All source code is hosted here.

Problem 5.1 (Bagging)

a) The results obtained from the evaluation of the base learner are:

$$confusion\ matrix = \begin{bmatrix} 0.50367908 & 0.147262467 \\ 0.14738038 & 0.20167806 \end{bmatrix}$$

Test accuracy: 0.70537, *Training accuracy:* 1.0 (The train accuracy equal to 1 is given by the tipical over-fitting problem of decision trees).

b) Implementation is reported in the code.

c) The results obtained from the evaluation of the ensemble learner are:

$$confusion\ matrix = \begin{bmatrix} 0.55758373 & 0.16068011 \\ 0.09347573 & 0.18826042 \end{bmatrix}$$

Test Accuracy: 0.745844155844156 *Training accuracy:* 0.9821617
Thanks to bagging we have less (not too much) variance and so less over-fitting.

d) We have seen that decreasing the bag size until a certain number of samples, the test accuracy was approximately the same, meanwhile the training accuracy decreased. This is due to the fact that, decreasing the number of samples in a bag and drawing with replacement from the training set, the variance is decreased more than the case of the same sized bags of training set, because is more probable that bags are different each other and so the classifiers are less "correlated" each other thanks to the unstability of the decision trees.

Problem 5.2 (Stacking)

In the enseble learning mulitiple week learners are trained for the same problem, and then combine them to get better results. There are several ways to to enseble learning: Bagging, Boosting and Stacking.
In Bagging multiple similar (homogenous) model are trained independenty from eachother, and some averegagin process are used to get the final, and better result. The goal is to decrease the variance.
In Boosting multiple homogenoues week model are train sequentially in an adaptive way to decrease the bias. (and also keep the variance low)
Stacking is a bit different, because it combines different type of models

a) Unlike bagging and boosting, stacking uses different models (heterogenous week learners) To point of stacking is to explore different models for the same problem and instead of selecting the best, combine the results to get an even better one. The different types of models are capable to learn different parts of the problem and together they resulting a better performance.
These models can be anything, it depends on the problem. For instance: KNN classifier, neural network, decision trees (with different sizes also), a logistic regression etc.

b) Stacking combines the results of the week models by teaching a meta model. This final model is stacked on the top of the other models. The input of this model is the outputs of the others and the output is the final prediction. The meta model also requires learning.
The most important constraint is that the data that was used to train the week learners can not be used to train the meta model. For this reason we have to split the data into 2 folds. The week learners are trained with the first fold, and then they are used to make predictions for the second fold. The next step is to train the meta model on the second fold of data, using the predictions made by the week learners.
The biggest drawback of this method is that we only have the half of the data to train. To overcome this limitation we can use some kind of k-fold cross training/validation technic.

Problem 5.3 (Multilayer Neural Network and Backpropagation)

a) The code for this exercise is hosted here. The output of the script `test_gradient.py` is the following:

```
Fully connected layer (20 nodes, 20 x 11 weights)
Fully connected layer (10 nodes, 10 x 21 weights)
Fully connected layer (3 nodes, 3 x 11 weights)
Checking gradients up to 6 positions after decimal point...OK
```

The output of the script `train_sine.py` is the following.

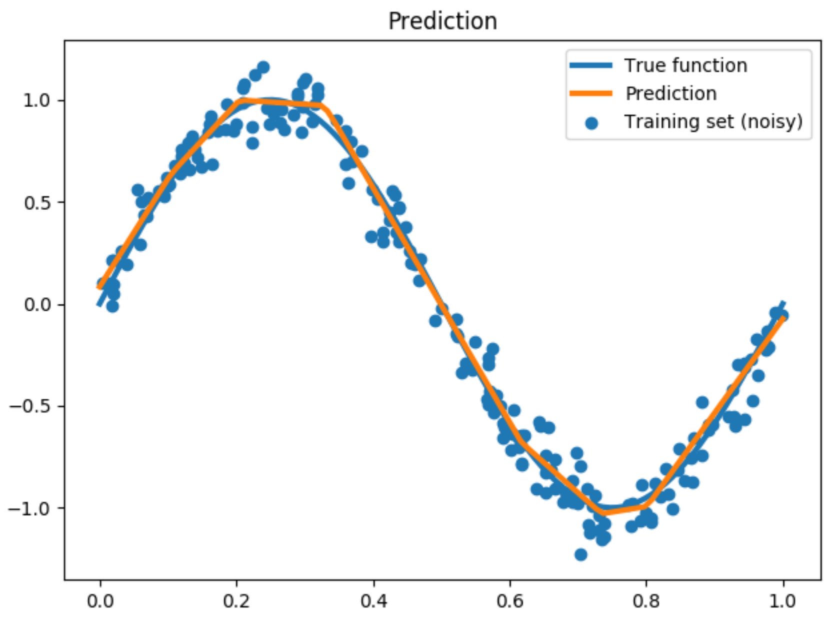

Abbildung 1: `train_sine.py` output

Problem 5.4 (Learning Inverse Dynamics)

a) The results for this exercise are the following:

```
Dimension 1: nMSE = 7.42 %
Dimension 2: nMSE = 10.10 %
Dimension 3: nMSE = 9.18 %
Dimension 4: nMSE = 5.13 %
Dimension 5: nMSE = 14.13 %
Dimension 6: nMSE = 28.24 %
Dimension 7: nMSE = 6.46 %
```

b) The results for this exercise are the following:

```
Dimension 1: nMSE = 4.81 %
Dimension 2: nMSE = 3.96 %
Dimension 3: nMSE = 3.42 %
Dimension 4: nMSE = 1.90 %
Dimension 5: nMSE = 6.37 %
Dimension 6: nMSE = 7.59 %
Dimension 7: nMSE = 2.95 %
```

c) Here is explained the approach to the problem. At the beginning we started searching information about neural network in general, and we found out that "Specifically, the universal approximation theorem states that a feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units"[1] and that "Since a single sufficiently large hidden layer is adequate for approximation of most functions, why would anyone ever use more? One reason hangs on the words "sufficiently large. Although a single hidden layer is optimal for some functions, there are others for which a single-hidden-layer-solution is very inefficient compared to solutions with more layers"'[2], so we tried to evaluate if it was better to use one or two hidden layers, using initial raw hyperparameters and a fixed nodes number (around 100). At the end of this very first step we decided to proceed with two hidden layers. We then have tried to find out the best distribution of 100 neurons over these two hidden layers (keeping fixed the others hyperparameters, that are still raw); we found that the best two options were 60%-40% or 40%-60%, with few trains (on the entire training set) we experimentally found that it was better to use the 40%-60% distribution (40 for the first layer and 60 for the second one). Then we increased the number of nodes, noticing that it was better to increase these, but at the same time it was time consuming to have a larger number of nodes, so we decided to keep the 100 neurons over the hidden layers before to proceed to the very expensive optimization. All these tests have been done by using a 10-fold cross validation over a model trained for 10 epochs. From now to optimize the hyperparameters we have proceeded using a Heuristic method (in particular the bayesian optimization). Before to go into the bayesian optimization we tried again randomly to restrict the range of the parameters, choosing each time values quite far one from the other. This is because the bayesian optimization require a long time to give good results, and we did not want to lose time evaluating values of hyperparameters that would have been useless. We found out that a great initial value of Alpha with a very rapid decay (such that in a couple of epochs goes to the minimum, that was around 5e-05) was a good starting point for the bayesian optimization step. The same reasoning was for Eta, and we noticed that a small eta with a rapid increase (around 20/30 epochs to reach the maximum around 0.9) was a good start value for the optimization step. From here we started with the bayesian optimization, the idea was to select (as done) a range of values for the hyperparameters and than wait for the response of the bayesian optimizator in order to refine the range for the next optimization step. The initial step of bayesian optimization was with a "large" (e.g. 0.06-0.1 for alpha) range of values and with only 20 epochs for each step of cross validation. From these first steps we found out better range (e.g. for the alpha hyperparameter the first steps pushed the value close to the 0.1); at this point we increased the number of nodes, proceding adding some nodes after nodes (as said before, at the beginning, we noticed that it was better to have more nodes, but the train and test was obiviously more expensive with more nodes). The sufficient large range of hyperparameters allowed us to increase the number of nodes almost without modifying the optimal parameters obtained by the Bayesian optimizer since now. So with these refined ranges (and a larger number of nodes) we proceeded with the next (and last) Bayesian optimization steps with more epochs (increasing step by step up to 100) but with less evaluations (around 10) these times. These last steps required some hours of simulations. The results we achieved after this process are the following:

```
Dimension 1: nMSE = 1.80 %
Dimension 2: nMSE = 2.28 %
Dimension 3: nMSE = 1.26 %
Dimension 4: nMSE = 0.47 %
Dimension 5: nMSE = 2.44 %
Dimension 6: nMSE = 3.13 %
Dimension 7: nMSE = 0.85 %
```

These results have been obtained using the following hyperparameters:

```
batch_size = 32
alpha = 0.0917196578307033
alpha_decay = 0.9796536618948516
min_alpha = 1.8228186808882324e-05
eta = 0.00046537234880058073
eta_inc = 0.0002883486118955981
max_eta =  0.982329770217266
layers = \
    [
        {
            "type": "fully_connected",
            "num_nodes": 156
        },
        {
            "type": "fully_connected",
            "num_nodes": 244
        }
    ]
```

Literatur

[1] Deep Learning, Ian Goodfellow, Yoshua Bengio, Aaron Courville, 2016

[2] Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks,by Russell Reed, Robert J. Marks, 1999