Project 2: Neural networks and logistic regression

Noel Gomariz Kühne

November 2021

Abstract

In this document we will investigate how two of the main methods described in the machine learning literature, neural networks and logistic regression, perform on two different problems: the regression problem and classification analysis. We will also discuss why each method applies best to a specific problem and compare their performance against the linear regression.

1 Introduction

The development of machine learning algorithms has led to a huge area of application in fields such as natural sciences, statistics or economy, and recently more and more problems can be approached by these methods. In an attempt to better understand how they work and gain some insights about their performance, we will study and compare three of the main methods, namely linear regression, logistic regression and neural networks, as well as the heart of machine learning: the gradient descent methods.

First, we will dig into the regression problem and compare which method works best: linear regression or neural networks. Then will be the turn of the classification problem: we will study the performance of neural networks and logistic regression over a binary classification problem, the Wisconsin Breast Cancer data set.

In Section 2 we describe the necessary algorithms and methods, and explain the structure of the different codes needed through the project. Later in Section 3 we show the results we obtain for the various methods and a little discussion about why we are getting these outputs.

2 Methods and algorithms

2.1 Linear regression

One of the most popular techniques within machine learning is linear regression, proved to be the right choice when it comes to fitting a continuous function. Even though there are analytical expressions for the parameters of the ordinary least squares and ridge regressions, we want to test how the gradient descent methods affect to their performance.

More specifically, we are going to implement stochastic gradient descent with mini-batches to optimize the parameters. The cost function of the standard linear regression is

$$C(\beta) = \frac{2}{n}(y_i - X_i\beta)^2 \tag{1}$$

As we want to optimize the parameters to minimize the cost function, we search for a vector $\hat{\boldsymbol{\beta}}$ such that $\nabla_\beta C(\hat{\boldsymbol{\beta}}) = 0$.

$$\nabla_\beta C(\beta) = \frac{1}{n}X^T(X\beta - y) \tag{2}$$

Thus, our strategy is to repeatedly compute the gradient of the cost function and actualize the parameters via

$$\beta \leftarrow \beta - \eta\nabla_\beta C \tag{3}$$

where η is the learning rate.

To accelerate the calculation of gradients and improve the performance of the optimization we introduce stochastic gradient descent[1] (SGD): in each iteration we choose a random selection from the training data to compute the gradient. Implementing this random choice instead of computing the gradient with the whole data set benefits us in two ways: first, the calculation will be much faster since we use only a few data points at each iteration; second, choosing random samples from the training set reduces the probability of getting stuck in a local minimum of the cost function.

If we are dealing with ridge regression instead, the only change needed is to add the regularization term to the cost function and the gradient.

$$\nabla_\beta C_{ridge}(\beta) = \frac{1}{n}(X^T(X\beta - y) + \lambda\beta) \tag{4}$$

In relation to the learning rate, sometimes it is useful to update it through each iteration. This is because the randomness of the SGD optimization makes the parameters oscillate around their optimal value, and by decreasing the learning rate at each step we reduce that undesired oscillation.

2.2 Neural Networks

Neural networks are designed to resemble the functioning of the human brain: through many layers of neurons connected one to each other, it is possible to transform an input, such as an

[1]Aurélien Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, O'Reilly Media, United States of America, 2017, chapter 4.

image or a sound, to a deeper concept. That's why neural networks are so widely used within the field of classification problems. But before a network can actually perform a classification task, it needs to go through a learning process, and the algorithm behind this process is *backpropagation*.

To set up a neural network, we need to define an architecture for it, meaning how many hidden layers it will present and how many nodes each layer will hold, besides the input layer, which is fed with the input data, and the output layer, giving us the outcome of the input. Let L be the number of hidden layers and n_i, $i = 0, 1, ..., L-1$ the number of nodes contained in each layer. n_L will be the number of categories (number of nodes in the output layer), $n_{samples}$ and $n_{features}$ the numbers of input and features of the data.

The next step is to create the weights W^i connecting each layer and the bias b^i for each neuron. In the previous notation, we will have $L+1$ vectors for the biases and $L+1$ matrices for the weights (L for each hidden layer and 1 for the output layer). The dimensions of the weights matrices will be $(n_{features} \times n_0)$ for the connection between the input layer and the first hidden layer, and $(n_{i-1} \times n_i)$, $i = 1, ..., L$ for the connections between hidden layers and the last hidden layer with the output.

Once all the parameters of the network are defined, we can start training the network. The first step is a *feed-forward* of the data. A design matrix is introduced to the input layer and connected with the following layer through the relation

$$a^i = f(a^{i-1}W^l + b^i) = \sigma(z^i) \tag{5}$$

Here, a^i is the input of each hidden layer and we have defined $z^i = a^{i-1}W^i + b^i$. f is the activation function of the hidden layers, a function mapping z^i to the interval $(0, 1)$ (usually the *sigmoid* function). When the feed-forward process reaches the output layer, we want to get a vector (or rather a matrix of dimension $(n_{samples} \times n_L)$) containing the probabilities of the output belonging to one of the categories. For that purpose, we need an output activation that normalizes $z^L = a^{L-1}W^L + b^L$. A common choice is the *softmax* function.

When the feed-forward is done, we end up with a vector of probabilities for each sample, which of course we would like to be as close to one as possible in the right category and nearly zero in the others. To achieve this we again make use of gradient descent methods. The equations describing the gradients of the cost function[2] with respect to the weights and biases of each layer are known as the *backpropagation algorithm*[3]:

$$\delta^L = (a^L - y) \odot \sigma'(z^L) \tag{6}$$

$$\delta^i = \delta^{i+1}(W^{i+1})^T \odot \sigma'(z^i) \tag{7}$$

$$\frac{\partial C}{\partial b^i} = \delta^i \tag{8}$$

$$\frac{\partial C}{\partial W^i} = a^{i-1}\delta^i \tag{9}$$

[2]For simplicity, we employ the mean squared error as cost function and the sigmoid function as the activation for the hidden and output layers.

[3]Michael A. Nielsen, *Neural Networks and Deep Learning*, Determination Press, 2015, chapter 2.

When we implement equations (6-9) in our SGD algorithm, the expression for the gradient with respect to the biases will be summed over the samples of the mini-batch. This method is implemented in the file "neural.py" just as described in this section. A new object of "Neural-Network" takes as arguments the training data, the architecture of the class (number of layers, nodes and categories), the activation functions of the hidden and output layers and the parameters related to SGD optimization, like the size of the mini-batch or the number of epochs. The method ".train()" performs the SGD optimization of the weights and biases.

2.3 Logistic Regression

When it comes to the problem of binary classification, another widely used method described in the machine learning literature is the logistic regression. Each data sample is given a probability of belonging to one of two categories by the operation

$$p(x_i|\beta) = \sigma(x_i\beta) \tag{10}$$

Here, β are the parameters of the logistic regression. Again, we want to optimize this parameters to minimize the cost function

$$C(\beta) = -\sum_{i=0}^{n-1} y_i \log p_i + (1 - y_i)\log(1 - p_i) \tag{11}$$

When we compute the gradient of the cost function with respect to the parameters β, we obtain the expression

$$\nabla_\beta C = -X^T(y - p) \tag{12}$$

where now X is the design matrix of $(n_{samples} \times n_{features})$ and y and p are vectors of dimension $n_{samples}$. Now this expression is ready to be implemented into our SGD algorithm.

The file "logistic.py" contains an object oriented implementation of a logistic regression system which parameters are the training data, the activation function and the parameters related to the gradient descent process.

3 Results

3.1 Simple test on SGD

First we are going to test the performance of stochastic gradient descent on the ordinary least squares regression of the Franke function. We want to find out what combination of epochs and batch size produces a better prediction on the test data. The learning rate will be updated through the following learning schedule:

$$\eta(t) = \frac{t_0}{t + t_1} \tag{13}$$

where t_0 and t_1 are fixed parameters, and t stands for the current iteration. Figure 1 shows the mean squared error of the test data obtained for the number of epochs running from $n_{epochs} = 30, 35, ...65$ and the size of the mini-batch from $M = 2, 3, ..., 7$.

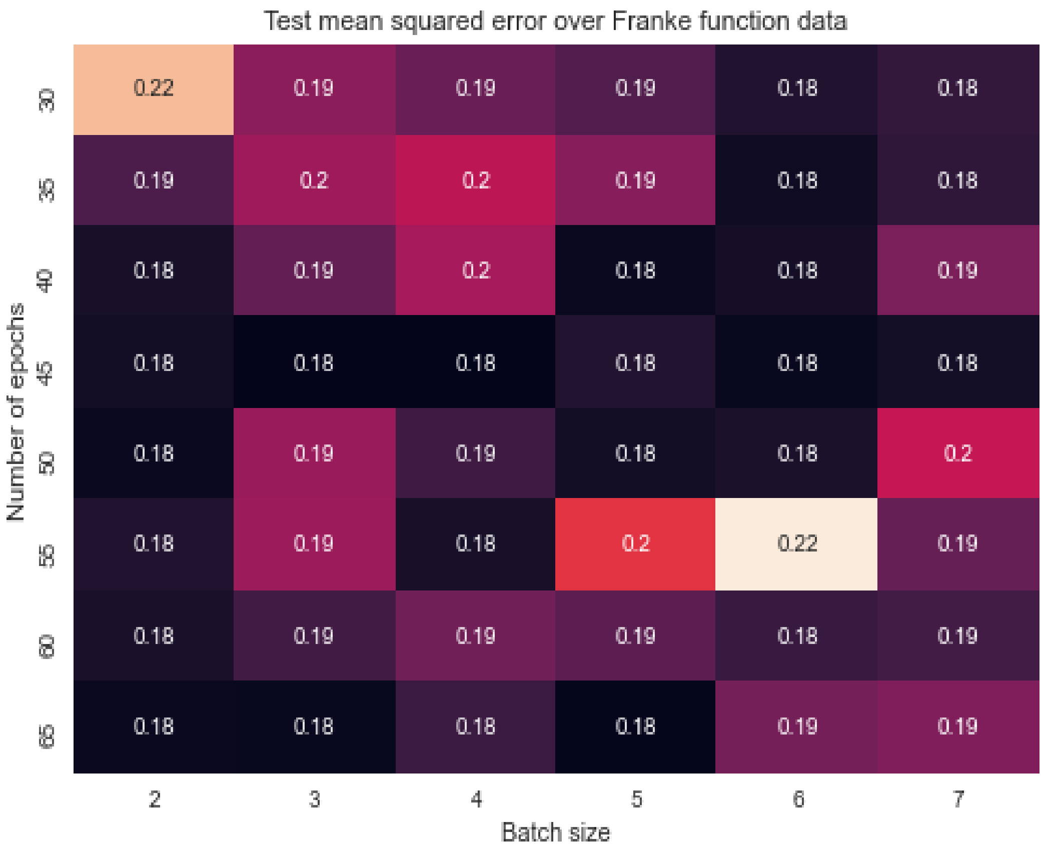

Figure 1: Test mean squared error of the Franke function regression.

We see that the mean squared error does not vary too much from one set of parameters to another, thus we can conclude that the number of epochs and the batch size are not critical parameters of the algorithm. Also, the analytical algorithm of the standard linear regression on the same data set gave us a mean squared error around $MSE_{analytical} \approx 0.179$, which indicates that SGD actually performs quite well in the ordinary least squares problem.

To further check the accuracy of stochastic gradient descent, we also tested it on the ridge regression. This time, the grid search for the optimal parameters will be done in terms of the regularization parameter λ and the learning rate η, which this time will not be updated at every iteration, but constant through each calculation. Now, as we see in Figure 2, the mean squared error is not stable anymore for each set of parameters, but it changes rapidly as a function of the learning rate and the parameter λ. As we should expect, when $\lambda \to 0$ the mean square error resembles the one from the ordinary least squares.

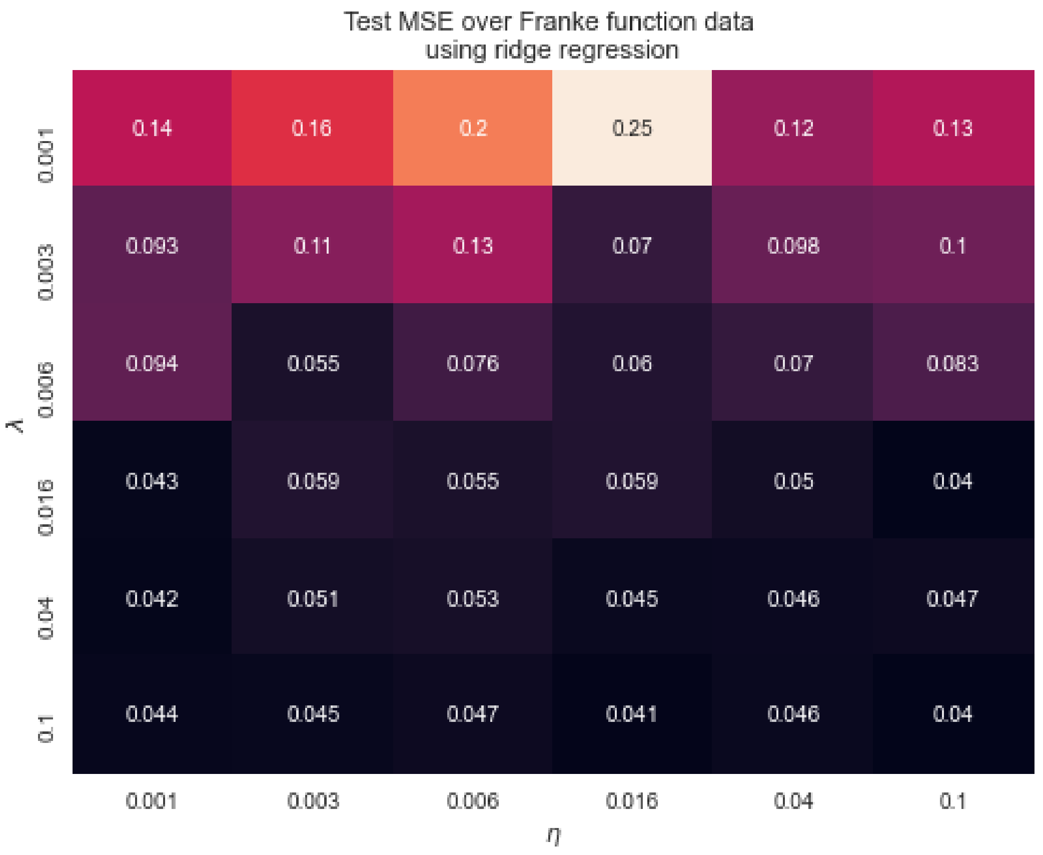

Figure 2: Test mean squared error of the ridge regression on Franke function.

3.2 Testing neural networks on regression and classification tasks

3.2.1 Fitting a continuous function

In this section we show the results of training our neural network to fit artificial data produced from the Franke function. But first, a rapid test of how well it does on a simpler case: fitting a straight line with random noise. About the architecture of the network, since we are dealing with a simple problem we wouldn't want to expend too much calculation resources to solve it, and thus the configuration of the neural network is just one hidden layer with 10 nodes, activated by the sigmoid function, and the output node, which is not modulated at all since we want to predict values from a continuous function. We trained it in 200 epochs with a batch size of 50 samples, and for four different sets of (η, λ). It is visible in Figure 4 that, for higher values of the learning rate, the model does not learn at all. Thus, the training needs to be done in small steps, which means it also needs more epochs to achieve optimal performance. This behaviour indicates that the neural network struggles to adapt to linearity, and this is exactly what we would expect since the internal structure of neural networks is not linear, but modulated by the activation function.

Now we move on to perform a similar test but this time on the Franke function with noise. This time is not simple to visualize in a plot how well the neural networks catches the behaviour of the function, so instead we are going to trust the mean squared error and R^2 scores. Our hope is to resemble as much as we can the results from applying ridge regression to the same data set. Table 1 shows the comparison between the analytical ridge regression (fitting a polynomial of third degree) for each regularization and the neural network.

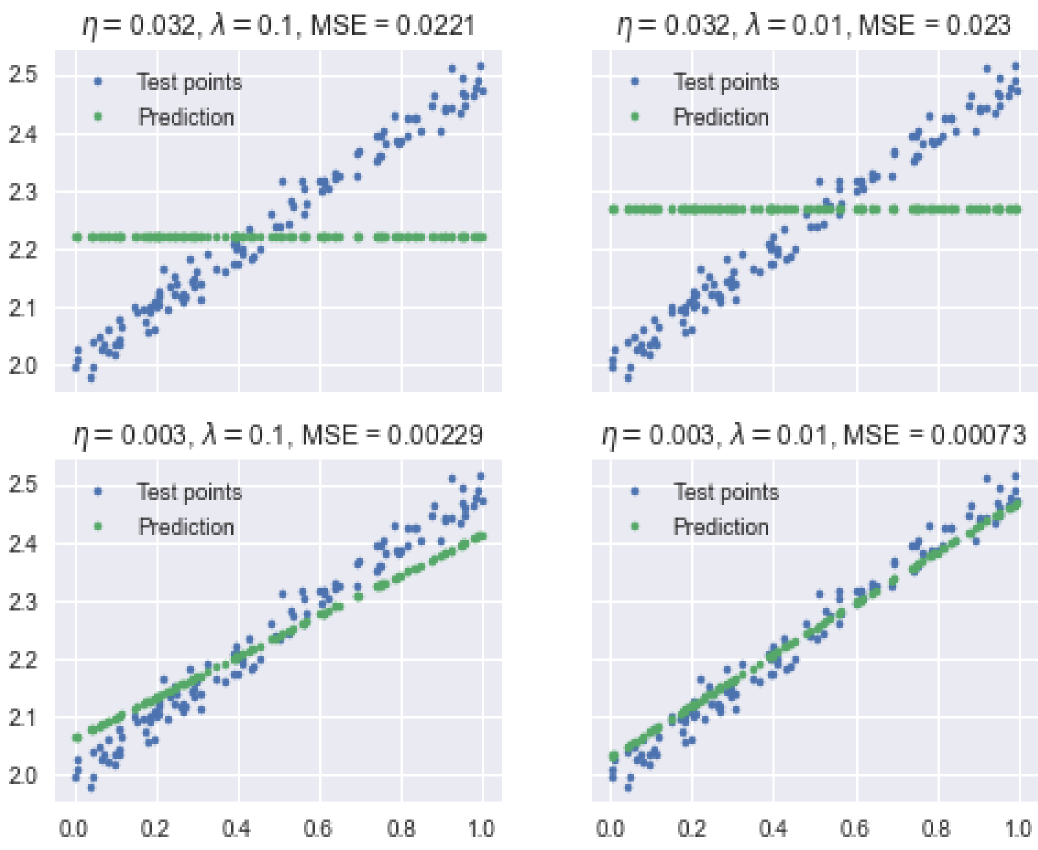

Figure 3: Results of fitting a first degree polynomial with a neural network.

η	λ	MSE of NN	R^2 of NN	MSE of ridge	R^2 of ridge
0.003 16	0.01	0.273	−0.007	0.112	0.586
0.003 16	0.0316	0.274	−0.010	0.114	0.578
0.003 16	0.1	0.280	−0.033	0.121	0.554
0.01	0.01	0.142	0.475	0.112	0.586
0.01	0.0316	0.143	0.471	0.114	0.578
0.01	0.1	0.152	0.437	0.121	0.554
0.0316	0.01	0.276	−0.018	0.112	0.586
0.0316	0.0316	0.177	0.345	0.114	0.578
0.0316	0.1	0.286	−0.054	0.121	0.554

Table 1: Comparison between ridge regression and neural network on the Franke function.

This time, since we included non-linearity and one more dimension, the problem is a bit more demanding and therefore we upgraded the structure of the network to two hidden layers of ten neurons each and the single node output layer, and it was trained in 250 epochs and mini-batches of 50 samples for each set of learning parameters. We again used the sigmoid as activation function for the hidden layers.

As we can see, the best results are obtained for $\eta = 0.01$, being λ less determinant in the calculation. Both the mean squared error and R^2 are quite similar to the ones obtained by applying ridge regression, so we can say that our neural network is capable of fitting continuous functions as well.

Now that we know that our neural network can deal with regression tasks, we can move on to test how different activation functions change the performance of the model. Between other less famous functions, we find the ReLU, Leaky ReLU and the tanh. The procedure was to train our network for each one of them and different values of λ, and compare the mean squared error that we obtain (see Figure 4). The learning rate employed for all runs was $\eta = 0.01$, since it gave the best results for the test on the sigmoid. The number of epochs and batch size are also as in the previous test.

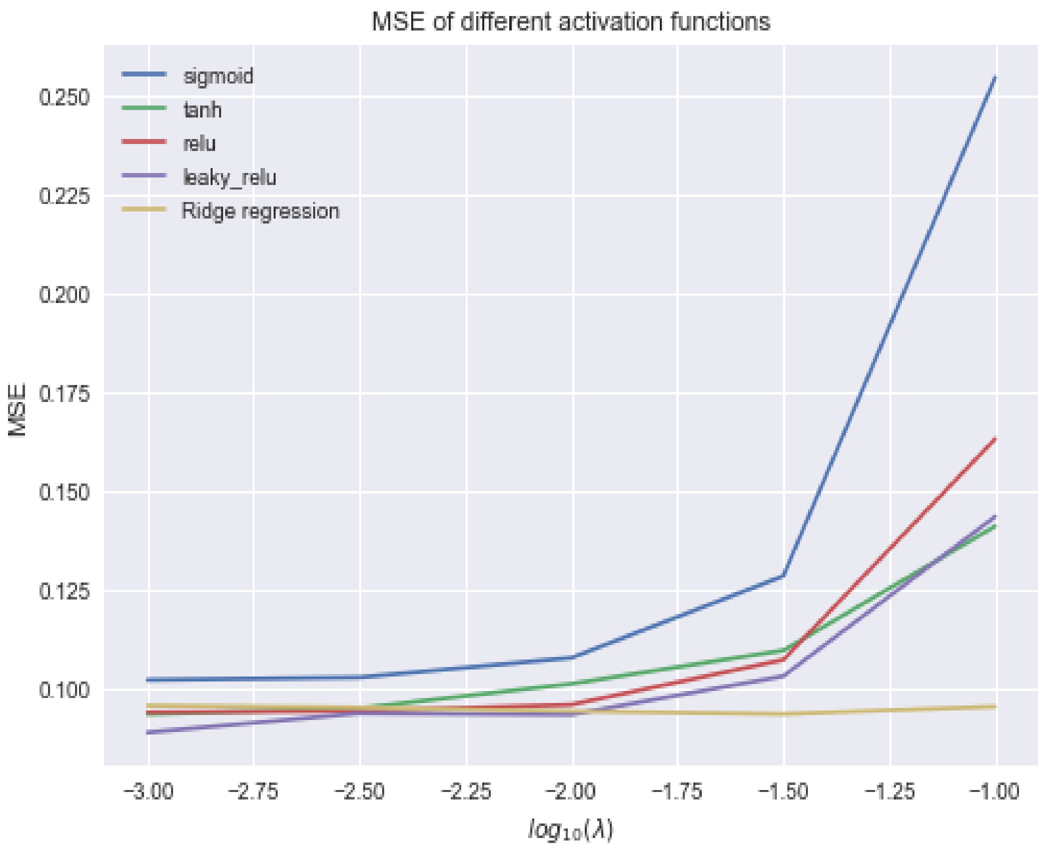

Figure 4: Test MSE of different activation functions

From the graph we can conclude two relevant features about the network: in contrast with the analytical regression, neural networks perform better in regression for smaller values of the regularization parameter. Nevertheless, when increasing λ, the tanh, ReLU and leaky ReLU are a bit more stable than the sigmoid.

Summarizing this section, neural networks are impressive due to their flexibility when it comes to adapt to different kind of problems. Even though regression is not what a neural network is meant for, we were able to fit a relatively complex function with only two hidden layers of complexity. However, as long as we dispose of linear regression, it should be the go-to option for fitting continuous functions, since neural networks are much more expensive to train in order to obtain good results.

3.2.2 Classification analysis

The classification problem is the one for which neural networks are the way to go. Unlike linear regression, the flexibility in the number of nodes in the final layer makes it possible to approach many different problems. In this section we are mainly going to study binary and multiple classifications.

The first test is to apply our neural network to a well studied data set: the Wisconsin Breast Cancer data set, which consists of 570 samples with 30 features each. For this problem, we are going to study the accuracy score $A = \frac{1}{n}\sum_{i=1}^{n} I(y_i = t_i)$ in terms of the structure of the network and the hyper-parameters η and λ, as well as the different activation functions.

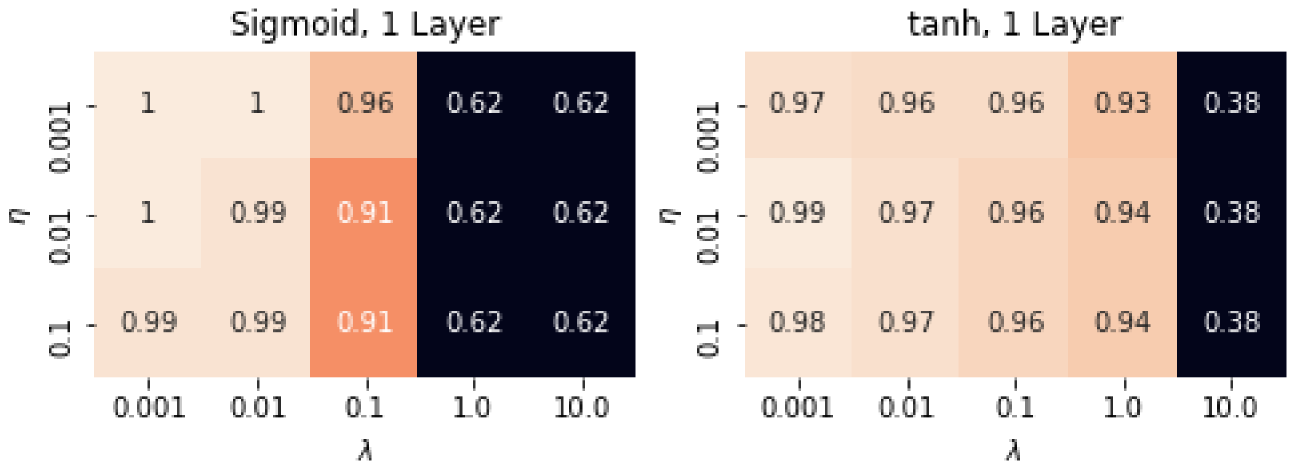

Figure 5: Accuracy score over Wisconsin test data.

The main difference we observe between using sigmoid and tanh as activation functions is their behaviour as we vary the hyper-parameters. Both functions return good enough accuracy scores, but tanh is stable for higher values of the regularization, and behaves better for larger values of the learning rate, which could be interesting if we want to accelerate the training by increasing η and lowering the number of epochs.

Since we had such satisfactory results on binary classification, it is time to head on to a more complex task: multiple classification. The classic, widely studied data set for this problem is the MNIST data. We will basically repeat the same analysis as we have done with the previous data set. Nevertheless, this time we will add a second hidden layer to the neural network and increase the number of hidden neurons, since the number of features is quite high (28×28 pixels each sample), and we expect the training to be more demanding. For the same reason, we will train the network in only 30 epochs, due to the long calculation time required. The overall characteristics of the network are:

Hidden Layer 1: 16 neurons
Hidden Layer 2: 16 neurons
Output Layer: 10 categories
Number of epochs: 30
Size of the mini-batch: 100 samples

As shown in Figure 6, even for a low number of epochs, the results are pretty acceptable. Nevertheless, as one could say beforehand, we need a fine learning rate in order to obtain an accuracy as close to 1 as possible. For both ReLU and tanh, values of the regularization parameter around $\lambda \approx 0.01$ are the best choice.

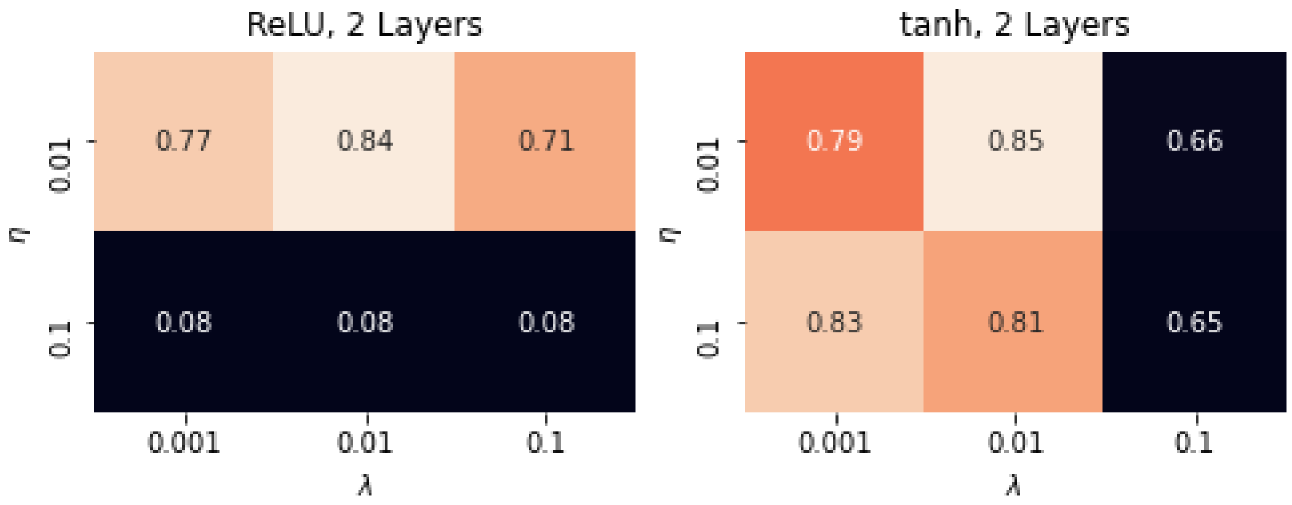

Figure 6: Accuracy score over MNIST test data.

So far, we have been able to implement the idea behind neural networks and test it against regression, binary and multiple classification. The results have been satisfactory, even though we lost accuracy in exchange for computation time. But having gained a bit more insight about the best parameters for this problem, we can overcome our previous results and try to train the network until we reach the highest possible accuracy over the test set. For example, a network with the same architecture as the previous one but trained in 70 epochs, with a batch size of 60 samples, learning rate equal to 0.01 and penalty parameter $\lambda = 0.03$ is capable of predicting over the test set with a 90% of accuracy.

3.3 Logistic regression for binary classification

The structure of logistic regression makes it way faster to train than a feed forward neural network, since it is characterised only by a number of parameters equal to the number of features of the train data. Thus, if its performance on binary classification is acceptable, we would rather use logistic regression than other more expensive methods.

We tested our own designed logistic algorithm on the Wisconsin Breast Cancer data set, and the results were quite satisfactory. We didn't even have to perform a grid search in η and λ to obtain an accuracy score of about 95%. When compared with the logistic regression model that the module *sklearn* provides, and using the same regularization parameter $\lambda = 0.1$, we obtained almost the same accuracy for both models.

Accuracy score of our own designed logistic regression for $\lambda = 0.1$, $\eta = 0.01$: 0.974

Accuracy score of logistic regression from sklearn with $\lambda = 0.1$: 0.965

A more careful search in the parameters η and λ showed that the training of logistic regression is not very sensible to changes in them. Neither the size of the mini-batch employed for SGD was a determinant parameter for the result of the accuracy test. In Figure 7 we show the accuracy score over the test data for 100 epochs and batch size of 50 samples.

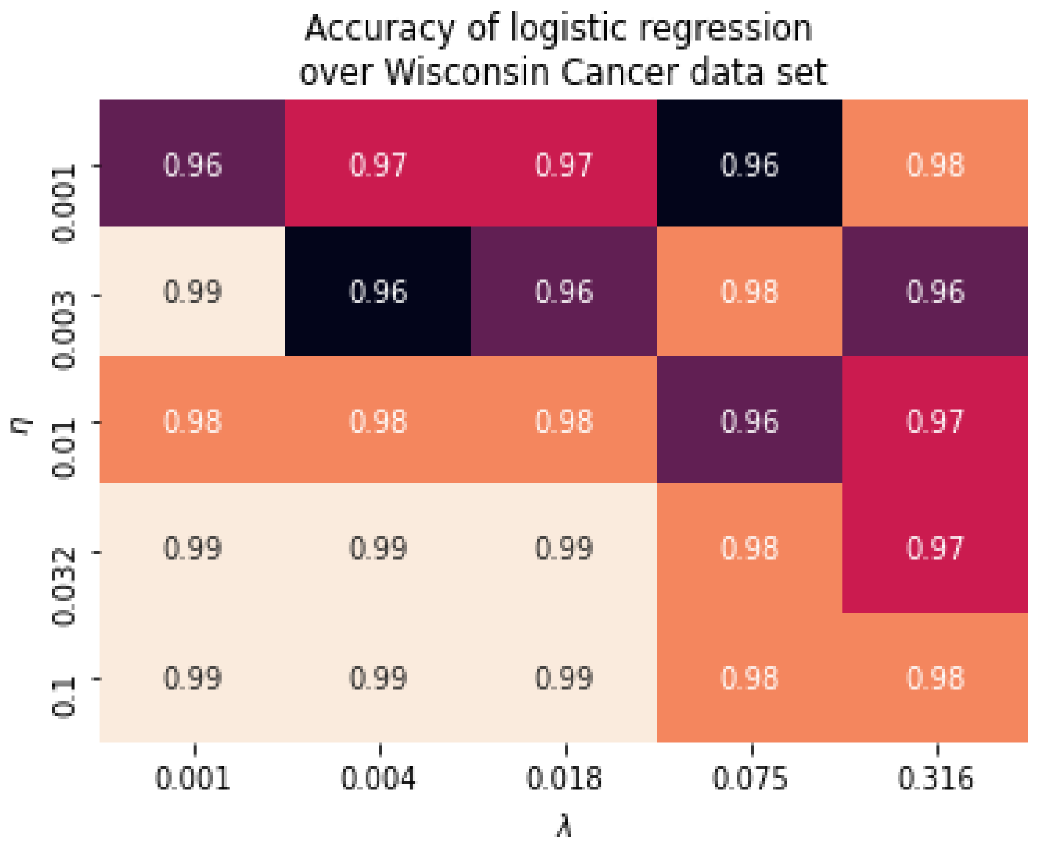

Figure 7: Variations in the performance as function of learning rate and penalty parameters.

It is worth pointing out once again the stability of this algorithm. Not only that, but also the efficiency of it. To reproduce the results in Figure 7 we had to train the model 25 times, each one of them needing a loop of $n_{epochs} \times \dfrac{n_{samples}}{n_{batch}}$, and the whole calculation only took a few seconds to complete.

We also tested our logistic regression on another binary classification problem: the Haberman's Survival data set, which consists of 306 samples of patients who had undergone surgery for breast cancer. The accuracy scores are shown in Table 2 as a function of the number of epochs, learning rate and regularization. In this case the results are not as accurate as the one obtained in the first set, and it is due to the smaller number of features and samples. We expect that a larger number of features would improve the model since there is more insight about the predictions. On the other hand, it is obvious that the more data samples we have, the better the training will be.

Epochs	η	λ	Accuracy
280	0.001	0.001	0.81
	0.001	0.316	0.81
	0.1	0.316	0.76
284	0.001	0.001	0.82
	0.001	0.316	0.82
	0.1	0.316	0.67
288	0.001	0.001	0.81
	0.001	0.316	0.55
	0.1	0.316	0.73

Table 2: Accuracy score over test data for Haberman's set.

4 Conclusion

After having tested all the methods discussed, there is no doubt that each model is best for what it is mean for: linear regression remains undefeated when it comes to fitting continuous functions, logistic regression remains a excellent choice for fitting binary classification models, and neural networks, even in their simplest form, have proven to be able to fit multiple classification problems without much effort. But there is a feature about neural networks that catches out attention: their capability of adapting to new problems. Just a slight modification in the design of the network, as it could be the number of nodes of the output layer and its activation function, allows us to move between one kind of task to another with the satisfactory results we have found out, and this is a great advantage when we want to solve a problem we hadn't encountered yet.

Nevertheless, training a neural network involves a few extra costs: first and most relevant is the long calculation times needed in the training. Even the simplest networks, as the ones we have discussed here, contain a huge amount of parameters that need to be updated through hundreds of iterations, meaning that thousands of floating point operations have to be computed. This brings us to the second point, the probability of running into overflow and underflow problems during the training, which can lead us to expend a fair amount of useless computation time. That is why, as long as we dispose of a simpler method for each problem, we should always chose it as our first option.