FYS-STK4155: Project 2

Camilla Dalby Borger

November 2023

Abstract

This report explores how computers can understand patterns in data for making predictions or sorting information. It focuses on creating a special program called a neural network from scratch to solve problems involving finding patterns (regression) and sorting items into categories (classification). The project uses what was learned about simpler methods like linear and logistic regression to build this program.

It starts by making existing regression methods better using new techniques called Stochastic Gradient Descent and other tricks to find patterns in data more efficiently. Then, it moves on to creating the neural network program, teaching it to find patterns in data and make predictions. The project also tests different ways the neural network learns, compares its performance to simpler methods, and examines how well it sorts things into groups.

Introduction

This project is all about teaching computers to learn and make decisions by themselves. It involves creating a program (a neural network) that mimics the way our brains work, enabling it to figure out patterns and solve problems in data. The journey begins by making improvements to existing methods used by computers to find patterns in data. We upgrade these methods to make them faster and more accurate using new tricks like Stochastic Gradient Descent.

Next, we dive deep into building our own program—the neural network. This program learns from examples and gradually improves at recognizing patterns and making predictions. We start by using it to find patterns in data and predict outcomes. Then, we check how well it performs compared to simpler methods we learned before.

After that, we explore different ways the neural network learns and how those affect its performance. We also put it to work on a task where it sorts things into groups, like determining whether tumors are harmful or not based on certain features. We carefully observe how well it does this and adjust different settings to make it better.

Finally, we compare our neural network program's performance to another method called Logistic Regression to see which one is better at classifying things into groups. We analyze the strengths and weaknesses of these methods and wrap up by discussing what worked best for different types of problems. This report is a detailed exploration of teaching computers to learn and make decisions on their own, and it shows which methods are more effective for different tasks.

Methods

In this section, we describe the methods and techniques used in our study to build and evaluate classification models from linear and logistic regression to

neural networks. We will also present the optimization algorithms gradient descent (GD) and stochastic gradient descent (SGD) to be used in both neural network and logistic regression.

Linear regression, both ordinary least squares and Ridge regression are used in this project as a case study for the gradient descent methods.

The function to be used in this project is the simple function

$$f(x_i) = 4 + 3x_i + 2x_i^2 \tag{1}$$

where $x_i \in [0, 1]$ is chosen randomly using a uniform distribution, and $y = f(x)$. Additionally, we have a stochastic noise chosen according to a normal distribution $\mathcal{N}(0, 1)$.

To compute the gradient for both gradient descent and stochastic gradient descent we need the cost function for the different linear regression methods.

For ordinary least squares (OLS) we have the cost function:

$$C_{OLS}(\boldsymbol{\beta}) = \frac{1}{n}\left\{(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})\right\} \tag{2}$$

For Ridge regression, we have the cost function:

$$C_{Ridge}(\boldsymbol{\beta}) = \left\{(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})\right\} + \lambda\boldsymbol{\beta}^T\boldsymbol{\beta} \tag{3}$$

where $\lambda \geq 0$

We will use both analytical expression for the gradient, and Autograd or JAX for computing the gradient for the different cost functions above 2 and 3. An analytical expression for the gradient using the cost function for ordinary least squares (OLS) is

$$\nabla_\beta C_{OLS}(\beta_k) = \frac{2}{n}\{X^T(X(\beta_k - y))\} \tag{4}$$

An analytical expression for the gradient using the cost function for Ridge Regression is

$$\nabla_\beta C_{Ridge}(\beta_k) = \frac{2}{n}\{X^T(X(\beta_k - y))\} + 2\lambda\beta_k \tag{5}$$

Gradient descent (GD)[1]

Gradient descent is an optimization algorithm for finding a local minimum of a differentiable function. When we use gradient descent in machine learning we use it to find the values of a function's parameters (coefficients) that minimize a cost function. As motioned above Linear regression, both ordinary least squares and Ridge regression are used in this project as a case study for the gradient descent methods.

When using gradient descent on the cost function the idea is that the function $C(\beta), \beta = (\beta_1, ..., \beta_n)$ decreases faster if one goes from β in the direction of the negative gradient $-\nabla C(\beta)$.

It can be shown that if

$$\beta_{k+1} = \beta_k - \eta_k \nabla C(\beta_k), \tag{6}$$

with $\eta_k > 0$.

For η_k small enough, then $C(\beta_{k+1}) \leq C(\beta_k)$. This means that for a sufficiently small η_k we are always moving towards smaller function values, i.e. a minimum.

The gradient descent method starts with an initial guess β_0 for a minimum of C and computes new approximations according to 4 for OLS, and 5 for Ridge for $k \geq 0$. The parameter η_kn is often referred to as learning rate. Ideally the sequence $\{\beta_k\}_{k=0}$ converges to a global minimum of the function, but in general we do not know if we are in a global or local minimum. If C is a convex function, all local minima are also global minima. Meaning in this case gradient descent can converge to the global solution.

We are often faced with non-convex high dimensional cost functions with many local minima. Since GD is deterministic we will get stuck in a local minimum, if the method converges, unless we have a very good initial $\beta_{\mathbf{0}}$ guess. This also implies that this method is sensitive to the chosen initial condition.

The gradient descent method is also sensitive to the choice of learning rate η_k. This is because we are only guaranteed that $C(\beta_{k+1}) \leq C(\beta_k)$ for sufficiently small η_k. The problem is to determine an optimal learning rate. If the learning rate is chosen too small the method will take a longer time to converge and if it is too large we can experience erratic behaviour. Many of these shortcomings can be alleviated by introducing randomness. One such method is that of Stochastic Gradient Descent (SGD), which we will introduce later.

Momentum based GD[2]

Introducing the momentum based GD to get a smooth transition to the SGD. The implementation of the momentum based GD is as follows

$$\mathbf{v}_k = \gamma \mathbf{v}_{k-1} + \eta \nabla_{beta} C(\beta_{\mathbf{k}}) \tag{7}$$

$$\beta_{k+1} = \beta_k - \mathbf{v}_k, \tag{8}$$

where we introduce a momentum parameter γ, with $0 \leq \gamma \leq 1$. From these equations, it is clear that $\mathbf{v}_k$ is a running average of recently encountered gradients and $(1-\gamma)^{-1}$ sets the characteristic time scale for the memory used in the averaging procedure. Consistent with this, when $\gamma = 0$, this just reduces down to ordinary SGD.

Stochastic Gradient Descent (SGD)[2]

Stochastic Gradient Descent (SGD) is a variation of the gradient descent optimization algorithm. Unlike regular gradient descent, which calculates the gradient using the entire dataset, SGD computes the gradient using a randomly

selected subset of the data for each step. This subset, known as a mini-batch, allows for a more efficient calculation of the gradient.

The concept behind SGD arises from recognizing that the function we aim to minimize, called the cost function, can often be expressed as a sum over 'n' data points, represented as $\{x_i\}_{i=1}^n$:

$$C(\beta) = \sum_{i=1}^{n} c_i(x_i, \beta) \tag{9}$$

Consequently, the gradient can be computed as a sum over individual gradients:

$$\nabla_\beta C(\beta) = \sum_{i}^{n} \nabla_\beta c_i(x_i, \beta) \tag{10}$$

The stochasticity or randomness in SGD emerges from computing the gradient on a subset of the data, i.e., minibatches. If there are 'n' data points and each minibatch contains 'M' data points, there will be 'n/M' minibatches denoted by B_k, where $k = 1, ..., n/M$.

The main idea is to estimate the gradient by replacing the sum over all data points with a sum over the data points within a randomly chosen minibatch during each step of gradient descent:

$$\nabla_\beta C(\beta) = \sum_{i=1}^{n} \nabla_\beta c_i(x_i, \beta) \rightarrow \sum_{i \in B_k}^{n} \nabla_\beta c_i(x_i, \beta) \tag{11}$$

In our project, we apply the cost function from various regression models to compute this gradient. During each iteration, the model parameters are adjusted in the opposite direction of this gradient, scaled by a learning rate η, aiming to minimize the cost function. The update equation for SGD is represented as:

$$\beta_{k+1} = \beta_k - \eta \nabla C(\beta_k; x_i, y_i) \tag{12}$$

Therefore, a single step in gradient descent using SGD can be described as:

$$\beta_{j+1} = \beta_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_\beta c_i(x_i, \beta) \tag{13}$$

Here, 'k' is randomly chosen with equal probability from the range [1, n/M]. Iterating over the number of minibatches ('n/M') is commonly termed as an epoch. It's typical to select a number of epochs and, for each epoch, iterate through the minibatches. Our project includes visual representations depicting SGD with different numbers of epochs.

The performance of SGD is influenced by hyperparameters such as the learning rate, mini-batch size, and convergence criteria. In the project we will be tunning these hyperparameters to achieve better convergence and prevent overfitting or underfitting in the models.

Determining the convergence point during Stochastic Gradient Descent (SGD) is pivotal for stopping the search for a new minimum. One approach is to check the norm of the gradient after a set number of epochs, stopping if it falls below a defined threshold. However, a small gradient norm only indicates proximity to a local or global minimum, not the exact minimum. Since we are dealing with a simple function. We will be using this stopping criteria for the this project.

An alternative is to assess the cost function at this stage, storing the result and continuing the search. If the convergence criterion triggers later, comparing the cost function values allows retaining the parameters associated with the lowest cost, indicating better convergence.

Momentum based SGD

Learning with stochastic gradient descent can sometimes be slow, the method of momentum is designed to speedup the learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients. The momentum algorithm accumulates an exponentially decaying moving average of past gradients and continues to move in their direction. The implementation of momentum based SGD is as for momentum based GD, the update rule is then given by equation 7.

Tuning the learning rate η[3]

In stochastic gradient descent, setting the right learning rates over time can be tricky. Ideally, we'd adjust our steps based on how flat or steep the path is. Fancy methods like calculating curvature help with this, but they're too slow for big models. Luckily, newer methods like AdaGrad, RMS-Prop, and ADAM track not only the gradient but also its behavior, making them better at adapting to different landscapes.

AdaGrad[3]

The AdaGrad algorithm adjusts the learning rates of different model parameters based on the historical information of gradients. It scales the learning rates inversely proportional to the square root of the sum of squared gradients from the past. Parameters associated with larger changes in the loss function have their learning rates reduced more rapidly, while those with smaller changes see a relatively smaller reduction in their learning rates. This approach emphasizes making more progress in directions where parameters change slowly. AdaGrad is particularly effective in converging quickly when dealing with convex functions. The update rule for AdaGrad is expressed as:

$$
\begin{aligned}
\mathbf{g}_t &= \nabla_\beta C(\boldsymbol{\beta}) \\
\mathbf{s}_t &= \mathbf{g}_t^2 \\
\boldsymbol{\beta}_{t+1} &= \boldsymbol{\beta}_t - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}}
\end{aligned} \tag{14}
$$

Here, $\mathbf{g}_t$ represents the gradient, $\mathbf{s}_t$ is the squared gradient, $\boldsymbol{\beta}_t$ denotes the model parameters at time t, η_t signifies the learning rate at time t, and ϵ prevents division by zero.

RMSProp[2]

In RMSprop, along with maintaining a running average of the first moment of the gradient, the algorithm also keeps track of the second moment denoted by $\mathbf{s}_t = \mathbb{E}[\mathbf{g}_t^2]$. The update rule for RMSprop is represented as follows:

$$
\begin{aligned}
\mathbf{g}_t &= \nabla_\beta C(\boldsymbol{\beta}) \\
\mathbf{s}_t &= \rho \mathbf{s}_{t-1} + (1-\rho)\mathbf{g}_t^2 \\
\boldsymbol{\beta}_{t+1} &= \boldsymbol{\beta}_t - \eta_t \frac{\mathbf{g}_t}{\sqrt{\mathbf{s}_t + \epsilon}},
\end{aligned} \tag{15}
$$

Here, ρ controls the averaging time of the second moment and is typically set to $\rho = 0.9$, η_t is a learning rate usually chosen as 10^{-3}, and $\epsilon \sim 10^{-8}$ is a small regularization constant to avoid divergences. The operations of multiplication and division by vectors are understood as element-wise operations. This formula clearly indicates that the learning rate is reduced in directions where the gradient norm is consistently large. This significantly accelerates convergence by enabling a larger learning rate for flat directions.

ADAM[2]

A related algorithm to consider is the ADAM optimizer. ADAM computes the average of both the first and second moments of the gradient and uses this information to adaptively adjust the learning rate for different parameters. It is an efficient method particularly useful for handling large problems involving extensive data and/or parameters. ADAM is a blend of the gradient descent with momentum algorithm and the RMSprop algorithm previously discussed.

In addition to maintaining a running average of the first and second moments of the gradient (i.e., $\mathbf{m}_t = \mathbb{E}[\mathbf{g}_t]$ and $\mathbf{s}_t = \mathbb{E}[\mathbf{g}_t^2]$, respectively), ADAM incorporates an additional bias correction to compensate for estimating these moments via running averages (indicated by the hats in the update rule below). The update rule for ADAM is given by the following equations, where multiplication and division are once again understood to be element-wise operations:

$$
\begin{aligned}
\mathbf{g}_t &= \nabla_\beta C(\boldsymbol{\beta}) \\
\mathbf{m}_t &= \rho_1 \mathbf{m}_{t-1} + (1-\rho_1)\mathbf{g}_t \\
\mathbf{s}_t &= \rho_2 \mathbf{s}_{t-1} + (1-\rho_2)\mathbf{g}_t^2 \\
\mathbf{m}_t &= \frac{\mathbf{m}_t}{1-\rho_1^t} \\
\mathbf{s}_t &= \frac{\mathbf{s}_t}{1-\rho_2^t} \\
\boldsymbol{\beta}_{t+1} &= \boldsymbol{\beta}_t - \eta_t \frac{\mathbf{m}_t}{\sqrt{\mathbf{s}_t} + \epsilon}
\end{aligned} \tag{16}
$$

Here, ρ_1 and ρ_2 determine the memory lifespan of the first and second moments, typically set to 0.9 and 0.99 respectively, while η and ϵ are identical to those in RMSprop. Similar to RMSprop, the effective step size of a parameter depends on the square magnitude of its gradient.

We will use these three types of tuning the learning rate on both gradient descent with and without momentum and stochastic gradient descent.

Feed Forward Neural Network[4]

In this study, a feedforward neural network was employed as a predictive model. The network architecture consisted of an input layer, one or more hidden layers, and an output layer. The network is in the study is used as a regressor and as a classifier. Therefor we will use different cost functions to find the gradient. For the regression task the cost function for both OLS and Ridge regression are used. The activation function used in the hidden layers was the sigmoid function, but other activation functions such as RELU and leaky RELU was also tested to increase the networks performance. The output layer employed a sigmoid activation for binary classification.

Architecture

The feedforward neural network was designed with L layers, the total number of layers varies from three to five layers including input and output layer. The input layer consists of the size of the design matrix X, the hidden layer of two hidden nodes and the output layer consists of one node. The connections between layers were governed by weights $W^{(l)}$ and biases $b^{(l)}$ where l represent each layer. The input data was fed into the input layer, and the network's forward pass computed activations layer by layer until the final output was obtained.

Activation Functions

The activation function used within the hidden layers was the sigmoid function, defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{17}$$

The sigmoid function introduced non-linearity into the network and enabled the modeling of complex relationships within the data.

Other activation functions tested on the network was RELU, defined as

$$f(x) = \max(0, x), \tag{18}$$

and the leaky RELU

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0 \end{cases} \tag{19}$$

For the output layer, the choice of the activation function depended on the task. For binary classification, a sigmoid activation function was used to produce probabilities between 0 and 1.

Cost Functions

When using the network as a regressor the cost funcion for the different linear regression models are used. Both the OLS cost function (2) and the cost function for Ridge regression (3). The neural network was tested as a regressor on the Franke function from project 1.

$$f(x,y) = \frac{3}{4}\exp\left(-\frac{(9x-2)^2}{4} - \frac{(9y-2)^2}{4}\right) + \frac{3}{4}\exp\left(-\frac{(9x+1)^2}{49} - \frac{(9y+1)}{10}\right) \tag{20}$$

$$+ \frac{1}{2}\exp\left(-\frac{(9x-7)^2}{4} - \frac{(9y-3)^2}{4}\right) - \frac{1}{5}\exp\left(-(9x-4)^2 - (9y-7)^2\right). \tag{21}$$

For binary classification task performed on the breast cancer data, the cross-entropy cost function was used. This function is defined as:

$$\text{Binary Cross-Entropy} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right] \tag{22}$$

where N is the number of samples, y_i is the true label, and $\hat{y}_i$ is the predicted probability for the i-th sample.

Logistic Regression[5]

In this study, logistic regression was utilized as a predictive model. Logistic regression is a linear model that uses the sigmoid function as its activation function and is well-suited for binary classification tasks.

The logistic regression model predicts the probability of an instance belonging to a particular class. The model's output is computed using the sigmoid function, defined as (17), where x is a linear combination of features weighted by coefficients.

The prediction for a sample with features $\mathbf{x}$ and coefficients $\mathbf{w}$ in logistic regression is given by:

$$\hat{y}(\mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x}) \tag{23}$$

where $\hat{y}(\mathbf{x})$ represents the predicted probability of the sample belonging to the positive class.

The activation function used in logistic regression is the sigmoid function. It transforms the linear combination of features into a range between 0 and 1, interpreting the output as a probability.

Stochastic Gradient Descent (SGD) Optimization

Stochastic Gradient Descent (SGD) was employed for optimizing the logistic regression model. The algorithm computed the gradient of the cost function with respect to the model parameters and updated the coefficients iteratively to minimize the cost.

In each iteration, a subset of the dataset was randomly selected to compute the gradient and update the coefficients using the update rule:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \nabla J(\mathbf{w}_t) \tag{24}$$

where $\mathbf{w}_t$ represents the parameters at iteration t, η is the learning rate, ∇J denotes the gradient of the cost function J with respect to the parameters.

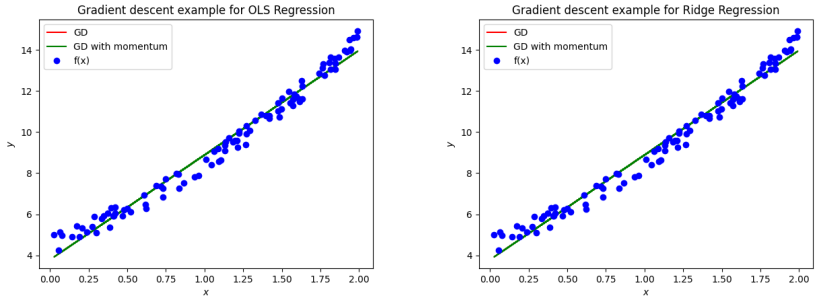
Results

In the following section we will present our result of our implementation of the gradient descent and stochastic gradient descent using the function we have presented above and the gradient is derived from the cost function for the two different regression methods, ordinary least squares and ridge regression. Later we will also present our result from our own Neural Network, and Logistic Regression.

The code used in this section is provided in:
`https://github.com/camilldb/FYS-STK4155/tree/master/Project2`

Gradient Descent

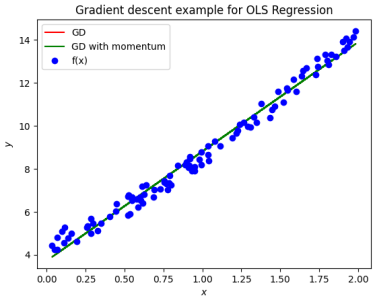
For all the results showed below we have used $\lambda = 0.0001$ after testing different λ values, this λ gave the best results. For the momentum based gradient descent, both for GD and SGD we use $\gamma = 0.3$.



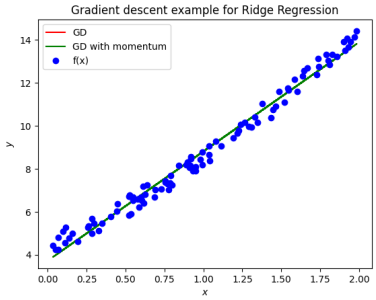
(a) Gradient Descent OLS (b) Gradient Descent Ridge Regression

Figure 1: Gradient descent with and without momentum plotted on top of the real data with the use of analytical expression for the gradient

Figure 1 and 2 shows the gradient descent for the OLS cost function and the Ridge regression cost function for the analytical expression for the gradient and then using the AutoGrad to find the gradient respectively.



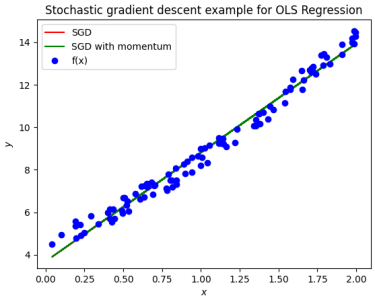
(a) Gradient Descent OLS



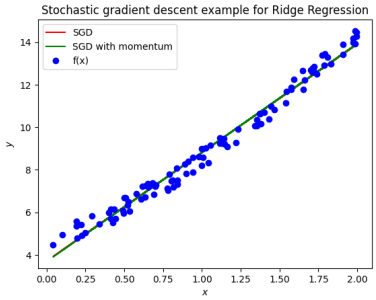
(b) Gradient Descent Ridge Regression

Figure 2: Gradient descent with and without momentum plotted on top of the real data with the use of AutoGrad to find the gradient

Stochastic Gradient Descent



(a) Stochastic Gradient Descent OLS

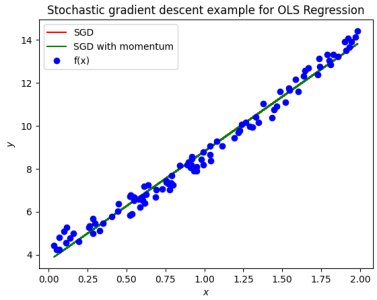


(b) Stochastic Gradient Descent Ridge Regression

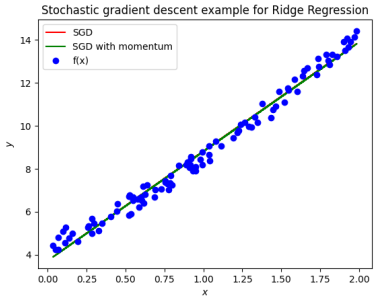
Figure 3: Stochastic Gradient descent with and without momentum plotted on top of the real data with the use of analytical expression for the gradient

Figure 3 and 4 shows the stochastic gradient descent for the OLS cost function and the Ridge regression cost function for the analytical expression for the gradient and then using the AutoGrad to find the gradient respectively.

We have summarized the resulting betas for the different methods in tables. For gradient descent and stochastic gradient descent with and without momentum we have the following betas. In table 1 we have the betas using the analytical expression for the gradient derived from the OLS cost function. Table 2 represent the betas using the analytical expression for the gradient derived from the Ridge regression cost function. Table 3 represent the betas from the OLS gradient derived with the use of Autograd, and table 4 represent the betas from the Ridge gradient derived with the use of AUtograd.



(a) Stochastic Gradient Descent OLS



(b) Stochastic Gradient Descent Ridge Regression

Figure 4: Stochastic Gradient descent with and without momentum plotted on top of the real data with the use of AutoGrad to find the gradient

Model (OLS)	β_1	β_2
Own Inversion	3.90059315	4.90991646
plain GD	3.90197813	4.90874349
GD with momentum	3.90062341	4.90989083
SGD	3.92431763	4.88971642
SGD with momentum	3.90170116	4.90732498

Table 1: Betas for the models with the use of the analytical expression for the gradient using the OLS cost function

Model (Ridge)	β_1	β_2
Own Inversion	3.90059249	4.90991331
plain GD	3.90208948	4.9082784
GD with momentum	3.9004224	4.90969028
SGD	3.8834614	4.92833792
SGD with momentum	3.90230476	4.9077207

Table 2: Betas for the models with the use of the analytical expression for the gradient using the Ridge cost function

Model (OLS)	β_1	β_2
Own Inversion	3.71309453	5.09430307
plain GD	3.71315472	5.09425051
GD with momentum	3.71310497	5.09429395
SGD	3.71571032	5.10011678
SGD with momentum	3.71229122	5.09394259

Table 3: Betas for the models with the use of AutoGrad to find the gradient using the OLS cost function

Model (Ridge)	β_1	β_2
Own Inversion	3.7130954	5.09429837
plain GD	3.71364576	5.09382127
GD with momentum	3.71312455	5.09427646
SGD	3.71576173	5.09818306
SGD with momentum	3.71189613	5.09365532

Table 4: Betas for the models with the use of AutoGrad to find the gradient using the Ridge cost function

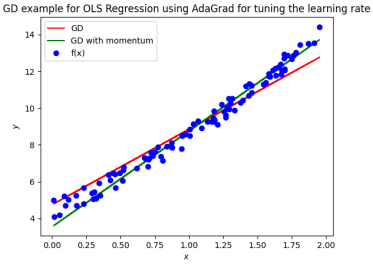
Tuning the learning rate η

In the method section we have discussed three different methods for tuning the learning rate η:

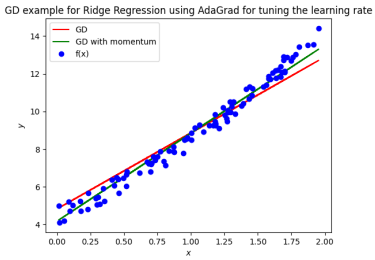
- AdaGrad
- RMSProp
- ADAM

The figures in this section are therefore plots of using the β obtain for each of this methods using fifty epochs and batch size of five for SGD. We will also look at the different plots for deriving the gradient, both with the analytical expression and the AdaGrad method.

The betas obtain from the different methods are summarized in tables in the end of this section.

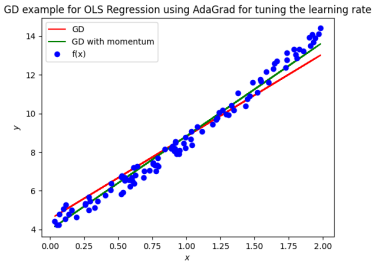


(a) Gradient descent OLS

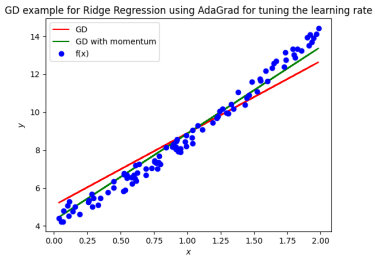


(b) Gradient descent Ridge Regression

Figure 5: Gradient descent with and without momentum plotted on top of the real data with AdaGrad method for tuning the learning rate

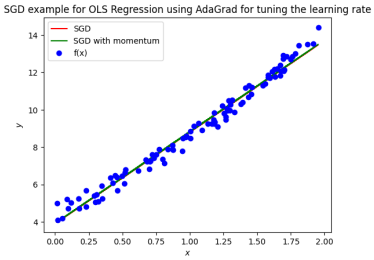


(a) Gradient descent OLS

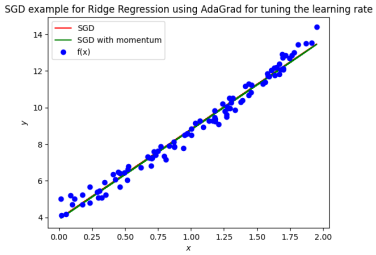


(b) Gradient descent Ridge Regression

Figure 6: Gradient descent with and without momentum plotted on top of the real data with AdaGrad method for tuning the learning rate and AutoGrad to find the gradient

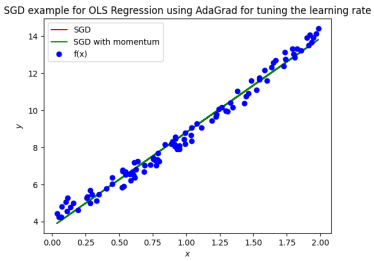


(a) Stochastic gradient descent OLS

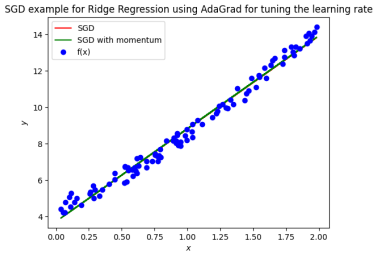


(b) Stochastic gradient descent Ridge Regression

Figure 7: Stochastic gradient descent with and without momentum plotted on top of the real data with AdaGrad method for tuning the learning rate

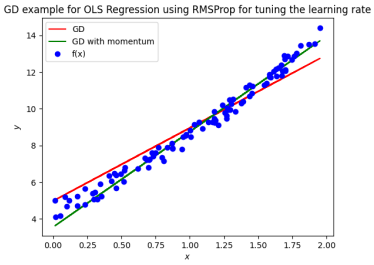


(a) Stochastic gradient descent OLS

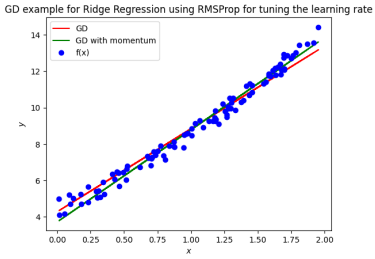


(b) Stochastic gradient descent Ridge Regression

Figure 8: Stochastic gradient descent with and without momentum plotted on top of the real data with AdaGrad method for tuning the learning rate and AutoGrad to find the gradient

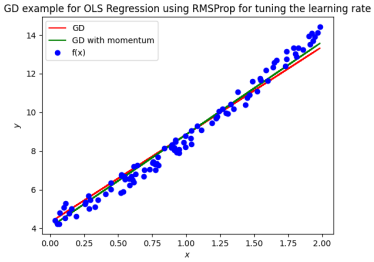


(a) Gradient descent OLS

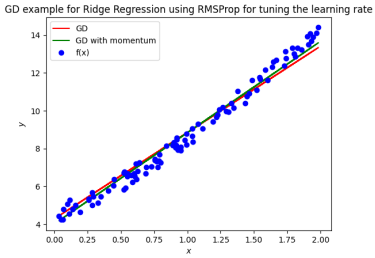


(b) Gradient descent Ridge Regression

Figure 9: Gradient descent with and without momentum plotted on top of the real data with RMSProp method for tuning the learning rate

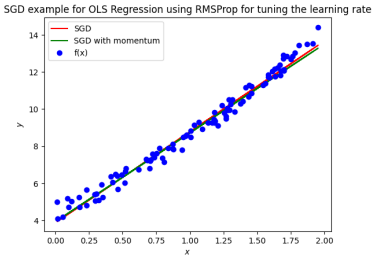


(a) Gradient descent OLS

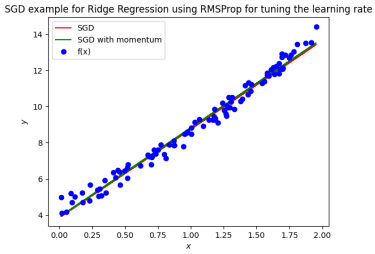


(b) Gradient descent Ridge Regression

Figure 10: Gradient descent with and without momentum plotted on top of the real data with RMSProp method for tuning the learning rate and AutoGrad to find the gradient

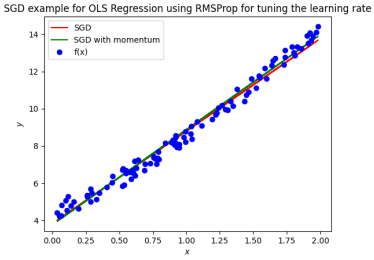


(a) Stochastic gradient descent OLS

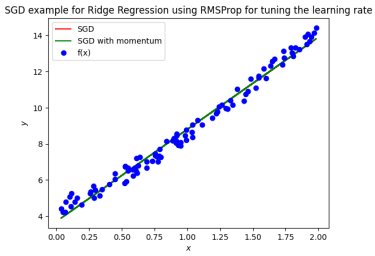


(b) Stochastic gradient descent Ridge Regression

Figure 11: Stochastic gradient descent with and without momentum plotted on top of the real data with RMSProp method for tuning the learning rate

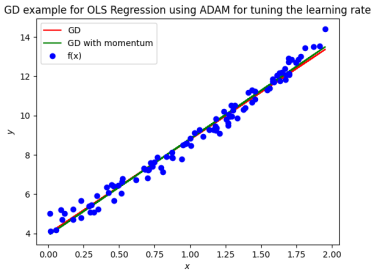


(a) Stochastic gradient descent OLS

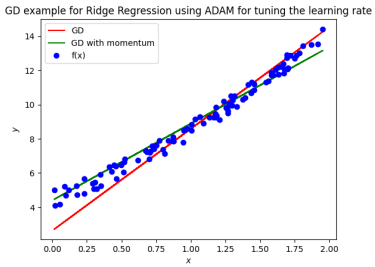


(b) Stochastic gradient descent Ridge Regression

Figure 12: Stochastic gradient descent with and without momentum plotted on top of the real data with RMSProp method for tuning the learning rate and AutoGrad to find the gradient

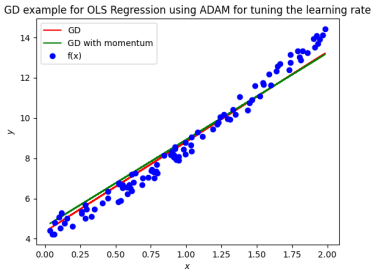


(a) Gradient descent OLS

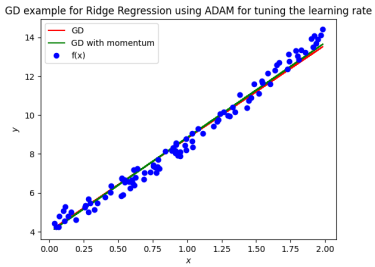


(b) Gradient descent Ridge Regression

Figure 13: Gradient descent with and without momentum plotted on top of the real data with ADAM method for tuning the learning rate

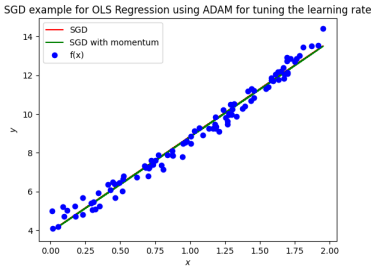


(a) Gradient descent OLS

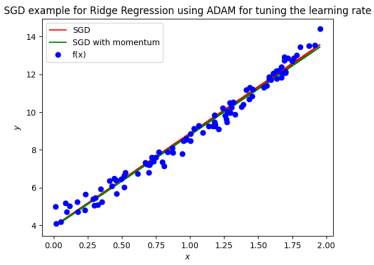


(b) Gradient descent Ridge Regression

Figure 14: Gradient descent with and without momentum plotted on top of the real data with ADAM method for tuning the learning rate and AutoGrad to find the gradient

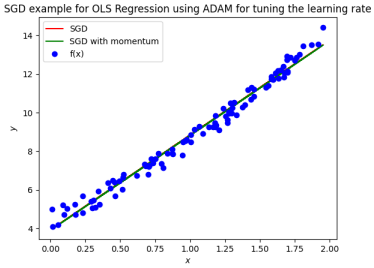


(a) Stochastic gradient descent OLS

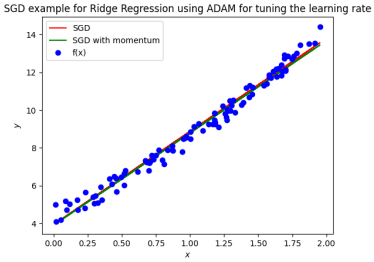


(b) Stochastic gradient descent Ridge Regression

Figure 15: Stochastic gradient descent with and without momentum plotted on top of the real data with ADAM method for tuning the learning rate



(a) Stochastic gradient descent OLS



(b) Stochastic gradient descent Ridge Regression

Figure 16: Stochastic gradient descent with and without momentum plotted on top of the real data with ADAM method for tuning the learning rate and AutoGrad to find the gradient

Model (OLS)	β_1	β_2
Own Inversion	3.90059315	4.90991646
GD AdaGrad	3.07015552	5.53932206
GD AdaGrad with momentum	3.99496515	4.82304118
SGD AdaGrad	3.89924906	4.91460749
SGD AdaGrad with momentum	3.90128605	4.92275664
GD RMSProp	4.35879223	4.50707821
GD RMSProp with momentum	3.54339616	5.20914892
SGD RMSProp	3.90444736	4.89189067
SGD RMSProp with momentum	3.93891647	4.83480816
GD ADAM	3.7469347	5.20576115
GD ADAM with momentum	3.34231077	5.38187612
SGD ADAM	3.91321478	4.9163429
SGD ADAM with momentum	3.89194655	4.9034898

Table 5: Betas for the models with the use of the analytical expression for the gradient using the OLS cost function

Model (Ridge)	β_1	β_2
Own Inversion	3.90059249	4.90991331
GD AdaGrad	4.06490011	4.69434831
GD AdaGrad with momentum	3.16918358	5.51988638
SGD AdaGrad	3.89734308	4.90346397
SGD AdaGrad with momentum	3.90711475	4.88804371
GD RMSProp	4.920012	4.00038349
GD RMSProp with momentum	3.32781211	5.39034802
SGD RMSProp	3.89490296	4.879673
SGD RMSProp with momentum	3.90047601	4.86996564
GD ADAM	3.93385092	4.83047055
GD ADAM with momentum	3.70950273	5.06993622
SGD ADAM	3.91198394	4.92518444
SGD ADAM with momentum	3.87763375	4.88514144

Table 6: Betas for the models with the use of the analytical expression for the gradient using the Ridge cost function

Model (OLS)	β_1	β_2
Own Inversion	3.71309453	5.09430307
GD AdaGrad	4.52915119	4.27617798
GD AdaGrad with momentum	3.97116015	4.85193915
SGD AdaGrad	3.7171783	5.08460127
SGD AdaGrad with momentum	3.71703568	5.08907423
GD RMSProp	4.31020375	4.53867984
GD RMSProp with momentum	4.05650355	4.79185309
SGD RMSProp	3.81784312	4.97223621
SGD RMSProp with momentum	3.75244551	5.10969611
GD ADAM	4.35682071	4.46432116
GD ADAM with momentum	4.60382476	4.31319768
SGD ADAM	3.68829141	5.09790712
SGD ADAM with momentum	3.6910917	5.09671577

Table 7: Betas for the models with the use of AutoGrad to find the gradient using the OLS cost function

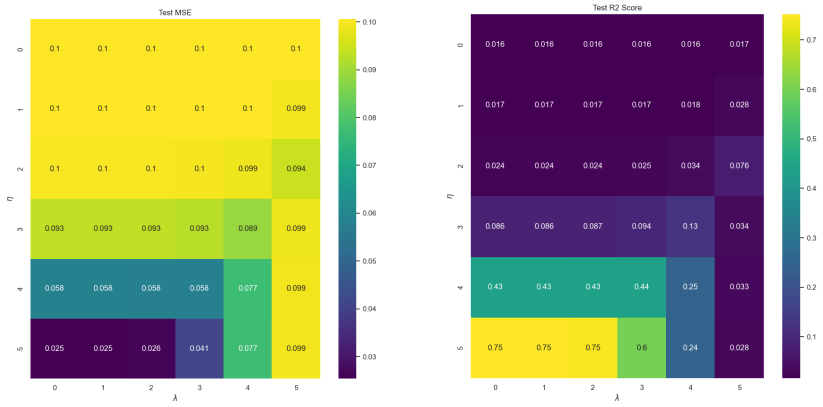
Model (Ridge)	β_1	β_2
Own Inversion	3.7130954	5.09429837
GD AdaGrad	5.08766686	3.8020834
GD AdaGrad with momentum	4.29985887	4.56769502
SGD AdaGrad	3.7258844	5.09400972
SGD AdaGrad with momentum	3.72553806	5.10320876
GD RMSProp	4.27412932	4.57117285
GD RMSProp with momentum	4.0056572	4.83380589
SGD RMSProp	3.70937941	5.08741714
SGD RMSProp with momentum	3.69822997	5.10109885
GD ADAM	4.01351381	4.792309
GD ADAM with momentum	3.94672583	4.88979228
SGD ADAM	3.72066417	5.08277785
SGD ADAM with momentum	3.72201783	5.08038795

Table 8: Betas for the models with the use of AutoGrad to find the gradient using the Ridge cost function

Feed Forward Neural Network

Regressor

We tested the FFNN as a regressor using the cost function for OLS and Ridge regression. We tested the network for different λ's and η's. For the sigmoid activation function we got these results.



(a) MSE OLS

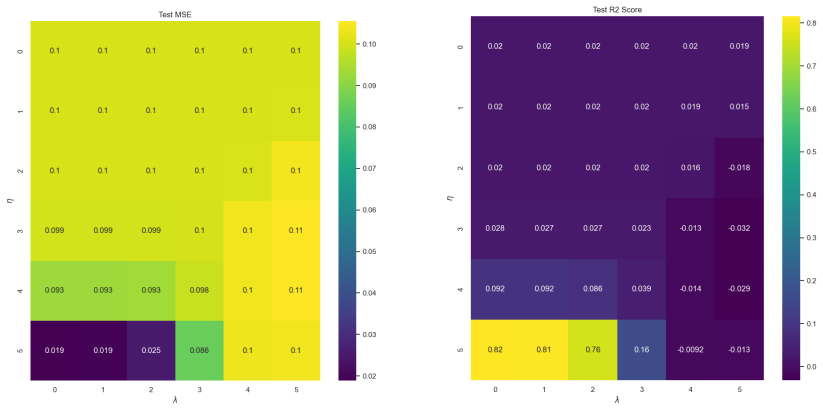
(b) R2 Score OLS

(c) MSE Ridge

(d) R2 Score Ridge

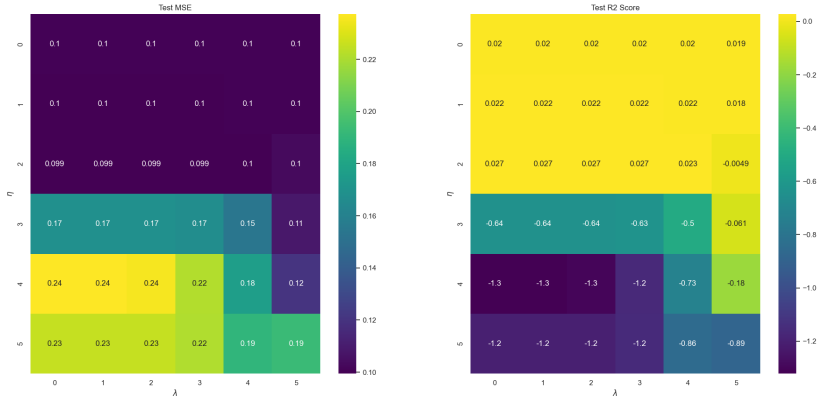
Figure 17: MSE and R2 Score for the Network performance as a regressor with no hidden layers

The network with the best MSE and R2 Score was used to test different activation functions. The first figure 19 shows the results from the activation function RELU, the second figure 20 shows the results from the activation function called leaky RELU.



(a) MSE OLS

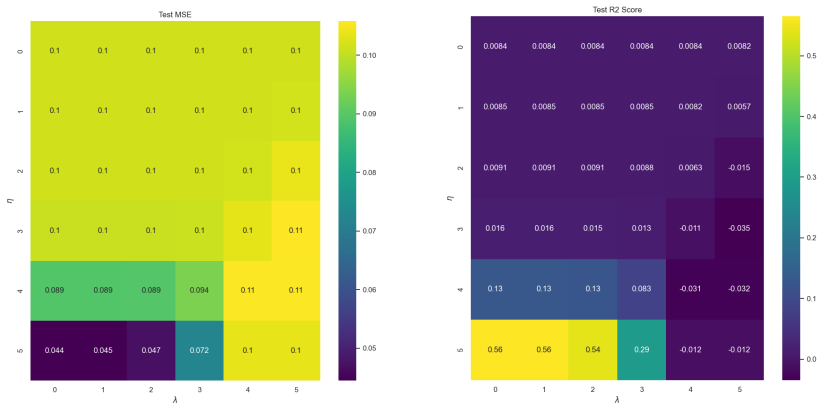
(b) R2 Score OLS



(c) MSE Ridge

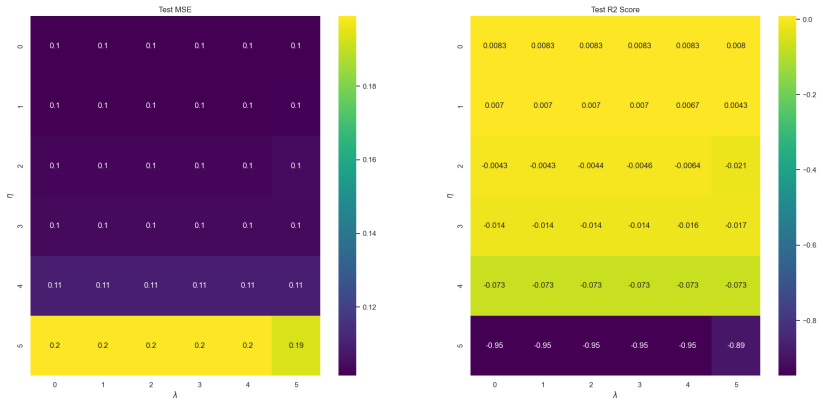
(d) R2 Score Ridge

Figure 18: MSE and R2 Score for the Network performance as a regressor with two hidden layers and the sigmoid activation function for the hidden layer



(a) MSE OLS

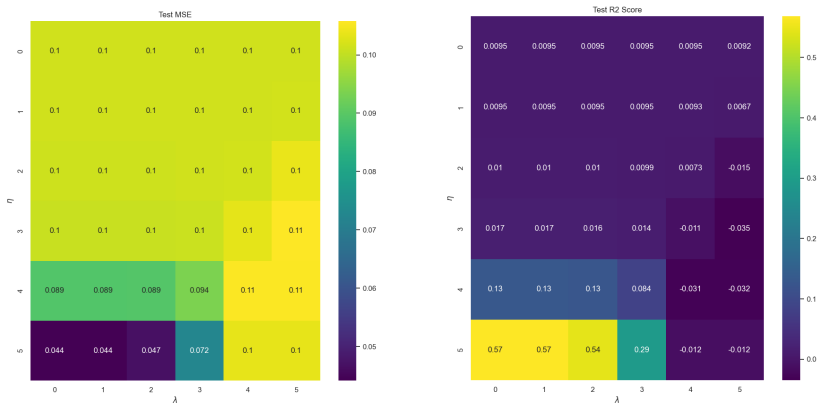
(b) R2 Score OLS



(c) MSE Ridge

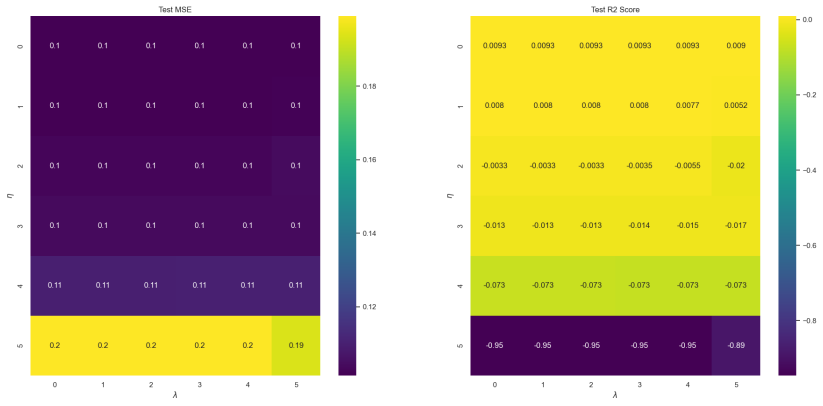
(d) R2 Score Ridge

Figure 19: MSE and R2 Score for the Network performance as a regressor with two hidden layers and the RELU activation function for the hidden layer



(a) MSE OLS

(b) R2 Score OLS



(c) MSE Ridge

(d) R2 Score Ridge

Figure 20: MSE and R2 Score for the Network performance as a regressor with one hidden layers and the leaky RELU activation function for the hidden layer

Classification

The FFNN gives the same accuracy score for both train and test set given different dimensions of the the networks layer and λ and η. The results are 0.354 for the training set and 0.336 for the test set.

Logistic Regression

Figure 21: Accuracy score Logistic regression given different learning rates

Conclusions

Looking at the gradient descent with and without momentum while using the analytical expression for the gradient for the ordinary least square cost function, we can see that the gradient descent with momentum have an almost perfect agreement with our own matrix inversion. This implies that this gradient descent method performs the best on the OLS gradient. We can see the same result when the gradient are derived from the Ridge regression cost function.

We also looked at the estimated for the betas using AutoGrad to find the gradient for the two different cost function. Again the gradient descent with momentum is the method that performs the best.

From the plots for the gradient descent and the stochastic gradient descent we see now major difference in the momentum based and the gradient without momentum for the different method GD or SGD.

The tuning of the learning rate η with different methods results in best prediction for the SGD AdaGrad without momentum for both analytic expression for the gradient and AutoGrad to find the gradient. There is no major difference between the performes of the SGD AdaGrad with or without momentum. This is also possible to see when we look at the figures 7 and 8.

The feed forward neural network was tested as a regressor for both OLS and Ridge regression, and was evaluate by the MSE and R2 score for a test set tested on the FFNN after training. From the figure 17 in the result section we can see that the the FFNN perform best with $\lambda = (1e^{-05}, 1e^{-04})$ and $\eta = 1$ for OLS. For Ridge regression we have a constant $\lambda = 0.1$ in the cost function, and the FFNN perform best with $\lambda = 1$ and $\eta = (1e^{-04}, 1e^{-04})$. This FFNN had no hidden layers only an input layer and output layer. Thereafter we tested the FFNN with one hidden layers the results represented in figure 18 shows that for OLS the $\lambda = 1e^{-05}$ and $\eta = 1$ gives best performance, which is equal to the FFNN with no hidden layers. For Ridge regression we can see that the network perform best with $\lambda = (1e^{-05}, 1e^{-04}, 1e^{-03}, 1e^{-02})$ and $\eta = 1e^{-03}$ which is a bit different than the FFNN with no hidden layers, but the MSE is lower and the R2 score higher for the FFNN with one hidden layers, which means that this network perform better as a regressor.

Thereafter the network with one hidden layer was tested with two different activation functions for the hidden layer. The RELU performed worse then both the other activation function. While the leaky RELU performed better than the RELU but a lot worse than the sigmoid function. We conclude therefor that the FFNN with one hidden layers and the sigmoid activation function perform good for the OLS regression not that good for the Ridge. Compared to our results from the project 1.

In the classification task the FFNN did not perform that good for the different regularization parameter as the train and test accuracy score are under 40%. The Logistic regression performed a lot better given different learning rates η, all scores over 50%. We can conclude that the Logistic regression was better for the classification task using the Wisconsin Breast Cancer data set.

Discussion

There would be ideal to use different λ's in the cost function for Ridge regression in the FFNN. For the classification task the neural network gives the same accuracy score when changing dimensions and regularization parameters. There is probably a bug in the code. The Logistic regression gives better test score than train score which is not what we expected. There may also here be some computational errors.

Bibliography

[1] Morten Hjorth-Jensen. Lecture Notes in FYS-STK4155. Applied data analysis and Machine Learning: 7. Optimization, the central part of any Machine Learning algortithm. 2023.
https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapteroptimization.html#optimization-the-central-part-of-any-machine-learning-algortithm

[2] Morten Hjorth-Jensen. Lecture Notes in FYS-STK4155. Applied data analysis and Machine Learning: Week 39. Optimization and Gradient Methods. 2023.
https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week39.html#gradient-descent-example

[3] Ian Goodfellow and Yoshua Bengio and Aaron Courville. Deep Learning. Chapter 8: Optimization for Training Deep Models. 2016.

[4] Morten Hjorth-Jensen. Lecture Notes in FYS-STK4155. Applied data analysis and Machine Learning: 13. Neural networks. 2023.
https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter9.html

[5] Morten Hjorth-Jensen. Lecture Notes in FYS-STK4155. Applied data analysis and Machine Learning: 6. Logistic Regression. 2023.
https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/chapter4.html