Miniproject 1: Classification and Regression

Pirashanth RATNAMOGAN

pirashanth.ratnamogan@ens-paris-saclay.fr

2. Simple Classification

In this first part of the miniproject, we have to create a linear classifier based on Neural Networks in order to classify images into three possible categories (i.e. rectangle, disk, triangle).
As described in class, the goal will be to create a classifier that will learn the parameters W and b to define the probability to belong to each classes as :

$$p(y = i|x) = \frac{exp(W^i x + b^i)}{\sum_{j=1}^{3} exp(W^j x + b^j)} \quad \forall i \in \{1, 2, 3\}$$

Considering that the label 1 corresponds to rectangles, 2 to disks and 3 to triangles.
We will use the test set `generate_test_set_classification()` in order to see how performs our algorithm on generalization.

We easily implement this simple linear classifier with categorical cross entropy loss. Here is the table reporting the final loss and accuracy for both train and test set for various choice of optimizers (with default parameters). We will use a batch size of 32 and generate 300 examples without free locations option in order to create our training set.

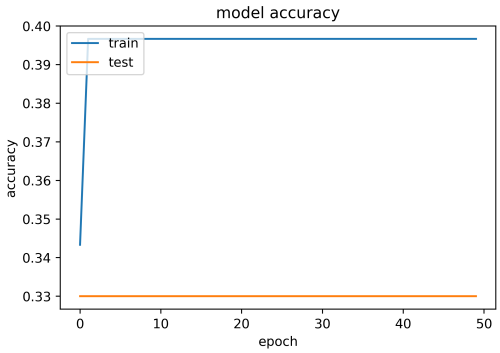
Optimizer	Epoch	Train Loss	Test Loss	Train Accuracy	Test Accuracy
SGD	50	10.1007	10.6379	0.3733	0.3400
Adam	50	0.0235	3.6103	1	0.3767
Adagrad	50	10.7991	10.7991	0.3300	0.3300
RMSprop	50	5.3191	7.5282	0.6700	0.3500

Table 1: Evaluation of the algorithm using various optimizers with the default parameters

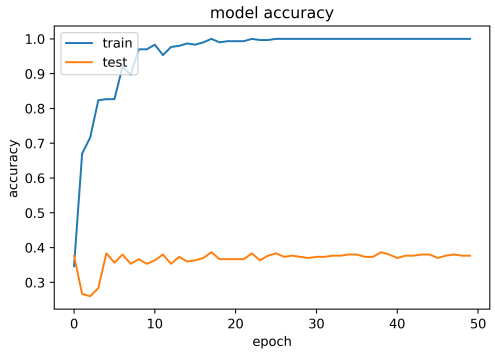
As we can see Adam optimizer that is a momentum based optimizer is known to perform quite well on simple task and it outperforms other methods in our test with the default parameters. As seen in class, Adam requires less tuning than the other methods. As we can see we have to lower the learning rate in order to make the other methods converge. Reducing the SGD learning rate from 0.01 to 0.001 allows to obtain the convergence of the algorithm but in a larger number of epochs than the one needed for Adam.

We can have a look at how behave the loss through the training epochs using the default parameters :

We can see that the model is not performing at all using SGD optimizer with the default parameters. Using Adam optimizer we can see that the model fits well the training data, however it doesn't perform well

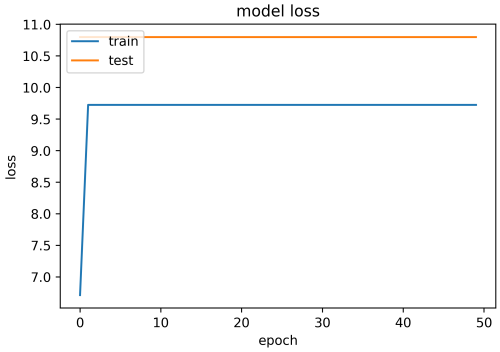


(a) Evolution of the accuracy using sgd optimizer

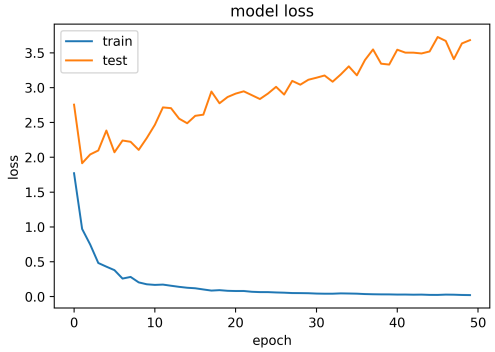


(b) Evolution of the accuracy using adam optimizer

Figure 1: Comparison of the evolution of the accuracy using adam et sgd optimizers



(a) Evolution of the loss using sgd optimizer



(b) Evolution of the loss using adam optimizer

Figure 2: Comparison of the evolution of the loss using adam et sgd optimizers

on test data. It can be explained by the fact that the model is quite simple and that the training se doesn't contains free location object while test data contains free location object. We will see in the next section how to improve our outcome. There is a variation between the data present in the train set and the data present in the test set that explains this gap.

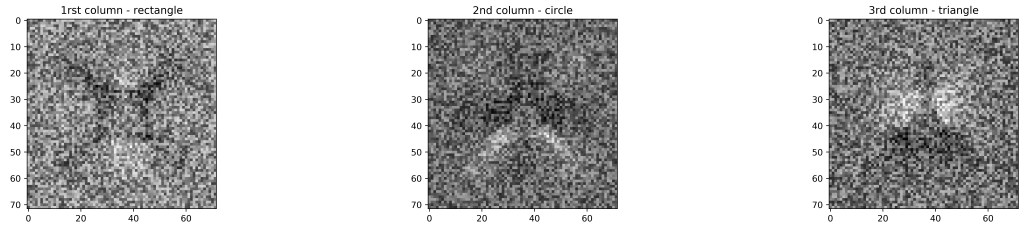
3. Visualization of the Solution

We can easily display one kind of filters our simple linear classifier has learned using really simple line of code :

The figure 3a is the filter that has to detect rectangles, 3b has to detect disks and 3c has to detect triangles. We can infer some sort of triangle detector in the filter represented in 3c. It's difficult to say anything else from that figures that doesn't seem perfectly adapted to the task that we want to solve.

4. A More Difficult Classification Problem

I will create a deep neural network to perform better on the previous task. In order to make the task more intuitive, it has been decided to reshape the images as matrices instead of using flat vectors.



(a) Display of the first row **(b)** Display of the second row **(c)** Display of the third row

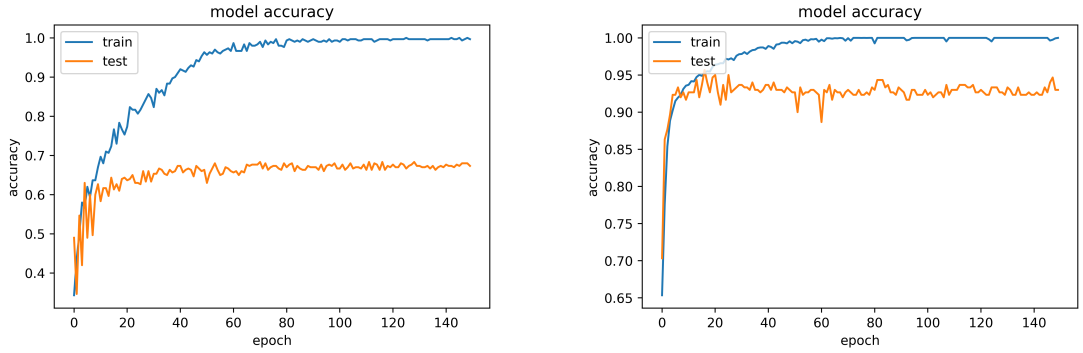
Figure 3: Display of each learned filters in the linear neural network (after 10 epochs)

A Simple Deep Neural Network

First I will try to solve the previous problem using a simple deep neural network with one convolutional hidden layer (with 16 5x5 filters) followed by a max pooling layer and a fully connected layer. Here are the outcomes obtained by using various number of training examples (using Adam optimizer) but keeping the test set given by `generate_test_set_classification()`.

Nb train data	Epoch	Train Loss	Test Loss	Train Accuracy	Test Accuracy
300	150	0.0258	1.3350	0.9967	0.6733
1000	150	0.0050	0.8589	1	0.8100
2000	150	0.0014	0.5541	1	0.8600
5000	150	9.7766e-04	0.4108	1	0.9200
10000	150	8.1917e-04	0.3133	1	0.9300

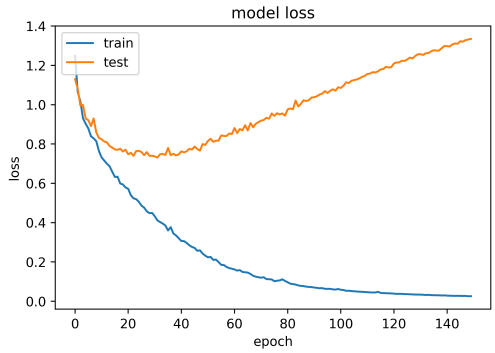
We can have a look at how behave some losses during the training phase for various number of training data :



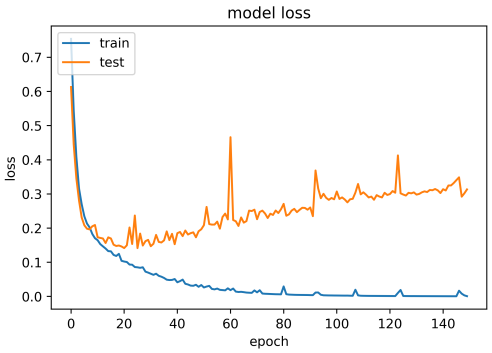
(a) Evolution of the accuracy through epochs (training data size = 300) **(b)** Evolution of the accuracy through epochs (training data size = 10000)

Figure 4: Comparison of the evolution of the accuracy using various size of training data

As we can see the more data we have, the more accurate our model generalize to the test dataset.



(a) Evolution of the loss through epochs (training data size = 300)

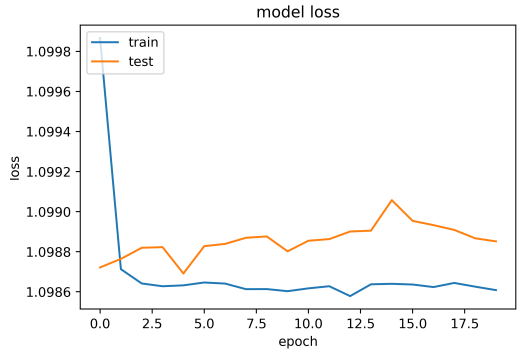


(b) Evolution of the loss through epochs (training data size = 10000)

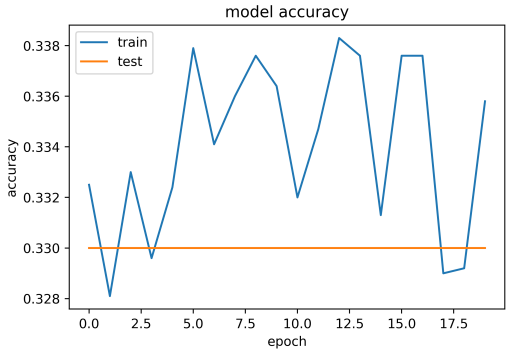
Figure 5: Comparison of the evolution of the accuracy using various size of training data

Using VGG to solve the classification task

In order to improve the outcome previously obtained. We could try to use one of the large and efficient architecture seen in class. We can for instance easily train a `VGG16` model for our classification task using adam optimizer with default parameters. Here are the loss and accuracy plot obtained through epochs :



(a) Evolution of the loss through epochs



(b) Evolution of the accuracy through epochs

Figure 6: Performance of VGG16 for our task training data set of 10 000 instances (no pretraining)

As we can see this network doesn't provide a good outcome. We haven't enough training data for this kind of complex network (14 millions examples in Image Net) and the learning rate needs to be tuned to improve the outcome. We should maybe tune the parameters in order to make the algorithm converge.

Reduce overfitting

As seen previously, its an overkill to try to use too complex and large neural networks. We can simply try to improve the performance of our deep neural network with one hidden layer. In order to perform better, we will try to create a better model that will be based on what has been seen in class. Hence, in order to regularize the model we will use in particular `Dropout` and texttttBatchNormalization. As one can see in the losses plot, the train loss converge quickly to 0 while in seems to increase from a certain point, we can use both `EarlyStopping` and `ModelCheckpoint` by using part of the training data as validation data in order to prevent overfitting by training the algorithm through too much epochs. We can also use `Dropout` in order

to improve the model generalization.

Final outcome

Using 10 000 training data (10% for monitoring overfitting).At test time using this approach we have obtained a final log loss of **0.1871** and an accuracy of **0.950** that represents quite good improvement in comparison with the outcome obtained without regularizations.

5. A Regression Problem

The task now is to predict the coordinates of the 3 vertices of a triangle, given an image of this triangle.

Normalize the data

In order to make our model performs well we will first normalize our data using the basic mean. We will note μ and σ the empirical mean and standard deviation of the training dataset output (μ and σ are vectors corresponding to each dimensions). In order normalize we will use the following formula for each of the target outputs :

$$\tilde{y}^i = \frac{y^i - \mu}{\sigma}$$

Where $\tilde{y}^i$ is the normalized version of y^i. If one need to get back the unnormalized data we simple have to inverse the formula to obtain :

$$y^i = \tilde{y}^i \sigma + \mu$$

In order to do this preprocessing we will directly use the class `StandardScaller` from `sklearn` that allows to easily handle the described normalization.

A simple linear regressor

We will first create a simple linear regressor to predict the target output. Here, the goal will be to learn the parameters the W and b that allows to go from the input images x to the target output y such that $y = Wx + b$. The parameters will be learned in order to minimize the mean squared error :

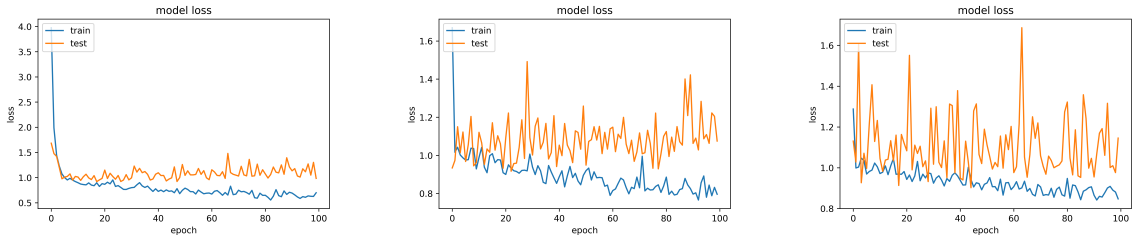
$$\sum_{(x,y)\in \text{train data}} ||y - Wx - b||^2$$

Using the test set provided by the function `generate_test_set_regression` and the normalization evoked earlier.. Here is the table obtained for various number of training data using this simple linear regressor :

Nb train data	Epoch	Train Loss	Test Loss
300	100	0.7001	0.9851
2000	100	0.7965	1.0754
5000	100	0.8476	1.1456

We can have a look at how behave those losses during the training phase :

This simple model is underfitting the training set, we must use a more complex model in order to improve performances.



(a) 300 training data **(b)** 2000 training data **(c)** 5000 training data

Figure 7: Evolution of the loss with various number of training data

A deep neural network architecture

In order to obtain better performances we will create a deep neural network architecture to solve the same task. I have used `Dropout` and `ModelCheckpoint` as regularizers We have created various models

(i) Model 1 - 1 linear layer
(ii) Model 2 - 1 Convolutional hidden layer (relu) followed by a dense layer
(iii) Model 3 - 2 Convolutional layer with ReLu activation functions followed by a dense one with dropout and monitoring using model checkpoint
(iv) Model 4 - 1 Common Convolutional layer followed by a dense network, taking this as input this 6 networks with 1 hidden layer are trained indepedently in order to do the regression for each of the components of the layer.

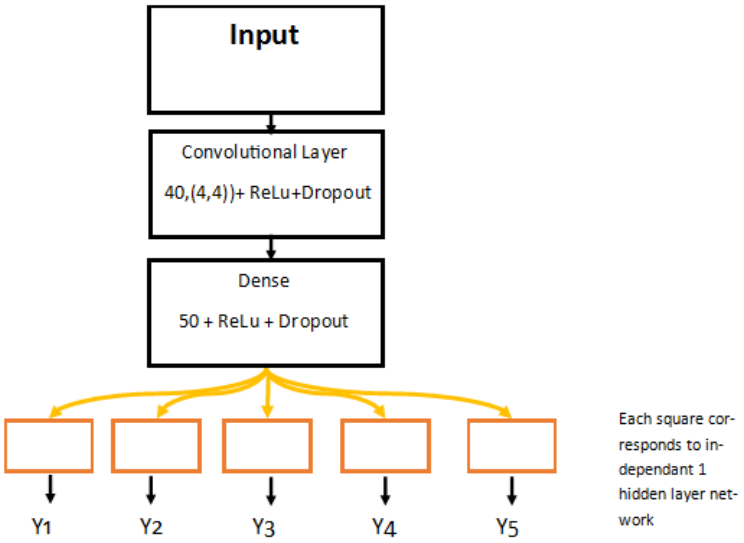


Figure 8: Model 4 Representation

Here is the table obtained for those Deep Neural Network Models using 5000 examples and normalized output.

Nb train data	Test Loss
Model 1	1.1456
Model 2	0.86
Model 3	0.76
Model 4	1.005

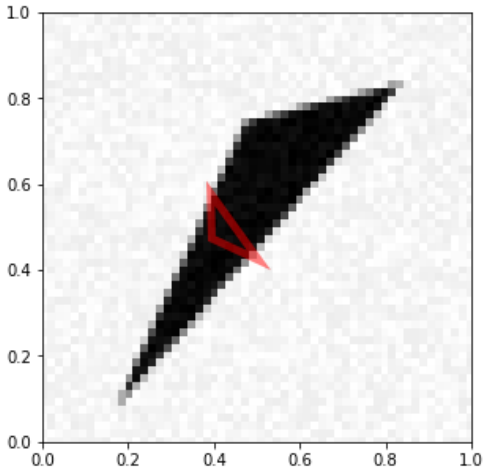


Figure 9: Outcome for the test example 100 using the Model 3

As we can see it's difficult to obtain really high performances in this regression task, one have to find a better architecture to fit this task. A basic explanation for why our algorithm could find really accurate outcome is because the triangles summits are given in an arbitrary order and we implicitly ask our network to predict the vertices in the exact order that appears to be quite difficult.

Sorting the vertices and improve the outcome

As it has been seen in the previous part, we get some awful outcomes taking the data as they were and trying to train a network on it. What makes the task difficult for the network is that we are asking him to predict the three vertices of a triangle, but those vertices are given in a random order and the network can't figure out this arbitrary order ! Hence a basic idea to improve the outcome is to sort the vertices by increasing first coordinates (x). Using 5000 examples and `ModelCheckpoint`, the Model 3 that appears to perform the best, and training the network for 50 epochs using Adam optimizer (taking the best outcome on the validation set (validation_split=0.1). **I have obtained a train loss of 0.06 and a test loss of 0.1699**.

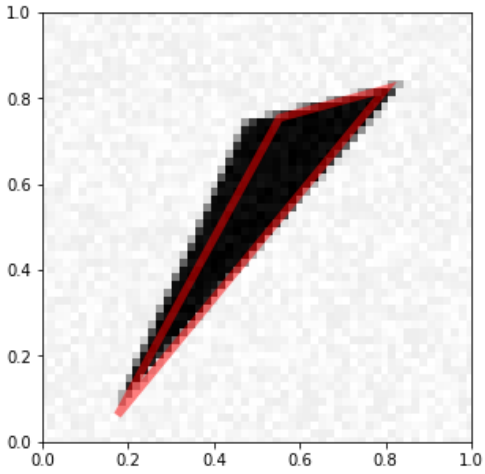


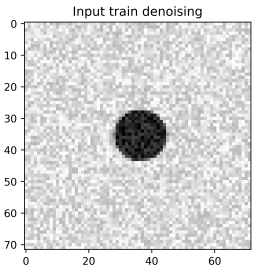
Figure 10: Outcome for the test example 100 using the Model 3 and sorting the vertices

The final outcome looks way better than the one previously obtained !

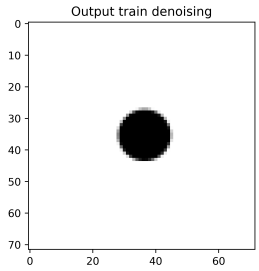
6. Bonus Question

Generating the data

In this question we will try to implement a hourglass network for denoising. We have slightly modified the `generate_a_*` functions. Setting randomly the noise amplitude between 10 and 30. We can see how looks our inputs and our target outputs :



(a) Example of training input for denoising)



(b) Example of training output for denoising)

Figure 11: Example of the generated dataset for the denoising task

Hourglass network architecture

I have created a quite simple Hourglass network to realize the denoising task. In the described convolutional layer we have used padding in order to keep the image size through convolutions. The difficult part was to control the size in order to have an outcome with the target shape.
Here is the architecture : 1 convolutional layer (60, filter size (3,3)) , a batchnormalization, a ReLu activation, a convolutonal layer (40, filter size (2,2), a batchnormalization, a ReLu activation, a **max pooling** layer (2,2) and stride of 1, 1 convolutional layer (60, filter size (3,3)) , a batchnormalization, a ReLu activation, a convolutonal layer (40, filter size (2,2), a batchnormalization, a ReLu activation, a **max pooling** layer

(2,2) and stride of 1, 1 convolutional layer (60, filter size (3,3)) , a batchnormalization, a ReLu activation, a convolutonal layer (40, filter size (2,2), a batchnormalization, a ReLu activation, a **upsampling** layer (2,2) and stride of 1,1 convolutional layer (60, filter size (3,3)) , a batchnormalization, a ReLu activation, a convolutonal layer (1, filter size (2,2), a batchnormalization, a ReLu activation, a **upsampling** layer (2,2) and stride of 1 finaly it ends with a padding and a convolutional layer (3,3).

Final outcome

Solving the problem as a regression task

A simple idea to denoise would be to solve the problem as a regression task : reconstruct the pixels as close as possible to the original one. For this purpose I used the MSE loss and 5000 samples for the training data during 50 epochs and a `ModelCheckpoint` with 10% of validation data we have obtained about 0.005 final loss on test data. As one can see the final algorithm is working fairly well :

(a) Example of test input for the denoising task)

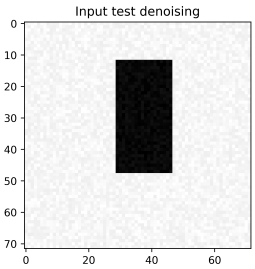
(b) Example of predicted output for the denoising task)

Figure 12: Example of denoising using Hourglass network and solving a regression task

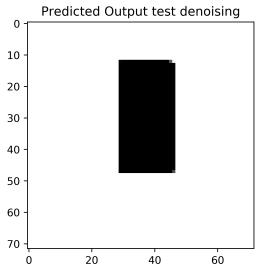
As we can see the outcome is not looking bad but we can deplore the fact that we are a poor approximation close to the border of the figure and the fact that the background is not white as before.

Solving the problem as a segmentation task

As we have seen in class, hourglass network are mainly used for segmentation purpose. Moreover, what we really care is whether the pixel is black or white. Hence we can view the denoising task as a segmentation task where we want to put the background in white and the figure in black. To do so we modify a bit the network previously described, adding a sigmoid layer. Here is an example of a figure obtained using 5000 samples as input, `ModelCheckpoint` with 10% of validation data.



(a) Example of test input for the denoising task)



(b) Example of predicted output for the denoising task)

Figure 13: Example of denoising using our Hourglass network and solving a segmentation

Here the outcome is looking way better ! About 99% of accuracy is obtained on a test set with 300 samples. This model is actually the one working the best.