Project 2 in FYS-STK4155

Adrian Martinsen Kleven
Simon Elias Schrader

Autumn 2020

Contents

List of Figures

List of Tables

1 Abstract

Machine learning - is it possible to learn this power? - Anakin
Strong in you The computational power is! - Yoda
It doesn't converge! This is outrageous! This is unfair! How can you call something machine learning but don't learn nothing at all? - Anakin
Meesa computer master now! - Jar Jar Binks
Hello np.where()! - Obi Wan Kenobi
INFINITE CONVERGENCE - The senate

2 Introduction

As can be seen in [3], both Ordinary Least Square (OLS) regression and Ridge regression failed to accurately fit a polynomial function to geographic data and did not manage to match the surface properly. In this article, we analyse whether regression with the help of feed forward neural networks (FFNNs) can give better results (in the form of a lower Mean Square Error and a better R^2 score) than OLS and Ridge regression. In order to do so, we implemented several stochastic gradient methods to find the approximate minimum of the Mean Square Error (MSE) function in parameter space. In order to evaluate their quality, we first compared their performance to the analytical expressions for OLS and Ridge regression for several parameters. Later, these methods were used in the back propagation of the neural network. For the regression problem, we used the sigmoid function as well as RELU and LeakyRELU as activation functions for the hidden layers and the linear function for the output layer. We did this comparison for several stochastic gradient methods and a flexible number of hidden layers and neutrons per hidden layer.
We also tested the neural network's performance on a categorization problem, namely the MNIST data set [4]. We compared several activation functions for the hidden layers, while using the Softmax function (or the sigmoid function) for the output layer. Finally, we compared the neural network's performance to the results achieved using logistic regression.

3 Methods

3.1 Logistic Regression

3.2 Gradient Descent Methods

One way to find the minima, both local and global, of a (multivariable) function, one can use the method of gradient descent. Simply speaking, this is done by iteratively changing the parameters in order to minimize a cost function [1]. As the gradient of a function always shows towards the point of steepest descent, following the gradient in the opposite direction will lead to a minimum. In both regression and classification problems, the function to be minimized is the cost function. In terms of linear regression, the cost function is the MSE function (possibly with additional regularization). The gradient is simply a vector containing the partial derivatives with respect to each coefficient β_i.
For OLS and Ridge regression, we have that

$$\nabla_\beta C(\beta) = \frac{2}{m}\left[X^T\left(X\beta - y\right) + \lambda\beta\right] \tag{1}$$

where C is the cost function, X is the design matrix and m is the number of inputs. The red part is only added for Ridge Regression.
After having an initial guess for the values β^0, The values β are then be updated iteratively by

following the gradient in the opposite direction:

$$\beta^{i+1} = \beta^i - \gamma \nabla_\beta C(\beta^i) \tag{2}$$

where we introduced the learning rate γ. The learning rate γ needs to be chosen in such a way that it is not too large (which can lead to divergent behaviour), but not too small either (which can lead to an extremely slow convergence). This is done either until convergence is reached, or for a given number of iterations, called *epochs*.

3.2.1 Stochastic Gradient Descent Methods

Because calculating the gradient of every parameter β can be rather costly for large data sets, the gradient can be approximated by the gradient at only one input variable which is chosen randomly. This introduces randomness and erratic behaviour to the way the minimum is found. It is hence likely that the minimum is well approximated, but not exact [1]. However, the advantage is that the stochastic method can "jump out of" local minima and find the global minimum. One closely related method is Mini-batch gradient descent, where the gradient is approximated by the gradient at several, but not all, randomly chosen input variables. This leads to a less erratic behaviour, but is still computationally cheaper than Gradient Descent. There are several ways to implement the actual gradient descent. It is useful to adapt the learning rate γ as the program proceeds - starting with a comparatively large learning rate, the algorithm can leave local minima and proceed to the global minimum, while the learning rate is gradually reduced to get better convergence. In the following, three methods of varying complexity will be introduced.

"Naive" Stochastic Gradient Descent has a constant learning rate γ, and the parameters β are just updated by (2) where the gradient is approximated. While this is easy to implement, has only one parameter (γ) to be fine tuned, and is cheap to calculate, the non-adaptive learning rate γ can lead to sub-optimal convergence.

Decaying γ - A simple way to make γ get smaller gradually is to implement a gradual decay. Defining

$$\gamma_t = \frac{t_0}{t_1 + t} \tag{3}$$

where t_0 and t_1 are initialization parameters and t is updated as $t = e \cdot m + i$ where e is the actual epoch, i is the actual mini batch and m is the number of mini batches. The update scheme remains the same (2), just that γ_t is used instead of a fixed γ. While this method has the advantage that γ_t gradually gets reduced, eventually γ gets so small that the steplength gets so small that no convergence is reached.

RMSProp describes a method where the learning rate is reduced gradually by accumulating the gradients from previous iterations, however, unlike the previous method, the impact of previous iterations decays exponentially. The update scheme is described as

$$\begin{aligned} \boldsymbol{s}^{i+1} &= \alpha \boldsymbol{s}^i + (1-\alpha) \nabla_\beta C(\boldsymbol{\beta^i}) * \nabla_\beta C(\boldsymbol{\beta^i}) \\ \boldsymbol{\beta}^{i+1} &= \boldsymbol{\beta}^i - \gamma C(\boldsymbol{\beta^i}) / \sqrt{\boldsymbol{s}^{i+1} + e} \end{aligned} \tag{4}$$

where * and / refer to element-wise multiplication and division, respectively. In this article, we chose the default value $\alpha = 0.9$, while $e = 10^{-8}$ simply has the purpose of avoiding zero division.

3.3 Neural networks

3.3.1 Feed forward?

3.3.2 Back propagation

3.3.3 Activation functions

3.4 Data sets

Blablabla geographic data blablabla
Blablabla MNIST dataset, resolution, number of pictures, blablabla imported from Scikit learn.

4 Computational implementation

4.1 Neural network

4.1.1 Setting up weights and biases for the neural network

As there is no clear rule how to set up weights and biases, other than that they should be initialized with a non-zero value, we first tried to set up the weights with a mean zero normal distribution with a small standard deviation $\sigma \approx 0.01$. However, we found that this yielded undesirable results, which made that especially ReLU and LeakyReLU gave unpredictable behaviour where the activation function gave very high numbers, eventually leading to numerical instability and overflow. Hence, we decided to follow the approach described in [2], where the weights are initialized randomly, following a mean zero normal distribution with standard deviation $\sigma = \sqrt{2/n\_inputs}$ where n_inputs refers to the number of input data points.

5 Results

5.1 Comparison of SGD methods for OLS

Figure 1 shows, for a given OLS problem, the test MSE as a function of the learning rate η for a fixed number of epochs a fixed batch size; the test MSE as a function of the number of epochs for a fixed learning rate a fixed batch size; and the test MSE as a function of batch size for a fixed number of epochs and a fixed learning rate.

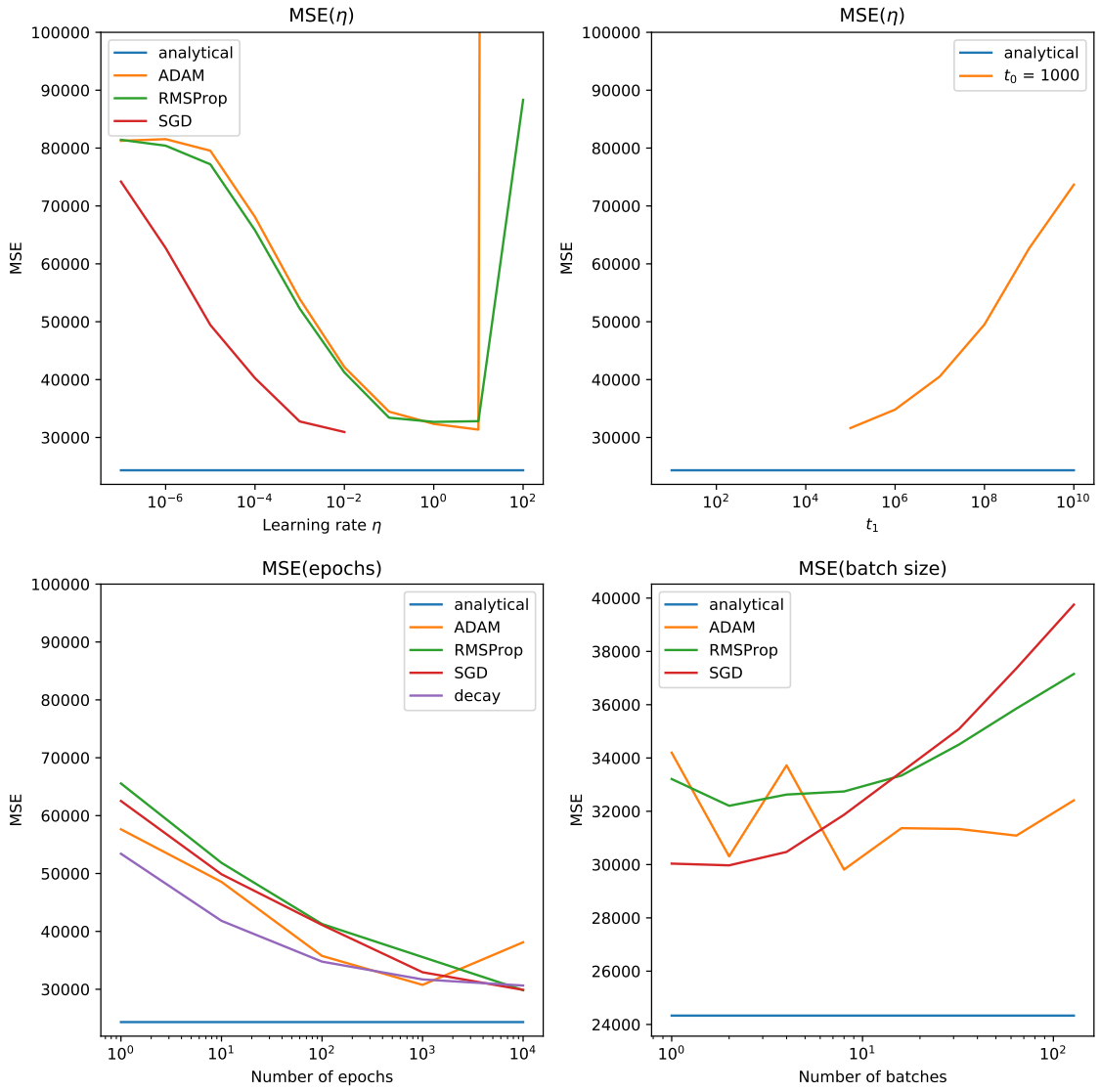

Figure 1: The simple SGD method (titled SGD), RMSProp, ADAM and decaying η as functions of the learning rate η (top left), the number t_1 (top right), the number of epochs (down left), and different batch sizes (down right). The number of data points is $N = 2000$, the polynomial degree used is $deg = 10$. For the top two plots, a batch size of 16 and and epoch of 1000 were chosen. The lower to plots use the ideal parameters η and t_1 which were chosen based on the ideal values from first two plots. No bootstrapping or cross-validation was performed.

As one can see, the number of epochs and the learning rate make a huge difference when it comes to approximating the analytical solution. For the decay-SGD and the simple SGD (titled SGD), the curves are truncated because too big or too small values lead to NaN-values. This shows that the ideal learning rate is dangerously close to a too high learning rate, leading to completely wrong numbers or even NaN-values. Similar observations can be done for both ADAM and RMSProp, but the change is not as drastic for these methods.
As expected, the number of epochs lead to increased error reduction for all methods. However, even though the number of epochs grows exponentially, the error reduction slows down and even ceases. This is hence a computationally expensive way of reducing the extra error. As the learning rate was chosen to be ideal for 1000 epochs, we also see that, at least with ADAM, the error actually increases - this might be due to overfitting, or leaving the reached minimum.
The number of batches does not seem to have a large impact on the quality of the fit for ADAM,

but we observe that the simple SGD method and RMSProp work best with small batch sizes. This might be because these methods work best when making many "small hops" instead of several larger hops.
One can see that the choice of method has a large impact on how fast the error is reduced. RMSProp and ADAM seem to be slightly superior to the simple SGD in terms of convergence to the true MSE, however their biggest advantage is that they are more stable and have "broader" ideal learning rates. This is not surprising as these methods were developed for this purpose. ADAM seems to be better at dealing with higher epochs than RMSProp though. The decay-fit method, while giving results as good as RMSProp and ADAM for ideal parameters, is too unstable to be used in practice - small fluctuations in the parameters lead to completely wrong values. It is also harder to tweak several parameters. Figure 2 contains the same plots as figure 1, however, the learning rates η were chosen so that they didn't exceed a value of 0.1. This is more difficult to do for the decay method, which we left unchanged. One can see that this leads to a slightly different behaviour. The convergence is, not surprisingly, slower, but the methods behave slightly less erratic. That way, the error keeps reducing as the number of epochs increases, but it takes more epochs to get it to the same level as before. Also, the error now increases for larger batch sizes for all methods, including ADAM. In this implementation, a larger batch size has no computational advantages, however, for the Neural Network later, larger batch sizes give increased run time.

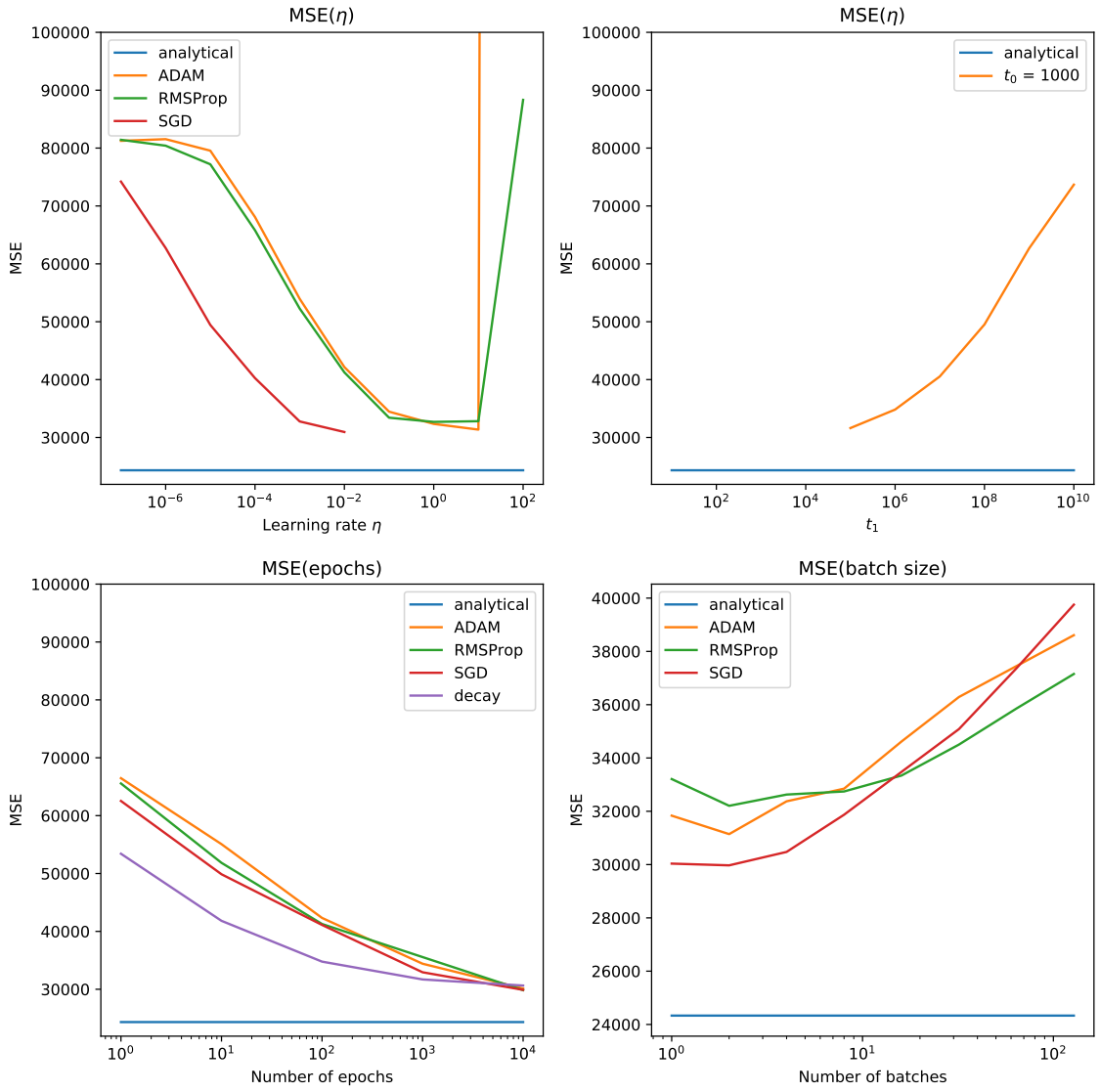

Figure 2: The simple SGD method (titled SGD), RMSProp, ADAM and decaying η as functions of the learning rate η (top left), the number t_1 (top right), the number of epochs (down left), and different batch sizes (down right). The number of data points is $N = 2000$, the polynomial degree used is $deg = 10$. For the top two plots, a batch size of 16 and and epoch of 1000 were chosen. The lower to plots use the ideal parameters η and t_1 which were chosen based on the first two plots, however, η was chosen so that $\eta \leq 0.1$ as larger numbers lead to instability later on. No bootstrapping or cross-validation was performed.

5.2 Comparison of SGD methods for Ridge regression

We repeated the same analysis as above with Ridge regression, only varying the learning rate and the regularisation parameter, keeping the batchsize fixed (16), as well as the number of epochs (1000). We chose a high polynomial degree where OLS is inferior to Ridge regression. The results can be seen in figure 3 in the appendix. The difference between the methods is baffling. We see that simple SGD gives NaN-values for too high learning rates, as before. RMSProp and ADAM, too, give worse results as the learning rate increases, but to a much lesser degree than simple SGD. As we did not perform Cross Validation, these numbers are only qualitatively correct, but we see that all methods, given the ideal parameters are chosen, can get very close to the analytical result. ADAM performs best and manages to come close to the analytical

solution, however, both RMSProp and the simple SGD method get quite close, too. We see that regularization gives improved values for Stochastic Gradient Descent methods, to, as very small regularization parameters λ yield worse test errors than the optimal parameters. We see however that the error is always larger than the ideal test error, implying that Stochastic Gradient Descent methods can get quite close, but not exactly equal to the ideal analytical parameters.

6 Appendix

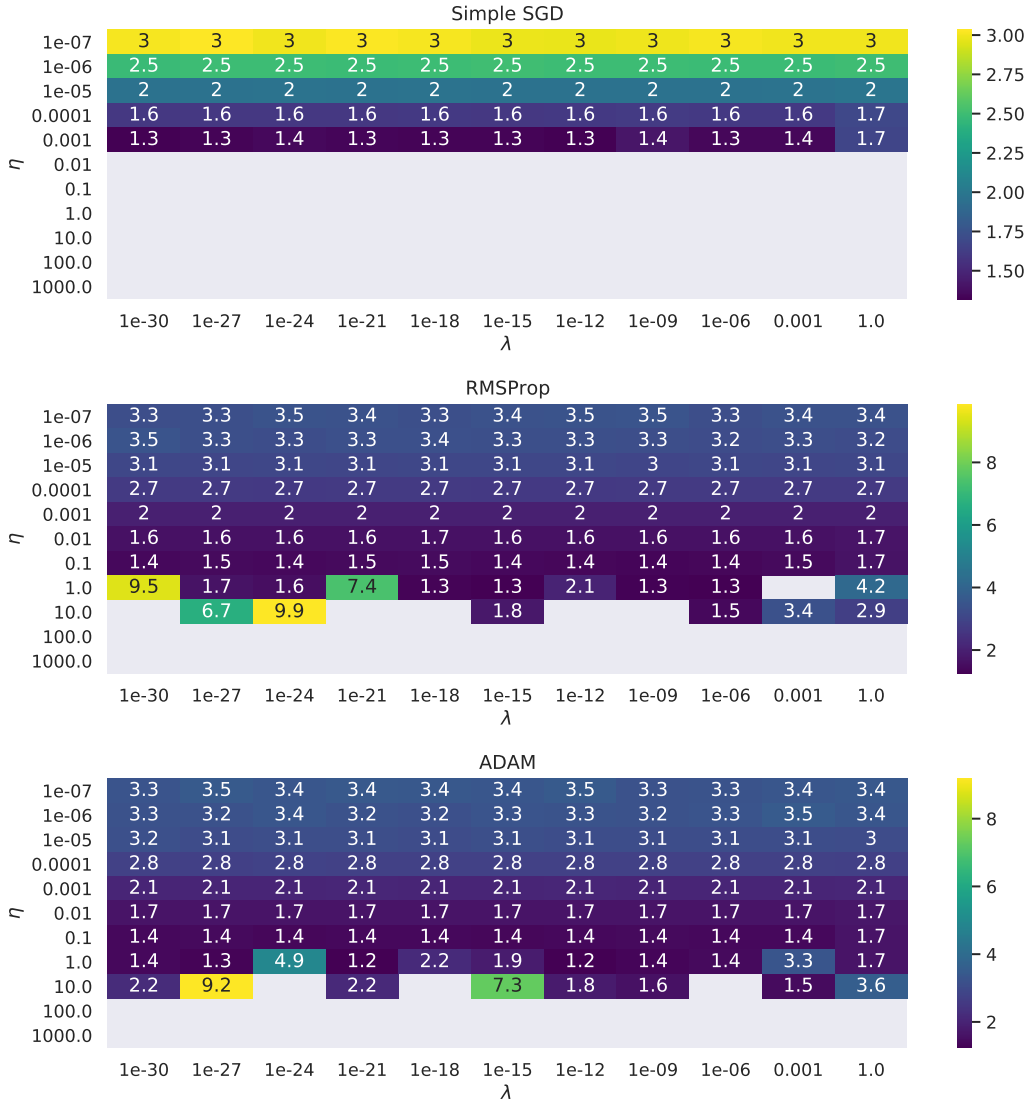

Figure 3: Relative Test MSE ($\frac{MSE_{SGD}}{MSE_{analytical}}$) for the simple SGD method (titled SGD), RMSProp and ADAM and as functions of the learning rate η and the regularization parameter λ. $N = 2000$, the polynomial degree used is $deg = 20$, where OLS fails. The batch size is 16, the number of epochs is 1000. Values exceeding 10 were removed, explaning the grey parts. No bootstrapping or cross-validation was performed.

6.1 Figures

References

[1] Aurelien Geron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, Inc., 1st edition, 2017.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, ICCV '15, page 1026–1034, USA, 2015. IEEE Computer Society.

[3] A. Kleven and S. Schrader. Project 1 in fys-stk4155. 2020.

[4] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.