# Homework 1 - Report

Fabio Vaccaro

30 Novembre 2020

# 1 Regression

## 1.1 Introduction

The goal of this section is to implement a simple neural network to solve a supervised problem, a regression task. We should train a neural network to approximate an unknown function:

$$f : \mathbb{R} \to \mathbb{R}$$

$$x \to y = f(x)$$

$$\text{network}(x) \approx f(x)$$

As input we have training samples, noisy measures of the target function:

$$\hat{y} = f(x) + noise$$

For the first test I decided to implement a simple train and validation split from the training dataset. The samples are few, actually 100 for the training dataset and other 100 for the test dataset. I tried 20 and 25% for the validation and the remaining for training, 25 was the best number during tuning phase. Training, validation and test points graphs can be found in the appendix (Figure 1, 2, 3). I also thought to use k-fold cross validation approach (used in the homework 2 for completeness) but I realized that it was not compatible with the plan of using a Bayesian Optimization strategy for hyper-parameters optimization. In fact, following this approach with many hyper-parameters, as intended, with high number of epochs, would require too many training loops, considering the cost and time limitations. In practice adding cross-validation to hyper-parameters tuning would increase the required time by a k factor. This led me to choose between a simple optimization strategy (maybe a grid search with few hyper-parameters and few hyper-parameters values) with cross-validation or an advanced optimization tuning strategy without any cross-validation. The choice fell on the more advanced optimization strategy as I was curious to learn and experiment with a new, never used library, instead of using only a loop and KFold sklearn function.

## 1.2 Method

### 1.2.1 Network architecture

The first network structure tried was:

```
Net(
  (fc1): Linear(in_features=1, out_features=config['Nh1'], bias=True)
  (fc2): Linear(in_features=config['Nh1'], out_features=config['Nh1']*2, bias=True)
  (out): Linear(in_features=config['Nh1']*2, out_features=1, bias=True)
  (dropout): Dropout(p=config['dropout_p'], inplace=False)
  (act): Sigmoid()
)
```

where the config parameter Nh1 and dropout_p are changed during the optimization phase. The network is made of two fully-connected hidden layers. The first layer's width is controlled by the Nh1 hyper-parameter and the second layer's width is double the width of the first one. The ration between the neurons in the first hidden layer and the second was not optimal as suggested by the analysis of the activation profiles, so I decided to make also the second layer width a hyper-parameter for optimization, becoming:

```
Net(
  (fc1): Linear(in_features=1, out_features=config['Nh1'], bias=True)
  (fc2): Linear(in_features=config['Nh1'], out_features=config['Nh2'], bias=True)
  (out): Linear(in_features=config['Nh2'], out_features=1, bias=True)
  (dropout): Dropout(p=config['dropout_p'], inplace=False)
  (act): Sigmoid()
)
```

The loss function is the mean square error, the optimizer used is ADAM. Also AdaGrad was tested during Grid Search but was performing less then ADAM so I decided to continue the tuning only with ADAM. As regularization methods I chose to use dropout for the two hidden layers. Other types of regularization were used in the classification problem to diversify the usage.

### 1.2.2 Hyper-parameters

For hyper-parameters optimization I decided to implement the Ray Tune library[1], integrating it into the train function to let it change the hyper-parameters and get the report of the validation and training loss. I first used a Grid-Search approach to get an idea on the right intervals for the hyper-parameters. Then I implemented the AxSearch Search Algorithm, a Bayesian Optimization algorithm[2] to tune the hyper-parameters in a finer way.

- *Hidden layers width.* Utilizing the first NN structure, during activation profiles analysis I found out that maybe 512 neurons for the second

[1] Ray Tune website: https://docs.ray.io/en/master/tune/index.html
[2] Ax website: https://ax.dev/

hidden layer is too much and I tried the second network architecture, allowing the tuning of both the first and second layers width. The choice was among the following choices: 64, 128, 256 neurons.

- *Learning rate*. Maybe one of the most important hyper-parameter to carefully tune is the learning rate. The range was [0.0001, 0.01].
- *Dropout probability*. The probability of a neuron to be zeroed. The range was [0, 0.5], including the 0 we were able to test the model without any form of dropout.
- *Training batch size*. Thanks to the small number of samples in the training dataset we can have a very small training batch: [4, 40].

Running 400 trials for a total of about 17 hours gave me the following results:

| | |
|---|---|
| First hidden layer width | 128 |
| Second hidden layer width | 64 |
| Learning rate | 0.0009836242556571962 |
| Dropout probability | 0.01674123853445053 |
| Training batch size | 10 |

Considering the very low value of dropout probability I decided to investigate and I found out that dropout does not help in this case, so I decided to set manually to 0.

## 1.3 Results

### 1.3.1 Train and validation loss

Train and validation loss are plotted in Figure 4. We can infer that after 1500 epochs in this configuration the neural network stops gaining in the train loss.

### 1.3.2 Train and test loss

I have then trained the model with the hyper-parameters found during optimization on the all train dataset (joining the previous train and validation dataset). The result of the train loss are plotted in the Figure 5. I were able to get 0.070817 as far as test loss.

### 1.3.3 Weight histograms

In Figure 7 NN weights are plotted on histograms divided on the different layers. I observe that the majority of weights are pretty small. A big module of weights could indicate overfitting, so in this case we are not in this situation. In the first layer we are in a neighborhood of about -0.4 and +0.4, in the second layer in a neighborhood of 0 and in the third one most of them are distributed between -1 and +1.

### 1.3.4 Activation profiles

I have chosen three values to analyze the activation profiles picking them from the left side, the center and the right side of the function. Only the neurons with values higher than zero are contributing to the output. In the first layer all neurons seem to play a role in the output, meaning that we can not have a smaller first layer with this configuration. In the second layer we have many zero activations, so many of neurons are not contributing to the output. This is suggesting that probably the network is too complex with regard to this layer (at the same time the smaller size proved during Grid-Search, 32 neurons, probably was too small. For this reason we got better result with 64. Probably an intermediate value would be the best).

# 2 Classification

## 2.1 Introduction

The goal of this section is to implement a neural network to solve a classification task, a simple image recognition problem. We have to classify images of handwritten digits from the MNIST dataset. To solve this task we can use two approaches: using a fully connected neural network or using a convolutional neural network. As the second one is specifically designed for image recognition tasks I expected to get better results with that one in respect to the first implementation. Moreover the fully connected neural network can be used in this scenario as we have very small images (28x28) translating to an input layer of 784 neurons. Using even slightly bigger images would translate into a very big first input layer.
For what concerns samples used, using a cross validation approach would require too much time and the benefit would not be considerable considering the large dataset in our possession. So I decided only to split the train dataset into train and validation.

I have applied data augmentation to the training dataset in a dynamic way. Every time the same sample is called from the dataset a different transformation is applied. In this way at every epoch, the neural network is able to learn on different images and gain accuracy. The transformations applied were: rotation (max 20 degrees), scaling (max 10% of the total width) and translation (max 5% of the total width).

## 2.2 Method

### 2.2.1 Network architecture

*Fully connected neural network.* The network is made of an input layer with 784 neurons (equal to the number of pixels in the image), a 128 neurons first hidden layer, a 64 neurons second hidden layer and a 10 neurons output layer (as we want to classify digits). The structure is as follows:

```
Net(
  (fc1): Linear(in_features=784, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=64, bias=True)
  (out): Linear(in_features=64, out_features=10, bias=True)
  (act): ReLU()
)
```

*Convolutional neural network.* I decided to focus on the convolutional neural network as this type of NN is more suited for image recognition tasks. Observations from now on are referred to the convolutional neural network. The CNN structure is as follows:

```
Net(
  (cnn_model): Sequential(
    (0): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU()
    (2): AvgPool2d(kernel_size=2, stride=2, padding=0)
    (3): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
    (4): ReLU()
    (5): AvgPool2d(kernel_size=2, stride=2, padding=0)
  )
  (fc_model): Sequential(
    (0): Linear(in_features=320, out_features=160, bias=True)
    (1): ReLU()
    (2): Linear(in_features=160, out_features=10, bias=True)
    (3): ReLU()
  )
)
```

The loss function used is the cross entropy loss, the optimizer is SGD.

### 2.2.2 Hyper-parameters

Also for this task I implemented Ray Tune for the hyper-parameters optimization. The hyper-parameters tuned are:

- *Learning rate*. The rate was set to be in the interval [0.0001, 0.01].
- *Momentum*. Using a momentum term helps accelerating gradient vectors in the right direction, improving convergence of SGD. We have to pick the right value, in the interval [0, 1].
- *Weight decay*. I decided to try also a regularization method using a L2 penalty. This is picked from this interval: [0, 0.1]
- *Train batch size*. Using a high value for this parameter would increase the training speed, but I realized during optimization phase that better results can be found with a small size. The interval chose was: [4, 400].

The tuning was done with 300 trials, less than regression as the number of samples and the computation load were higher. The optimization phase lasted about 10 hours and gave me the following results:

| | |
|---|---|
| Learning rate | 0.0076909336118989105 |
| Momentum | 0.6096049329165835 |
| Weight decay | 1.320447863976398e-18 |
| Training batch size | 4 |

## 2.3 Results

Testing the CNN on the test dataset we get a 99,46% accuracy, much more than about 97% of the fully connected neural network. I have then investigated the filters of the CNN of the first and second convolutional layer (Figure 10). To fully understand the functioning of the neural network we have to plot the feature maps, in some way we can say that we are analyzing what the NN sees. In the feature maps of the first convolutional layer (Figure 11) we can still clearly see the shape of the digit analyzed by the network. The feature maps of the second convolutional layer are more difficult to interpret, but this is explained by the small size and the fact that in deeper convolutional layers features extracted are more abstract.

# A Appendix

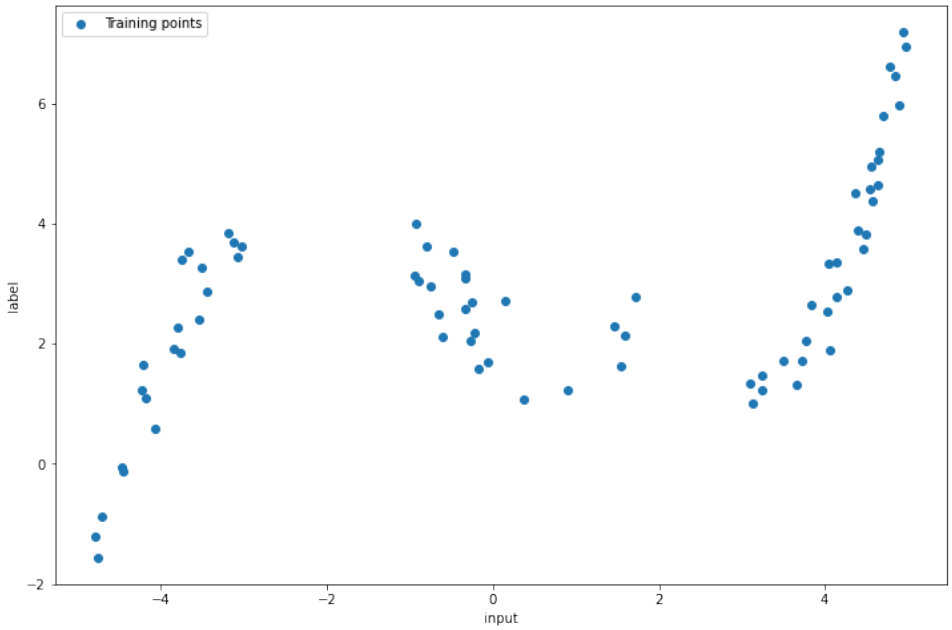


Figure 1: Training points

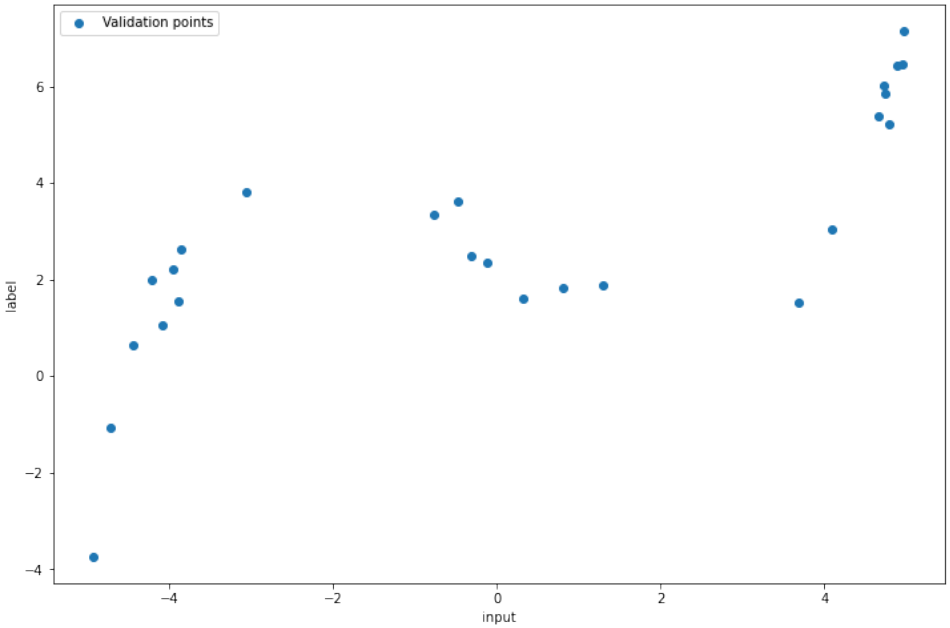


Figure 2: Validation points

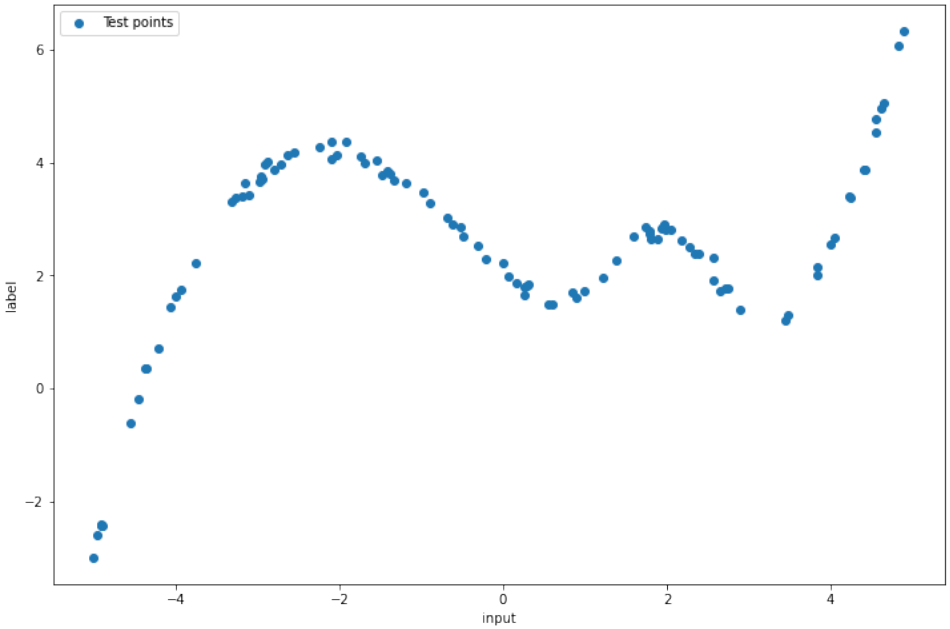


Figure 3: Test points

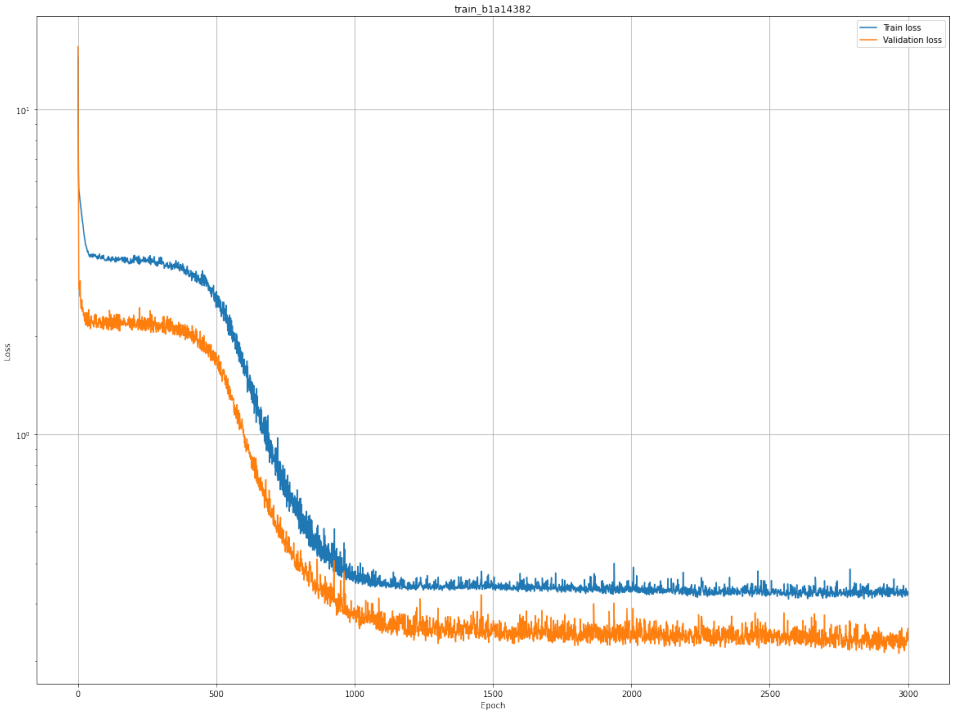


Figure 4: Train and validation loss

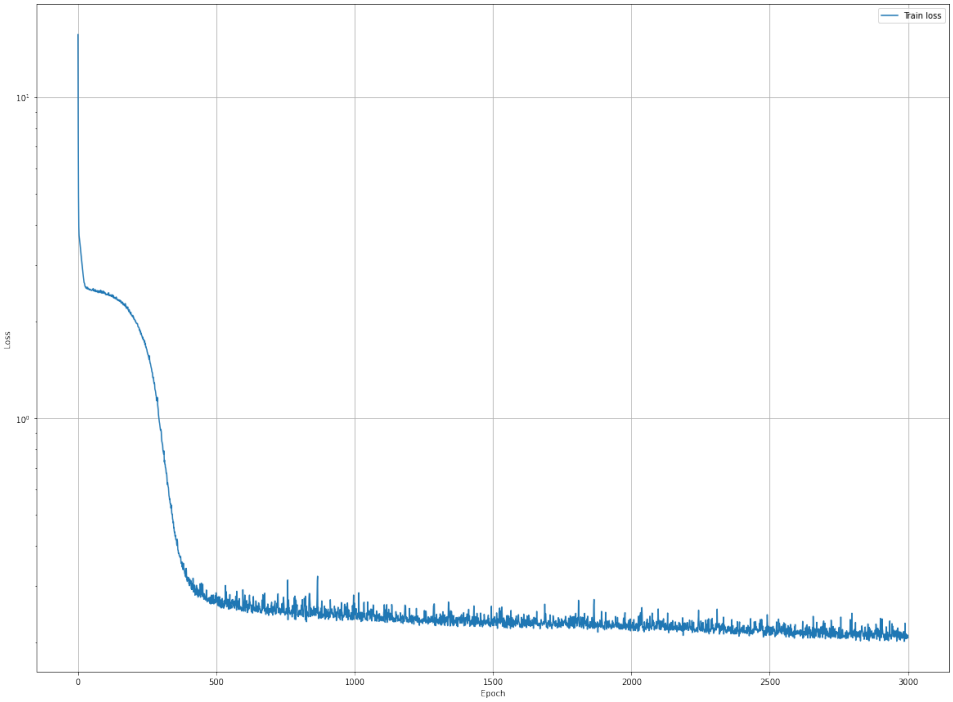


Figure 5: Train loss - Final model

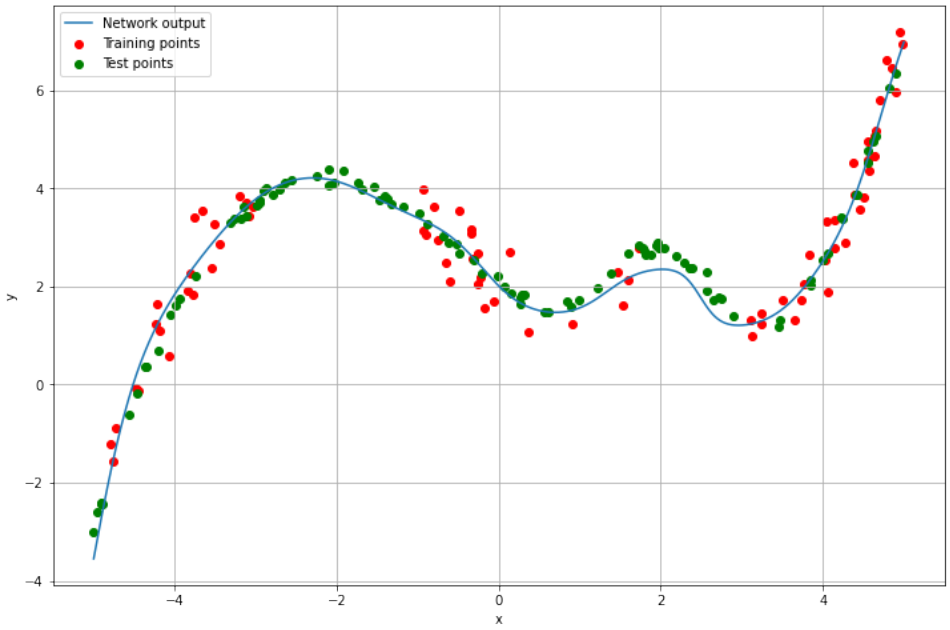


Figure 6: Final result

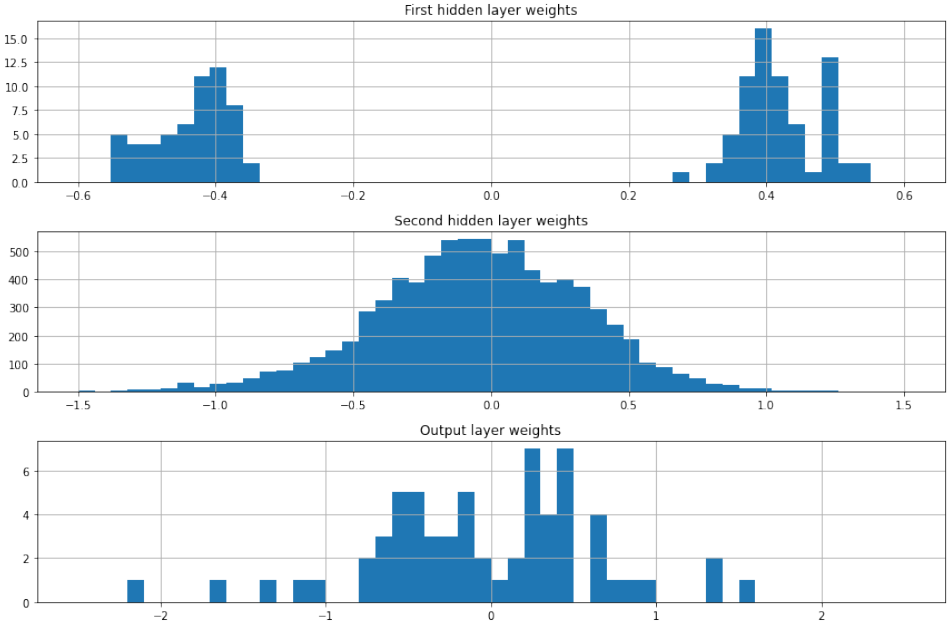


Figure 7: Weight histograms

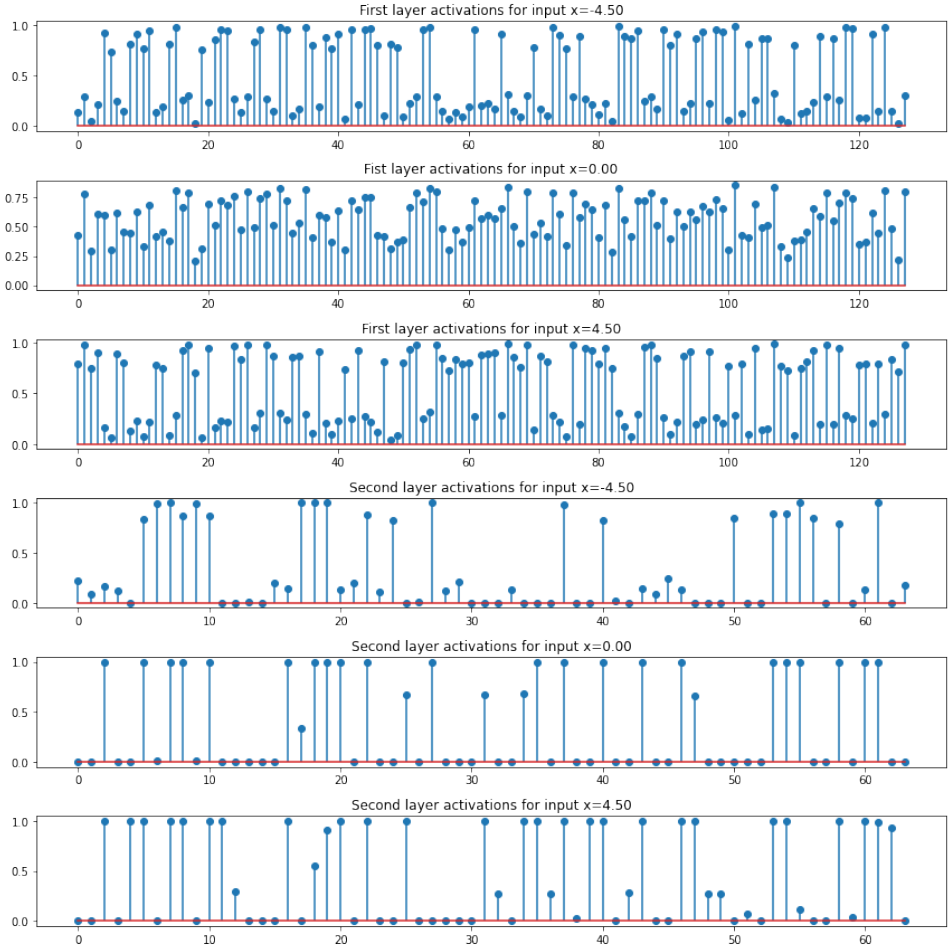


Figure 8: Activation profiles

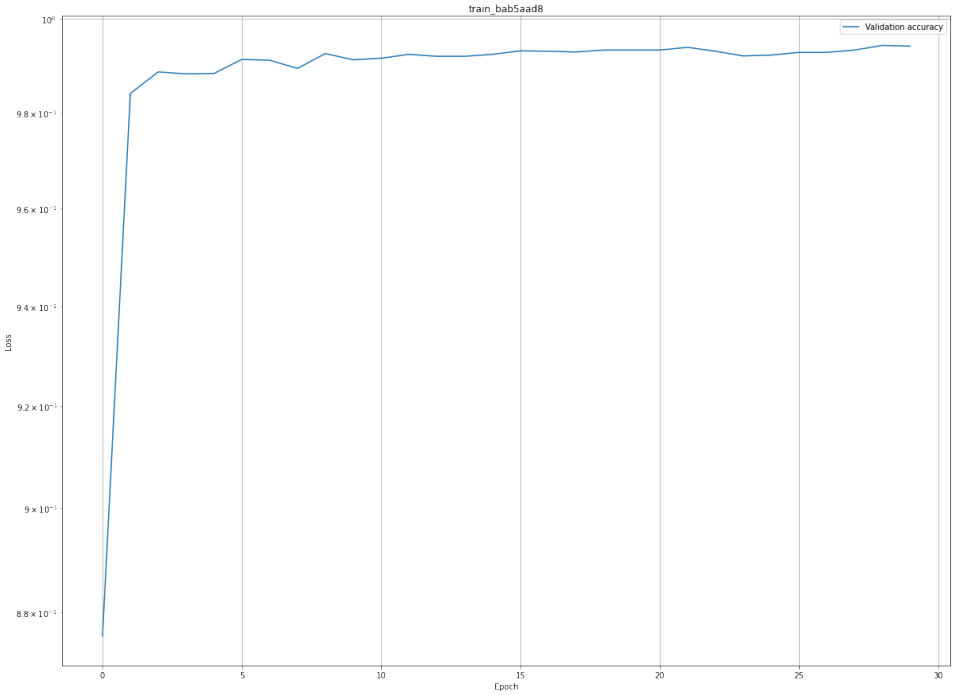


Figure 9: Validation accuracy


Figure 10: CNN filters

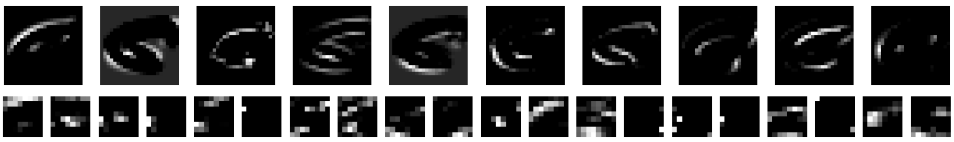

Figure 11: CNN feature maps