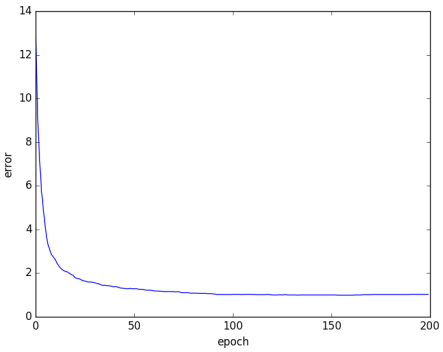


(e) Using Gaussian weights init.

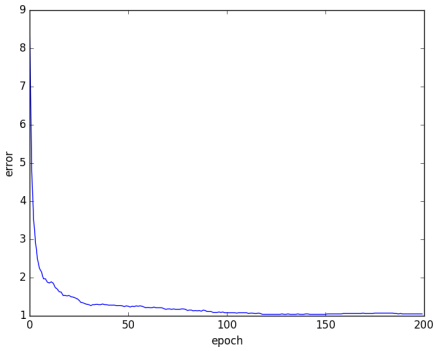
Figure 2.5: Cost function of Lenet and Lenet Modified.



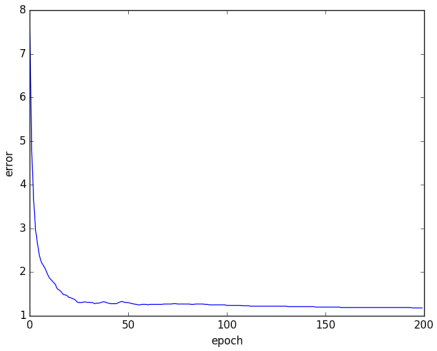
(a) Original LeNet



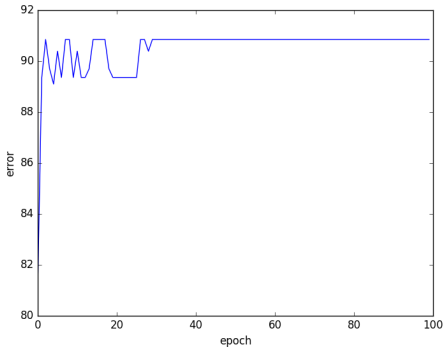
(b) Using LRN



(c) Using ReLu



(d) Using ReLu and LRN



(e) Using Gaussian weights init.

Figure 2.6: Valid error of Lenet and Lenet Modified.

2.2 Start working with Faces databases

The project has been developed from LeNet, changes would be explained below.

2.2.1 Using Labeled Faces in the Wild

With the purpose of reading and working with face images, a script has been developed where images are preccesed. All images are pseudo-randomized and grouped in training set (49% of the total database), testing test (30% of the total data) and validatng set (21% of the data).

To split the data, train_test_split function from sklearn.cross_validation has been used. This function pseudo-randomize the data, so you can repeat the randomized split data in the same way assigning the same seed to the function.

With If the batch size is 500, 13 batches are going to be used for training, 6 for validation and 6 for testing.

- $learning_rate = 0.1, n_epochs = 12, nkerns = [20, 50], batch_size = 500$
- T.tanh activaction function
- rng = numpy.random.RandomState(23455)
- kernel size = [5,5]
- images resized to 28x28
- 12 epoch
- The number of neurons at the input of the regression class is 10000, and the number of neurons at the output is 5748 (Number of people).

The results obtained are not good, because of the fact that parameters have been chosen randomly. and the network has not been optimized to this purpose.

Figure 2.7 represents the validation error % in each epoch, and it could be seen that in the last epoch, the error is the smallest one, and the test error in that point is 95.125000 %.

In order to know how the net works with different learning rates, it has been changed to 0.001 and the number of epoch has been raised to 50.

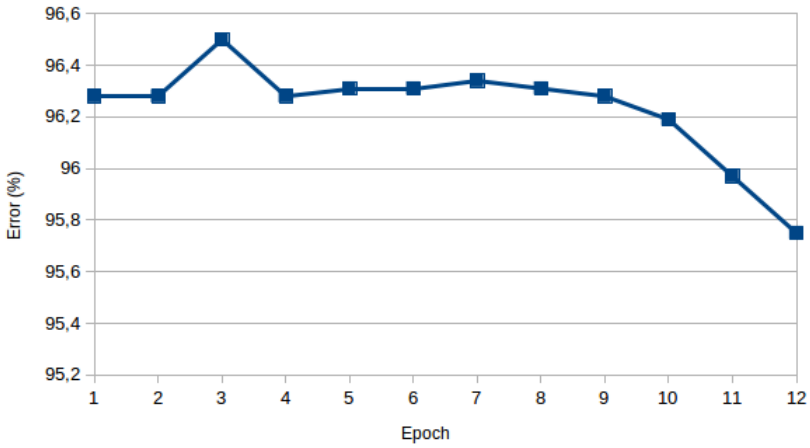


Figure 2.7: Error of Lenet using LFW.

The error in each epoch could be seen in figure 2.8 , where it is shown that the net does not learn because the learning rate is too small and it would need more epoch. The cost function of the training could be seen in figure 2.9, where it has been reducing during epochs, but it has been reduced a bit, from 8.7 to 8.1.

A 97,73% error of test performance has been gotten, which has been obtained in iteration 13.

The conclusion is the learning rate is too small to this configuration of the net and the data given.

Changing learning rate

Despite the fact that the configuration of the networks is not to this database, learning rate is going to be changed so it is possible to know how it affects.

If the learning rate is increased to 0.01: Best validation score of 96.266667 % obtained at iteration 481, with test performance 95.733333 %

If learning rate is increased to 0.1:Best validation score of 96.300000 % obtained at iteration 13, with test performance 95.733333 %

If learning rate = 0.5 Best validation score of 96.3 % obtained at iteration 143, with test performance 95.73%

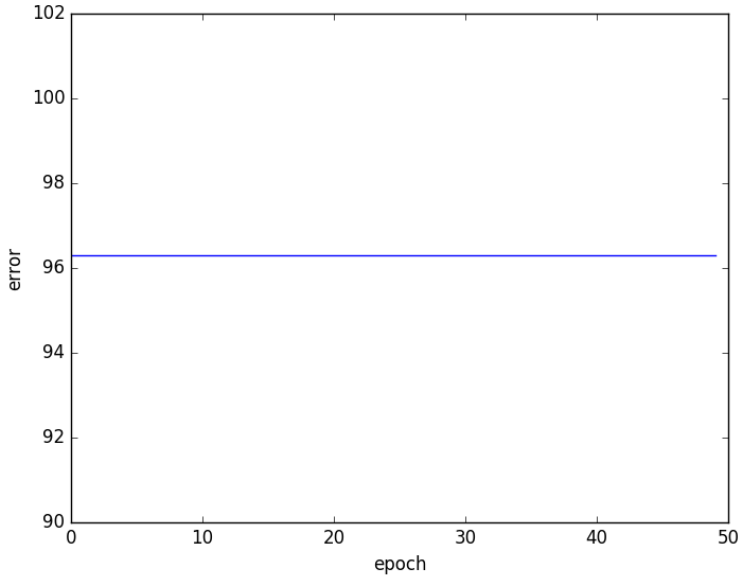


Figure 2.8: Error of Lenet using LFW with a learning rate of 0.001.

In conclusion, if the learning rate is too big, the network do not get a optimal minimum and if the learning rate is too small it takes too mch iteration to learn or getting to to optimal minimum.

Changing convolutional parameters

Convolutional layers are built with the Theano function *theano.tensor.nnet.conv2d*. The size of convolutional layers depends on the number of filters that users would like, the dimension of images, it is not possible to use the same convolutional layer for 3d images (rgb) than grey scale images (1 dimension), and also depends on the high and width user would like to give.

The output of a convolutional layer depends on the size of the filter and the size of input images. At the output, a new bunch of images are created from the input images and the characteristics of the layer.

Also, it is important to consider the batch size, because the layer is not fed by a individual image; the layer is fed by the bunch of images.

Lets have a bit of fun with convolutional layers, the number of filters and the size of them it is going to be changed. A learning rate of 0.1 is going to be used for 50 epoch.

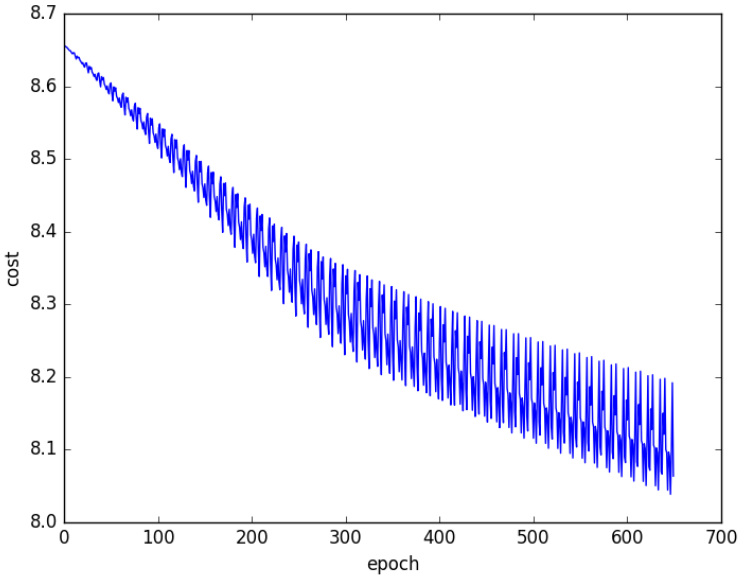


Figure 2.9: Cost of Lenet using LFW with a learning rate of 0.001.

In the first example, the number of filters of the first layer is going to be increased from 20 to 40, and the number of the second layer from 40 to 60.

In figure 2.13 could be seen the error which has been gotten in each epoch, where the best vest validation score of 94.9% has been obtained at iteration 559, with test performance 95%.

If the number of filters has been increased, the time that the code takes to run is increased significantly. Also, the results of the network changes, this is a parameter that should have in consideration.

Also, it is possible to modify the filter shape, in the previous examples, the size of both kernels were 5x5, in this example, the second one, the first convolutional layer would have a filter size of 3x3, the second one would keep its original size.

It is interesting seeing how the error of the network has been improved when the number of kernels has been changed and, in this example, has been improved too when the size of the filter has been changed.

The error in each epoch of this example could be seen in figure 2.14, where the best validation score obtained has been 94.567% error iteration 611, with test performance error 93.967%.

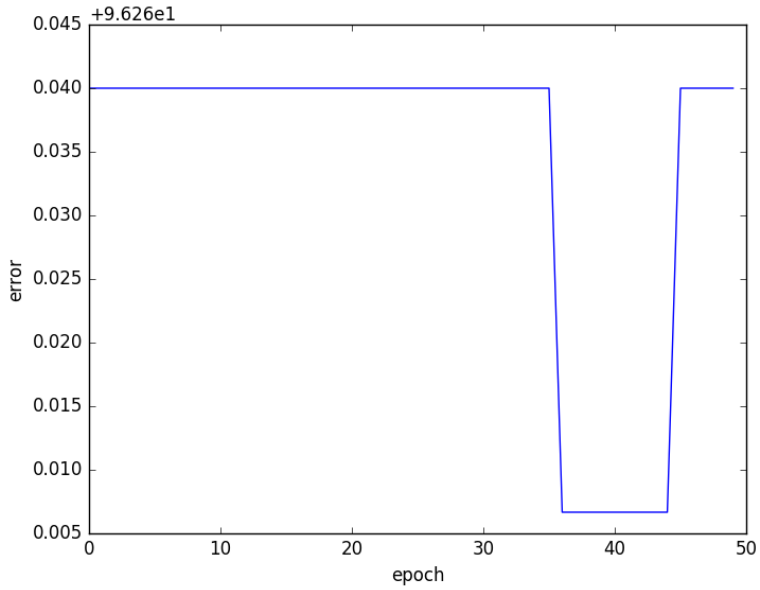


Figure 2.10: Error of Lenet using LFW changing learning rate to 0.01.

In order to see the difference between using a big size filter and one of a small size, in the next example, third example, 40 and 60 kernels are going to be used, and the size of each one is 3 and 10 for layer 0 and layer 1 respectively. In epoch number 40, the weighs of each layer has been saved and in figure X are represented. At irst sight, it is possible to see that with a big size.

Also, it is possible to see the output at the convolutional layer, in figure 2.16, the first 25 output images are shown for both layers.

The result of the third example is a best validation score of 95.73 % obtained at iteration 546, with test performance 95.23 %.

The cost function og those three experiments are represented in 2.17

2.2.2 Using ReLu as an activation function instead of tanh

Originally LeNet uses as activation function tanh(), but in this section, ReLu (Rectified linear units) activation function is going to be used. In equation 2.1 is shown how it is defined.

$$f(x) = max(0, x) \tag{2.1}$$

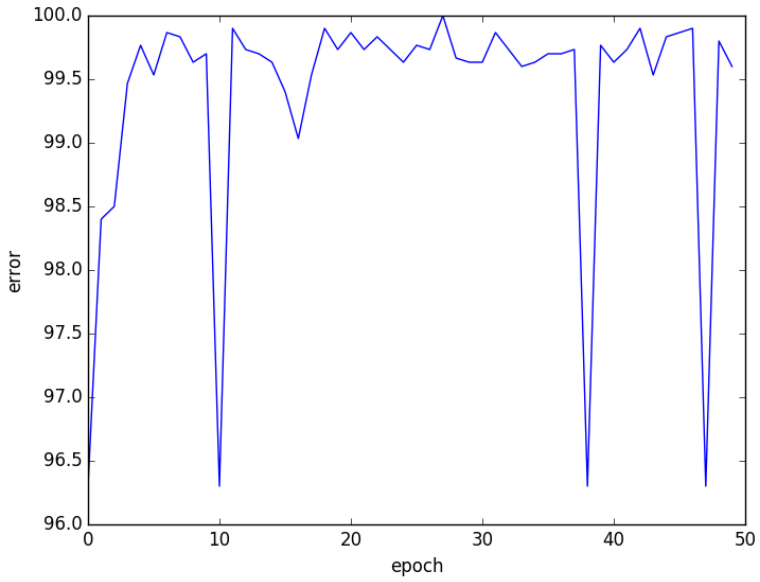


Figure 2.11: Error of Lenet using LFW changing learning rate to 0.1.

The error and the cost in each epoch could be seen in figure 2.18

2.2.3 using FRAV dataset

The experiments made with this database are just with RGB images (for the time being), so 939 images of people has been used, 489 train images has been used, 162 validation images and 279 test images.

Because images has not the same shape, they have been re-sized into 252x180, this new shape is proportional 0.7*height = weight because all images studied save that proportion. In addition, making images smaller also gives the security of not having memory problems, because the huge quantity of used images.

The network has been tested with this databases in the two ways of classify images, with two classes (genuine and attacks) and five classes (genuine and four classes, one per type of attack). Also, different learning rates have been used.

The first experiment made was based in the neural network LeNet, without changing parameters:

– 25 epoch

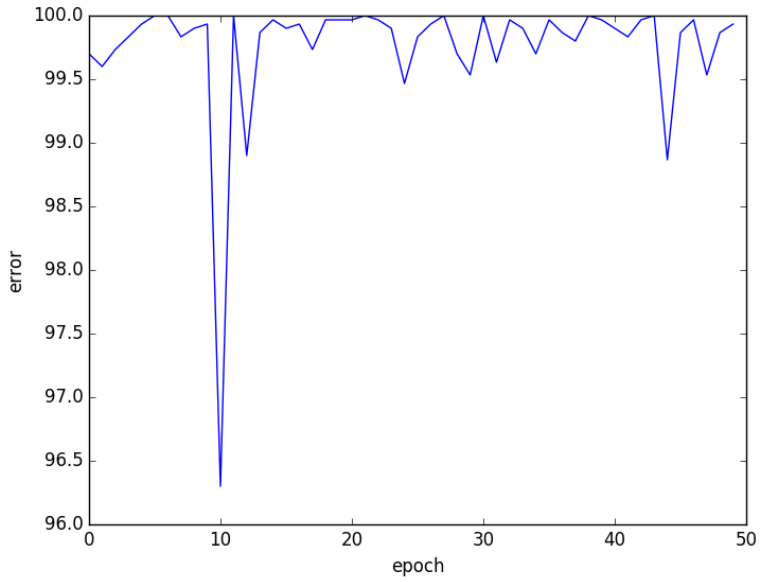


Figure 2.12: Error of Lenet using LFW changing learning rate to 0.5.

- nkerns=[20, 50]
- batch_size=50
- learning rate = 0.01
- Logistic regresion with 10 neurons.

The results that were obtained there were not as bad as the one with labeled faces in the wild (LFW).

Figure 2.19 shows the validation error of five classes, the best validation error has token place in epoch 7 with test performance 60.4%. It has token 68.78 minutes to run.

In figure 2.20, the validation error in different epoch could be visualized using two classes to classify. It could be seen that in each epoch the validation error is the same. Test error performance at the first epoch is 23.2 %. The total time of the running has been 64.0167 minutes.

After this, the CNN has been changed in order to build a CNN similar to the one described in *Learning Temporal Features Using LSTM-CNN Architecture for FaceAnti-spoofing.*

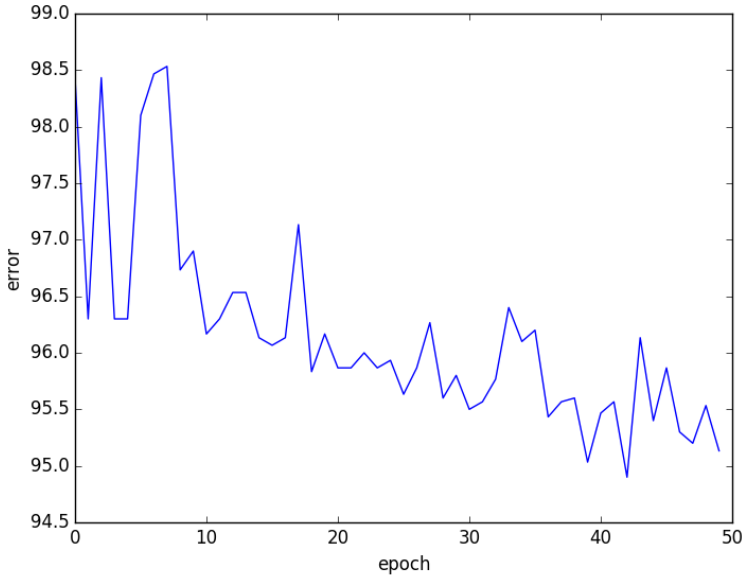


Figure 2.13: Error of Lenet using LFW changing number of kernels in example 1.

This CNN has two conv. nets., the first one with 48 filters and the second one with 96. The size of the filters is 5x5. The conv. nets. are followed by max pool layers of size 2x2.

The activation function is the rectified linear activation function (ReLu). I have used a normalized distribution of weights and bias, the same which was implemented in LeNet: weights are sampled randomly from a uniform distribution in the range [-1/fan-in, 1/fan-in], where fan-in is the number of inputs to a hidden unit. To use ReLu I have used the one implemented in *theano.tensor.nnet.relu*.

Because of the huge number of images, the batch_size has been changed into 20. And the learning rate is 0.001, like in the paper.

The number of neurons at the output of the hidden layer are 100. The classifier is the sigmoidal function. 25 epoch have been used to run the net at the training.

This database has been run with different seeds when the data is split in order to observe the behavior of the net with different sets of data.

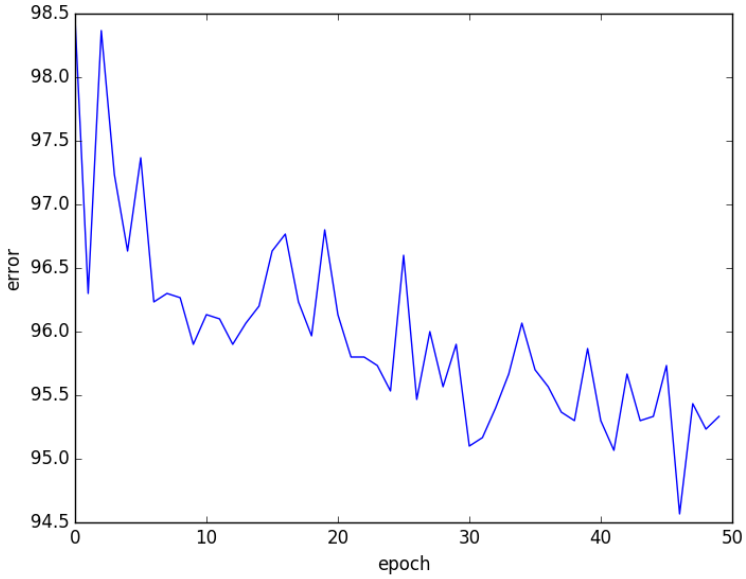


Figure 2.14: Error of Lenet using LFW changing number of kernels and the filter size in example 2.

2.3 Regenerating the databases

The databases have been generated in a different way from the used before. In this new way, the possibility of access to the missclasified samples is possible. SO the characteristic of misclassified images could be studied. This is not used now, but in the future it could be useful.

In order to build the new database, a script as been programmed manually because the test-train split sklearn function does not let know what image has been read and now it is possible to compare the right class of each image with the predicted one.

All this work has been changed because it is interesting to know the characteristics of people (if has beard, glasses or long hair) that are not well classified.

It has been necessary to shuffle the data twice in order to mix it properly and traning, testing and validating test would have data of the five classes.

2.3.1 using RGB and NIR FRAV database

To use both images at the same time, it is possible to do it in two different ways:

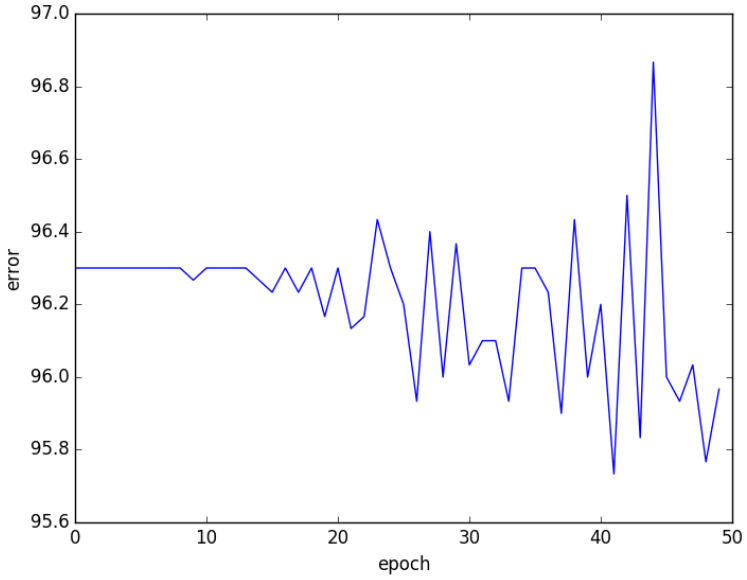


Figure 2.15: Error of Lenet using LFW changing number of kernels and the filter size in example 3.

- Characteristic level: adding the NIR image as another layer to RGB image, so the resultant image have hightxweightx4 dimensions (NIR images has one layer because it is a grey scale image and RGB images has tree layers, one per each primary color). The network is feed with the resultant images like other times.
- Classification level: training two network twice, first, with RGB images and then with NIR images and when the classification is produced combine them.

Adding images in characteristic level

In order to get this particular goal, each image is re-sized proportionally to original image to 52x104 dimensions. Each NIR image is appended to its correspondent RGB image.

In order to create a list of images where images are not putted in order, each image followed by another could be of a different class. Two shuffles has been necessary, the first one with a seed = 0.5 and the second one with a seed = 0.1, so the randomize order of images could be repeated.

For training, the 70% of the total data has been used, for testing the 20% and for validating the resting 10%. There are 157 images in each class and the distribution is represented in the table **??**.

	Class 0	Class 1	Class 2	Class 3	Class 4	Total of samples
Training set	94	119	116	145	76	550
Testing set	33	38	37	7	42	157
Validating set	30	0	4	5	39	78

Table 2.1: Distribution of samples FRAV (RGB + NIR) database

The code runs for 150 epoch with a learning rate of 0.001 and a batch size of 20 images.

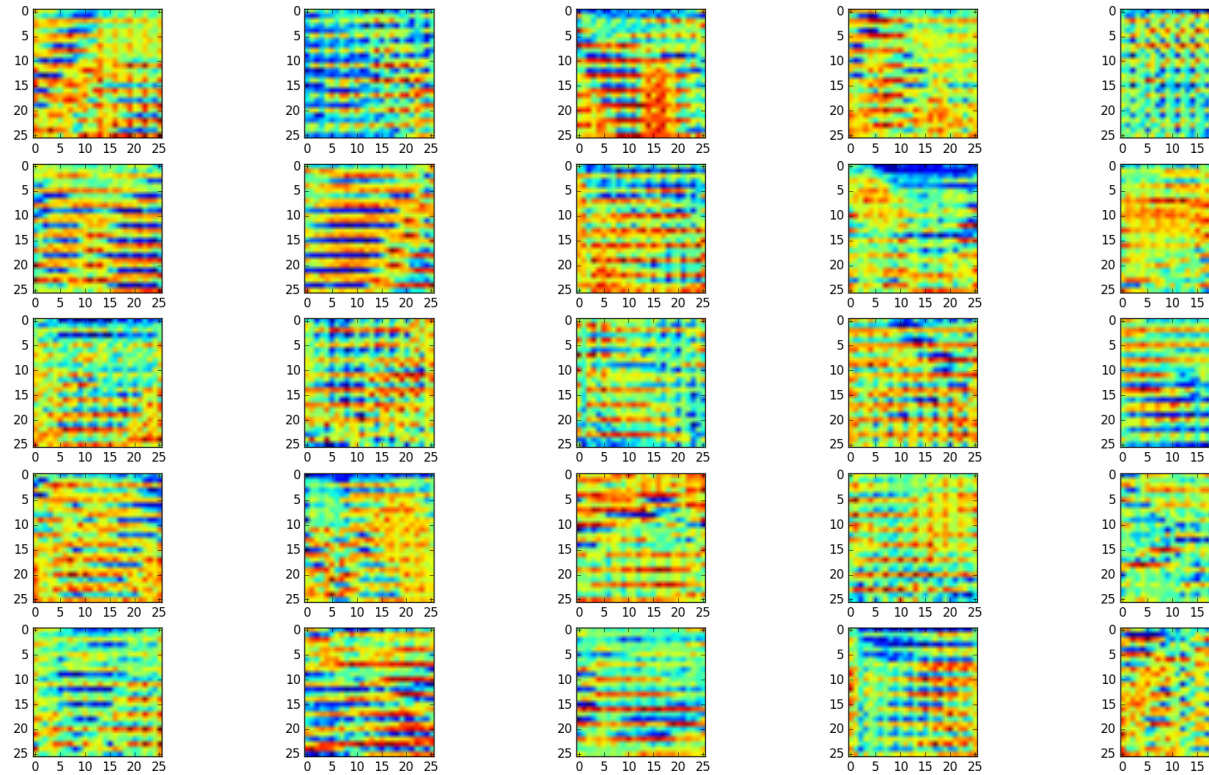
The disadvantage of using mini-batches is that there are some images remain and the quantity os less than a mini-batch, that images are not used, so from 157 testing images, just 140 has been used.

The test error that has been gotten is 30% at iteration 1863 where the best validation score was gotten (26,67%). Where:

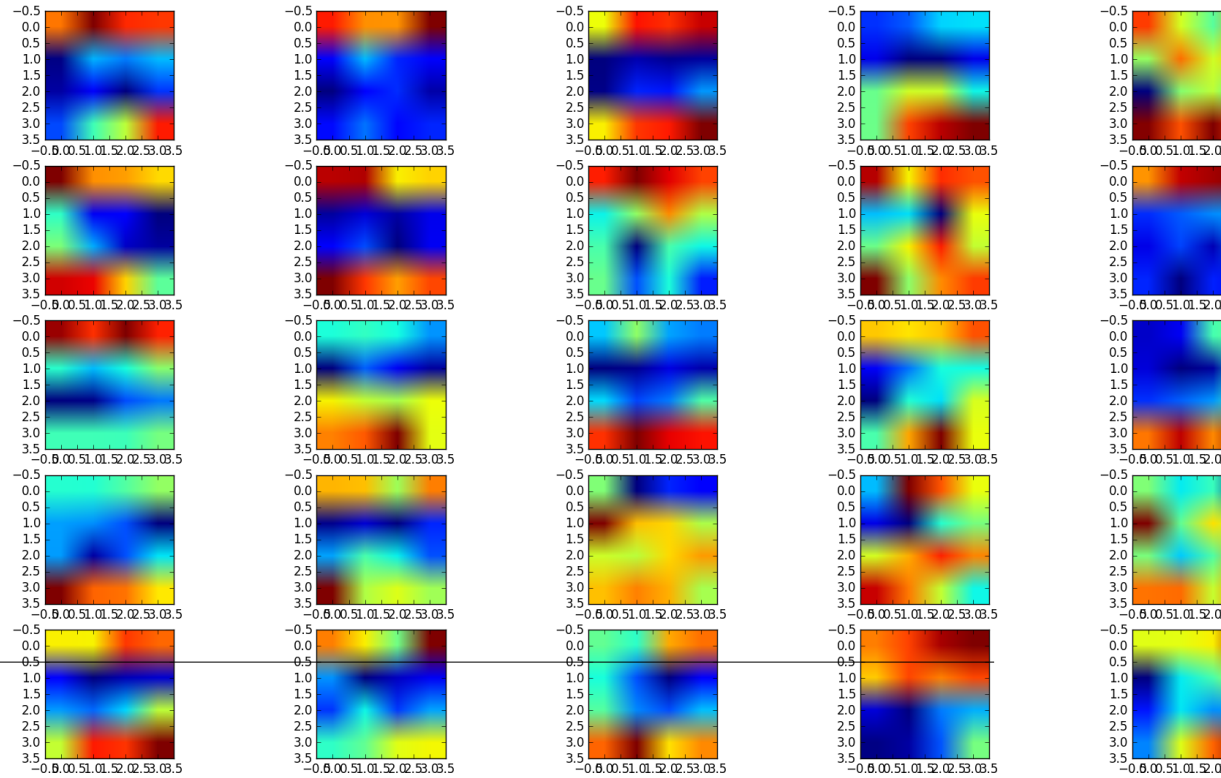
- Class 0 has been misclassified 14 times
- Class 1 has been misclassified 6 times
- Class 2 has been misclassified 7 times
- Class 3 has been misclassified 2 times
- Class 4 has been misclassified 2 times

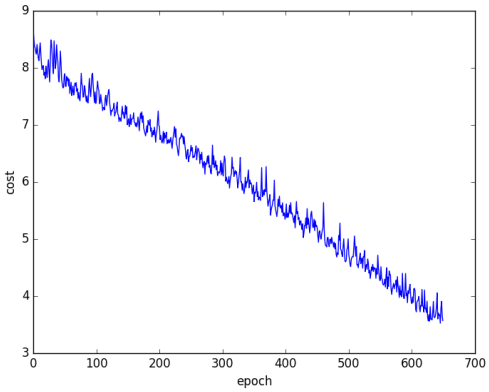
To get that results 3,04 hours was needed to run the code.

25 first images at the output of the first convolutional layer.

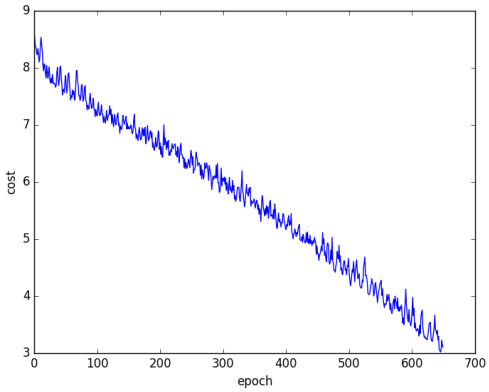


25 first images at the output of the second convolutional layer.

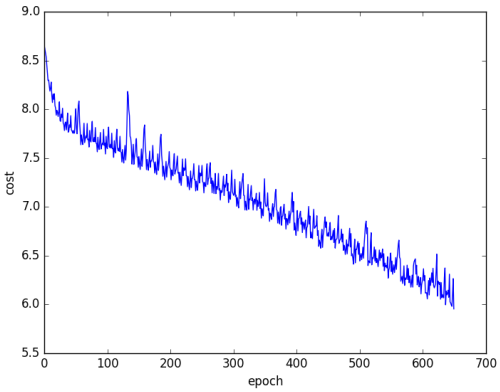




(a) Example 1

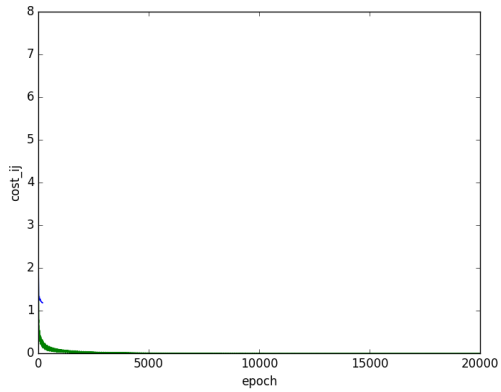


(b) Example 2

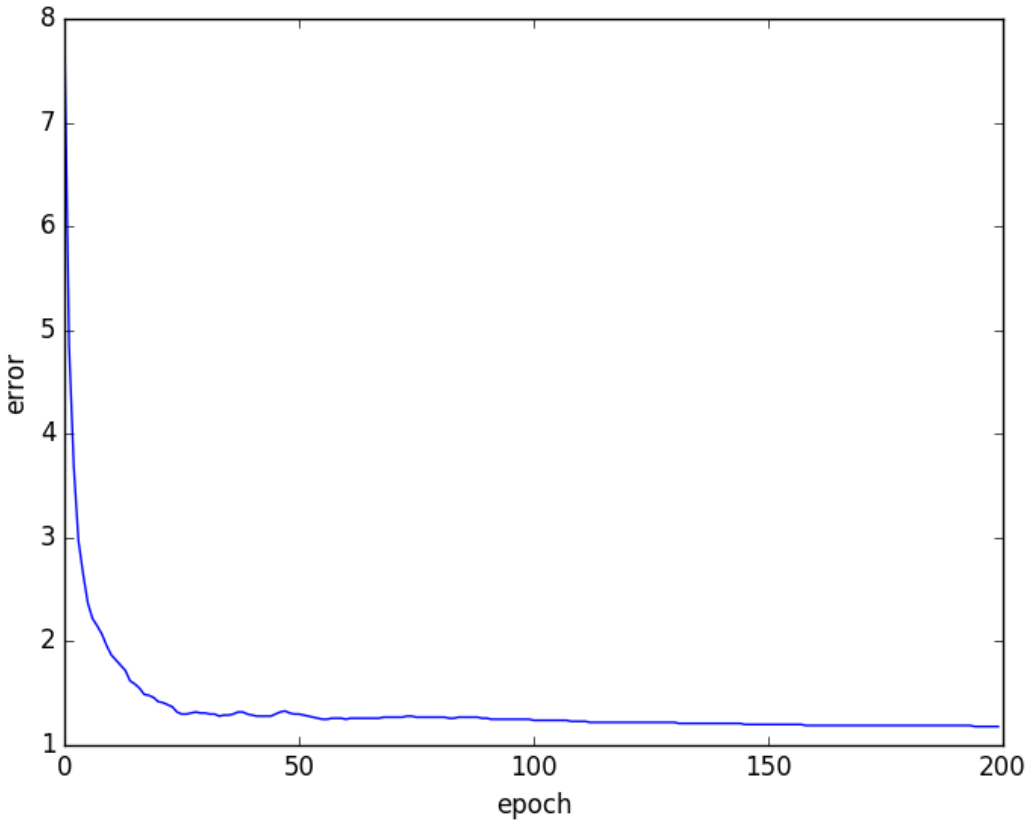


(c) Example 3

Figure 2.17: Cost function at training of the three examples chanching the convolutional parameters.



(a) Cost at training



(b) Error at validation

Figure 2.18: Error and cost using ReLu instead of tanh

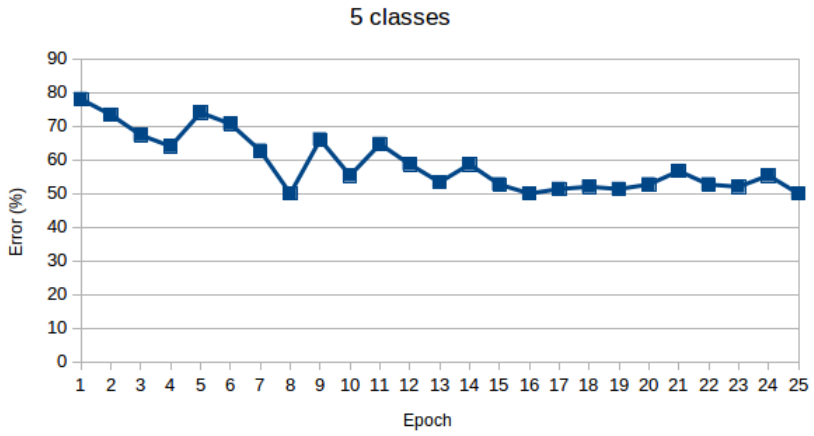


Figure 2.19: Error using FRAV database and five classes.

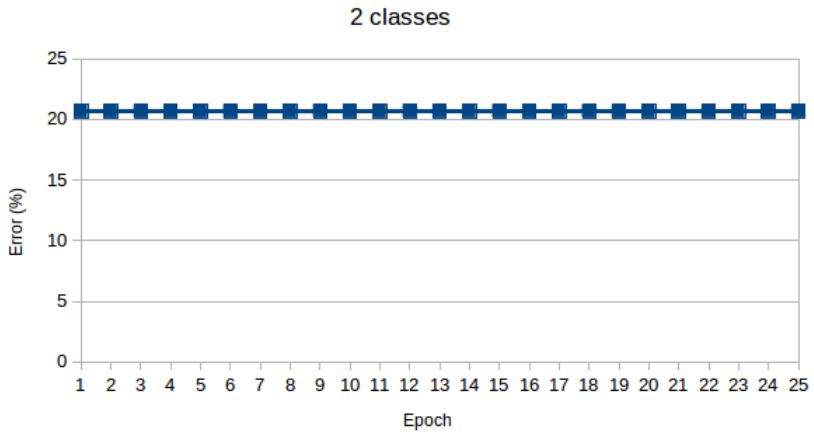


Figure 2.20: Error using FRAV database and two classes.

2.3.2 Architecture implemented in Casia videos

In this section, the architecture used in the paper of the Casia videos would be repeated partially. The paper is learn convolutional neural network for face anti-spoofing.

The architecture followed in the paper is the same used in Imagenet, it is formed by five convolutional layers, followed by three fully-connected layers. The two first conv layer and the last one are followed by a max-pool layer. The two first conv layers are followed by response normalization layers too. Authors use ReLu like activation function in each layer. The first two fully-connected layers are followed by two dropout layers, and the last layer output is followed by softmax.

Authors do not explain anything else about the architecture. In the paper of Imagenet is said that:

- The ReLU non-linearity is applied to the output of every convolutional and fully-connected layer.
- The first convolutional layer filters the 2242243 input image with 96 kernels of size 11113 with a stride of 4 pixels.
- The second convolutional layer takes as input the (response-normalized and pooled) output of the first convolutional layer and filters it with 256 kernels of size 5 5 48.
- The third, fourth, and fifth convolutional layers are connected to one another without any intervening pooling or normalization layers.
- The third convolutional layer has 384 kernels of size 3 3 256 connected to the (normalized, pooled) outputs of the second convolutional layer.
- The fourth convolutional layer has 384 kernels of size 3 3 192.
- The fifth convolutional layer has 256 kernels of size 3 3 192.
- The fully-connected layers have 4096 neurons each.
- The maxpool size filter si 2x3 because it reduces the error 0.4% - 0.6% compared to 2x2 filters.
- In Imagenet uses 224x224 images.

It is not explained how authors change the architecture of Imagenet.

The data used to this experiment is the Casia database, the one that uses in the paper, although authors, use Reply-attack too.

The data is composed by two folders, a training folder and a test data, in each folder there are some user folders, in each user folders there are two videos from the real user, tho videos of the user with a mask, two videos with the mask and with a hole in the eyes and two videos with a digital screen where a user is shown in the screen. 3 attacks are presented, so four classes are used (real users, users with mask, users with mask and eyes and a digital screen). It is not specified how many frames authors use, so it s supposed that they use all the video.

For each frame, the face is looked for in the image with viola jones algorithm implemented in opencv, and the image is cropped and saved, but in that image cropped there is no background, so different scales, 1.4, 1.8, 2.2, 2.6, are used to get background, because in learn convolutional neural network for face anti-spoofing and then images are re-sized to 128x128.

In order to carry out this experiment, a better computer has been needed because it was no possible to run with the same that the utilized in the previous experiments.

But with a better computer, it is not possible read more than 2 or 3 frames per video to run the net, because it has a huge architecture with a big quantity of filters per layer.

So the experiment with the videos has not been possible to be carried out.

In addition, is it not possible to carry out the architecture of imagenet with the size of Casia images because if it is started with 128x128 images, at the end, images sizes are images of ¡ 1px.

In order to know how strides work, a python file has been created called understanding strides, in which a pickle format file is loaded. This pickle format file is the output of a layer, and it is possible to see the size of images of the layer. So it is possible to know the size of images after striding and this number could be saved to be written in the conv layer.

In this computer, the theano function used to build the maxpool layer has changed bacuse of Theano version, the previous function used in Theano *theano.tensor.signal.downsample.max_pool_2d* has been replaced by *theano.tensor.signal.pool.pool_2d* using the mode *max*.

To sum up, In this first experiment, the architecture is formed by the convolutional and pooled out layers but without strides. The folder where this experiment has been developed has been in frav_casia_imagenet. The architecture is just based in conv, pool and hidden layer (no dropout, softmax or normalization layer). In figure 2.21 it is possible to see how the error is descending in each epoch and get stabilized with a 5% aprox. error.

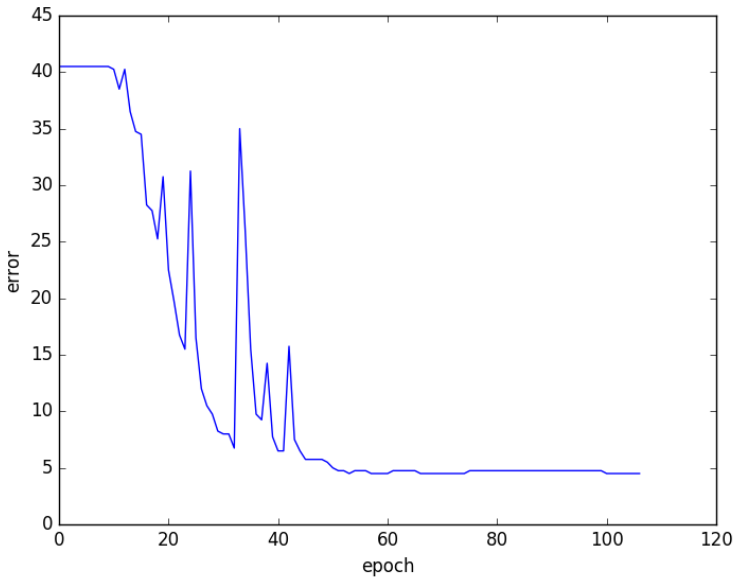


Figure 2.21: Error of Lenet in the first try to build imagenet.

The code had been running for 364,39 hours. And the true positive rate, true negative rate, false positive rate and false negative rate are available in table **??**.

TP	TN	FP	FN
156	623	10	11

Table 2.2: TP, TN, FP, FN rates in the first trying of imagenet.

2.3.3 As close as possible as Imagenet

Because of the lack of information about the convolutional neural network described in casia paper, it has been necessary to though the original code that authors use, Imagenet. Authors use the same architecture, although how it has been modified it is not explained.

It has been possible to implement Imagenet, the difference between the implementation and the original one is that the strides used in the first convolutional layer has not been used because the input of the images need to be bigger to have more than one neuron in the last layer.

The convolutional neural network has been tested with different databases: FRAV, CASIA and MFSD and different experiments have been carried out with each one.

Network configuration for each experiment

The configuration of each experiment of each network are described in the following lines. When it is said that Gaussian weight initialization has been used, the mean value is 0 and the std used is 0.01 in all the times ans bias has been initialized to 1, if not, bias has been initialized to 0. When Gaussian initialization has not been used, weights are sampled randomly from a uniform distribution in the range [-1/fan-in, 1/fan-in], where fan-in is the number of inputs to a hidden unit [copy-paste from deeplearning.net].

The goal of the next experiments is getting an optimal architecture changing the parameters. the difference between using a normal distribution o a Gaussian distribution of weights initialization has been tested and using SVM with linear kernel and RBF kernel has been tried.

FRAV database (Common parameters: n_epochs=400, nkerns=[96, 256, 386, 384, 256], batch_size=20):

- frav1: Gaussian weights initialization. SOFTMAX used as classifier. Learning rate = 0.001 Figure 2.22.
- frav_gaussian_initinizialization: Gaussian weight Initialization. Learning rate = 0.001. SOFMTAX used as classifier. Figure 2.23 .
- svm_gauss: Using SVM (with RBF kernel) as classifier. Gaussian weight Initialization. Learning rate = 0.01 2.24.
- svm_genera: Using as a classifier SVM with RBF kernel. Learning rate = 0.01 2.32.

– svm_linear: Classifying with SVM (linear) and Gaussian weight initialization. Learning rate = 0.01

CASIA (images) (nkerns=[96, 256, 386, 384, 256]):
First, four test has been carries out (in which the learning rate has been changed and the number of epochs. trying to get the best learning rate configuration. In four test the batch size used is 25 samples, the classifier used at testing is Softmax and the weight initialization used is normal distribution.:

– Test1: learning_rate=0.01, nepochs=400.

– Test2: learning_rate=0.001, n_epochs=400.

– Test3: learning_rate=0.0005, n_epochs=400.

– TEst4: learning_rate=0.001, n_epochs=1000,

It has not been possible getting a train loss that converge in a minimum in none of the four tests. A good train loss should decrease in each epoch until converge in a minimum. But in the tests, the - casia_gaussian_init: gausiana de pesos con mean 0 y std 0.01. SOFMTAX como clasificador. learning_rate=0.01, n_epochs=400, nkerns=[96, 256, 386, 384, 256], batch_size=20
- svm_gauss: Utilizando svm (rbf) como clasificador, con inizializacion normal. learning_rate=0.01, n_epochs=400, nkerns=[96, 256, 386, 384, 256], batch_size=20
- svm_general: utilizando SVM (rbf) con inicializacion de pesos gaussiana. learning_rate=0.01, n_epochs=400, nkerns=[96, 256, 386, 384, 256], batch_size=20
- svm_linear: SVM (linear) con inicializacion de pesos gaussiana. learning_rate=0.01, n_epochs=400, nkerns=[96, 256, 386, 384, 256], batch_size=20

MFSD (learning_rate=0.01, n_epochs=400, nkerns=[96, 256, 386, 384, 256], batch_size=20):
- svm_gauss: Utilizando svm (rbf) como clasificador, con inizializacion gaussianana.
- svm_genera:utilizando SVM (rbf) con inicializacion de pesos gaussiana
- svm_linear: SVM (linear) con inicializacion de pesos gaussiana

IMPORTANT: The valid graphs are made with SOFTMAX independly of the classifier used to test.

frav results

In this section, cost (at training) and error (at validating) is going to be visualized. First when FRAV database has been trained with Gaussian initialization and with a learning

rate = 0.001 2.22. Second with the same learning rate, but Gaussian initialization for weights 2.23. Third, decreasing the learning rate to 0.01 and with normal initialization 2.25 and the last one, whit the same learning rate but with Gaussian weights initialization 2.24.

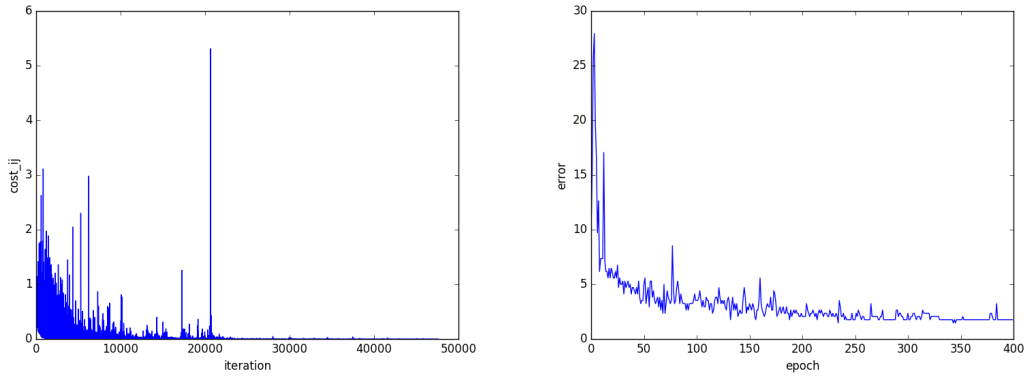


Figure 2.22: Cost at training and error at validating. Normal initialition. Learning rate = 0.001 (frav1).

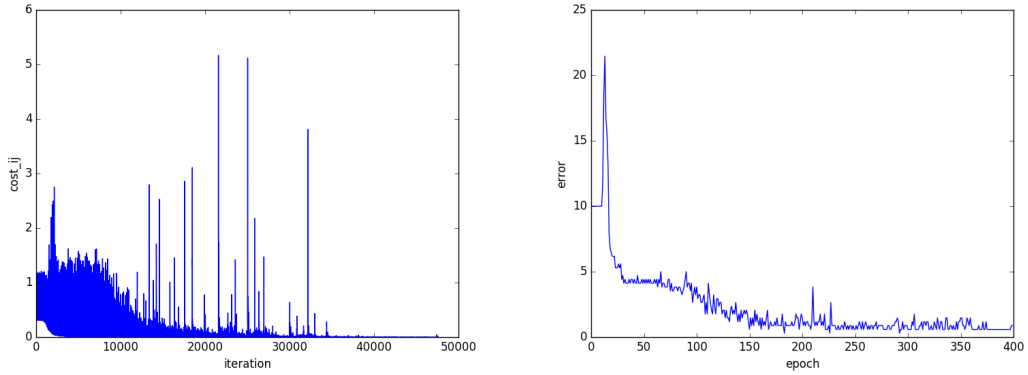


Figure 2.23: Cost at training and error at validating. Gassian initialization. Learning rate = 0.001 - frav_gaussian_init

First FRAV experiment (SOFTMAX as classifier and regular initialization) gives good results. The cost converges to 0 at training, the validation gets a 1.470588 % best error rate with test performance of 5 %. In testing, just 10 samples has been misclassified (7 samples of class 0 and 3 of class 1, from 679 test samples as total. The ROC and precision-recall curves could been visualized figure 2.26

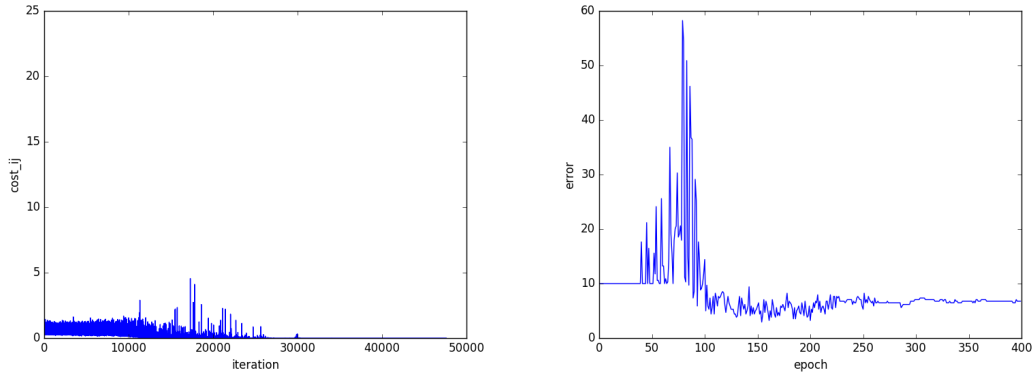


Figure 2.24: Cost at training and error at validating - Gaussian initialization. learning rate = 0.01 frav svm_gauss.

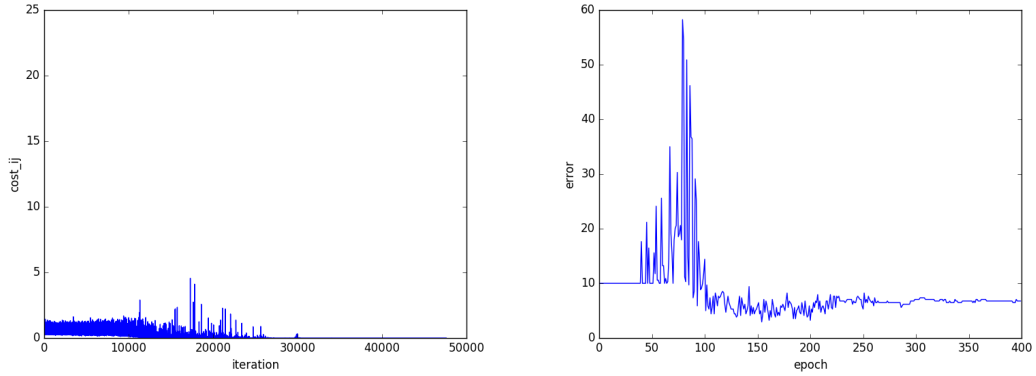


Figure 2.25: Cost at training and error at validating - Noraml initialization. learning rate = 0.01 frav svm_general.

It could be seen in the graphics that the Gaussian initialization does not matter when the learning rate is 0.01, but the cost changes when the learning rate is 0.001. In this case, the learning rate 0.001 should be the chosen one.

In the table **??** The positive and negative rates are visualized from the different classifier (softmax and SVM) the two different weights initialization and the two learning rates used. The results of the table are the same when the learning rate is 0.01 independently of the weight initialization or the kernel used to classify. The metrics rate are not too bad, with the learning rate the results are worse, the learning rate is too big.

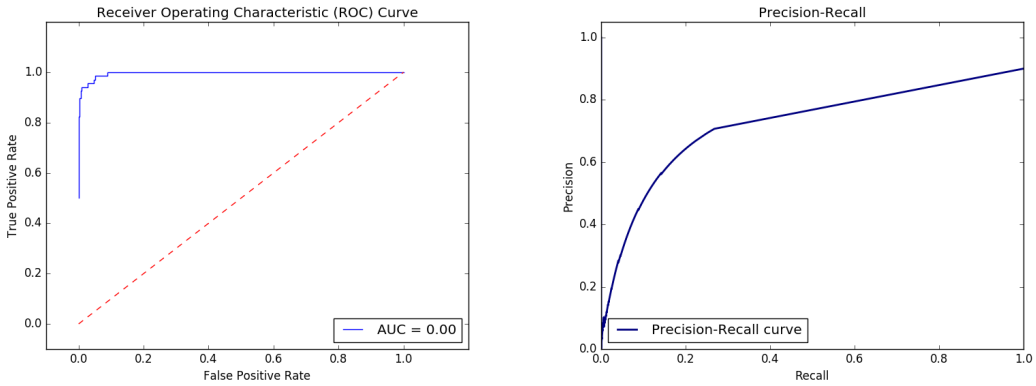


Figure 2.26: ROC and Precision- Recall courve - Noraml initialization. learning rate = 0.01 frav svm_general.

Classifier	Weight initialization	learning rate	TP	TN	FP	FN
Softmax	Normal	0.001	61	609	3	7
Softmax	Gaussian	0.001	66	605	7	2
SVM RBF(C=5)	Normal	0.01	63	593	19	5
SVM RBF(C=5)	Gaussian	0.01	63	593	19	5
SVM lineal(C=5)	Gaussian	0.01	63	593	19	5

casia results

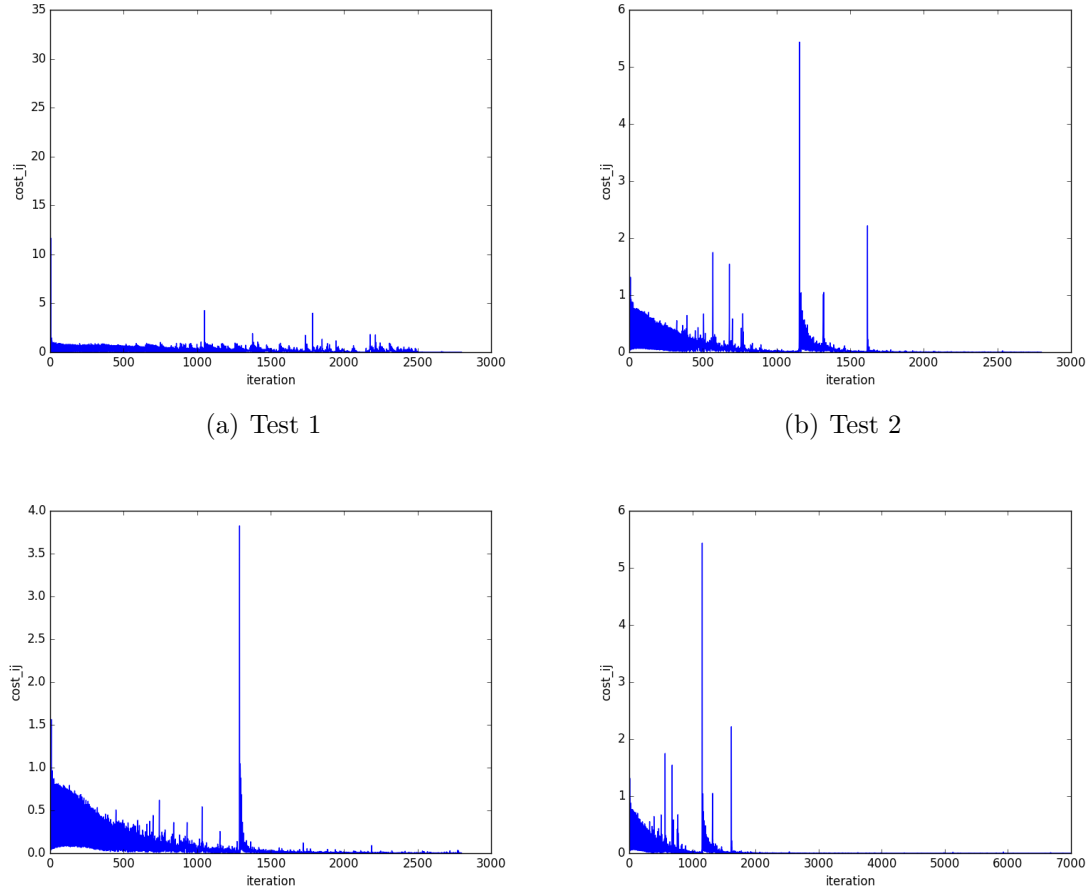
For Casia, different experiments has been carried out in order to train the network, using SOFTMAX as classifier and normal weight initialization:

- Test 1: Learning rate = 0.01 y 400 epoch.
- Test 2: Learning rate = 0.001 y 400 epoch.
- Test 3: Learning rate = 0.0005 y 400 epoch.
- Test 4: Learning rate = 0.001 y 1000 epoch.

In figure 2.27 the cost at training could be visualized. In general, should decrease varying its value but decreasing logarithmically. In the first experiment (test 1), the value varies but in a small range, and in general it is constant, in the other three experiments, the cost decreases and this is what must happen, but in test 2 the loss converges two times, after converge the first time, then it increase again ans converges again.

In figure 2.28 it is possible to see that the error changes but not in a logarithmically way. It increase and decreases its value in each epoch but it all experiments it gets in a

42% or 40% error. The desired curve should be as the loss one, but the error should not decrease as much as the training loss does.



(a) Test 1

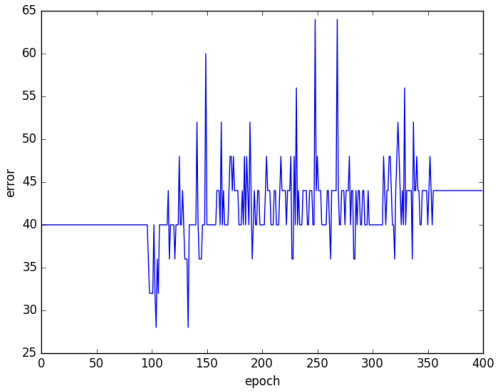
(b) Test 2

(c) Test 3

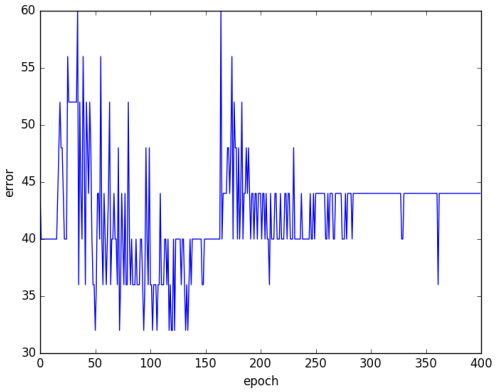
(d) Test 4

Figure 2.27: Training loss of four test Casia database

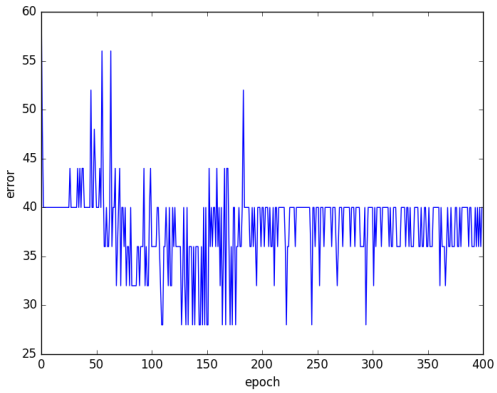
As the same way that in the first experiment that FRAV database converges is that would be expected from Casia, but it has not gotten.



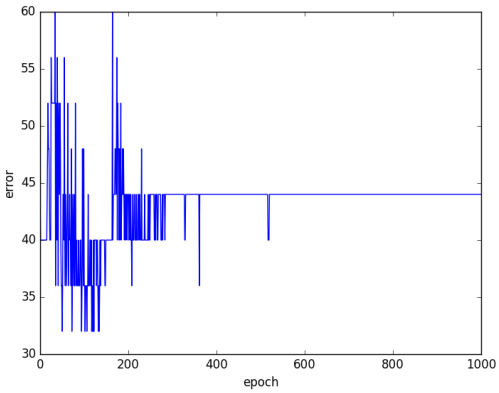
(a) Test 1



(b) Test 2



(c) Test 3



(d) Test 4

Figure 2.28: Validation error of four test Casia database

At the same way as FRAV, the classifier and the weight initialization has been changed in order to know how the networks behavior with these changes. The learning rate used for this experiment is 0.01. First, the cost (at training) and the error (at validating) are going to be visualized when the weight initialization is normal 2.30. Second when the weight initialization used is Gaussian 2.29.

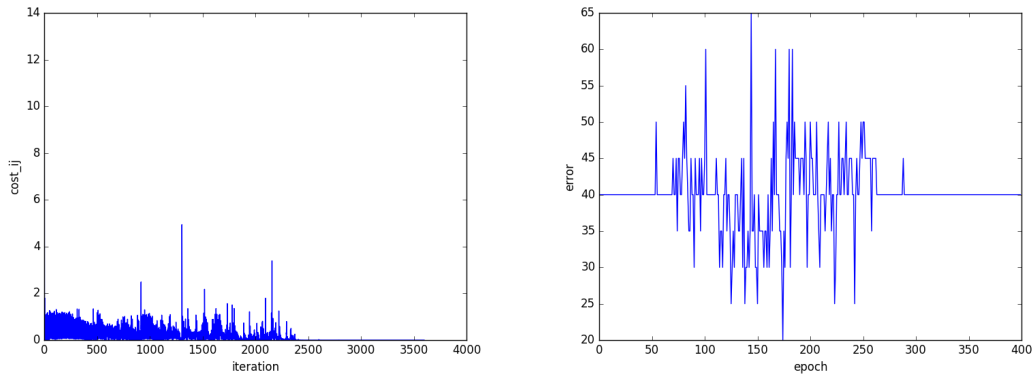


Figure 2.29: Cost at training and error at validating -casia svm_gauss.

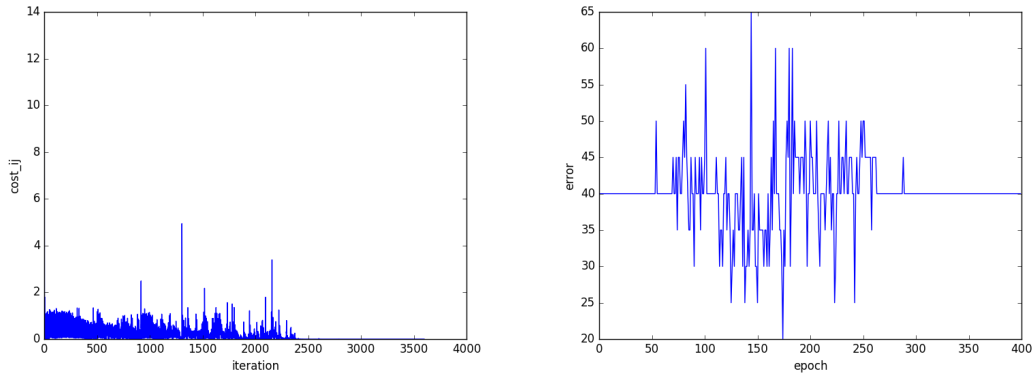


Figure 2.30: Cost at training and error at validating -casia svm_general.

As the same way than in FRAV database, the cost and at training the error at validating has not changed. The positive and negative rates are the following ones in the three experiments: TP = 8; TN = 21; FP = 3; FN = 8.

MFSD results

As the same way in FRAV and CASIA, but now with MFSD, it is going to be tested the network with a learning rate of 0.01, Gaussian initialization 2.31 and normal distribution initialization 2.32 and classifying the test with SVM (RBF kernel and linear) and softmax.

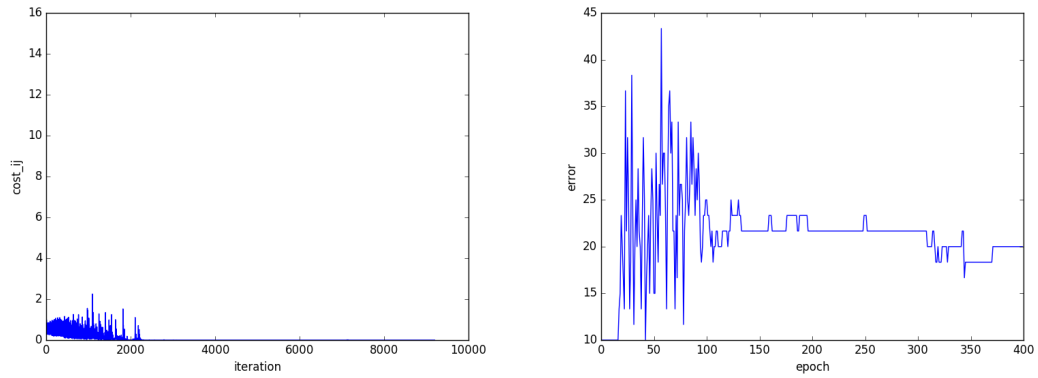


Figure 2.31: Cost at training and error at validating - mfsd svm_gauss.

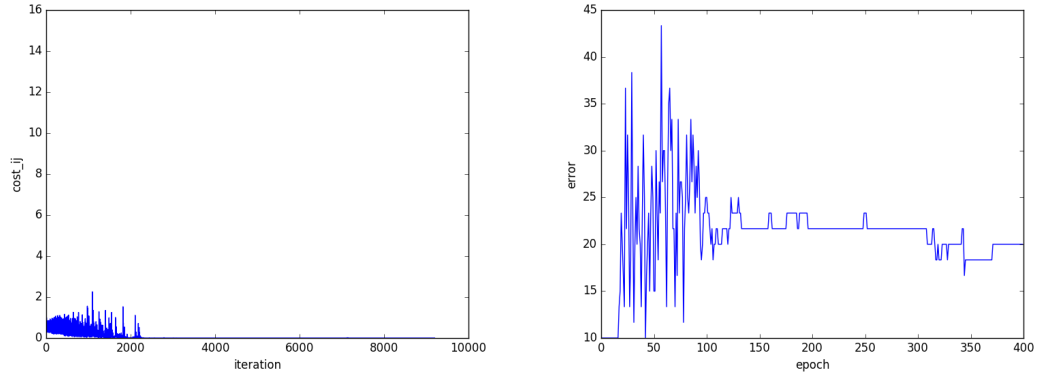


Figure 2.32: Cost at training and error at validating - mfsd svm_general.

The same problem occurs. The train and valid graphs are the same in both cases, independently of the initialization.
Also, the result at testing is the same in three cases: TP = 12, TN = 3, FP = 105, FN = 0

2.3.4 New database

From images, a new database has been developed. Images are read and re-sized directly to 128x128. The reason of using 128x128 is the size that authors in the Casia paper use. The databases explained in **??**:

-	FRAV	FRAV (rgb+nir)	CASIA images	CASIA video	MFSD
n train samples class 0	157	133	26	255	30
n train samples class 1	459	417	111	81	68
n test samples class 0	19	16	16	540	3
n test samples class 1	167	141	23	180	25
n valid samples class 0	10	8	7	105	2
n valid samples class 1	83	70	13	39	12

I can not obtain results with Casia RGB + NIR appended in classifier because it said that there is not space enough.

experiments

With that databases. Some experiments has been carried out:

- general experiment, with Gaussian weight initialization, classification with SVM RBF and SOFTMAX.
- general experiment but with a small database in order to make over-fitting in the network and check it. It has been used 20 train, test and validation images in all databases but MFSD that has been used 14 images for each subset.
- The same experiment that above but the test has been realized with the same subset that in training, this is to check that the network has over-fit or should have over-fitted.
- The same as above but decreasing the learning rate from 0.01 to 0.001.

From images, could be concluded that in general, decreasing the learning rate for this experiment has not been a good idea.

In table **??** could be seen the positive and negatives rates when has been used SVM RBF to classify. The result obtained with FRAV (rgb + NIR) is really good because just 3 samples have been missclaffied from 140 images.

I do not know why the test is the same for minidataset and minidatset tested with itself.

-	Optima CSVM	TP	TN	FP	FN
FRAV	0.05	136	24	11	9
FRAV (rgb+nir)	0.1	113	24	1	2
CASIA images	5	9	2	0	9
CASIA videos	0.1	478	75	105	62
MFSD	10	19	1	8	0

Looking the graphs where a minidataset has been used (20 images or 14), if the cost (training) is visualized, could be expected that the train is learning the images because the cost decreases to 0 (almost zero) so that means that if it is tested with itself the error should be 0, but that does not happen.

The conclusion is the needed of a balanced database, at least to do this experiment. In which the number of class 0 samples are the same that the number of class 1 samples, because the network would not learn in the same way if in some cases the number of samples of attack class is four times than the number of samples of class 1, just predicting

0 would have 25% accuracy, and having less than 5 samples in validation or testing is not a good generalizer (2 samples in class 0 MFSD database).

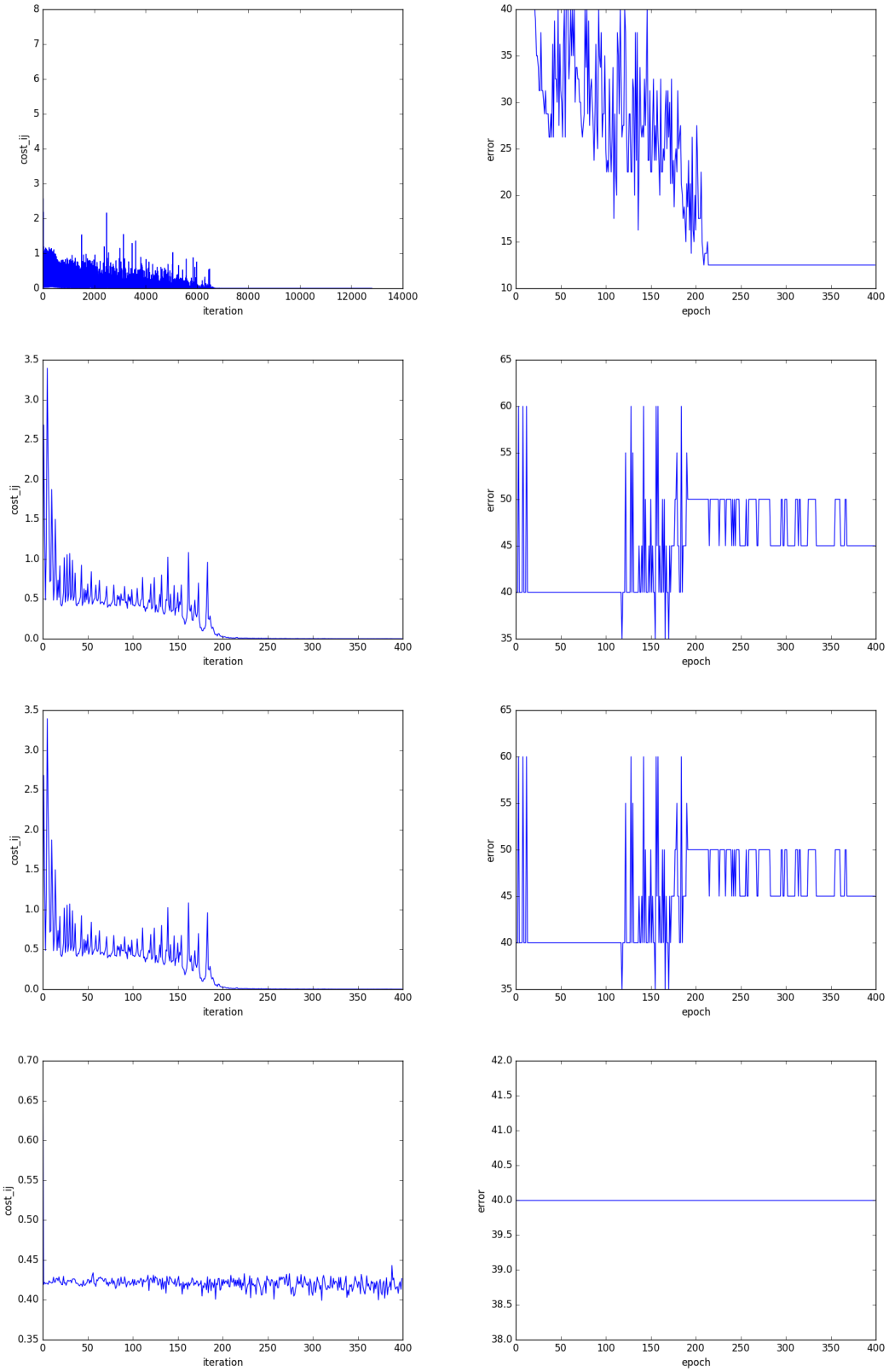


Figure 2.33: cost and error of the tree experiments with frav.

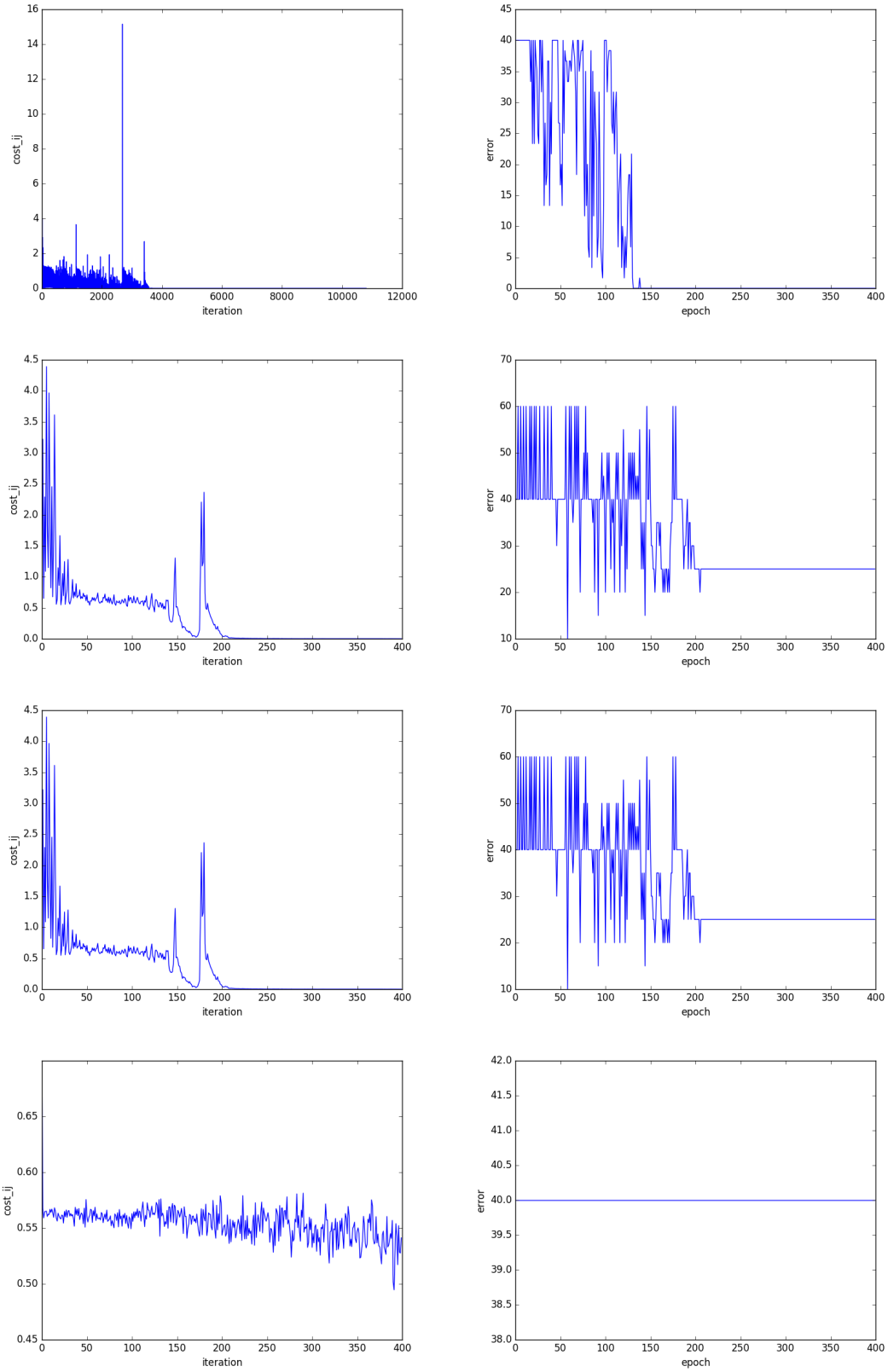


Figure 2.34: cost and error of the tree experiments with FRAV (rgb + nir) image level images.

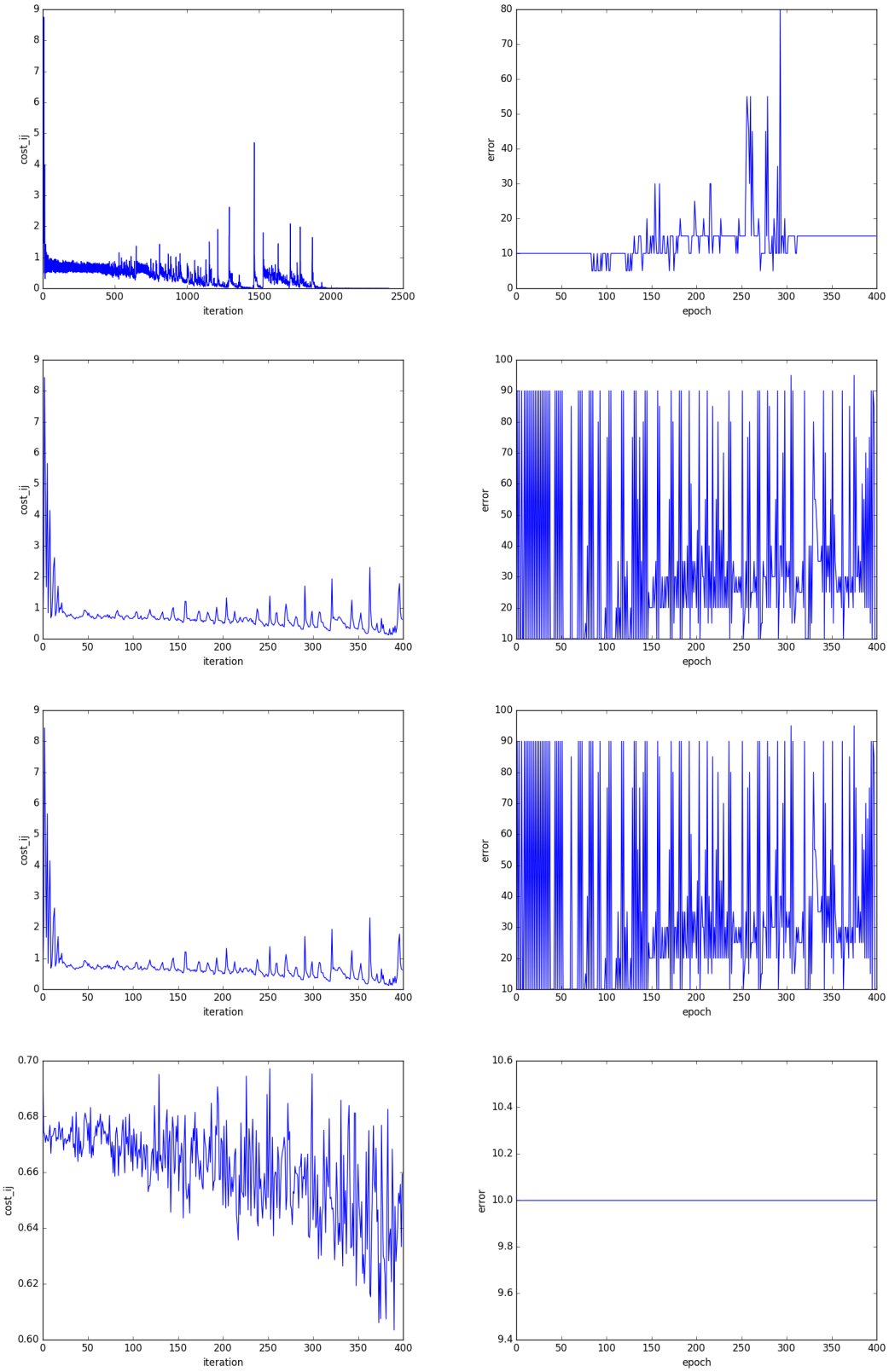


Figure 2.35: cost and error of the tree experiments with CASIA images.

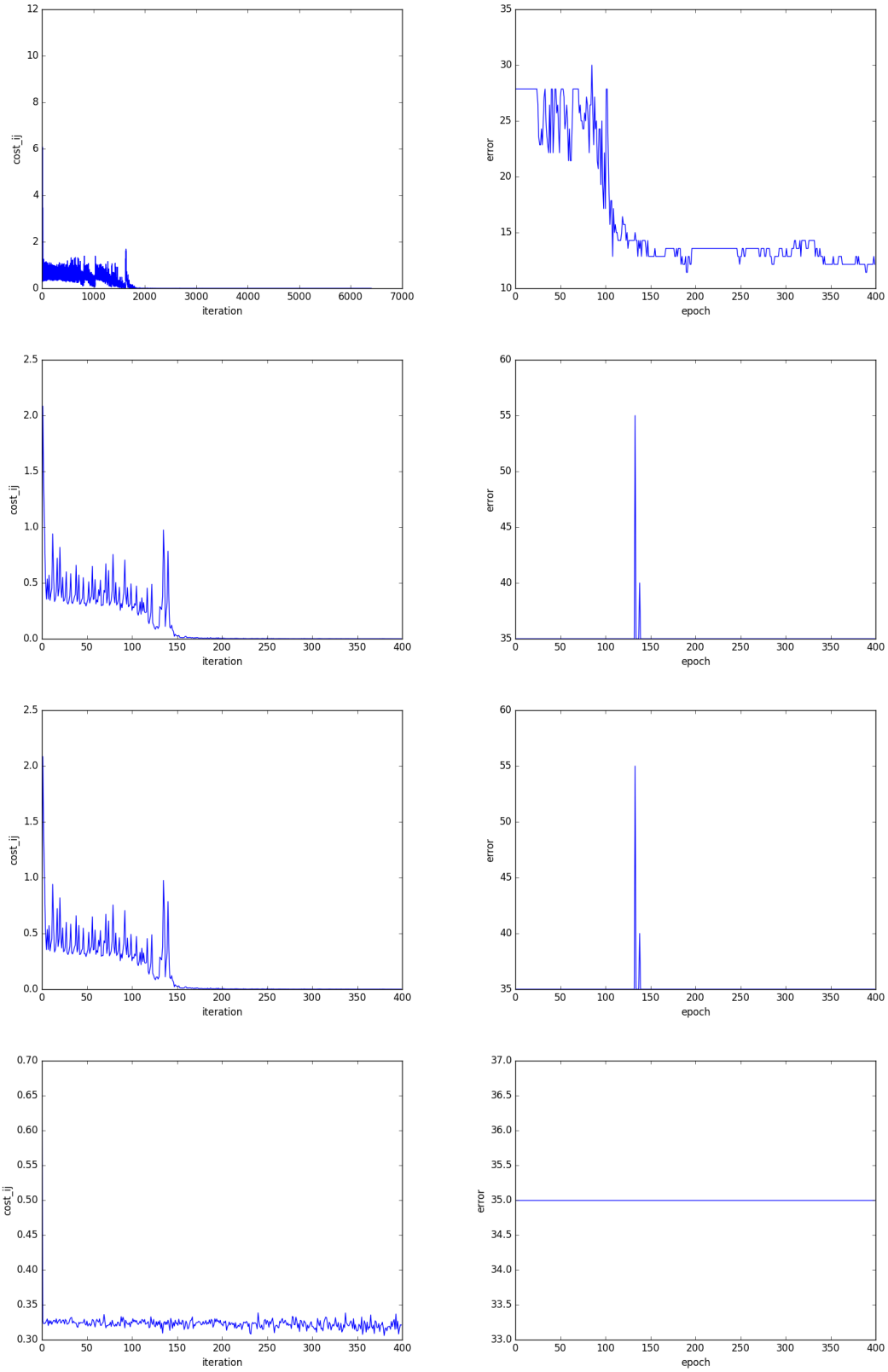


Figure 2.36: cost and error of the tree experiments with CASIA videos.

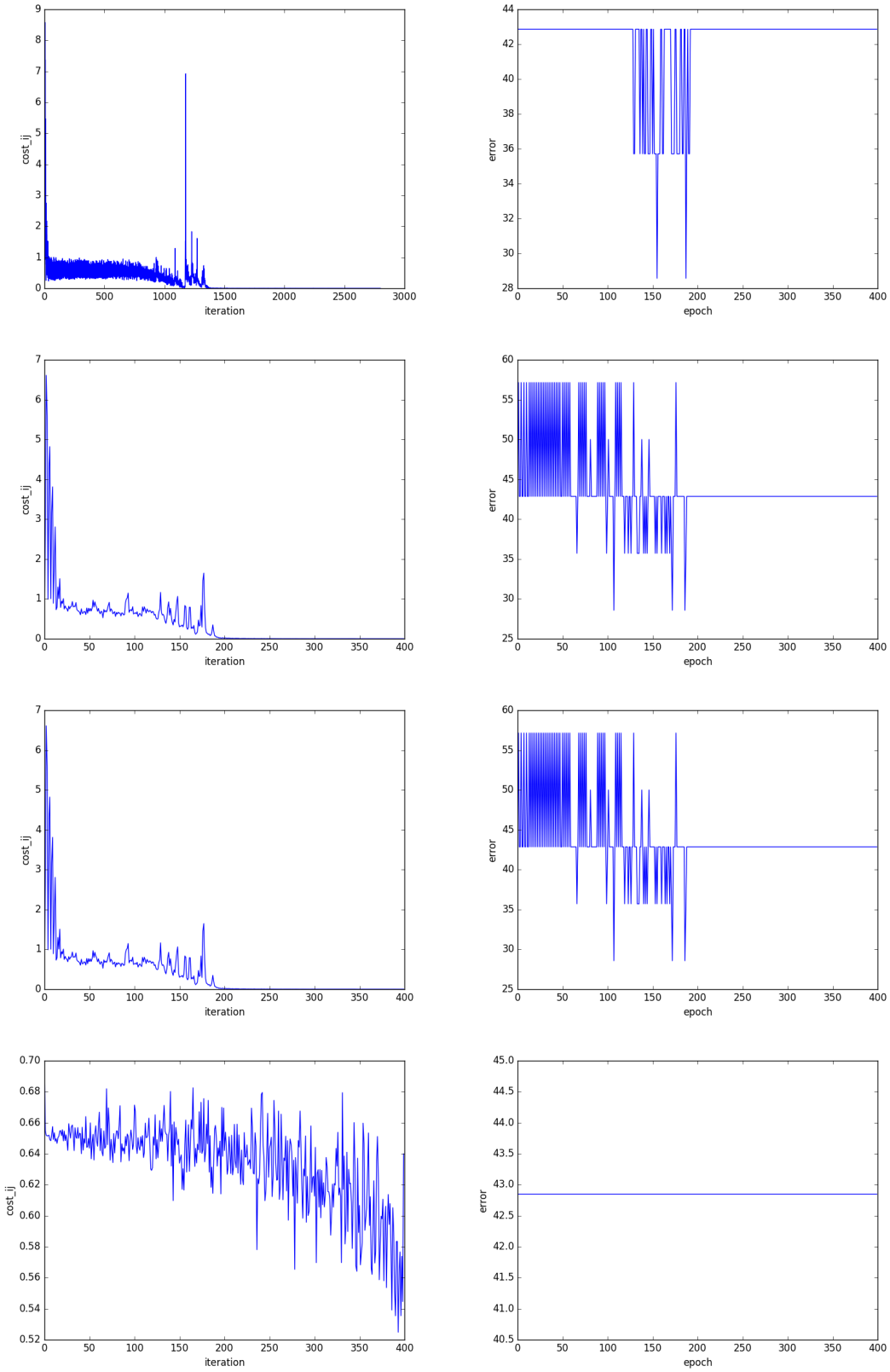


Figure 2.37: cost and error of the tree experiments with MFSD images.

Bibliography

[1] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016.

[2] M. Sokolova, N. Japkowicz, and S. Szpakowicz, "Beyond accuracy, f-score and roc: A family of discriminant measures for performance evaluation," in *Proceedings of the 19th Australian Joint Conference on Artificial Intelligence: Advances in Artificial Intelligence*, pp. 1015–1021, 2006.

[3] "Sc37 iso/iec jtc1," 2014.

[4] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, pp. 2278–2324, 1998.

List of Figures

List of Tables