# Homework1, Regression and Classification tasks

Daniele Foscarin

23 January 2020

## 1 Regression task

### 1.1 Introduction

The goal of this homework is to implement and evaluate a neural network that is capable to perform the modeling of an unknown function.
In other words, given a function $f : \mathbb{R} \rightarrow \mathbb{R}$ we want to train a neural network so that it approximate the function $NN(x) \approx f(x)$. We have only access to some measures from the target function that are performed with the addition of noise, $\hat{x} = f(x) + noise$.
We use those measures as the training set for a neural network implemented in a python notebook using the PyTorch framework.

### 1.2 Method

#### 1.2.1 Preparation

At the very beginning of the notebook we import the libraries used for the data manipulation like numpy and pandas, as well as the useful modules from PyTorch. We also install and import the library *torch_optimizer* that includes a set of optimizer that have been recently developed and are not widely used as the optimizers that are present in the default *torch.optim* package.
The data used as the training set and the test set is loaded as a Pandas DataFrame from two .csv files, so we define an extension of the *torch.util.data.Dataset* class that is able to read the data from a DataFrame variable and it can provide a DataLoader that will be used in order to implement an efficient pipeline for the data during the training process.

#### 1.2.2 Definition of the model

We can easily define a model for our regression task extending the class torch.nn.Module. We implement a model with three fully connected layers, activated by the sigmoid function. A fourth layer, defined as the output layer, is defined without any activation function since in this task we do not want to apply any normalization or constraint on the output values during a regression task.
The number of neurons in every layer is given in input, since we want to control these parameters during the grid search process.

### 1.2.3 Grid Search and k-fold Validation

It is a good practice to implement k-fold validation when we have a small number of samples in the training set, like in this case. So instead of splitting a priori the training set into the actual training set and a validation set, we perform every steps in the training process using the training set splitted into 5 subsets, and we use alternatively one of those subset as the validation set, while the others represent the actual training set. this solution allow us to reduce the problem caused by a potential bad choice of the samples of the validation set.
Every consideration about the quality of the models we explore during the grid search will be performed using the averaged values obtained during the training in the 5 folds.
In order define a model and a training process that lead us to a good solution we performed multiple trainings using different set of hyperparameters regarding the network model and the training process. This allow us to be more confident about the quality of the model that we use in the end in order to solve our regression task.
We choose to explore the results obtained with the variations of 5 parameters. They are

- optimizer type, chosen from Adam, DiffGrad, RAdam;
- batch_size, chosen from [4,8,16];
- first layer size, chosen from [512, 1024];
- second layer size, chosen from [512, 1024];
- third layer size, chosen from [512, 1024].

So we need to evaluate 72 sets of parameters.

We choose to implement in the grid search the optimizer Adam since it is one of the most widely used, and also the optimizers Diffgrad[1] and RAdam[2] that are available in the library *torch.optimizer*. We choose those since it was interesting to try and compare some unusual optimizers against the well established Adam. Diffgrad and RAdam were chosen among the ones implemented in the *torch.optimizer* library since, using default parameters, they were able to reach the global minimum during the evaluation of the Rastrigin function [3]. For those optimizers we kept the default values for the learning rate (that is $1e-3$) and the others algorithm specific parameters.
In this regression part we also did not applied any regularization technique as the weight decay inside the optimizer (differently from the classification task). The motivation behind this choice is that since the effect of regularization is to reduce the influence on the network weights that some samples can assume, due to their outlying characteristics with respect to the other samples in the dataset, and since we can observe that the training set does not present such outlying samples and we have and already small number of samples available, the use of weight decay could probably decrease the performances of the model. We used mean squared error as the loss functions in every training. We use a fixed number of epochs for every training, that is 100 epochs.

The grid search with k-fold cross validation was implemented with the following steps:

[1]diffGrad:An Optimization Method for Convolutional Neural Networks. (2019)[https://arxiv.org/abs/1909.11015]
[2]On the Variance of the Adaptive Learning Rate and Beyond (2019) [https://arxiv.org/abs/1908.03265]
[3]https://pythonawesome.com/a-collection-of-optimizers-for-pytorch/

- We generate a DataFrame containing every combination of the parameters we want to evaluate.
- for every combination we call the function that perform the trainign for every fold and we store its output.
- the *kfold_train()* perform the training of the model five times (one for every forld) using the current set of parameters. In order to do this, the model, the DataLoaders and the optimizer are instantiated inside this function using the chosen parameters. For the five trainings, we store the values of the validation loss obtained during the training.
- For every set of parameters we get in a DataFrame the data about the validation loss in 100 epochs for 5 folds. We add a column that is the average of the validation loss among the 5 folds. We finally evaluate the set of parameters computing the average on the last 10 epochs of the averaged loss. We use the averaged value in the last 10 epochs instead of simply the value during the last epoch in order to reduce the effect of the noise in the validation loss, due to the small number of samples available.

#### 1.2.4 Training the best model

At the end of the grid search we can use the values computed before to identify the best set of parameters, that is:

| | |
|---|---|
| batch size | 4 |
| optimizer type | Adam |
| fully connected layer 1 | 1024 |
| fully connected layer 2 | 1024 |
| fully connected layer 3 | 1024 |

At this point we can use the best parameter to train the best model using the entire training set and evaluating it with the test set.

### 1.3 Results

The final loss in the test set is 0.2163 (Figure 1). We can feed the network with a np.linspace vector in the range of values of the samples from the training set in order to scatter the approximated function in the 2 dimensional space and compare it against the function described by the noisy samples of the dataset. We can see that it approximate the function with good precision where the train set but the reconstruction was not able to properly approximated it in the neighbor of the local maximum at $x = -2$ and $x = 2$ (Figure 2). This result was expected since in the training set there were not samples in that neighbor. Since our model underfitted the data, we could define a more complex model using more neuron per layer or a higher number of layers that could probably better approximate the function even the areas that are not well represented in the training set.
We can visualize the values learned by the network displaying the histogram of the values assumed by the weights (Figure 3) and the activation values of the different layers (Figures 4, 5).

# 2 Classification task

## 2.1 Introduction

The second part of this homework implements a simple classification task using the MNIST dataset. More precisely our goal is to develod a model that, given in input a batch of 28x28 pixel image, it is able to produce a batch of ten elements vectors which state the membership of the input samples in one of the ten classes.

## 2.2 Method

### 2.2.1 Definition of the models

We define a convolutional model CnnNet extending the class torch.nn.Module. It contains two convolutional layers and two fully connected layer. After the first convolutional layer the the tensor gets reduced with a MaxPool2d layer that halves the number of pixel for every sample,

- Convolutional layer, kernel size is 3, padding 1, activation function ReLU, dropout.
- MaxPool2d layer halves the number of of pixels for every sample.
- Convolutional layer, kernel size is 3, padding 1, activation function ReLU, dropout
- Fully connected layer with activation function ReLU and dropout.
- Fully connected output layer with activation function LogSoftmax.

The number of channels in the convolutional layers, the number of neurons in the first fully connected layer and the dropout value is given in input to the model constructor. The output layer has 10 neurons since we have 10 classes in the dataset.
After the convolutional layers, we needed to flatten the tensor and to assign the right input size at the fully connected layer. The size of every axis can be computed using this formula:

$$out\_size = \left\lfloor \frac{in\_size + 2 \times padding - kernel - (kernel - 1)(dilation - 1)}{stride} \right\rfloor + 1.$$

So remembering that we used the default values $dilation = 0$ and $stride = 1$, and we applied maxpooling once, the first fully connected model receives an input tensor of size equal to the number of channels in the second conv layer multiplied by $14 \times 14$.

### 2.2.2 Random Search

Similarly to the regression task, we implmented a grid search in order to explore the performances of the model using many set of parameters. The varying parameters are the following:

- batch size, chosen from [64, 128, 256],
- weight decay value, chosen between 0 and $4e - 3$[4],

[4]The value of weight decay $4 \times 10^{-3}$ is taken from this paper Leslie N. Smith *A disciplined approach to neural network hyperparameters* https://arxiv.org/pdf/1803.09820.pdf

- learning rate, chose from [0.0005, 0.001,0.005][5],
- conv1 layer number of channels, chosen from [8,16,32],
- conv2 layer number of channels, chosen from [8,16,32],
- fully connected layer numer of neurons, chosen from [50,100],
- dropout value, chosen from [0.1, 0.2, 0.3].

With this choice of parameters that need to be tested, we define 972 sets of parameters. That would make the grid search very long and we decided to randomly choose 100 combinations of the parameters among the 972 defined above. Anyway since the process is still long, we made sure to save locally the results of the search in a DataFrame every 5 trainings. In every training we used 30 epochs, NLLLoss as the lossfunction, Adam optimizer.
There is no need to set up a k-fold evaluation this time since the MNIST dataset provides for a high number of samples. We split the training dataset in order to use 50000 samples in the actual training set and 10000 samples in the validation set.
The evaluation of every set of parameters is simpler than before, at the end of every training the last value of the evaluation loss is stored in a DataFrame containing the set of values used in that training. At the end we just need to sort this DataFrame and pick as the best parameters the ones written in the first row.

#### 2.2.3 Training of the best models

The best set of parameters is

| | |
|---|---|
| batch size | 64 |
| weight decay | 0 |
| learning rate | 0.0005 |
| conv1 channels | 32 |
| conv2 channels | 8 |
| fully connected layer size | 100 |
| dropout value | 0.3 |

The model initialized with those parameters has been trained with the full training set and the test set as validation.

## 2.3 Results

At the end of the training with the best model on the full training set we obtain a on the test set a loss of 0.0246 and accuracy of 0.9928. In Figure 6 we can see that the learning stops at circa 30 epochs with no overfitting and the results on te training and the test set are very close.
Looking at the confusion matrix (Figure 7) we can see that there are only few misclassified samples in the test set. Considering also the very high accuracy that we obtained we can affirm that this model has very good performancies and it satisfies the assignment.
We can visualize the histrogram of the weigths that has been learned by the model and values of the kernels as images (Figure 8, 9).

[5]The range of the learning rate is chosen according to this article https://medium.com/octavian-ai/which-optimizer-and-learning-rate-should-i-use-for-deep-learning-5acb418f9b2

## 3 Appendix: figures

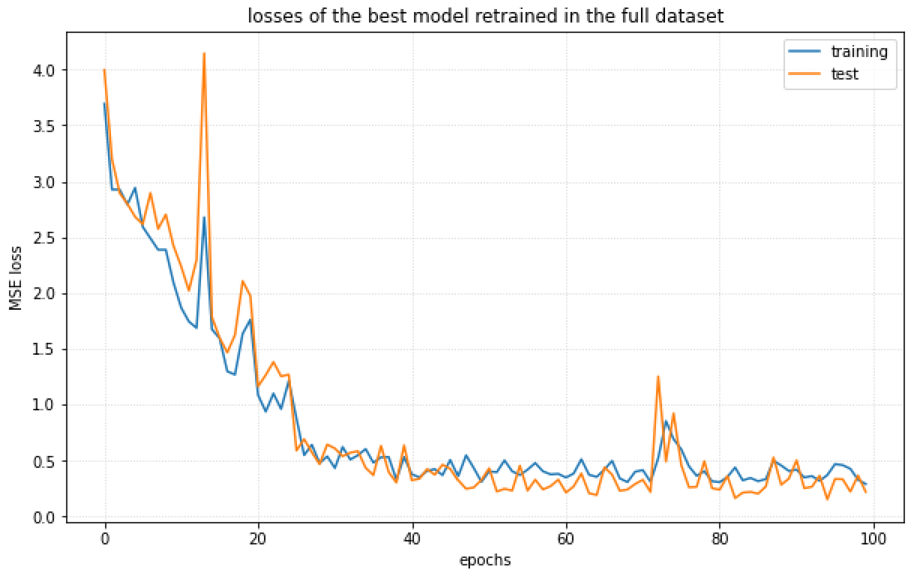


Figure 1: Training on the best model for the regression task, using the entire training set and the test set.

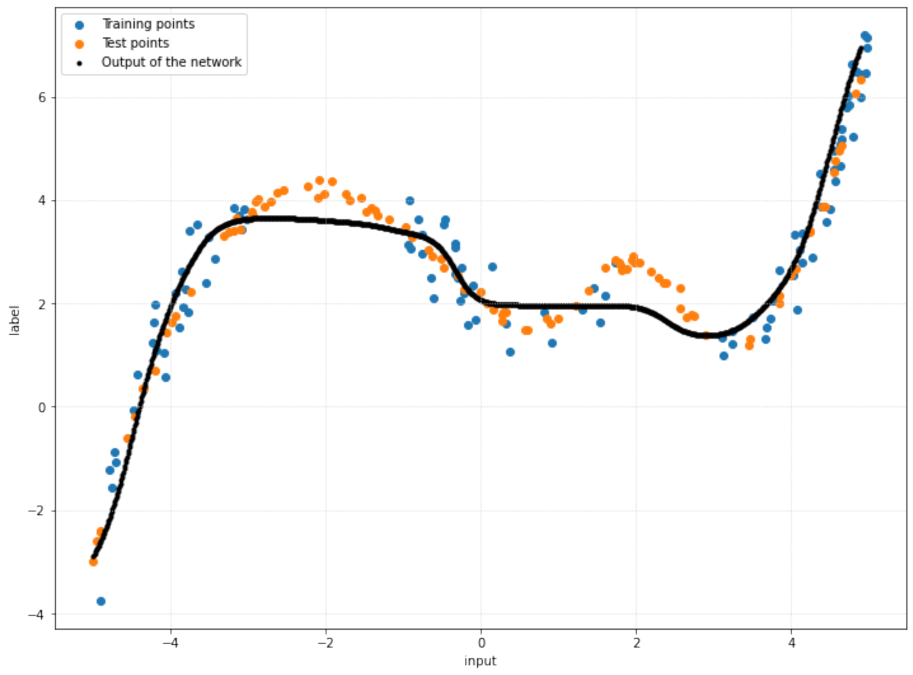


Figure 2: Scatter of the approximated function compared with the dataset.

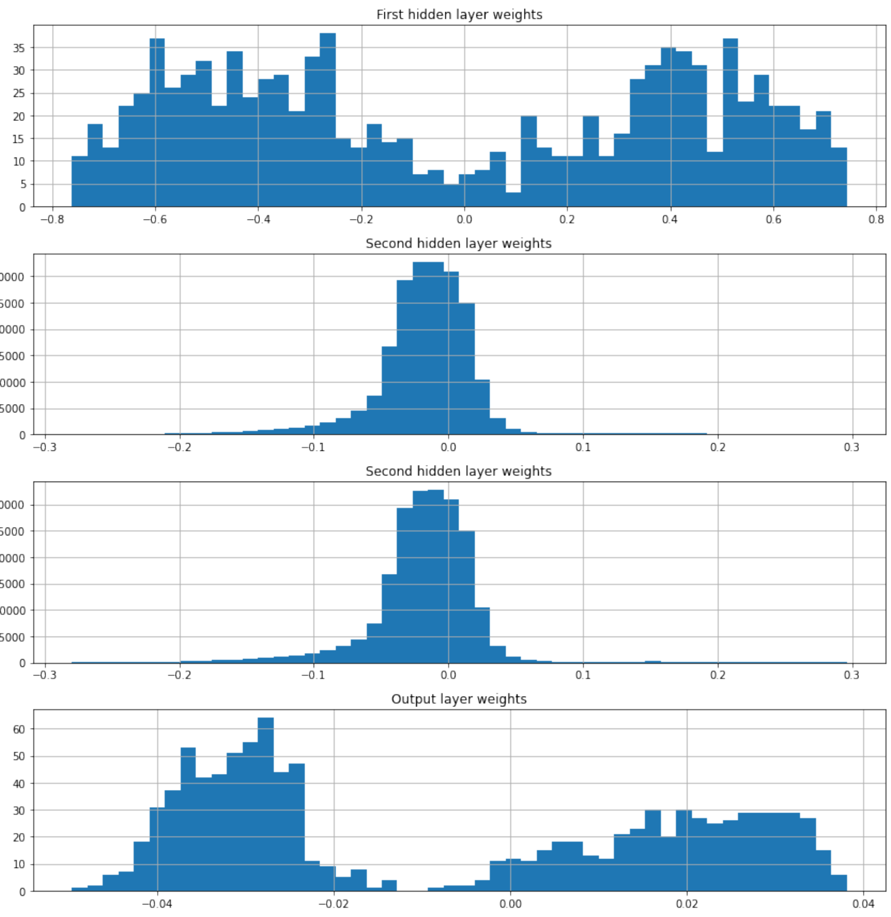
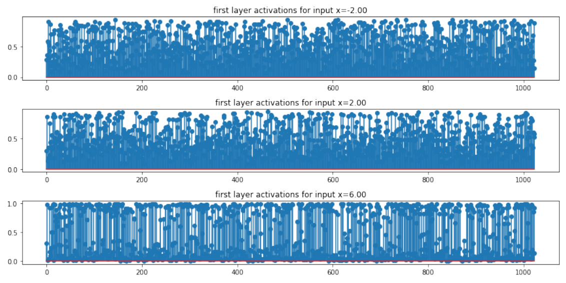


Figure 3: Histogram of the weights values in the four fully connected layers.



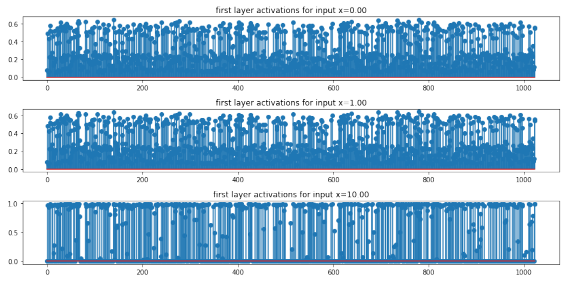


Figure 4: Examples of the activation of the neurons in the first layer.

Figure 5: Examples of the activation of the neurons in the third layer.

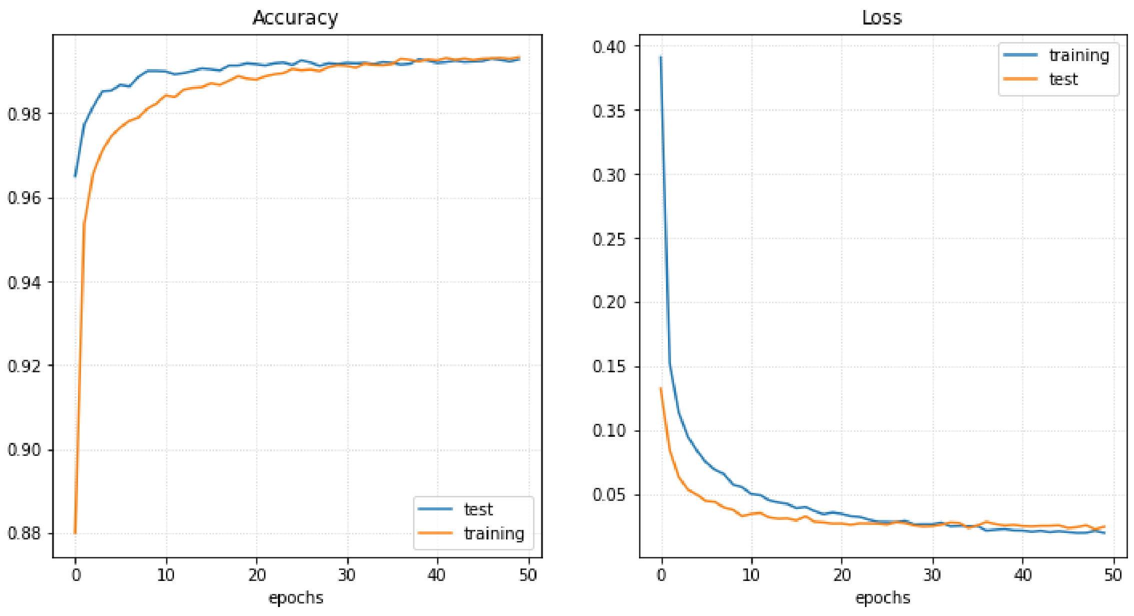


Figure 6: Loss and accuracy in the training of the best model with the entire dataset.

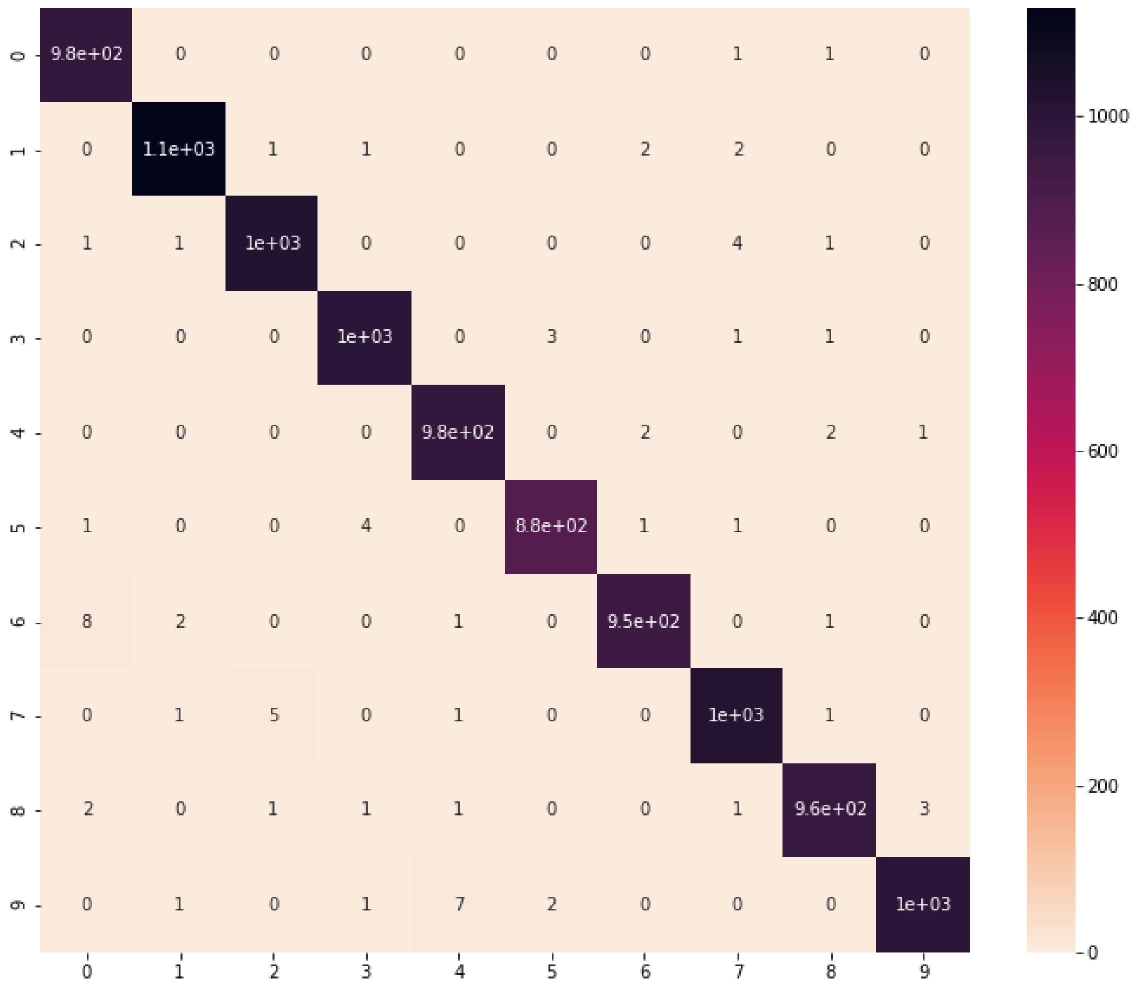


Figure 7: Confusion matrix about the classification on MNIST dataset with the best model

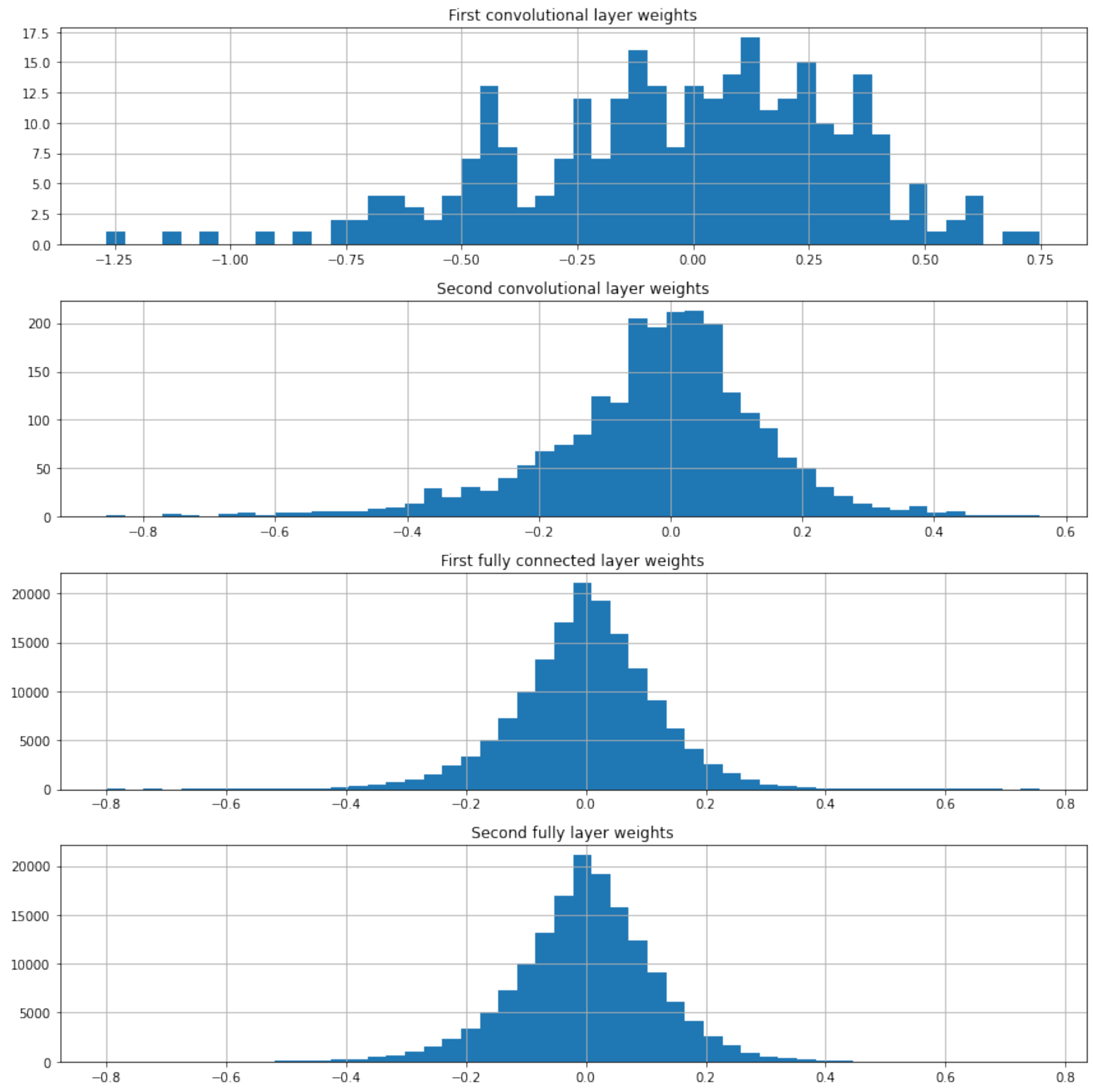


Figure 8: Histogram of the weights value in the four layers of the model.

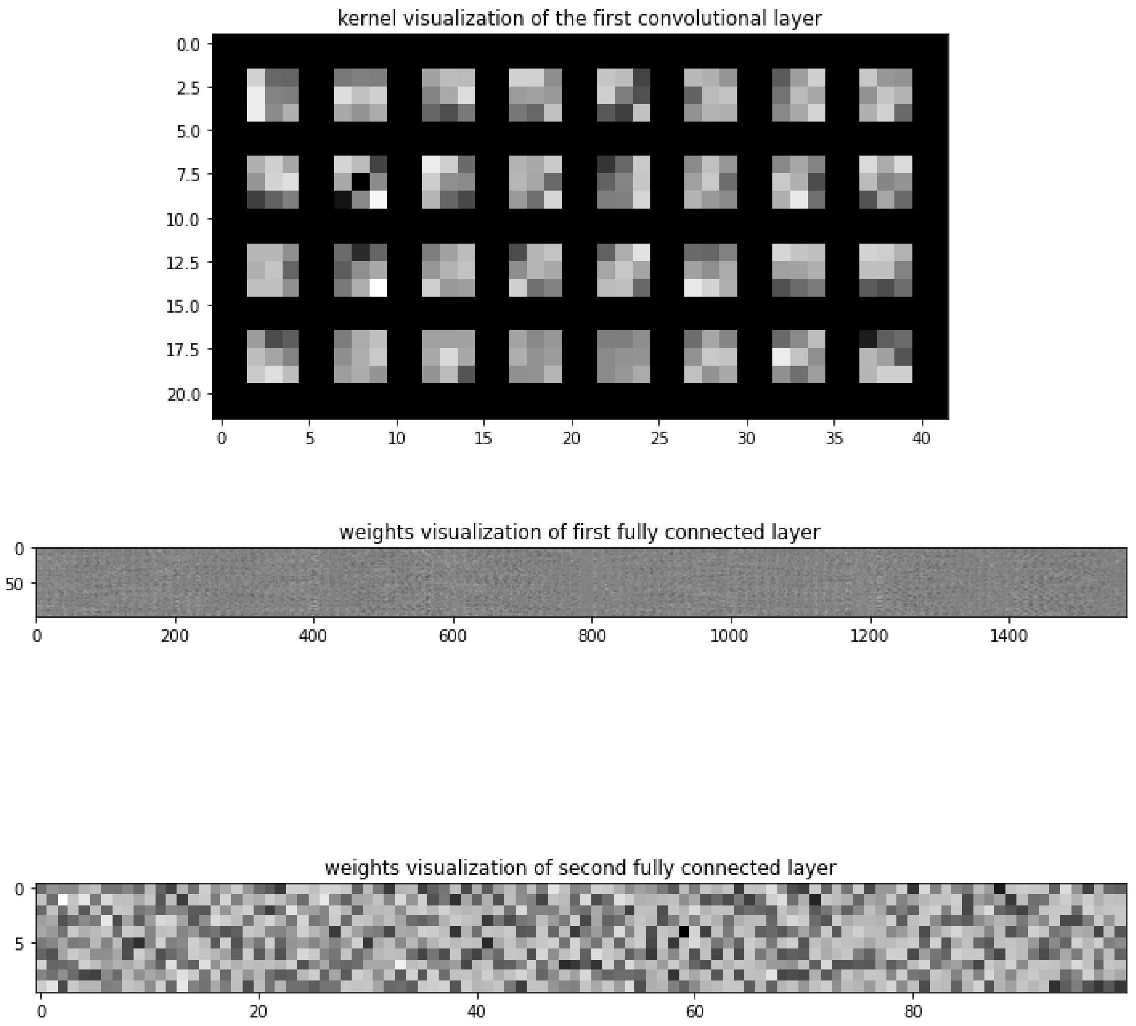


Figure 9: Visualization of the weights in the kernel of the first convolutional layer, and the weights in the fully connected layers.