Using Deep Learning for Regression and Classification

Carlsen, J.[*]

(Dated: November 25, 2023)

We present results from both linear and logistic regression methods, and artificial neural networks (ANN) used on both simple and complex datasets generated pseudo-randomly, and on the Wisconsin Breast Cancer (WBC) dataset. The hyperparameters for learning rate and L2 term are found through extensive searches and presented in a visually appealing matter. This paper also contains results that suggest the Adam optimizer for stochastic gradient descent converges faster than both constant and scaling learning rates. Additionally, we present a model that predicts the tumors of the WBC dataset with an accuracy of 0.974.

I. INTRODUCTION

Ever since pattern recognition became a field of study, the idea of making computers, or rather programs to fulfill such tasks have intrigued scientists (LeCun, Bengio, and Hinton 2015). With the tremendous growth deep learning has shown to solve challenging problems in several categories including object detection and disease diagnostics (Dubey, Singh, and Chaudhuri 2021), the study of artificial neural networks (ANN) is not only attractive but highly valuable. From a programmers point of view in this day of age, the world is your playground. Why, you may ask, and the answer is in the complexity and engineering that lies in the very core of deep learning.

As new algorithms emerge, so does the need for optimizing them. In this paper, we take advantage of the well-known feed forward neural network implemented with the stochastic gradient descent (SGD) algorithm and solve both regression problems and classification problems. In Section II we present the basic ideas of the feed-forward ANN with the back propagated error, and different activation functions used for the hidden layers. The different algorithms used are described in detail as well, together with simple figures for better visualisation. From our research, our findings are presented and shown in Section III, and cross-compared to our own and external sources such as the python library SciKit-Learn (Pedregosa *et al.* 2011). All of the results are discussed in Section IV, and we conclude our findings in Section V. In Appendix A, we include the algorithms used in this paper.

[*] https://github.com/JohanCarlsen/

II. METHODS

A. Stochastic gradient descent

Let us start with the plain gradient descent (GD). Say we want to find the minimum point of the function $f(x) = x^2$. With a given initial value for $\tilde{x}$, GD updates $\tilde{x}$ by decreasing each guess with the negative gradient $\nabla_f(\tilde{x})$ multiplied with a hyperparameter η, more commonly known as the learning rate. Assuming we have a learning rate of 0.1 and an initial guess $\tilde{x} = 5$, estimate the minimum point to be $4.904 \cdot 10^{-6}$ after 62 iterations, which can be seen in Figure 1. Imagine now the dataset is very large, and the function

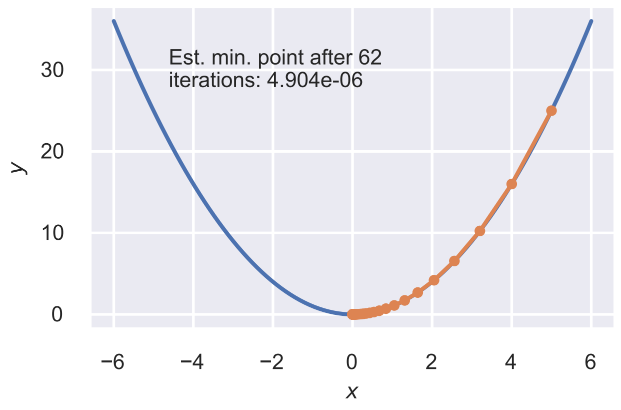

Figure 1: Example use of GD to find the minimum point of the function $f(x) = x^2$, starting from $x = 5$ with a learning rate $\eta = 0.1$.

we want to minimize is a complicated function. Making a guess for the initial value and letting the program run its course can be inefficient. Enter stochastic gradient descent (SGD). Or rather, the mini-batch version of the SGD, as we use in this paper. Here, the idea is that we divide the dataset into mini-batches of a chosen size, with the elements chosen randomly, and iterate over these for a given number of times, known as epochs. As stated by Amari (1993), SGD updates the estimate for the optimal parameter in the direction where the cost decreases. In Algorithm 1, we show the update scheme for SGD, where $\nabla_{\boldsymbol{\theta}}$ is the gradient of the cost function w.r.t. the parameter $\boldsymbol{\theta}$.

The learning rate η can be adapted through the adaptive optimizer Adam (Silva and Rodriguez 2023), implemented as shown in Algorithm 2. For SGD, the results may be noisy (Goodfellow, Bengio, and Courville 2016). To deal with this, we can set a learning schedule given by

$$\eta_k = \frac{t_0}{t + t_1}, \tag{1}$$

where t_0, t_1 are scale parameters which we set to 1 and 50, respectively, and $t = e_j m + i$, where $j = 0, 1, 2, \ldots$ is the number of epochs, m is the number of mini-batches, and $i = 0, 1, 2, \ldots$ is the number of samples within the current mini-batch.

B. Regression

1. *Linear regression*

Linear regression seeks to minimize the mean squared error (MSE), i.e. the cost function:

$$C(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=1}^{n}(\tilde{y}_i - y_i)^2, \tag{2}$$

where $\tilde{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{\beta}$ is the prediction from our model. $\boldsymbol{X}$ is the feature matrix containing powers of x, and $\boldsymbol{\beta}$ are the coefficients we want to optimize. Ridge regression has a slightly different cost function, introducing a penalty term λ to the parameters $\boldsymbol{\beta}$. Here, the cost function takes the form

$$C_{\text{Ridge}}(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=1}^{n}(\tilde{y}_i - y_i)^2 + \lambda\sum_{i=1}^{p}\beta_i^2, \tag{3}$$

where p are the polynomial degrees to consider. A more comprehensive description can be found in Hoerl and Kennard (2000). To generate data, we asked ChatGPT to give us a complicated function that through linear regression can be approximated by a third-order polynomial. ChatGPT provided the function

$$f(x) = 2\sin(2x) - 0.5\cos(3x) + 0.3x^3, \tag{4}$$

which we add noise $\epsilon \sim \mathcal{N}(0, 0.1)$ to. Eq. 4 is evaluated on 101 points for $x \in [-4, 4]$, and normalized to range from 0 to 1. We show the data in Figure 2. We also add momentum to the updates, which improves the convergence rate (Silva and Rodriguez 2023). This is implemented as shown in Algorithm 3.

C. Artificial neural network

As described in Gardner and Dorling (1998), the multilayer perceptron (MLP) consists of interconnected neurons or nodes, which is a model representing a nonlinear mapping between an input vector and an output vector. The nodes are connected by weights and output signals, a function of the sum of the inputs to the node modified by a simple nonlinear transfer, or activation, function. Generally, the MLP consists of an input layer, a given number of hidden layers, and an output layer. In this paper, we create a multilayer feed-forward neural network (FFNN), which is fully connected, meaning that every neuron in each layer is connected to every other neuron in the next forward layer (Sazli 2006). Inspired by the human brain, the mathematical expression for an artificial neuron is given by

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b_i\right) = f(\boldsymbol{z}), \tag{5}$$

$\boldsymbol{w}$ are the weights of the the synapses between the neurons, $\boldsymbol{x}$ are the signal data, and $\boldsymbol{b}$ are the biases of each neuron. The activation function $f(\boldsymbol{z})$ can take different forms, depending on the problem at hand. In this paper, the activation function of the hidden layers is the Sigmoid function for classification problems, given as

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{6}$$

and the leaky rectified linear unit (LReLU) for regression problems, given as

$$f_{\text{LReLU}}(x) = \begin{cases} x & , \ x > 0 \\ kx & , \ \text{otherwise} \end{cases} \tag{7}$$

where $k = 0.01$ to reduce the occurrence of silent neurons (since the derivative is always non-zero) (Xu *et al.* 2020). Figure 3 shows the shapes of the Sigmoid and LReLU activation functions in the top panel, and their derivatives in the bottom panel, for $x \in [-10, 10]$. Whether or not to activate the output layer depends, again, on the situation. For regression problems, a linear output is used, whereas for classification problems, the output is activated by $\sigma(x)$, and the cost function is the log-loss for binary outcome,

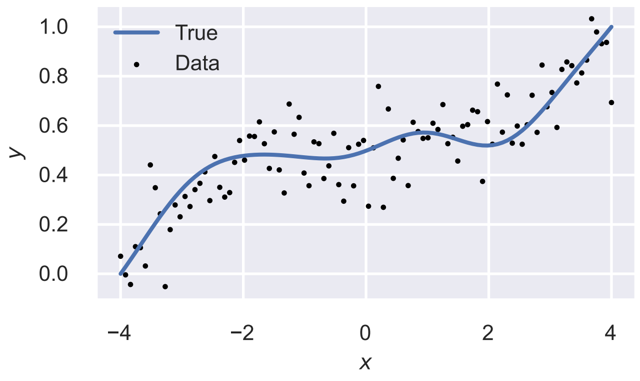

Figure 2: Data used to perform the linear regression analysis. The solid blue line is the true function, and the scatter points are the input data used by the model.

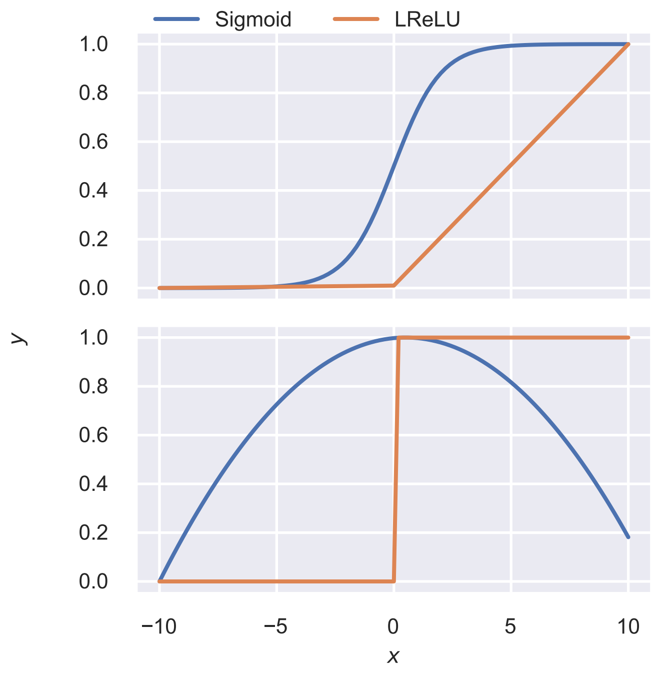

Figure 3: The Sigmoid and LReLU activation functions (top) and their derivatives (bottom).

given by

$$C(\beta) = -\frac{1}{n}\sum_{n}\left[y_n \log(\tilde{y}_n + \delta) + (1 - \tilde{y}_n)\log(1 - \tilde{y}_n + \delta)\right], \tag{8}$$

where $\delta = 10^{-9}$. The scoring metric is the accuracy of the prediction $\tilde{\boldsymbol{y}}$ relative to the target $\boldsymbol{y}$, i.e.

$$\text{Accuracy} = \frac{\sum_{i=1}^{n} A(\tilde{y}_n)}{n}, \tag{9}$$
$$A(\tilde{y}_n) = \begin{cases} 1 & , \tilde{y}_n = y_n \\ 0 & , \text{otherwise} \end{cases}$$

For the FFNN to learn, it has to be able to calculate the error of a prediction. The back-propagation algorithm is perhaps the most popular for this. Let $\boldsymbol{a}^l$ be the activation of layer l, $\boldsymbol{z}^l$ be the input to layer l from the previous layer, and $\boldsymbol{\delta}^l$ be the propagated error of layer l, for $l = 1, 2, 3, \ldots, L$, where L is the output layer. Starting from the output layer, the error is

$$\boldsymbol{\delta}^L = f'(\boldsymbol{z}^L) \circ \frac{\partial C}{\partial(\boldsymbol{a}^L)}, \tag{10}$$

where $\circ$ is the Hadamard product. Note that in the regression case, there is no activation of the output layer, meaning that $f'(\boldsymbol{z}^L) = 1$. Then, for layers $l = L-1, L-2, \ldots, 2$, the error from each neuron is

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l), \tag{11}$$

where the sum runs over all k neurons. Before we continue, we have to take a step back.

Prior to feeding the input data through the network, we initialize the weights and biases. As suggested in Gurbuzbalaban and Hu (2021), we initialize weights according to the activation function of the layers. If the activation function is the Sigmoid function, we use the Xavier/Glorot initializer, given as

$$W_{\text{X/G}}(S_{\text{in}}, S_{\text{out}}) = \mathcal{N}\left(0, \sqrt{\frac{2}{S_{\text{in}} + S_{\text{out}}}}\right), \tag{12}$$

where S are the sizes of the input (subscript in) and output (subscript out) layers, relative to the layer the weights are created for. If the activation function is LReLU, the LeCun initializer, given as

$$W_{\text{LeCun}}(S_{\text{in}}, S_{\text{out}}) = \mathcal{N}\left(0, \sqrt{\frac{1}{S_{\text{in}} + S_{\text{out}}}}\right). \tag{13}$$

The biases are simply initialized as $b_j^l = 0.01$ for all the nodes. To continue with the backpropagation algorithm, the only thing that remains is the update of the weights and biases of the neurons. This is done by

$$\begin{aligned} w_{jk}^l &\leftarrow w_{jk}^l - \eta \delta_j^l a_k^{l-1}, \\ b_j^l &\leftarrow b_j^l - \eta \delta_j^l, \end{aligned} \tag{14}$$

following the SGD method as described in Section II A. Similarly to the linear regression, we generate data from a function, however this time we choose a simpler model given by

$$f(x) = 1 + 0.09x - 0.3x^2 + 0.1x^3, \tag{15}$$

and add noise $\epsilon \sim \mathcal{N}(0, 0.1)$ to it. Eq. 15 is evaluated for 1001 points for $x \in [-4, 4]$, and normalized to range from 0 to 1. Figure 4 shows the function and the data points. The classification dataset we use is the Wisconsin Breast

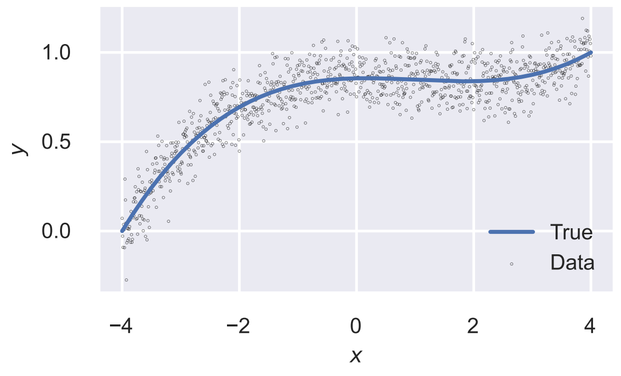

Figure 4: Data used to perform regression analysis with the ANN. The solid blue line is the true function, and the scatter points are the input data used by the ANN.

Cancer (WBC) dataset (Wolberg 1992). We also compare results with the Python package Sci-Kit Learn (SKL).

III. RESULTS

In this section, we present all the results we have produced, starting with the linear regression results.

A. Linear regression

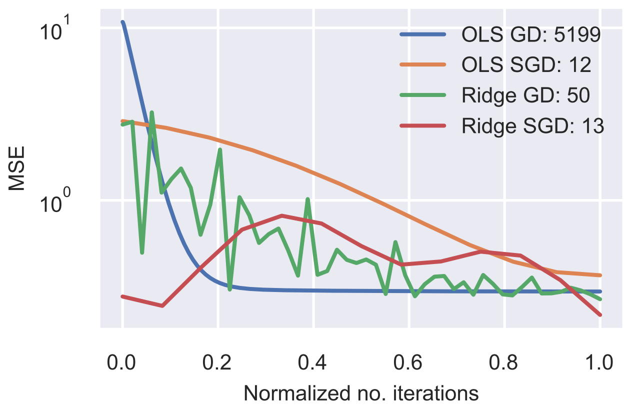

Figure 5: Test MSEs as functions of normalized iterations for the regression models together with the number of iterations before reaching the best score.

From the data generated from Eq. 4, we fit a third-order polynomial, by searching for the hyperparameters that yield the lowest MSE. For runs OLSGD and OLSSGD, we search for values for the learning rate and momentum. In runs RidgeGD and Ridge SGD, we search for the regularization parameter and learning rate. The models that that result in the lowest MSE are compared to the equivalent SKL model, shown in Table I. As seen, the SKL model is much more consistent than our models, with MSE 0.31049 for the OLS runs, and 0.31048 for the Ridge runs. Our models outperform SKL, except for run RidgeGD, where the MSE is 0.50825. Our best model is OLSGD, with an MSE of 0.26240. In Figure 6, we show heat maps of our search for the hyperparameters. The upper row shows plain GD runs and the lower shows SGD runs. In the left column, the OLS runs are shown, while Ridge runs are in the right column. The figure shows the lowest test MSE obtained during training. Except for the OLSGD run, all the runs have quite low MSE scores, with relatively large variations. In the OLSGD run, however, a momentum value of 0 paired with a learning rate of 10^{-5} results in an MSE of 6.4, which is high compared to the other runs. Run RidgeGD has the best score of all the runs when $\lambda = 10^{-3}$ and $\eta = 10^{-2}$, with an MSE of 0.063. With these values for the hyperparameters for run RidgeGD, and $\gamma = 0.9$ and $\eta = 10^{-3}$ for run OLSGD, $\gamma = 0.9$ and $\eta = 10^{-2}$ for run OLSSGD, and $\lambda = 10^{-3}$ and $\eta = 10^{-3}$ for run RidgeSGD, we show the evolution of the MSEs as function of the normalized number of iterations in Figure 5. The number of epochs for the SGD runs is 10, with a minibatch size of 65. In the figure legend, we also show how many iterations it took before each run obtained the lowest MSE. Run OLSGD has a very slow convergence, not reaching it before 5199, where run OLSSGD uses 12, RidgeGD 50, and RidgeSGD 13. In the next section, we show the results from our ANN when used for regression[1].

Run	**Run test MSE**	**Equiv. SKL**
OLSGD	0.26240	0.31049
OLSSGD	0.50825	0.31049
RidgeGD	0.29702	0.31048
RidgeSGD	0.27814	0.31048

Table I: MSE of the best scoring runs and the equivalent result from SKL.

B. ANN for regression

We set up the network with one hidden layer containing 100 hidden nodes. We use the LReLU activation function for the hidden layers with no activation of the output layer. The weights are initialized with the LeCUN initializer, and the minibatch size is 100. We present three ANN runs, adam, constant, and var-η. Run adam uses the Adam update algorithm, run constant uses plain SGD with a constant learning rate, and run var-η uses SGD with a variable learning rate as described by Eq. 1. In Figure 7, we show two heat maps; one for run constant and one for run adam. They show the MSEs obtained from runs with different hyperparameters for η and $L2$ regularization α. For run constant, $\eta = \alpha = 10^{-3}$ yields the lowest MSE at 0.075. Run adam has the best score with $\eta = 10^{-3}$ and $\alpha = 10^{-4}$. Though the figure shows that the MSE is the same for $\alpha = 10^{-5}$, the round-off of the figure does not show that it is indeed 10^{-4} that gives the best result. The convergence of the MSEs as functions of epochs for runs constant and adam, together with run-η is shown in Figure 8. It shows that the constant learning rate performs the worst, and the Adam optimizer the best. Also interesting is that the variable learning rate performs similarly to Adam, yet not quite, with the Adam optimizer converging a lot faster. Choosing the hyperparameters that result in the lowest MSE for runs constant and adam, and running the var-η results in the final test MSEs shown in Table II. We also add the results from what is thought to be the equivalent model from the MLPRegressor method from SKL. Run

[1] When developing the ANN code, we found an error in the linear regression code. We tried to solve this but were not successful. Lacking proper version control caused the code not to be able to be reset.

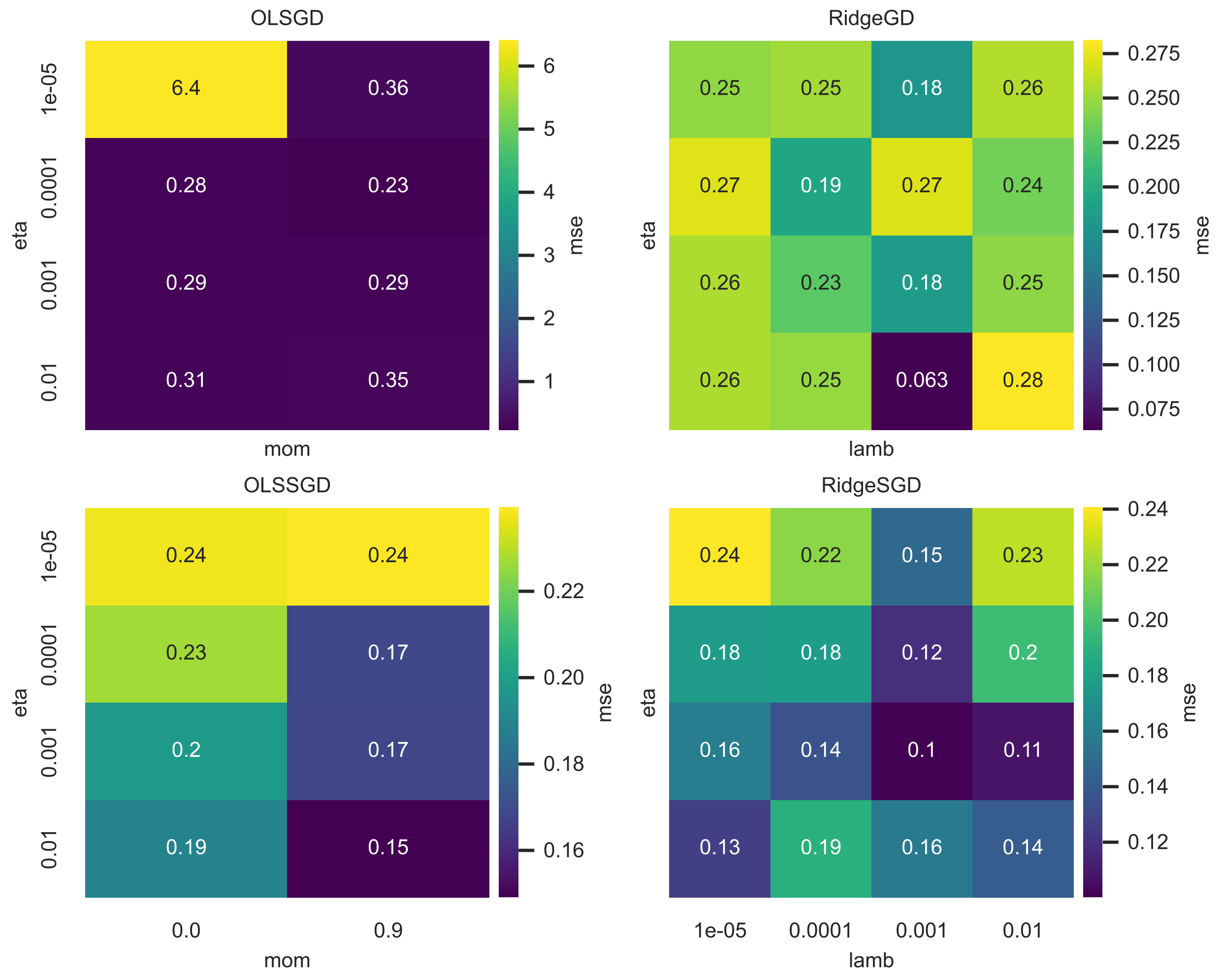

Figure 6: Heat maps showing the resulting MSE with different hyperparameters for the runs. Darker colors represent lower MSEs.

Run	**Run test MSE**	**Equiv. SKL**
constant	0.087	18.519
adam	0.012	20.743
var-η	0.016	270.376

Table II: MSEs from the different runs together with the equivalent MSE for the MLPRegressor method in SKL.

var-η has the best prediction, with an MSE of 0.011. The worst fit is the constant model, with an MSE of 0.087, and the adam model has a final MSE of 0.012. Comparing these to the SKL results is not useful, as they are clearly terrible. We come back to this in the next section.

C. ANN for classification

We set up the ANN with one hidden layer containing 100 hidden neurons. Since the ANN will now be used for classification, the activation function for the hidden layer and the output layer is the Sigmoid function. The cost function is the log-loss function. Both the ANN and the logistic regression model (LogReg) are set up to run 10^4 epochs, with a minibatch size of 100. In Figure 9, we show the result from the search for optimal hyperparameters. We notice that the LogReg model has the same score of 0.97 for $\alpha \in [10^{-6}, 10^{-2}]$, for any values of η between 10^{-6} and 10^{-1}. The ANN model performs the best with an accuracy score of 0.93 for $\alpha = 10^{-3}$ and $\eta = 10^{-4}$. The convergence of the accuracy scores is shown in Figure 10. The LogReg model converges to 0.974 after 21 epochs, while the ANN never changes from 0.623, i.e. the score is the same after every epoch. This is seemingly the same for the equivalent model from SKL, which obtained an accuracy score of 0.632.

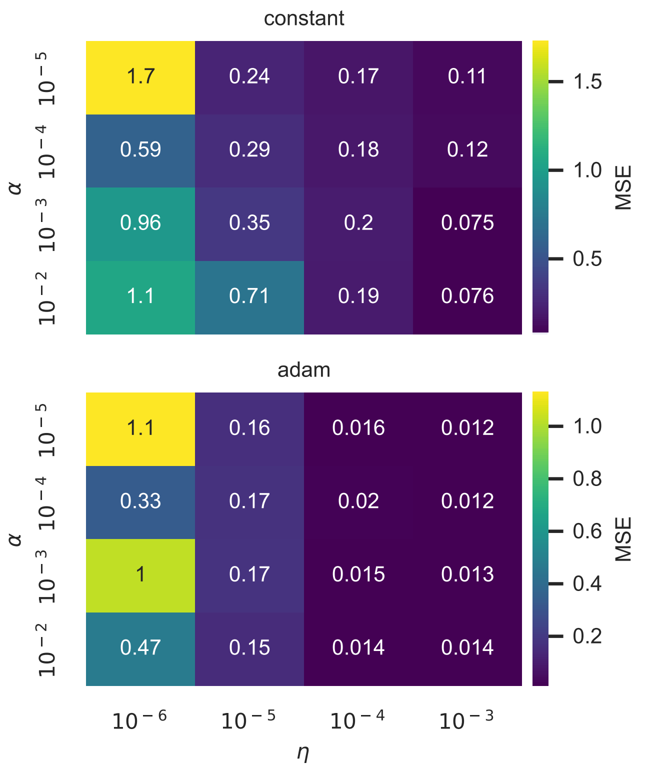

Figure 7: Heat map showing the resulting MSE with different hyperparameters for runs constant and adam.

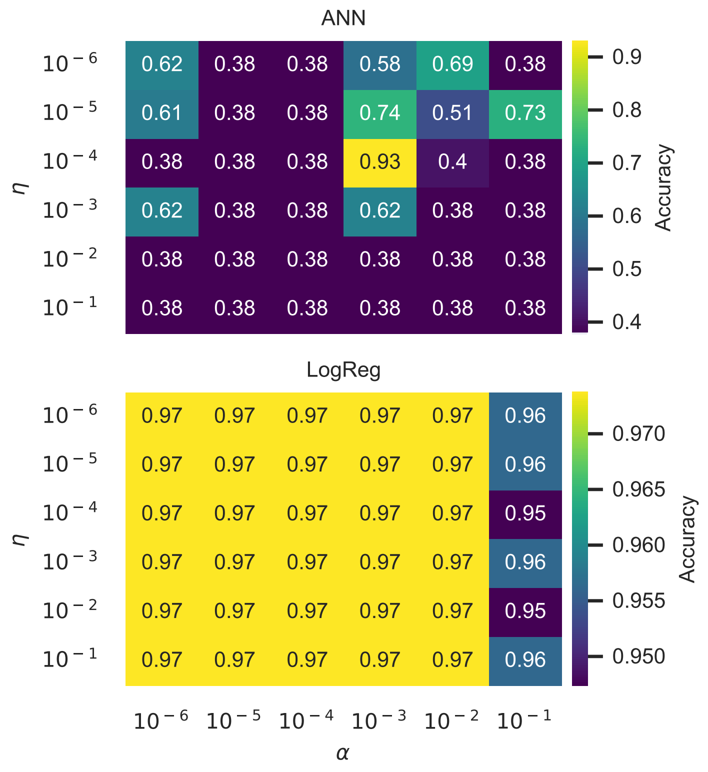

Figure 9: Heat map showing the accuracy scores for different hyperparameters for the ANN and LogReg models.

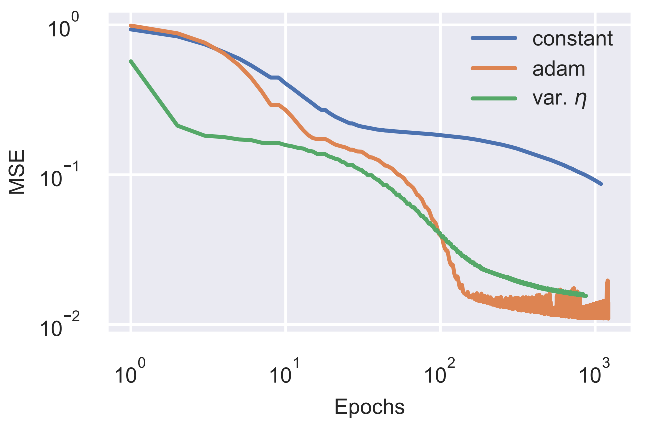

Figure 8: Convergence of the MSEs as functions of the number of epochs.

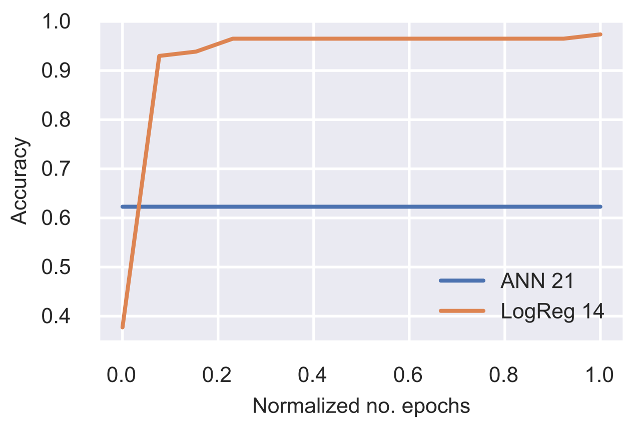

Figure 10: Convergence of the accuracy as a function of the normalized number of epochs.

IV. DISCUSSION

A. Linear regression

In Figure 6, the MSEs of all of the linear regression models indicate that they all can be used on relatively complex data, without risking very bad results. However, the optimal hyperparameters have to be found prior to using the selected model. With the direct use of the model and hyperparameters that yielded the lowest MSE, we get a better prediction than SKL. This can be explained by the fact that the implementation of the SKL model is biased towards letting SKL choose the parameters itself. This is of course not how one would do it if the goal was to actually see how well a model compares to SKL, but we are more interested in how the models we present compare to each other, rather than to SKL. From Table I, we see that the OLSGD model obtains the lowest MSE of 0.26240, but Figure 5 reveals that it took over 5000 iterations, whereas the runner-up RidgeSGD results in an MSE of 0.27814 in 13 epochs. This result shows just how important model selection is, where one would consider both computational cost and runtime when performing linear regression on a given dataset.

B. Regression with ANN

With the implementation of the Adam optimizer with SGD, we find that the prediction improves, as shown in Figure 8. While the plain SGD method gave good results, both the Adam and var-η runs converge faster. We find this result expected since the constant un-adapted learning rate does not benefit from the model knowing anything during the iterations. As Algorithm 2 shows, the momentums are computed and corrected in each iteration when using the Adam optimizer. Also, when using the learning-rate scheduler described by Eq. 1, the model is sensitive to the number of iterations. A downside of this can of course be that the model does not converge, but since the parameters t_0 and t_1 are tunable, one can modify these to the problem at hand. In Table II, we see that both the adam and var-η runs yield better MSEs than run constant, while it is crucial to note that this is not general. The table also shows a huge difference between our models and the equivalent models from SKL. Different from the way we did in the linear regression cases, we now make the models as equal as we can. It is therefore greatly unsettling that the results are as bad for the SKL as they are. However, we are aware that our initialization of the SKL parameters could have been wrong, without being able to locate where this has happened. Our guess is that when setting the maximum number of iterations, even though SKL should read this as the number of epochs, can cause the overall number of iterations to be modified, i.e. counting both the number of epochs and elements of the minibathces. A lot more analysis has to be made to fully understand this.

Unfortunately, we were unable to compare the ANN to our linear regression model, due to a major incident regarding version control of our code. Just when we had a working model (the model that the results in Section III A are obtained from), we discovered an error in the way we implemented the SGD algorithm. Sadly, we were unable to sort the problem out, and when we tried to recover the working model, we did not succeed.

C. Classification

Our results from the classification of cancer data show that our ANN fails with the parameters we used, and the LogReg model seems to have some issues as well. Especially confusing is the fact that the ANN yielded good accuracy for $\alpha = 10^{-3}$ and $\eta = 10^{-4}$, but when trying to recreate the model to get the convergence, we see that the ANN actually does not improve from 6.23 at all. Of course, this could indicate an error in the classification implementation of the source code, which is highly probable, as the authors had a hard time reaching the deadline of this paper. The reason we label the issue as not located is that the SKL model yields roughly the same score with the same parameters. This suggests that perhaps the number of epochs is not sufficient, or that the hyperparameters need finer tuning. The latter should in general not be a big problem to fix, but with the situation being as is, we were unable to do so. In contrast, our LogReg model performs well, with the resulting accuracy of 0.974 after 14 epochs. As indicated by the heat map for the LogReg model in Figure 9, a learning rate and L2 value of 0 could actually yield the best result. Again, this could easily be implemented.

V. CONCLUSION

We have presented in this paper regression methods for both linear and logistic regression, including an ANN. For linear regression, we created our own dataset and tested our models. We found that using an ANN works well when compared to a linear regression method like OLS and Ridge. In particular, we found that an ANN with the SGD method and Adam optimizer converges much faster towards a low MSE than the constant learning rate method, as seen in Figure 8. The un-optimized method with varying η following a scheduler (Eq. 1) is not far behind, with an MSE of 0.016 compared to the SGD+Adam's 0.012.

We also used our ANN to classify the WBC dataset and compared it with a logistic regression. Here, we found that our implementation of the classifier could be inadequate and benefit from a review. When classifying the tumors, the logistic regression code converged to an accuracy score of 0.974 after 21 epochs, while the ANN remained constant at 0.623.

Future work from the authors will include fixing the errors and bugs in the codes so that they can be used together when solving numerical problems. Also, we suggest researchers take the time to unify the methods presented in this paper.

REFERENCES

Amari, S., Neurocomputing **5**, 185 (1993).

Dubey, S. R., Singh, S. K., and Chaudhuri, B. B., CoRR **abs/2109.14545** (2021), 2109.14545.

Gardner, M. W.and Dorling, S. R., Atmos. Environ. **32**, 2627 (1998).

Goodfellow, I., Bengio, Y., and Courville, A., *Deep Learning* (MIT Press, 2016) `http://www.deeplearningbook.org`.

Gurbuzbalaban, M.and Hu, Y., in *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics*, Proceedings of Machine Learning Research, Vol. 130, edited by A. Banerjee and K. Fukumizu (PMLR, 2021) pp. 2233–2241.

Hoerl, A. E.and Kennard, R. W., Technometrics **42**, 80 (2000).

LeCun, Y., Bengio, Y., and Hinton, G., Nature **521**, 436 (2015).

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E., Journal of Machine Learning Research **12**, 2825 (2011).

Sazli, M. H., Commun. Fac. Sci. Univ. Ank. Series **50**, 11 (2006).

Silva, G.and Rodriguez, P., "Optimal hyperparameter ϵ for adaptive stochastic optimizers through gradient histograms," (2023), arXiv:2311.11532 [cs.LG].

Wolberg, W., "Breast Cancer Wisconsin (Original)," UCI Machine Learning Repository (1992), DOI: https://doi.org/10.24432/C5HP4Z.

Xu, J., Li, Z., Du, B., Zhang, M., and Liu, J., in *2020 IEEE Symposium on Computers and Communications (ISCC)* (2020) pp. 1–7.

Appendix A: Algorithms

Algorithm 1 Stochastic gradient descent algorithm

```
Require: Initial parameter $\boldsymbol{\theta}$
Require: Learning rate schedule $\eta_1, \eta_2, \eta_3 \dots$
  $k \leftarrow 1$
  while stopping criterion not met do
      Sample a minibatch
      Compute the gradient: $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{\theta}}$
      Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta_k \boldsymbol{g}$
      $k \leftarrow k + 1$
  end while
```

Algorithm 2 The ADAM optimizer algorithm

```
Require: Initial parameter $\boldsymbol{\theta}$
Require: Learning rate $\eta$
  Initialize 1st and 2nd moment variables $\boldsymbol{m}_1 = \boldsymbol{m}_2 = 0$
  Initialize decay rates $\rho_1 = 0.9$, $\rho_2 = 0.999$
  Initialize small constant $\epsilon = 10^{-8}$
  Initialize time step $t = 0$
  while stopping criterion not met do
      Compute gradient: $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{\theta}}$
      $t \leftarrow t + 1$
      Update first moment: $\boldsymbol{m}_1 \leftarrow \rho_1 \boldsymbol{m}_1 (1 - \rho_1) \boldsymbol{g}$
      Update second moment: $\boldsymbol{m}_2 \leftarrow \rho_2 \boldsymbol{m}_2 (1 - \rho_2) \boldsymbol{g} \circ \boldsymbol{g}$
      Correct first moment: $\hat{\boldsymbol{m}}_1 \leftarrow \frac{\boldsymbol{m}_1}{1 - \rho_1^t}$
      Correct second moment: $\hat{\boldsymbol{m}}_2 \leftarrow \frac{\boldsymbol{m}_2}{1 - \rho_2^t}$
      Compute update: $\Delta\boldsymbol{\theta} = \eta \frac{\hat{\boldsymbol{m}}_1}{\sqrt{r} + \epsilon}$
      Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \Delta\boldsymbol{\theta}$
  end while
```

Algorithm 3 Momentum algorithm

```
Require: Initial parameter $\boldsymbol{\theta}$
Require: Learning rate $\eta$
Require: Momentum variable $\gamma$
  Initialize change $\Delta\boldsymbol{\theta} = 0$
  while stopping criterion not met do
      Compute gradient: $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{\theta}}$
      Update change: $\Delta\boldsymbol{\theta} \leftarrow \eta \nabla_{\boldsymbol{\theta}} + \gamma \Delta\boldsymbol{\theta}$
      Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \Delta\boldsymbol{\theta}$
  end while
```