Exploration of Machine Learning Methods Applied to Regression and Classification Tasks

Jonas Jørgensen Telle[1], Marius Torsheim[1], and Maria Klüwer Øvrebø[2]
[1]*Department of Physics, University of Oslo, N-0316 Oslo, Norway*
[2]*Department of Mathematics, University of Oslo, N-0316 Oslo, Norway*
(Dated: November 2, 2024)

Abstract: We explore different approaches to fitting continuous and discrete data. On continuous data, we compare the performance of various gradient descent methods, linear regression methods and a neural network, and evaluate the time-efficiency of gradient descent methods and their robustness to changes in learning rate. We find that Adam is the most robust and that our neural net outperforms a degree 5 linear regression on the Franke function. On classification tasks, we compare a neural network to logistic regression on the Wisconsin Breast Cancer Dataset [1], obtaining an accuracy of 97.37%. In this simple case, we find no advantage in using a neural network over standard logistic regression. In both cases, we analyze the effect of various hyperparameters such as the activation function for the hidden layer, the learning rate and the regularization parameter using grid-search. A core part of this project is the class implementation of common gradient descent methods and a Feed-Forward Neural Network, which are used to perform the analyses.

I. INTRODUCTION

Machine learning has transformed how complex predictive problems are addressed across disciplines. This report focuses on developing and evaluating fundamental machine learning techniques for classification and regression tasks, motivated by the potential of these methods to optimize decision-making processes in real-world applications.

We explore various regression and classification techniques in the context of machine learning, focusing on methods for gradient descent, logistic regression and the implementation of a neural net. We begin with a class implementation of various gradient descent methods to fit simple polynomial functions, and analyze the convergence stability and efficiency of gradient descent optimizers such as Adagrad, RMSProp and Adam. In real-world applications, the optimal learning rate is rarely known, and a robust method can save time otherwise spent in search of this parameter. Our goal is therefore determining which models perform best, taking both computational cost as well as robustness at sub-optimal learning rates into consideration.

We then implement our own feed-forward neural network (FFNN), which we use both for continuous data fitting and classification tasks. We use the Franke function, using linear regression as a comparison to benchmark the performance of the neural network, and analyze the effect of different hyperparameters using cross-validation and grid-search from `scikit-learn` [1]. Importantly, both our class implementations are made to be compatible with these tools, making it particularly easy to find good parameters for a given problem.

This neural network is then used to classify tumors from the Wisconsin Breast Cancer Dataset, where we again analyze the effect of different hyperparameters. These results are compared to those obtained using logistic regression, implemented as a neural net with no hidden layers.

The report is organized as follows: It begins with an exploration of the methodology, detailing the design and coding of regression and classification algorithms. Subsequent sections present the theoretical background of our work, then experimental results, and finally a critical evaluation of these results with recommendations for future work.

II. THEORY

A. Universal Approximation Theorem

The Universal Approximation Theorem states that a feed-forward neural network with a single hidden layer with a non-constant, bounded, monotonically-increasing, continuous activation function can approximate any continuous function to an arbitrary level of accuracy, given a sufficient number of neurons and iterations [2]. Importantly, it does not guarantee that we can actually *find* this approximation, nor does it guarantee that a specific number of iterations will be sufficient in all cases; it is only an existence theorem. Still, it underscores the capability of neural networks to capture complex data patterns, forming the theoretical foundation for their use in a wide range of fields. While deep networks theoretically are not required for universal approximation, in practice, deep architectures might capture hierarchical representations that improve learning efficiency. Still, we find that in our relatively simple test cases, a single layer is sufficient with more complexity adding no improvement to performance.

B. Backpropagation and Gradient Descent

Backpropagation is the algorithm used to compute the gradient of the loss function with respect to each weight in a neural network, enabling efficient optimization. This

method operates through reverse-mode differentiation, using the chain rule to calculate gradients layer-by-layer from the output back to the input.

1. **Chain Rule in Backpropagation:** For each weight w_{ij} connecting neuron j in layer l to neuron i in layer $l+1$, the partial derivative of the error C with respect to w_{ij} is given by:

$$\frac{\partial C}{\partial w_{ij}} = \frac{\partial C}{\partial z_i^{(l+1)}} \frac{\partial z_i^{(l+1)}}{\partial w_{ij}}, \tag{1}$$

   where $z_i^{(l+1)}$ represents the input to neuron i in the subsequent layer.

2. **Gradient Descent Weight Update:** In Gradient Descent (GD), weights are updated iteratively to minimize the error by moving in the direction opposite to the gradient:

$$w^{(t+1)} = w^{(t)} - \epsilon \frac{\partial C}{\partial w}, \tag{2}$$

   where ϵ is the learning rate.

The update rule for the biases is obtained similarly.

As a first improvement on basic GD, one may introduce a momentum parameter m and use the update rule:

$$\mathbf{v}_{t+1} = m\mathbf{v}_t + \epsilon \nabla_\theta C,$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \mathbf{v}_{t+1}.$$

This adjustment might make the algorithm converge faster by avoiding oscillations, especially in areas with steep gradient changes.

C. Stochastic Gradient Descent and Adaptive Optimizers

Stochastic Gradient Descent (SGD) enhances GD by introducing randomness through mini-batch processing, allowing the network to avoid local minima and potentially converge faster by updating weights based on a subset of the training data. The mini-batch size then becomes another tunable hyperparameter.

Adaptive optimization algorithms like Adagrad, RMSprop, and Adam dynamically adjust learning rates based on past gradient information, thereby improving the stability and convergence rate of the model.

1. **Adagrad:** This method adjusts learning rates based on the accumulation of past squared gradients and is particularly a great choice for convex optimization problems. The update rule is:

$$\mathbf{r}_t = \mathbf{r}_{t-1} + (\nabla_\theta C_t)^2,$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \epsilon \frac{\nabla_\theta C_t}{\delta + \sqrt{\mathbf{r}_t}}.$$

   We find that this method is extremely robust to high learning rates on simple convex problems, since the accumulated squared gradients counteract a large learning rate over time. For the same reason, it is slow for small learning rates. If used with SGD, one can reset the accumulated square gradient after each epoch, which comes at a cost of robustness but might increase the rate of convergence for smaller learning rates.

2. **RMSprop:** This optimizer maintains a moving average of squared gradients, stabilizing learning rates over time.

$$\mathbf{r}_t = \rho \mathbf{r}_{t-1} + (1-\rho)(\nabla_\theta C_t)^2,$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \epsilon \frac{\nabla_\theta C}{\sqrt{\mathbf{r}_t + \delta}}.$$

   It is less stable than Adagrad for large learning rates, but faster when the learning rate is small, as is intuitive since the moving average will not steadily grow like the accumulated squared gradient.

3. **Adam:** Adam combines momentum with adaptive learning rates by computing moving averages for both gradients and squared gradients [3]. For each time step t, these moving averages s_t and r_t are calculated as follows:

$$\mathbf{s}_t = \rho_1 \mathbf{s}_{t-1} + (1-\rho_1)\nabla_\theta C_t,$$
$$\mathbf{r}_t = \rho_2 \mathbf{r}_{t-1} + (1-\rho_2)(\nabla_\theta C_t)^2,$$

   where ρ_1 and ρ_2 are decay rates. Then, the parameters are adjusted

$$\hat{\boldsymbol{s}}_t = \frac{\mathbf{s}_t}{1-\rho_1^t},$$
$$\hat{\boldsymbol{r}}_t = \frac{\mathbf{r}_t}{1-\rho_2^t},$$

   and the weight update rule for Adam is

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \epsilon \frac{\hat{\boldsymbol{s}}_t}{\sqrt{\hat{\boldsymbol{r}}_t} + \delta}. \tag{3}$$

   We find that Adam is a balanced medium of the prior methods, working quite well both for small and high learning rates.

D. Automatic Differentiation

Automatic differentiation tools, such as `Autograd` [4] and `JAX` [5], simplify the gradient computation process in neural networks. `Autograd` enables easy differentiation by tracking operations and calculating gradients automatically. These tools are crucial for deep learning research, offering flexibility in model development by decreasing development time at a small increase in computation time, a worthwhile investment on modern hardware.

III. METHOD

A. Gradient Descent Methods

We developed a custom `scikit-learn` compatible class implementation of several gradient descent methods, allowing for easy adjustment of parameters, cost functions and stopping criteria, enabling in depth analyses using `GridSearchCV` from `scikit-learn` [1].

In our analysis of convergence properties of different GD methods, we use a simple quadratic function $f(x) = 1 + 2x + 3x^2$. The gradient of the Mean Squared Error for Ordinary Least Squares (OLS) regression was implemented analytically:

$$\nabla_{\boldsymbol{\theta}} \text{MSE}_{\text{OLS}} = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}), \tag{4}$$

where $\mathbf{X}$ represents the input data matrix, $\boldsymbol{\theta}$ denotes the model parameters, and $\mathbf{y}$ is the target vector. For Ridge regression, an additional regularization term with hyperparameter λ is added to the gradient:

$$\nabla_{\boldsymbol{\theta}} \text{MSE}_{\text{Ridge}} = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}) + 2\lambda\boldsymbol{\theta}. \tag{5}$$

For a more in depth discussion of OLS and Ridge regression, we refer to our previous work [6].

Building on this prior work, we computed the analytical optimal coefficients $\boldsymbol{\beta}$ and the Hessian matrix, from which we obtain the greatest eigenvalue and an upper bound $\epsilon_0 = 1/\lambda_{max}$ on our learning rate with non-adaptive methods. With these analytical values, we can, importantly, ensure fair time comparisons between models by using a custom stopping criterion requiring that the final parameters $\boldsymbol{\theta}$ are within a small distance of $\boldsymbol{\beta}$, and test a range of learning rates less that ϵ_0 with non-adaptive methods without fear of divergence. For the adaptive models, we test learning rates well above this threshold, and still achieve fast convergence. Note that we keep the other hyperparameters fixed at the values recommended in *Deep Learning* by Goodfellow, Bengio and Courville [7], and use a mini-batch size of 16 throughout the analysis.

We then measure the number of iterations and time in milliseconds to convergence for all models as a function of learning rate. The number of iterations of a GD-method is not directly comparable to the number of epochs in a SGD-method, since a single epoch updates the parameters multiple times and is much more computationally intensive than a single GD iteration. Therefore, the timer-function is important, ensuring a fair comparison. We do fits for learning rates ranging from $10^{-3}\epsilon_0$ to ϵ_0. With adaptive models, we allow the learning rate to go as high as $10,000$ to explore their robustness.

Note that we, for these models, do *not* reset the accumulated gradients between epochs, since this seems to decrease robustness dramatically by inhibiting the model's ability to recover from bad learning rates. Still, the option for using this reset can be adjusted in the class implementation. In `scikit-learn`, the adaptive `SGDRegressor` will by default use the entire dataset as a single batch, and thus also does not reset the gradients - if it did, there would be no adaptive learning rate [1]. We then plot MSE as a function of number of epochs for momentum SGD, AdaGrad and `scikit-learn`'s adaptive `SGDRegressor` for different learning rates, showing clearly the ability of the adaptive methods to recover from bad learning rates.

We run a similar analysis for Ridge regression, now also testing different regularization parameters. Note that the analytical learning rate changes as a function of the regularization, as does the analytical optimal parameters. Finally, we use autograd to evaluate the cost gradients to gauge how much more time is spent on calculation when using automatic differentiation.

B. Neural Network Implementation

We constructed a feed-forward neural network (FFNN) for both regression and classification tasks. The implemented neural network model is designed with flexibility in mind, allowing for custom configurations of hidden layers, activation functions, and optimization settings. This adaptable structure facilitated various experiments, including regression with the Franke function and classification of the Wisconsin Breast Cancer dataset. The neural network was implemented as a class compatible with the `scikit-learn` API by inheriting from `BaseEstimator`, `RegressorMixin`, and `ClassifierMixin`, making it compatible with `Pipeline` and `GridSearchCV` for efficient model evaluation and parameter tuning.

The choice of cost function was tailored to each predictive task: For regression, the FFNN minimized the MSE, and for classification, cross-entropy loss was used, guiding the model to minimize prediction errors in class distribution. For the hidden layers, multiple activation functions including sigmoid, ReLU, leaky ReLU and ELU were tested to understand their effects on performance.

The network's weights and biases were initialized using a normal distribution, introducing slight random variation to prevent symmetry in early training, and were updated using backpropagation and gradient descent. The gradients were calculated using either analytical or automatically differentiated expressions for the loss and activation functions using the chain rule backwards as outlined in II B. The updates were performed using stochastic gradient descent.

In applying the model to a given problem, a pipeline was constructed using `Pipeline`, where the data was first scaled and then fitted using the neural network. This pipeline was then used when finding the optimal parameters using `GridSearchCV` [1].

Regression: Franke's function

We first applied the FFNN on a regression problem, aiming to reproduce the Franke function. We used MSE as cost function and a linear activation function in the final layer. The dataset was scaled using the `StandardScaler` from `scikit-learn` [1].

The model used 50 nodes for the hidden layer and 500 epochs. Hyperparameter tuning was conducted with `GridSearchCV` to determine optimal values for the learning rate, regularization parameter λ, and the activation function for the hidden layer. Note that the learning rate $\epsilon = 0.1$ was only used in combination with a sigmoid activation function as preliminary testing found that ReLU and leaky ReLU diverge for so large a learning rate in this problem.

The effect of the batch size was examined through the MSE as a function of the learning rate for different batch sizes. Preliminary testing showed similar behavior for different batch sizes, so only a batch size of 512 was used for the gridsearch to reduce computational cost. When measuring the performance of the model on the test dataset, the batch size was reduced to 32 to improve performance.

After finding near-optimal hyperparameters with gridsearch, the performance of the best model was tested on the test dataset. The performance of this model was also compared to that of an implementation of `PyTorch` with the same architecture and hyperparameters [8]. A study of the performance of these two models was also performed, by studying the MSE as a function of the learning rate.

Classification: Wisconsin Breast Cancer Dataset

When studying the Wisconsin breast cancer dataset the FFNN was used for classification. The softmax activation function was employed in the output layer, translating raw network predictions into probabilities suitable for binary classification. To calculate the derivative of softmax, the jacobian was computed and an einsum was used to obtain the final expression [9]. The dataset was scaled using the `MinMaxScaler` from `scikit-learn` [1].

Additionally, we created a comparison model using `MLPClassifier` from `scikit-learn` by building a neural network with the same architecture and parameters as the best model found with gridsearch for our FFNN. We also compared the two models' behavior as a function of the learning rate.

For the classification task, model accuracy was the primary metric. The accuracy score was calculated as:

$$\text{Accuracy} = \frac{1}{n}\sum_{i=1}^{n} \mathbb{I}(t_i = y_i),$$

where $\mathbb{I}$ is the indicator function, t_i represents the true label and y_i the predicted output. This metric provided a direct measure of predictive correctness, calculated across training and test datasets to monitor generalization. For data which heavily skewed, e.g. where less than 1% of the data is of one binary class, this metric may be a poor choice, but in our dataset there is a relatively even split with 212 malignant and 357 benign tumors [1]. For real world medical applications, most people are presumably healthy and other metrics should be used. Either way, a confusion matrix was constructed, displaying the distribution of true and false positives and negatives to ensure that the model produced useful classifications.

To identify optimal network configurations, hyperparameter tuning was performed with `GridSearchCV`. Parameters such as the learning rate, regularization parameter, the number of nodes and activation functions for the hidden layer were systematically tested to evaluate their impact on model accuracy. Additionally, we tested the impact of the number of hidden layers. The impact of the batch size was studied by comparing the accuracy of a specific neural network for different batch sizes as a function of the learning rates.

C. Logistic Regression

We implemented logistic regression using our neural net with no hidden layers. We used cross entropy as the loss function and softmax as activation function, even though simpler options are available for the binary case. To find the optimal hyperparameters we used the `GridSearchCV`-function from `scikit-learn`. The model parameters were a batch size of 10 with 500 epochs, and we tested a grid of values for the learning rate and regularization parameter. When the optimal model parameters were found, the optimal model was evaluated on the test dataset. A comparison was made against `scikit-learn`'s `LogisticRegressor` using the Limited-memory BFGS-solver ('lbfgs') and 500 epochs [1].

IV. RESULTS

A. Convergence Properties of Gradient Descent Methods

The convergence times in milliseconds for GD and SGD with and without momentum on a second order polynomial regression problem are displayed in Figure 1 as a function of learning rates at or below the analytical upper bound ϵ_0. Unsurprisingly, we see that, below this bound, higher learning rates lead to faster convergence and that both momentum and mini-batching improves efficiency. Compared with plain GD, we obtain nearly 10 times faster convergence with MomentumSGD at each of the displayed learning rates.

Figure 2 shows the convergence times in milliseconds for the adaptive optimizers AdaGrad, RMSprop, and Adam across a range of learning rates, now vastly exceeding the analytical bound (here $\epsilon_0 \approx 0.1$). We observe

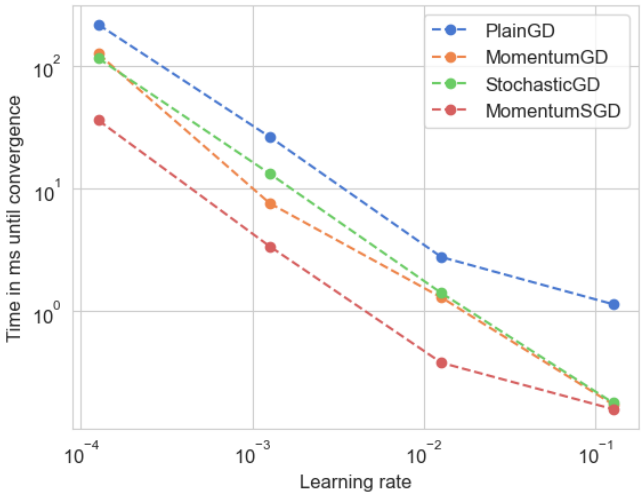

Figure 1. Convergence times for non-adaptive gradient descent methods (Plain GD, Momentum GD, SGD and Momentum SGD) across different learning rates. Adding momentum and mini-batch processing clearly improves convergence speed. Note the logarithmic axes.

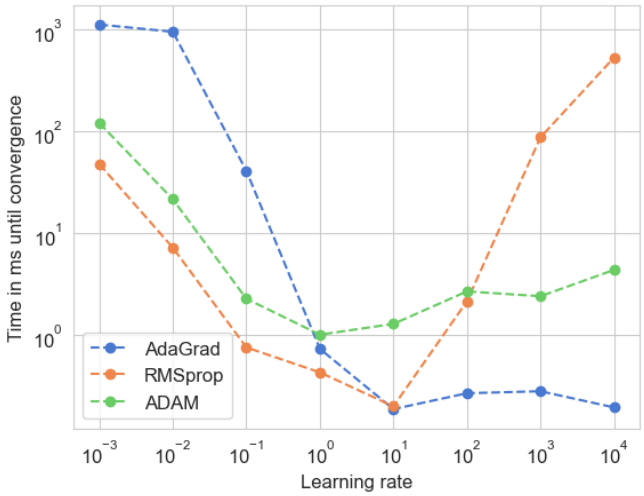

Figure 2. Convergence times in milliseconds of adaptive methods (AdaGrad, RMSprop and Adam) across varying learning rates with analytically implemented gradients.

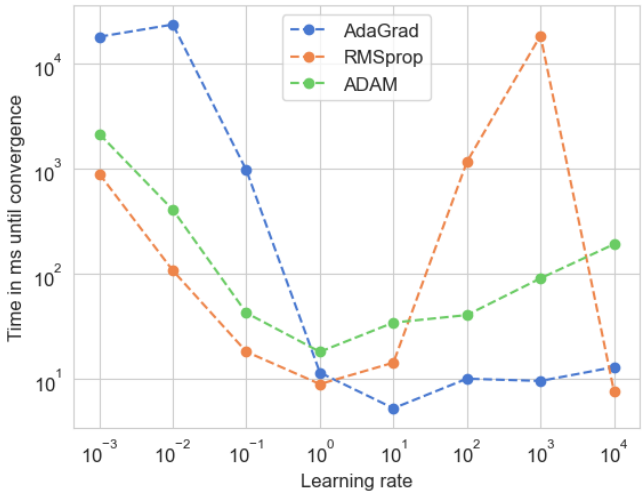

Figure 3. Convergence times in milliseconds of adaptive methods (AdaGrad, RMSprop, and Adam) across varying learning rates with automatic differentiation.

that

- Adagrad converges very fast for very high learning rates, yet performs very poorly for more typical learning rates. Recall that we do not reset the accumulated gradient between epochs; this only slightly improves performance for low learning rates while drastically decreasing robustness. We will give a more in depth analysis of this method shortly, as we initially found its behavior surprising.
- RMSprop shows relatively consistent performance across low to moderate learning rates, but becomes unstable at higher learning rates; here, randomness decides whether it performs well or not, and thus Figure 2 changes dramatically between initial conditions. To showcase this instability, one should compare with Figure 3.
- While never the fastest model, Adam showcases both stability - giving similar results across reruns - and robustness across a very wide spectrum of learning rates, and is thus our general recommendation as a first method to try on a new problem.

These results can be partly explained from the update rules of the methods, and we refer to our brief discussion in the theory section.

Figure 2 shows a similar plot, now using `autograd` to differentiate the cost functions. This gives a plot which is indistinguishable beyond random effects, due to SGD and initial conditions, from the analytical gradient case except that it is shifted up: Every convergence takes between 10 and 100 times longer. With more complex cost functions, this might be well worth it, when computation time is not critical.

We now make a more in depth analysis for select models, looking at how the MSE evolves as a function of epochs for different learning rates. First, we make a plot for momentum SGD as seen in figure 4 for learning rates proportional to ϵ_0, which reiterates what we knew already about the non-adaptive methods.

Figure 5 shows a similar plot for Adagrad with learning rates up to $1,000,000$ without resetting the accumulated gradient between epochs. Notice the instability of the model; depending on the initial parameters, with a high learning rate, it might immediately jump in the right or wrong direction. Still, it is robust, in the sense that it, after this initial jump, converges from this point with a similar slope no matter the initial learning rate. In the

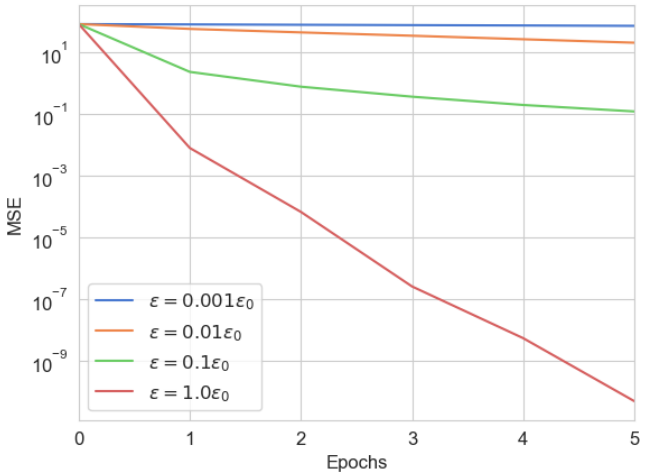

Figure 4. MSE as a function of epochs for momentum SGD for learning rates at or below the analytical bound ϵ_0, here on the order of 10^{-1}, still with batch size $B = 16$.

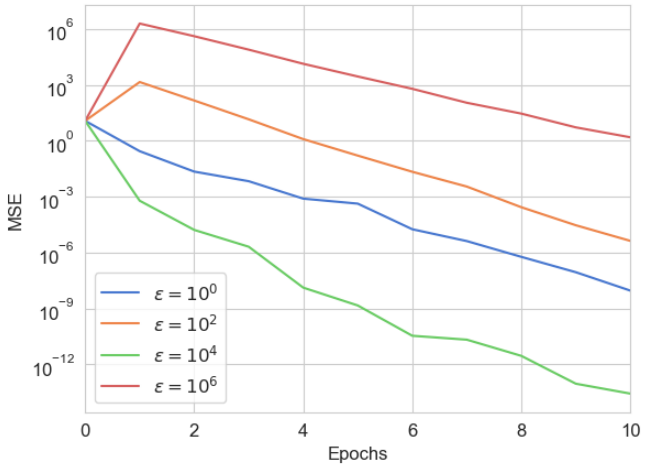

Figure 5. MSE against epochs for Adagrad for selected large learning rates with batch size 16. Notice how the accumulated gradient quickly counteracts the large learning rate to produce a steady convergence, though initially the model can get markedly off course.

specific plot of Figure 5 we see that a learning rate of $10,000$ does by far the best while $\epsilon = 1,000,000$ has a ridiculous MSE of 10^6 after one epoch. However, due to instability, reruns will give different results, and the best bet is generally a learning rate between 1 and 100, though increasing the number of epochs somewhat gives convergence either way.

Including the reset at every epoch will give similar jumps as we see at epoch 1 at every step, and thus introduces more randomness into the model. We have discussed this with an expert in the field, Morten Hjorth-Jensen, who reproduced our results by removing the reset in his own code. Additionally, `scikit-learn`'s adaptive learner exhibits similar, and arguably even stronger, robustness to high learning rates as seen in Figure 6.

Figure 7 presents convergence times in milliseconds for the adaptive optimizers AdaGrad, RMSprop, and Adam for Ridge regression across a range of learning rates and regularization parameters λ. Each panel represents a different optimizer, with convergence time values color-coded on a logarithmic scale. The general conclusion is that lower regularization leads to faster convergence to the optimal Ridge-coefficients, though the effect is not dramatic. We also clearly see the instability of RMSprop at higher learning rates, with some instances converging in less than half a millisecond while others take much longer. For Adagrad and Adam, the results are stable, though Adam is more robust across learning rates while Adagrad is more robust across regularizations.

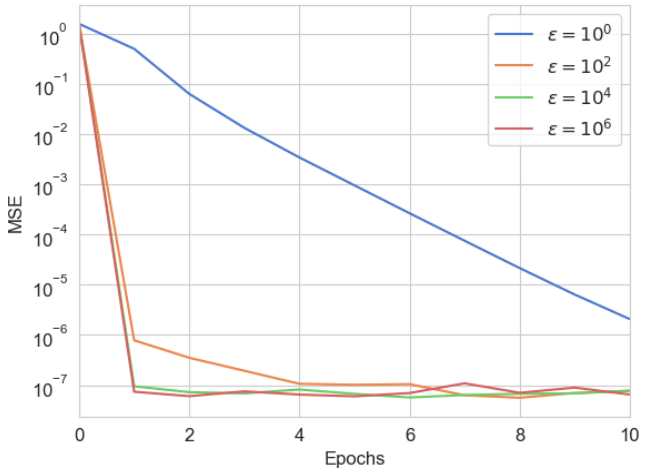

Figure 6. MSE against epochs for `scikit-learn`'s adaptive `SGDRegressor` for selected large learning rates. Notice the different scale on the axes as compared with Figure 5, and keep in mind that the batch size here is the entire dataset. The plots are still similar in showing strong robustness on simple convex problems.

B. Regression with a Neural Network

In Figure 8, the MSE is plotted as a function of the learning rate for different batch sizes with a neural network using one hidden layer with a sigmoid activation function and 50 nodes over 10 epochs. The MSE curves for different batch sizes follow similar trends, but smaller batch sizes perform slightly better.

Figure 9 compares the MSE as a function of the learning rate for our FFNN implementation and `PyTorch`'s, using the same hyperparameters (one hidden layer with sigmoid activation, 50 nodes, and 10 epochs). The `PyTorch` model is more stable at smaller learning rates, while our implementation exhibits higher variability. However, at larger learning rates, `PyTorch`'s stability decreases, showing greater variability, while our model ap-

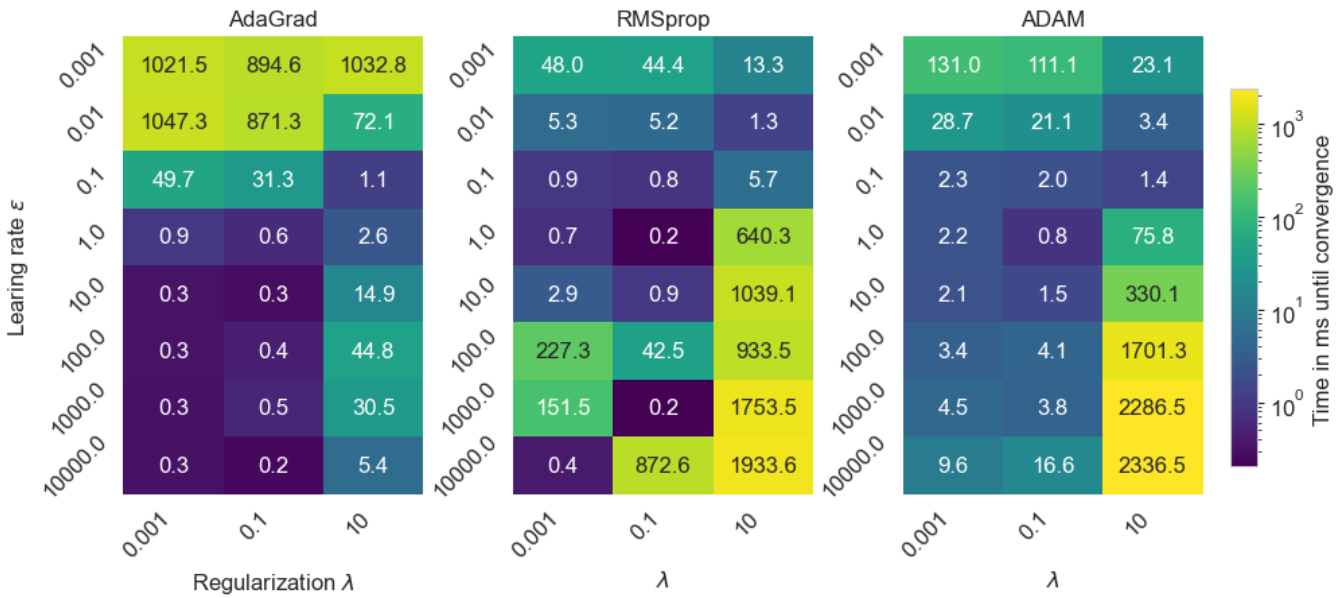

Figure 7. Time to convergence in milliseconds for different regularization parameters and learning rates for adaptive GD methods. Our goalpost for convergence is the analytical ridge parameters.

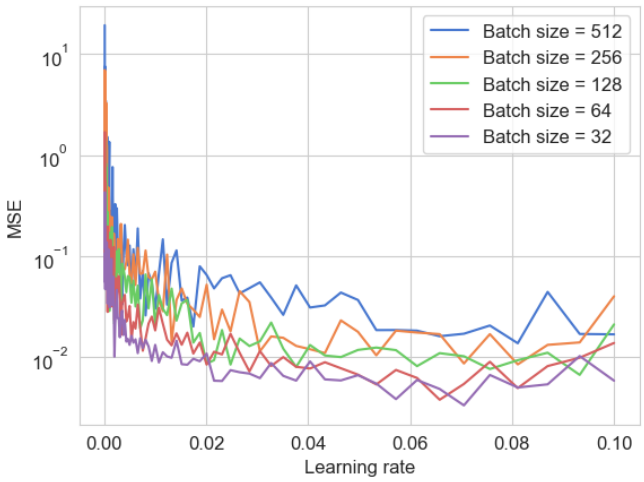

Figure 8. MSE as a function of the learning rate for different batch sizes (32, 64, 128, 256, 512) in a neural network with one hidden layer with sigmoid activation, 50 nodes and batch size 32 over 10 epochs.

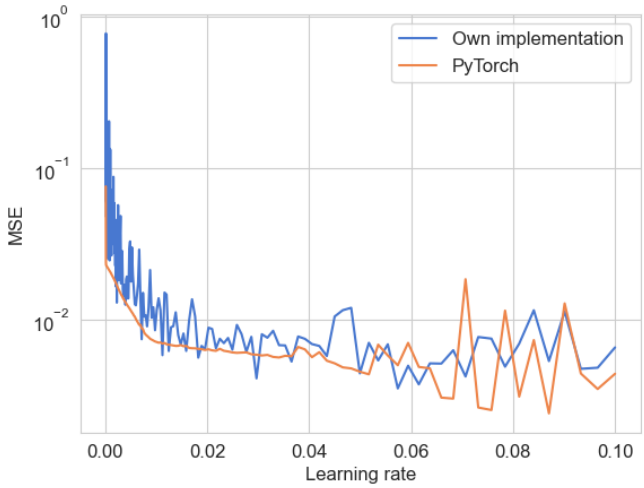

Figure 9. Comparison of MSE as a function of learning rate for our FFNN and `PyTorch`'s implementation. Neural network parameters: one hidden layer with 50 nodes, sigmoid activation, and 10 epochs.

pear to remain slightly more stable.

Figure 10 shows the negative MSE for different combinations of learning rates (rows) and regularization parameters λ (columns) for three activation functions (sigmoid, ReLU, and leaky ReLU) used in the hidden layer. This setup used a neural network with one hidden layer containing 50 nodes, a batch size of 512, and 500 epochs. The results from gridsearch indicate minimal differences across activation functions, though slight variations in performance can be observed at specific learning rates and regularization values. A learning rate of 0.1 was only used with the sigmoid function since ReLU and the leaky ReLU functions diverge for this value. Too small a learning rate and too high a regularization parameter leads to poorer performance for all activation functions. Overall, all models perform the best with a higher learning rate, though still so low as to not diverge, and no regularization, aligning with our results from the isolated analysis of non-adaptive GD methods.

The best performing model used a sigmoid activation function for the hidden layer with a learning rate of 0.1

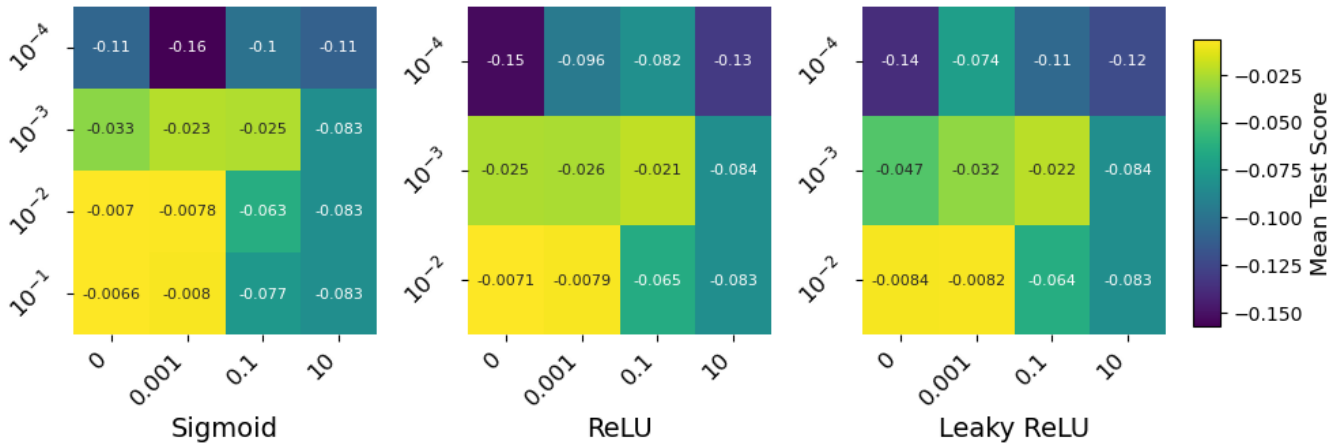

Figure 10. Negative MSE for different activation functions (sigmoid, ReLU, and leaky ReLU) across combinations of learning rates and regularization parameters λ.

and no regularization. On the test dataset, this model got a score of $2.2 \cdot 10^{-3}$ for a batch size of 512. When reducing the batch size to 32, otherwise keeping the aforementioned hyperparameters, the MSE is reduced to $8.045 \cdot 10^{-4}$. The `PyTorch` implementation using the optimal model parameters found with gridsearch had a test score of $1.2 \cdot 10^{-3}$, and our previous research shows that a fifth degree linear regression obtains at best an MSE of about $1.126 \cdot 10^{-3}$ [6] .

C. Classification

Neural Network

Figure 11 shows the accuracy of a neural network as a function of the number of epochs with SGD (model parameters: one hidden layer with sigmoid activation, batch size 10 and learning rate 0.001). The accuracy increases sharply up to approximately 50 epochs, after which the rate of improvement slows significantly, showing only marginal gains with further increases in epochs.

In Figure 12, the accuracy score as a function of the learning rate for our neural network implementation is compared to that of `MLPClassifier` from `scikit-learn`. Both implementations use one hidden layer with 50 nodes, sigmoid activation, a batch size of 10, and 10 epochs. Performance is generally similar, with `MLPClassifier` achieving slightly better stability at lower learning rates.

Figure 13 shows the accuracy as a function of learning rate for various batch sizes (10, 20, 50, 100, and 200). At low learning rates, accuracy is fairly consistent across batch sizes. However, for learning rates above 0.04, larger batch sizes display greater instability, with a notable drop in accuracy and larger fluctuations. Smaller batch sizes maintain more consistent performance across the learning rate range, but are more computationally expensive.

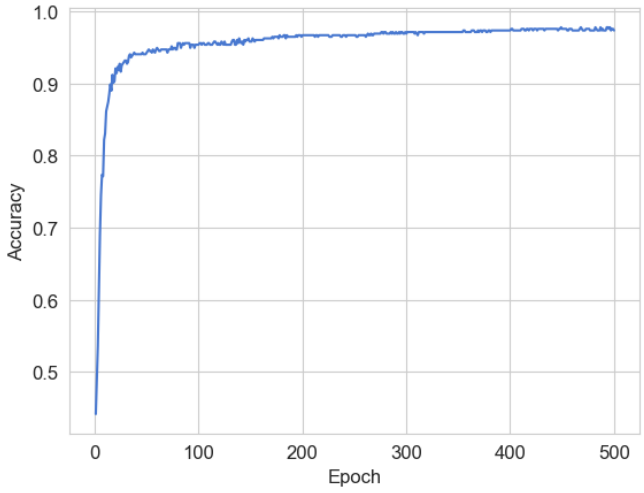

Figure 11. Accuracy as a function of the number of epochs for a FFNN with SGD for a learning rate of 0.001 using sigmoid activation for a hidden layer with 50 nodes.

Figure 17 displays a heatmap of accuracy scores obtained using `GridSearchCV` with 50 epochs and a batch size of 10 across various combinations of learning rates (rows), regularization parameters λ (columns), activation functions, and hidden layer configurations. The results reveal that most models perform similarly, with poorer performance at lower learning rates or with high regularization. A grid-search using two hidden layers gave very similar results, indicating that the one-layer model is already sufficiently complex.

When the model was retrained with 500 epochs, the best configuration was found to be one hidden layer with sigmoid activation and 100 nodes, a learning rate of 0.1, and $\lambda = 0.001$. This model had a test accuracy of 0.9737. The confusion matrix for this score is shown in Figure 14 along with the confusion matrix for the `MLPClassifier`

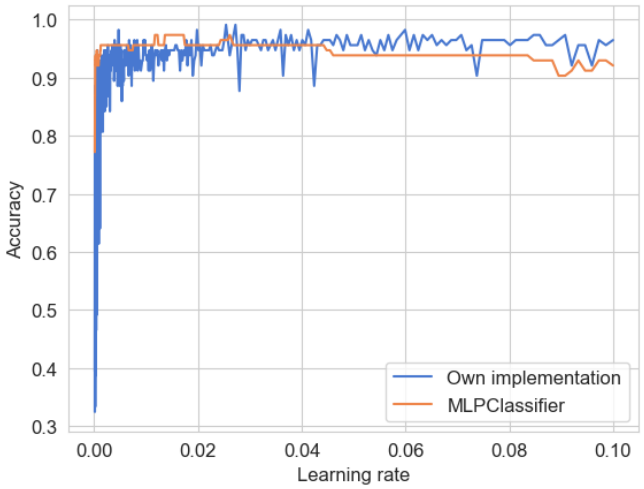

Figure 12. Comparison of accuracy scores as a function of learning rate for our FFNN and `MLPClassifier` from `scikit-learn`. Parameters: one hidden layer (50 nodes, sigmoid activation), batch size 10, and 10 epochs.

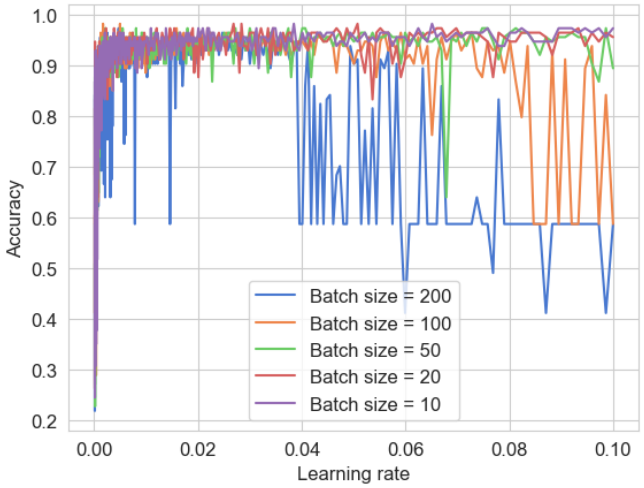

Figure 13. Accuracy as a function of learning rate for various batch sizes (10, 20, 50, 100, 200) with a neural network configured with one hidden layer (50 nodes, sigmoid activation) over 10 epochs.

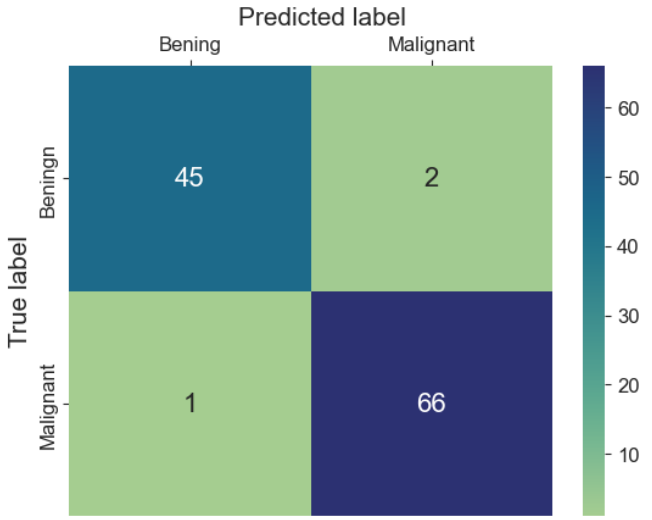

Figure 14. Confusion matrix for our FFNN with optimized parameters on the Wisconsin Breast Cancer Dataset.

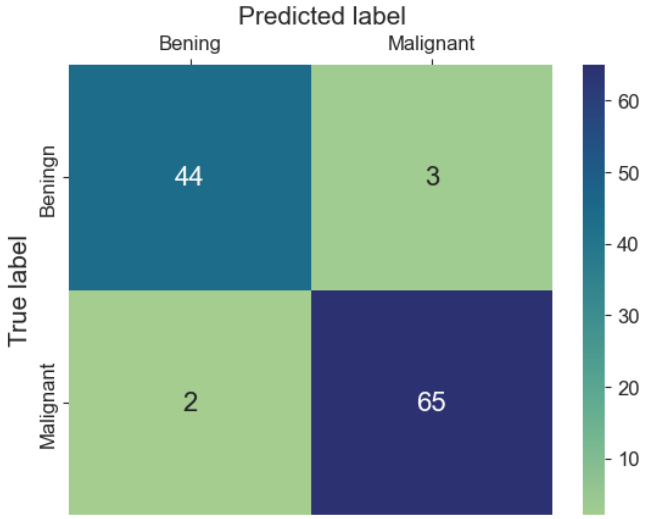

Figure 15. Confusion matrix for `MLPClassifier` from `scikit-learn` with optimized parameters.

from `scikit-learn` for the same model hyperparameters in Figure 15. The `MLPClassifier` had a test score of 0.9561. Both models perform well, with minimal misclassifications. Notably, our model shows slightly more false positives than false negatives, which is favorable in the context of breast cancer detection, where it is preferable to flag benign cases as potentially malignant rather than missing actual malignant cases. The implementation from `scikit-learn` had two more misclassifications, as seen in Figure 15.

1. *Logistic Regression*

Figure 16 shows a heatmap of the accuracy scores for different combinations of learning rates and regularization parameters using gridsearch with a neural net without a hidden layer with a batch size of 10 and 500 epochs. The results are quite similar for most of the hyperparameter combinations, with a learning rate of 10^{-4} being the only parameter to result in a significant drop in performance, resulting in a poorer accuracy for all regularization parameters. Additionally, a learning rate of 0.01 results in the highest accuracy irrespective of regularization parameter. The best model is given by a learning

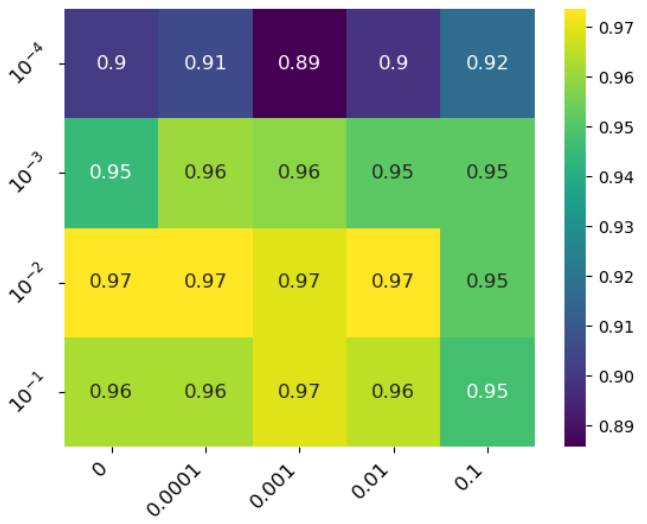

Figure 16. Heatmap of accuracy scores for logistic regression for different combinations of learning rates and regularization parameters using a neural net with no hidden layers, a batch size of 10 and 500 epochs.

rate of 0.01 and no regularization.

When testing this model on the test dataset it had a accuracy score of 0.9737. The confusion matrix was identical to that of the neural net. The test-score of `scikit-learn's LogisticRegressor` with a 'lbfgs'-solver and 500 epochs was slightly worse than our implementation with an accuracy of 0.9649.

V. DISCUSSION

A. Gradient Descent Methods

Given that our analysis was conducted on very simple artificial data, our results may not directly transfer to complex real-world situations where gradient landscapes are more challenging. Still, we try to draw out some general insights:

- **Adam's Stability and Versatility:** Adam's consistent performance suggests it is likely the most versatile optimizer for complex models, offering robustness across a range of learning rates and shows stability to initial conditions between reruns.

- **RMSprop's Speed with Careful Tuning:** While RMSprop can achieve extremely fast convergence, its high sensitivity to learning rate and regularization makes it better suited for controlled settings where learning rate tuning is feasible and reasonable guesses for the initial conditions are known.

- **AdaGrad's Unique Convergence Behavior:** AdaGrad's preference for very high learning rates, combined with its reliance on non-resetting of accumulated gradients in our experiments, suggests it may not be the best choice for complex tasks. However, for simple convex optimization problems where the results can be easily verified, AdaGrad might be the best choice.

- **Simple models:** In general, momentum and SGD should be used over basic GD, though one is likely better off using an adaptive method when the analytical upper bound on the learning rate is not known since the non-adaptive models diverge when the learning rate exceeds this value.

In our simple setting, AdaGrad shows unusual behavior, particularly requiring high learning rates to achieve optimal convergence. At these rates, AdaGrad may converge in just a single epoch due to large initial gradient steps, followed by rapid stabilization.

Notably, AdaGrad's effectiveness with such high learning rates depends on not resetting the accumulated gradient between epochs. When the gradient reset is removed from the main loop, AdaGrad can recover quickly from the large initial leaps and stabilize, achieving convergence with remarkable efficiency. However, with a reset in place, these large steps are repeated each epoch, preventing convergence and leading to instability.

We initially found this very surprising, and so discussed it with an expert in the field who was able to reproduce our results with his own code. The adaptive SGD method in `scikit-learn` shows similar robustness to high learning rates. We suspect that this relies on the simplicity of our problem, and would be surprised if our results generalize in future studies on more challenging regression tasks.

B. Regression with a Neural Network

Through this exploration of neural nets we found that the different hyperparameters have variable effect on the performance.

Activation function was found to have little impact both for continuous fitting and classification, though the sigmoid function barely came out on top. In some cases, the vanishing gradient might make this function less useful, and one might expect that the choice of activation function has more of an impact for more complex datasets and with models with more than a single hidden layer.

Regarding learning rate and regularization, we get similar results for both regression and classification: Performance is stable for moderate learning rates and regularization parameters, but drops when the regularization becomes large or the learning rate becomes small. This aligns with the findings from our study of the convergence properties of GD methods, and we might expect that including momentum would improve the results. Future studies might integrate our class implementations of GD

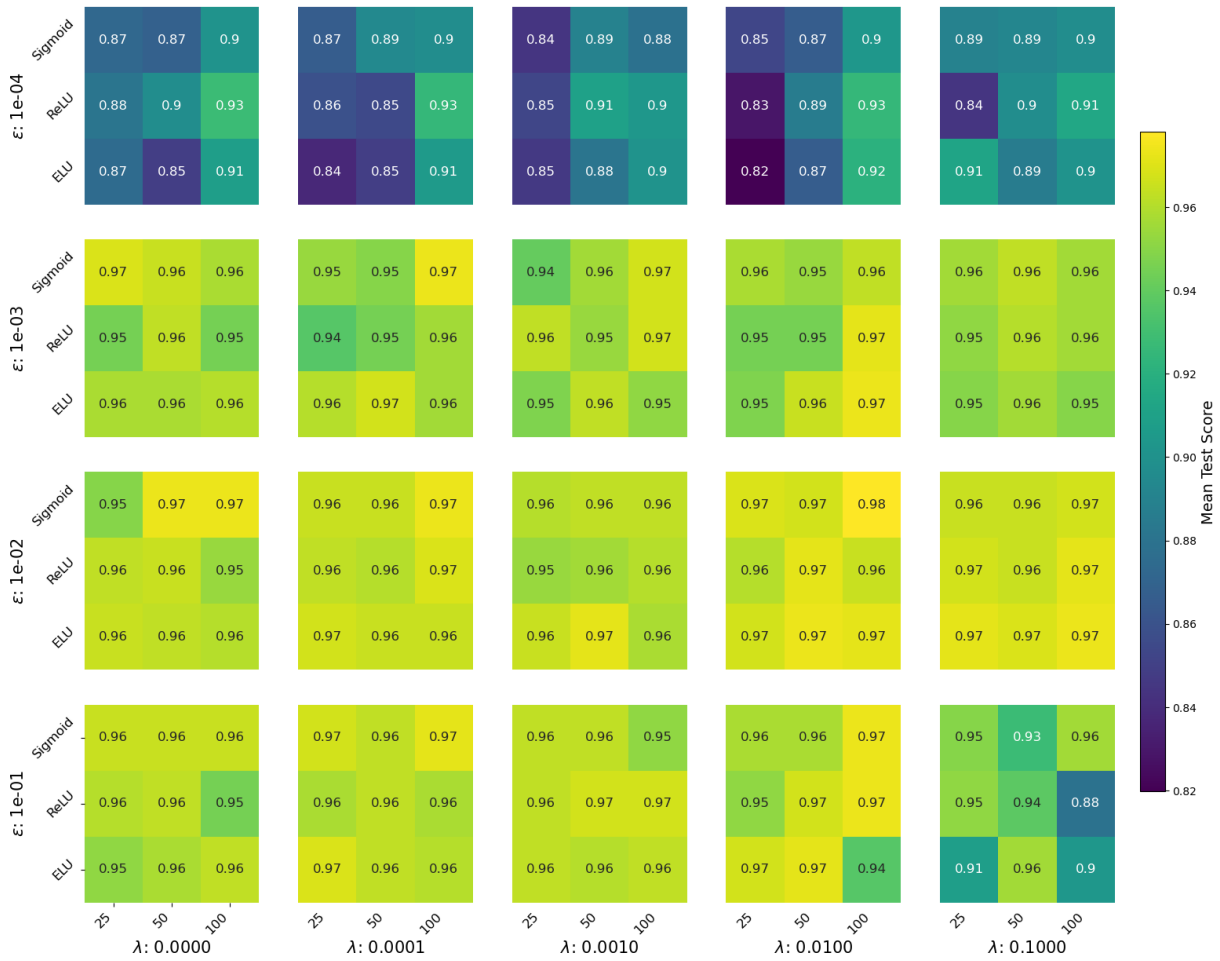

Figure 17. Heatmap of accuracy scores for various combinations of hyperparameters (learning rates, regularization parameters, activation functions, and hidden layer configurations) with a batch size of 10 and 50 epochs.

methods with our FFNN, allowing simple use of adaptive methods for backpropagation.

Smaller batch sizes slightly improve both stability and loss for both continuous fitting and classification, though at the cost of increased computational costs. Since the change in behavior is small, we believe we are justified in performing grid-search with a bigger batch size to save computational costs, and that the obtained parameters are still transferable to a lower batch size.

The findings on layer complexity suggest diminishing returns when additional layers are added, even reducing performance slightly. This observation, particularly in the context of the Wisconsin dataset, suggests potential over-fitting when using two layers, given the dataset's simple structure, where properties in the images have already been transcribed to arrays. If classifying tumors directly from images, the dimensionality is much greater and higher complexity is certainly warranted. While some two-layer configurations yielded marginally better performance, the additional complexity associated with these setups was not justified by the minimal accuracy gains.

When using neural nets and logistic regression to classify a binary dataset we got an identical test error for both to methods. Thus, for less complex cases, logistic regression is preferable, being simpler to implement and less computationally costly. Neural networks have the benefit of being more flexible and scale better to more complex problems, though at a higher computational cost. If we were to repeat this procedure with a more complex dataset, we would expect to find that the neural net outperforms logistic regression which is unable to accurately replicate the complexity.

Finally, for contionous regression tasks, one can use either a neural net or standard linear regression. The FFNN provides a great advantage in that it does not require us to set up the degree of the fit, which is practical for non-polynomic data such as the Franke function. When fitting the Franke function with a fifth degree linear regression in project 1, we got a test score of $1.126 \cdot 10^{-3}$ for OLS and $1.127 \cdot 10^{-3}$ for Ridge [6]. With a sufficiently small batch size, the neural net outperforms these methods. Still, we generally expect that, with a sufficiently large degree polynomial fit with linear

regression, one will do better or as well as with a neural net, and that a degree 5 fit on the Franke function does not provide a fair comparison. Still, the similarity in test error suggests that a well-tuned neural network is on par with traditional regression models, as we might expect from the universal approximation theorem.

VI. CONCLUSION

Our study reveals several important insights regarding the performance of neural networks and gradient descent methods on relatively simple datasets. Among the gradient descent methods tested, Adam proved to be the most robust, providing consistent and reliable performance across different scenarios. The use of `autograd` somewhat extended the computation time, taking at least ten times longer compared to analytical gradients. Additionally, omitting the reset of optimizations between epochs increased the robustness of all the adaptive models in our limited tests, making the greatest difference for Adagrad.

For our neural network, the choice of activation function did not have a substantial impact on model performance, indicating that simple neural networks can be relatively flexible with respect to this hyperparameter. Regularization did not play a crucial role when working with our small neural networks, but might be more important for more complex architectures to counteract over-fitting. It was also found that selecting an appropriate learning rate was critical. Specifically, too small learning rates required a lot of epochs to converge and were thus computationally expensive, while larger learning rates caused divergence. Future studies should explore using adaptive GD methods in training to improve robustness.

Although neural networks excelled in both classification and continuous regression tasks, the multitude of hyperparameters necessitated careful tuning to achieve optimal performance. Logistic regression performed as well on binary classification tasks, and is quick to implement and computationally efficient. For simple problems, simple solutions are often preferable; also on our simple continuous regression tasks, a high order linear regression fit seems preferable to the complexity of the neural network. Still, the simple tests provide general insights into the behavior of neural nets which should prove useful on more challenging problems.

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

[2] M. Hjorth-Jensen, "Applied Data Analysis and Machine Learning," https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/week40.html (2021).

[3] D. P. Kingma and J. Ba, arXiv preprint arXiv:1412.6980 (2014).

[4] D. Maclaurin, D. Duvenaud, M. Johnson, and J. Townsend, "Autograd," .

[5] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," (2018).

[6] J. J. Telle, M. K. Øvrebø, and M. Torsheim, "Project 1 FYS-STK3155," .

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

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," (2019), arXiv:1912.01703 [cs.LG].

[9] A. Kim, "How to implement the derivative of softmax independently from any loss function," (2017), accessed: 2024-10-17.