University of Oslo

Project 2

FYS-STK4155

José Luis Barreiro Tomé

Romain Corseri

November 18, 2022

Abstract:

Neural networks introduce non-linearity by providing a mapping that is not only a weighted sum of the data inputs. Gradient descent techniques are iterative solvers that are necessary to tackle optimization of high-dimensionality, non-linear cost functions. Those are incorporated to our implementation of feed-forward neural network code. The results are compared to linear regression techniques as we gouge the strengths and weaknesses of our deep learning models. We find that a single hidden layer and 4 nodes neural networks only slightly overperforms polynomial regression for 100 data points of the Franke function by reaching a mean squared test error of 0.013 (scikit) – 0.018 (own code) compared to 0.015. For the tumor classification problem (Wisconsin Breast Cancer dataset), we find that a neural network made of 3 hidden layers containing 4 nodes each provide the best predictive accuracy score of 93% (both scikit-learn and own implementation). Logistic regression model reaches a prediction accuracy score of 91% and is outperformed by deep learning in this case. Neural networks seem more robust to overfitting than linear regression. We experience that the architecture of the networks must be emphatically wide (>10 nodes for regression) or deep (>10 layers for classification) to deteriorate its predictive power. However, the problem of numerical instability (i.e., exploding gradients) seem much more salient for neural networks.

Introduction

The Universal approximation theorem states that a feed-forward neural network (FFNN) with one hidden layer can approximate any continuous function to a desired accuracy. Neural networks have become a popular method to solve problems within the supervised machine learning framework, with widespread applications in many fields: from image classification to speech recognition or medical diagnosis. Simpler models for regression or classification can perform well in many cases. But their applicability is limited when the target behaves in a non-linear way. By expanding to non-linear and high dimensional cost function optimization problem, the necessity of iterative solvers becomes evident as the computational cost of matrix inversion becomes unbearable. Those are the main motivations behind implementing FFNN and iterative solvers

The first aim of this project is to implement and test various gradient descent techniques for linear regression cases. Secondly, the iterative solvers are incorporated into our FFNN code. We assess the performance of a FFNN for both regression and classification tasks. Lastly, FFNN models are compared with linear and logistic regressions models to conclude on the most appropriate supervised machine learning tool.

As a benchmark test, we test the iterative solvers on a simple 2nd order polynomial and gouge the effect of various improvements, like introducing randomness in the choice of the training set. We incorporate our iterative solver to linear regression and FFNN algorithms and solve a regression task withh data generated by the Franke Function (Franke, 1979). For classification, we use the Wisconsin Breast Cancer dataset available from the Scikit-learn library (Pedregosa *et al.* 2011). We seek the most adapted FFNN architecture and systemically test the depth and width of the network. We also scan trough learning rates, epochs, batch size and activation function to fine tune our deep learning model.

1. Data and Methods

1.1. Theory

1.1.1. Gradient descent techniques

- *Gradient descent*

Gradient descent (GD) is an optimization method, used to find the minima of the cost function. Cost function is defined as a single, overall measure of loss incurred in taking any of the available decisions or actions (Bishop, 2007). Gradient is defined as the derivative of a multivariable function, in our case the cost function. The main idea of the GD is that a convex function $E(\theta)$ decreases fastest when going from x in the direction of the negarive gradient $\nabla_\theta E(\theta_t)$.

$$v_t = \eta_t \nabla_\theta E(\theta_t)$$

$$\theta_{t+1} = \theta_t - v_t$$

η_t is the learning rate or step-size we use for moving in the direction of the gradient.

Taking steps iteratively in this direction the minima will be found if it exists. One common problem in these methods is to find local minima instead of global minima in non-convex functions. The simplest approach is to take a step size (or learning rate η).

In practice, the GD method is very sensitive to initial conditions (i.e., starting point for the iterations) and choice of learning rates. It is as well a method computationally expensive, especially for large datasets, since it computes the cost function and its gradient for all data points.

The performance can be accelerated by introducing momentum term γ, where the previous steps are remembered. This allows to build inertia in a direction in the search space and accelerate the search specially through noisy gradients or flat spots.

$$v_t = \gamma v_{t-1} + \eta_t \nabla_\theta E(\theta_t)$$

$$\theta_{t+1} = \theta_t - v_t$$

Linear regression has several desirable properties for calculating GD methods: it has an analytical solution; the gradient can be also computed analytically, and the cost function is convex.

- *Stochastic Gradient Descent*

Stochastic Gradient Descent (SGD) deals with the sensitivities of GD to the initial conditions, choice of learning rate or convexity of the cost function. It chooses a random instance in the training set at every step and computes the gradients based on only one instance (Géron, 2019). This algorithm is less regular than the GD because of its stochastic nature. But it can help avoid the local minima, although it can make it harder to converge to a minimum. To solve this, it is common to use an adaptative learning rate, where the steps taken are larger at start, and get smaller as the algorithm approach the global minimum.

In addition, to further accelerate the computations, the gradient is computed over so-called mini batches of the training data, instead of computing the cost function and gradient for all points of the training dataset. This is especially useful with large training datasets

- *Algorithms with adaptative learning rate*

The learning rate is one of the hyperparameters that is the most difficult to set because it has a significant impact on model performance. The momentum algorithm can mitigate these issues in part, but at the cost of introducing another hyperparameter (Goodfellow et al., 2016). Recently there has been developed different methods that can adaptively change the step size to match the search space without having to calculate or

approximate the Hessian, that represents the second derivative of the cost function. They accomplished this by tracking the gradient plus the second moment of the gradient.

ADAGrad: short for Adaptative Gradients (Duchi et al., 2011), is an extension of the GD. It uses an aggregate of the squares of previously observed gradients, and scales down the gradient vector along the steepest dimension. In practice, its significantly decrease learning rates on large gradient parameters, while the decrease is smaller on parameters with smaller gradients. ADAgrad converged rapidly when we use it in convex functions, but it is not ideal with non-convex functions.

RMSProp: it accumulates the exponentially weighted moving average of just the most recent iterations, as opposed to the gradient accumulation since the beginning of training in ADAgrad. This implies a better performance in non-convex functions, such as those typically used in machine learning and neural networks. RMSProp is widely used in practical optimization algorithms for neural networks (Goodfellow et al., 2016).

Adam: short from adaptive moment estimation, (Kingma and Ba, 2014) it combines properties from both RMSProp and GD with momentum optimization. ADAM keeps track of the running average of the first and the second moment of the gradient and consequently adapt the learning rate for different parameters. In addition, ADAM includes a bias correction to account for the running average of the first and second moments.

1.1.2. Feed-forward neural networks (FFNN)

- *Definitions*

Neural Networks are nonlinear, statistical models, that can learn to approximate some regression function or classification model. They are designed to mimic a human brain, in which an arbitrary number of neurons within a layer send signals to other neurons in a different layer. Each neuron accumulates its incoming signals, and they must exceed an activation threshold to yield an output. Each connection is represented by a weight variable plus some bias. They are called *feedforward* when the information flows just in one direction through the model, from the input, through the layers and produces an output. They are called *networks* because they can be composed by many different functions.

In Fig. 1, a network diagram of a multilayer perceptron (MLP) is depicted. This is a fully connected Feed-Forward Neural Network (FFNN) with an input layer with four neurons (x_p), one hidden layer (it could be more) with 5 neurons and an output layer with 2 neurons (y_k). Each circle represents neurons, the weight parameters are represented by links between the neurons. Activation functions presented on the hidden layer (z_h). For classification problems, there is often an output function on the output layer. For regression there is typically only one output, and the output function would often be the identity function. The length of the network gives the depth of the model, while the dimensionality of the hidden layers determines the width.

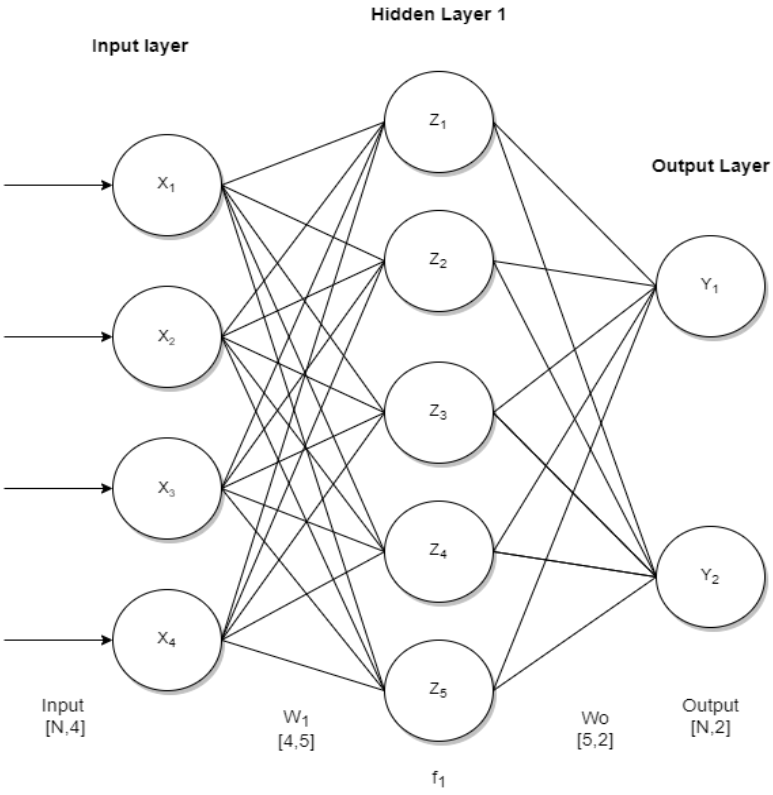

Fig. 1: Diagram of an example of single hidden layer, FFNN.

The performance of the neural network is as follows. Each input x_p is multiplied by a weight w_i, that indicate the strength of the connection to each node in the next layer. To avoid outputs of only zeros, a small bias is added to each weight. These weights are usually initialized with random small numbers distributed around zero. The result is analyzed by an activation function (z_h), and the result is the output for each node, which at the same time becomes the input for the next layer

The training data shows the behavior of the output layer: the output data should be close to y. But it doesn't specify the behavior of the hidden layers, the learning algorithm decides how to use them to end producing the desired output (that is the reason for calling them hidden layers).

- *Activation functions*

The choice of activation layers can have big influence on the performance of the neural network. In a FFNN the activation functions are required to be non-constant, bounded, monotonically increasing and continuous (Hjort-Jensen, 2022) for the network to work well in complex datasets. Common choices include sigmoid function, the Rectified Linear Unit function (ReLU), the hyperbolic tangent function (than) or the Leaky ReLU.

Sigmoid: The Sigmoid (or Logistic) function is the preferred one for classification problems with binary outcome, which aims to predict outputs in form of discrete variables (yes/no, true/false etc.).

$$p(t) = \frac{1}{1 + e^{-t}} = \frac{e^t}{1 + e^t}$$

$$1 - p(t) = p(-t)$$

The sigmoid function gives outputs in the range $p \in [0,1]$.

To measure the performance of the classification network, we apply an accuracy score to some data that has not been part of the training dataset. This term is simply the number of outputs classified correctly divided by the total. The expression is as follows:

$$Accuracy = \frac{\sum_{i=1}^{n} I(\tilde{y}_i = y_i)}{n}$$

Where I is the indicator function, 1 if $\tilde{y}_i = y_i$, and 0 otherwise.

Softmax: The Softmax function, also called Multinomial Logistic Regression, can be used for outputs with multiple K classes. It produces positive probabilities that sum to one:

$$softmax(z)_i = \frac{e^{z_i}}{\sum_{l=1}^{K} e^{z_l}}$$

ReLU: This function only activates when the input variable that goes through it is positive:

$$ReLU = \begin{cases} 0 & if\ x < 0 \\ x & if\ x \geq 0 \end{cases}$$

ReLU works well in deep neural networks and it's quite fast to compute. However, it suffers from a problem known as dying ReLU. This is due to some neurons stop outputting anything other than 0. To solve this problem, there is a variant of this method called Leaky ReLU.

Leaky ReLU: This form of ReLU has a small positive gradient for negative values:

$$Leaky\ ReLU = \begin{cases} 0.01\ x & if\ x < 0 \\ x & if\ x \geq 0 \end{cases}$$

tanh: The hyperbolic tangent activation function is related to the sigmoid. The difference is the range of outputs is from (-1 to 1). The main advantage with the sigmoid is that tanh is faster.

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$

- *Backpropagation algorithm*

After one feed-forward pass through the network (i.e., one iteration), we need to adjust the weights and biases that connect the neurons in the network, to minimize the errors in the output. After performing the feed-forward, the output error (last layer L) is given by

$$\delta_j^L = f'(z_j^L)\frac{\partial C}{\partial(a_j^L)}$$

where $f'(z_j^L)$ is the derivative of the activation function of the last j neuron, a_j^L is the output data of the last j neuron and C is the cost function.

The back-propagate error can be computed for each layer $l = L-1, L-2, \dots, 2$ as

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l)$$

Weights and biases can now be updated using GD for each $l = L-1, L-2, \dots, 2$ according to

$$w_{jk}^l \leftarrow= w_{jk}^l - \eta \delta_j^l a_k^{l-1},$$

$$b_j^l \leftarrow b_j^l - \eta \frac{\partial C}{\partial b_j^l} = b_j^l - \eta \delta_j^l$$

Where η is the learning rate

1.1.3. Python programming and benchmark tests

All the algorithms were implemented in python 3.8. The packages we used were numpy (Harris et al. 2020), seaborn (Waskom 2020) and scikit-learn (Pedregosa *et al.* 2011). We coded our own neural network code for both regression and classification problem in form of a python class. Different classes were created for all variants tested for ease of use, documenting all the process and quality check the results.

The results from our neural network classes were compared with ready functionalities from scikit-Learn. For regression tasks, our own implementation of FFNN was benchmarked against Scikit-Learn's *MLPRegressor*. For classification problem, we compared our results with scikit-Learn's *MLPClassifier.*

1.2. Data

1.2.1. For regression task: Franke Function data

The reader is referred to section 2.3.1 in Project 1 report.

1.2.2. For classification task: Wisconsin Cancer data

The dataset used for classification problem is the Wisconsin Breast Cancer dataset, and it was uploaded directly in our python script through the scikit-learn package. The dataset

has a series of input variables extracted from patients with solid breast masses, such as mean radius, mean perimeter, mean texture, or mean compactness. In Fig. 2, we can visualize the correlation between some of the tumor features. The tumor perimeter and radius are evidently highly correlated features.

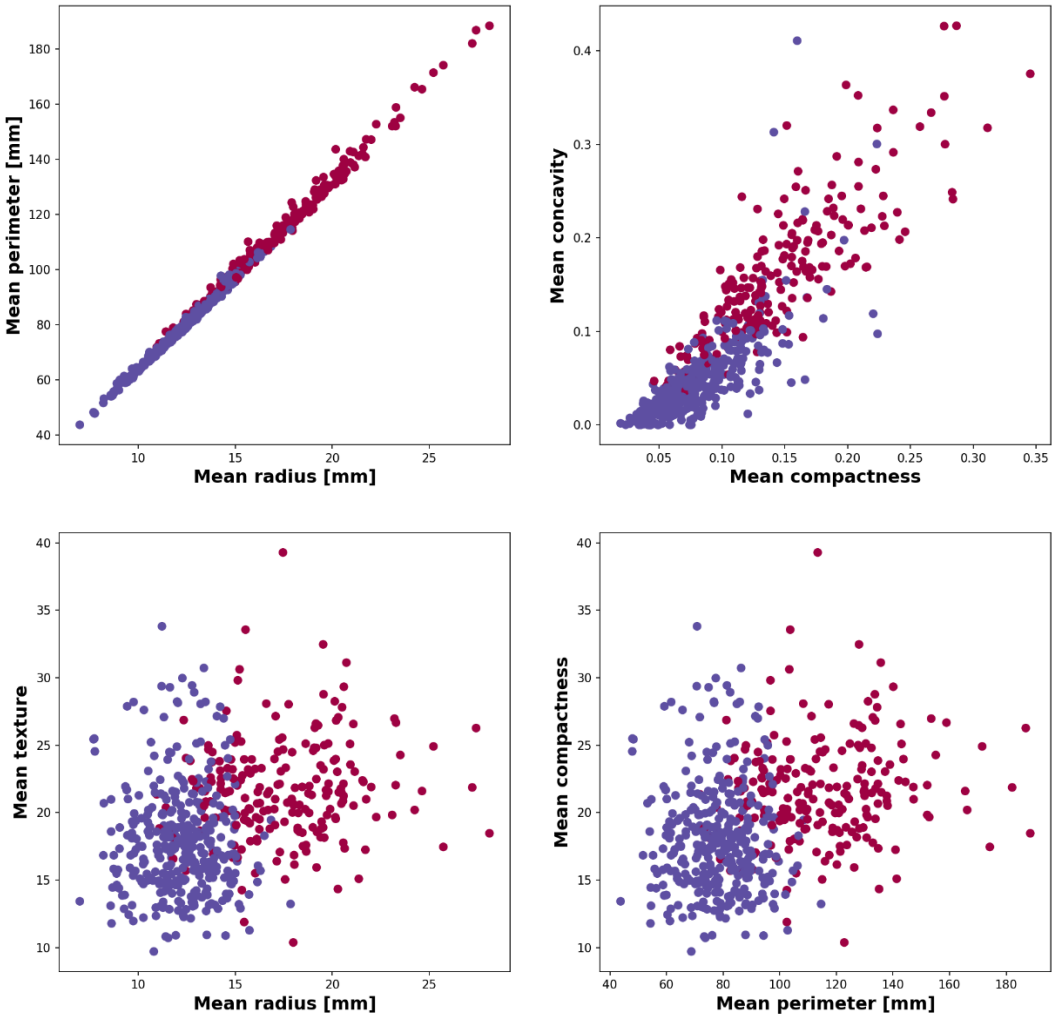

Fig. 2: Visualization of five tumor features from the Wisconsin Breast Cancer dataset, color-coded according to the final diagnosis (red: malignant, purple: benign)

2. Results

2.1. Implementation and tests of various gradient descent techniques for a simple 2nd order polynomial.

We implemented the gradient descent algorithms and tested it on a regression task for a bi-variate polynomial function of degree 2 with all coefficients equal to 1 (Figs. 3 and 4). The programs also provide the mean squared error and predictors values (β values) from scikit-learn in-built function *LinearRegression()* and matrix (pseudo) inversion (Project 1) for a fixed learning rate (η=0.01), 100 data points and problem complexity of 2.

We chose to represent the convergence curves as function of iterations (Fig. 3) for plain GD and epochs for SGD (Fig. 4).

For the plain GD, we set a stop criteria $\varepsilon = 10^{-6}$ but we note that the MSE would decrease to an arbitrarily low values if we decide to set $\varepsilon = 10^{-30}$ with computation time as the only limiting factor. To reach a training MSE level of $\sim 10^{-10}$, it takes ~50000 more iterations for the plain GD implementation without momentum than with it. Adding momentum improves the convergence time of the GD iterative solver by ~30%. Meanwhile, scikit-learn and matrix inversion implementation overperforms the GD and reaches rapidly a training MSE of $\sim 10^{-30}$ and $\sim 10^{-27}$.

For the SGD, the number of epochs is set to 30000 and minibatch size of 20. We observe that SGD converges to a similar MSE level of $\sim 10^{-10}$. Again, adding momentum to the descent improves convergence time by ~30% by reducing the number of epochs to 20000 to reach the same MSE level (Fig. 3).

Comparing the convergence time of SGD with momentum (~5s) and GD with momentum (~12s), we find that adding randomness to the initialization of the training set improves the convergence time by ~60%.

$$f(x,y) = 1 + x + y + xy + x^2 + y^2$$

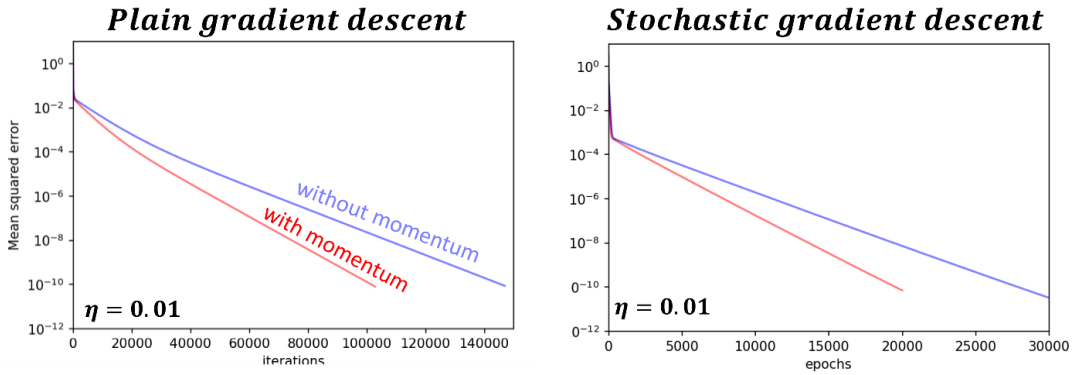

Fig. 3: Error convergence curve for plain GD (left plot) and SGD (right plot) with and without momentum for a fixed learning rate of 0.01. Note the y-axis is on a logarithmic scale.

Then, we apply three adaptive learning rates tuning methods (ADAgrad, ADAM and RMSprop with momentum) to the same regression task. Importantly, the learning rate had to be initially set to η=0.00005 to reach a training MSE level of ~10^{-10}. The ADAgrad and RMSprop behave almost identically and converge to similar MSE at ~12000 epochs. Both implementations yield a very similar convergence trajectory. The ADAM implementation converges at ~14000 epochs and is outperformed by the other tuning methods in terms of convergence time. ADAgrad and RMSprop improves the convergence time by ~40% and ADAM by ~30% with respect to "plain" SGD.

$$f(x,y) = 1 + x + y + xy + x^2 + y^2$$

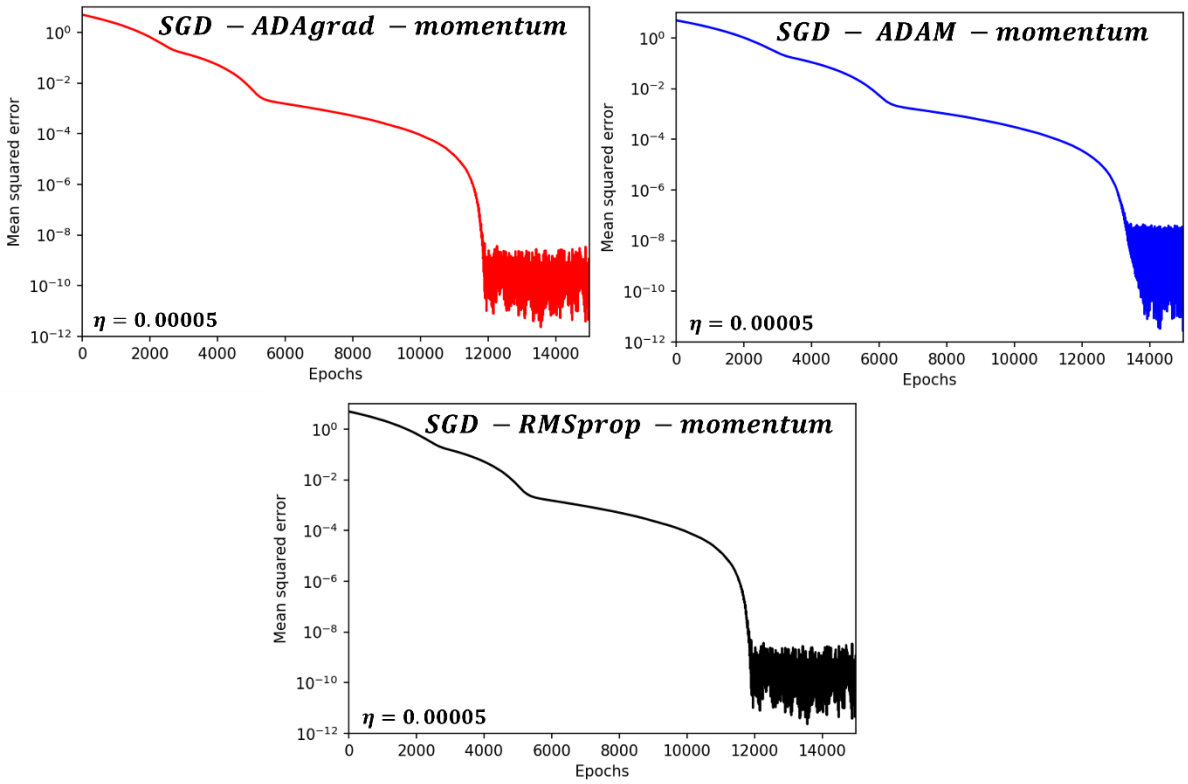

Fig. 4: Error convergence curve for stochastic GD with and without momentum for a fixed learning rate of 0.01, 800 epochs and mini batch size of 20. Note the y-axis is on a logarithmic scale

Code used to produce the plots for section 3.1 in Generic_codes_solver/:

GD.py
GD_momentum.py
SDG.py
SDG_momentum.py
SDG_adagrad_mometum.py
SDG_RMSprop_momentum.py
SDG_adam_momentum.py

2.2. Comparison of various GD techniques and learning rate tuning method for the Franke Function data regression task

2.2.1. Ordinary least square (OLS)

In Fig. 5, the linear regression results for the Franke Function data are displayed as a grid with epochs and learning rates as hyperparameters. Note that for all tuning methods (ADAgrad, ADAM, RMSProp), momentum is included in the implementation. We set the problem complexity to the optimal model: a 4^{th} order polynomial following the conclusion of project 1. We use 100 datapoints (splitted into 80% training and 20% test sets) and 10 as minibatch size. First, we observe that the best training and prediction MSE is obtained for a learning rate within the range of 10^{-3} to 10^{-1} regardless of the tuning method. In this range of η, the optimal number of epochs is 10000, as we observe that the training and predictive ability gets better with increasing epochs. In this case, the best prediction is obtained with ADAGrad, η=0.1 and 10000 epochs but we note that the overall performance for each tuning method is similar across the range of hyperparameters investigated.

In Fig. 6, the batch size is a hyperparameters while the number of epochs is set to 10000. We find that the optimal batch size is at 20 data points (out of 80 training data points), for a learning rate of η=0.1 using either ADAgrad or RMSprop as tuning methods. These two techniques give very similar results except for η =1 or 10 at M=20 and their overall predictive power is slightly better than ADAM. From this exercise, we will keep as rule-of-thumb that the optimal batch size represents a quarter of the training set.

2.2.2. Ridge regression

By introducting the regularization term into the cost function, we only slightly degrade the training and prediction error of a polynomial of degree 4 fitting the Franke Function (Fig. 7). Unsurprisingly, we converge to a similar conclusion of project 1, which is that OLS model has a better predictive power than the Ridge regression model for the Franke Function data. Implementing SGD as an iterative solver yield equivalent training MSE level as with matrix inversion (0.013 in project 1 versus 0.011 with SGD). The optimal learning rate is $\eta=10^{-2}$ - 10^{-1} and regularization $\lambda=10^{-4}$. ADAgrad and RMSprop give almost identical results, marginally overperforming ADAM for the two aforementioned learning rates.

Code used to produce the plots (section 3.2):

FF_OLS_hyperparam_eta_epochs_SGD_ADAGRAD_momentum.py
FF_OLS_hyperparam_eta_epochs_SGD_adam_momentum.py -
FF_OLS_hyperparam_eta_epochs_SGD_RMSprop_momentum.py –
FF_OLS_hyperparam_eta_batch_size_SGD_RMSprop_momentum.py -
FF_OLS_hyperparam_eta_batch_size_SGD_ADAGRAD_momentum.py -
FF_OLS_hyperparam_eta_batch_size_SGD_adam_momentum.py –
FF_Ridge_gridsearch_SGD_ADAGRAD_momentum.py
FF_Ridge_gridsearch_SGD_adam_momentum.py
FF_Ridge_gridsearch_SGD_RMSprop_momentum.py

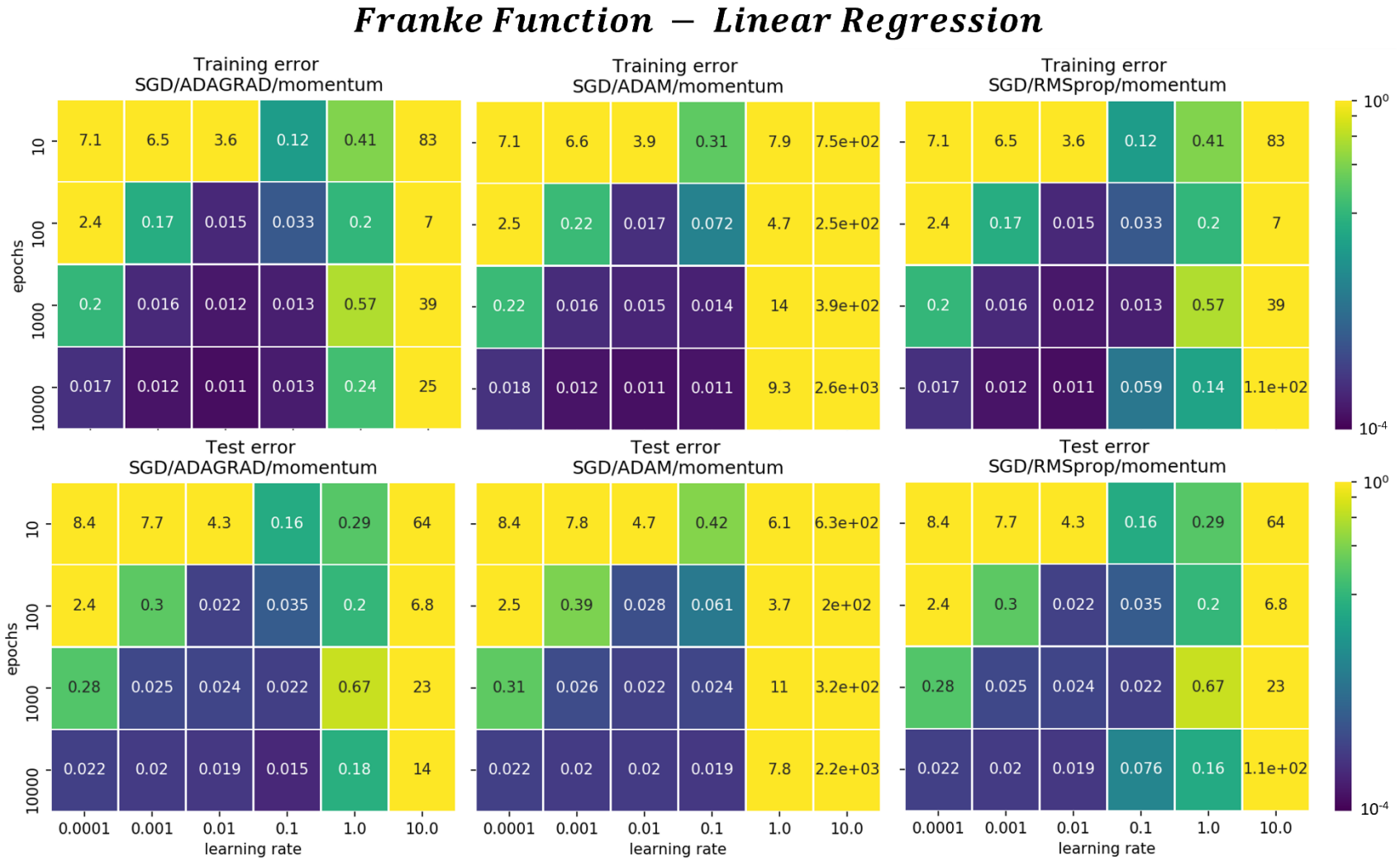

Fig. 5: Grid search plot for the Franke Function OLS regression results (training and test MSE) with learning rate and number of epochs as hyperparameters with ADAGRAD (left), ADAM (middle) and RMSprop (right) as tuning methods.

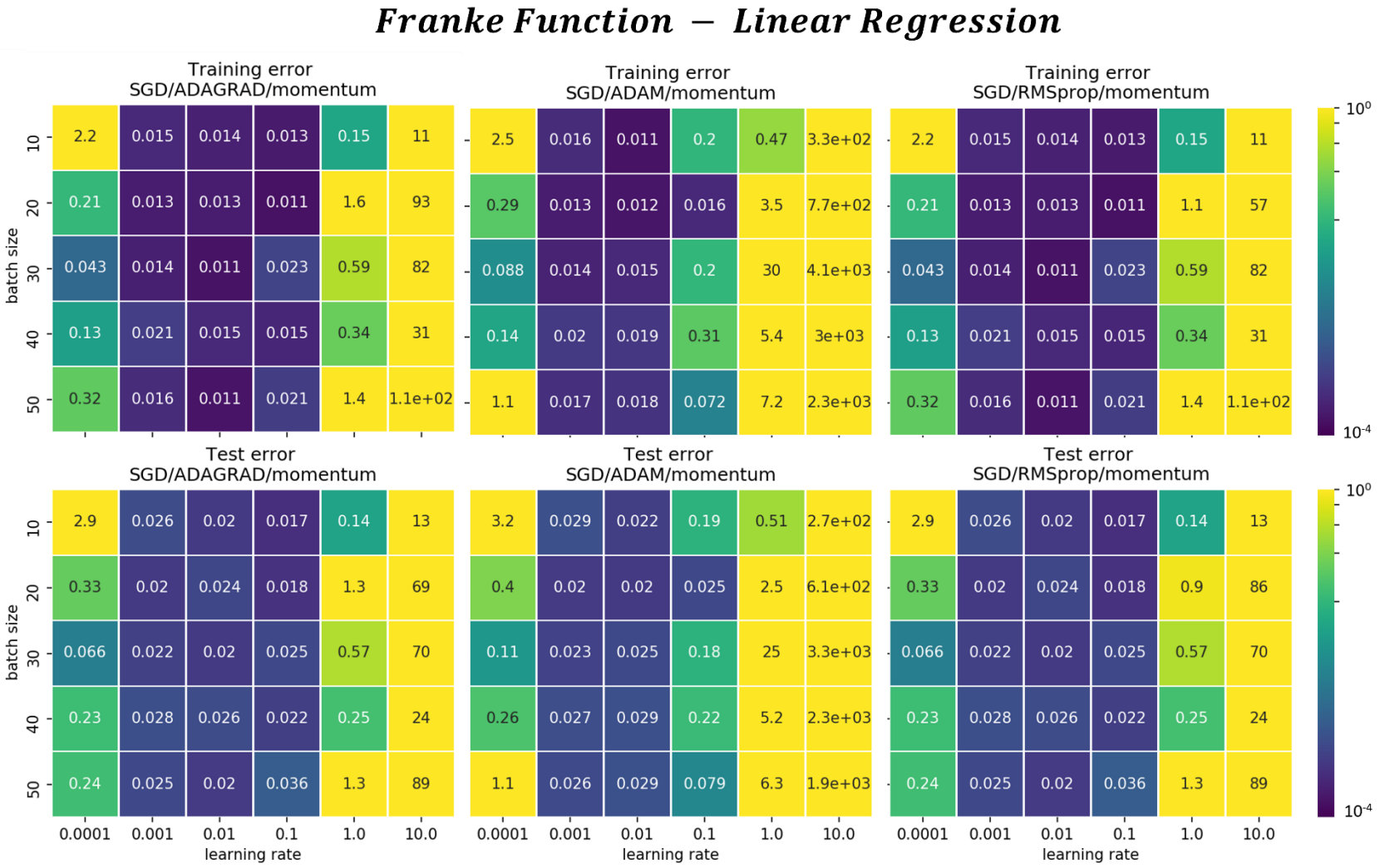

Fig. 6: Grid search plot for the Franke Function OLS regression results (training and test MSE) with learning rate and batch size as hyperparameters with ADAGRAD (left), ADAM (middle) and RMSprop (right) as tuning methods.

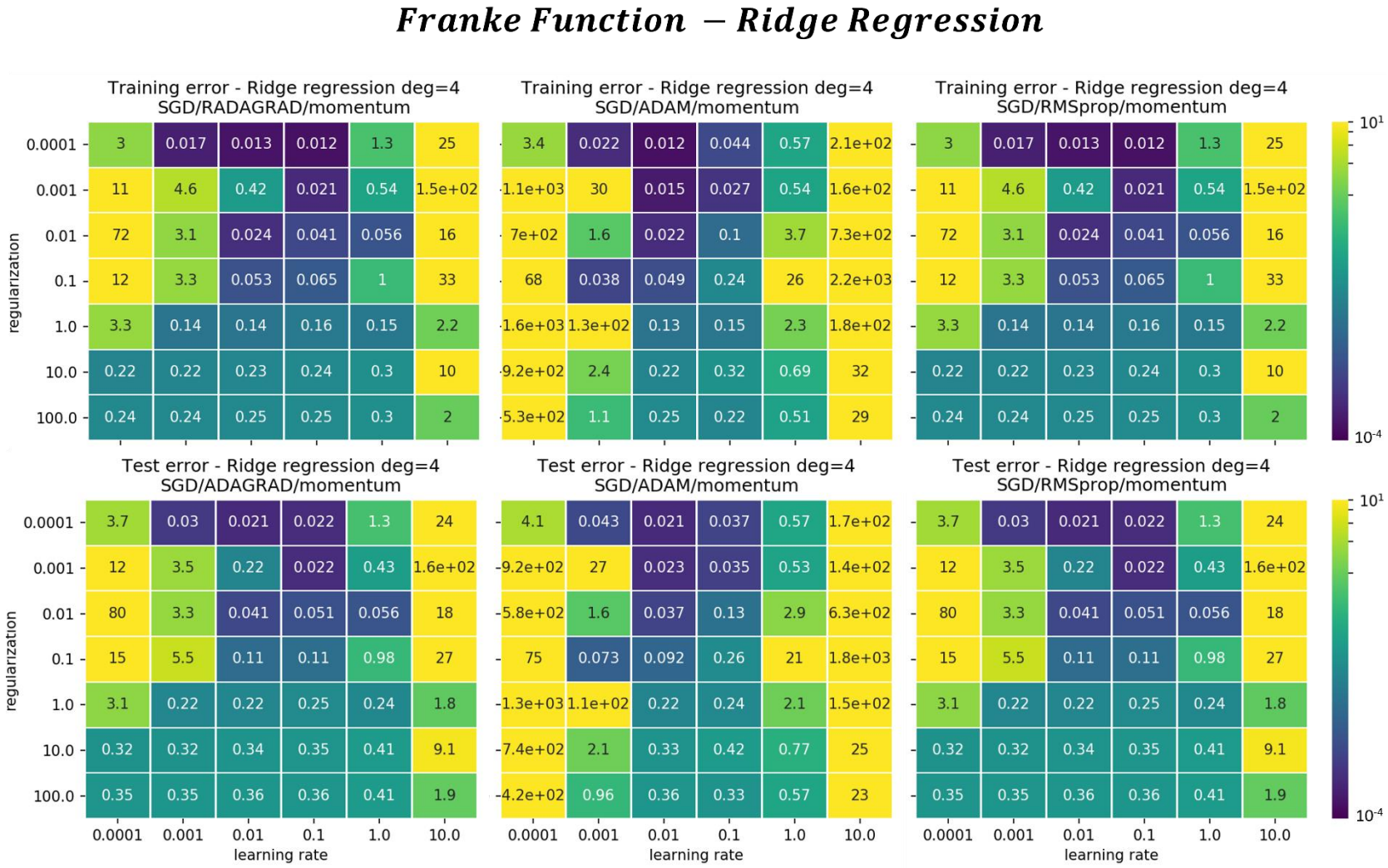

Fig. 7: Grid search plot for the Franke Function Ridge regression results (training and test MSE) with learning rate and regularization as hyperparameters with ADAGRAD (left), ADAM (middle) and RMSprop (right) as tuning method

2.3. Own implementation of FFNN using Numpy

2.3.1. For a regression task: The Franke Function data

We implemented FFNN for regression as python class in *NeuralNetwork_regression_*.py* with * standing for the activation function (leakyrely / relu / sigmoid) of the hidden layers. For the regression tasks, the activation function of the output layer is the identity function. For the Franke Function, the input layer is made of two nodes (x1, x2) and the output layer of a single node (z), thereby defining the input and output layers of our FFNN. The cost function includes a L2-norm regularization term. We have implemented a stochastic GD in the backpropagation algorithm to update the weight and biases. The weights are initialized according to a "standard normal" distribution and the biases are initially set to 0.01.

The first part of the numerical experiment is to test the effect the neural network architecture on the regression problem for a given activation function (Sigmoid function in Figs. 8 to 12). After determining an optimal architecture, we investigate the effect of the activation function (Fig. 13) and conclude on the optimal combination of FFNN architecture and activation function for this regression problem. Note that we systematically compare our FFNN implementation with a professionally written code of the scikit-learn library.

First, as a general observation on comparing our FFNN implementation with Scikit-learn, we find that the Scikit-learn FFNN outperforms our own implementation in the sense that it often reaches lower MSE values (both train and test) and that the numerical stability is maintained across the whole range of learning rates investigated (10^{-5} to 10^{-1}). In our own FFNN implementation, we experience exploding gradients values and diverging MSE for η=0.1 and beyond. In some instances where the FFNN architecture becomes wider (number of nodes superior to 10), we also observe instability for η=0.01 (Figs. 11 and 12).

- Effect of FFNN architecture

In Figs. 8, 9, 11 and 12, we scrutinize the effect of increasing the number of nodes of a single-layered FFNN from 1 to 100. The training and test error seem to effectively decrease when increasing from 1 to 4 nodes. On the contrary, both training and predictive power tend to deteriorate for the 100 nodes, single-layered FFNN (Fig. 12). From 4 to 10 nodes, the test error look unchanged or slightly higher for 10 nodes and at lowest for η=0.01. The training error still decreases (especially marked for the Scikit-learn implementation) which likely indicates overfitting for a single-layered FFNN for single layer FFNN with 10 nodes and above.

On Figs. 9 and 10, we increase the number of hidden layers to 3 while keeping a fixed number of nodes of 4. We thereby test the effect of the depth of the FFNN on the regression problem. Overall, we observe marginally lower test and train error values in both scikit and our own implementation of the single layer FFNN compared to the deeper FFNN with 3 hidden layers. We would rather favor the simplest model and thereby

conclude that there is no need to increase the depth of the FFNN from 1 to 3 layers for our regression problem.

- Effect of activation function

After determining our optimal neural network architecture (a single layer with 4 nodes), we investigate the effect of the activation functions on the training and predictive power of our FFNN (Fig. 13). We observe that using ReLU and Leaky ReLU gives lower MSE across the relevant range of learning rate (η=0.001-0.01) and regularization ($\lambda=10^{-3}$ to 10^{-5}) albeid more instability and exploding gradient values. The optimal hyperparameters are η=0.01 and $\lambda=10^{-3}$, and we converge to a training MSE of 0.011 and test MSE of 0.018 in our own FFNN implementation using ReLU and leaky ReLU as activation functions. For comparison and same set of hyperparameters,the scikit-implementation of FFNN using SGD as solver and ReLU as activation function gives a training MSE of 0.022 and test MSE of 0.028.

We summarize the observations and describe the optimal set of parameters for the regression task using FFNN of fitting 100 datapoints generated from the Franke Function with added normally distributed random noise $N(0,0.1)$:

i. The optimal FFNN architecture is a single hidden layer containing four nodes
ii. The optimal activation function is either ReLU or Leaky ReLU
iii. The optimal learning rate (SGD) is η=0.01 and the optimal regularization parameter $\lambda=10^{-3}$

Code used to produce the plots (section 3.3.1):

Grid search:
FF_NN_gridsearch_sigmoid.py
FF_NN_gridsearch_leakyReLU.py
FF_NN_gridsearch_ReLU.py

Neural Network class definition:
NeuralNetwork_regression_sigmoid.py
NeuralNetwork_regression_leakyrelu.py
NeuralNetwork_regression_relu.py

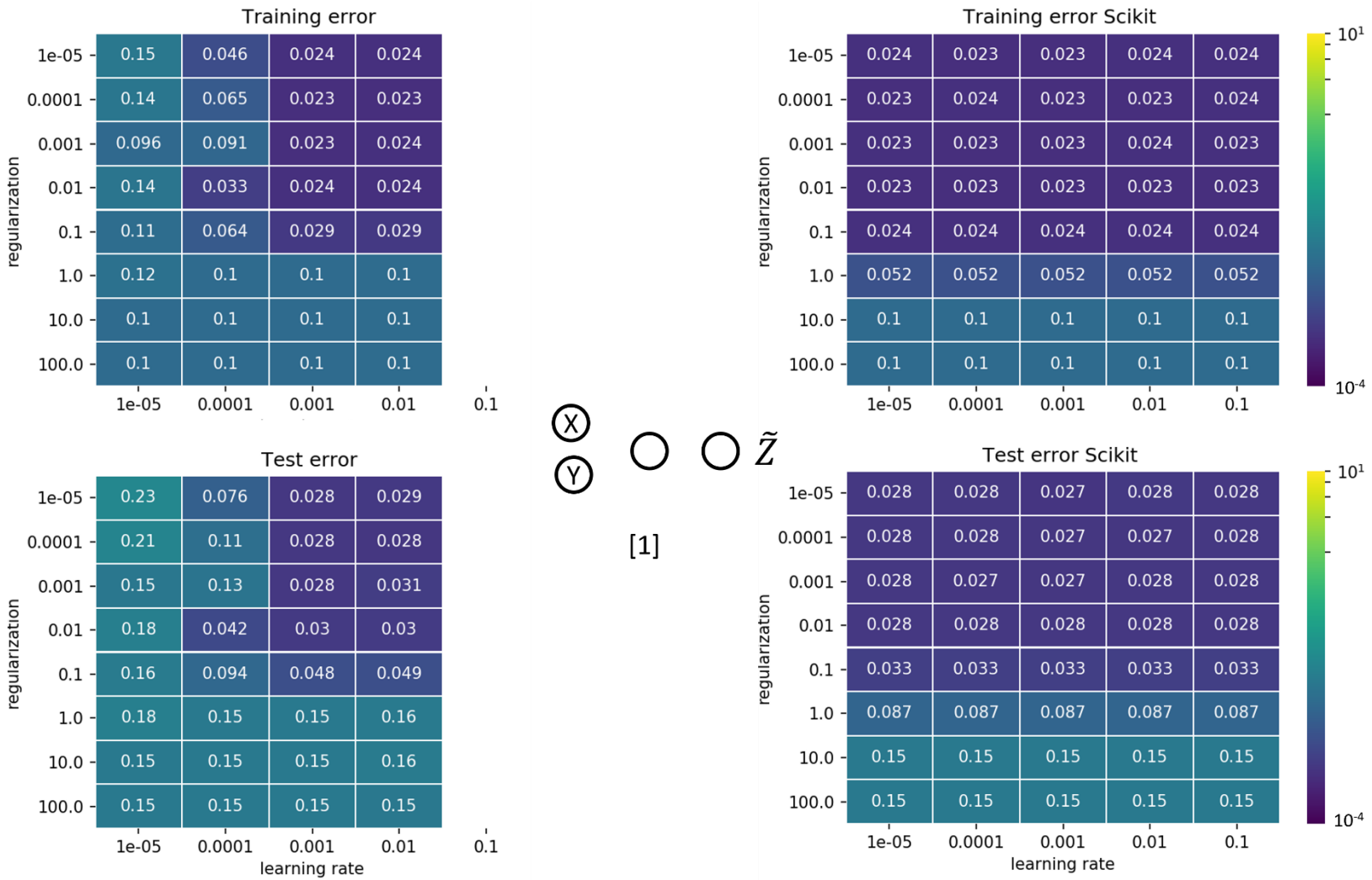

Fig. 8: Grid search plot for the Franke Function regression task with learning rate and regularization as hyperparameters using a FFNN with a single hidden layer and 1 node (left) own implementation (Right) Scikit-learn implementation

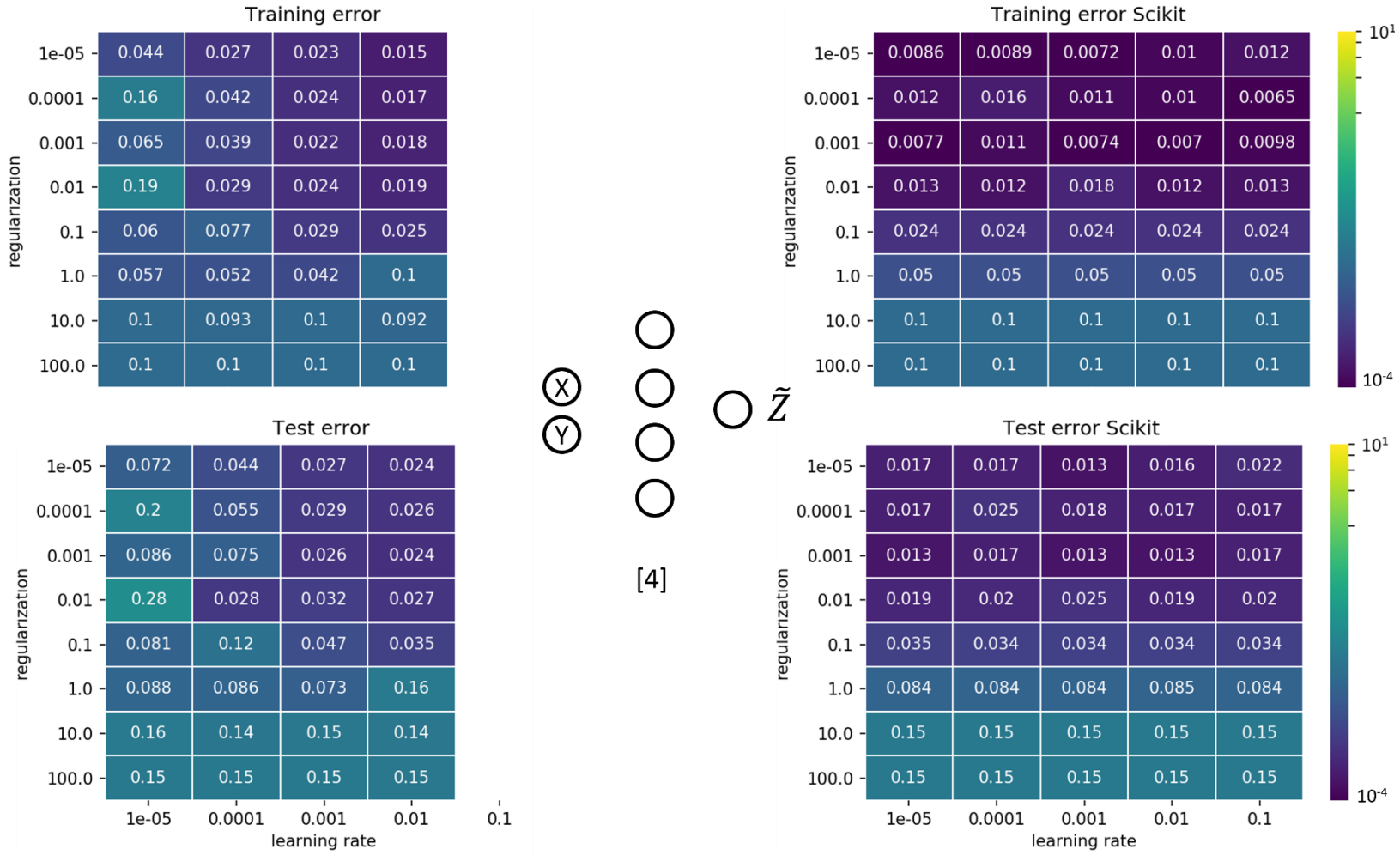

Fig. 9: Grid search plot for the Franke Function regression task with learning rate and regularization as hyperparameters using a FFNN with a single hidden layer and 4 nodes (left) own implementation (Right) Scikit-learn implementation

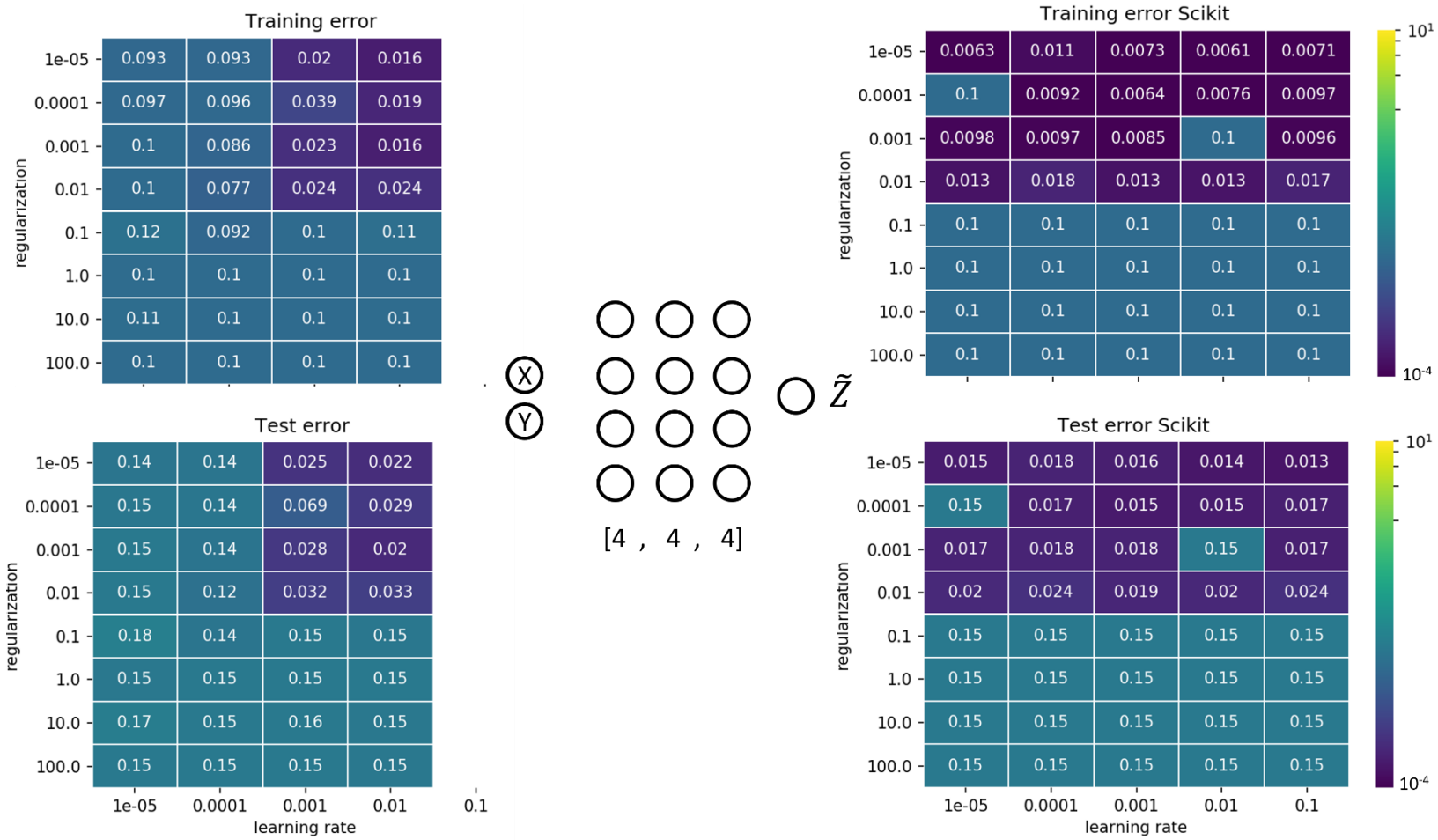

Fig. 10: Grid search plot for the Franke Function regression task with learning rate and regularization as hyperparameters using a FFNN with 4 hidden layers with 4 nodes (left) own implementation (Right) Scikit-learn implementation

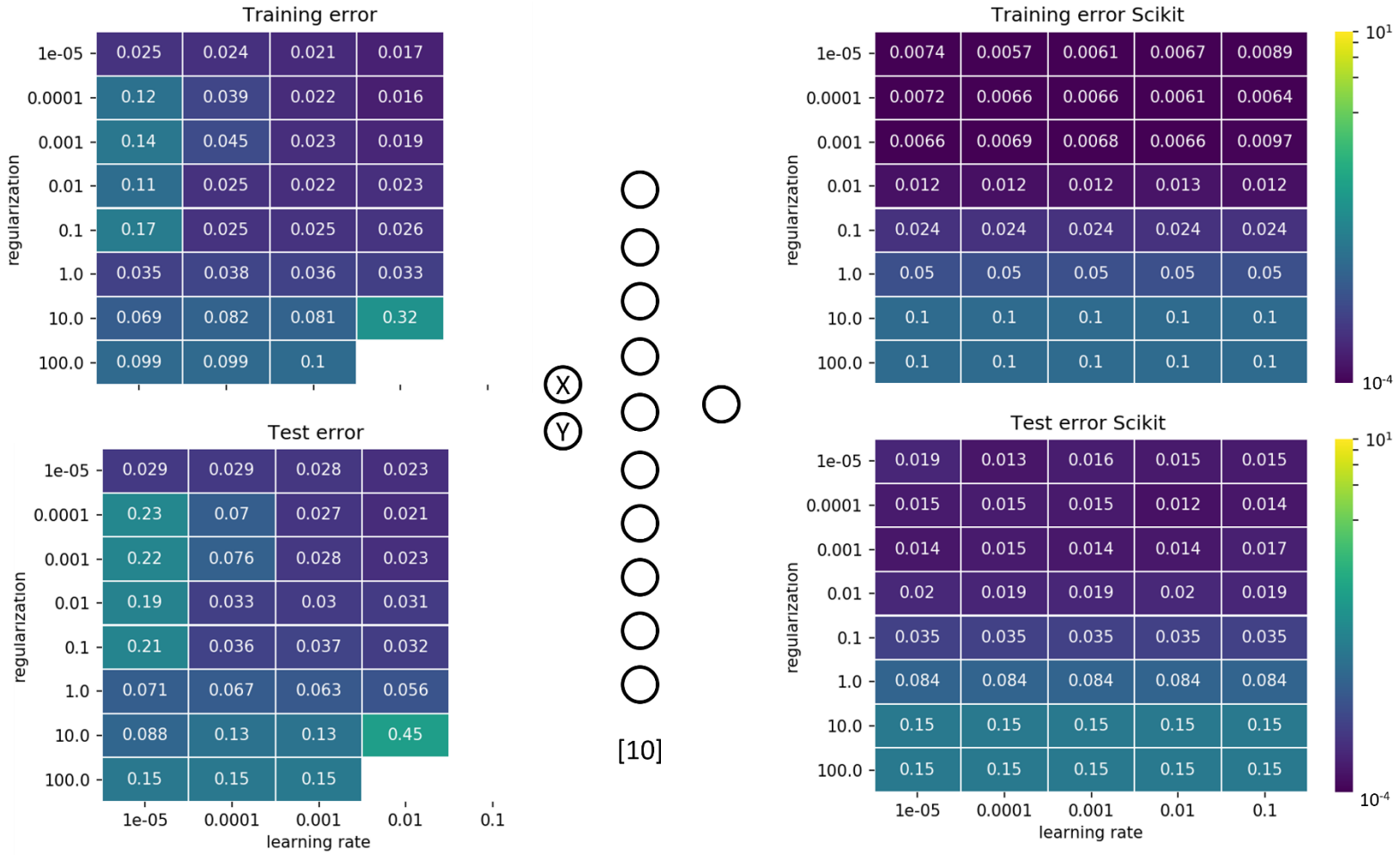

Fig. 11: Grid search plot for the Franke Function regression task with learning rate and regularization as hyperparameters using a FFNN with a single hidden layer and 10 nodes (left) own implementation (Right) Scikit-learn implementation

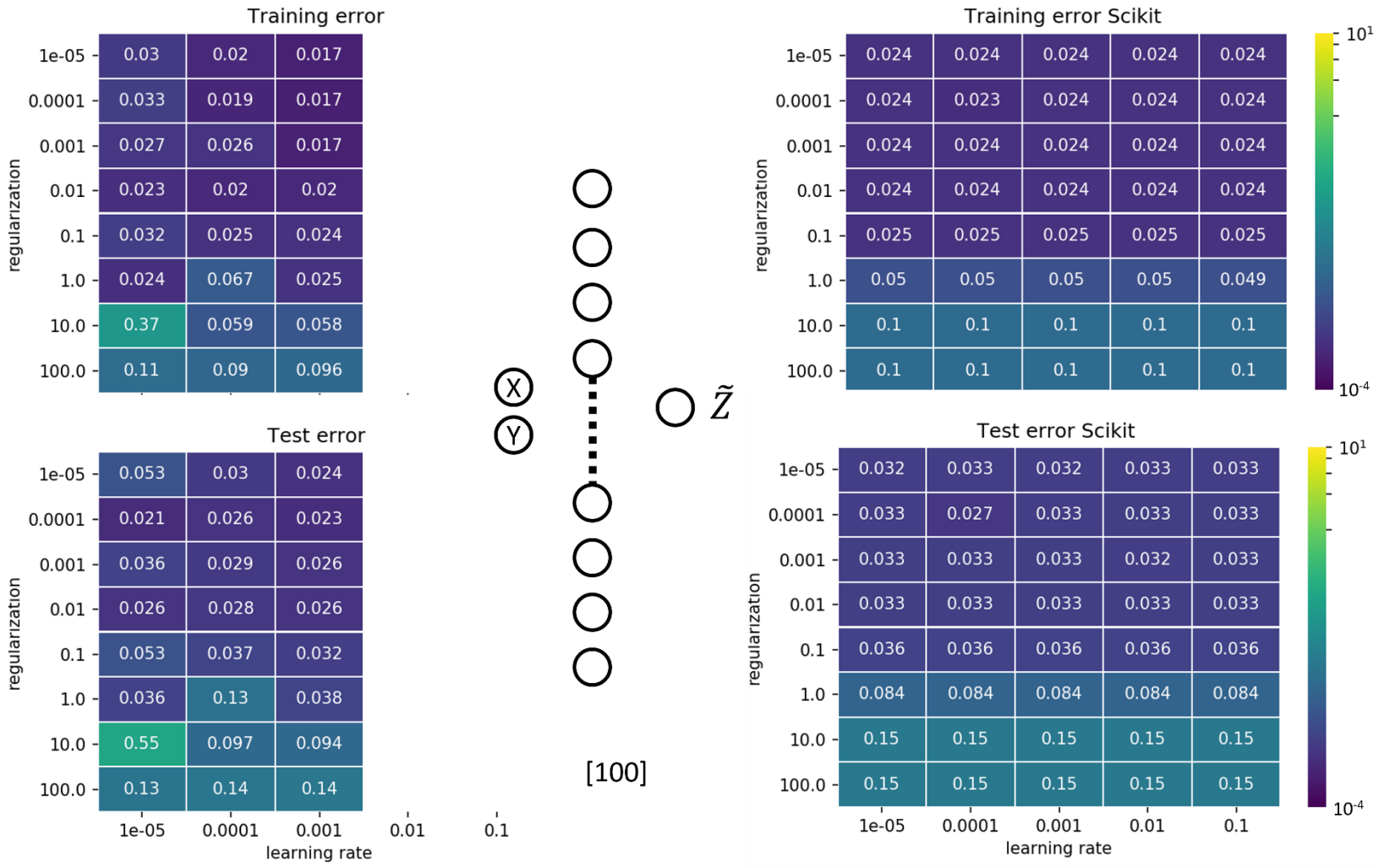

Fig. 12: Grid search plot for the Franke Function regression task with learning rate and regularization as hyperparameters using a FFNN with a single hidden layer and 100 nodes (left) own implementation (Right) Scikit-learn implementation

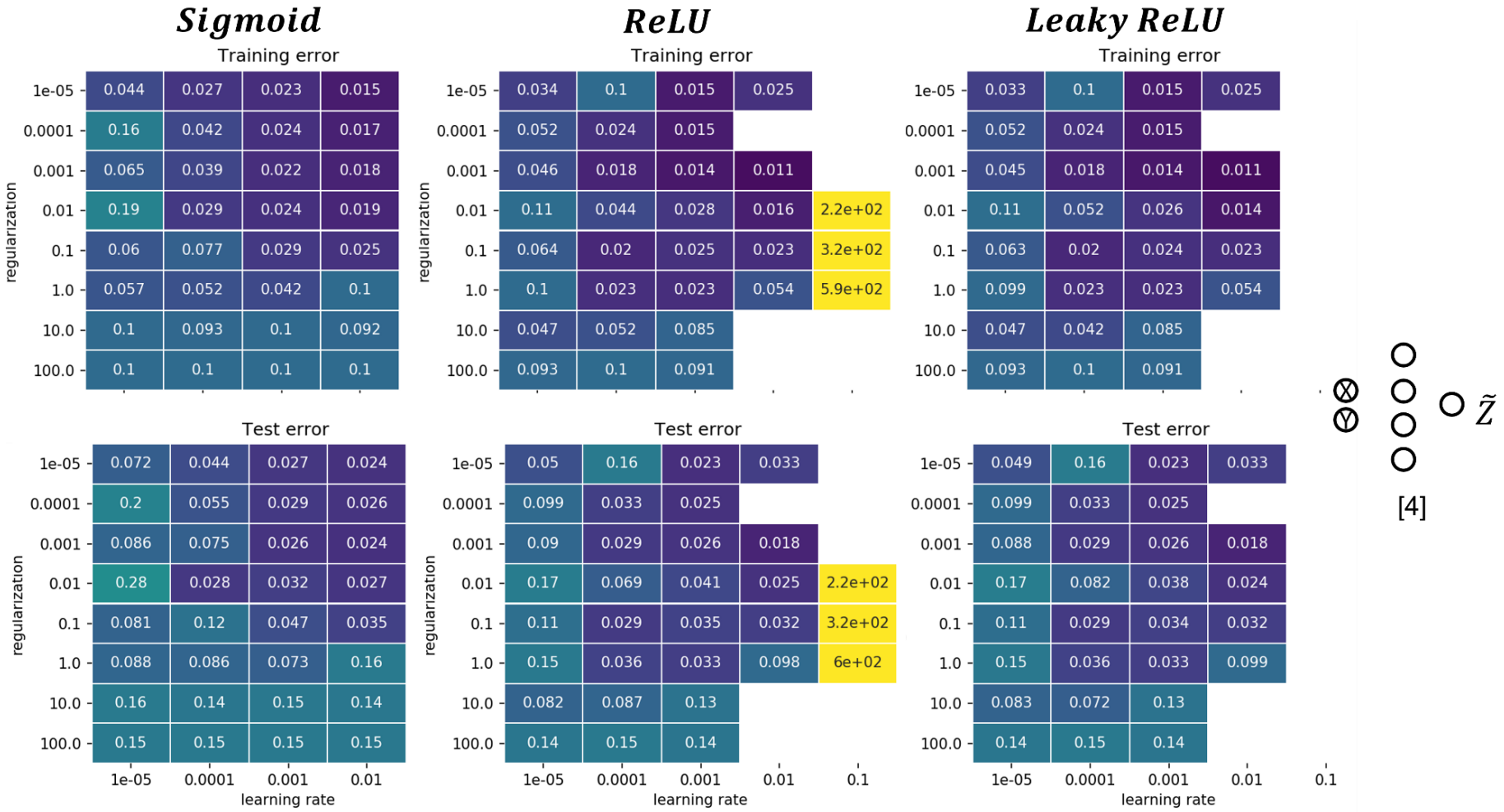

Fig. 13: Grid search plot for the Franke Function regression task with learning rate and regularization as hyperparameters using three activations functions for a FFNN with 4 hidden layers and 4 nodes (left) Sigmoid (middle) ReLU (Right) Leaky ReLU

2.3.2. For a classification task: The Wisconsin Breast Cancer data

The Wisconsin Breast Cancer database is readily available from Scikit-learn database. Although the original database is made of 30 different tumor features, we select here four of them for our classification task. Indeed, some features are evidently correlated like the mean radius and mean perimeter of the tumor (Fig. 2). For our classification task, it is reasonable to select only four tumor features (Mean compactness, mean texture, mean perimeter and symmetry), yielding about 113 data entries for each feature for the training set. Importantly, all data are scaled by subtracting their mean values and dividing by their standard deviation. Without data normalization, we experience that the FFNN training will systematically fail.

Like for the previous regression experiment, we first investigate the effect of the architecture (layers and nodes) on the training of our FFNN. Secondly, we scrutinize the effect of the activation functions. For benchmarking purpose, we compare our "own FFNN implementation" results with the one from scikit-learn.

First, we test a single layer FFNN and vary the number of nodes from 1, 4 to 50 (Figs. 14, 15 and 16). We note that across the range of hyperparameters, the training accuracy score improves when the number of nodes is strictly superior to 1 (up to 100% accuracy for 4 and 50 nodes) while the test accuracy remains mostly unchanged regardless of the number of nodes (within 90% to 92% accuracy). The optimal learning is $\eta=0.1$ and the regularization strength $\lambda < 10^{-2}$.

We increase the depth of the FFNN by adding 3 and 10 hidden layers with 4 nodes each (Figs. 17 and 18). We observe an increased predictive power (92%-93%) of the "3 hidden layer" FFNN (but the training becomes unstable for $\eta=10^{-4}$ and $\lambda>10^{-1}$ (Fig. 17). With 10 hidden layers, the FFNN training fails for all combination of hyperparameters both with own and scikit implementation (Fig. 18).

We retain the FFNN architecture of 3 hidden layers and 4 nodes for subsequent testing of the activation function (Fig. 19). Replacing the hidden layer activation function by Leaky ReLU yield a better training accuracy for $\eta=10^{-4}$ but the training generally fails for $\eta=10^{-1}$. We find that the best prediction accuracy (93%) with our implementation of the FFNN is obtained for the Leaky ReLU (hidden layers) and tanh function (output layer) with $\eta=10^{-3}$ and $\lambda=10^{-2}$. We have tested the Softmax activation function, but the FFNN training failed for each experiment maybe due to a numerically unstable implementation.

We summarize the observations and describe the optimal set of parameters for the classification task using our own implementation of FFNN for four features of the Breast Cancer dataset.

i. The optimal FFNN architecture is three hidden layers containing four nodes each
ii. The optimal activation function is a combination of Leaky ReLU (hidden layers) and tanh (output layer)

iii. The best prediction accuracy score is 93% and obtained for a learning rate $\eta=10^{-3}$ and regularization parameter $\lambda=10^{-2}$

Code used to produce the plots (section 3.3.2):

Grid search:
BreastCancerData_NN_classification_leakyReLU_sigmoid.py
BreastCancerData_NN_classification_leakyReLU_tanh.py
BreastCancerData_NN_classification_sigmoid_sigmoid.py

Neural Network class definition:
NeuralNetwork_classification_sigmoid_sigmoid.py
NeuralNetwork_classification_leakyrelu_sigmoid.py
NeuralNetwork_classification_leakyrelu_tanh.py
NeuralNetwork_classification_sigmoid_softmax.py

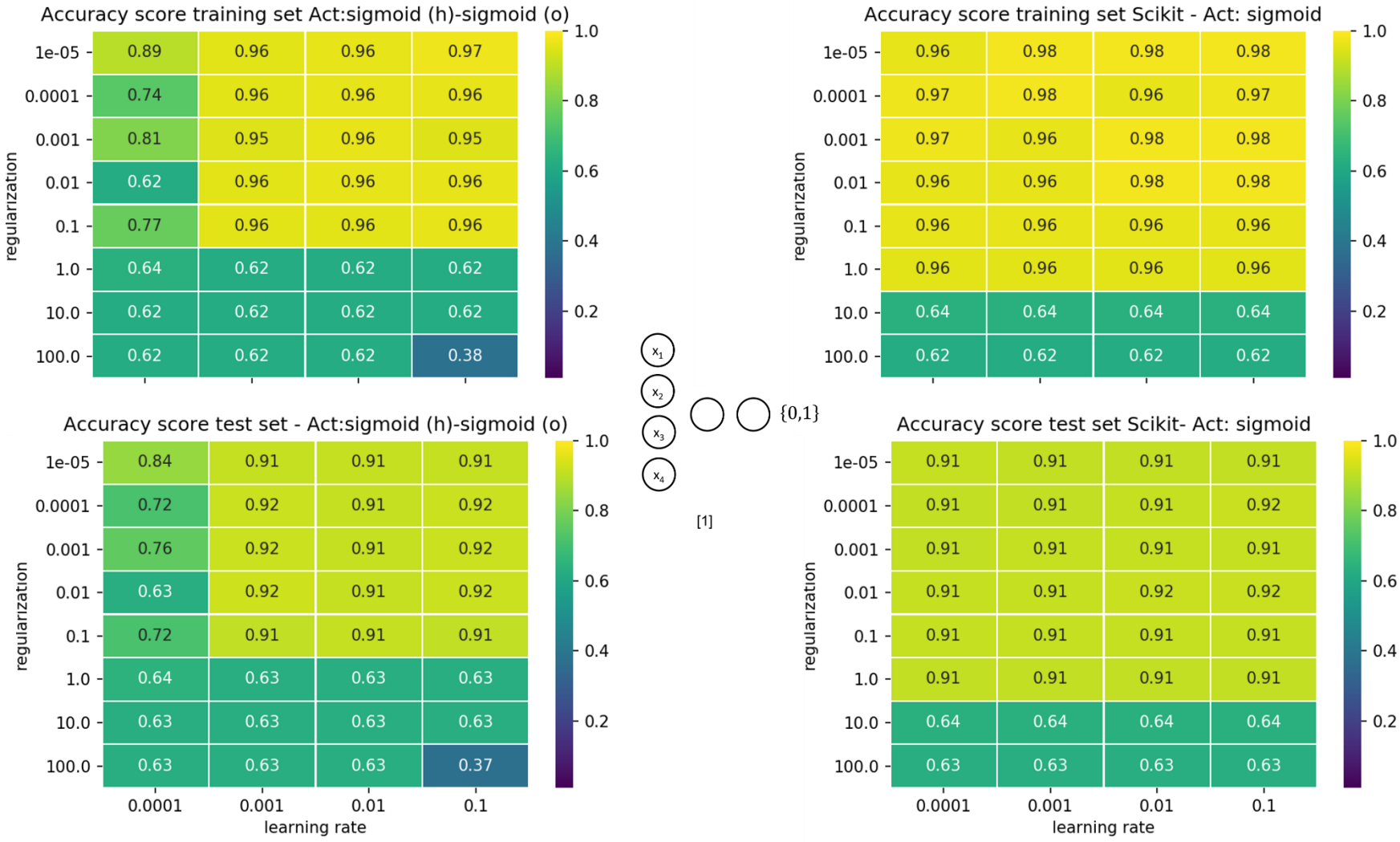

Fig. 14: Grid search plot for the Wisconsin Cancer data classification task with learning rate and regularization as hyperparameters using a FFNN with a single hidden layer and 1 node (left) own implementation (Right) Scikit-learn implementation

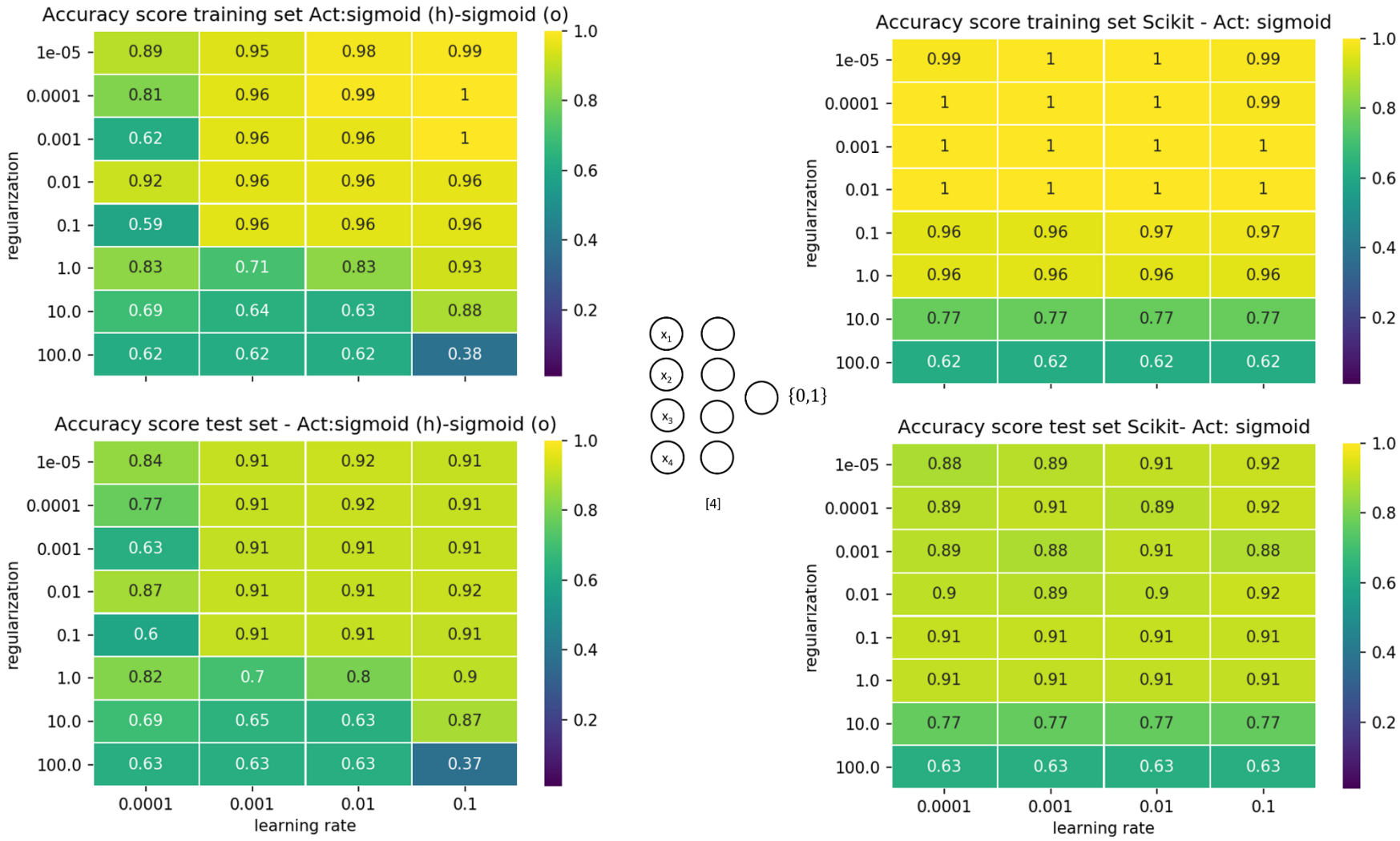

Fig. 15: Grid search plot for the Wisconsin Cancer data classification task with learning rate and regularization as hyperparameters using a FFNN with a single hidden layer and 4 nodes (left) own implementation (Right) Scikit-learn implementation

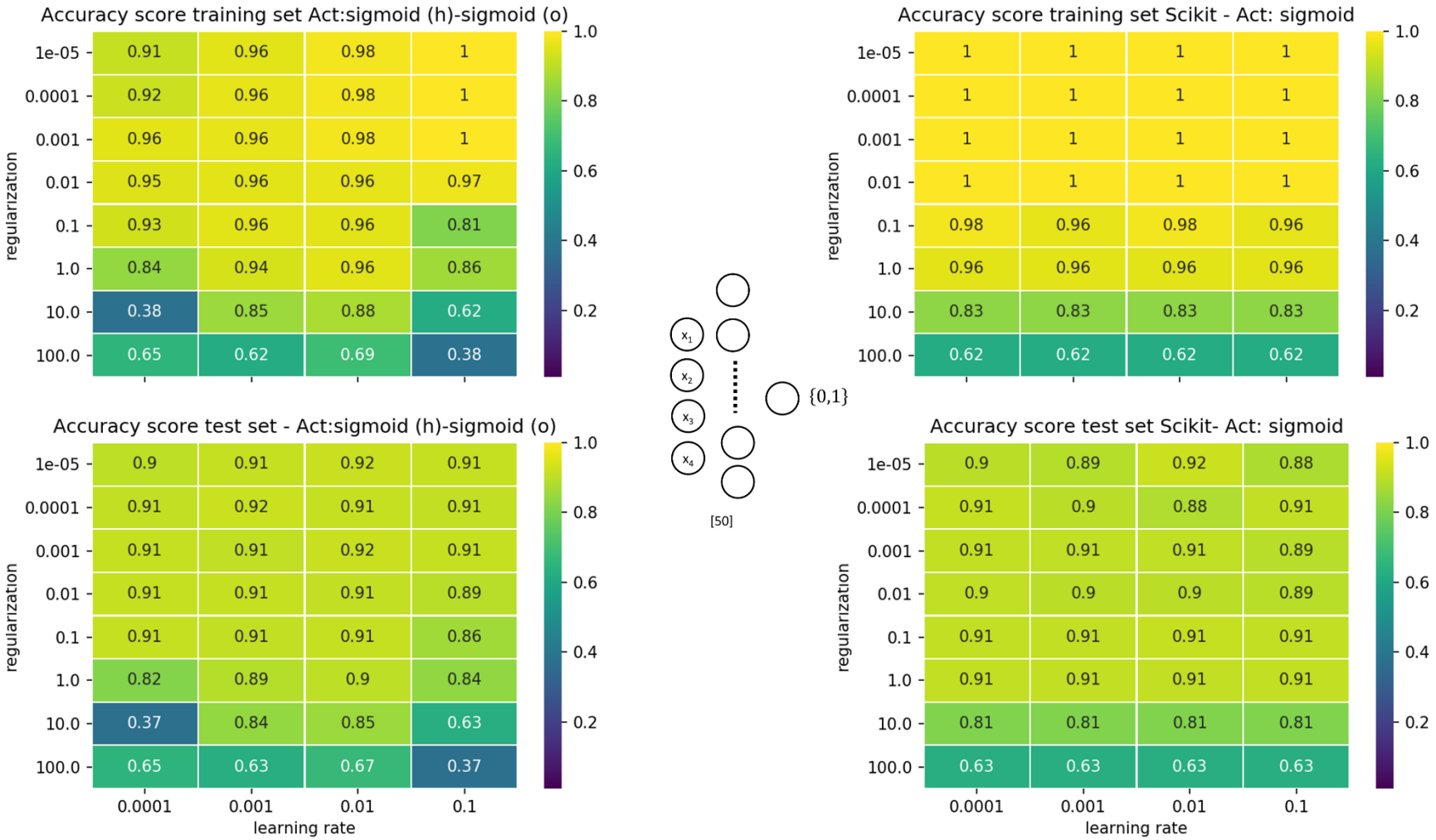

Fig. 16: Grid search plot for the Wisconsin Cancer data classification task with learning rate and regularization as hyperparameters using a FFNN with a single hidden layer and 50 nodes (left) own implementation (Right) Scikit-learn implementation.

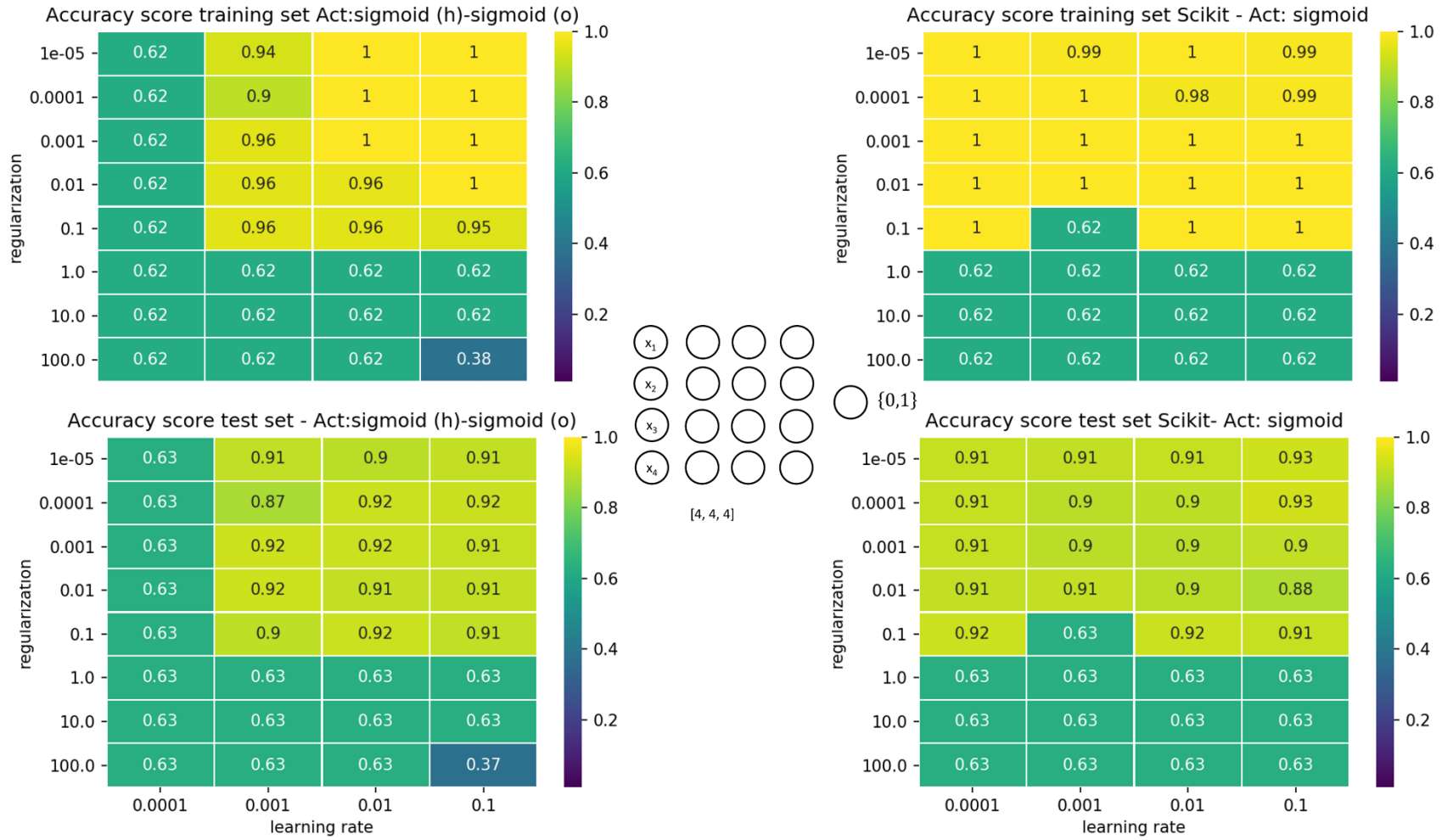

Fig. 17: Grid search plot for the Wisconsin Cancer data classification task with learning rate and regularization as hyperparameters using a FFNN with 4 hidden layers containing 4 nodes each (left) own implementation (Right) Scikit-learn implementation.

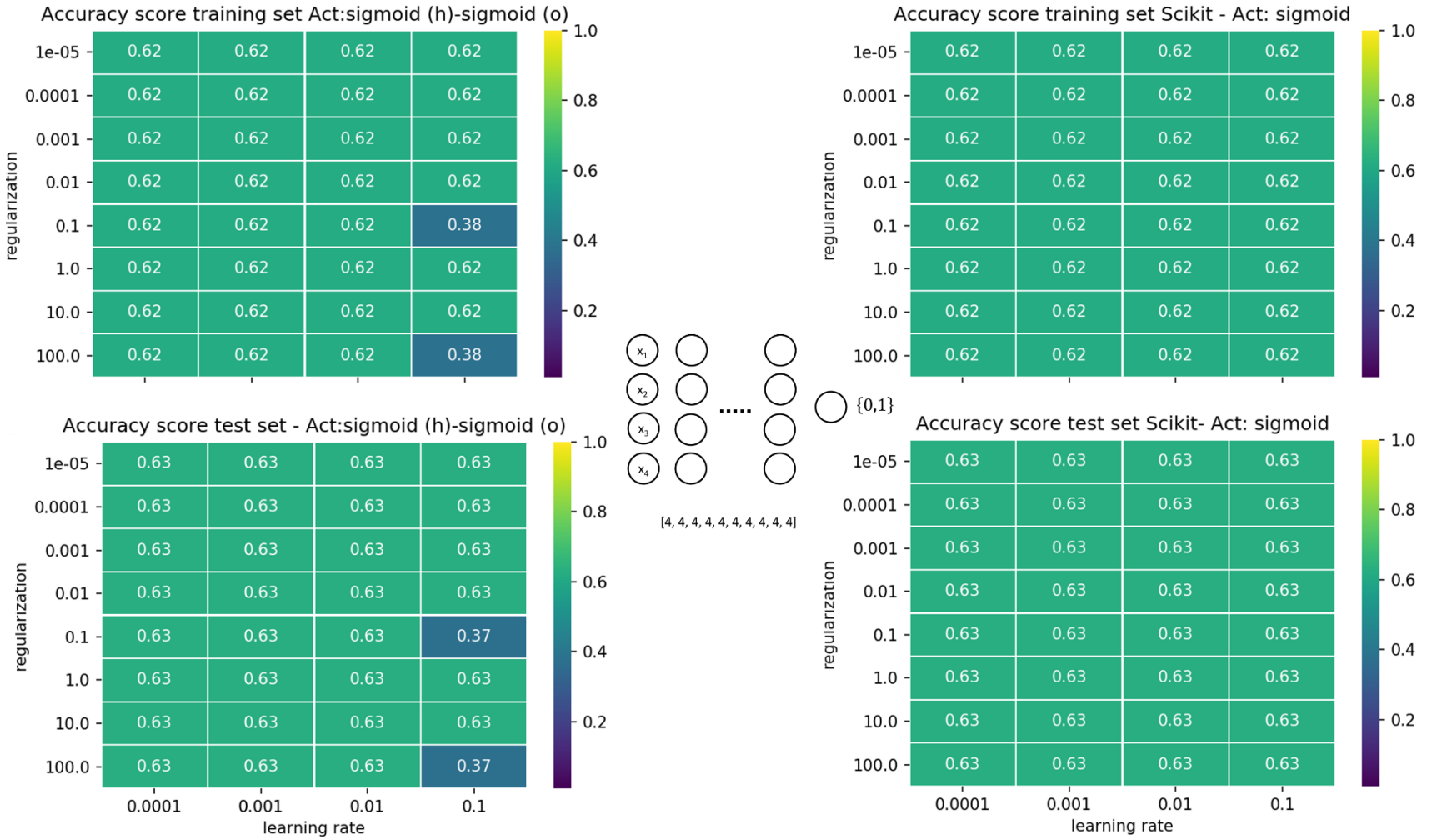

Fig. 18: Grid search plot for the Wisconsin Cancer data classification task with learning rate and regularization as hyperparameters using a FFNN with 10 hidden layers containing 4 nodes each (left) own implementation (Right) Scikit-learn implementation.

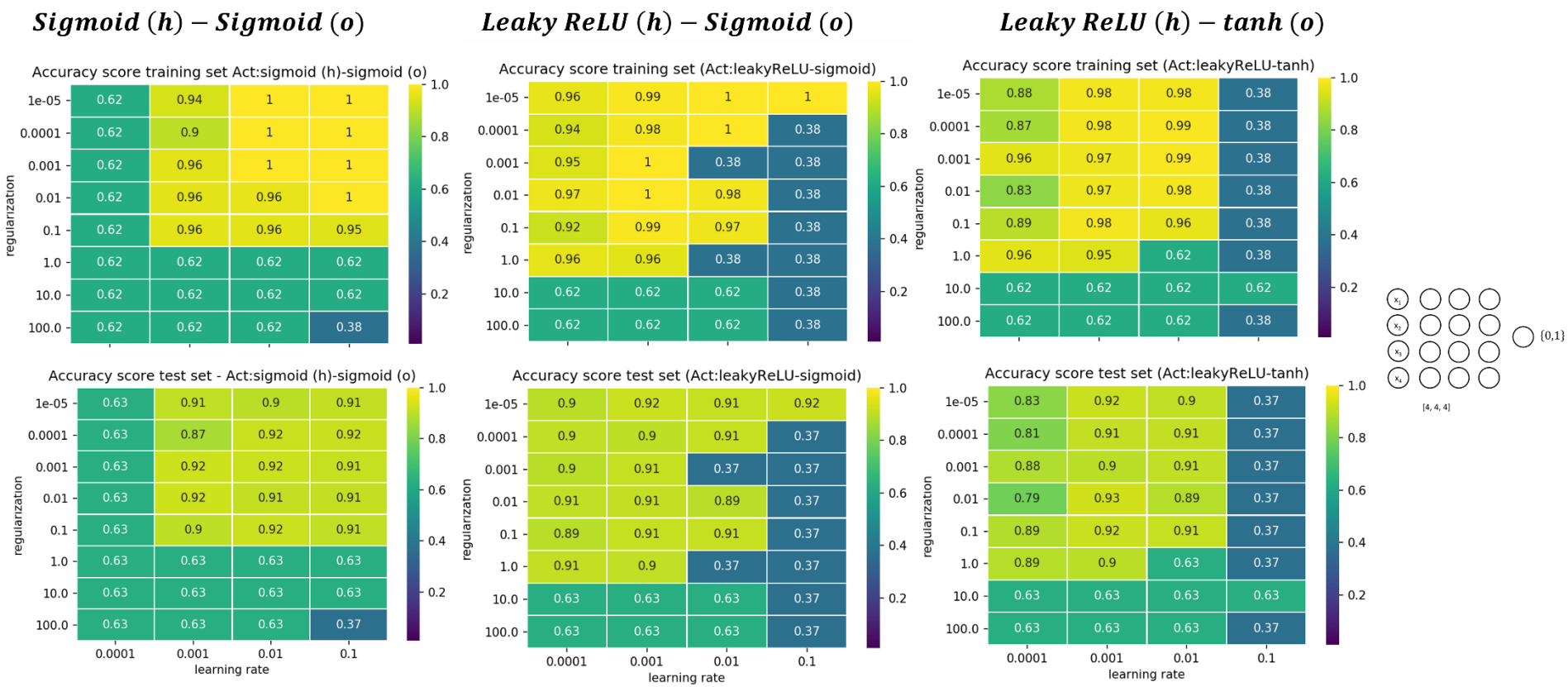

Fig. 19: Grid search plot for the Wisconsin Cancer data classification task with learning rate and regularization as hyperparameters using a FFNN with 4 hidden layers containing 4 nodes each using (left) sigmoid as activation function for both hidden and output layers (middle) Leaky ReLU as activation function for the hidden layers and Sigmoid for the output layer (right) Leaky ReLU as activation function for hidden layers and tanh for the output lay

2.4. Logistic regression

Our own implementation of the logistic regression is based on the FFNN implementation. Indeed, if we remove all hidden layers and use the sigmoid function in the output layer, we end up with the logistic regression problem where the weight and bias parameters (4 weights w_1, w_2, w_3, w_4 and one bias b_0) form the optimal β-predictor vector that minimize the cost function.

The results of the classification using logistic regression using four features of the Breast Cancer dataset are presented in Fig. 20, providing a comparison between our own implementation and scikit-learn *LogisticRegression()* function.

We find that for the optimal set of hyperparameters (learning rate $\eta=10^{-4}$ and regularization parameter $\lambda=10^{-2}$), we obtain the best training and prediction accuracy of respectively 95% and 90% (own implementation) and 96% and 91% (scikit). In this case, the results shows that our implementation marginally underperforms compared to built-in function of scikit-learn.

Code used to produce the plots (section 3.4):

Grid search:
BreastCancerData_LogisticReg.py

Class definition:
LogisticReg.py

Fig. 20: Grid search plot for the Wisconsin Cancer data classification task with learning rate and regularization as hyperparameters (left) Logistic regression Numpy implementation (Right) Scikit-learn logistic regression implementation

3. Discussion and conclusions

3.1. GD techniques

By testing several GD techniques on a simple bi-variate polynomial function, we can list the various improvements in convergence time using the plain GD as a reference by implementing:

i. Randomness in the choice of the training set (SGD): +60%
ii. Momentum: +30%
iii. Tuning the learning rates with ADAgrad or RMSprop: +40%
iv. Tuning the learning rates with ADAM: +30%

However, we do not expect this list to apply to the optimization of more complex, multi-dimensional cost functions. We also anticipate that the various tuning methods for the learning rates would also perform differently depending on the size of the dataset.

3.2. Comparison of FFNN and linear regression performance for the Franke Function data regression task

Overall, the FFNN performs marginally better than the OLS and Ridge regression technique as the best predictive power is obtained for one hidden layer with 4 nodes FFNN with a test MSE of 0.013 (scikit) and 0.018 (own implementation). For the OLS, we found that a 4^{th} order degree polynomial yields a MSE test = 0.016 (Fig. 3 in project 1 report).

FFNN seems to be more robust with respect to overfitting than linear regression as we observe the depth and width of the network only start to impact the prediction error for an exaggerated number of nodes (100 nodes in Fig. 12). Our opinion is that the numerical stability of the training seems to be a more salient issue than overfitting for FFNN as opposed to OLS. This is especially noticeable when the learning rates goes beyond η=0.01 (Figs 8 to 13). In our regression case, the instability of the FFNN training is often caused by exploding gradients values, a common issue for deep learning method like FFNN.

The ReLU and leaky ReLU activation functions are preferred.

3.3. Comparison of FFNN and Logistic regression performance for Wisconsin Breast cancer classification task

For the classification task presented in this report, we find that the FFNN performs better than the logistic regression model both in terms of training and prediction accuracies: 100% for training and 93% tests scores for FFNN and 96% for training, 91% for tests accuracy for logistic regression.

We note that the scikit-learn implementation of FFNN and Logistic regression performs only marginally better than our own implementation for the classification problem!

Like for the regression task, we observe that FFNN are robust with respect to overfitting but are subject to more numerical instability as the architecture of the network becomes wider and deeper. On Fig. 18, we clearly see that a deeper learner (10 hidden layers) fails.

Regarding the activation function, the experiments in Fig. 19 shows that ReLU and leaky ReLU are best suited as hidden layer activation functions and that tanh should be favored over sigmoid for the output layer.

3.4. Tackling FFNN instability, exploding and vanishing gradients

3.4.1. Numerically stable implementation of activation functions

We notice that the way the activation function is implemented has a crucial impact on the stability of the FFNN. As an example, below, we show first the “crude” and numerically stable implementation of the sigmoid function in python:

Numerically unstable for FFNN:

```
def sigmoid(X):
    return 1. / (1. + np.exp(-X))
```

Numerically stable for FFNN:

```
def sigmoid(X):
    if X.all()>=0:
        z = np.exp(-X)
        return 1. / (1. +z)
    else:
        z = np.exp(X)
        return z / (1. + z)
```

Implementing the numerically stable version of the sigmoid function was required to stabilize the training and avoid exploding gradients. We think that our implementation of the softmax function is not stable enough and causes exploding gradient values.

3.4.2. Gradient clipping and weights initialization

As highlighted in the first part of the discussion, exploding gradients are a major issue for FFNN and especially for learning rates $\eta > 0.1$. A possible way to improve numerical stability would be to implement gradient clipping i.e., setting a cap value that will not be overtaken at each iteration.

Another possible way to improve our implementation of FFNN lies in the initialization of the weights (Mishkin and Matas, 2016). The Xavier-initialization of the weights would likely improve both running times and training of our FFNN.

Appendix

1. Source code and test results – GitHub:

https://github.com/rcorseri/UiO/tree/main/Project2

References

Bishop, C. M. (2007). Pattern Recognition and Machine Learning (Information Science and Statistics). *Springer. ISBN: 0387310738*

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*. 307

Franke, R., (1979). A critical comparison of some methods for interpolation of scattered data. *Tech. Rep. NPS-53-79-003, Dept. of Mathematics, Naval Postgraduate School, Monterey, Calif.*

Géron, A. (2019). Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems (2nd ed.). *O'Reilly.*

Goodfellow, I., Bengio, Y. & Courville, A. (2016). Deep Learning. *MIT Press*.

Harris, Charles & Millman, K & Walt, Stéfan & Gommers, Ralf & Virtanen, Pauli & Cournapeau, David & Wieser, Eric & Taylor, Julian & Berg, Sebastian & Smith, Nathaniel & Kern, Robert & Picus, Matti & Hoyer, Stephan & Kerkwijk, Marten & Brett, Matthew & Haldane, Allan & Río, Jaime & Wiebe, Mark & Peterson, Pearu & Oliphant, Travis. (2020). Array programming with NumPy. *Nature. 585. 357-362*. 10.1038/s41586-020-2649-2 (Accessed 10 November 2022).

Hjort-Jensen, M. (2022). Applied Data Analysis and Machine Learning. https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/intro.html. (Accessed 10 November 2022)

Kingma, D. P., & Welling, M. (2014). Auto-Encoding Variational Bayes. *International Conference on Learning Representations (ICLR)*.

Mishkin, D., & Matas, J. (2016). All you need is a good init. CoRR, abs/1511.06422.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., and Thirian, B. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning, v. 12, p. 2825-2830*.

Waskom, Michael. (2021). Seaborn: statistical data visualization. *Journal of Open Source Software. 6. 3021*. https://doi.org/10.5281/zenodo.883859 (Accessed 10 November 2022).