Neural Networks and Deep Learning - academic year 2020-21

Homework 1: Supervised Deep Learning

Luca Dal Zotto - 1236343
luca.dalzotto.1@studenti.unipd.it

Assignment

In this homework, I will implement and test simple neural network models for solving supervised problems. It is divided into two tasks. The regression task will consist in a simple function approximation problem, similar to the one discussed during the Lab practices. The classification task will consist in a simple image recognition problem, where the goal is to correctly classify images of handwritten digits (MNIST). In both cases, but especially for the classification problem, we are encouraged to explore the use of advanced optimizers and regularization methods (e.g., initialization scheme, momentum, ADAM, early stopping, L2, L1 sparsity, dropout...) to improve convergence of stochastic gradient descent and promote generalization. Learning hyper-parameters should be tuned using appropriate search procedures, and final accuracy should be evaluated using a cross-validation setup. For the image classification task, we can also implement more advanced convolutional architectures and explore feature visualization techniques to better understand how the deep network is encoding information at different processing layers.

1. Introduction

In this homework, I have faced two **supervised learning** problems. The first one, a **regression** task, consisted in using a neural network to approximate a simple univariate function. The dataset, provided by the teacher assistant, was very small: 100 training samples and 100 test samples, where each sample was a simple input output pair.

For this first task, I tried some shallow architectures, with a relatively small number of neurons. Moreover, I focused only on a restricted set of hyper-parameters, which have been optimized through a **grid-search** procedure. Due to the small dimension of the training set, I have used a **10-fold cross-validation** approach instead of extracting a separate validation set, which probably would have represented badly the actual data distribution.

After this optimization procedure, I selected the two best models in terms of validation loss, trained them in the entire training set, and evaluated with the test set. Essentially, I simply computed the network predictions and plotted the results.

In the second part, we were asked to solve a classification task using the **MNIST** dataset. It is a large collection of hand-written digits, each of them associated with a label.

In my implementation, I trained both **fully connected** and **convolutional** architectures, exploring different **optimizers** and **regularization** techniques. Moreover, I also tested the effectiveness of **data augmentation**, by perturbing the input images. The main hyper-parameters have been optimized with a **random-search** procedure, using a separate validation set.

After having found some configurations leading to satisfactory results in the validation set, I selected three models:

1. the best fully connected architecture;
2. the best CNN not exploiting data augmentation;
3. the best CNN exploiting data augmentation.

In the first and third case, I also performed a deeper analysis of the network, by visualizing **weight histograms**, the **activation profiles** and the **receptive fields**.

The structure of the report is the following: in the next section, I will describe the two tasks more in details, discussing the architectures and the training procedure used. Instead, in the third section, I will list the most relevant results of the analysis. Both these two sections are split into two sub-sections, reflecting the task division. After some final remarks, I also included an Appendix with some additional images. However, for the complete list of the results and plots, the reader is invited to look at the notebooks LDZ_nndl_2020__homework_1_regression.ipynb and LDZ_nndl_2020__homework_1_classification.ipynb.

2. Methods

2.1. Task 1 - Regression

The first task of this project consisted in approximating a simple univariate function using a neural network. The dataset has been provided by the teacher assistant in a csv format. After having loaded it using the library Pandas, I performed a quick **data exploration** to have a brief idea about the data at handle.

First of all, it was already split in training and test set, each of size 100. Each sample was a simple input output pair. Looking at the training set, I found that the input values range was approximately (-5; 5) while the output values were in the interval (-4; 7). Then, when I visually inspected it (Figure 2), I noticed that certain input ranges were not present in the training set: indeed there are no samples with input between (approx.) -3 and -1 and between 2 and 3. If the test set includes points in these regions, probably the network will not be very accurate, since those samples will be different from what it has seen during training.

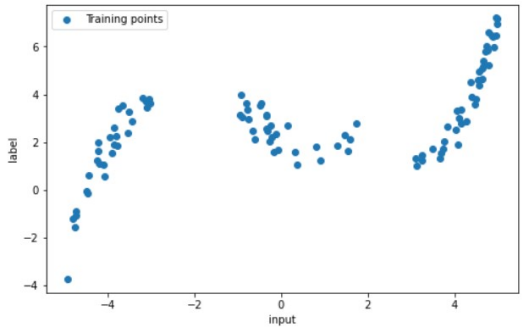


Figure 1: Training set for the regression task.

Another crucial observation is that the first rows of the training set corresponds to the points on the left-hand side of the plot, while the last ones correspond to the right-hand side. This is an aspect to keep in mind if dividing the training set into training and validation set: indeed, we may be unlucky in this split obtaining a non representative validation set, which would badly influence our hyper-parameter's choice. To solve this issue, I shuffled the dataset.

Since the complexity of the problem was not that high, I decided to use some **fully-connected** architectures with a small number of layers and hidden neurons. Another hyper-parameter that I considered for this part was the **activation function**. I tried the **Sigmoid**, the **Hyperbolic tangent** and other ones related to the **ReLU** activation:

- **LeakyReLU**: it solves the dying ReLU problem adding a small negative slope for inputs smaller than 0;
- **ELU** (Exponential Linear Unit): it solves the dying ReLU problem and also saturates for large negative values, allowing these values to be 0, leading to sparse activations;
- **GELU** (Gaussian Error Linear Unit): it is defined as $x\Phi(x)$ where $\Phi(x)$ is the standard Gaussian cumulative distribution function. The resulting shape is similar to the ReLU but it has no discontinuity;
- **Softplus** (or SmoothReLU): it is a smooth convex approximation of the ReLU function. (Figure 2)

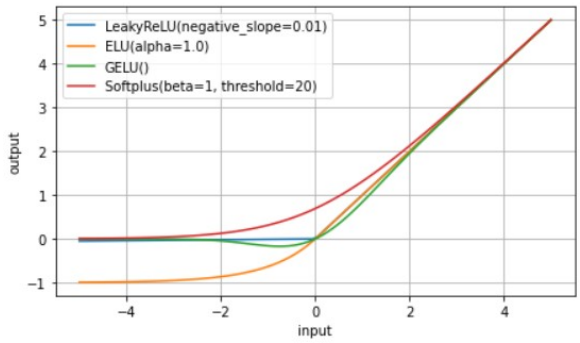


Figure 2: Comparison between some activation functions.

As loss function, I used the **Mean Squared Error**, while as optimizer, I considered **Adam**, trying different learning rates.

These hyper-parameters have been optimized performing a **grid-search**. As anticipated in the introduction, because of the small size of the training set, I decided to use a **cross-validation** procedure, instead of selecting a separate validation set. The number of folds used was 10, so, at each step, the training set had size 90, while the validation set 10. The size of each training batch was set equal to 18.

After some preliminary trials, I divided the optimization in the following stages, where, in each of them, I considered different hyper-parameters configurations:

1. networks with 2 or 3 hidden layers, each with 16, 32 or 64 neurons, Sigmoid, ELU or GELU as activation and learning rate 0.01 or 0.001;
2. same as 1, but with a larger number of hidden neurons, and also learning rate possibly equal to 0.0001;
3. same as 1, but considering also other activations (Tanh, LeakyReLU and Softmax);
4. changing the number of iterations considering only the best models.

After having identified the two best models, I re-trained them on the merged training-validation set and evaluated them in the test set, computing the loss function and plotting the model output together with the test points. Results will be discussed in the next section.

NB: at the end of the notebook where I faced the regression problem, I added an additional section, where I showed a possible way to implement a **nested cross-validation procedure**. I decided not to use it because of the higher computational cost, preferring other strategies (only an inner cross validation in the training/validation set for the regression task and a simple division in training, validation and test set for the classification task).

The first step for implementing a nested CV consists in merging the training and test sets. I implemented an outer CV with 4 folds, meaning that I have performed 4 iterations and in each of them 75% of the dataset will form the training/validation set and the remaining 25% the test set. Since in this way I have obtained 4 different test evaluations, after having terminated the whole procedure, I have reported the average value of the loss function along with its standard deviation.

Inside each iteration of the outer loop, I performed another CV over the training/validation set, this time of 5 folds. The purpose of this part is to optimize the hyper-parameters. In my simple example, I optimized just the number of hidden layers with 2 and 3 as possible values. After having found the configuration offering the best average validation loss, the inner loop ended, and the model with the best hyper-parameters has been re-trained in the full training/validation set and evaluated in the the test set. More details about the implementation can be found in the notebook.

2.2. Task 2 - Classification

In the second part of the homework, I faced a **classification** task. For this purpose, I considered the **MNIST** dataset (Figure 3). It is a collection of images which depict handwritten digits. To give more details, each sample is a gray-scale image of 28 by 28 pixels and has an associated label. The problem is therefore a **multi-class** classification task (with 10 mutually exclusive classes).

Figure 3: Examples from the MNIST dataset.

It is one of the public available datasets in **Pytorch** and it is divided into a training and test set of 60,000 and 10,000 PIL images respectively. Since we have a large dataset for a relatively simple task, in this case, I decided to use the classic division in training, validation and test set, since a cross validation approach would require high computational resources. In this way, I had the possibility to test a wider range of options, in a smaller amount of time.

The validation set considered was composed by 10,000 images randomly extracted from the training set. By looking at the class frequency (Figure 8), I noticed that each subset was balanced, therefore, accuracy would have been a reliable metric for this task.

At this stage, it is possible to implement a data augmentation strategy, exploiting some of the Pytorch transformations. Essentially, it consists in modifying the input images adding some forms of noise. This is an effective regularization technique, since each time we use a certain input in the training loop, it is always different due to these artificial modifications, and this often implies a better generalization capacity. Since in this case the digits are always more or less in the middle of an image, large translation or crops would not be useful. Instead, I applied different kind of distortions or rotations to the images (Figure 4).

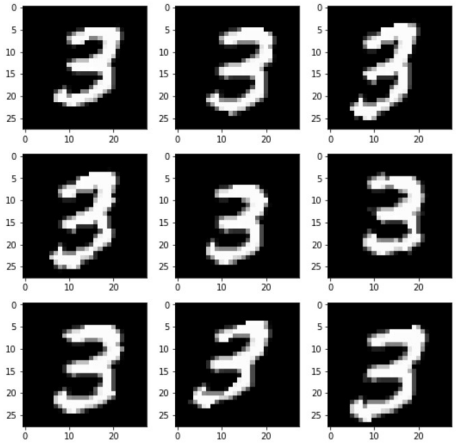


Figure 4: Effect of data augmentation to a single input.

Regarding the networks, I considered both fully connected and convolutional architectures. Since the input is an image, I expected that the second type could reach better performances. In both cases, I defined a function having as arguments the most relevant hyper-parameters of the networks, such as the number of **hidden layers**, **neurons** per layer, **kernel** size, **stride** and the **activation functions**. Moreover, I also considered some **pooling layers** (having a black background, the max-pooling operation is the most effective). Note that to avoid reducing too much the size of the tensors, I made some constrains in the hyper-parameter choices: for example, when using more than 2 convolutional layers, the padding size could not be zero, and also strides larger than 1 and max pooling layers were not used at the same time.

For what concerns the regularization techniques adopted,

I tried **dropout** layers with a certain probability p after the fully connected layers and the **L2 regularization** (actually, I modified the **weight decay** parameter of the optimizers, but it can be proved that under some conditions, these two approaches are equivalent). Moreover, I also tested the effect of **batch normalization**, which should improve learning stability. However, I always used at most one of these three methods at a time, to avoid possible interference.

To prevent overfitting, I implemented from scratch an **early stopping** routine, which interrupts the training loop if the validation loss is not improved for a certain number of successive epochs (**patience** parameter). In any case, I also set a maximum number of epochs.

The loss function considered was the **Cross Entropy**, while for the **optimizers**, I tried different ones: **SGD**, **RMSprop** and **Adam**. For the first one, I considered a basic implementation without momentum. The advantage of the other two consists in **adapting the learning rate** individually for each weight depending on the training circumstances. In particular, RMSprop considers a running average of past values of a certain weight to modify the learning rate of that weight, while Adam optimizer takes into consideration also an estimate of the second moments of the gradients.

In order to find a satisfactory configuration of the previously mentioned hyper-parameters, I used a **random search** approach, where the values of each hyper-parameter were sampled from a list of candidates. For each configuration tested, I computed training and validation loss and accuracy, and I also plotted the learning curves to have an additional feedback about the learning process.

The optimization was done in more than one stage Indeed, I started by considering a wide range of values for each hyper-parameter, and once individuated a region in the space leading to good results, I focused on those values making other more precise random searches.

After having found some satisfactory configurations (in terms of validation accuracy), I evaluated them using the test set. I decided to consider three models: the best fully connected network, the best convolutional network not using data augmentation and the best CNN using data augmentation. After having computed the final accuracy, I analyzed more in depth the first and the third architectures by visualizing the weight histograms, the activation profiles and the receptive fields.

3. Results

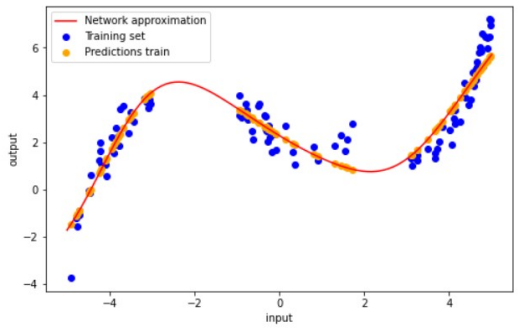
3.1. Task 1 - Regression

As described in the previous section, I optimized some of most important hyper-parameters using a 10-fold cross-validation procedure. After having completed the loop, for each configuration, I computed the average train and validation loss over the 10 folds. The validation loss was the criterion used for selecting the best hyper-parameters. In order to have an additional feedback, at each step, I also plotted the output of the network trained in the full training-validation set. I noticed that the validation loss was indeed a reliable measure for evaluating the model, since the smoothest outputs corresponded to the smallest validation losses.

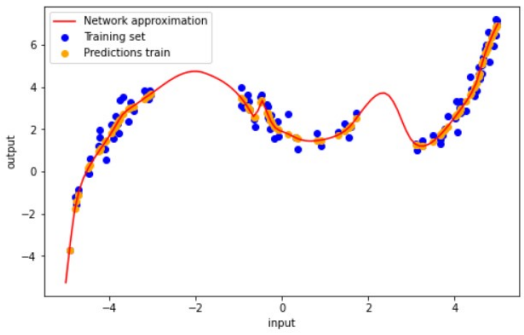
The best results (in terms of average validation loss) were achieved with 3 hidden layers and using ELU or GELU activation functions. To be precise, the hyper-parameters of the best models were:

- 3 hidden layers with 64 hidden neurons, ELU activation and learning rate = 0.001;
- 3 hidden layers with 32 hidden neurons, GELU activation and learning rate = 0.001;

Then, I further tuned these two models, considering different number of epochs: 100, 250, 500, 1000, 2000 or 5000. For the first model, the best value was 500, while for the second 1000. Looking at the output of the models using a smaller or larger number of epochs, it is evident the under-fitting or over-fitting phenomenon due to the wrong model complexity (Figure 6).



(a) *Example of under-fitting with 100 epochs.*

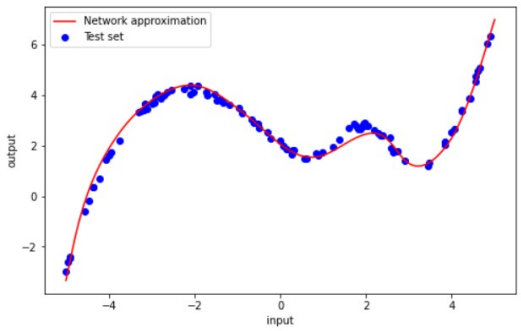


(b) *Example of over-fitting with 5000 epochs.*

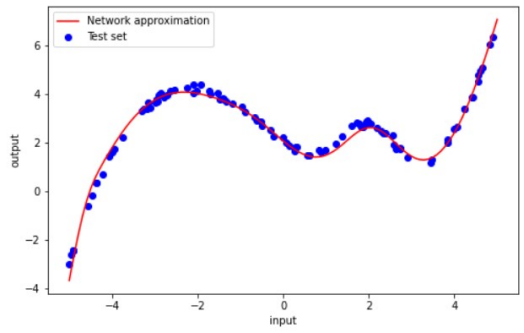
Figure 5: Under-fitting and over-fitting examples.

In the first case, the model is not able to fit the training set well enough, while, in the second case, the model tries to describe more accurately the training set, introducing some artifacts (the most evident is that with input values around -1), reducing the level of smoothness of the output function.

To conclude, I evaluated the two best models in the test set. The model using ELU reached a test loss of 0.049, while the one using GELU 0.045. In the following figures, I represented the model approximations and the test dataset.



(a) *Model using ELU activations.*



(b) *Model using GELU activations.*

Figure 6: Evaluation of the final models.

As expected, the larger prediction errors occur in the regions where we did not have any training point (such as for input values around 2). However, in both cases, the fit is satisfactory.

3.2. Task 2 - Classification

When implementing the second part, I defined two types of fully connected networks and three types of CNN. The difference lies in how the parameters could change, modifying the overall structure. This made easier the organization of the optimization procedure in different stages. For example, when optimizing the fully connected architecture, in the first random search I used the same number of neurons for each hidden layer, just to have an initial idea of the complexity required by the network. Then, in later successive searches, I also allowed different number of hidden neurons for each layer.

In the following list, I report the main hyper-parameters considered.

- fully connected layers
- hidden neurons
- activation function
- convolutional layers
- number of kernels
- kernel size
- stride
- pooling layers
- optimizer
- learning rate
- regularization
- weight decay (if L2)
- train batch size
- max number epochs
- patience parameter

All the values tried for each hyper-parameter and the complete search procedure can be found in the notebook.

Let's now focus on the best architectures. The fully connected network able to reach the higher validation accuracy had just two hidden layers with respectively 4096 and 1024 neurons. It used LeakyReLU as activation function and batch normalization. It has been optimized using Adam, with a learning rate of 0.003, and the number of epochs required was 40. After having trained this configuration in the merged training-validation set, it has been evaluated in the test set, obtaining an accuracy of 99.06%, which is very high for an architecture using only dense layers. Moreover, for this architecture I visualized the weight histograms (Figure 9) and the activations of each layer (Figure 10). These figures are reported in the Appendix section.

Secondly, I considered the best CNN that did not use data augmentation. It has 4 convolutional layers. The first and the third have 16 kernels, while the second and the fourth 64. The kernel size of the first layer is 7x7, while for the other layers 3x3. The stride used in all cases is 1, but for the first layer, a 0-padding of size 2 is applied. Moreover, the first and third layers are followed by a max-pooling layer. After the application of the flattening operation, I added a fully connected layer with 512 neurons, a batch normalization layer and another fully connected layer of 128 neurons. In this case, the optimizer was RMSprop with a learning rate of 0.003. This configuration led to a test accuracy of 99.20%.

Finally, I also considered the best CNN exploiting data augmentation, to test the effectiveness of this strategy. In this case, the network architecture was simpler: only 2 convolutional layers with 128 kernels of size 5x5 (each of them followed by a max-pooling layer) and a single fully connected layer with 64 hidden units. The optimizer used was still RMSprop but with a learning rate of 0.001.

Moreover, the number of training epochs was just 32. In this way, I was able to reach a test accuracy of 99.62%. For completeness, in Figure 11, I reported the confusion matrix, while, in Figure 12, the classification report for this last model.

To conclude, I analyzed different aspects of this last network, by visually inspecting the following points:

1. weight histograms of the two fully connected layers (Figure 13): for the hidden layer, the distribution was approximately bell-shaped with mean around 0, while for the output layer, the distribution was much more different, and there was no more a pick in 0;

2. hidden layer activations (Figure 14): I considered different input digits (in the picture, 0, 3 and 5), and I plotted their different activation patterns after the hidden layer;

3. kernels of the 2 convolutional layers (Figure 15): in some cases, we obtain something that may resemble the derivative of Gaussian filter or the Sobel filter, but, in most of the cases, the interpretation is not so simple, especially for the second convolutional layer;

4. the activations after the convolutional kernels obtained with a certain input image (Figure 16): after the application of the first convolution, we can see that the edges of the digit are highlighted in many different ways, in order to extract as much information as possible. After the second convolution, the digit is more difficult to recognize. This is due to the fact that the deeper we go, the more complex and abstract the features become.

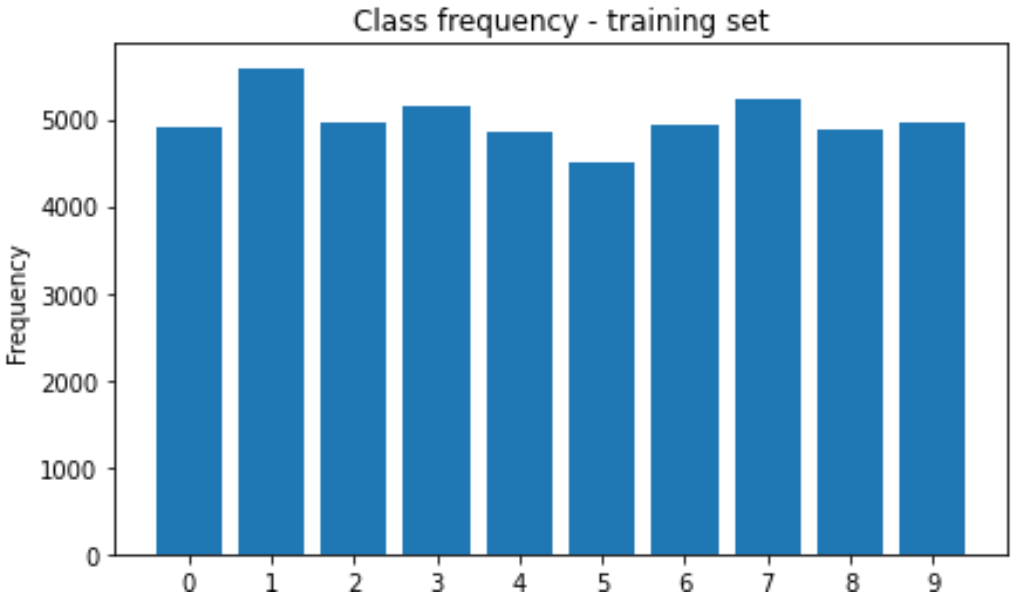
4. Conclusions

In this homework, I have faced a regression and a classification task. In the first case, I have used a neural network to approximate a simple univariate function. For this purpose, I have defined some relatively simple architectures. Regarding the hyper-parameters, I focused on the number of layers, number of neurons, the activation function, number of iterations and learning rate.

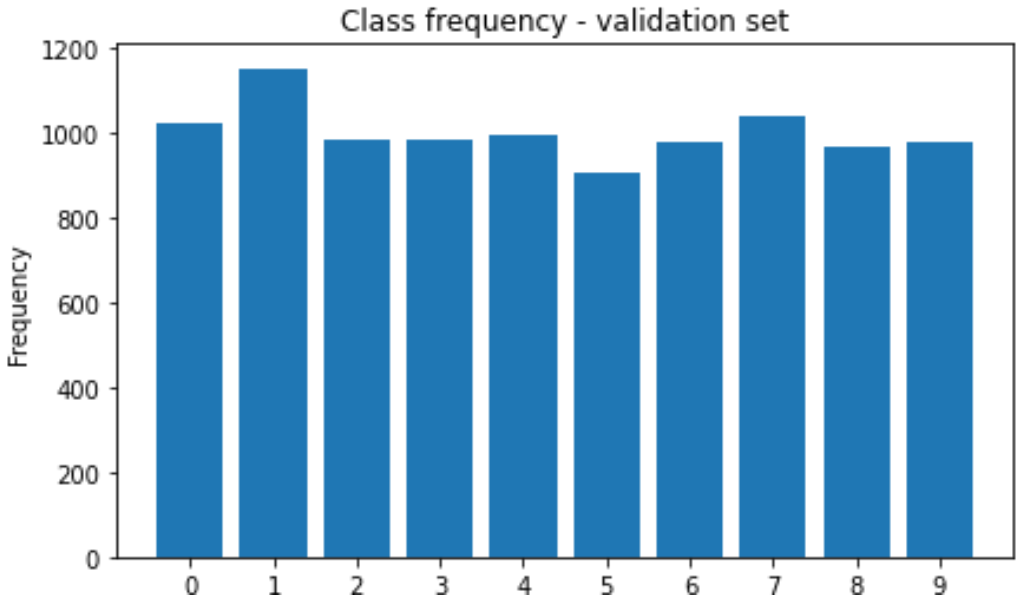
Many other learning tools, such as advanced optimizers and regularizers, have been explored in depth in the second part, where I have considered different hyper-parameters and learning strategies. Most importantly, I have tested the effectiveness of CNN as feature extractor, when the input (in this case an image) has a particular structure, and also the improvements, in terms of generalization capacity, obtained using data augmentation.

Appendix

4.1. Task 1 - Regression

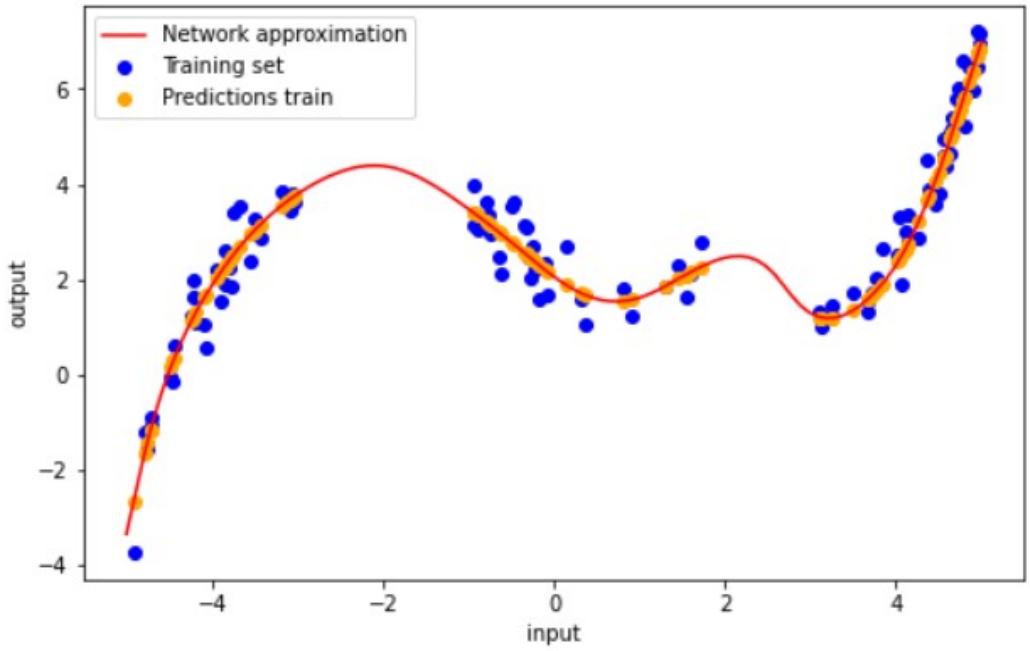


(a) *Training set.*

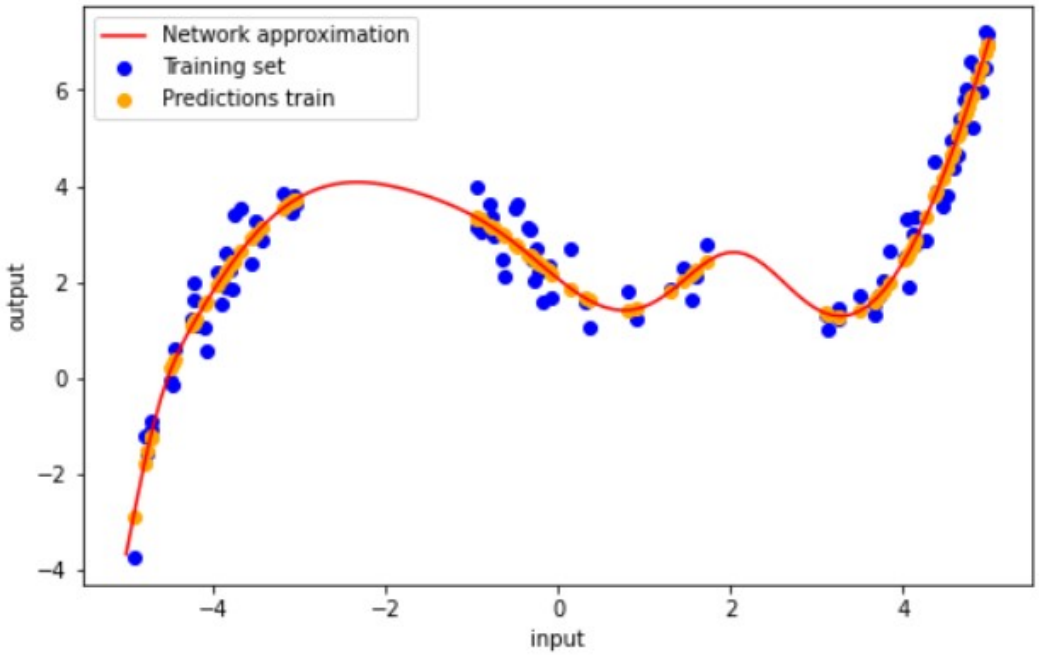


(b) *Validation set.*

Figure 7: Class frequency of training and validation set.



(a) *Model using ELU activations.*



(b) *Model using GELU activations.*

Figure 8: Training points and network approximation of the two best models.

4.2. Task 2 - Classification

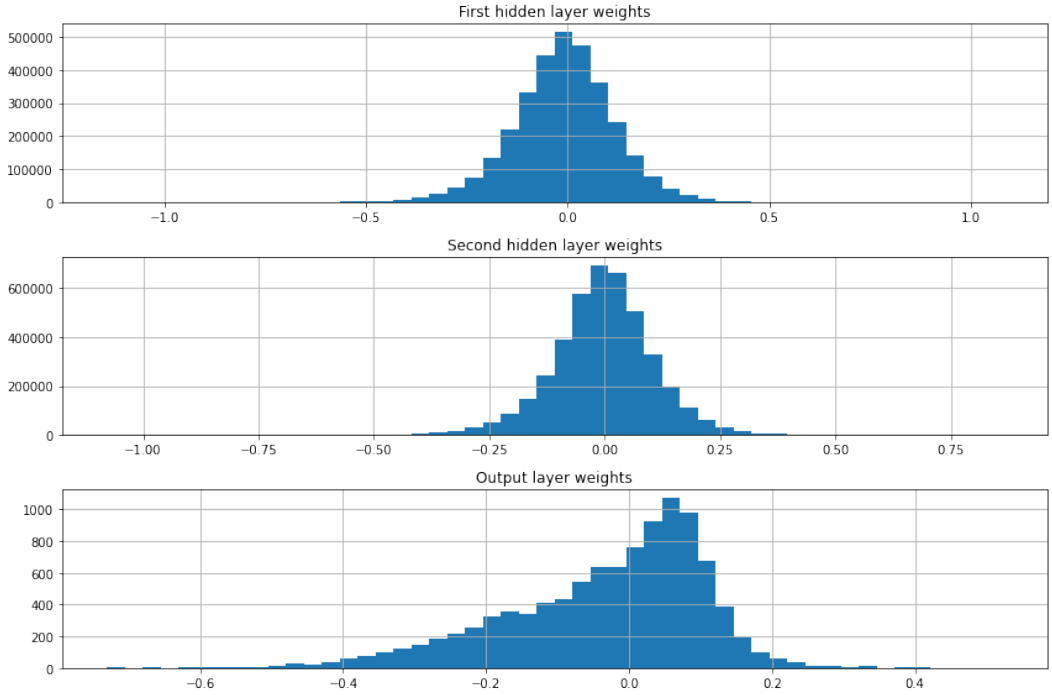


Figure 9: Weight histograms of the best fully connected network.

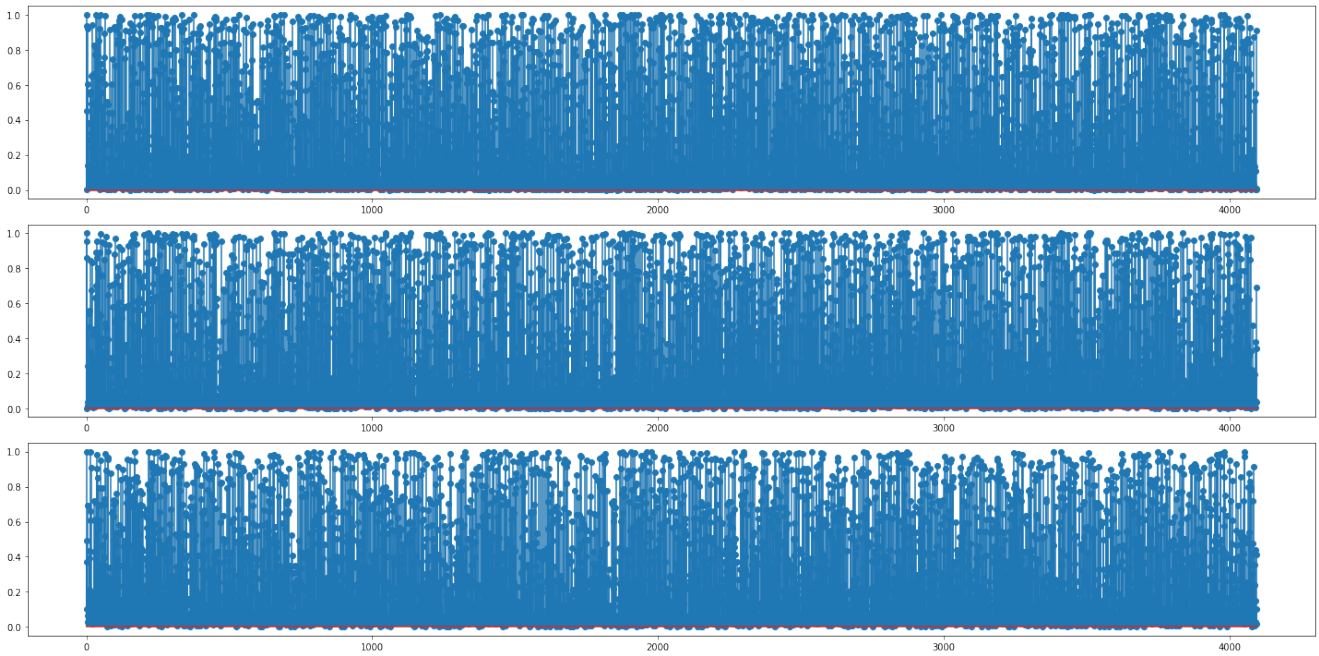


Figure 10: Activations after the first hidden layer of the best fully connected network for 3 different inputs.

	0	1	2	3	4	5	6	7	8	9
0	979	0	0	1	0	0	0	0	0	0
1	0	1134	0	1	0	0	0	0	0	0
2	0	0	1027	1	0	0	0	3	1	0
3	0	1	0	1008	0	1	0	0	0	0
4	0	0	0	0	974	0	1	1	0	6
5	0	0	0	3	0	888	1	0	0	0
6	2	0	1	0	1	0	953	0	1	0
7	0	2	2	0	0	0	0	1022	1	1
8	1	0	2	0	0	0	0	0	971	0
9	0	0	0	0	2	1	0	1	0	1005

Figure 11: Confusion matrix of the best CNN model.

	precision	recall	f1-score	support
0	1.00	1.00	1.00	980
1	1.00	1.00	1.00	1135
2	1.00	1.00	1.00	1032
3	0.99	1.00	1.00	1010
4	1.00	0.99	0.99	982
5	1.00	1.00	1.00	892
6	1.00	0.99	1.00	958
7	1.00	0.99	0.99	1028
8	1.00	1.00	1.00	974
9	0.99	1.00	0.99	1009
accuracy			1.00	10000
macro avg	1.00	1.00	1.00	10000
weighted avg	1.00	1.00	1.00	10000

Figure 12: Classification report of the best CNN model.

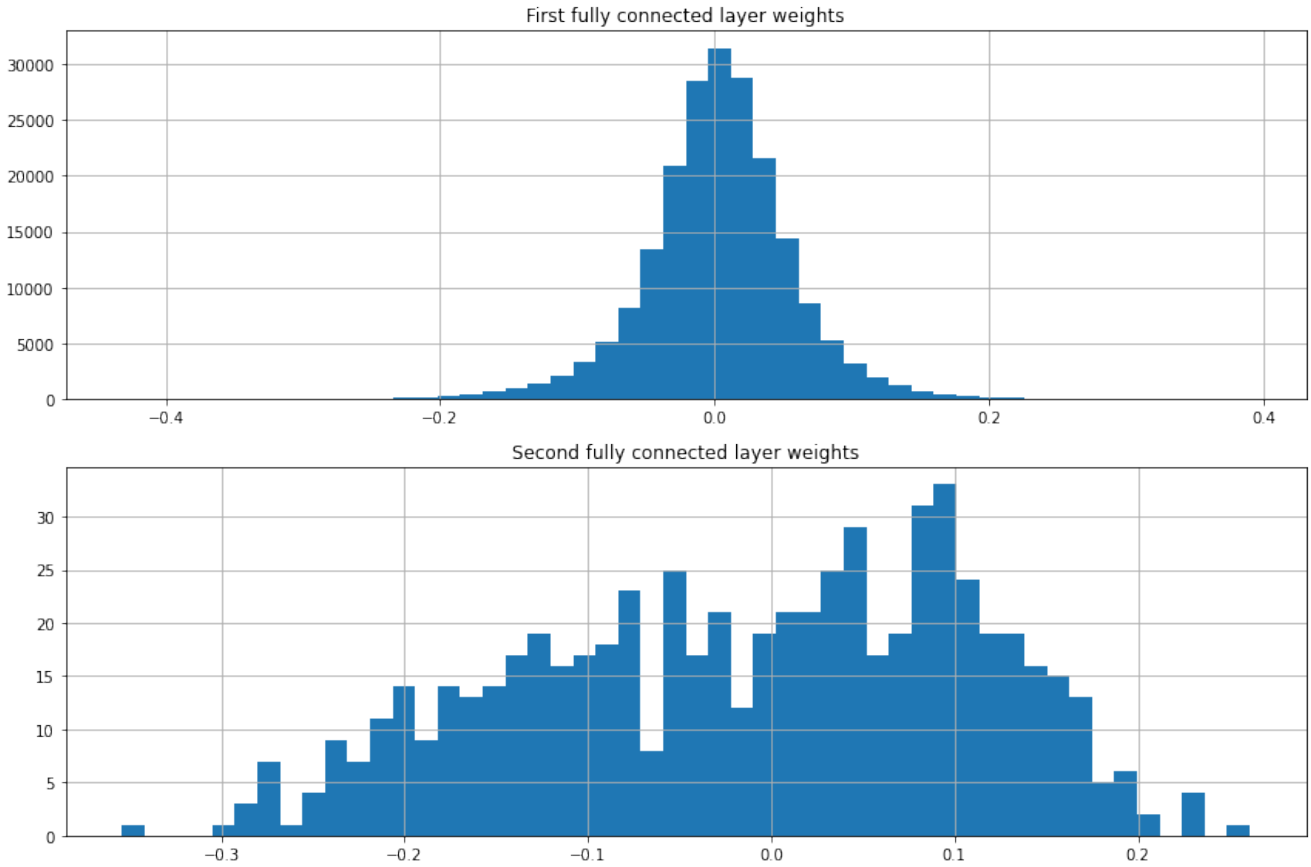


Figure 13: Weight histograms of the best CNN model using data augmentation.

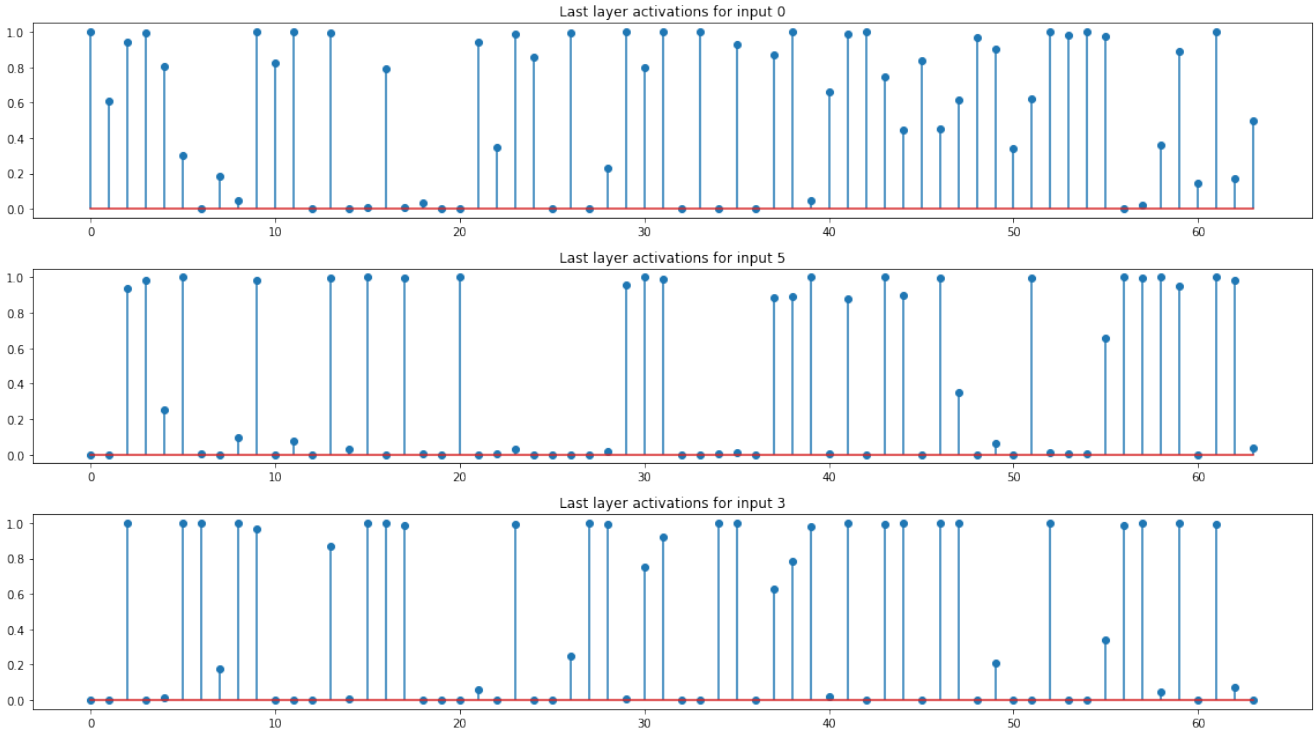
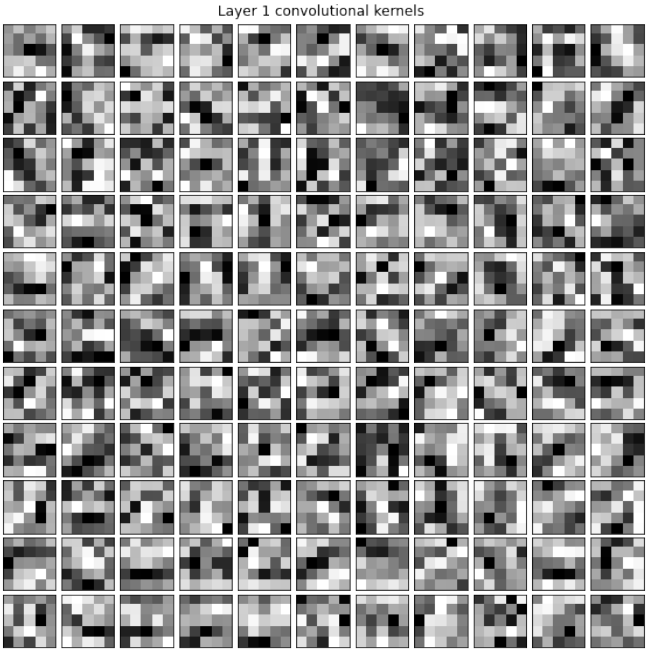


Figure 14: Hidden layer activations for 3 different inputs.



(a) *Layer 1 convolutional kernels.*

(b) *Layer 2 convolutional kernels.*

Figure 15: Kernels of the 2 convolutional layers.

(a) *First convolutional layer.*

(b) *Second convolutional layer.*

Figure 16: Activations of the two convolutional layers.