

Using Feedforward Neural Networks for Breast Cancer Detection on Few Features

Lars Bogner
University of Oslo
 (Dated: November 3, 2025)

TODO

Contents

I. Introduction	1
II. Methods	1
A. Neural Networks	1
B. Activation Functions	2
C. Loss Functions	2
D. Regularization	3
E. Optimization Algorithms	3
F. Data Splitting Techniques	3
1. Train-Test Splitting	3
2. Out-of-Fold prediction	4
G. Implementation	4
H. Use of AI tools	5
III. Results and Discussion	5
A. Introduction to the Dataset	5
B. Regression on Limited Feature Set	5
1. Hyperparameter Tuning	5
2. Comparison against Linear Regression	6
C. Classification on Regression Output	6
1. Using Logistic Regression for Classification	6
IV. Conclusion	6
References	6

I. INTRODUCTION

Sampling the cell nuclei of tissue extracted using a fine needle aspirate from the breast allows for a very easy separation between benign and malignant tissue. While very good performance is already shown with straightforward approaches, like decision trees [1], we try to improve the efficiency of the detection process in this paper.

Using a very small subset of features which can be obtained using basic image recognition, we enhance them to a larger set of nuclei features using a regression model, based on which the straightforward classification can be continued using the large set of features.

Starting from the features of radius and area from the Wisconsin Breast Cancer Diagnostic dataset [2] we obtain the full set of features as presented in the dataset using an optimized regression model. Using this regression model it will be possible in the future to use standard classifiers while only needing to measure the radius and area of the cell nuclei in diagnosis. In section II

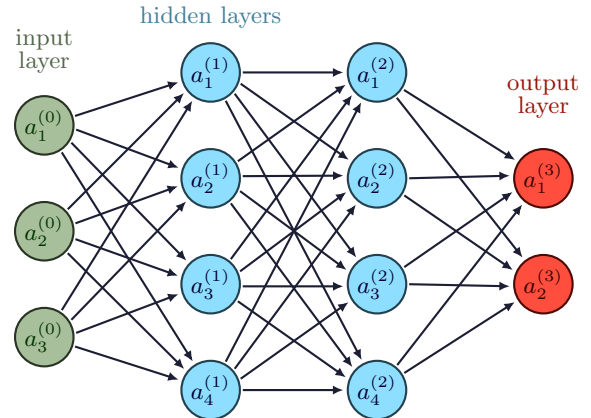


Figure 1: Visualization of a neural network, modified work of [3].

we present the methods necessary to implement the corresponding neural networks and how to optimize them. The results and a discussion thereof is presented in section III. Finally, our findings are concluded in section IV.

II. METHODS

A. Neural Networks

Neural networks are the continuation of a linear combination by introducing nonlinearity. While linear regression uses a linear combination of input features to create an output, neural networks add a nonlinear function to the output of the linear combination. This is the entire secret behind the powerful possibilities neural networks have shown over the last years.

A feedforward neural network (FFNN) uses a layered structure as shown in figure 1, where each node represents a separate linear combination of its inputs. A general node uses the inputs x_i to create a weighted sum z , which is then activated using a so called activation function $g(z)$. More detail on the activation function is provided in section II B. The output of the general node can then be calculated as

$$y = g(z) = g\left(\sum_i (w_i x_i) + b\right), \quad (1)$$

where b is the so-called bias of the node which in turn leads to a shift of the weighted sum, irrespective of the inputs. The linear combination $z = \sum_i (w_i x_i) + b$, can

also be rewritten as a vector-vector multiplication of vectors $\vec{a}_j^{(k)} = (b, \vec{w})$ and $\vec{X} = (1, \vec{x})$. This leads to a simple expression for the output of a node of $y = g(\vec{a} \cdot \vec{X})$. All parameters of a single node in layer k and position j in the layer are contained within $\vec{a}_j^{(k)}$.

We use the term layer, as the output of each layer will be used as the input of the following layer, thus presenting a clear hierarchy. This hierarchy is also useful for calculating the gradient of the final output of the neural network, with respect to all the weights in the layers, as we can use chained differentiation. Calculating the gradient with respect to all the weights is necessary as the optimization techniques presented in section II E rely on the knowledge of the gradients to find the optimal set of weights. The gradient of a single node given the input $\vec{x}$ is given by

$$\frac{\partial y}{\partial \vec{a}} = \vec{X} g'(\vec{a} \cdot \vec{X}), \quad (2)$$

where g' is the derivative of the activation function with respect to its input. Then given the input of a node in the previous layer we can calculate the gradient with respect to its weights using the chain rule of derivatives.

B. Activation Functions

Activation functions provided a source of nonlinearity in the otherwise linear combination of inputs of a neural network. This nonlinearity is key, as it provides the basis for the famous Universal Approximation theorem, which states, that FFNN can approximate any arbitrary real function given a large enough number of nodes [4]. While one might come to the conclusion, that if this nonlinearity is of uttermost importance, the right choice of activation is complicated, it is proven by many practical applications, that already very simple activation functions may provide superb performance.

a. Rectified Linear Unit For simplicity this paper will therefore focus on a set of three activation functions. A very simple non-linear function is provided by the Rectified Linear Unit (ReLU), which shall be defined as

$$\text{ReLU}(z) = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}. \quad (3)$$

While this function provides all necessary properties of an activation function, we will provide an explanation, why it is not well suited for the optimization procedure of FFNN. The issue with the ReLU is that for $z < 0$ the gradient of the function vanishes. This poses a problem, as mentioned above, the optimization algorithm relies on the gradient of the output to adjust and optimize the weights of the neural network. A vanishing gradient disables a possible optimization. This problem is further amplified by the fact, that the gradient of weights in the first layers of the network depends on the product of the

gradients with the previous layers, thus a vanishing gradient in one of the last layers has adverse effects on the optimization of weights in the previous layers.

b. Leaky Rectified Linear Unit As the ReLU is otherwise quite optimal due to its simplicity, we will mainly use a slightly modified version of the ReLU for the majority of nodes in the studied neural networks, namely the Leaky ReLU (LReLU). The leaky ReLU modifies the standard ReLU by introducing a very small but non-vanishing gradient for $z < 0$. It is defined as

$$\text{LReLU}(z) = \begin{cases} \epsilon z & z < 0 \\ z & z \geq 0 \end{cases}, \quad (4)$$

with ϵ being the leak coefficient, a small number, set to $\epsilon = 10^{-5}$ for all studies. Due to the step in the derivative at $z = 0$ our function is still nonlinear, but providing a positive derivative over the whole domain.

c. Softmax The last activation to introduce for our studies will be the Softmax function. It shall be defined as

$$\text{Softmax}(z_j^{(k)}) = \frac{\exp z_j^{(k)}}{\sum_{j'} \exp z_{j'}^{(k)}}. \quad (5)$$

Additionally, to providing a source of nonlinearity, the softmax function has the special property, that the total output of a layer, i.e. the sum of its outputs is regularized to a value of 1. This property is especially useful in the case, where it is used as a classifier, where each of the layers outputs corresponds to a probability of being member in a certain class. While there is no guarantee, that the output of such a layer corresponds to a frequency in the frequentist interpretation, it is very useful in combination with the Cross Entropy introduced in the next section, as we have a nice behaving gradient, useful for optimization.

C. Loss Functions

The function of the loss function is to quantize the discrepancy between the prediction of a numerical model and the true values. It is also sometimes called cost function. It must be a differentiable function which is minimal when, the prediction aligns with the true values. The choice of loss function needs to be specific to the task for which the model should be optimized for. For the regressional part of this paper we will use the mean squared error (MSE) as a loss function. The MSE is defined using the prediction $\hat{y}$ and the true values y as

$$\text{MSE} = \frac{\sum_i (\hat{y}_i - y_i)^2}{N}. \quad (6)$$

Its derivative is therefore defined as

$$\frac{\partial \text{MSE}}{\partial \hat{y}_i} = \frac{2(\hat{y}_i - y_i)}{N}. \quad (7)$$

The advantage of the MSE as a loss function is, that the loss value can be directly interpreted in the context of the metric on which we optimize. Furthermore, it gives the advantage, that compared to the mean absolute error, which uses the absolute deviation between the target and the prediction, that it is continuous differentiable and that large deviations are penalized more than small deviations. Especially in the context of small measurement uncertainties this is important.

The second loss function used is the cross entropy loss. It is used for the case of classification tasks and is defined as

$$\text{CE} = \sum_i y_i \log \hat{y}_i. \quad (8)$$

The cross entropy loss is inspired by the definition of the physical entropy of stochastic systems and is useful, as it is minimal when the probabilities $\hat{y}$ and y align. Combined with the softmax activation function from section II B, the derivative simplifies to a proportional relation with respect to $\hat{y} - y$.

D. Regularization

As the number of parameters in a neural network grows very quickly, special attention needs to be paid to avoid overfitting the training dataset. Overfitting means, that the available degrees of freedom of the model will be used to represent all the datapoints in the training dataset perfectly. This will lead to very poor generalization ability of the model. This phenomenon is part of the bias-variance trade-off, which is ubiquitous in machine learning. With increasing numbers of parameters in the model, i.e. due to an increase in layer size and/or increase in the number of layers, the number of degrees of freedom also increases. With a larger number of degrees of freedom the variance of the model increases. If there are too few parameters, the bias of the model will increase. The goal is therefore to find the middle point, where neither the bias nor variance are increased.

A useful tool to control the bias variance trade-off is regularization in neural networks. Using regularization we can decrease the effective number of degrees of freedom, therefore decreasing the tendency to overfit the data. The decrease is thereby controlled by the regularization coefficient λ . In the limit of $\lambda \rightarrow 0$, the solution approaches the unregularized model, while there is a steady decrease of the number of effective degrees of freedom with an increase of λ . Regularization works by adding a term to the cost function of the form

$$C_{\text{regularized}} = C + \lambda \sum_k \sum_j \|a_j^{(k)}\|_n^n. \quad (9)$$

Equation (9) is the generalized expression for a L_n -regularization, where n is the degree of the norm in equation (9). The most common types of regularization

are L1- and L2-regularization. Using a 1-norm for regularization leads to the benefit that it has the tendency to force an ever-increasing number of parameters of the model towards zero. It thus effectively reduces the number of weights that influence the final result. Especially in highly optimized architectures this may be advantageous, as it reduces the number of computations needed for each evaluation. On the other hand it is known to produce less stable results, as the L2-regularization.

E. Optimization Algorithms

The goal of an optimization algorithm is to tune the weights of a FFNN in a way, so that the cost function will be minimized, i.e. the predictions and the targets align. Most algorithms use a way of gradient descent to update the weights and find the optimum via iterative optimization. Gradient descent uses the derivatives of the cost function

$$\delta_j^{(k)} \equiv \frac{\partial C}{\partial a_j^{(k)}} \quad (10)$$

and update the weights given a learning rate α using

$$a_j^{(k)} \rightarrow a_j^{(k)} - \alpha \cdot \delta_j^{(k)}. \quad (11)$$

To decrease the computational resources needed we will use a more advanced version of this algorithm in this paper. The Adam optimizer updates the weights according to

$$\begin{aligned} m_j^{(k)} &\rightarrow \beta_1 m_j^{(k)} + (1 - \beta_1) \delta_j^{(k)}, \\ v_j^{(k)} &\rightarrow \beta_2 v_j^{(k)} + (1 - \beta_2) (\delta_j^{(k)})^2, \\ \hat{m}_j^{(k)} &= \frac{m_j^{(k)}}{1 - \beta_1^k}, \\ \hat{v}_j^{(k)} &= \frac{v_j^{(k)}}{1 - \beta_2^k}, \\ a_j^{(k)} &\rightarrow a_j^{(k)} - \alpha \frac{\hat{m}_j^{(k)}}{\sqrt{\hat{v}_j^{(k)} + \epsilon}}, \end{aligned} \quad (12)$$

with the optimization parameters $\beta_{1/2}$. As shown in our previous article [5], Adam tends to converge faster towards an optimal solution. It is also thus widely used in machine learning optimizations and will be used for all training procedures throughout the studies.

F. Data Splitting Techniques

1. Train-Test Splitting

As we mentioned before, there is the possibility for our model to overadjust for the datapoints present during the

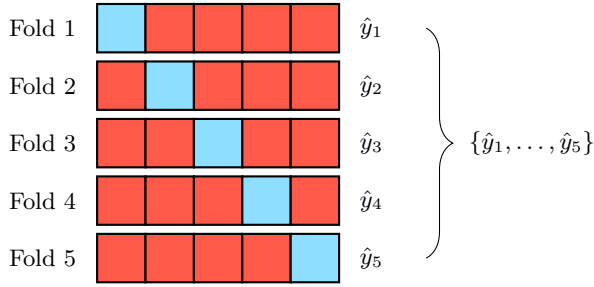


Figure 2: Illustration of predicting the full dataset using out-of-fold prediction to avoid contamination of training data. Modified from [5]. Red represents the training dataset, while the features from the blue part are used to make the prediction. Finally, all predictions are combined into one dataset of the same size as the input dataset.

optimization process. To still get an accurate measure for the models performance we will only use a subset of the dataset for the process of training. A random subset of 20 % is reserved for testing the model only. This means, that all performance metrics are evaluated in the same way as new unseen data would perform. Due to the random nature of the subset, the performance between runs may differ stochastically. This may be counteracted by redoing the tests with different choices of training and testing datasets and averaging the metrics.

2. Out-of-Fold prediction

A further hindrance in our approach with using the output of a regression model to make classifications is, that for an efficient workflow, we need a full but independent prediction of the dataset. To achieve this, we use a workflow inspired by cross-validation. In cross-validation you use multiple folds of the dataset, whereby for every fold, a different part of the dataset is used as testing data, and the rest of the dataset is used for training purposes. This provides independance of training and testing data while being able to test the models general performance for the entire dataset. For out-of-fold prediction, we use the same approach of changing the training data to leave out a specific part each time, but furthermore we use the model trained on each training set, to make predictions for the remaining set. Combining all the predicted sets into one set, allows for having independent predictions for the entire dataset. Having the predictions be independent is important, as otherwise no statement can be made on the generalization ability of the workflow. Using this approach we can expect the same performance on new and yet unseen data.

G. Implementation

All the code necessary to use a FFNN is implemented from scratch in Python3 as the **easynn** library. The source code therefore is located in the `/src/easynn/` folder of the accompanying git repository. It is separated into two main modules, called **schedulers** and **feedforward**. The scheduler modules use modified versions of our previous work on using gradient descent techniques for ordinary least squares techniques. It is also structured in a modular way, so that scheduling tasks like early stopping and dynamic learning rate adjustment are easy to implement in further versions. The base scheduler implements an update method of signature `def update(self, gradients: gradient_type) -> gradient_type:`, with `gradient_type = List[Tuple[np.ndarray, np.ndarray]]`. This can be overwritten to implement any optimization algorithm. The method takes the gradients of the cost functions with respect to the weights and biases of each layer. Each layer is assigned a tuple of two arrays of the gradients. It should then return the updates for each of the weights and biases in the same format. The training process which fetches new updates after each evaluation of the loss and gradients will continue with the `cont` property of the base class is true.

The main structures of the neural network are implemented as classes in the **feedforward** module. Again it is structured in a way, where it is easy to be extended in the future. The FFNN can be constructed using the arguments `class FFNN: def __init__(self, layers: List[Layer], scheduler: Scheduler, loss_fn: LossFunction)`. The layers are constructed using `class Layer: def __init__(self, input_dim: int, num_nodes: int, activation_function: ActivationFunction, regularization: Optional[Regularization] = None)`. Using this structure it is easy to control the hyperparameters of each hidden layer carefully and for example choose the activation and regularization of each layer precisely. The weights and biases of each layer are initialized from a uniform random distribution in the bounds $[-s, s]$ with

$$s = \frac{1}{\sqrt{I \cdot N}}, \quad (13)$$

where I is the input dimension and N is the number of nodes. This initialization is taken from the standard implementation in the widely used **pyTorch** library.

For the entire numerical computation of the neural network, the **numpy** [6] library is used. Numpy is well-known for its performance in linear algebra performance and vectorization of array computations. Other libraries are solely used for plotting purposes. The libraries used during the plotting procedure include **pandas** [7], **scikit-learn**[11] [8], **matplotlib** [9] and **seaborn** [10].

To check the functionality of the developed code, in a first stage, the loss history is evaluated visu-

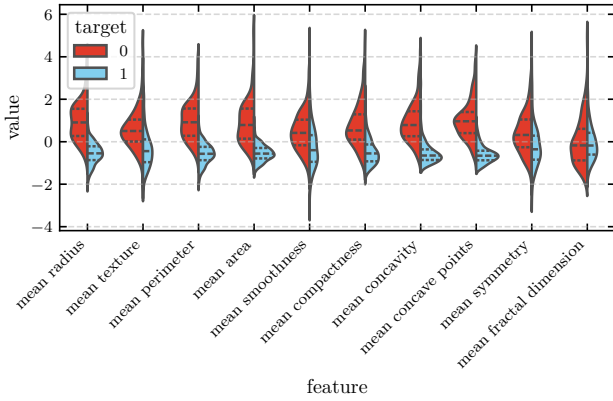


Figure 3: Violin plot of the distribution of all features in the dataset, separated by benign and malignant samples. A target value of 0 corresponds to malignant samples, while a target value of 1 corresponds to benign samples.

ally showing a clear convergence after a high enough number of epochs. Furthermore, the quality of the models is evaluated using toy-datasets created using `scikit-learn`. In detail, both a regression toy-dataset as well as a classification dataset are validated. A MSE of 0.000631 on a dataset created from `make_regression(n_samples=1000, n_features=5, n_informative=3, noise=0.1)` is achieved. For classification the `make_classification(n_samples=1000, n_features=5, n_informative=3, n_classes=2)` function is used achieving a testing accuracy of 0.944. For the classification task a one-hot encoding is used. No hyperparameter tuning was used for the toy-datasets. The performance is assessed to be more than sufficient. Similar performance would be expected using a widespread machine learning framework.

H. Use of AI tools

During the writing of this paper artificial intelligence in the form of large-language models (LLMs) has been used. Two different LLMs have been used for a faster writing of the code used. Mainly for short rewrites of code structure and brain storming the model GPT-5 by OpenAI and the model DeepSeek by Hangzhou DeepSeek Artificial Intelligence have been used. At no point was artificial intelligence used for writing the report. All output by LLMs has been checked by the author.

III. RESULTS AND DISCUSSION

A. Introduction to the Dataset

For the entire study we use the dataset presented by Wolberg [2]. It includes features recorded from nuclei in

breast tissue and their geometric properties. Each physical feature is present three times, where the means and standard deviations of the measured physical features are used as separate features. Furthermore, the mean of the three largest measurements of each physical feature is present in the dataset. The physical features include: the radius, the standard deviation of the images gray-scale values (called texture), the perimeter, the area and the local variation in the radius length, called smoothness. Additionally, the concavity, the number of concave points, the compactness, the symmetry and the fractal dimension are present. For normalization, all features are rescaled to be centered around 0 and have a standard deviation of 1. The normalization is achieved using the `StandardScaler` class of the `scikit-learn` library. A quantitative overview on the average of each of the physical features is presented in figure 3. The violinplot shows the feature distribution for both classes of benign and malignant cell nuclei.

The separability of the two classes is easily visible given all features in the class. On the other hand, there is no single feature which allows for very good separation. Thus reproducing separability using only a limited set of features is the challenge of this study.

B. Regression on Limited Feature Set

We use a FFNN with six input nodes and hidden layers of equal node count. For the output we use a layer using linear activation $g(z) = z$, i.e. no activation function or in other words a linear regression. To measure the performance of the model, the MSE of the models outputs is used as a loss metric. The model is fitted using the Adam optimizer algorithm with a learning rate of $\lambda = 10^{-2}$ and decay parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The mean squared error per target variable is evaluated. For a simple model without any hyperparameter optimization the MSE is in the range of $2 \times 10^{-4} \sigma^2$ to $2 \times 10^1 \sigma^2$, where σ signifies the standard deviation of the feature values.

1. Hyperparameter Tuning

As mentioned in section II B, the Leaky ReLU combines the disadvantages of the ReLU while conserving non-vanishing gradients on the entire domain. Thus it is not surprising, that they perform similar in our studies with the LReLU providing a more stable training process. Due to this finding we will use the LReLU for all of the hidden layers. Figure 4 shows the resulting MSE for different numbers of hidden layers and numbers of nodes in the layers. We can observe a general trend of decrease in MSE with a decrease in the number of hidden layers and an increase in the number of neurons per hidden layer. This general trend is to be expected as with an increase in the number of hidden layers the gradients in the last layers can converge towards zero with deep networks due

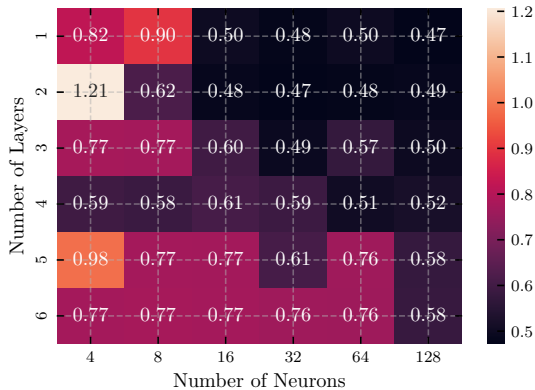


Figure 4: $\log_{10}$ MSE for different numbers of hidden layers in the regression network and nodes per hidden layer.

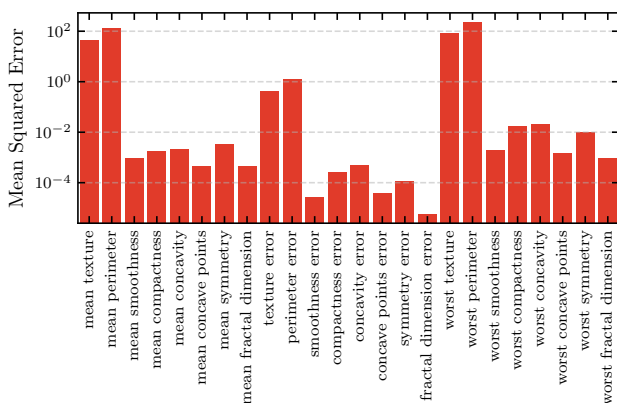


Figure 5: Average MSE of the predicted features using the regression model by feature label.

to the multiplicity nature of the gradients. Additionally, can an increase number of nodes per layer better capture even complicated relationships in the data. Due to the adverse effects of an increase in node number, we chose a conservative set of parameters with 32 neurons per hidden layer and two hidden layers.

In a second substudy we observe the influence of the regularization constant with and L2-regularization on all layers. Further studies on variable regularization per layer were not carried out. The studies show a steady MSE for a range of 0 to 10^{-3} with a sharp increase by a factor of 10 with the step to $\lambda = 10^{-2}$. Thus, we choose to use $\lambda = 10^{-3}$ for the regression model. This provides a stable insurance against overfitting of the training data while at the same time not negatively affecting the mod-

els' generalization ability.

The final set of hyperparameters is then again tested for MSE on all of the features of the dataset it is optimized to predict. The results of the per target MSE are displayed in figure 5, while we see an increase in the upper bound of the per target MSE, to $10^2 \sigma^2$, in comparison to the unoptimized model, the general trend for the MSE shows a lower baseline. The lowest accuracy show the predictions for the texture and perimeter. Both show an MSE of $100 \sigma^2$ in the mean and worst values. Especially the perimeter values are surprising, as one would expect the perimeter to follow from a simple relation with the nuclei's radius. All the other physical features behave similar with respect to the prediction error of $10^{-3} \sigma^2$. In general the prediction uncertainties are smaller on the error values of the physical features.

2. Comparison against Linear Regression

C. Classification on Regression Output

1. Using Logistic Regression for Classification

- Present your results
- Give a critical discussion of your work and place it in the correct context.
- Relate your work to other calculations/studies
- An eventual reader should be able to reproduce your calculations if she/he wants to do so. All input variables should be properly explained.
- Make sure that figures?? and tables contain enough information in their captions, axis labels etc. so that an eventual reader can gain a good impression of your work by studying figures and tables only.

IV. CONCLUSION

- State your main findings and interpretations
- Try to discuss the pros and cons of the methods and possible improvements
- State limitations of the study
- Try as far as possible to present perspectives for future work

[1] K. Bennett, Proceedings of the 4th Midwest Artificial Intelligence and Cognitive Science Society Conference,

Utica, Illinois (1992).

[2] O. M. William Wolberg, *Breast Cancer Wisconsin (Di-*

- agnostic*) (1993).
- [3] *Neural networks – TikZ.net* (2024).
 - [4] M. Elstner and T. Kubar, *Lecture: Machine Learning for Chemistry* (2025).
 - [5] L. Bogner (2025).
 - [6] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al., *Nature* **585**, 357 (2020).
 - [7] T. pandas development team, *Pandas-dev/pandas: Pandas*, Zenodo (2025).
 - [8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al., *Journal of Machine Learning Research* **12**, 2825 (2011).
 - [9] J. D. Hunter, *Computing in Science & Engineering* **9**, 90 (2007).
 - [10] M. L. Waskom, *Journal of Open Source Software* **6**, 3021 (2021).
 - [11] Solely used for data splitting and performance metrics.