Regression methods for the Runge Function
FYS-STK4155 - Project 1

Philipp Brückelt, Thomas Engl, Lars Bosch*
University of Oslo

(Dated: October 6, 2025)

We investigate different polynomial regression methods for modeling the Runge function with noise, treating it as a representative one-dimensional problem. While this function can not be interpolated well with polynomials, we can use polynomials for approximation. We do this by linear regression. Specifically, we compare Ordinary Least Squares (OLS), Ridge, and LASSO regression. Our simulations show that OLS achieves the lowest error, a result consistent with theoretical analysis. To address computational considerations, we extend the discussion to modern gradient-based optimization methods, including Stochastic Gradient Descent and variants with adaptive learning rates such as Momentum, Adagrad, RMSprop, and Adam. In our numerical experiments, we find that the deterministic gradient descent methods lead to better results than the stochastic ones. Among the former, Adam and Gradient descent with momentum perform the best. Although numerical approaches are generally more efficient than, if they exist, analytic solutions, they tend to produce less precise results. Based on this trade-off, we recommend analytic OLS for small datasets. Finally, we present bootstrap analyses of the bias–variance trade-off and demonstrate that cross-validation provides more reliable error estimates than bootstrap in this context.

I. INTRODUCTION

We study the performance of different regression methods for the so-called Runge function

$$f\colon \mathbb{R} \to \mathbb{R}, \quad x \mapsto \frac{1}{1+25x^2}.$$

This is a well-known example for failure of basic polynomial interpolation using equidistant grid points [1]. One can show that interpolating f with polynomials of increasing degree leads to divergence towards the boundary of the interval $(-1,1)$. This problem can be solved by, e.g., interpolation with Chebyshev polynomials [8] where the grid points are not chosen equidistantly.

In this work we are not interested in interpolating, but in approximating a noisy model including the Runge function by a polynomial, i.e., we search for a good approximation to the data set $\mathbf{y} = (y_i)_{i=1}^n$ for some $n \in \mathbb{N}$ where

$$y_i = f(x_i) + \varepsilon_i$$

with $\varepsilon_i \sim \mathcal{N}(0,\sigma^2)$ for some $\sigma > 0$ and $x_i, i = 1, ..., n$, are uniformly distributed grid points on $(-1,1)$. We hope to obtain less oscillation towards the boundary in this case since we are not fitting the points directly, but search for a polynomial of given degree minimizing the mean squared error

$$\mathrm{MSE}(\mathbf{y},\tilde{\mathbf{y}}) = \frac{1}{n}\mathbb{E}\left[|\mathbf{y}-\tilde{\mathbf{y}}|^2\right] \approx \frac{1}{n}\sum_{i=1}^{n}(y_i-\tilde{y}_i)^2$$

where $\tilde{\mathbf{y}} = (\tilde{y}_i)_{i=1}^n$ denotes the approximations to $\mathbf{y}$.

Several analytic or numerical methods for this problem exist [4]. We will focus on the optimization problems

i) ordinary least squares (OLS)

ii) Ridge regression

iii) LASSO regression

OLS simply aims to minimize the MSE. Ridge and LASSO extend this approach by adding a bias term to the loss function.

One can show that analytic solutions exist for OLS and Ridge [4, Sc. 2.3]. We will compare those with numerical methods such as (stochastic) gradient descent. Of particular interest is also the dependence of the approximation quality of the hyper parameters when using Ridge or LASSO regression.

If numerical optimization methods are used, the quality of the numeric scheme should depend on the choice of parameters used therein. For gradient descent methods, the parameter of highest interest is the learning rate η. Our goal is to determine which value of η leads to the best results and optimize the performance of the algorithm by tuning of parameters.

The remainder of this work is structured as follows. In Section II we first present the three regression methods, OLS, Ridge and LASSO, and possibly analytic solutions. This is followed by a presentation of gradient descent methods that allow for solving the given optimization problems numerically. Afterwards we turn to the implementation in Python. In Section III we discuss the results of our numerical experiments. Particularly, we compare OLS, Ridge and LASSO, as well as analytic and numerical approaches. We work with five different gradient descent methods and use stochastic and non-stochastic versions of those. Finally, in Section IV we conclude with our main findings and a short interpretation of the results from Section III.

[*] https://github.com/thomas-engl/Project-1—Applied-ML

II. METHODS

A. Ordinary Least Squares

The first method we want to discuss is Ordinary Least Squares (OLS). The theoretical results from this section can be found in [4, Sc. 2.3.1] We aim to solve the minimization problem

$$\min_{\tilde{\mathbf{y}}\in\mathbb{R}^n} \frac{1}{n}\sum_{i=1}^{n}(y_i-\tilde{y}_i)^2. \tag{1}$$

We want to fit the Runge function by a polynomial of a certain degree p. Thus, for $i = 1, ..., n$ and given input $\mathbf{x} = (x_i)_{i=1}^n$ the prediction $\tilde{y}_i$ is given by

$$\tilde{y}_i = \beta_0 + \sum_{j=1}^{p}\beta_i x_i^j \tag{2}$$

where the optimal parameters $\boldsymbol{\beta} = (\beta_i)_{i=0}^p$ must be determined. If the data are scaled, the intercept β_0 vanishes, so we may assume $\boldsymbol{\beta} = (\beta_i)_{i=1}^p \in \mathbb{R}^p$. The equation (2) can be rewritten as

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} \tag{3}$$

where $\mathbf{y} = (y_i)_{i=1}^n$ and $\mathbf{X} = (x_i^j)_{i,j=1}^{n,p} \in \mathbb{R}^{n\times p}$ is the feature matrix. Thus, (1) is equivalent to

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^p} \frac{1}{n}(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})^\mathsf{T}(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}). \tag{4}$$

Once we found an optimizer $\hat{\boldsymbol{\beta}}$, we obtain an approximation $\tilde{\mathbf{y}}$ of the true values $\mathbf{y}$ by inserting $\hat{\boldsymbol{\beta}}$ in (3), i.e.,

$$\tilde{\mathbf{y}} = \mathbf{X}\hat{\boldsymbol{\beta}}.$$

We can solve the above optimization problem analytically, assuming that the matrix $\mathbf{X}^\mathsf{T}\mathbf{X}$ is invertible.

Theorem 1. *The solution $\hat{\boldsymbol{\beta}} \in \mathbb{R}^p$ to* (4) *is given by*

$$\hat{\boldsymbol{\beta}} = \left(\mathbf{X}^\mathsf{T}\mathbf{X}\right)^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}.$$

Proof. The gradient of the functional $L\colon \mathbb{R}^p \to \mathbb{R}$ defined by the expression in (4) is given by

$$\nabla L(\boldsymbol{\beta}) = -\frac{2}{n}\mathbf{X}^\mathsf{T}\left(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}\right).$$

Thus, ∇L vanishes if and only if

$$\mathbf{X}^\mathsf{T}\mathbf{y} = \mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\beta}$$

which is equivalent to

$$\boldsymbol{\beta} = \left(\mathbf{X}^\mathsf{T}\mathbf{X}\right)^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}.$$

Since L is convex, this is the unique minimizer. □

Remark 1. *If the matrix $\mathbf{X}^\mathsf{T}\mathbf{X}$ is not invertible, we can use the* Moore-Penrose pseudo inverse *instead, which we denote by $\left(\mathbf{X}^\mathsf{T}\mathbf{X}\right)^\dagger$. In fact, in the case $n = p$, for a nonsingular matrix, the pseudo inverse is equal to the inverse [2]. Thus, we use the numpy [3] implementation of the pseudo inverse in all of our numerical experiments.*

From Theorem 1 we can infer the following result.

Corollary 1. *OLS is an unbiased estimator.*

Proof. A straight-forward calculation shows

$$\begin{aligned}\mathbb{E}[\hat{\boldsymbol{\beta}}] &= \mathbb{E}\left[\left(\mathbf{X}^\mathsf{T}\mathbf{X}\right)^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}\right]\\ &= \mathbb{E}[\left(\mathbf{X}^\mathsf{T}\mathbf{X}\right)^{-1}\mathbf{X}^\mathsf{T}\mathbf{X}\boldsymbol{\beta}]\\ &= \boldsymbol{\beta}.\end{aligned}$$

□

In fact, by the Gauss-Markov theorem, among all unbiased linear estimates, OLS is the one with the smallest variance [4, Sc. 3.2.2]. However, it is useful to not restrict ourselves to unbiased estimators. Thus, we consider two biased estimates in what follows.

B. Ridge and LASSO regression

Ridge regression [4, Sc. 4.3.1] expands the OLS optimization problem by penalizing the size of the coefficients. This is particularly useful, when we have high correlations in the data of our model. Imposing a shrinking of the coefficients prevents that the coefficients become too large and avoids overfitting.

In fact, one optimizes

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^p} \frac{1}{n}(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})^\mathsf{T}(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}) + \lambda\|\boldsymbol{\beta}\|_2^2 \tag{5}$$

with a hyper parameter $\lambda > 0$. Here, $\|\cdot\|_2$ is the Euclidean norm. In general, we can define for $\mathbf{x} = (x_i)_{i=1}^n \in \mathbb{R}^n$ and $1 \leq p < \infty$

$$\|\mathbf{x}\|_p := \left(\sum_{i=1}^{n}|x_i|^p\right)^{1/p}.$$

Using similar arguments as in the proof of Theorem 1, we obtain

$$\hat{\boldsymbol{\beta}} = \left(\mathbf{X}^\mathsf{T}\mathbf{X} + \lambda\mathbf{I}_p\right)^{-1}\mathbf{X}^\mathsf{T}\mathbf{y}$$

where $\mathbf{I}_p \in \mathbb{R}^{p\times p}$ is the identity matrix. From this result, it follows immediately that Ridge is indeed biased since $\lambda > 0$.

LASSO regression [4, Sc. 4.3.2] works analogously, but uses the $\|\cdot\|_1$ norm in (5) instead of the Euclidean norm, i.e., we minimize

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^p} \frac{1}{n}(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})^\mathsf{T}(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}) + \lambda\|\boldsymbol{\beta}\|_1 \tag{6}$$

Thus, LASSO is a nonlinear estimate. Moreover, there exists no analytic solution to this optimization problem. Hence, one has to use numerical methods, namely gradient descent, to solve (6). Those will be discussed in the subsequent subsection.

In order to use gradient descent, we compute the gradients analytically. Denote the loss functions of OLS, Ridge and Lasso by L_{OLS}, L_{Ridge} and L_{LASSO}, respectively. Then we have

$$\nabla L_{\text{OLS}}(\boldsymbol{\beta}) = \frac{2}{n}\mathbf{X}^{\mathsf{T}}\left(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\right) \tag{7}$$

$$\nabla L_{\text{Ridge}}(\boldsymbol{\beta}) = \frac{2}{n}\left(\mathbf{X}^{\mathsf{T}}\left(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\right) + \lambda\boldsymbol{\beta}\right) \tag{8}$$

$$\nabla L_{\text{LASSO}}(\boldsymbol{\beta}) = \frac{2}{n}\mathbf{X}^{\mathsf{T}}\left(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}\right) + \lambda\,\text{sgn}(\boldsymbol{\beta}) \tag{9}$$

where the sign function $\text{sgn}\colon \mathbb{R} \to \mathbb{R}$, defined by

$$\text{sgn}(x) = \begin{cases} 1, & \text{if } x > 0, \\ 0, & \text{if } x = 0, \\ -1, & \text{if } x < 0, \end{cases}$$

is applied component-wise. Note that L_{LASSO} is only differentiable in the distributional sense.

Remark 2. *For gradient descent methods, we have to compute the gradient in every iteration. However, by equations* (7) - (9)*, the matrix products or matrix vector products* $\mathbf{X}^{\mathsf{T}}\mathbf{X}$ *and* $\mathbf{X}^{\mathsf{T}}\mathbf{y}$ *are the same in every iteration for nonstochastic gradient descent methods. Since these operations yield a high computational cost, it is therefore beneficial to precalculate them once and then use the obtained matrices and vectors in every iteration.*

C. Numerical methods

Algorithm 1 Gradient Descent

```
Initialize $\theta$
for $k = 1$ to $K$ do
    $\theta \leftarrow \theta - \eta\nabla L(\theta)$
    if converged then
        return $\theta$
    end if
end for
```

We limit ourselves to variants of Gradient Descent methods. These have the big advantage that they only require the calculation of the gradient and not of the Hessian which is computationally way more expensive. In its simplest form gradient descent can be described as in Algorithm 1. Therein, η denotes the learning rate and L is the loss function we aim to minimize.

In the simple gradient descent, the learning rate η is constant for every iteration. This leads to small steps if the gradient is small and large steps for larger gradients. A possible improvement is Gradient Descent with Momentum [6].

$$\mathbf{m}^{(k)} = \gamma\mathbf{m}^{(k-1)} - \eta\nabla L(\theta^{(k)}) \tag{10}$$

$$\theta^{(k+1)} = \theta^{(k)} + \mathbf{m}^{(k)} \tag{11}$$

The momentum term leads to larger steps if the gradients in consecutive steps go into the same direction. In the opposite, oscillations can be damped which occur when the gradients change frequently the direction completely.

A more sophisticated approach is to adjust the learning rate per parameter which defines the class of Adaptive optimizers. Now parameters with frequent large gradients can profit from a smaller learning rate whereas parameters with small gradients get a larger learning rate. The following update rule is called Adagrad [5].

$$\theta_d^{(k+1)} = \theta_d^{(k)} - \eta\frac{1}{\sqrt{s_d^{(k)} + \epsilon}}\nabla_d L(\theta^{(k)}) \tag{12}$$

with components d and

$$s_d^{(k)} = \sum_{i=1}^{k}\nabla_d L(\theta^{(k)})^2. \tag{13}$$

By ε we denote a small constant that ensures numerical stability. One can easily observe that the learning rate for Adagrad is monotonic decreasing and converges to 0. This slows down or almost stops the updates after many iterations.

On the one hand this ensures convergence. On the other hand, we do not want the step size to decrease too fast. The RMSProp algorithm [5, Sc. 8.4.6.2] solves this problem by setting

$$\theta_d^{(k+1)} = \theta_d^{(k)} - \eta\frac{1}{\sqrt{s_d^{(k)} + \varepsilon}}\nabla_d L(\theta^{(k)})$$

where

$$s_d^{(k+1)} = \rho s_d^{(k)} + (1-\rho)\nabla_d L(\theta^{(k)})^2$$

for $k \geq 0$ and $\mathsf{s}^{(0)} = \mathsf{0}$. Inductively, one shows that

$$s_d^{(k)} = \sum_{i=0}^{k-1}\rho^{k-1-i}(1-\rho)\nabla L(\theta^{(i)})^2.$$

We have $\rho \in (0,1)$. Thus, recent gradients are of higher weight. A typical choice of the parameters is $\rho \in [0.9, 0.99]$.

The ADAM algorithm [5, Sc. 8.4.6.3] combines RMSProp and gradient descent with momentum. The update is given by

$$\theta_d^{(k+1)} := \theta_d^{(k)} - \frac{\eta}{\sqrt{\hat{s}_d^{(k)}} + \varepsilon}\hat{m}_d^{(k)}$$

for $d = 1, ..., p$ where $\hat{\mathbf{m}} = (\hat{m}_d)_{d=1}^p$ and $\hat{\mathbf{s}} = (\hat{s}_d)_{d=1}^p$ are given by

$$\hat{\mathbf{m}}^{(k)} = (1 - \rho_1^k)^{-1}\mathbf{m}^{(k)}, \tag{14}$$

$$\hat{\mathbf{s}}^{(k)} = (1 - \rho_2^k)^{-1}\mathbf{s}^{(k)} \tag{15}$$

and

$$\mathbf{m}^{(k)} = \rho_1 \mathbf{m}^{(k-1)} + (1 - \rho_1)\nabla L(\theta^{(k)}),$$
$$\mathbf{s}^{(k)} = \rho_2 \mathbf{s}^{(k-1)} + (1 - \rho_2)\nabla L(\theta^{(k)})^2.$$

In practice, a typical choice for the parameters is $\rho_1 = 0.9$ and $\rho_2 = 0.999$.

Hence, Adam smoothes the first and the second moment of the gradient. This leads to a faster convergence and scaling the learning rate differently in each dimension ensures numeric stability. Moreover, (14) and (15) perform a bias correction on the moments.

All the gradient descent methods mentioned above can be extended to stochastic gradient descent methods. The idea is to not compute the gradient on the entire data set $\mathbf{x} = (x_i)_{i=1}^n$ but only on subsets of size m. These *minibatch* are sampled randomly. There are different ways to implement stochastic gradients. We use the variant shown in Algorithm 2 where a number k of minibatches is sampled and the gradient is computed seperately on all minibatches. Afterwards, we take the average of all gradients.

Algorithm 2 Stochastic Gradient Descent

```
Initialize $\theta$
for $k = 1$ to $K$ do
    sample $k$ minibatches $M_i, i = 1, .., k$ of size $m$
    Compute $\nabla L_{M_i}(\theta)$ on all minibatches
    $g := \frac{1}{n}\sum_{i=1}^k \nabla L_{M_i}(\theta)$
    $\theta \leftarrow \theta - \eta g$
    if converged then
        return $\theta$
    end if
end for
```

D. Resampling methods

After fitting a model on a given data set, we want to estimate the quality of the model using different metrics, e.g., the mean squared error. The trivial approach is to divide the data set once into training and test data, fit the model on the former, and test it on the latter. This approach has a high variance because we only do the calculation once. To solve the problem, we can resample our data and simulate multiple training and test data tests.

One common used resampling technique is bootstrap. Here, we repeatedly draw samples of size n with replacement from the training data $\mathbf{x} = (x_i)_{i=1}^n$ and then fit the model for each of those samples. Finally, we can take the average of our estimates.

In contrast, k-fold cross-validation [4, Sc. 7.10] divides the data set into k roughly equally-sized parts and uses $k - 1$ of them for training. The k-th part serves as test data. This done k times, the test data is always given by a different part. At the end, the estimates from all k iterations are combined. A special case of k-fold cross-validation is *leave-one-out* cross-validation in which case, we set $k = n$. That is, in each iteration, up to one point, the entire data is used for training and the testing is performed on the remaining data point.

E. Implementation

In the implementation, we use Python, in particular, the modules numpy [3] and Scikit-Learn [7].

The data is scaled in all of our numerical experiments. The targets $\mathbf{y}$ are centered, i.e., we work with $\mathbf{y}_{\text{cent}}$ instead of $\mathbf{y}$ where

$$(\mathbf{y}_{\text{cent}})_i := y_i - \frac{1}{n}\sum_{j=1}^n y_j$$

is obtained from $\mathbf{y}$ by subtracting its mean. The feature matrix $\mathbf{X}$ is scaled as follows: We replace the columns $\mathbf{X_j}, j = 1, ..., p$, of the feature matrix $\mathbf{X}$ by $(\mathbf{X}_{\text{scaled}})_j$ where

$$(\mathbf{X}_{\text{scaled}})_j = \frac{1}{\sigma_j}\left(\mathbf{X}_j - \overline{\mathbf{X}_j}\right) \tag{16}$$

with $\overline{\mathbf{X}_j}$ being the mean of the feature $\mathbf{X}_j$ and σ_j being its standard deviation. If an intercept is used in the feature matrix, the first column, which is now zero, is removed.

We do not want the training data to affect the test data. Thus, it is important to scale the data *after* splitting it into a training and a test part. That is, we first create a feature matrix $\mathbf{X}_{\text{train}}$ for the training data and $\mathbf{X}_{\text{test}}$ data and apply (16) to both matrices separately. Thereby, we use the sample mean and standard deviation of $\mathbf{X}_{\text{train}}$ for the scaling of both, $\mathbf{X}_{\text{train}}$ and $\mathbf{X}_{\text{test}}$ Moreover, we center the test targets $\mathbf{y}_{\text{test}}$ by subtracting the mean of the training data $\mathbf{y}_{\text{train}}$ since we simulate the case that we only know the training targets and the test targets are unknown. After calculating the predictions $\mathbf{y}_{\text{predict}}$, we add the mean of $\mathbf{y}_{\text{train}}$ again in order to have an unbiased result.

Scaling of the data is particularly important if we have big differences between the variances and means of the features $\mathbf{X}_j$. Moreover, while OLS yields equivalent models with and without scaling, Ridge and LASSO only lead to correct results with scaled data [4, Chapter 3].

Let us now turn to the implementation of gradient descent methods.

The challenge for implementing the various gradient descents is to find a simple implementation which is easy to use and minimzes redundance by shared code. We wrote for normal gradient descent, Momentum, Adagrad,

Algorithm 3 Implementation of Gradient Descent

```
Require: $\mathbf{X} \in \mathbb{R}^{n \times p}$, $\mathbf{y} \in \mathbb{R}^n$, gradient function $g(\cdot)$, learning
    rate $\eta$, maximum iterations $K$, precision $\epsilon$, stochastic flag
    $s$, batch size $b$, initial value $\theta^{(0)}$ (optional)
Ensure: $\theta$, number of iterations $k$
$g_{\text{eff}} \leftarrow$ Effective Gradient Calculator$(\mathbf{X}, \mathbf{y}, g, s, b)$
if $\theta^{(0)}$ is None then
    $\theta^{(0)} \leftarrow$ Initial Value Generator$(\mathbf{X})$
end if
$\theta_{\text{new}} \leftarrow \theta^{(0)}$
$\theta_{\text{old}} \leftarrow \theta_{\text{new}} + 1$
$k \leftarrow 0$
while $\|\theta_{\text{old}} - \theta_{\text{new}}\|_2 > \epsilon$ and $k < K$ do
    $\theta_{\text{old}} \leftarrow \theta_{\text{new}}$
    $\theta_{\text{new}} \leftarrow \theta_{\text{old}} - \eta \cdot g_{\text{eff}}(\theta_{\text{old}})$
    $k \leftarrow k + 1$
end while
return $(\theta_{\text{new}}, k)$
```

RMSProp and Adam each one function which can handle non stochastic as well as stochastic gradient descent and all our regression methods, i.e. OLS, Ridge and Lasso. All these functions which we call gradient descent functions have the option to generate a random initial value. This is done by an extra function Initial Value Generator which generates a multivariate uniform distribution where the bounds for every feature are the corresponding minima and maxima in $\mathbf{X}$. The other shared function is the Effective Gradient Calculator. This is executed once in the beginning of the gradient descent methods and returns a function effective gradient descent which only depends on θ and is used in the iterations. For non stochastic gradient descent it precalculates $\mathbf{X}^\mathsf{T}\mathbf{X}$ and $\mathbf{X}^\mathsf{T}\mathbf{y}$. For stochastic gradient descent it generates random batches, calculates the gradient for every batch and the effective gradient is the average gradient of all batches. We stop our iterations in the gradient descent if the Euclidean norm of the change of θ is smaller than our predefined precision or if we have reached the number of maximum iterations. The code for normal gradient descent is illustrated in Algorithm 3.

Suitable learning rates can not be calculated analytically. Thus, we implemented a semi automatic numeric procedure. Given a set of possible learning rates, usually on a log scale, we perform a gradient descent for every learning rate. We save the value of used iterations and the value of the objective function for the resulting θ. It may happen that the algorithm stops after few iterations but does not end in the global minimum and it may happen that two algorithms result in the same value of the objective function, but one needs less iterations. Therefore, our procedure was to look at both values and to decide with expert judgement which learning rate to use. Important to notice is that gradient descent can diverge for bad learning rates and the resulting runtime warnings have to be caught automatically.

F. Use of AI tools

ChatGPT was used to improve the writing of the abstract and the list of references. The whole conversations with ChatGPT was exported and is available on Github.

III. RESULTS AND DISCUSSION

In this section, we present the results of our numerical experiments. Nevertheless, we present two more theoretical results first.

To minimize the MSE of our model, we need to understand which components contribute to the error and how big their impact is. As the following proposition shows, the MSE consists of three quantities: The bias of our model, the variance of the estimates and the variance of the noise.

Proposition 1. *The mean squared error can be written as*

$$\mathbb{E}\left[|\mathbf{y} - \tilde{\mathbf{y}}|^2\right] = \text{Bias}[\tilde{\mathbf{y}}] + \mathbb{V}[\tilde{\mathbf{y}}] + \sigma^2$$

with the bias

$$\text{Bias}[\tilde{\mathbf{y}}] = \mathbb{E}\left[(f(\mathbf{x}) - \mathbb{E}[\tilde{\mathbf{y}}])^2\right],$$

the variance

$$\mathbb{V}[\tilde{\mathbf{y}}] = \mathbb{E}\left[(\tilde{\mathbf{y}} - \mathbb{E}[\tilde{\mathbf{y}}])^2\right]$$

and σ^2 being the variance of the noise $\boldsymbol{\varepsilon}$.

Remark 3. *From this result, we can immediately infer that if our model fitted the data perfectly, i.e., $y_i = \tilde{y}_i$ for all $i = 1, ..., n$, then the MSE would be given by only the variance σ^2 of the noise.*

Proof of the Proposition. We have

$$\begin{aligned}\mathbb{E}\left[(\mathbf{y} - \tilde{\mathbf{y}})^2\right] &= \mathbb{E}\left[(\mathbf{y} - \mathbb{E}[\tilde{\mathbf{y}}] + \mathbb{E}[\tilde{\mathbf{y}}] - \tilde{\mathbf{y}})^2\right] \\ &= \mathbb{E}\left[(\mathbf{y} - \mathbb{E}[\tilde{\mathbf{y}}])^2\right] + \mathbb{E}\left[(\mathbb{E}[\tilde{\mathbf{y}}] - \tilde{\mathbf{y}})^2\right] \\ &\quad + 2\mathbb{E}\left[(\mathbf{y} - \mathbb{E}[\tilde{\mathbf{y}}])(\mathbb{E}[\tilde{\mathbf{y}}] - \tilde{\mathbf{y}})\right].\end{aligned}$$

For the first term, we use that $\mathbf{y} = f(\mathbf{x}) + \boldsymbol{\varepsilon}$ to obtain

$$\begin{aligned}\mathbb{E}\left[(\mathbf{y} - \mathbb{E}[\tilde{\mathbf{y}}])^2\right] &= \mathbb{E}\left[(f(\mathbf{x}) + \boldsymbol{\varepsilon} - \mathbb{E}[\tilde{\mathbf{y}}])^2\right] \\ &= \mathbb{E}\left[(f(\mathbf{x}) - \mathbb{E}[\tilde{\mathbf{y}}])^2\right] + \mathbb{E}\left[\boldsymbol{\varepsilon}^2\right] \\ &\quad + 2\mathbb{E}\left[(f(\mathbf{x}) - \mathbb{E}[\tilde{\mathbf{y}}])\boldsymbol{\varepsilon}\right] \\ &= \text{Bias}[\tilde{\mathbf{y}}] + \sigma^2 + 2\mathbb{E}\left[(f(\mathbf{x}) - \mathbb{E}[\tilde{\mathbf{y}}])\right]\mathbb{E}[\boldsymbol{\varepsilon}] \\ &= \text{Bias}[\tilde{\mathbf{y}}] + \sigma^2.\end{aligned}$$

The second term is equal to $\mathbb{V}[\tilde{\mathbf{y}}]$. The third term can be rewritten as

$$\begin{aligned}&2\left(\mathbb{E}[\mathbf{y}]\mathbb{E}[\tilde{\mathbf{y}}] - \mathbb{E}[\mathbf{y}\tilde{\mathbf{y}}] - \mathbb{E}[\tilde{\mathbf{y}}]^2 + \mathbb{E}[\tilde{\mathbf{y}}]^2\right) \\ &\quad = 2\left(\mathbb{E}[f(\mathbf{x}) + \boldsymbol{\varepsilon}]\mathbb{E}[\tilde{\mathbf{y}}] - \mathbb{E}[(f(\mathbf{x}) + \boldsymbol{\varepsilon})\tilde{\mathbf{y}}]\right) \\ &\quad = 2\left(\mathbb{E}[f(\mathbf{x})]\mathbb{E}[\tilde{\mathbf{y}}] - \mathbb{E}[f(\mathbf{x})]\mathbb{E}[\tilde{\mathbf{y}}]\right) \\ &\quad = 0\end{aligned}$$

by linearity of the expectation and the fact that $\boldsymbol{\varepsilon}$ has mean 0 and is independent of $\tilde{\mathbf{y}}$. This proves the claim. □

A natural question here is, which of those three quantities dominates the MSE. In fact, this depends on the complexity of the model. Naively, one might think that a high complexity model will lead to a better result but this only true up to a certain point. As we see in Figure 8, the train error decreases when the model complexity increases, because the model fits better to the training data. However, the test MSE increases again when the model complexity is increased more and more because of overfitting.

To measure the approximation quality of the different optimization algorithms, we will not only analyze the MSE, but also the R^2 score which is defined as

$$R^2(\mathbf{y}, \tilde{\mathbf{y}}) := 1 - \frac{\sum_{i=1}^n (y_i - \tilde{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}.$$

Here, $\bar{y} = \frac{1}{n}\sum_{i=1}^n y_i$ is the sample mean. Note that the denominator on the right hand side is, up to the factor $1/n$, equal to the sample variance of the data $(y_i)_{i=1}^n$. This ensures that a higher variance of our models does not lead to a higher error, i.e., lower R^2 score. Thus, in contrast to the MSE, the R^2 score is only affected by the quality of our model.

If one estimated all y_i just by their sample mean, this would still result in a R^2 score of 0. This is one of the simplest estimates one can do. Hence, we may expect that a somewhat useful model should always achieve a positive R^2 score. If all y_i were estimated correctly, one would obtain an R^2 score of 1, but due to our noise term this can happen only in overfitting. As our model regarding the Runge function is rather simple, we can explicitly calculate the R^2 value of the perfect model i.e. the true underlying function.

Theorem 2. *The R^2 score of the perfect model*

$$\tilde{y}(x) = \frac{1}{1+25x^2}$$

with σ^2 the variance of the noise term can be expressed as

$$R^2(y, \tilde{y}) \;=\; 1 - \frac{\sigma^2}{\frac{1}{52} + \frac{1}{10}\arctan(5) - \frac{1}{25}\arctan(5)^2 + \sigma^2}.$$

Proof. Under the assumption that our data $X \sim \mathrm{Unif}(-1,1)$ we can derive

σ^2	0.01	0.25	0.5
optimal R^2	0.89026	0.24499	0.13959

Table I. R^2 values (rounded to five decimals) for the perfect model with certain noise variances, calculated using Theorem 2.

$$\begin{aligned}
\mathrm{Var}\Big(\tfrac{1}{1+25X^2}\Big) &= \mathbb{E}\Big[\Big(\tfrac{1}{1+25X^2}\Big)^2\Big] - \Big(\mathbb{E}\Big[\tfrac{1}{1+25X^2}\Big]\Big)^2 \\
&= \int_0^1 \frac{1}{(1+25x^2)^2}\,\mathrm{d}x - \Big(\int_0^1 \frac{1}{1+25x^2}\,\mathrm{d}x\Big)^2 \\
&= \Big[\frac{x}{2(1+25x^2)} + \frac{1}{10}\arctan(5x)\Big]_0^1 \\
&\quad - \Big(\big[\tfrac{1}{5}\arctan(5x)\big]_0^1\Big)^2 \\
&= \frac{1}{52} + \frac{1}{10}\arctan(5) - \frac{1}{25}\big(\arctan(5)\big)^2
\end{aligned}$$

where we changed variables $x = \tan(u)$ in the first integral. Now we can conclude

$$\begin{aligned}
R^2(y, \tilde{y}) &= 1 - \frac{\sum_{i=1}^n (y_i - \tilde{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2} \\
&= 1 - \frac{\frac{1}{n}\sum_{i=1}^n \epsilon_i^2}{\frac{1}{n}\sum_{i=1}^n (y_i - \bar{y})^2} \\
&= 1 - \frac{\mathrm{Var}(\epsilon)}{\mathrm{Var}(y)} \\
&= 1 - \frac{\mathrm{Var}(\epsilon)}{\mathrm{Var}\Big(\frac{1}{1+25X^2}\Big) + \mathrm{Var}(\epsilon)} \\
&= 1 - \frac{\sigma^2}{\frac{1}{52} + \frac{1}{10}\arctan(5) - \frac{1}{25}\big(\arctan(5)\big)^2 + \sigma^2}.
\end{aligned}$$

□

We start the discussion with results about the analytic solutions to OLS and Ridge regression. Afterwards, we test our implementations of the numerical optimization methods, analyze the MSEs for the different algorithms. Moreover, we compare their stochastic and deterministic variants. From here, we also consider LASSO regression.

Finally we turn to resampling techniques where we perform a bias-variance trade-off and cross-validation.

In our experiments with analytic solutions to OLS and Ridge regression, we used $n = 1000$ data points, uniformly distributed on the interval $(-1, 1)$. We chose random noise $\varepsilon \sim \mathcal{N}(0, \sigma^2)$ where σ^2 took the values 0.01, 0.25 and 0.5. Finally, we fitted the Runge function with polynomials of degree up to 15. The train-test split was set to 0.25.

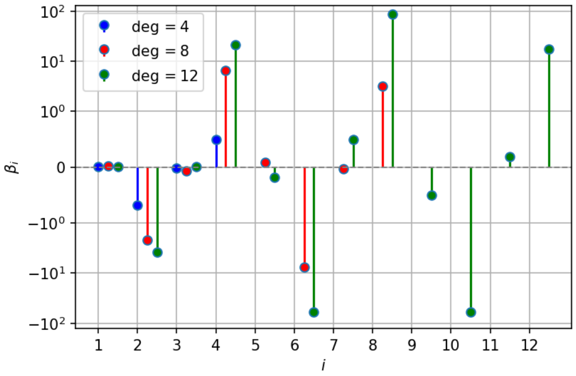

Figure 1. Coefficients of the analytically computed OLS estimate $\hat{\boldsymbol{\beta}}$ for polynomial degrees 4, 8 and 12. The x-axis denotes the position of the entry in the vector, and the y-axis the value.

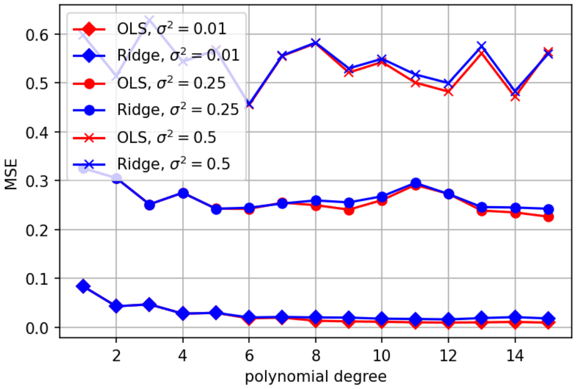

Figure 2. MSE on the test data for analytic implementations of OLS and Ridge with different noise

In Figure 1 one can see the coefficients of the OLS estimate $\hat{\boldsymbol{\beta}}$ for different polynomial degrees. Here, noise with variance $\sigma^2 = 0.01$ was used. For each i the value β_i is plotted for the estimates corresponding to the polynomial degree 4, 8 and 12. The coefficients are plotted slightly offset for a better overview.

The polynomial of degree 4 has low coefficients in comparison to the polynomials of higher degree. In fact, the higher the degree of the polynomial is, the higher are also its coefficients. This might be because when the polynomial degree is low, larger coefficients directly lead to larger oscillations. As other coefficients can not balance this error, the optimal coefficients should be smaller. In contrast, this effect does not arise when we fit the function with a polynomial of a higher degree, since in this case, those oscillations are balanced better.

Moreover, we see that the contribution of odd powers of x is low in comparison to even powers. This is expected since f is even.

The Figures 2 and 3 show the MSE and R^2 score, respectively, using OLS and Ridge regression. In this case, the hyper parameter for Ridge was set to $\lambda = 0.5$.

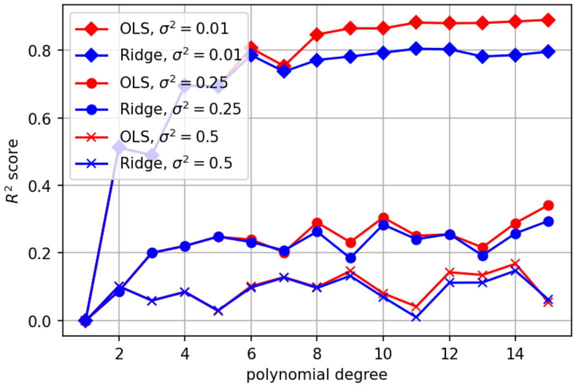

Figure 3. R^2 score on the test data for analytic implementations of OLS and Ridge with different noise

When polynomials of low degree are used for the fitting, OLS and Ridge regression give similar results. However, as the polynomial degree grows, OLS achieves a better MSE and R^2 score and the differences therein are higher for small variances. The reason is that we work with a rather simple model here. There are no correlations between the data and only small variance of the estimates. Thus, the bias of the Ridge regression has the largest impact on the errors, while OLS is unbiased and hence, has smaller errors.

It is also easy to see that the MSE of OLS is approximately converging to the variance σ^2 as the polynomial degree increases. Moreover, the MSE and R^2 score are better for even degrees since we fit a symmetric function. This effect is better to observe for small degrees and noise.

From the plot in Figures 3, we also find that for decent polynomial degrees, the R^2 scores of OLS are approximately equal to the optimal R^2 scores of the optimal model, which we can compute from Theorem 2 and are shown in Table I.

Figures 4 and 5 show how the mean squared error and R^2 score, respectively, of Ridge regression depend on the choices of the polynomial degree and the hyper parameter. We used a noise variance of 0.25 here. For a low model complexity, i.e., for low polynomial degrees, the results obtained with smaller and larger hyper parameters λ do not differ significantly. However, the results are not satisfying for any choice of λ as the MSE is always higher than approximately 0.31 and the R^2 scores are close to 0.

Increasing the polynomial degree leads to better performance of Ridge with larger λ. This is because we obtain overfitting for higher degrees. The parameter λ penalizes high parameters which we, e.g., obtain with OLS, see Figure 1.

Next, recall that from Proposition 1 it is clear that it would be optimal to find a model of low variance and low bias. However, minimizing both at the same time is not possible in general: A low model complexity leads to a

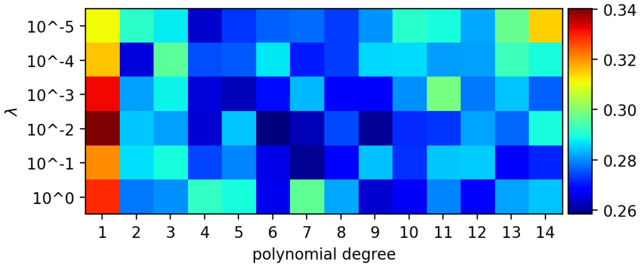

Figure 4. MSE on the test data of analytic Ridge regression for different polynomial degrees and choices of the hyper parameter λ

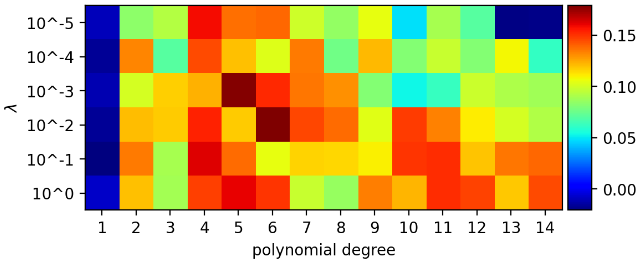

Figure 5. R^2 score on the test data of analytic Ridge regression for different polynomial degrees and choices of the hyper parameter λ

small variance, but higher bias whereas a more complex model yields a lower bias but higher variance.

Moreover, too complex models can lead to overfitting: In training, we fit the model better to the given data, but on unknown data, this results in fitting the noise, too. Thus, the test error gets higher. This can be seen in Figure 8.

We therefore perform a *bias-variance trade-off* to find the optimal model complexity for our problem. Figure 6 shows the bias-variance trade-off for OLS as a function of the polynomial degree. We used 100 datapoints with variance $\sigma^2 = 0.01$ and 1000 bootstrap iterations. Clearly, models of low complexity yield a low variance but a higher bias. Hence, the MSE is almost only affected by the bias of the model. As the complexity increases, the bias decreases but the variance of the estimates grows. As soon as we reach a polynomial degree of 11, the MSE of our estimate mainly consists of the variance. The figure suggests that the optimal polynomial degree in this case lies around 6 to 10.

We can look at the bias-variance tradeoff as function of the number of data points. This is illustrated in Figure 7, where we used again 1000 bootstraps iterations a variance of $\sigma^2 = 0.1$ and a polyomial degree of 8. The variance of the model decreases with the number of data points as the model varies a lot for few data points. The bias is not very affected by the number of data points, because it mainly relies on the choice of our model.

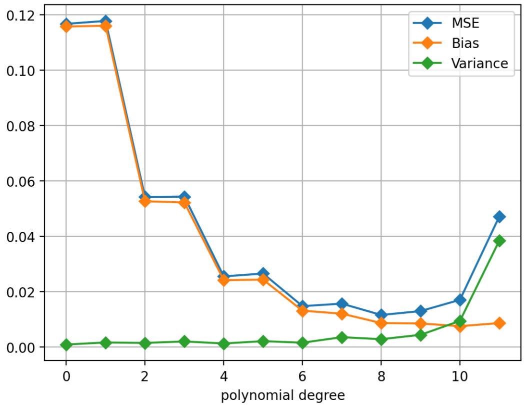

Figure 6. Bias-Variance trade-off for analytic OLS using the bootstrap method, w.r.t. the polyomial degree

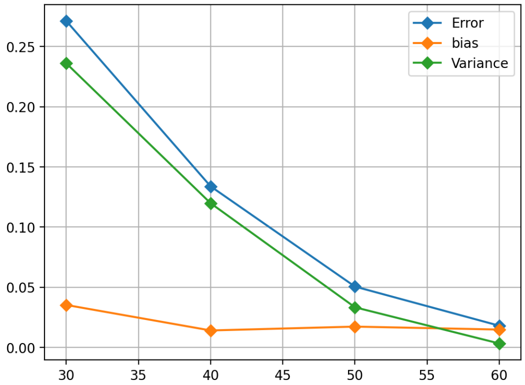

Figure 7. Bias-Variance trade-off for OLS using the bootstrap method, w.r.t. the number of data points

Next, we use k-fold cross-validation to assess the ability of the model to fit $n = 50$ data points with variance $\sigma^2 = 0.01$ with polynomial degree of 6. As expected, in Figure 9 we observe that using larger k results in a lower MSE as our training data set is larger. For a decent number of folds crossvalidation is stable and produces reasonable MSEs which are slightly above the Test MSE. We want to compare these results with bootstrap. Here the results are not stable for a low number of iterations and especially for OLS far off the true test errors as we can see in Figure 10 It is important to notice that this does not occur for a higher number of samples and bootstrap is a valid option in that setting.

Let us now turn to the numerical methods. In Section II we described five different gradient descent methods. Now we want to find the one that performs the best in our given setting.

There are two aspects that should be taken into account when evaluating the quality of a numerical algorithm. Firstly, we investigate how close the output of the algorithm is to the true solution. Secondly, we want the numerical approximations to converge quickly to the solution. That is, we want to compare the runtime of the numerical algorithms considered.

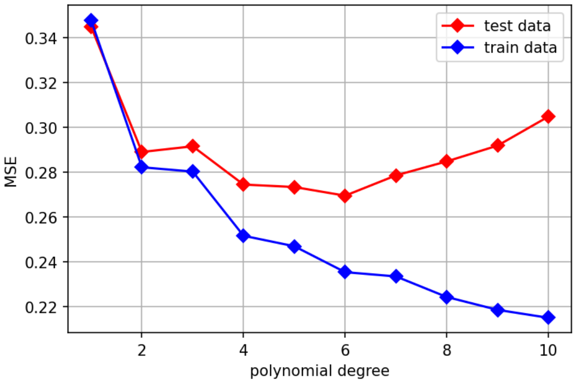

Figure 8. Train and test MSE of the analytic solution of OLS plotted against the degree of the approximating polynomial

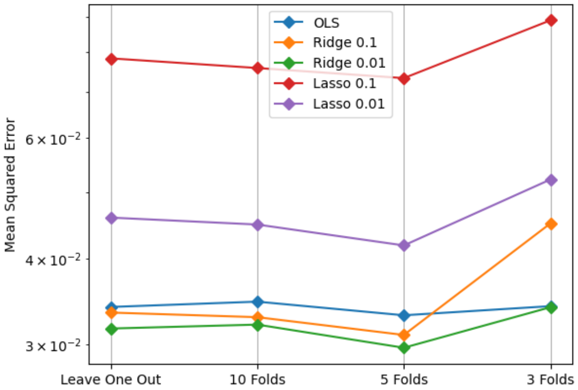

Figure 9. MSE from cross-validation for different numbers of k-folds

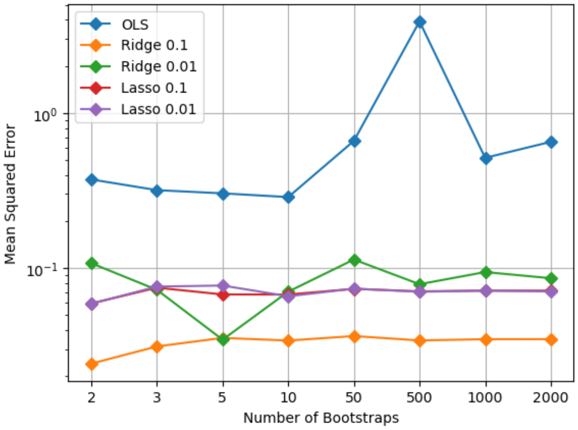

Figure 10. MSE from bootstrap for different numbers of resamples

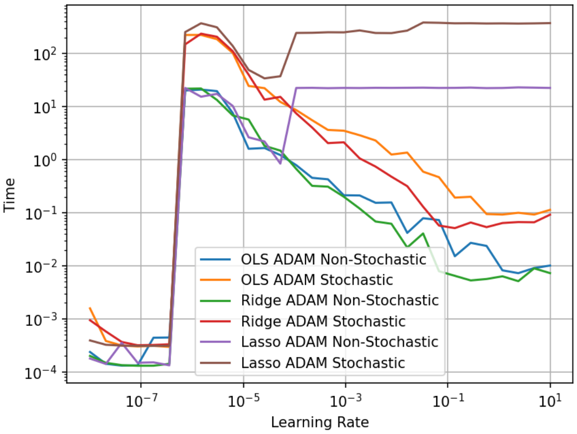

Figure 11. Time the ADAM algorithm takes to converge to a minimum, plotted against the learning rate. We applied ADAM to all three optimization problems and used the deterministic and the stochastic version in each case.

In our discussion we include the stochastic and non-stochastic versions of all five gradient descent methods and perform it on OLS, Ridge and LASSO. This results in a total of 30 numerical algorithms. For each of those, we find the best learning rate by the tuning scheme explained before and perform the gradient descent with precision 10^{-6} and a maximal number of iterations of 10^6. The corresponding results are summarized in the tables II, III and IV.

OLS is the easiest regression method to fit with gradient descent. Every method converges to exactly the same point, resulting in the same training error, as seen in Table II. The methods differ only in the time needed to converge. Ridge needs a similar amount of time as we can see not only in Table III, but also in Figure 11. In this plot we can see that for a very small learning rate the algorithm stops after one iteration because the parameter update is so small that the stopping criterion is executed. For a too large learning rate, the algorithm diverges. Our tuning scheme usually took the largest learning rate that does not diverge. At Ridge, both Adagrad variants have a bigger training error than the rest and the training error of the non-stochastic variant is always slightly worse than the stochastic variant. This effect is way more pronounced at LASSO, where the five stochastic gradient descents are exactly the five slowest ones as you can read in Table IV. Figure 13 further illustrates this. We can observe with Tables II to IV and Figure 11 that for every method, the non-stochastic version is faster than the stochastic one. In that plot, Lasso yields a higher training error than Ridge and OLS due to the definition of the objective. This occurs because our data set is very small, containing only 1000 observations, and our chosen stochastic method, which calculates multiple gradients for one gradient step. In combination with our results regarding the training error, this leads to the result that non-stochastic gradient descent is better suited for our small dataset in this problem.

In order to finish, we have to compare the five different gradient descent methods. Looking at Figure 14, we can see that Momentum, Adagrad, and Adam produce good results for a large range of learning rates. Although we

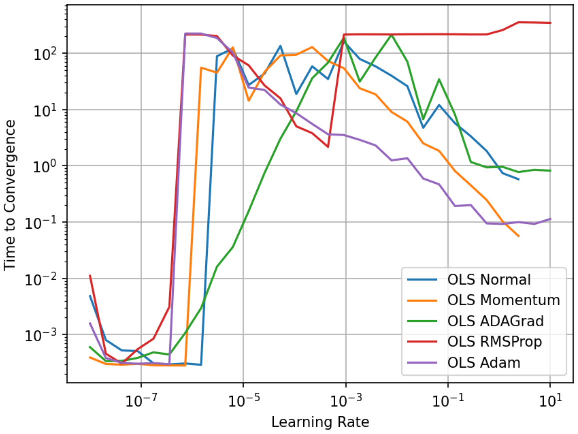

Figure 12. Comparison of the run time, depending on the learning rate, of the five different gradient descent methods applied to OLS.

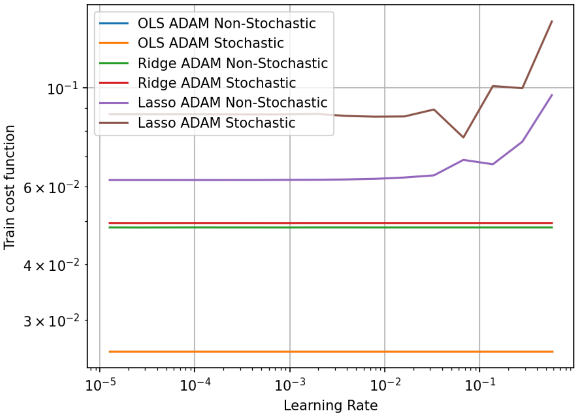

Figure 13. Stochastic and non-stochastic ADAM algorithm for all three optimization problems. The plot shows the evaluation of the train cost function in the optimum found by ADAM with the corresponding learning rate.

only perform the gradient descent with a tuned learning rate, this is a nice effect, because this means that errors in tuning do not have large impacts. Having a look at the time in Figure 12 tells us that the non-stochastic Adam algorithm and non-stochastic gradient descent with momentum are the fastest algorithms. This coincides with our results from the tables where we used only the best learning rate in contrast to the plot. In particular, for LASSO, this time difference is significant.

IV. CONCLUSION

To summarize our main findings, let us start with the results we obtained by analytic computations. Here, we saw that OLS gives the better results compared to Ridge when used to fit the Runge function, and the MSE and R^2 score are close to the predicted optimal values. Using Ridge, we can infer from the plots that, with increasing polynomial degree, a higher regularization parameter is useful for the regression.

Considering numerical approaches, we find that, in this example, deterministic gradient descent methods beat the stochastic ones. In particular, Adam and gradient descent with momentum perform superior in our setting. Both algorithms yield satisfying approximations and converge in a short time span compared to the other algo-

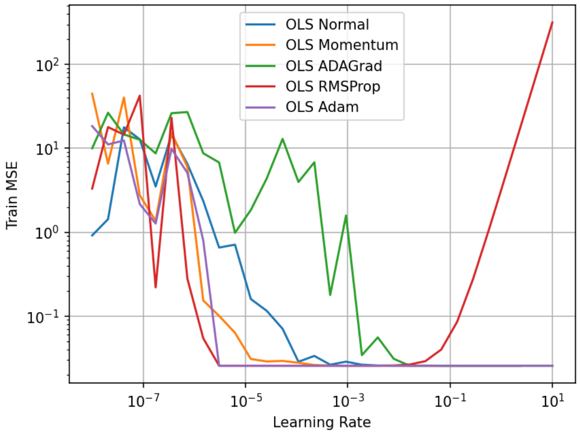

Figure 14. Application of all five gradient descent methods to OLS. The plot shows for every method, how the train MSE depends on the learning rate.

OLS	Train	Test	Time
normal non-stoch	2.5538e-02	3.0705e-02	2.17e-02
momentum non-stoch	2.5538e-02	3.0705e-02	5.74e-03
ADAGrad non-stoch	2.5538e-02	3.0705e-02	7.64e-02
RMSProp non-stoch	2.5538e-02	3.0705e-02	6.89e-01
Adam non-stoch	2.5538e-02	3.0705e-02	7.28e-03
normal stoch	2.5538e-02	3.0705e-02	5.78e-01
momentum stoch	2.5538e-02	3.0705e-02	5.68e-02
ADAGrad stoch	2.5538e-02	3.0705e-02	7.74e-01
RMSProp stoch	2.5538e-02	3.0705e-02	2.17e+00
Adam stoch	2.5538e-02	3.0705e-02	9.26e-02

Table II. Train / test error and runtime for the different numerical algorithms applied to OLS

Ridge	Train	Test	Time
normal non-stoch	7.5154e-02	8.3503e-02	6.29e-03
momentum non-stoch	4.3628e-02	5.1976e-02	4.46e-03
ADAGrad non-stoch	2.0679e-01	2.1514e-01	8.50e-03
RMSProp non-stoch	3.3715e-02	4.2063e-02	7.15e-01
Adam non-stoch	4.3627e-02	5.1976e-02	7.91e-03
normal stoch	4.8276e-02	5.8506e-02	5.60e-01
momentum stoch	5.0359e-02	6.0590e-02	5.30e-02
ADAGrad stoch	7.0921e-02	8.1153e-02	2.07e-02
RMSProp stoch	4.7638e-02	5.7869e-02	1.46e+00
Adam stoch	1.4488e-01	1.5511e-01	6.62e-02

Table III. Train / test error and runtime for the different numerical algorithms applied to Ridge

Lasso	Train	Test	Time
normal non-stoch	4.6152e-02	5.6201e-02	1.77e+01
momentum non-stoch	4.6128e-02	5.6172e-02	5.02e-01
ADAGrad non-stoch	4.6419e-02	5.6463e-02	2.37e+01
RMSProp non-stoch	4.6120e-02	5.6164e-02	1.73e+01
Adam non-stoch	4.6128e-02	5.6172e-02	8.44e-01
normal stoch	9.3390e-02	1.0515e-01	2.29e+02
momentum stoch	8.7136e-02	9.9990e-02	1.99e+01
ADAGrad stoch	3.1804e-01	3.3052e-01	2.31e+02
RMSProp stoch	6.7718e-02	7.8741e-02	3.79e+02
Adam stoch	7.4149e-02	8.6010e-02	3.80e+02

Table IV. Train / test error and runtime for the different numerical algorithms applied to LASSO

rithms investigated. This is true for all three optimization problems, OLS, Ridge and LASSO.

In contrast to OLS and Ridge, we can not compute analytic solutions for LASSO, so we have to stick to numerical methods here. The results are shown in Table IV. With all methods, we obtain a higher train error. Meanwhile, in terms of test errors, LASSO is relatively competitive with OLS and Ridge. However, the time until the algorithms converge for LASSO is significantly higher than for the other two regression methods.

Returning to the mathematical properties, we want to outline the following. Since the function considered f is symmetric, it was in some way expected that one should ideally fit it with polynomials of even degree. To extend this work, it would be interesting to investigate how well we can approximate functions that are neither even nor odd.

Moreover, although the Runge function causes problems for polynomial interpolation, it is still a smooth function and easy to fit in approximation theory. For a better testing of our methods, one should use a more sophisticated function in future work - since by the Weierstrass-Stone theorem any continuous function on a compact interval can be approximated arbitrarily well by polynomials, it might be interesting to also consider discontinuous functions and see if one can still achieve good approximations.

[1] James F Epperson. On the runge example. *The American Mathematical Monthly*, 94(4):329–341, 1987.

[2] Roland W Freund and Ronald HW Hoppe. *Stoer/Bulirsch: Numerische Mathematik 1*, pages 260–263. Springer, 2007.

[3] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi:10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

[4] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[5] Kevin P. Murphy. *Probabilistic Machine Learning: An introduction*. MIT Press, 2022. URL http://probml.github.io/book1.

[6] Goran Nakerst, John Brennan, and Masudul Haque. Gradient descent with momentum — to accelerate or to super-accelerate?, 2020. URL https://arxiv.org/abs/2001.06472.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[8] Theodore J Rivlin. *Chebyshev polynomials*. Courier Dover Publications, 2020.