Machine Learning Project Report

Nicola Gugole - 619625 - n.gugole@studenti.unipi.it - MSc in AI
Elia Piccoli - 621332 - e.piccoli1@studenti.unipi.it - MSc in AI

Type of project: **A with CM**

ML course (654AA)
June 15, 2021
A.Y. 2020/2021

Abstract

This report presents the implementation of two Machine Learning approaches: an Artificial Neural Network and a Support Vector Regression model both developed from scratch. We propose as final model a combination of an ensemble of 5 Neural Network and a Multi-Output Regressor SVR where all constituent components are selected through a coarse to fine grid search using Hold Out validation. In the following sections we describe the key features of the project, involving the models validation and training schemas.

1 Introduction

This project is focused on developing different Machine Learning approaches to tackle a regression task on a provided dataset:

1. First implementation is an Artificial Neural Network developed using Python. The main aim in the development was to pursue a rigorous method to select the best models and validate their results. To find the best configuration a coarse-to-fine grid search was adopted using *Hold Out* as cross validation method. Furthermore, this first approach has also been tuned on the simple **Monk** classification task.

2. Second implementation follows an SVR-type approach including various different kernels. The Python implementation uses a dual approach algorithm with appropriate choices of the constraints, where

the Lagrangian Dual solution is given by an algorithm taken from the Deflected Subgradient class.

2 Method

The project has been developed entirely in Python, with the help of some libraries - **numpy, math, random** - and inspired by Keras for what concerns the interface. The main part of the code is exploited to define a nice and easy interface for a fully connected feed forward Neural Network. The model learns its parameters using classical Gradient Descent, and can be combined with different variations of the naive approach. In fact we implemented different features such as: momentum, L2 regularization, gradient clipping, different learning rate schedules and weights update policies. The models hyper-parameters can be easily fine-tuned in order to find the best configuration. We also realized a Grid Search interface to search over different architectures of the models. The same philosophy has been applied to the implementation of the SVR model, where the naive model has been enriched with the possibility of using different kernels and optimization parameters.

2.1 Neural Network Implementation

Starting from the building blocks, in the **Layer** class we can define the *number of nodes*, *activation function*, *weights* and *bias range*. The layer object will handle the output of the nodes and the backpropagation computation. The **Model** class has a simple interface *_add_layer* that can be used to define sequentially the various layers. They will be linked - computing input/output dimension - when the *_compile* method is called. During the compilation of the model we can also define the main hyperparameters, that will be analyzed in the section **2.2**, as well as the model task (*classification, regression*). The core method of the class is *_train*, where we can define the hyperparameters related to the training procedure, and then the model is trained. In order to do so in the training loop we will use other key methods such as: *_feed_foward* to compute the output given an input, *_compute_loss* computing the current error, *_back_propagation* that computes the backpropagation for each layer of the model, *_update_weights_bias* to update the model parameters. During the training process, that can be done in different fashion - **online, mini-batch, batch** - we perform *validation* provided at training time. We can also test the model using the *_infer* method. Thanks to this modular implementation the model is highly flexible, adaptive to different tasks and configurations of its hyperparameters.

2.2 Neural Network Parameters

Layer

- **nodes**: number of units in a layer
- **activation_function_type**: name of the activation function - *linear, tanh, sigmoid, relu, leaky_relu*
- **weights/bias range**: range for random initialization

Model

- **eta**: learning rate
- **loss_function**: name of the loss function
- **_lambda**: regularization coefficient
- **alpha**: momentum coefficient
- **stopping_eta**: minimum value for the learning rate that can be reached during decay
- **is_classification**: boolean value to indicate the task - used to get the correct evaluation metric
- **gradient_clipping**: boolean value to indicate if we need to apply clipping to the gradient given a threshold
- **batch_size**: define batch dimension - the integer value is used to represent *online, mini-batch, batch*
- **epoch**: number of training epochs
- **decay**: learning rate decay factor

2.3 SVR Implementation

The task chosen for the *Computational Mathematics for Learning and Data Analysis* course requires the implementation of an SVR model where a dual approach with appropriate choices of the constraints to be dualized is used. Moreover, the Lagrangian Dual is solved by an algorithm of the class of *deflected subgradient methods*. Given the constraints imposed by the formal definition of the model, to compute the correct value of the Lagrangian multipliers it is necessary to solve a *Convex Separable Knapsack Problem*. For further details on the mathematical formulation of the problem, the optimization algorithm and the convergence analysis we suggest looking at our report [1]. Moving to the code implementation of the model we can find the main components in the **SVR** class. In the constructor we can

define the main parameters that are: the *kernel type* and its parameters, the regularization coefficient C and the epsilon-tube radius ε. The *fit* method deals with the training of the model. It requires *input/output data* as well as the *optimization arguments* that will be used by the optimization algorithm. This procedure call will exploit different sub-methods: *solveDeflected* which computes the optimal Lagrangian multipliers, these values are then used by *compute_sv* to find the support vectors and - if the kernel type is linear - the slope value. Testing the model is possible by using the *predict* method.
Once again this implementation is highly modular, leading to a flexible model where it is easy to configure its parameters.

2.4 SVR Parameters

SVR

- **kernel**: name of the chosen kernel - *linear, radial basis function, polynomial, sigmoid*
- **kernel_args**: kernel parameters - *gamma, degree, coefficient*
- **box**: regularitazion coefficient C
- **eps**: epsilon-tube radius

Deflected Subgradient

- **x**: initial values of β - Lagrangian multipliers
- **maxiter**: maximum number of iterations
- **delta_res**: reset value for delta - delta $\rightarrow$ threshold of *target value*
- **rho**: discount factor for delta
- **eps**: minimum relative value for the displacement of delta
- **alpha**: deflection coefficient
- **psi**: discount factor for the stepsize

2.5 Validation Schema

Validation was treated similarly in the Neural Network and Support Vector Regression cases, with differences implied by the model intrinsic nature.
Starting with the Neural Network validation schema, first things first the dataset *ML-CUP20-TR* was divided into a **development set** and a **internal test set** - 80/20% respectively. The design set was exploited to perform model training and selection, while the latter to evaluate the chosen models generalization. The validation procedure we implemented is **Hold Out**. After the first partitioning of the entire dataset, the development set was further split into training and validation sets - 80/20% respectively.

The models were trained with the *training set* (TR), then the *validation set* (VL) was exploited to perform model selection and lastly the *test set* (TS) was used to estimate the generalization error of the final model. To rigorously search for the best models we adopted a two step grid search procedure, applying a **coarse grid search** to rapidly decrease the search space, followed by a **fine grid search** where various random perturbations of the best out-coming model are chosen as grid configurations. Eventually, in order to further decrease the generalization error and remove bias due to the use of a single model, we decided to apply **ensemble learning**.
Turning the attention to the Support Vector Regression validation schema, again we implemented a **Hold Out** validation procedure, with the same partitions splitting and use for each partition. Also in this case we applied a two step grid search to first decrease the search space and eventually refine the search in a more compact space (after perturbing several time the best selected model).
The main difference with respect to the Neural Network model stands in the fact that an SVR has an output space which is strictly one-dimensional, whereas a Neural Network can have as many output dimensions as needed. Since the ML-CUP dataset is defined having a two-dimensional output, we therefore needed two different grid searches, with each grid search working on one output dimension.

3 Experiments

3.1 Monk's Results

All inputs were transformed with a self-implemented **one hot encoding** as pre-processing procedure, leading to 17 binary inputs per pattern. Model's topology was kept simple, with all models consisting of 1 hidden layer of 4 neurons. For Monk 1 and Monk 2 the *tanh* was used as hidden layer activation function whereas for Monk 3 the *ReLU* was used, keeping for all three tasks the *Sigmoidal* in the output layer. Monk 3 results both with and without *L2 regularization* are shown to better study the effects of overfitting and regularization. Loss was computed using *Mean Square Error* function on a full batch gradient descent to avoid loss/accuracy oscillations in an already sensitive and little dataset. The sensitivity of the dataset showed itself in the weights initialization, leading to the need of shortening the range for the initial synaptic weights. Monk 1 and Monk 2 were therefore restricted between (-0.01, 0.01) and the more robust Monk 3 between (-0.5, 0.5). Scores in **Table 1** resulted from averaging 5 different trials.

Task	**Net Topology**	**eta**	**alpha**	**lambda**	**Loss(TR/TS)**	**Accuracy(TR/TS)**
MONK 1	(17,4,1)	0.05	0.90	0	0.000725/0.001040	100%/100%
MONK 2	(17,4,1)	0.04	0.90	0	0.000457/0.000536	100%/100%
MONK 3 (no reg)	(17,4,1)	0.035	0.99	0	0.000990/0.022707	100%/93.52%
MONK 3 (L2 reg)	(17,4,1)	0.035	0.99	0.01	0.032866/0.027733	93.48%/97.22%

Table 1: Avg loss and accuracy obtained on the Monk datasets

Figure 1 and **Figure 2** show loss and accuracy plots for Monk tasks, including the aforementioned regularization analysis for Monk 3. The qualitative analysis shows how the Monk 3 task loss diverges incurring in overfitting, a defect which can be cured with *L2 regularization*. The direct effect of regularization is a better model generalization to the cost of a decrease in the training set performance, a good trade-off which is definitely worth the cost.
Another interesting theoretical aspect we wanted to see in practice is the use of a *learning scheduler*: results are shown in **Figure 3**, where we forced Monk 2 to have a zigzag behaviour using a higher-than-needed *eta*. The exponential decay of the learning rate smooths out the learning curve, leading to a *slower but steadier* convergence.

3.2 Cup Results

Approaching the ML-CUP task we went through a **screening phase** to get a first glance at the search space. We tried architectural depths varying from two to five layers, noticing how having less than three layers leads to poor performances. Regarding the hyperparameters, *eta* and *learning rate decay* are the elements to focus on since a wrong choice makes the model learning unstable, susceptible to **gradient exploding** issues, which we solved implementing **gradient clipping**. We also found out that having a full batch implementation was the way to go, avoiding oscillations. This lead to a careful choice of the initial *eta* and *decay* parameters, preferring smaller values for a pleasing learning curve. Trying different activation functions we opted to the use of *Leaky ReLU* for better computational and learning performances.
As far as concerns Neural Network models we implemented a coarse to fine grid search. The coarse grid search parameters were inserted by hand, fixing some values following what has been observed in the screening phase. **Table 2** shows the chosen configurations with which all the permutations were created. Then we proceeded with fine tuning through a random search. In total

we tested around 2500 configurations requiring approximately 36 hours, a process which would have been much longer if it was not for our parallel implemented grid search. The utilized hardware consisted of an *8-Core Intel i9-9880H*.

Hyper-parameters	**Configuration Tested**
layers	{(8,8,8,8,2), (20,20,20,20,2), (16,16,16,2), (32,32,32,2), (16,16,16,16,16,2)}
eta	{1e-5, 9e-6, 5e-6, 1e-6}
alpha	{0.1, 0.8, 0.9}
batch_size	{*full_batch*}
epoch	{200}
lr_decay	{5e-6, 1e-6}
_lambda	{1e-3, 1e-4}

Table 2: *Hyperparameters* for Neural Network grid search

Moving to SVR models, a key observation is the fact that ML-CUP has two output features, leading to the need of a *Multi-Output Regressor* consisting of two independent SVR models. This guided our approach towards two separated grid searches - one for each dimension - both exploiting a sub-grid search over all implemented kernels. This methodology is applied in order to find the best model for the single output feature, eventually creating a final model that is obtained by coupling the best model for each dimension. In **Table 3** all grid search configurations are shown, divided by kernel.

Kernel	SVR Parameters	Optimization Parameters
Linear	box: 1/10, eps: 0.5/1	{eps : 1e-2, maxiter : 3e3}, {eps : 5e-4, maxiter : 3e3}, {eps : 1e-4, maxiter : 3e3}
RBF	box: 1, eps: 0.1, gamma: *scale*/*auto*/0.1/1/2/5	{eps : 1e-2, maxiter : 3e3}, {eps : 5e-4, maxiter : 3e3}
Sigmoid	box: 1, eps: 0.1, gamma: *scale*/*auto*/0.1/1/2/5	{eps : 1e-2, maxiter : 3e3}, {eps : 5e-4, maxiter : 3e3}
Poly	box: 1, eps: 0.1, gamma: 0.1, degree: 2/3/4/5/10	{eps : 1e-2, maxiter : 3e3}, {eps : 5e-4, maxiter : 3e3}

Table 3: *Hyperparameters* for SVR grid search - these same grid configurations have been used both for first and second dimensions

For both NN and SVR models, as analyzed in **2.5**, the selected validation schema is *Hold Out*, using an 80/20% split between development and internal test sets, further dividing the development set into training and validation using once again an 80/20% split.
Model selection was performed at the end of each grid search, returning the best performing model. All the models were evaluated and sorted increasingly based on the *MEE* metric computed on the validation set. For NN models a further and more clever evaluation was applied, still based on the *MEE* metric but also adding a **scoring function**, to penalize training/validation oscillations in the learning trend.

Our idea in finding the final model was trying to have a more unbiased architecture. This lead to defining the final model as the coupling of a NN ensemble and a Multi-Output SVR model. As far as concerns the NN ensemble, it is a methodology which helps decreasing the generalization error by basing the inference not only on a single configuration, but averaging instead multiple results coming from the ensemble constituents. In our case, the ensemble is constituted of the 5 best performing models coming out of the fine grid search. The ensemble models configurations, along with their training/validation MEE, are reported in **Table 4**.

Model	Training MEE	Validation MEE	Topology	Activation Function	Eta	Lr_decay	Alpha	Lambda
model 0	3.7534378	3.6634012	(20,20,20,20,2)	(leaky_relu, linear)	9.53e-06	5e-06	0.9	7.125e-05
model 1	3.7580695	3.6612348	(20,20,20,20,2)	(leaky_relu, linear)	9e-06	5e-06	0.834	7.124e-05
model 2	3.7634378	3.6755305	(20,20,20,20,2)	(leaky_relu, linear)	1.075e-05	5e-06	0.859	7.102e-05
model 3	3.7586945	3.6633073	(20,20,20,20,2)	(leaky_relu, linear)	1.062e-05	5e-06	0.723	7.124e-05
model 4	3.7582515	3.675782	(20,20,20,20,2)	(leaky_relu, linear)	1.042e-05	5e-06	0.859	7.243e-05

Table 4: *Hyperparameters* and *performances* of the five best models - ensemble - in term of MEE

Regarding the Multi-Output SVR model, *rbf* kernel resulted to outperform all other kernels, as can be seen in **Table A.1** and **Table A.2**. Eventually, the final SVR model couple was defined with the following parameters, see **2.4** for details:

- **Dimension 1:** SVR model with 1.0 C value, 0.1 epsilon-tube radius, *rbf* kernel with 0.1 gamma and the following optimization arguments *eps* : 0.08737368906085892, *maxiter* : 3000.0
- **Dimension 2:** SVR model with 1.0 C value, 0.1 epsilon-tube radius, *rbf* kernel with 0.1 gamma, and the following optimization arguments *eps* : 0.06803885259228548, *maxiter* : 3000.0

Interestingly enough, analyzing the different behaviours of kernels on this particular task, *rbf* kernel proved to be the most precise whereas the *poly* kernel tended to be more noisy in its predictions. *Sigmoid* and *linear* kernel proved to be the least performing, with *linear* being the *fanalino di coda*.
In the end, since using the validation error is not a perfect estimate for the generalization error, we tested each model generalization capabilities on previously unseen data by using an internal test set, rigorously kept apart from the model training and selection process.
Results are shown in **Table 5**. Plots for NN ensemble ans SVR prediction over the ground truth are shown in **Figure 4** and **Figure 5-8**.

4 Conclusions

The combined prediction of the ensemble and the SVR model were computed on the blind test by averaging the outputs from the constituent models - SVR and the already averaged ouputs coming out from the ensemble. The results were saved in the file **ff15_ML-CUP20-TS.csv**. We are confident in saying that the model will score a *MEE* greater than zero over the blind test.

MODEL	**Development Set MEE**	**Internal Test Set MEE**
model 0	3.7534378	3.6757293
model 1	3.7580695	3.6633072
model 2	3.7634378	3.6755305
model 3	3.7586945	3.6634373
model 4	3.7582515	3.6650032
ensemble	3.7583782	3.6686015
SVR	3.6586178	3.5291038
ensemble+SVR	3.708498	3.59885265

Table 5: MEE on the development set and internal set of the *ensemble* (singularly and together), *SVR* and *ensemble+SVR*

Acknowledgements

This experience turned out to be extremely interesting and useful to get a deep understanding of how neural networks work and how a different and historical model such as SVR differs from them. Developing both those models by ourselves truly gave a deeper insight into the different aspects seen during the course. The coding, the rigorous testing and experimenting stimulated our reasoning and growth as Machine Learning students, acknowledging how important this experience has been for us, building our knowledge to face future challenges and problems in this field.

We agree to the disclosure and publication of our names, and the results with preliminary and final ranking.

References

[1] E. Piccoli, N. Gugole. *Support Vector Regression using Deflected Subgradient Methods.* (`https://github.com/EliaPiccoli/ML-CM-Project/blob/main/CMReport/deflected_convergence_analysis.pdf`).

Appendix

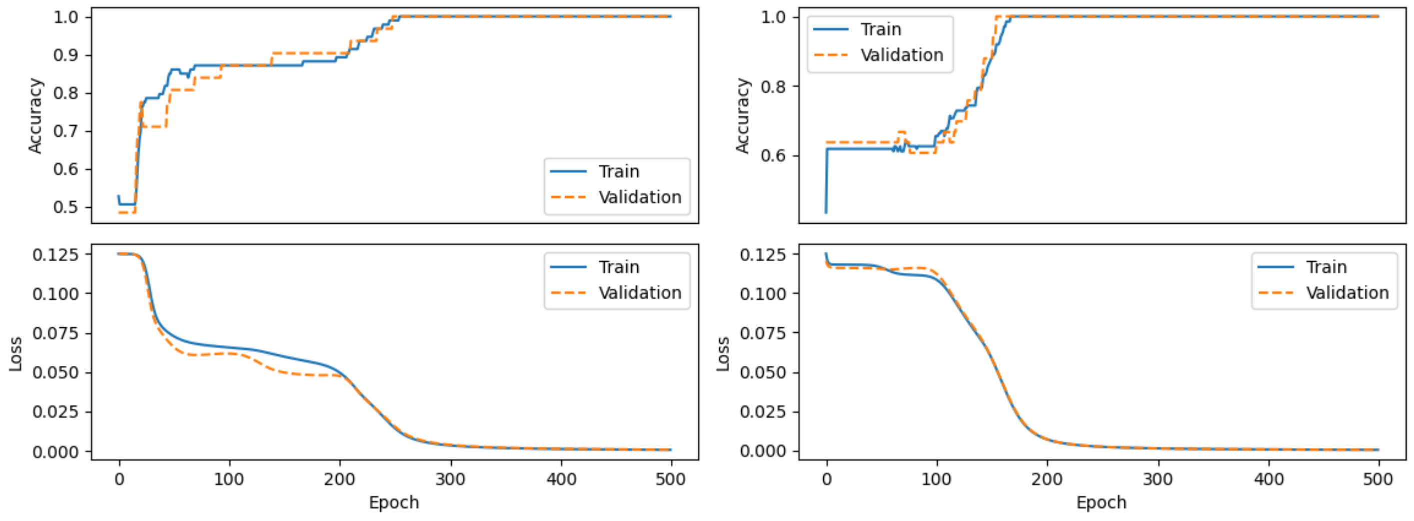

Figure 1: Monk 1 (left) and Monk 2 (right)

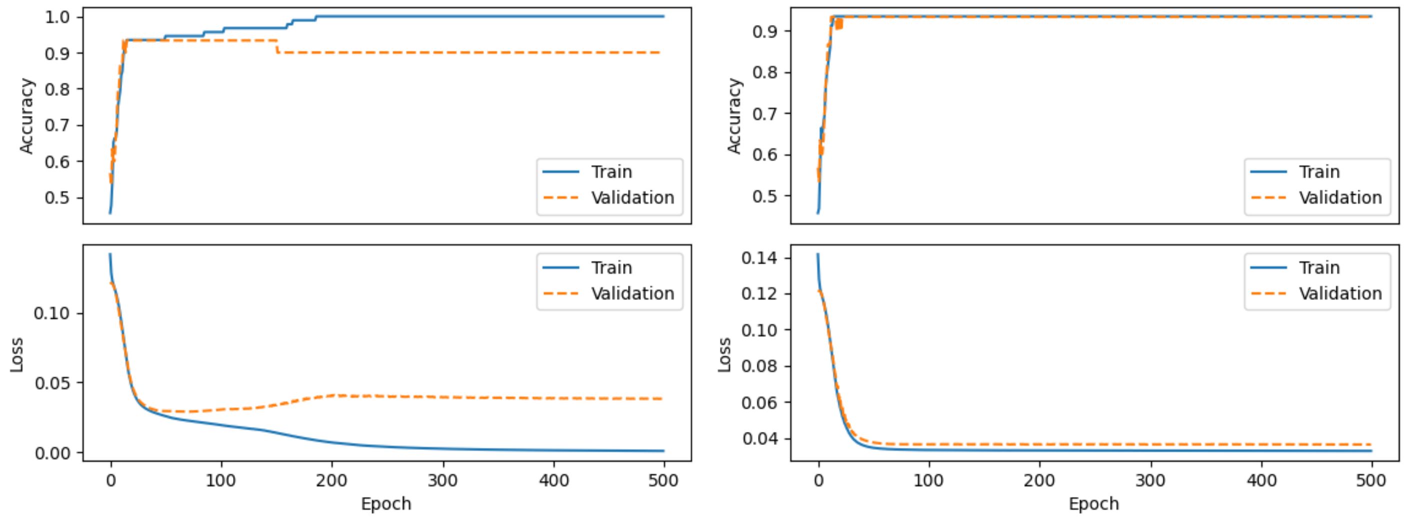

Figure 2: Monk 3 without (left) and with *L2 regularization* (right)

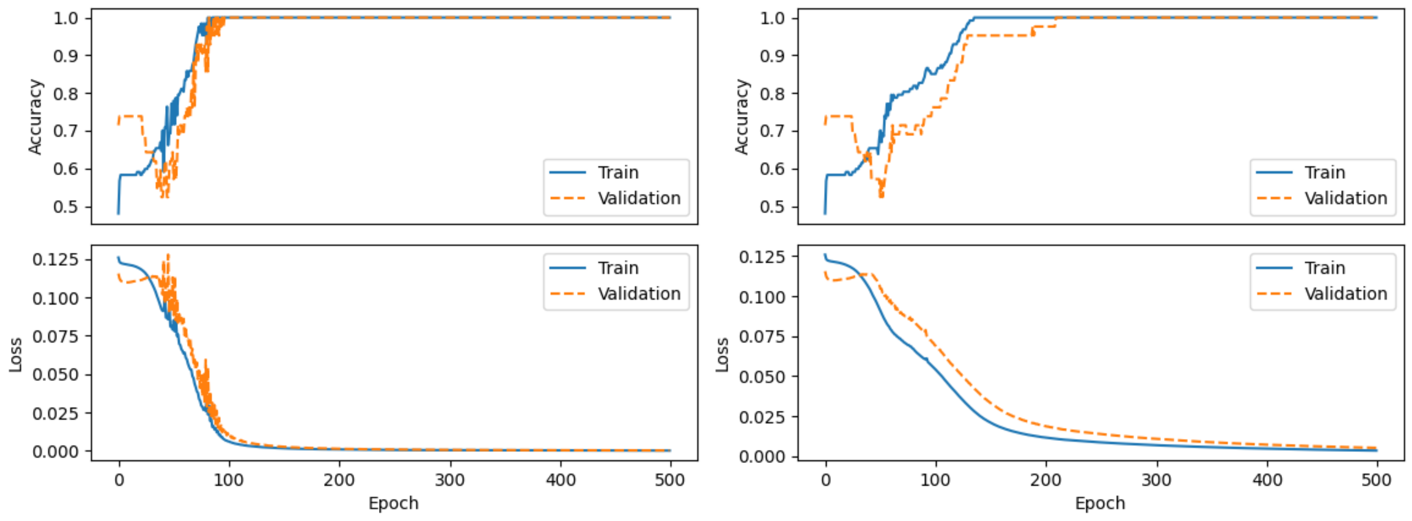

Figure 3: Monk 2 particular case with and without *exponential decay*

kernel	Output1 Evaluation	Output1 Eps-Insensitive Loss	Output2 Evaluation	Output2 Eps-Insensitive Loss
linear	12.0022842	178744.7206688	7.3719217	70578.4152311
rbf	2.2861038	8604.5173578	2.3588791	9639.4944259
poly	3.5201186	20735.5328493	3.2552161	18231.598317
sigmoid	7.5866864	85700.9586803	5.978907	49662.5428402

Table A. 1: Training results for **best** SVR model per kernel divided per dimension

kernel	Output1 Evaluation	Output1 Eps-Insensitive Loss	Output2 Evaluation	Output2 Eps-Insensitive Loss
linear	12.7620909	62047.5765036	8.0599301	26790.4408205
rbf	2.4428449	3120.1290861	2.7011621	3785.1852096
poly	3.846782	7653.0546364	3.5427364	6702.9617752
sigmoid	7.4151605	25926.8501961	6.078233	16227.4176559

Table A. 2: Test results with same SVR model of **Table A.1** per kernel divided per dimension

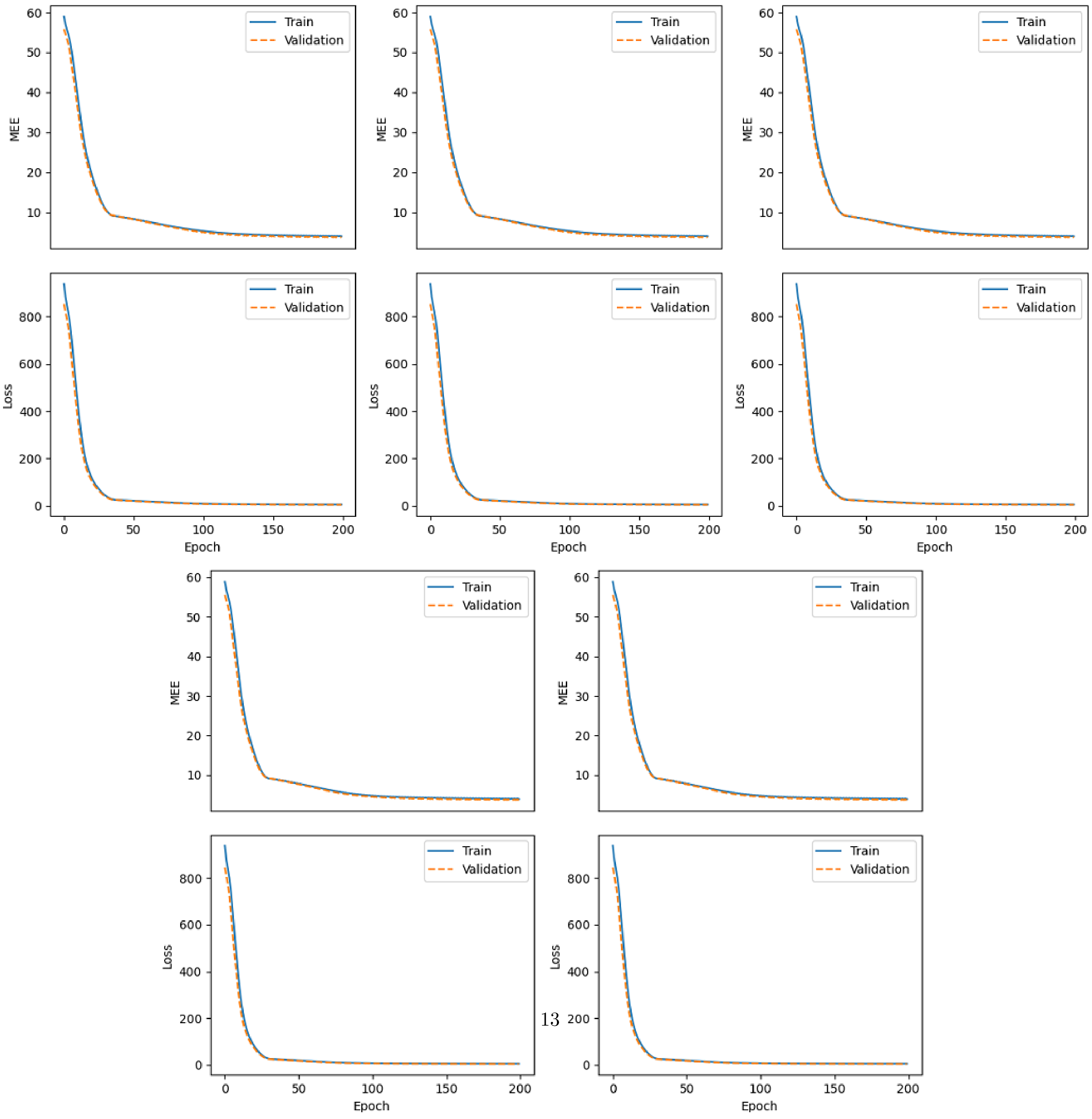

Figure 4: Ensemble models with respective *loss* (above) and *Mean Euclidean Error* (below)

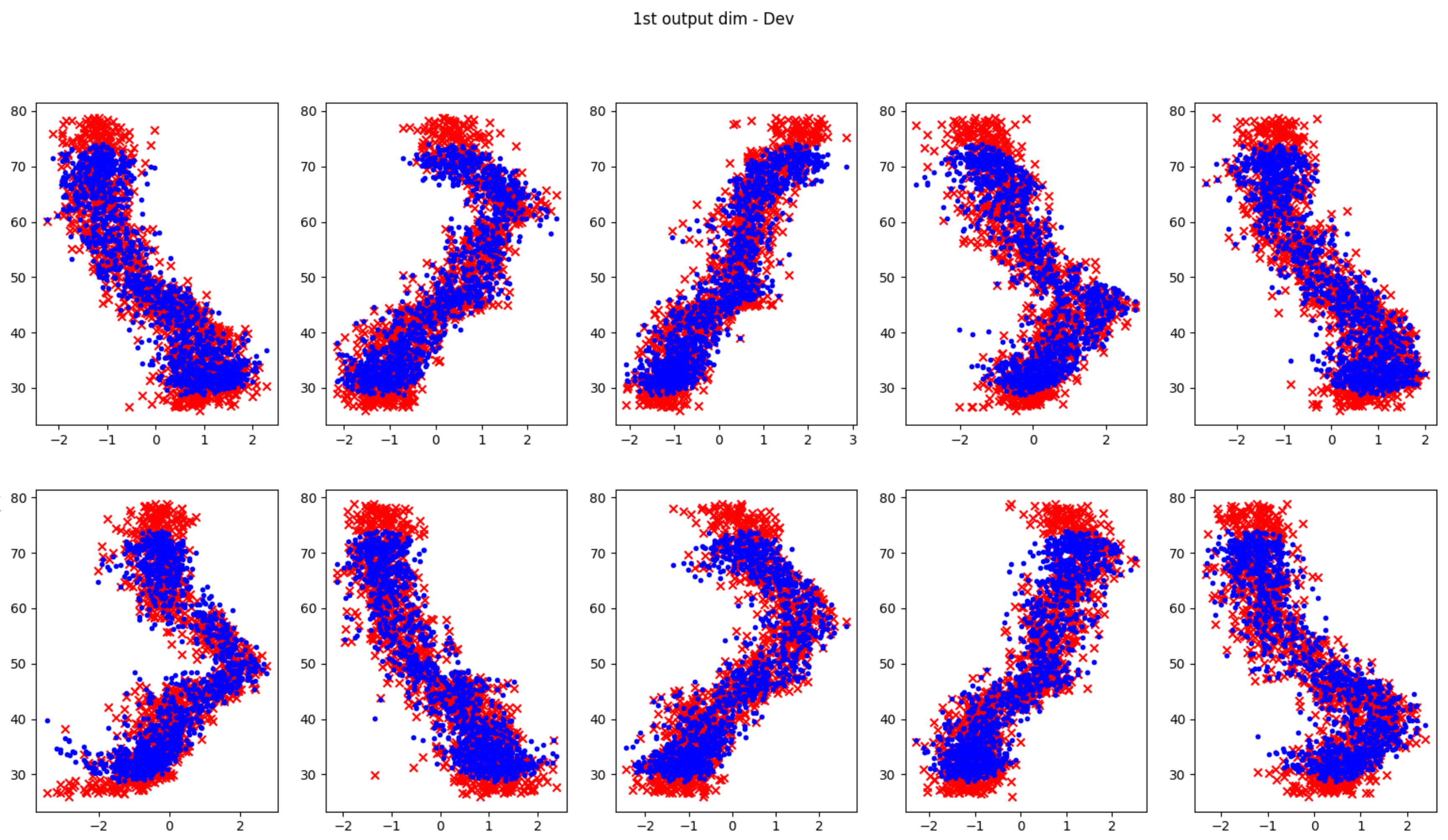

Figure 5: *Development set*: ground truth ('x' marker) vs SVR prediction ('.' marker) on the 10 input dimensions for the first output feature - *x-axis*: i-th input value, *y-axis*: output value

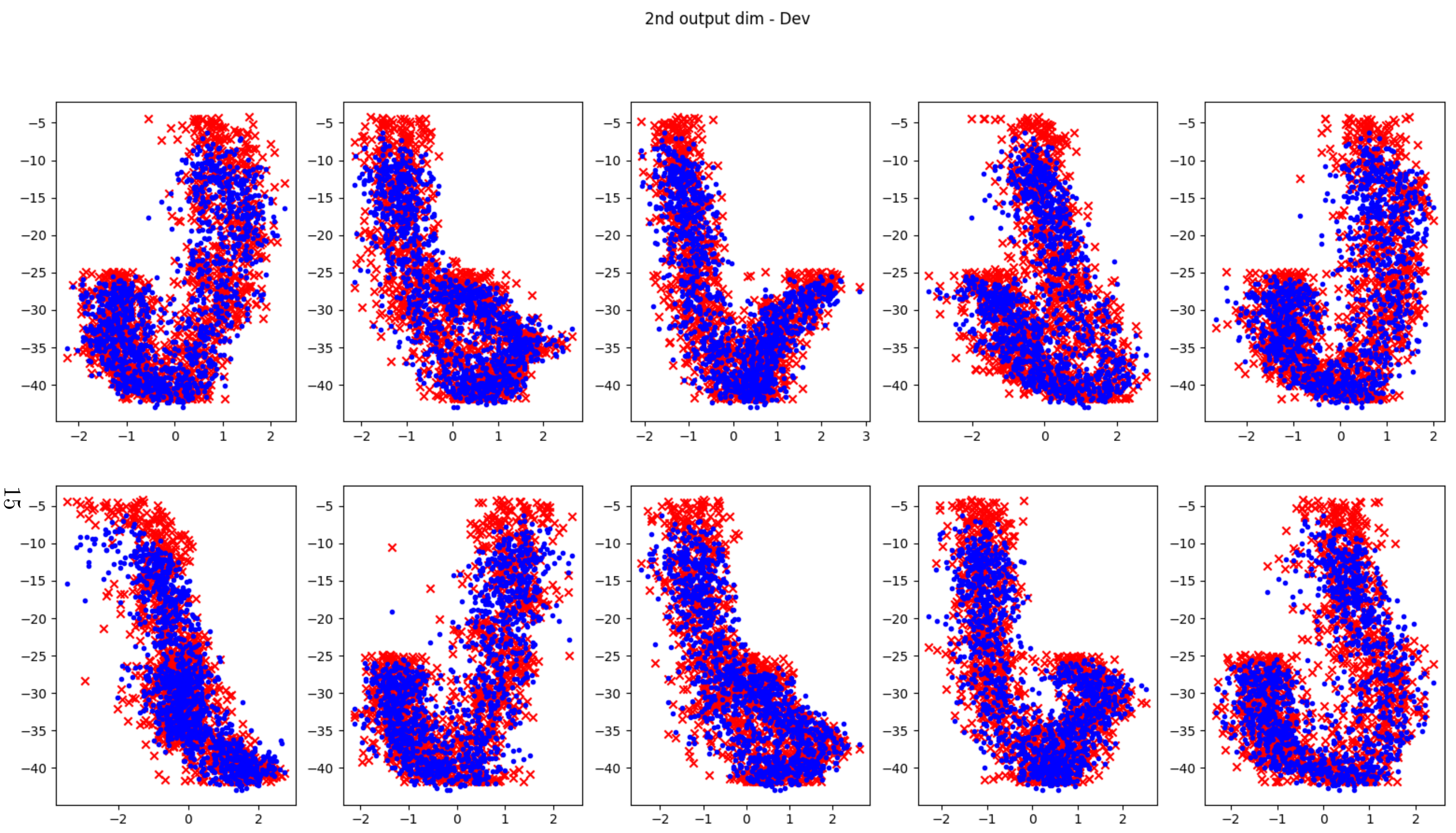

Figure 6: *Development set*: ground truth ('x' marker) vs SVR prediction ('.' marker) on the 10 input dimensions for the second output feature - *x-axis*: i-th input value, *y-axis*: output value

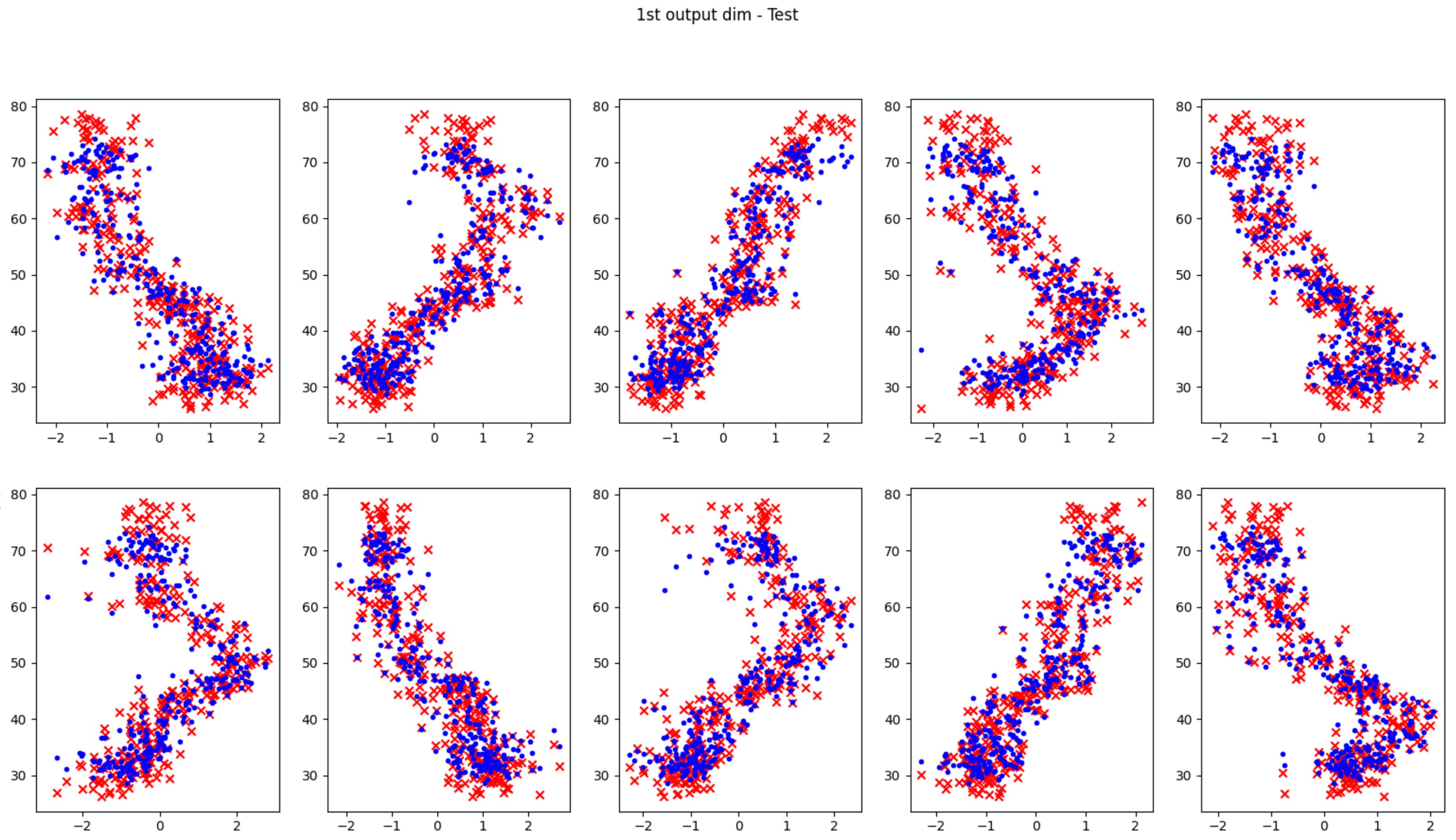

Figure 7: *Internal test set*: ground truth ('x' marker) vs SVR prediction ('.' marker) on the 10 input dimensions for the first output feature - *x-axis*: i-th input value, *y-axis*: output value

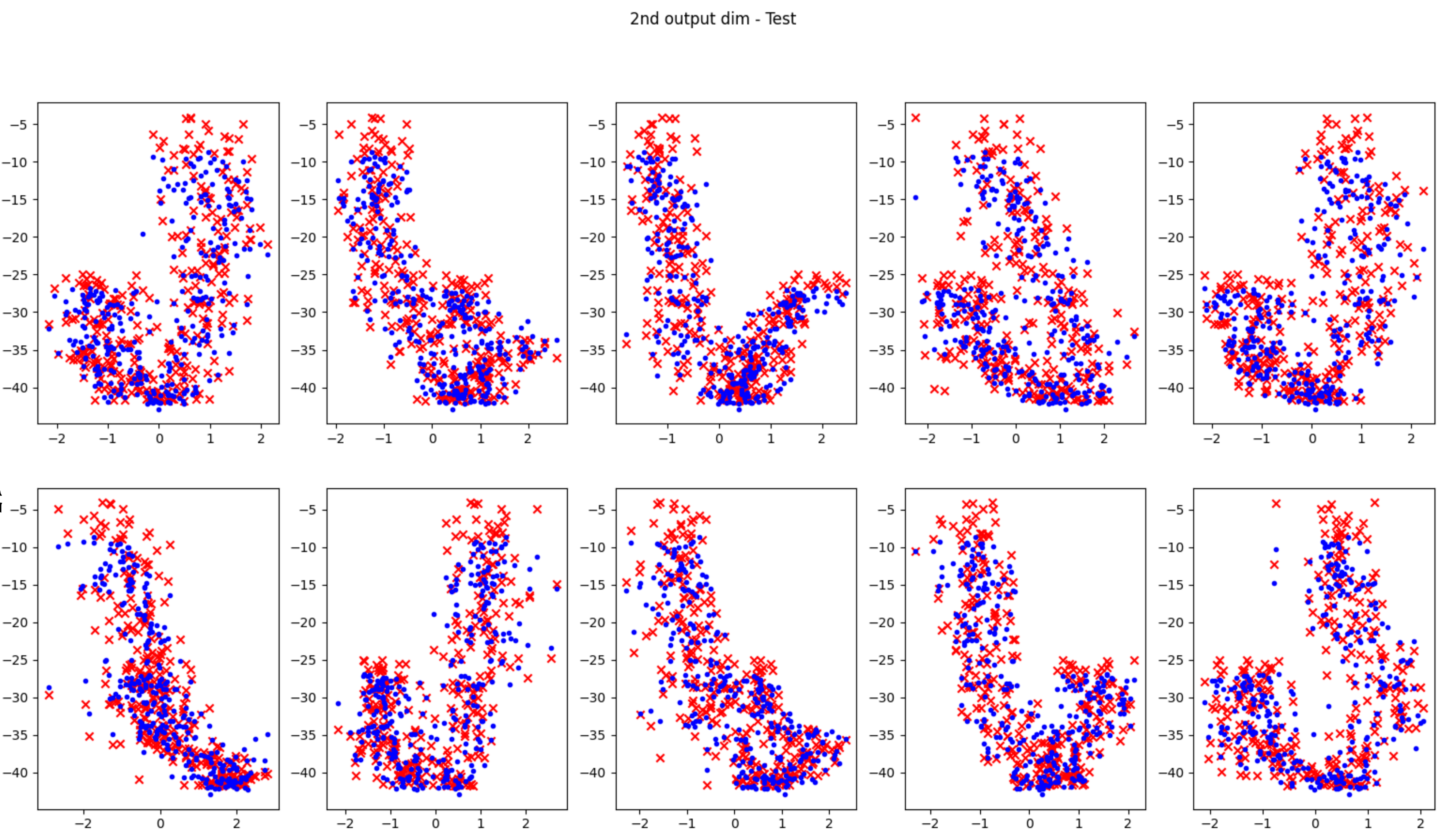

Figure 8: *Internal test set*: ground truth ('x' marker) vs SVR prediction ('.' marker) on the 10 input dimensions for the second output feature - *x-axis*: i-th input value, *y-axis*: output value