Regression and classification analysis using neural networks and other models

Ivar Lønning
(Dated: November 18, 2022)

We investigate a regression problem of a polynomial function using linear regression and a feed-forward neural network (FFNN) model. The model parameters were found by gradient descent methods. We discuss the performance of the various methods, including gradient descent with momentum and stochastic gradient descent. We also investigated methods to tune the learning rate in the gradient descent algorithm. The linear regression model gave the best $R2$ scores of nearly 1, although some setups of the FFNN model give scores around 0.8. We then performed a classification analysis on the Wisconsin breast cancer dataset, using both a FFNN model and a logistic regression model. Using stochastic gradient descent to find optimal parameters, we got a prediction accuracy that was close to 100% in the logistic regression model as well as for some setups of the FFNN.

(The code used is available at https://github.com/ivarlon/fys-stk4155/tree/main/project2.)

I. INTRODUCTION

We live in an information society where lots of data are constantly generated, and there is often a desire or need to make sense of the data in order to make predictions when faced with new data. This leads to regression and classification problems. Solving such problems generally requires positing a model and thereafter optimising the model. There exist several models: for regression, we have linear models such as ordinary least squares and ridge regression, or we can use feed-forward neural networks (FFNNs), which can be non-linear. For classification problems, there is for instance the logistic regression model, and we can use FFNN models for classification also. The optimisation part of the problem consists in finding the parameters that minimise a cost function, which typically describes how much the model deviates from desiderata such as agreement with the data and model simplicity. Some models have a closed-form solution for their optimal parameters, e.g. OLS and Ridge. But in general the optimal parameters will have to be found by iteration in an optimisation process called gradient descent. This involves calculating the gradient of the cost function with respect to the parameters and using it to nudge the parameters in the direction of a smaller cost function value.

We looked at regression and classification problems by implementing these models and optimisation methods. To this effect we wrote a Python code. Computational efficiency was a guiding principle in the code, using array-based computations. We applied regression to a simple polynomial function in order to demonstrate the basics of each method. Then we performed a classification analysis on a dataset of breast tumors, classifying them as either malignant or benign.

The structure of this report is as follows. In section II we describe the methods and theory used in our analysis, including a brief description of our code. We present and discuss the results in section III, and summarise our work in section IV.

II. METHODS

Models

We will discuss some common models used in regression and classification problems. For regression, there are linear models like ordinary least squares (OLS) and ridge regression, as well as non-linear models such as feed-forward neural networks (FFNNs or NNs). In the case of classification, NNs can be used, as well as the method of logistic regression. We will expound on these models in the following.

Regression analysis

A regression problem consists in using given datapoints (x_i, y_i), where x_i is the independent variable(s) and y_i the response, to find a model that closely approximates the function $f(x)$ from which the data is assumed to come. In our present study, we looked at a simple real-valued polynomial function of one variable, and generated data points from it.

The most basic model in these problems is the linear regression model.[1] It assumes the response variable y follows a function that is linear in some parameters $\beta = [\beta_i] \in \mathbb{R}^p$, plus some random noise ϵ (assumed to be normally distributed with mean 0):

$$y = \sum_k \beta_k x_k + \epsilon = \beta^{\mathrm{T}}\mathbf{x} + \epsilon = f(\mathbf{x};\beta) + \epsilon. \qquad (1)$$

Linear regression approximates each response data point with

$$\tilde{y}_i = \sum_j x_{ij}\beta_j$$

or if we let x_{ij} be the element in the i-th row and j-th column in the design matrix X:

$$\mathbf{y} = X\beta. \qquad (2)$$

[1] This was studied in greater depth in an earlier project.

An optimal regression model is specified by optimal parameters $\hat{\beta}$ which minimise the *cost function*. This is a function of β that is specified by the data points. The desiderata for an optimal model are a notion of "closeness" to the given data points, as well as ideally a low variance in the model parameters. We define a cost function by the mean squared error (MSE) and a penalty term:

$$C(\beta; x_i, y_i) = MSE(\tilde{y}_i, y_i) + \lambda||\beta||^2 \tag{3}$$

$$= \frac{1}{n}\sum_{i=1}^{n}(\tilde{y_i} - y_i)^2 + \lambda\sum_{j=0}^{p-1}\beta_j^2$$

$$= \frac{1}{n}(X\beta - \mathbf{y})^T(X\beta - \mathbf{y}) + \lambda\sum_{j=0}^{p-1}\beta_j^2.$$

Here, λ is a regularisation parameter that drives down the variance in the estimated β. $\lambda = 0$ means no regularisation, and this gives ordinary least squares (OLS) regression; $\lambda > 0$ is called ridge regression. The optimal parameters $\hat{\beta}$ minimise the cost function, which means that the gradient $\nabla_\beta C = \mathbf{0}$.[2] A benefit of ridge regression is that there exists a closed form solution to this equation and therefore for the optimal parameters:

$$0 = \nabla_\beta C = \frac{2}{n}X^T(X\beta - \mathbf{y}) + 2\lambda\beta \tag{4}$$

$$\hat{\beta} = (X^TX + n\lambda I)^{-1}X^Ty. \tag{5}$$

(Here, I is the p by p identity matrix.) The equation can also be solved by iterative methods such as gradient descent, which we will discuss later in the article.

We now look at another regression model, the feed-forward neural network (FFNN), sometimes called the multi-layer perceptron (MLP). It is a non-linear model that is specified by its architecture as well as activation functions used on that architecture.

A FFNN consists of $L+1$ *layers* ($L > 1$), each made up of n_l *nodes* (or *neurons*) ($l = 0, 1, ..., L$) which hold numerical values. There is an input layer $l = 0$, one or more "hidden" layers, and an output layer. Each layer l has a set of *weights* w_{jk}^l which determine how much each node j in the layer interacts with nodes k in the subsequent layer $l+1$, in addition to a set of *biases* b_j for each node. The mechanism of the FFNN is the *feed-forward* algorithm, by which an input is transformed into an output, layer by layer:

input: $x \to$ layer 1: $a^1 \to ... \to$ output at layer L: $a^L = \tilde{y}$.

Here, $a^l \in \mathbb{R}^{n_l}$ is the node values in layer l. (Note that for n inputs $\mathbf{x}_i$ we will correspondingly get n node vectors a_i^l.) We define $a^0 = x$. For $l \geq 1$, a^l is determined by the following computation:

$$z^l := W^{lT}a^{l-1} + b^l \tag{6}$$

An *activation function* $\sigma(z)$ is applied to this intermediate value to produce the actual layer values a^l:

$$a^l := \sigma(z^l). \tag{7}$$

The function is applied to each individual vector element independently. The activation function for each layer may be different. Some common activation functions include the sigmoid:

$$\text{sigmoid(z)} = \sigma(z) = \frac{1}{1+\exp(-z)}, \tag{8}$$

as well as the somewhat similar hyperbolic tangent $\tanh(z)$. Another common function is the rectified linear unit (ReLU):

$$ReLU(z) = \max(0, z). \tag{9}$$

To find the optimal model, we still want to minimise the cost function defined in (3), but calculating the gradients is a more complex task than in the case of linear regression, involving repeated usage of the chain rule of differentiation. The gradient for the final layer, which is directly responsible for the predictions a_i^L ($i = 1, 2, ..., n$), is gotten by:

$$\nabla_{W^L}C = \frac{\partial C}{\partial w_{jk}} = \frac{1}{n}\sum_i a_i^{L-1}\delta_i^{LT} + \lambda W^L, \tag{10}$$

$$\nabla_{b^L}C = \frac{\partial C}{\partial b_j} = \frac{1}{n}\sum_i \delta_i^L + \lambda b^L, \tag{11}$$

where $\delta_i^L \equiv 2\sigma'(z_i^L)(a_i^L - y_i)$. The mean is taken with respect to all the n samples. The gradients for the other layers are gotten through the *backpropagation* algorithm. Going through the layers in reverse order, $l = L-1, L-2, ..., 1$:

$$\delta_i^l = W^{l+1\ T}\delta_i^{l+1}\ \sigma'(z_i^l), \tag{12}$$

and then calculate

$$\nabla_{W^l}C = \frac{1}{n}\sum_i a_i^{l-1}\delta_i^{lT} + \lambda W^l, \tag{13}$$

$$\nabla_{b^l}C = \frac{1}{n}\sum_i \delta_i^l + \lambda b^l. \tag{14}$$

The gradients are then used in an iterative process to minimise the cost function and find the optimal parameters. One problem that may arise when doing so, is

[2] Note that the gradient being zero is necessary but not sufficient for there to be a minimum, unless the function is convex.

that the gradients may become zero, and hence the iterative process essentially comes to a halt without reaching a minimum. This can happen when using the sigmoid or hyperbolic tangent as activation functions: notice that their derivatives $\sigma'(z)$ are nearly zero when z grows in absolute value. Because $\delta^l \propto \sigma'(z^l)\delta^{l+1} \propto \sigma'(z^l)\sigma'(z^{l+1})\sigma'(z^{l+2})...$, the gradients in earlier layers are a product of very small gradients, and so approximately become zero. This is the problem of vanishing gradients. The ReLU function (9) was introduced to remedy this problem: its gradient is non-zero for $z > 0$ and so avoids this problem.

As a result of its potentially complex and non-linear architecture, a FFNN can approximate a function to arbitrary precision. This is known as the universal approximation theorem. We can therefore be confident that a FFNN model can make accurate predictions, but the problem lies in finding an optimal architecture for the network and training it. For more on FFNNs, see [2], [1], [3].

As a last note on regression, in addition to the MSE there is another useful metric of the goodness of the fit, *viz.* the R2 score, defined by

$$R2 = 1 - \frac{\text{MSE}(\tilde{y}, y)}{\text{Var}(y)}, \tag{15}$$

where a value of 1 indicates a perfect fit, i.e. perfect predictions.

Classification analysis

In classification problems we are interested in assigning an input $\mathbf{x}$ into one of two or more classes. We will be concerned with the binary case. This means we want a model that assigns $\mathbf{x}$ to $\tilde{y} = 0, 1$.

A common model is logistic regression, which assigns a probability $p(\mathbf{x}_i)$ to $\mathbf{x}_i$ belonging to class $y_i = 1$ using a sigmoid function:

$$p(\mathbf{x}_i) = P(y_i = 1|\mathbf{x}_i) = \frac{1}{1 + \exp(-\beta^T \mathbf{x})}. \tag{16}$$

The prediction $\tilde{y}_i$ is then the rounded value of $p(\mathbf{x}_i)$. A metric for the model performance is the accuracy score, i.e. the percentage of correct guesses:

$$\text{Accuracy} = \frac{1}{n}\sum_{i=1}^{n} I(\tilde{y}_i = y_i), \tag{17}$$

where $I(\tilde{y}_i = y_i) = 1$ if $\tilde{y}_i = y_i$, 0 otherwise. When optimising our model, there is no closed-form solution and we have to use iterative methods (see below). The cost function in our case now is

$$C(\beta; (\mathbf{x}_i, y_i)) = \sum_{i=1}^{n} (y_i - p(\mathbf{x}_i))^2 + \lambda ||\beta||^2, \tag{18}$$

and this has gradient

$$\nabla_\beta C = \sum_i \mathbf{x}_i (p(\mathbf{x}_i) - y_i) + 2\lambda\beta. \tag{19}$$

As an alternative to logistic regression, we can also use FFNNs as our models, if we modify the cost function:

$$C(\beta; (\mathbf{x}_i, y_i)) = \frac{1}{2}\sum_{i=1}^{n} (y_i - \tilde{y}_i)^2 + \lambda ||\beta||^2 \tag{20}$$

$$= \frac{1}{2}(\tilde{\mathbf{y}} - \mathbf{y})^T (\tilde{\mathbf{y}} - \mathbf{y}) + \lambda \sum_{j=0}^{p-1} \beta_j^2.$$

For the output layer $l = L$, we then use the sigmoid as activation function, giving us a value in the range $[0, 1]$.

Optimisation

We wish to find parameters that give a minimum of the cost function. The method of gradient descent (GD) is an iterative method where the parameters get updated by moving in the opposite direction of the gradient. The gradient is zero at a minimum, so the iteration stops there. GD is inspired by Newton's relaxation method:

$$\beta^{k+1} = \beta k - H^{-1}\mathbf{g}, \tag{21}$$

where $\mathbf{g} = \nabla_\beta C$ is the gradient and $H = (\nabla_\beta \nabla_\beta^T)C$ is the Hessian, or second derivative. The issue with the Newton algorithm is that it requires matrix inversion and matrix multiplication, which are computationally expensive. In gradient descent the Hessian is exchanged for a *learning rate* η, which becomes a hyperparameter that needs to be tuned for optimal results. The iterative scheme is as follows:

$$\beta^{k+1} = \beta^k - \eta\mathbf{g}. \tag{22}$$

Iterations run for as long as needed to reach convergence. A modification of GD is to introduce a "momentum" term in the iteration, designed so that the parameter update takes into account the previous parameter change:

$$\beta^{k+1} = \beta^k - \eta\mathbf{g} + P(\beta^k - \beta^{k-1}). \tag{23}$$

P represents the magnitude the momentum. If the previous step was large, the momentum term effects an additional movement in this direction, giving a "memory" to the gradient descent and improving the convergence rate.

For large datasets, computing the gradient of the cost function for the entire dataset may be computationally expensive. *Stochastic gradient descent* (SGD) remedies this by computing the gradient only for a smaller, randomly chosen batch of the dataset. If the batch size is m, the iteration loops over the entire dataset in n/m

iterations. This process represents one *epoch*, and the iteration runs for a desirable number of epochs until convergence is reached. Although the gradients computed are not as accurate as for regular GD, the smaller computational cost allows for increased iterative speed and hence potentially a faster convergence rate.

Tuning the learning rate

The learning rate η determines the size of the step that is taken in each iteration. We want a step size that hits the balance between not overshooting the desired minimum and not taking too long to reach it. The ideal learning rate may not be constant: the first few iterations will typically have a larger relative importance than later ones, and so the learning rate can be made to decay as the iterations run, eventually leading to a natural stop to the iterative loop. We will now briefly discuss scheduling of the learning rate. For more in-depth discussion, see [2].

There may be directions in parameter space that are slower to converge. This suggests having a learning rate that is specific to each parameter. One method that achieves this is the adaptive gradient method (Adagrad). It updates the learning rate as

$$\eta_k = \frac{\eta_0}{\delta + \sqrt{\text{diag}(r_k)}}. \tag{24}$$

Here η_0 is the initial learning rate and r_k is a matrix defined by a sum of the squared gradients $\mathbf{g}_i$ at each step i:

$$r_k = \sum_{i=1}^{k} \mathbf{g}_i \mathbf{g}_i^T.$$

For computational efficiency only the diagonal of r_k is used. The parameter $\delta \sim 10^{-7}$ is used to avoid division by zero. Note that η_k is a vector. It multiplies the gradient element-wise, giving each parameter its own learning rate.

Another common learning rate schedule is RMS propagation. The learning rate is given by the same schedule as for Adagrad (24), but with a different definition of the matrix r_k:

$$r_k = \rho r_{k-1} + (1-\rho)\mathbf{g}_k \mathbf{g}_k^T,$$

where $r_0 = 0$ and $\rho \in [0, 1)$ is a hyperparameter that controls how much the learning rate "memorises" earlier rates. Unlike Adagrad that keeps a history of all earlier gradients, RMS propagation prioritises the most recent ones, which can improve convergence for non-convex problems.

A third and very popular learning rate schedule is Adaptive moments (Adam). It requires two hyperparameters ρ_1 and ρ_2. It defines the matrix r_k similar to RMS prop:

$$r_k = \rho_2 r_{k-1} + (1-\rho_2)\mathbf{g}_k \mathbf{g}_k^T,$$

but it also accommodates a vector proportional to the first moment of the gradients:

$$s_k = \rho_1 s_{k-1} + (1-\rho_1)\mathbf{g}_k.$$

These quantities are then scaled by how far the iteration has run:

$$\hat{s}_k = \frac{s_k}{1-\rho_1^k},$$

$$\hat{r}_k = \frac{r_k}{1-\rho_2^k},$$

and then the parameter update is given by

$$\beta^{k+1} = \beta^k - \eta_0 \frac{\hat{s}_k}{\delta + \sqrt{\text{diag}(\hat{r}_k)}}. \tag{25}$$

Code

Our work was based on a Python code centered around two original classes: a GradientDescent class that performs various flavours of gradient descent, and a NeuralNetwork class that defines a FFNN with desired architecture and activation functions. Computational efficiency and algorithmic optimisation are crucial when performing gradient descent with a lot of parameters and data points. The code was written to leverage the array-based computational capabilities of the Numpy library, resulting in faster performance.

III. RESULTS AND DISCUSSION

Regression analysis

We generated $n = 1000$ data points (x_i, y_i) by using a simple polynomial function. The simplicity of the function is intentional, in order to highlight basic aspects of the different models and optimisation methods. We sampled random x_is from a normal distribution with mean 0 and variance 0.8^2, and the response variable was calculated by

$$y_i = 2 + 0 \cdot x_i + 2 \cdot x_i^2 + 2 \cdot x_i^3 + \epsilon, \tag{26}$$

where ϵ is drawn from the standard normal distribution. Because the x_is were around order unity, there was no need to scale the data. We split the data into training data and test data. The models were trained on the training data, and performance metrics were evaluated on the test data.

An initial part of our research was to discover how the convergence of stochastic gradient descent varies with the number of epochs and size of batches m. To see this, we calculated the test MSE for the OLS model using RMS propagation as the learning rate schedule. The results

are in fig. 1. As expected, the convergence is better after more epochs, but we get diminishing returns as we increase the number beyond a certain value, here apparently around 50-60 epochs. Reducing the batch size improves the convergence as well, because this has the effect of increasing the number of iterations per epoch. For the rest of our research we used 50 epochs and $m = 10$ as standard.

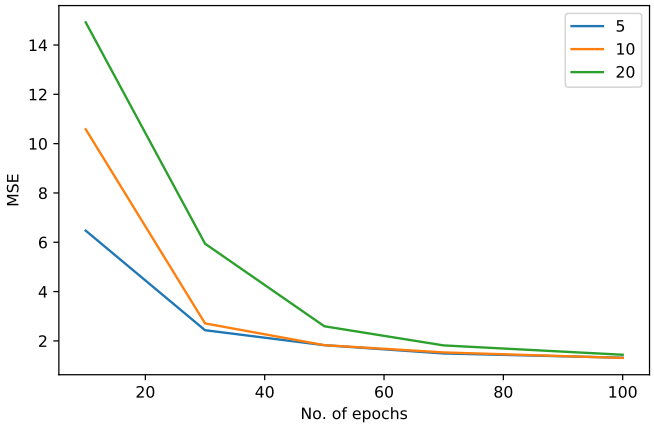

FIG. 1. Test MSE as function of number of epochs and batch size $m = 5, 10, 20$.

We then investigated the linear regression model by using the analytic solution for the optimal parameters, as well as various gradient descent optimisation methods, among them:

- regular GD
- GD with momentum (0.8 gave good results across the board)
- SGD with constant learning rate
- SGD with Adagrad with and without momentum (0.8)
- SGD with RMS prop
- SGD with Adam.

Each setup was tested with a regularisation parameter $\lambda \in [0, 10^2]$ (logarithmically evenly spaced). A heatmap of the resulting $R2$ scores can be seen in fig. 3, with regularisation parameter along the vertical and optimisation method along the horizontal axis. The data along with a few selected fits are shown in fig. 2. We used an initial learning rate $\eta_0 = 0.001$ in all cases except with Adagrad, which had bad convergence unless the learning rate was increased. We increased it by a factor of 50. This demonstrates a general phenomenon that hyperparameters may not be directly transferable between different methods.

As can be seen from fig. 3, the fit is very good for all setups, except for the strongest regularisation at $\lambda = 100$. An exaggerated regularisation having a negative impact on the fit is no surprise, because it forces the parameters to zero, and hence our model becomes useless. We can see some smaller variation in the $R2$ scores for the same model but with different optimisation methods. In particular, Adagrad seems to have a slightly lower score on all models. This could be a result of a suboptimal initial learning rate, even though we had already adjusted it to be 50 times larger for Adagrad than for the other methods. However, we see that adding momentum ensures a better convergence. And from visual inspection of the predictions in fig. 2 we see that the fits all lie very close together, except with some deviation at the extremes of the input space.

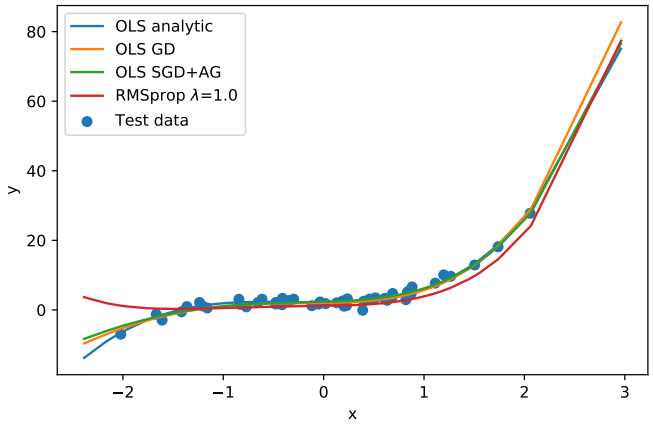

FIG. 2. This figure shows some of the data points along with some of the prediction curves.

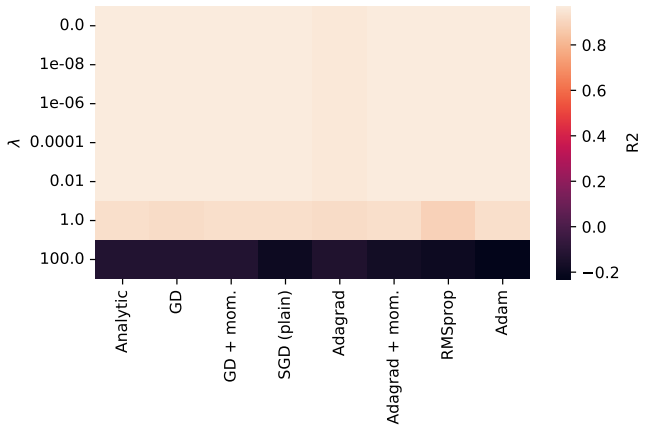

FIG. 3. R2 scores as a function of λ for various optimisation methods.

The FFNN is a more complex model than the linear regression models, and there are several hyperparameters that need to be defined: its architecture and activation functions, the optimisation method and the regularisation parameter λ. This represents a vast space of possible model configurations. To investigate the effects of these

parameters on the performance of our model, we defined the following setups:

- Number of hidden layers:

  2, 1

- Number of nodes per layer:

  (3,2), (5,3), (5), (3)

- Activation function used (for simplicity on every layer):

  sigmoid, ReLU, tanh

- Optimisation:

  SGD with momentum 0.8, with Adagrad (learning rate $\eta_0 = 0.1$), RMS prop, Adam (both learning rate $\eta_0 = 0.001$)

- Regularisation parameter:

  $\lambda = 0, 10^{-8}, 10^{-6}, ..., 10^{2}$.

The $R2$ score as a function of λ is shown for the various configurations in fig. 4. The rows correspond to the architecture (no. of hidden layers and nodes per layer): (3,2), (5,3), (5) and (3). The columns are dedicated to the learning rate schedule. The three curves in each figure correspond to the sigmoid (0), ReLU (1) and tanh (2) activation functions. Because of the random nature of the SGD algorithm, the model weights and biases may not be identical for identical model setups, and for some setups we experienced that the optimisation method struggled to converge to parameters that didn't predict all zeros on the test set. For this reason we computed the average $R2$ score of three runs for every setup, in order to reduce the effect of bad runs.

From the plots we see that there is only one activation function that gives good results, namely the ReLU. This is likely due to the fact that the sigmoid and tanh functions give vanishing gradients, and so the network training stops before good parameters have been reached. Another takeaway from the plots is that stronger regularisation λ doesn't seem to give better results. This suggests that our models have not bloated the number of weights and biases to an excessive degree, making the regularisation moot. When it comes to the learning rate schedule, the results seem to be similar for all three, though it may be noted that Adam scores very badly for very small regularisation λ. This is likely only a result of bad convergence of the method in some cases, drawing down the average $R2$ score. Furthermore, we see that the architecture of the network seems to have negligible effect on the goodness of the fit. This may be a disappointing result, but it suggests that the simple polynomial data used is too simple for the architecture of the FFNN to play a significant role.

We may therefore conclude that in the present case, regular linear regression seems to perform best, as it gives higher $R2$ scores and is more parsimonious. This should come as no surprise, as our data are a linear function of the parameters and not complex enough for neural networks to offer any advantage.

Classification analysis

We performed a binary classification using both logistic regression and FFNNs on the well-known Wisconsin breast cancer dataset. This consists of $n = 569$ samples $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i$ contains 30 statistics gotten from medical imaging of a tumour, concerning its texture and shape. y_i is 0 if the tumour is malignant and 1 if benign. Because the tumour features are of heterogeneous size, the data had to be scaled before performing the analysis.

In the case of logistic regression, we used SGD with momentum = 0.8 and RMSprop, with a learning rate $\eta_0 = 0.001$. We calculated the accuracy of our predictions for 10 iterations and took the average. In addition to this we used the `scikit-learn` LogisticRegression class to compare its accuracy with that of our own code. The results are plotted as a function of regularisation $\lambda = 10^{-6}, 10^{-4}, ..., 10^{4}$ in fig. 5. Before commenting on the results, we remark some differences between our code and the `scikit-learn` regression class: the latter takes an inverse regularisation strength `C` as a parameter,[3] and in our comparison we therefore defined $C = 1/\lambda$. `scikit-learn` also uses a different optimisation method from our SGD with RMS propagation. In any case, the predictions are good in both cases: the accuracy is close to 100% for most regularisation values. We note that the accuracy for our own code falls more drastically for stronger regularisation. This suggests that the `scikit-learn` implementation of regularisation is somewhat different from ours.

[3] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

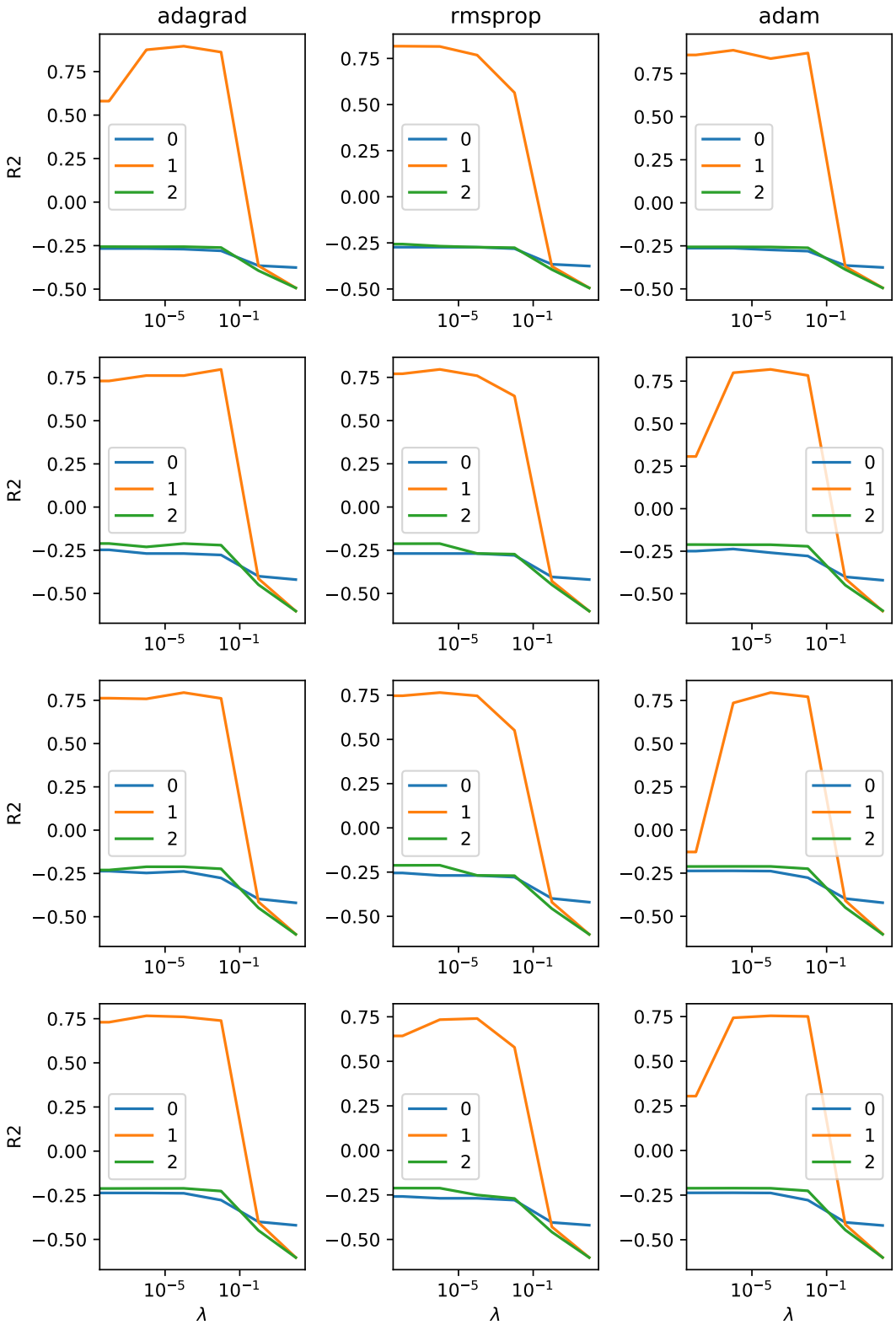

FIG. 4. These plots show the $R2$ score as a function of λ for various configurations of the network. The rows, going from top to bottom, correspond to a different number of nodes per layer: (3,2), (5,3), (5) and (3). The columns correspond to the schedule for the learning rate. The three curves in each figure correspond to the sigmoid (0), ReLU (1) and tanh (2) activation functions.

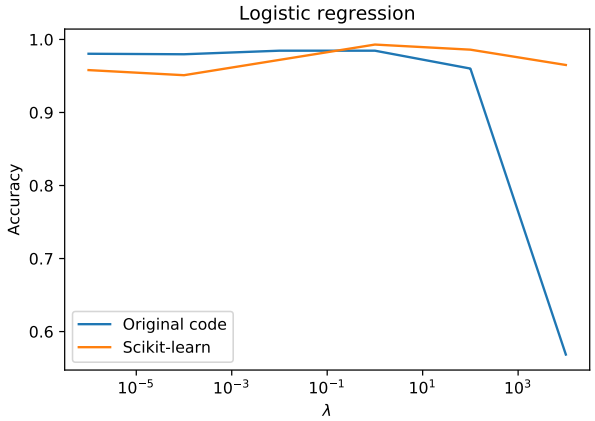

FIG. 5. This shows the accuracy of the predictions on the test data as a function of regularisation strength λ.

We then used the FFNN model to classify the data. Here we used the same setups for the network as when performing the classification, except we now used a sigmoid on the output layer and only varied the activation function on the hidden layer(s), and the architectures were in this case specified by (15, 8), (10,5), (8), (5) nodes per layer. The accuracy score for all setups is shown in fig. 6 as a function of regularisation. We see that for small regularisation the accuracy is close to 100%. Interestingly, unlike for the regression case, now all three activation functions give similar results. The sigmoid and hyperbolic tangent are therefore more appropriate to be used as activation functions when performing classification. Furthermore, we note again that the architecture of the network is negligible. This indicates that the statistical correlation between the tumour features and the tumour malignancy is not very complex and can be modelled using even just one hidden layer with a handful of nodes. Finally, we note that the predictions are worse than random guessing when using RMS prop and Adam with higher regularisation. Possibly the optimisation parameters could be tuned to give better results. In any case, strong regularisation only worsens the score and for small regularisation RMS prop and Adam give decent results.

But even though the FFNN model can give good results, there are reasons to prefer using a logistic regression model in this case: the parameters in the logistic regression model have a straightforward interpretation, namely the relative importance of the feature they multiply. So in our study we found for instance that the area of the tumour is predictive of cancer risk, whereas its compactness was of smaller importance. In contrast, the parameters in the FFNN are weights and biases that are less immediately informative. The significance of this observation will depend on the complexity of the problem. The intricate nature of the neural network may have the upper hand in other types of classification problems where the input data isn't so neatly divided into features, e.g. when classifying images. In these cases we may be satisfied with getting a good prediction accuracy and remaining agnostic about the actual values of the model parameters.

IV. CONCLUSION

We investigated a regression problem of a simple polynomial function by applying linear regression and a feed-forward neural network model. The model parameters were found by gradient descent methods, and in the case of linear regression also by an analytic expression. The linear regression model performed best, as it gave the highest $R2$ scores. When using a FFNN to perform the regression, the only activation function that gave a decent result was a ReLU function, with the sigmoid and tanh suffering from the problem of vanishing gradients. As regards the optimisation methods, they performed well and gave good model results for a range of setups, but in some cases when training the FFNN they didn't converge to a good solution.

The problem of classifying tumours based on the Wisconsin breast cancer dataset was solved by applying logistic regression and a FFNN model. Both models were able to give a prediction accuracy of well over 90%. Because the parameters in the logistic regression model are more easily interpreted, we recommend using logistic regression in this case. In other problems, e.g. image classification, a neural network model may be preferable.

There are clearly many more investigations that could be made. For one thing, performing regression on a more complex function than a polynomial, maybe even a discontinous function. This would invalidate the linear regression model but reveal the strengths of the FFNN model. Furthermore, one could further explore classification with neural networks by applying it to e.g. image input data. One could also look at multinomial classification, i.e. classifying an input into one of more than two classes, where one would use a softmax function instead of a sigmoid.

[1] Christopher M. Bishop (2006). *Pattern Recognition & Machine Learning*, Springer.
[2] Ian Goodfellow, Yoshua Bengio & Aaron Courville (2015). *Deep Learning*, MIT Press.
[3] Trevor Hastie, Robert Tibshirani & Jerome Friedman (2009). *The Elements of Statistical Learning*, 2.ed, Springer.

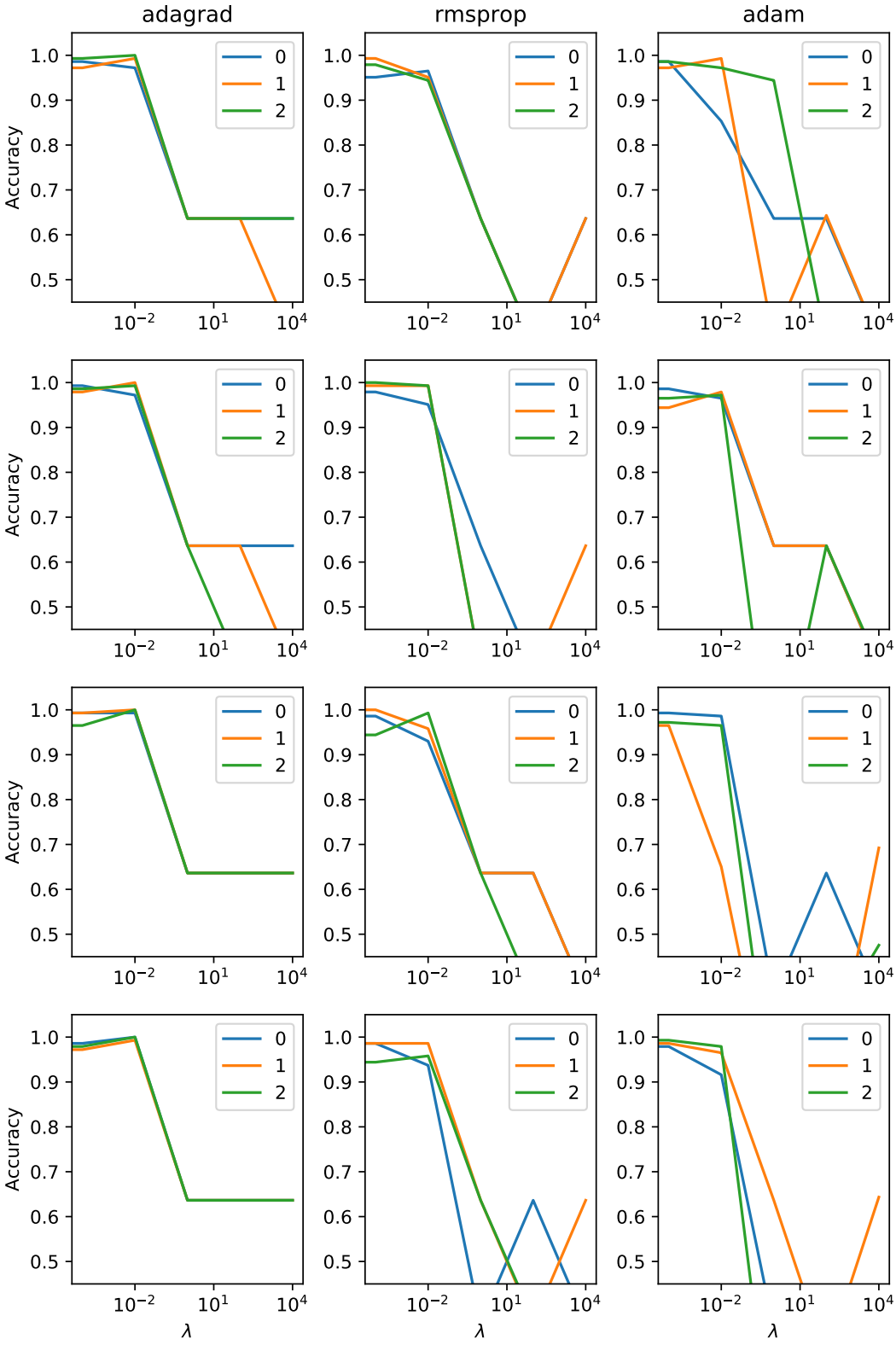

FIG. 6. These plots show the accuracy score as a function of λ for various configurations of the network. The rows, going from top to bottom, correspond to a different number of nodes per layer: (15,8), (10,5), (8) and (5). The columns correspond to the schedule for the learning rate. The three curves in each figure correspond to the sigmoid (0), ReLU (1) and tanh (2) activation functions being applied to the hidden layer(s).