Supervised Deep Learning

Assignment 1

Andrea Nicolai - 1233407

31/08/2021

In this homework we implemented and tested simple Neural Networks to solve supervised learning tasks. The two tasks tackled involved Regression, where we needed to approximate some simple function, and and Classification, where our network needed to correctly classify images from the MNIST digits dataset. In order to find the best model with best regularisers and optimizers, a hyperparameter optimization was made using random search.

Introduction

In this homework we were asked to solve two tasks that are related to Supervised Learning.
The first one is Regression, where we need to approximate some function $f : \mathbb{R} \to \mathbb{R}$, given some input $x \to y = f(x)$, in such way that the $network(x) \approx f(x)$. The network implemented is a is a Fully Connected Network (FCN) with two hidden layers, and during training is fed using noisy measures coming from the target function $\tilde{y} = f(x) + \text{noise}$

The second task instead involved Multiclass Classification, where we need to correctly classify images of MNIST handwritten digits dataset, whose labels are $\{0, 1..., 9\}$. In this case, the network slightly changes architecture and therefore name: the Convolutional Neural Network (CNN) implemented started with two convolutional layers, followed by two fully connected layers as in the previous case.

Finally, using random search, the hyperparameters such as regularisers, optimizers, optimal number of hidden units were tuned in both networks in order to implement the best model. This was pursued using K-Fold CV for regression task, due to the limited amount of training data, whereas a simple CV was used for the classification task, at the end computing the final accuracy. In addition to this, some plots were produced to better inspect the "inside" of the network and spot eventual pathological behaviors, such as too large weights.

1 Regression

As already stated the training points for this homework were few, namely 100. Moreover, at a further inspection one could see that some intervals $x \in [-3, -1]$, $x \in [2, 3]$ had no training points at all. In such way, during the final test, we were able also to evaluate model's generalization properties and eventually understand whether the model could predict the images $f(x)$ of the function, where x being "out-of-domain" points.

1.1 Methods

We will now state the Feed Forward Neural Network architecture implemented, which takes the form, using PyTorch "formalism":

1. **Input Layer**: 1 unit

2. **First Linear Layer**: N_{h_1} units with *ReLu* activation function
3. **First Dropout Layer**: p_1 probability to drop connections
4. **Second Linear Layer**: N_{h_2} units with *ReLu* activation function
5. **Second Dropout Layer**: p_2 probability to drop connections
6. **Output Layer**: 1 unit

where the only constraint we had was about the number of units in the input and output layers, due to the nature of the problem. Moreover, the activation functions between layers were chosen to be *ReLU*s to avoid the eventual arising of gradient vanishing problem, as well as to to relief the computational effort wrt other choice of activation functions (e.g. sigmoid). As the loss function, we chose *MSELoss* and, finally a $L2$ regularization term was added too avoid network weights taking too large values.

In order to find the best hyperparameters values for the model, we implemented a CV-5-folds random search. The hyperparameters to be tuned were initially drawn according to some distribution, and finally sampled randomly at every iteration in order to return the average, over the 5 folds, validation loss for the actual model. The optimal model was selected to be the one that returned with the lowest validation loss, since one would expect that the one with this characteristic is the one reaching convergence faster, in addition to having the most accuracy.

The set of parameters taken into account for the random search, namely the ones where to sample from, were:

- $\mathbf{p_1}$: sampled *uniformly* $\in [0, 0.4]$
- $\mathbf{p_2}$: sampled *uniformly* $\in [0, 0.6]$
- **n_epochs** : [100, 200, 300, 500]
- $\mathbf{N_{h_1}}$: [20, 50, 75, 100]
- $\mathbf{N_{h_2}}$: [50, 100, 150]
- **optimizer** : ['adam', 'sgd', 'adagrad'], with *SGD* momentum taking default value of 0.9.
- **learning rate**: sampled according to a *loguniform* distribution $\in [1e-5, 1e-2]$
- **weight decay** : sampled according to a *loguniform* distribution $\in [1e-3, 1e0]$. One should note this parameter refers to nothing more than $L2$ regularization.

The number of random searches pursued was 500.

1.2 Results

Using such approach, the best set of hyperparameters resulted out to be (rounded to 5th decimal):

- $\mathbf{p_1} \rightarrow 0.00616$
- $\mathbf{p_2} \rightarrow 0.359936$
- **n_epochs** $\rightarrow$ 300
- $\mathbf{N_{h_1}} \rightarrow 20$
- $\mathbf{N_{h_2}} \rightarrow 150$
- **optimizer** $\rightarrow$ *adam*
- **learning rate** $\rightarrow$ 0.003911
- **weight decay** $\rightarrow$ 0.001472

After having found the optimal set, the network was trained using the full training set. While the performances of the network, with predicted outputs compared with true labels is depicted in Fig. 1. Briefly discussing about network performances, it clearly can approximate the function in the range where training points are present, though failing in predicting the function behavior in the ranges $x \in [-3, -1]$, $x \in [2, 3]$. Indeed, it simply connects the two points with straight lines, which is clearly not the correct solution. Finally, around -4 one can see as the network makes some sort of overfitting, since it predicts a curve which is not there in reality.

However, looking at Fig. 2, weights of the network do not exhibit any particular behavior and are in acceptable ranges, given that they are not exploding. This effect is given by $L2$ regularization.

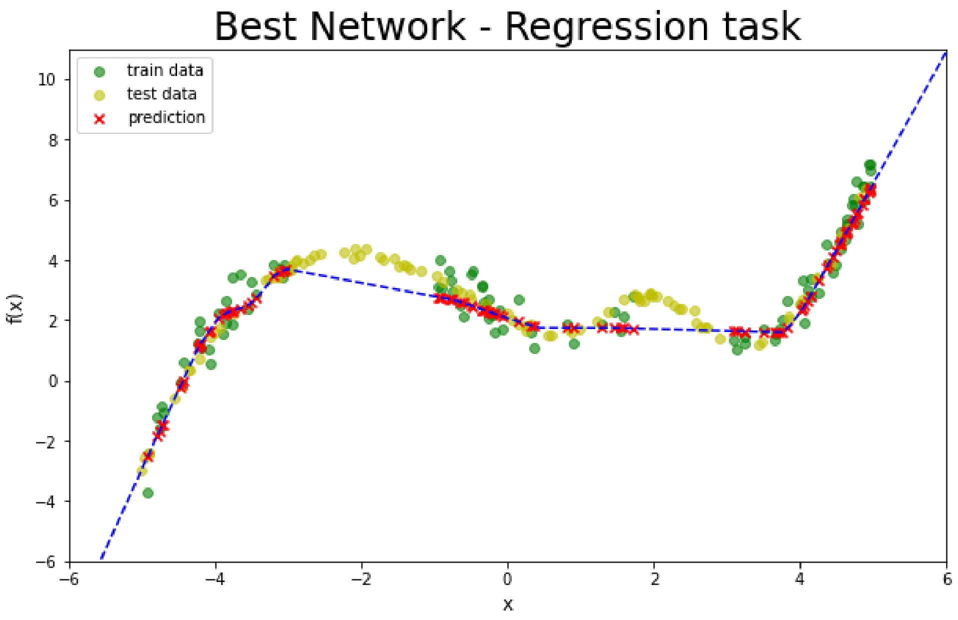


Figure (1) – Predictions of the network trained with optimal set of parameters for the Regression task.

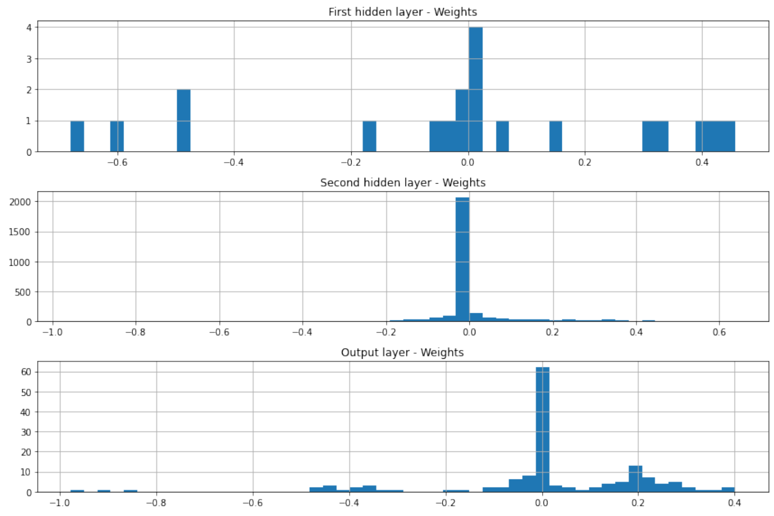


Figure (2) – Network weights histogram, effects of regularization can be clearly seen.

Finally, taking instead a look at the activation profiles of Figg. 3 and 4 one can see as the input layer has only few nodes that activate: this might be an effect of sparseness introduced by the use of *ReLU* activation functions. On the other hand, this might turn into a problem: if we wanted to relieve this effect, one could have used *LeakyReLU* activation function thus avoiding the presence of too many inactive units.

2 Classification

For the second part of the assignment we were asked to train a neural network mapping an input, grey-scale 28×28 pixels image, to 10 classes, namely being able to correctly classify handwritten digits.

2.1 Methods

The overall training set consisted of 60'000 samples, while the test one of 10'000. Therefore, in order to better train and evaluate model performances, the training set was split into the actual train and validation sets, having respectively 80% and 20% of the initial number of samples devoted to training

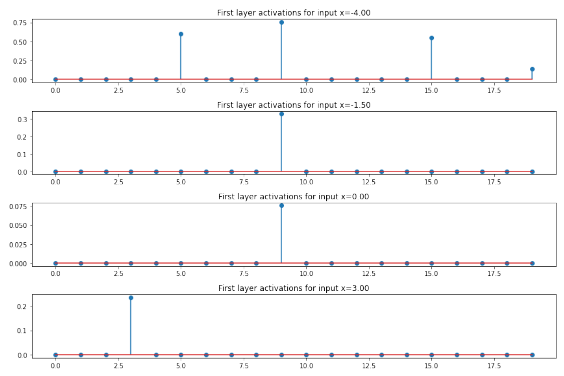


Figure (3) – First layer activations for different inputs. One should take care that scale over y axis is different.

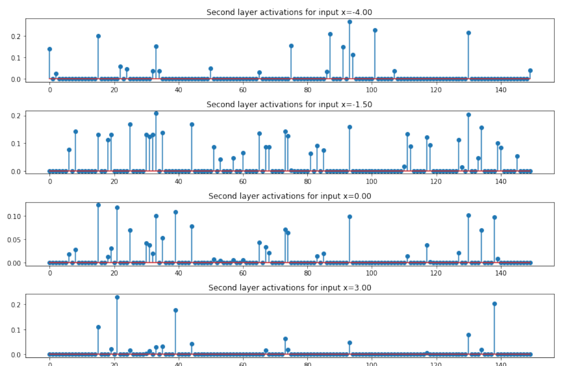


Figure (4) – Second layer activations for different inputs. Here one can see as there are neurons that always fire regardless the input, as well as some others do not.

(48'000 and 12'000). Due to the larger number of training samples rather than before, it was chosen to not perform k-fold-CV.

When data was imported we decided, in addition to converting the image to PyTorch tensor (via *ToTensor()*), to normalise the datasets providing its mean and standard deviation via *Normalize((0.1307,), (0.3081,))()*.
Moreover, with the aim to improve the performance of the model, the training data after normalization was finally augmented either adding some Gaussian noise or applying a random rotation of 45 degrees. This should result into a better generalization property for the network.

The network architecture implemented was the following:

1. **First Convolutional Layer**: with 1 channel in input, being the image in grey scale, n channels as output, kernel 4×4, *padding* 1 and *stride* 2, in order to reduce the dimensionality of the image to 14×14. The activation function was a ReLU.
2. **Second Convolutional Layer**: with n channels in input, $2 \cdot n$ channels as output, kernel 5×5, and *stride* 2, in order to reduce the dimensionality of the image to 5×5. The activation function chosen was the ReLU.
3. **Flatten Dense layer**: consisting of 128 units. The activation function was the ReLU.
4. **Dropout Layer**: connections were dropped with p probability
5. **Output Layer**: consisting of 10 units

Instead of using any *Pooling* layer, it was chosen to exploit *stride* parameters proper of Convolutional layer in order to reduce the dimensionality of the problem, thus diminishing the computational effort needed for the training. As before, the number of output units was constrained due to the nature of the task. The loss used was the *CrossEntropyLoss*, moreover since the training took much more time rather then the previous task, it was implemented a *EarlyStopping* class which eventually stopped the network in the case some metrics (e.g. validation loss) was not improving for a number of iterations more than the *patience* parameter.

Finally, the set of hyperparameters where to sample from was the following:

- **n channels** : sampled from $[2, 4, 8, 16]$
- **p** : sampled *uniformly* $\in [0, 0.5]$
- **optimizer** : ['sgd', 'adagrad', 'adam']
- **learning rate**: sampled according to a *loguniform* distribution $\in [1e-4, 1e-2]$
- **weight decay** : sampled according to a *loguniform* distribution $\in [1e-3, 1e-1]$. One should note that using this parameter is equivalent to perform $L2$ regularization.

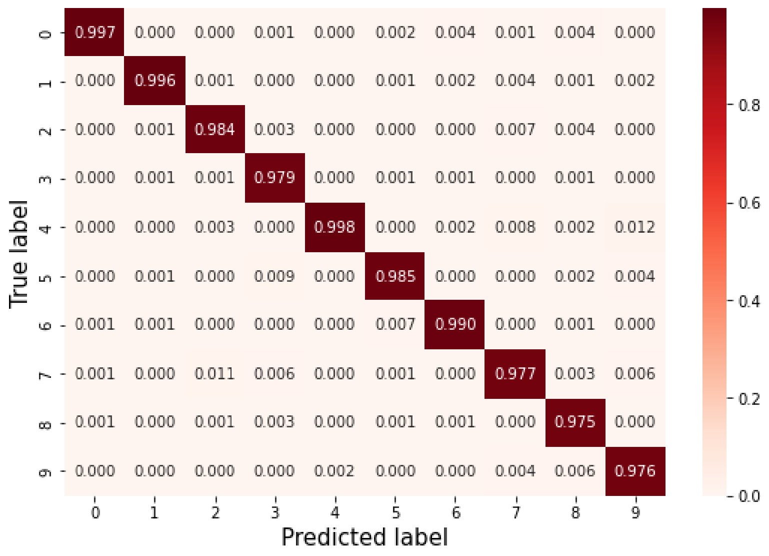
With referral to the previous task, we decided to use a less large number of parameters to be optimized due to the already high computational effort to train such "simple" network. Adding more dimensionality to the hyperparameters space would have resulted most likely too inefficient. Moreover, we defined the optimal set as the one according to which the model had the highest accuracy (i.e. minimum loss), namely the one with largest number of correctly classified samples in the validation set, reached during the random search.

2.2 Results

After performing the random search for the optimal set of hyperparameters for 40 iterations, the best model was:

- **n channels** → 16
- **p** → 0.028178
- **optimizer** → 'adam'
- **learning rate** → 0.000881
- **weight decay** → 0.002693

With the performance which is depicted in the confusion matrix in Fig. 5. The model is indeed able to classify well digits with an overall accuracy of 98.6%, which is still a good result.



True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	0.997	0.000	0.000	0.001	0.000	0.002	0.004	0.001	0.004	0.000
1	0.000	0.996	0.001	0.000	0.000	0.001	0.002	0.004	0.001	0.002
2	0.000	0.001	0.984	0.003	0.000	0.000	0.000	0.007	0.004	0.000
3	0.000	0.001	0.001	0.979	0.000	0.001	0.001	0.000	0.001	0.000
4	0.000	0.000	0.003	0.000	0.998	0.000	0.002	0.008	0.002	0.012
5	0.000	0.001	0.000	0.009	0.000	0.985	0.000	0.000	0.002	0.004
6	0.001	0.001	0.000	0.000	0.000	0.007	0.990	0.000	0.001	0.000
7	0.001	0.000	0.011	0.006	0.000	0.001	0.000	0.977	0.003	0.006
8	0.001	0.000	0.001	0.003	0.000	0.001	0.001	0.000	0.975	0.000
9	0.000	0.000	0.000	0.000	0.002	0.000	0.000	0.004	0.006	0.976

Figure (5) – Normalized confusion matrix which returns the test accuracy of the best model. The average test accuracy over all samples is 98.6%.

If one inspects the filters of the two convolutional layers in Figg. 6 and 7, we can see as the feature maps act really different from one another. The first convolutional layer (Fig. 6) mainly focuses on specific portions of the input image, trying to capture an overall behavior. This is well seen also in Fig. 8, where what is the input image can still be told without effort.

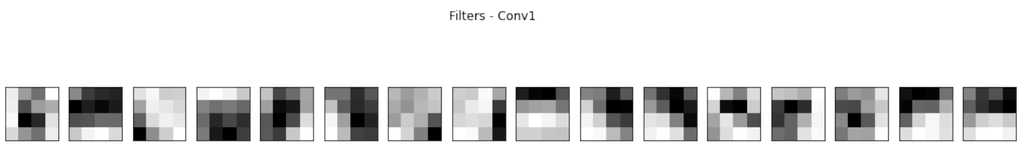


Figure (6) – Filters of the first convolutional layer. They are mainly focused on specific portions of the input image. Number of filters are 16, as expected, being the optimal number of channels 16, and their dimensions is 4 × 4.

On the other hand, the second convolutional layer is able to learn more abstract features (see Fig. 7), and according to the filter weights it seems like it is acting as an edge detector with the edges having different shapes and orientations. Activations, now, do not help us in understanding what might be input image any more, as one can see from Fig. 9.

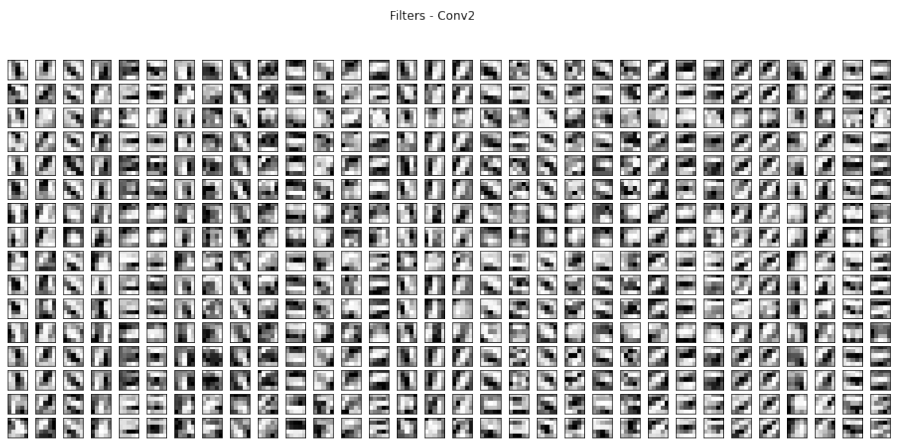


Figure (7) – Filters of the second convolutional layer. One can see as they act like *edge detectors* with different orientations. The number of filters are now $16 \times (2 \times 16) = 16 \times 32$ as expected.

Moreover, taking a look at the weights of the network of the fully connected and output layer, we cannot spot any weird behavior. Indeed they are in acceptable ranges and well distributed, thus showing us that the network was properly trained and might not be suffering of overfitting.

Conclusions

In conclusion, one can tell that the both networks were able to solve sufficiently good the task they had been implemented for. For this purpose, also random search was implemented in order to find the model's best hyperparameters.

In particular, for the regression task, the target function was well approximated in the intervals where training points were present without seeming to fit (too much) the noise. However, in the interval where points were lacking, the network presented some criticalities since it was simply connecting the interval bounds with a line. This is a typical behavior of ReLU activation functions, whose output is not as smooth as would be the one of a *tanh* or *sigmoid.* Thus, the performances of the network seem to be acceptable, despite the lack of training point in some plays a crucial role and some overfitting is present.

On the other hand, for the classification task, the network implemented was able to correctly classify 98.6% of the handwritten digits. One should note that the testing was performed only using the original test set, despite the network was trained using noisy images to enhance its generalization properties: it would have been interesting to see how the network would have behaved also in presence of some disturbs during the "testing" phase. Moreover, one could have added also activation function to the output layer, thus not simply taking the class with the maximal score. In this way, using for example a *LogSoftMax* function, we would have been working with class probabilities. Despite these being too much complex technicalities, the results using a simpler approach were still satisfying to us.

Appendix

This section contains some additional figures, not fundamental for the drafting of the report, nevertheless worth to be showed.

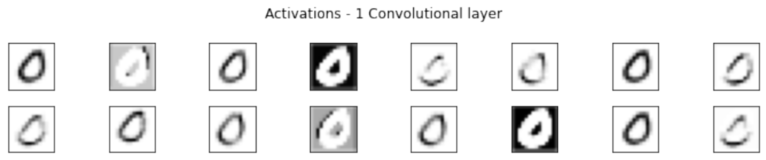


Figure (8) – Activations profile of first convolutional layer of the network for a certain input. Here, it still can be told the input image is most likely a zero.

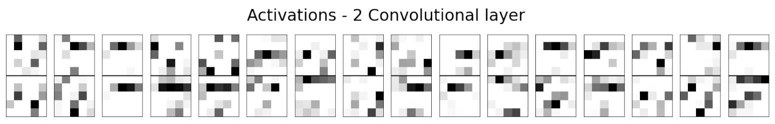


Figure (9) – Activations of second convolutional layer of the network for a certain input. We are not able to tell what might be the input any more.

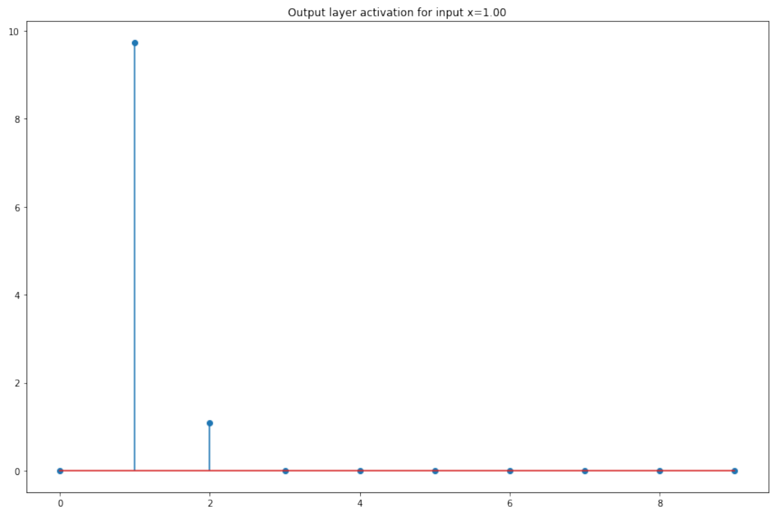


Figure (10) – Activations of the output layer of the network devoted to multiclassification task. In this case, the input label was 1, which is correctly classified.