

Regression Analysis and Resampling Methods

Lars Bogner

<https://github.uio.no/larsbog/FYSSTK-Project1>

Advances in artificial intelligence/machine learning methods provide tools that have broad applicability in scientific research. These techniques are being applied across the diversity of nuclear physics research topics, leading to advances that will facilitate scientific discoveries and societal applications. This review provides a snapshot of nuclear physics research which has been transformed by artificial intelligence and machine learning techniques.

CONTENTS

I. Introduction	1
II. Methods	1
A. Theoretical Background	1
1. Bias-Variance Tradeoff and Regularization	2
2. Numerical Optimization	2
3. Stochastic Gradient Descent	3
4. Resampling Methods	3
B. Software Tools and Implementation	4
1. Programming Language and Libraries	4
2. Other Software Tools	4
3. Implementation Details	4
III. Results	5
A. Influence of the Number of Parameters on Model Performance	5
1. Ordinary Least Squares	5
2. Regularization Techniques	5
B. Stochastic Gradient Descent	6
C. Alternative Optimization Algorithms	7
D. Bias-Variance Tradeoff and Resampling Techniques	7
IV. Conclusion	7
V. Perspective	7
References	7

I. INTRODUCTION

Optimizing numerical models to model a set of data has been a challenge of the natural sciences for many decades. But with the introduction of artificial neural networks in all disciplines over the last years a special focus on those optimization techniques is relevant.

In this paper different methods of such optimization are studied in the context of large datasets and big data with possibly noise-affected input data. To render the best possible fits a multitude of cost functions, minimization algorithms and other approaches to reduce the computation associated with the optimization process are studied and evaluated.

In the following section the theoretical background of the methods used in this paper is presented. In section II secondly the implementation of these methods is discussed. The results of applying these methods to different datasets are presented in section III. Finally, a

conclusion is drawn in section IV and an outlook on possible future work is given in section V.

In a first step, the dependence of the bias-variance tradeoff on the number of degrees of freedom in a model is studied. For this purpose, polynomial models of different degrees are fitted to data generated from the Runge function with significant noise. The results are then compared to different Ridge and Lasso models to study the effect of regularization on the bias-variance tradeoff. In the context of big data, the performance of different numerical minimization algorithms is studied. For this purpose, the performance of gradient descent, stochastic gradient descent and different optimization algorithms such as Adam are compared in the context of Ridge and Lasso regression. Furthermore the possible advantages and disadvantages of stochastic gradient descent techniques are evaluated. Finally, the effect of resampling methods such as bootstrapping and k-fold cross-validation on the bias-variance tradeoff is studied.

II. METHODS

A. Theoretical Background

To find the optimal solution to a linear model in the form

$$X\vec{\theta} = \vec{y}, \quad X \in \mathbb{R}^{n \times p}, \vec{y} \in \mathbb{R}^n, \vec{\theta} \in \mathbb{R}^p, \quad (1)$$

the ordinary least squares (OLS) method can be used. The OLS method minimizes the squared deviation between the model prediction and the actual data. This is done by minimizing the cost function

$$C_{\text{OLS}}(\vec{\theta}) = \frac{1}{n} \left\| X\vec{\theta} - \vec{y} \right\|_2^2. \quad (2)$$

This method has been widely used in the past ultimately also because it has an analytical solution in the case that the matrix product $X^T X$ is invertible, i.e. $\det(X^T X) \neq 0$. In this case the optimal parameters $\vec{\theta}_{\text{OLS}}$ can be found by solving the normal equations

$$\vec{\theta}_{\text{OLS}} = (X^T X)^{-1} X^T \vec{y}. \quad (3)$$

Taking the derivative of the cost function in eq. (2) with respect to the parameters $\vec{\theta}$ and inserting the solution in eq. (3) shows that this is indeed a minimum of the cost function:

$$\frac{\partial C_{OLS}}{\partial \vec{\theta}} \Big|_{\vec{\theta}_{OLS}} = \frac{2}{n} X^T (X\vec{\theta} - \vec{y}) \Big|_{\vec{\theta}_{OLS}} = 0. \quad (4)$$

1. Bias-Variance Tradeoff and Regularization

The cost function of the ordinary least squares method can also be identified as the mean squared error (MSE) between the model prediction and the actual data

$$\text{MSE}(\vec{y}, \vec{\tilde{y}}) = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2 = \mathbb{E}[(y - \tilde{y})^2], \quad (5)$$

where $\vec{\tilde{y}} = X\vec{\theta}$ is the model prediction. $\mathbb{E}$ denotes the expectation value. The MSE can be decomposed into three components, the squared bias, the variance and the irreducible error σ^2 as

$$\text{MSE}(\vec{y}, \vec{\tilde{y}}) = \text{Bias}^2(\vec{y}, \vec{\tilde{y}}) + \text{Var}(\vec{\tilde{y}}) + \sigma^2, \quad (6)$$

where the bias is defined as

$$\text{Bias}(\vec{y}, \vec{\tilde{y}}) = \mathbb{E}[\vec{\tilde{y}}] - \mathbb{E}[\vec{y}], \quad (7)$$

and the variance as

$$\text{Var}(\vec{\tilde{y}}) = \mathbb{E}[(\vec{\tilde{y}} - \mathbb{E}[\vec{\tilde{y}}])^2]. \quad (8)$$

The irreducible error σ^2 is the variance of the noise in the data and cannot be reduced by any model. The bias-variance decomposition in eq. (6) shows that there is a tradeoff between bias and variance when trying to minimize the MSE. A model with low complexity will have a high bias but low variance, while a model with high complexity will have a low bias but high variance. This is known as the bias-variance tradeoff. The OLS method can lead to overfitting of the data, especially when the number of parameters p is large compared to the number of data points n . In this case, the model will fit the noise in the data rather than the underlying trend, leading to a low bias but high variance. On the contrary, a model with too few parameters will not be able to capture the underlying trend in the data, leading to a high bias but low variance. To mitigate the effects of overfitting, regularization techniques such as Ridge and Lasso regression can be used. These techniques add a penalty term to the cost function that penalizes large values of the parameters $\vec{\theta}$. The cost function for Ridge regression is given by

$$C_{\text{Ridge}}(\vec{\theta}) = \frac{1}{n} \|X\vec{\theta} - \vec{y}\|_2^2 + \lambda \|\vec{\theta}\|_2^2, \quad (9)$$

where λ is the regularization parameter that controls the strength of the penalty term. The cost function for Lasso regression is given by

$$C_{\text{Lasso}}(\vec{\theta}) = \frac{1}{n} \|X\vec{\theta} - \vec{y}\|_2^2 + \lambda \|\vec{\theta}\|_1, \quad (10)$$

where the L_1 norm is defined as $\|\vec{\theta}\|_1 = \sum_{i=0}^{p-1} |\theta_i|$. Both Ridge and Lasso regression can help to reduce the variance of the model by penalizing large values of the parameters $\vec{\theta}$, leading to a different bias-variance tradeoff. For Ridge regression, we can derive the effective number of parameters also called degrees of freedom as

$$n_{\text{effective}} = \sum_{i=1}^{n_{\text{features}}} \frac{d_i^2}{d_i^2 + \lambda}, \quad (11)$$

where d_i are the singular values of the design matrix X . Thus regularization effectively reduces the number of parameters in the model, leading to a bias-variance tradeoff that can be tuned by the regularization parameter λ . While OLS models can be analytically optimized using eq. (3) for the case where $X^T X$ is invertible, the same is true for Ridge regression in the case where $X^T X + \lambda I$ is invertible. The optimal parameters for Ridge regression can be found by solving the modified normal equations

$$\vec{\theta}_{\text{Ridge}} = (X^T X + \lambda I)^{-1} X^T \vec{y}. \quad (12)$$

2. Numerical Optimization

However, for Lasso regression there is no analytical solution and numerical methods must be used to obtain the optimal parameters $\vec{\theta}_{\text{Lasso}}$. All numerical optimization methods are based on the idea of iteratively minimizing the cost function. Around the current guess for the parameters $\vec{\theta}_i$ the cost function can be approximated by a Taylor expansion as

$$\begin{aligned} C_{OLS}(\vec{\theta}) &= C_{OLS}(\vec{\theta}_i) + \nabla C_{OLS}(\vec{\theta}_i)^T (\vec{\theta} - \vec{\theta}_i) \\ &+ \frac{1}{2} (\vec{\theta} - \vec{\theta}_i)^T H(\vec{\theta}_i) (\vec{\theta} - \vec{\theta}_i) \\ &+ \mathcal{O}(\|\vec{\theta} - \vec{\theta}_i\|_2^3), \end{aligned} \quad (13)$$

where $\nabla C_{OLS}(\vec{\theta}_i)$ is the gradient and $H(\vec{\theta}_i)$ the Hessian of the cost function at the current guess $\vec{\theta}_i$. Based on this approximation, the next guess for the parameters can be found by minimizing the quadratic approximation of the cost function. Since the calculation of the Hessian is computationally expensive, it is often neglected in practice. This leads to the gradient descent algorithm, where the next guess for the parameters is given by

$$\vec{\theta}_{i+1} = \vec{\theta}_i - \eta \nabla C_{OLS}(\vec{\theta}_i), \quad (14)$$

Expand
on the
deriva-
tion
here?

where η is the learning rate that controls the step size of the update. The gradient descent algorithm can be further improved by using techniques such as momentum, Adagrad, RMSProp and Adam. These techniques adapt the learning rate based on the history of the gradients, leading to faster convergence and better performance.

a. Adagrad The Adagrad algorithm adapts the learning rate for each parameter based on the history of the gradients. The update rule for Adagrad is given by

$$\begin{aligned} g_i &= g_{i-1} + \nabla C_{\text{OLS}}(\vec{\theta}_i) \odot \nabla C_{\text{OLS}}(\vec{\theta}_i), \\ \vec{\theta}_{i+1} &= \vec{\theta}_i - \frac{\eta}{\sqrt{g_i + \epsilon}} \odot \nabla C_{\text{OLS}}(\vec{\theta}_i), \end{aligned} \quad (15)$$

where g_i is the accumulated squared gradient, $\odot$ denotes the element-wise product and ϵ is a small constant to prevent division by zero.

b. RMSProp The RMSProp algorithm is a modification of Adagrad that aims to reduce its aggressive, monotonically decreasing learning rate. It does this by maintaining a moving average of the squared gradients, which allows it to adapt the learning rate for each parameter more effectively. The update rule for RMSProp is given by

$$\begin{aligned} s_i &= \beta s_{i-1} + (1 - \beta) \nabla C_{\text{OLS}}(\vec{\theta}_i) \odot \nabla C_{\text{OLS}}(\vec{\theta}_i), \\ \vec{\theta}_{i+1} &= \vec{\theta}_i - \frac{\eta}{\sqrt{s_i + \epsilon}} \odot \nabla C_{\text{OLS}}(\vec{\theta}_i), \end{aligned} \quad (16)$$

where s_i is the moving average of the squared gradients, β is a hyperparameter that controls the decay rate of the moving average, and the other symbols have the same meaning as in the Adagrad update rule.

c. Adam The Adam (Adaptive Moment Estimation) algorithm combines the ideas of momentum and RMSProp. It maintains a moving average of both the gradients and the squared gradients, allowing it to adapt the learning rate for each parameter based on both the first and second moments of the gradients. The update rule for Adam is given by

$$\begin{aligned} m_i &= \beta_1 m_{i-1} + (1 - \beta_1) \nabla C_{\text{OLS}}(\vec{\theta}_i), \\ s_i &= \beta_2 s_{i-1} + (1 - \beta_2) \nabla C_{\text{OLS}}(\vec{\theta}_i) \odot \nabla C_{\text{OLS}}(\vec{\theta}_i), \\ \vec{\theta}_{i+1} &= \vec{\theta}_i - \frac{\eta}{\sqrt{s_i + \epsilon}} \odot m_i, \end{aligned} \quad (17)$$

where m_i is the moving average of the gradients, s_i is the moving average of the squared gradients, β_1 and β_2 are hyperparameters that control the decay rates of the moving averages, and the other symbols have the same meaning as in the previous update rules.

3. Stochastic Gradient Descent

As the computation of the gradient $\nabla C_{\text{OLS}}(\vec{\theta}_i)$ can be computationally expensive, especially for large datasets, a common approach is to use stochastic gradient descent (SGD). In SGD, the gradient is approximated by calculating it on a small subset of the data, called a mini-batch. This leads to a noisy estimate of the gradient, but it can significantly reduce the computational cost of each update step. The main challenge with SGD is to choose an appropriate mini-batch size. A small mini-batch size leads to a noisy estimate of the gradient, which can slow down convergence. On the other hand, a large mini-batch size can diminish the benefits of using SGD, as it approaches the full gradient descent computation cost. In practice the optimal mini-batch size depends on the specific problem and dataset.

4. Resampling Methods

To better understand the performance of a model, e.g. in terms of the bias-variance tradeoff, resampling methods such as bootstrapping and k-fold cross-validation can be used. These methods allow to estimate the performance of a model on unseen data by repeatedly splitting the data into training and test sets.

a. Bootstrapping In bootstrapping, multiple datasets are generated by randomly sampling the original training dataset with replacement. Each of these datasets is then used to train a model, and the performance of the model is evaluated on the unmodified test dataset. This process is repeated multiple times, and the performance metrics are averaged to obtain an estimate of the model's performance. Bootstrapping can be used to estimate the bias and variance of a model by analyzing the distribution of the performance metrics across the different bootstrap samples. The number of bootstrap samples is arbitrary, but a thousand samples are used in this work.

b. k-Fold Cross-Validation In k-fold cross-validation, the original dataset is divided into k equally sized folds. The model is then trained on $k - 1$ folds and evaluated on the remaining fold. This process is repeated k times, with each fold being used as the test set once. The performance metrics are then averaged across the k iterations to obtain an estimate of the model's performance. The process of k-fold cross-validation is illustrated in fig. 1. For this work $k = 5$ folds are used.

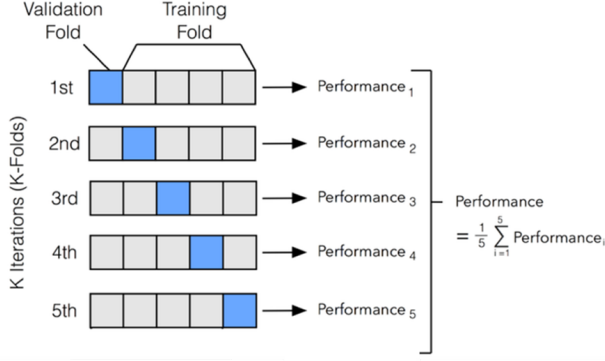


FIG. 1 Illustration of k-fold cross-validation with $k = 5$. The dataset is divided into 5 folds, and the model is trained on 4 folds and evaluated on the remaining fold. This process is repeated 5 times, with each fold being used as the test set once.

B. Software Tools and Implementation

1. Programming Language and Libraries

The methods described in the previous section have been implemented in Python. The implementation is structured in a modular way, allowing to easily switch between different optimization algorithms and resampling methods. The code is available on GitHub at <https://github.uio.no/larsbog/FYSSTK-Project1>. The following libraries have been used in the implementation:

numpy: It is a fundamental package for scientific computing in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. It is used in this project for all matrix and vector operations, as well as for generating random numbers and performing statistical calculations.

scikit-learn: It is a machine learning library for Python that provides simple and efficient tools for data mining and data analysis. It includes implementations of various machine learning techniques, like data manipulation and performance evaluation metrics. It is used in this project for rescaling the data, splitting the data into training and test sets, and for implementing the resampling methods. Furthermore, it provides the implementation of the mean squared error (MSE) metric used to evaluate the performance of the models.

matplotlib: It is a plotting library for Python that provides a wide range of tools for creating different types of plots and visualizations. It is used in this project to visualize the results of the different methods and to create plots for the report.

2. Other Software Tools

The code has been developed using the **Visual Studio Code** editor, which provides a wide range of features for Python development, including syntax highlighting, code completion, and debugging tools. The version control system **Git** has been used to manage the codebase and track changes. The Large Language Model (LLM) **GitHub Copilot** has been used to assist in the development of the code by providing code suggestions and autocompletions based on the context of the code being written. All LLM-generated code has been reviewed and modified as necessary to ensure its correctness and suitability for the project.

3. Implementation Details

To evaluate the performance of the different methods, a synthetic dataset has been generated using the Runge function. The Runge function is defined as

$$f(x) = \frac{1}{1 + 25x^2}, \quad x \in [-1, 1]. \quad (18)$$

The x -values are sampled uniformly in the interval $[-1, 1]$, and the corresponding y -values are generated by adding Gaussian noise with mean 0 and standard deviation $\sigma = 1$, unless otherwise specified. To evaluate the performance in dependence of the degrees of freedom, the x -values are expanded into multiple polynomial features up to a specified degree: $X_i = \{x_i^0, x_i^1, \dots, x_i^p\}$. All features and outputs are then rescaled to have zero mean and unit variance using the **StandardScaler** from **scikit-learn**. The mean and variance of the training data are used for rescaling the test data to avoid data leakage. There are analytical implementations of the OLS and Ridge regression methods, as described in eq. (3) and eq. (12). Furthermore numerical optimizers are implemented from scratch based on a class inheritance structure to reduce code duplication. The base class **GradientDescent** implements a fitting procedure consisting of a precomputation step, an optimization procedure over multiple iterations and a memory of the cost function values. The different optimization algorithms, i.e. vanilla gradient descent, Adagrad, RMSProp and Adam, are implemented as subclasses that override the update rule for the parameters $\bar{\theta}$ in each iteration. The cost functions for OLS, Ridge and Lasso regression are also implemented as separate classes that provide methods to calculate the cost and its gradient. The stochastic gradient descent methods are implemented as further subclasses that modify the data used for fitting in each iteration. In the stochastic gradient descent method, the cost history property is furthermore modified to return the per epoch average of the cost function values. The resampling methods, i.e. bootstrapping and k-fold cross-validation, are implemented as methods with identical

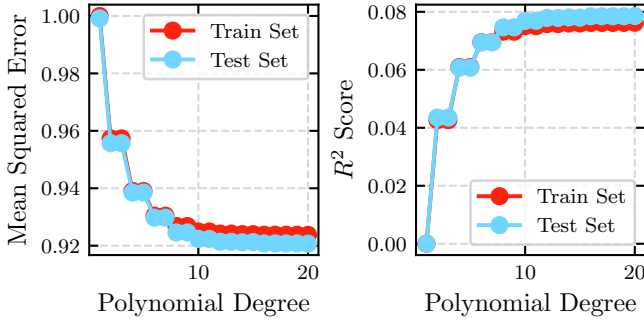


FIG. 2 Mean squared error (MSE) and R^2 score for polynomial fits of different degrees to noisy data from the Runge function using ordinary least squares regression. $N_{\text{train}} = 80\,000$, $N_{\text{test}} = 20\,000$.

interfaces which provide multiple sets of training and test data. This allows to easily switch between the different resampling methods when evaluating the performance of a model.

III. RESULTS

A. Influence of the Number of Parameters on Model Performance

1. Ordinary Least Squares

Even the best optimization algorithm cannot turn a bad model into a good one. If a model is not suited to describe the trend in the data irrelevant of the parameters, it is logical to assume that no optimization algorithm can find a good fit. A common approach to counteract this problem is to increase the number of parameters in a model and use a very flexible model, e.g. a polynomial of high degree. In this section the influence of the number of parameters on the model performance is studied in the context of ordinary least squares (OLS) regression.

How the mean squared error (MSE) and the R^2 score depend on the polynomial degree of the model is shown in fig. 2. As expected, the MSE decreases and the R^2 score increases with increasing polynomial degree. However, it is also visible that for very high polynomial degrees the performance does not improve significantly anymore. Using a dataset with 80 000 samples, no negative side effects of overfitting are visible even for polynomial degrees as high as 20. To highlight the danger of negative impact due to high parameter count it is also interesting to study the parameter values of the fitted models. Figure 3 shows the parameter values for polynomial fits of different degrees to the same dataset. It is visible that for high polynomial degrees the parameter values become very large. Especially it is observable that the parameter values flip sign for every second parameter, i.e. compensating negative effects of one parameter with the next.

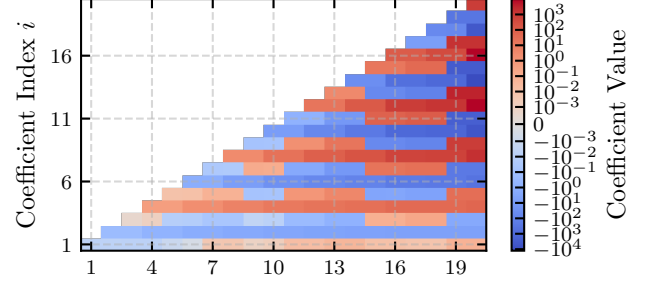


FIG. 3 Parameter values for polynomial fits of different degrees to noisy data from the Runge function using ordinary least squares regression. $N_{\text{train}} = 80\,000$.

While such behavior is not necessarily a problem within the feature space of the training data, it can lead to very bad performance outside of this feature space. This is especially relevant for extrapolation tasks, where the model is used to predict values outside of the feature space of the training data. In such cases, the model can produce completely meaningless results.

In the limit of small datasets this issue becomes even more relevant. As the number of parameters approaches the number of data points, the model can fit the data perfectly, leading to a MSE of zero. However, this is usually not a desirable outcome, as the model will not generalize well to new data, as it has simply “memorized” the noise in the training data. This is a classic example of overfitting, where the model performs well on the training data but poorly on unseen data. In such cases, it is crucial to use techniques to prevent overfitting, such as regularization or cross-validation.

2. Regularization Techniques

To counteract the negative effects of high parameter count, regularization techniques such as Ridge and Lasso regression can be used. These techniques add a penalty term to the cost function that discourages large parameter values. In Ridge regression, the penalty term is proportional to the square of the parameter values, while in Lasso regression, it is proportional to the absolute value of the parameter values. This leads to a trade-off between fitting the data well and keeping the parameter values small. With an increasing regularization strength λ the parameter values are pushed towards zero, as visible in fig. 4. This effect is especially strong for $\lambda > 0$. For very high regularization strengths the parameter values become very small, leading to a model that is almost constant. This is also visible in fig. 5, where the MSE is shown for polynomial fits of different degrees to noisy data from the Runge function using Ridge regression with different regularization strengths λ . For very high regularization strengths the MSE increases significantly, as

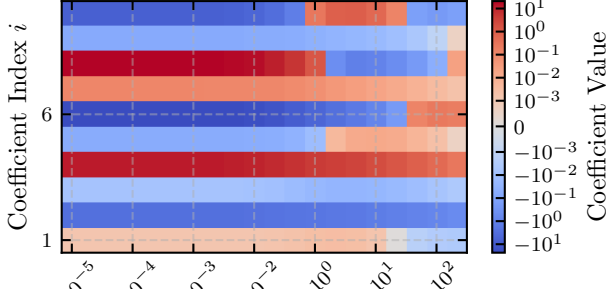


FIG. 4 Parameter values for polynomial fits of different degrees to noisy data from the Runge function using Ridge regression with different regularization strengths λ . $N_{\text{train}} = 80\,000$.

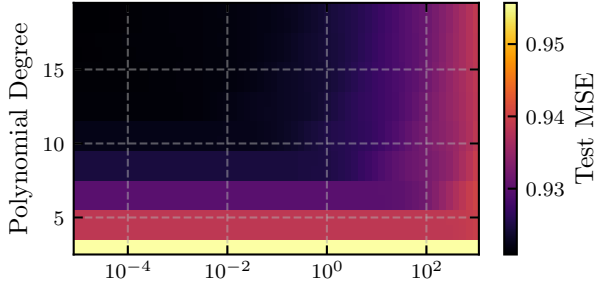


FIG. 5 Mean squared error (MSE) for polynomial fits of different degrees to noisy data from the Runge function using Ridge regression with different regularization strengths λ . $N_{\text{train}} = 80\,000$, $N_{\text{test}} = 20\,000$.

the model is not flexible enough to fit the data well anymore. The optimal regularization strength is promoted to a hyperparameter that needs to be tuned to the problem at hand. However, with a suitable choice of λ it is possible to use models with a high number of parameters without suffering from the negative effects of overfitting.

B. Stochastic Gradient Descent

While the high variance of models with a high number of parameters can be counteracted with a large amount of data, the computational cost of fitting such models can become very high. This is especially relevant in the context of big data, where datasets can contain millions or even billions of samples. In such cases, it can be infeasible to use optimization algorithms that require the computation of the gradient over the entire dataset, such as ordinary gradient descent. For a dataset with N_{train} samples and a model with p parameters, the computational cost of calculating the gradient

$$\nabla_{\theta} C_{\text{OLS}} = \frac{2}{N} \left(X^T X \vec{\theta} - X^T \vec{y} \right), \quad (19)$$

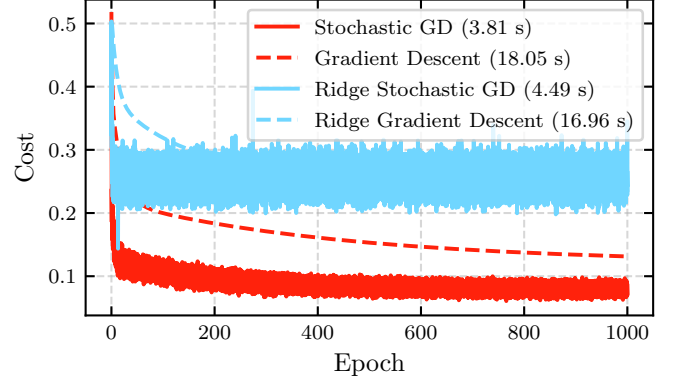


FIG. 6 Convergence of GD and SGD for polynomial fits of degree 10 to noisy data from the Runge function using OLS and Ridge regression with $\lambda = 0.1$. $N_{\text{train}} = 800\,000$, $B = 512$, $N_{\text{batches}} = 100$ and $\sigma = 0.1$.

is of order $\mathcal{O}(Np^2)$, as the matrix-vector products $X^T X$ needs to be computed. For very large datasets this can lead to very long computation times, as the entire dataset needs to be processed to compute the gradient. A common approach to counteract this problem is to use stochastic gradient descent (SGD) or mini-batch gradient descent, where the gradient is computed over a small subset of the data, called a batch. This leads to a significant reduction in computational cost, as the gradient can be computed in $\mathcal{O}(Bp^2)$, where B is the batch size. However, this comes at the cost of increased variance in the gradient estimate, as the gradient is only computed over a small subset of the data. This can lead to slower convergence and more oscillations in the optimization process.

Figure 6 shows the large decrease in computational cost, i.e. the decrease in computation time for 1000 epochs when using SGD instead of GD. While the decrease in computation time is significant, the result is even less black and white, when taking a closer look at the details. While conventional GD can benefit largely by precomputing the matrix products $X^T X$ and $X^T \vec{y}$, this is not possible for SGD, as the batches change in every epoch. This leads to a significant increase in computation time per epoch for SGD compared to GD. However, as the number of epochs needed to reach convergence is usually much lower for SGD than for GD, the overall computation time can still be significantly lower for SGD. This is especially relevant for very large datasets, where the computational cost of GD can become prohibitive. Because while GD needs to process the entire dataset and thus the computational cost is of order $\mathcal{O}(Np^2)$, SGD only needs to process a small batch of size B and thus the computational cost is of order $\mathcal{O}(Bp^2)$ and invariant of the size of the dataset, given the batch size is large enough to estimate the complete gradient well enough. This leads to a significant reduction in computation time for very large datasets, as the computational cost of SGD

does not increase with the size of the dataset.

A second aspect of SGD is visible in fig. 6. While GD converges smoothly to the optimal solution, SGD shows significant oscillations around the optimal solution. This is a direct consequence of the increased variance in the gradient estimate, as the gradient is only computed over a small subset of the data. This can lead to slower convergence and even no convergence at all, if the batch size is too small or the noise in the data is too high. The intrinsic noise in the gradient estimate leads also to the issue, that SGD may be unable to reach an optimal solution with a large amount of noise in the training data. To counteract this problem, the noise in the dataset has been reduced by one order of magnitude compared to the previous sections. However, even with a reduced noise level, SGD is unable to reach the optimal solution. Further fine tuning of the new hyperparameters introduced by SGD,

i.e. the batch size and the batches per epoch, could lead to better results. However, this is a non-trivial task and is highly dependent on the problem at hand. To arrive at a robust solution it might be necessary to increase the batch size until reaching the limit of full-batch GD, which would defeat the purpose of using SGD in the first place.

C. Alternative Optimization Algorithms

D. Bias-Variance Tradeoff and Resampling Techniques

IV. CONCLUSION

V. PERSPECTIVE

REFERENCES