FYS-STK4155: Project 2

Simen Nyhus Bastnes

13. November 2020

Abstract

In this project, we want to study a few methods commonly used in data science for both normal regression problems, as well as classification problems. The methods we will be employing are the stochastic gradient descent (SGD) and feed-forward neural networks (FFNN). For regression, we will be looking at is the Franke function studied in the previous project [1], comparing our results to the ones found there. For classification the MNIST database of handwritten digits will be studied. Cross-validation will be used for the hyperparameter search for both problems in order to make sure the model is the optimal one. For the Franke function, we found that both SGD and FFNN performed worse than both OLS and Ridge, however their R^2 scores, 0.84 and 0.93 respectively, indicate that the results are not too terrible, at least not for the neural network. For the MNIST data set, both the SGD and FFNN performed very well, with an accuracy of $\sim 95\%$, surpassing the results gotten with Scikit-Learn. The FFNN results are a bit weird as it seems to converge extremely fast, with very few nodes in the network.

1 Introduction

In the last two decades, computation power and availability of said machines has increased drastically, making it possible to employ more and more complex methods of solving both regression and classification problems.

In this project, we will be looking at two different methods of performing regression and classification, and see how they compare against each other. For regression, we can also compare to the results we found in Project 1 [1]. The methods we will be looking at is the stochastic gradient descent and a feed-forward neural network. These methods contain a few hyperparameters that will need to be adjusted in order to find the best model. In order to remove some dependency on the exact training/test data split, we will employ k-fold cross-validation during the hyperparameter search.

The data sets we will be looking at is the Franke function from [2], as well as a reduced MNIST data set [3] consisting of approximately 1800 handwritten digits. First, in Chapter 2 we will briefly introduce the theory of logistic regression, and then go more in depth on stochastic gradient descent and artificial neural networks. A more in-depth description of the data sets can be found in Chapter 3. Then, in Chapter 4 we go through the results of both the Franke function and the MNIST data set, while discussing them. Finally, we conclude our findings in Chapter 5.

2 Theory

2.1 Logistic regression

Like with linear regression, our model is the matrix equation $\mathbf{X}\beta + \boldsymbol{\epsilon}$, with $\mathbf{X}$ being the design matrix consisting of n inputs and p predictors. The regression parameters β is a matrix n_{classes}, p for a classification problem with K categories/classes. We want the output to be the probability that the input belongs to a specific class. From [4] Eq 4.18, we get that the probabilities are.

$$\Pr(G = k|X = x) = \frac{\exp(\beta_{k0} + \beta k^{\mathsf{T}} x)}{1 + \sum_{l=1}^{K-1} \exp(\beta_{l0} + \beta_l^{\mathsf{T}} x)}, \text{for}\, k = 1, \dots, K-1$$
$$\Pr(G = K|X = x) = \frac{1}{1 + \sum_{l=1}^{K-1} \exp(\beta_{l0} + \beta_l^{\mathsf{T}} x)}$$

And the probabilities sum to one. To classify a sample, we take $\text{argmax}(Pr)$. To fit this, we use the maximum likelihood, and want to maximize the probability of seeing the observed data. The log-likelihood for N observations is given by

$$\ell(\theta) = \sum_{i=1}^{N} \log p_{g_i}(x_i; \theta)$$

where $p_k(x_i; \theta) = \Pr(G = k|X = x_i; \theta)$. The cost/loss function is the negative log-likelihood. Following [5] we get the cost-function for the so-called cross-entropy for N observations and K classes

$$C(\beta) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} y_k^{(i)} \log(p_k^{(i)})$$

where $y_k^{(i)}$ is a one-hot vector (equal to 1 if the class is k, 0 otherwise). As with linear regression we can add additional regularization terms like we did with Ridge and Lasso regression. The question now is how to find the optimal regression coefficients β, and one of these methods is the gradient descent method.

2.2 Stochastic gradient descent

Gradient descent is an optimization algorithm that finds local minimums by computing the gradient for the loss function with regards to the current coefficients β (or weights when dealing with neural networks). Since the gradient points in the direction of the fastest ascent, the negative gradient points in the direction where the loss function decreases the most. Then, we take a step towards the local minimum by subtracting the coefficients with some factor multiplied with the gradient w.r.t. that coefficient. This is the learning rate, and picking the correct value for this learning rate is critical in how fast the GD converges. Too large, and the solution won't converge. If the loss function is convex, the local minimum is also the global minimum. The starting point $\beta^{(0)}$ is usually set to random normal distributed values.

Algorithm 1 shows the normal gradient descent method, where η is the learning rate (where we do a set amount of steps/epochs, but we could have just continued until the gradient reaches zero, or below some threshold)

Algorithm 1 Gradient descent

```
1: Given $\mathbf{X}$, $y$, $\beta^{(0)}$, $\eta$
2: for $i = 0, N_{\text{epochs}} - 1$ do
3:     Compute $\nabla_i$ from $\beta^{(i)}$
4:     (Update learning rate $\eta$ if not using constant learning rate)
5:     $\beta^{(i+1)} = \beta^{(i)} - \eta\nabla_i$
6: end for
```

One of the problems with the gradient descent is that it is extremely computationally heavy to compute the full gradient at every step, especially for a large set of inputs and parameters. Another problem is that you end up at a local minimum, so results are very dependent on the starting values. One of the ways to reduce these problems is the stochastic gradient descent, where instead of computing the full gradient, for each step, a single random instance of the training set is chosen, and the gradient descent step is done based only on that instance. Thus, each step is significantly faster, but each step won't take the direction of the fastest descent due to the random nature. However this also means that the algorithm can jump out of local minima, making it more likely to find the global minimum. In this project specifically, we will be implementing the minibatch stochastic gradient descent, where we compute a random mini-batch for each step instead of just one instance. Algorithm 2 shows the minibatch SGD.

Algorithm 2 Stochastic Gradient descent

```
1: for $i = 0, N_{\text{epochs}} - 1$ do
2:     Shuffle $\mathbf{X}_{\text{train}}$ and $y_{\text{train}}$
3:     Split into $N_{\text{mb}}$ minibatches
4:     for $j = 0, N_{\text{mb}}$ do
5:         Compute gradient using minibatch $j$
6:         Update $\beta$
7:     end for
8: end for
```

One of the ways we can assure that the SGD stops at a reasonable point is to scale the learning rate with the number of steps taken. This way, the first steps we do are the largest, and allows the algorithm to jump around the different minima, and as we continue, the step size is decreased and we try to go towards the minimum we are currently in. How the learning rate changes over time is often called the learning schedule, and the way we will be testing in this project is as follows.

$$\eta_t = \frac{t_0}{t + t_1}$$

where t_0 and t_1 are parameters we have to tune, and $t = i \cdot N_{\text{mb}} + j$ (following the notation from Algorithm 2)

For our regression problem, with a mean squared error loss function, the gradient can be expressed as

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

And for multiclass logistic regression with cross-entropy as our loss function, the gradient is given by

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

Ridge regularization (L^2 norm) can be added to the gradients by simply adding a term $\lambda \cdot \beta$.

2.3 Artificial neural networks

The concept behind an artificial neural network (ANN) is a system that attempts to simulate the behavior of neurons in a human mind. While the idea was introduced as early as 1943 [6], its use has only really exploded in recent years thanks to the processing power available. The type of neural net we are looking at here is the so-called feed-forward neural net, which consists of a number of "neurons" or "nodes" structured in several layers, where each node in one layer is connected to all the nodes in the next layer. This net has at least three layers; one input layer, one output layer, and at least one hidden layer. From the universal approximation theorem, a neural net with at least one hidden layer can be used to approximate "all" types of functions. The amount of neurons in each layer and the amount of hidden layers is up to the user, and determining which sets of parameters approximates the problem best can be difficult.

Information is fed through forward through the system, going from layer to layer until it reaches the output layer. Each connection between nodes have a weight associated with them, giving us that the input in the first hidden layer is given by the matrix equation

$$z_i = \mathbf{w_i}^T\mathbf{x} + b_i$$

where $\mathbf{x}$ is the input, $\mathbf{w}$ is the weights connecting the node in the hidden layer with all the nodes in the input layer, and b_i is the bias term for each node. This is then fed through an activation function $\sigma(z)$ which determines whether or not that node "fires" based on the input, giving us that the output from a node is given by

$$a_i = \sigma(z_i)$$

Examples of the activation functions used is shown in Section 2.5, and is also an important part in trying to figure out which model fits best. Each layer can have different activation functions. Once all the outputs a for a layer is computed, we continue to the next layer all the way until the end. This is called the forward pass, and the output is the neural nets prediction given the input.

2.4 Back-propagation

Since the first prediction is probably pretty bad, we need to adjust the weights. This is done via gradient descent, in a process called back-propagation. Using the loss function, we compute the gradient, and go backwards through the neural net to update the weights and bias. Then, the process is repeated based on the new prediction.

Following [7], the first step in the back-propagation is to compute the error in the output layer

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

we see that it depends on the derivative of both the loss function and the output layer activation function. The exact expressions for our cases is given in Section 2.6. Then we compute the error recursively for the previous layers

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

Then we have the following equations for the gradient w.r.t. the bias and weights

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$$
$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

Which we can use to update the weights and bias like we discussed earlier for gradient descent.

2.5 Activation functions

In order to do the forward pass and back-propagation, we need expressions for both the activation function $\sigma(z)$ and their derivative. Some of the most common activation are listed below.
First, we have the sigmoid function

$$\sigma(z) = \frac{1}{1+\exp(z)}$$

and its derivative

$$\sigma'(z) = \sigma(z)(1-\sigma(z))$$

ReLU (Rectified Linear Unit)

$$\sigma(z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

and its derivative

$$\sigma(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

Leaky ReLU

$$\sigma(z) = \begin{cases} z & \text{if } z \geq 0 \\ 0.01z & \text{if } z < 0 \end{cases}$$

and its derivative

$$\sigma(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0.01 & \text{if } z < 0 \end{cases}$$

Hyperbolic tangent

$$\sigma(z) = \tanh(z)$$

and its derivative

$$\sigma'(z) = 1 - \tanh(z)^2$$

For our output layers we will be using the following activation functions. For regression, the activation function is just identity.

$$\sigma(z) = z$$
$$\sigma'(z) = 1$$

And for classification we use the softmax function

$$\sigma(z_i) = \frac{\exp(z_i)}{\sum_k \exp(z_k)}$$
$$\sigma'(z_i) = z_i(\delta_{ij} - z_j)$$

where δ_{ij} is the Kronecker delta.

2.6 Loss functions and their gradient

Since the error of the output layer depends on both the gradient of the loss function as well as the derivative of the output activation function, we want to look at what that expression is for our cases. First, for regression, we use the mean squared error as our loss function. For a single sample, this is

$$C_{\text{MSE}} = \frac{1}{2}(a^L - y)^2$$

the derivative with respect to a is then

$$\frac{\partial C_{\text{MSE}}}{\partial a^L} = a^L - y$$

With the identity function as the output activation function, this gives us that the error in the output layer is

$$\delta^L = \nabla_a C \odot \sigma'(z^L) = a^L - y$$

For the classification problem, we use the softmax function as the output activation function, and Cross-Entropy for the loss function.

$$C_{\text{CE}} = -\sum_{i=1} y_i \log a^L$$

The output error can be shown to be

$$\delta^L = a^L - y$$

which is very nice and the same as we found for the regression case. With other combinations of activation and loss functions this will probably not be as well behaved.

2.7 Learning rate and weight initialization

For the learning rate, we will be looking at the two same methods as described earlier in Section 2.2, where we either keep it constant, or scale it with the number of iterations.

In order to start the training of the neural network, we need to set some starting value for the weights and bias. One way to do this is to set the weights to random normal distributed values. The bias is set to some small value in order to avoid problems with vanishing gradients.

Another method of initializing the weights and bias is the way described in [8], where the weights and bias is drawn from a uniform distribution between

$$\left[-\frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j+n_{j+1}}}\right]$$

where n_j is the amount of nodes in the previous layer, and n_{j+1} is the amount of nodes in the current layer.

2.8 Measure of performance

In order to gauge how good our model functions, we will be looking at the mean squared error and R^2 score for the regression problems. The equations for those can be found in [1]. For the classification case, we are interested in the accuracy score, which is the number of correctly guessed targets t_i divided by the total number of targets.

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

where I is the indicator function, which is 1 if $t_i = y_1$ and 0 otherwise. y_i is the output from the forward pass.

3 Data sets

In this project, we will be using two different data sets. The first, is the Franke function we looked at in the previous project [1]. The equation for the Franke function can be found there and in [2]. This function is a two-dimensional function that has been widely used for testing implementations for regression and interpolation.

The second data set is the so-called MNIST data set [3], which is a large data set consisting of handwritten digits from 0 to 9, and is commonly used for testing various types of classification methods. The original data set consists of 70 000 images of size 28×28 pixels. However, as the scope of this project is fairly limited, we will be looking at a reduced version of the data set both in size and resolution in order to have time to produce results. The specific version of the data set we will be using is the one available within the scikit-learn python package, and consists of 1797 images of size 8×8 pixels. Figure 1 shows an example of some of the images in the data set. For the full resolution MNIST data set, human accuracy has been noted to be roughly 0.2%, according to [9]. However, given the reduction in resolution, it is not unlikely that it would be somewhat lower for the data set we will be looking at.

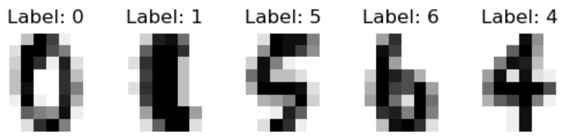


Figure 1: Some of the images in the reduced MNIST data set, with their label printed above.

4 Results and discussion

The code used to generate the results presented in this section can be found in the Github repository [10]. For both data sets, we employ cross-validation when searching for the hyperparameters. The results section will be split into, the first one pertaining to the Franke function, while the second contains the analysis of the MNIST data set

For simplicity, for all the neural network results we will deal with hidden layers consisting of identical amount of nodes, using the same activation functions for all layers. Strictly speaking there is nothing stopping us from having arbitrary choices of amount of nodes for the layers in the network, and some neural net structures (like for example autoencoders) rely on that. Taking this into account drastically increases the potential things to test, and is thus outside of the scope of this report.

4.1 Franke function

First, we look at the results from the Franke function. As we are primarily interested in seeing how both the SGD and FFNN compares our results in [1] where we used OLS and Ridge-regression, we will be looking at the same realization of the Franke function, with $N = 529$ randomly drawn points with a noise level of $\sigma = 0.05$.

4.1.1 Stochastic Gradient Descent

For the stochastic gradient descent results, we use the same approach to creating the design matrix $\mathbf{X}$ as we did in [1], where each predictor is a combination of polynomial degrees of x and y. While we could have performed the SGD for the same range of polynomial degree p as we did there, due to time limitations we will focus on the values of p which gave the best test error for both OLS and Ridge. This was found to be $p = 8$ for OLS, and $p = 15$ for Ridge. The hyperparameters we want to adjust for our stochastic gradient descent is shown in Table 1.

Table 1: Different hyperparameters for minibatch SGD.

Hyperparameter	**Description**
N_{epochs}	Number of epochs
batch size	size of each minibatch
η_0	Learning rate
λ	L^2 regularization strength

As mentioned earlier, we also want to look at a training schedule where the learning rate gets smaller with the number of iterations. Sadly, due to time constraints, we will only compare a single set of (t_0, t_1) against the number of epochs, as it is the most relevant.

With the learning rate set as constant, we try to fit our model with the following sets of hyperparameters, going through each combination and computing the test error for each one.

$$\begin{aligned} N_{\text{epochs}} &\in \{10, 25, 50, 100\} \\ \text{batch size} &\in \{1, 5, 10, 50\} \\ \eta_0 &\in \{0.1, 0.01, 0.001\} \\ \lambda &\in \{0.0, 0.1, 0.01, 0.001\} \end{aligned}$$

Table 2 shows the best-fit parameters for this data set and set of hyperparameters.

Table 2: Best-fit hyperparameters for Franke function SGD, $p = 8$

Parameter	Value
N_{epochs}	100
batch size	5
η_0	0.1
λ	0.0

To see how the different hyperparameters interact, we plot heatmaps with the different hyperparameters on the x and y axis, and either MSE or R^2 for the z values. Since both of them should show the same trends, we opt to look at the R^2 score since the specific values makes it easier to determine how good the fit is.

Figure 2 shows the R^2 score for the number of epochs plotted against the learning rate, where all the other parameters are set to the best-fit parameter shown in table 2. We see that the smallest learning rate and fewest amount of epochs give the worst fit by far. As either the step size is increased or the number of epochs, the fit improves, indicating that the model manages to converge more towards a minimum.

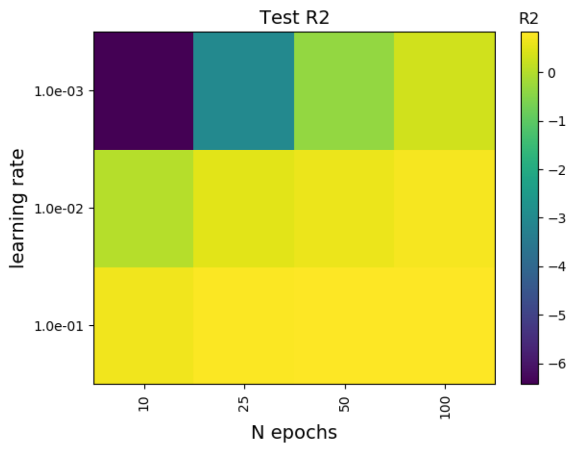


Figure 2: R^2 score for number of epochs plotted against learning rate. All other parameters are set to the best-fit values.

Figure 3 shows N_{epochs} plotted against batch size. We see that most values yield decent results, though higher N_{epochs} and batch size greater than 1 gives the best result. For some reason, $N_{\text{epochs}} = 100$ and batch size $= 1$ gives a really horrible result, which might indicate that the learning rate is not ideal for that specific case, or some other parameter. Since the value is the average of all the k-folds, it seems unlikely that it is purely due to back luck with starting values.

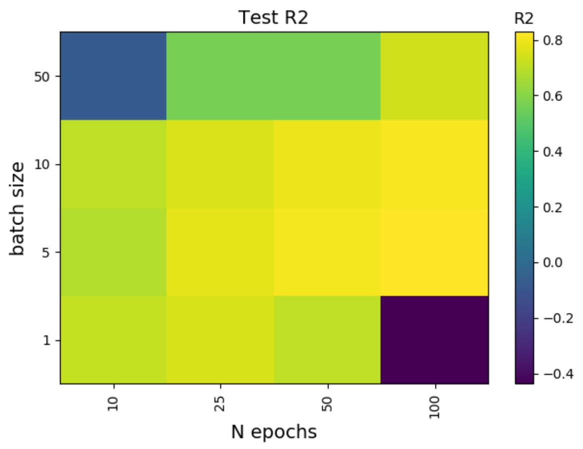


Figure 3: R^2 score for number of epochs plotted against the batch size. All other parameters are set to the best-fit values.

Figure 4 shows the number of epochs plotted against the Ridge regularization λ. Like we saw in project 1, the model seems to prefer the lower values of λ, and the largest $\lambda = 0.1$ gives terrible results for all N_{epochs}.

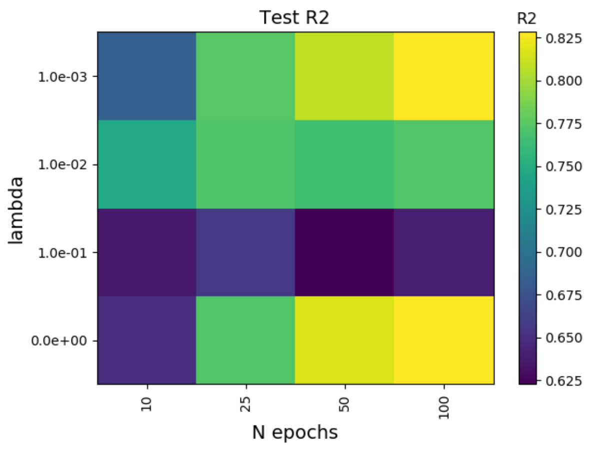


Figure 4: R^2 score for number of epochs plotted against regularization λ. All other parameters are set to the best-fit values.

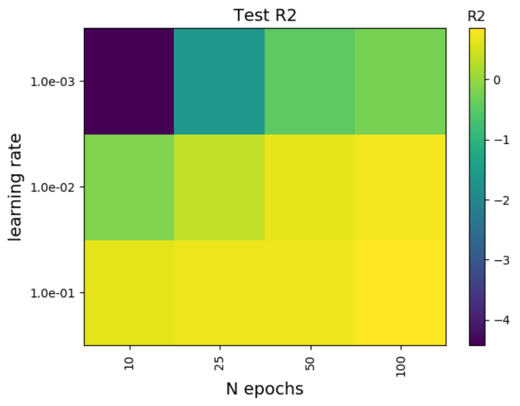
Moving on to the $p = 15$ case, we fit the model with the parameters, and get the following best fit, shown in Table 3

Table 3: Best-fit hyperparameters for Franke function SGD, $p = 15$

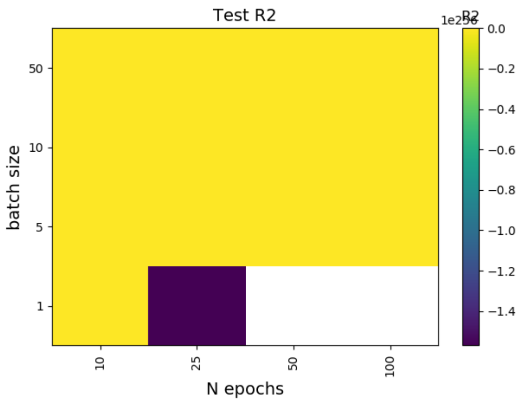
Parameter	Value
N_{epochs}	100
batch size	5
η_0	0.1
λ	0.001

Which interestingly enough is the same as for $p = 8$ except that the regularization is now $\lambda = 0.001$ instead of zero. The preferred λ for Ridge in [1] was $5.46 \cdot 10^{-6}$, however we did not test for λ values that low.

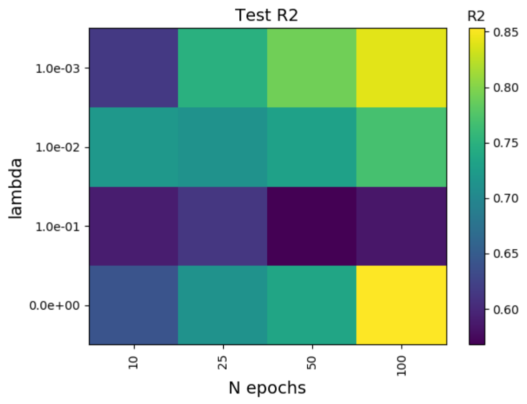
Figure 5 shows the same three heatmap plots plotted for $p = 15$. The learning rate vs number of epochs is mostly identical, however the batch size plotted against N_{epochs} shows that something weird is happening. For some reason, the smallest batch size causes an overflow. The bottom panel, showing the regularization plotted against N_{epochs} is also mostly similar to the one shown in Figure 4



(a) N_{epochs} vs learning rate



(b) N_{epochs} vs batch size



(c) N_{epochs} vs λ

Figure 5: R^2 score for different combinations of hyperparameters. All other parameters are set to the best-fit values shown in Table 3.

Comparing the results more directly with OLS and Ridge, we get Table 4, where we see that our results are not quite as low as OLS and Ridge. However, the amount of parameters we checked was somewhat limited, so a more thorough analysis would have been a good idea. The R^2 score for our SGD was around 0.83, so its not directly bad.

Table 4: Comparison between the test MSE for SGD, OLS, and Ridge for $p = 8$ and $p = 15$. OLS and Ridge results taken from [1].

p	SGD	OLS	Ridge
8	$1.2 \cdot 10^{-2}$	$3.3 \cdot 10^{-3}$	$3.5 \cdot 10^{-3}$
15	$1.0 \cdot 10^{-2}$	$9.7 \cdot 10^{-3}$	$3.3 \cdot 10^{-3}$

To verify our results, we compare to the SGD functionality in the python package Scikit-Learn. Table 5 shows the MSE and R^2 score for both our code and Scikit-Learn. Confusingly enough, my model seems to outperform the Scikit-Learn method, despite trying to set similar settings, though looking more properly at what it does might reveal some implementation differences not accounted for. Curiously enough, while only the test error is printed here, both my own SGD and SKL gets very similar results for train and test.

Table 5: Comparison between MSE and R^2 for our code and Scikit-Learn.

p	SGD MSE	SKL MSE	SGD R^2	SKL R^2
8	0.012	0.032	0.82	0.53
15	0.010	0.031	0.84	0.54

Finally, before moving on to neural networks, we test with a non-constant learning rate. To test this, we set $N_{\text{epochs}} \in \{10, 25, 50, 75, 100, 200\}$ as well as the different lambdas we tested for earlier. For $t_0 = 1$ and $t_1 = 10$, we get Figure 6, which surprisingly shows that the largest regularization is preferred, although the fit itself (with the maximum R^2 score at around 0.5) is not very good. We also see that not just the largest N_{epochs} gives the best fit, which makes sense since we scale the learning rate with iterations, and after some point there is no reason to continue iterating.

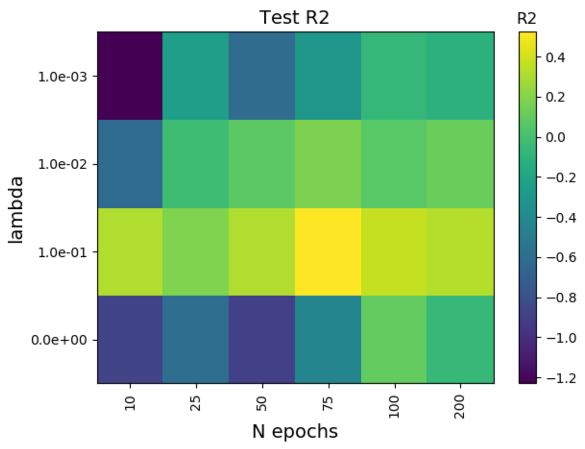


Figure 6: R^2 score for number of epochs plotted against regularization λ with a learning rate that decreases over time ($t_0 = 1$, $t_1 = 10$). All other parameters are set to the best-fit values for $p = 8$.

4.1.2 Feed-forward Neural Network

Now, it is time to look at the neural network. The design matrix this time is not the polynomial matrix we used for SGD, instead we use the x and y coordinates of the Franke function as our two input predictors. Like with SGD, we have the same hyperparameters from Table 1 as well as the amount of nodes in each hidden layer $N_{h,\text{nodes}}$, and the amount of hidden layers $N_{h,\text{layers}}$. We fit the model with the following parameters

$$\begin{aligned}
N_{\text{epochs}} &\in \{10, 25, 50, 100\} \\
\text{batch size} &\in \{1, 5, 10, 50\} \\
\eta_0 &\in \{0.1, 0.01, 0.001\} \\
\lambda &\in \{0.0, 0.1, 0.01, 0.001\} \\
N_{h,\text{nodes}} &\in \{10, 25, 50\} \\
N_{h,\text{layers}} &= 1
\end{aligned}$$

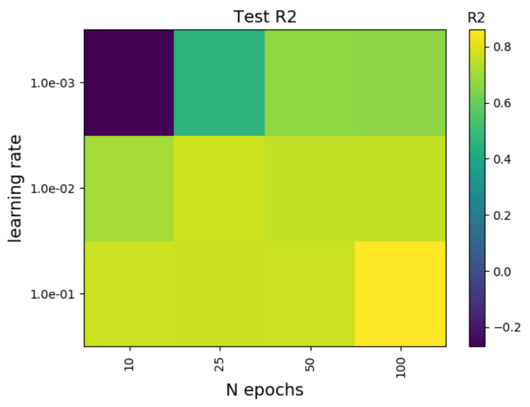
and a constant learning rate, random weights/bias initialization, and the sigmoid function as the activation function for the hidden layers. It is worth noting that the tests were done with only one hidden layer, as we already had 576 combinations to test, and just adding one more hidden layer would double the amount of testing, which while not the biggest issue with this specific data set, would have caused some problems time wise. This topic is further discussed in Section 4.3. After finding the fit for these parameters we will look at the some combinations of $N_{h,\text{nodes}}$ and $N_{h,\text{layers}}$ directly in order to see if the model prefers higher amounts of hidden layers.

Table 6 shows the best fit parameters for the neural net with the combinations specified above. We plot the resulting heatmaps to study how they depend on each other.

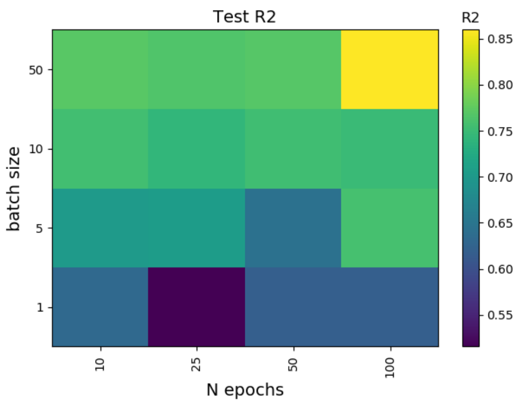
Table 6: Best-fit hyperparameters for Franke function using our neural network code. The amount of hidden layers $N_{h,\text{layers}}$ had only one combination to test, and mostly added here for completeness.

Parameter	Value
N_{epochs}	100
batch size	50
η_0	0.1
λ	0.001
$N_{h,\text{nodes}}$	25
$N_{h,\text{layers}}$	1

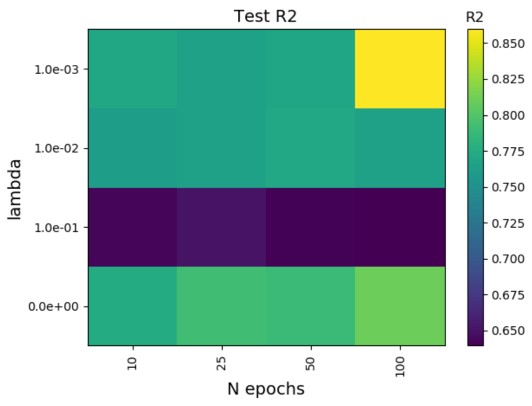
Figure 7 shows the R^2 heatmaps for 5 combinations of hyperparameters plotted against each other. Panel (a) shows the same behavior we saw for SGD, with the lowest amounts of epochs and learning rate does not have time to converge. Panel (b) shows is a bit interesting though, as unlike SGD, it seems to vastly prefer larger minibatches. The batch size will be discussed a bit more later when comparing to Scikit-Learn. Panel (c) shows the that the largest regularization performs the worst, like with SGD. The preferred model however is the smallest λ combined with the highest N_{epochs}. Panel (d) shows that few combinations are "good", however the scale on the z axis starts at roughly $R^2 = 0.74$, so the differences aren't too large. Panel (e) shows that the largest learning rate gives the best results regardless of $N_{h,\text{nodes}}$.



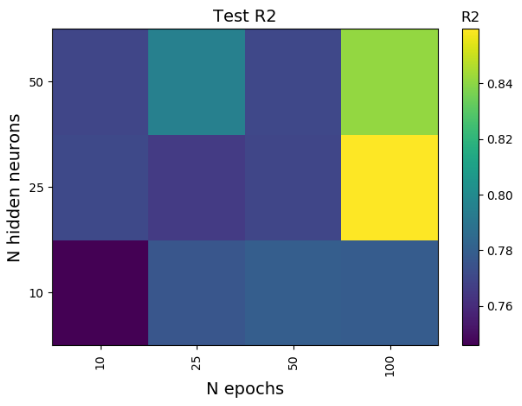
(a) N_{epochs} vs learning rate



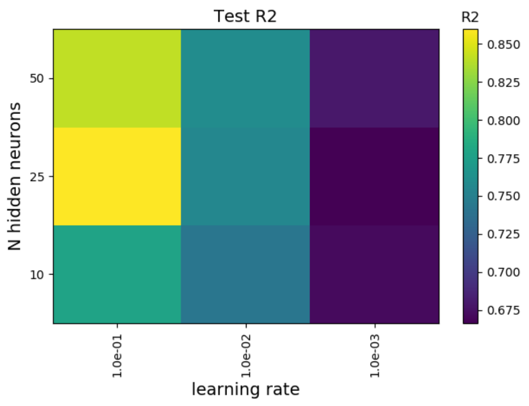
(b) N_{epochs} vs batch size



(c) N_{epochs} vs λ



(d) N_{epochs} vs $N_{h,\text{nodes}}$



(e) Learning rate vs $N_{h,\text{nodes}}$

Figure 7: R^2 score for different combinations of hyperparameters. All other parameters are set to the best-fit values shown in Table 6.

Finally, Table 7 shows the MSE (and R^2 for NN/SGD) score for NN, SGD, OLS and Ridge. We see that our neural network outperforms the SGD by a factor of 2, though is still not quite as good as OLS or Ridge. That said, we did not take the amount of hidden layers into consideration, and it is worth checking how changing the activation function affects the results. The final R^2 score for our neural net is 0.93 though, which is a really good fit.

Table 7: Comparison between MSE scores for NN, SGD, OLS and Ridge. The R^2 score is added for NN and SGD, however the same results for OLS and Ridge are not available without going back to the code used in [1].

Method	MSE	R^2
NN	$5.0 \cdot 10^{-3}$	0.93
SGD	$1.0 \cdot 10^{-2}$	0.84
OLS	$3.3 \cdot 10^{-3}$	-
Ridge	$3.3 \cdot 10^{-3}$	-

Testing the amount of hidden layers

Since our best-fit found previously didn't take the amount of hidden layers into consideration, we perform a test where all the other parameters except the amount of hidden layers and nodes per hidden layer is kept at the best-fit parameters from Table 6. We test the following combinations

$$N_{h,\text{nodes}} \in \{10, 25, 50\}$$
$$N_{h,\text{layers}} \in \{1, 2, 3, 4\}$$

Figure 8 shows the R^2 score heatmap for $N_{h,\text{nodes}}$ plotted against $N_{h,\text{layers}}$. We see that the trend is towards a higher amount of total nodes ($N_{h,\text{layers}} \cdot N_{h,\text{nodes}}$), however the best fit is for 4 hidden layers and 25 nodes per layer. All of them however give a fit of $R^2 > 0.8$, so none of them are bad.

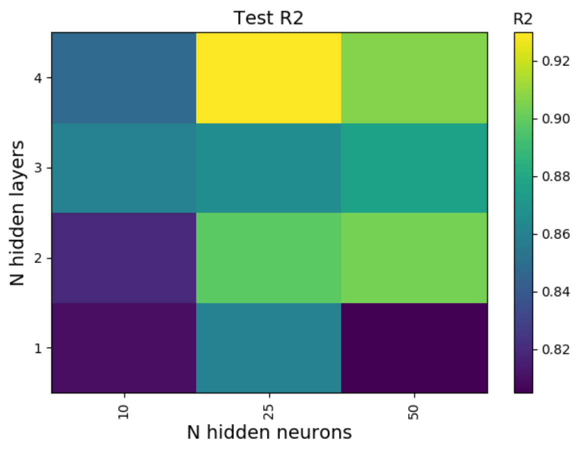


Figure 8: R^2 score for number of hidden layers vs number of nodes per hidden layer. All other parameters are set to the best-fit values.

Testing different activation functions

We test different activation functions for the hidden layers in order to see how the results depend on it. The results of this is shown in Table 8, with the Scikit-Learn results added for reference.

Interestingly enough, the using the sigmoid/logistic function gives us the worst fit of them all. The tanh-function almost halves the MSE with the sigmoid function, and both ReLu's are almost 1/5th of the sigmoid function. Using the leaky ReLu we end up with an R^2 score that is almost 0.9, which is extremely good. Scikit-learn however is a bit better then our sigmoid results, despite using the same parameters.

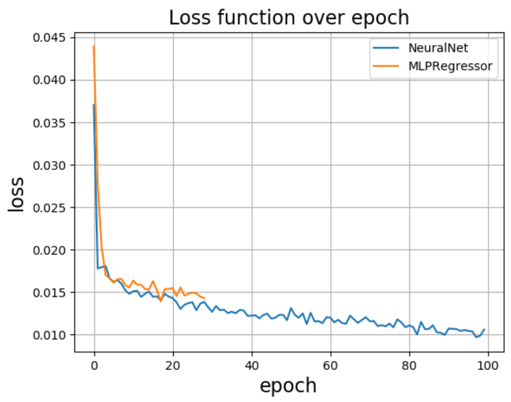
Table 8: Comparison between the train/test MSE and R^2 scores for different activation functions. The hyperparameters are $N_{\text{epochs}} = 50$, batch size= 1, $\eta_0 = 0.1$, $\lambda = 0$, and one hidden layer with 25 nodes. For reference, the Scikit-Learn value found using the logistic function as $\sigma(z)$ is added to the bottom.

$\sigma(z)$	Train MSE	Test MSE	Train R^2	Test R^2
sigmoid	0.027	0.030	0.586	0.566
ReLu	0.006	0.007	0.904	0.888
leaky ReLu	0.006	0.007	0.909	0.893
tanh	0.014	0.015	0.784	0.768
SKL	0.021	0.023	0.685	0.653

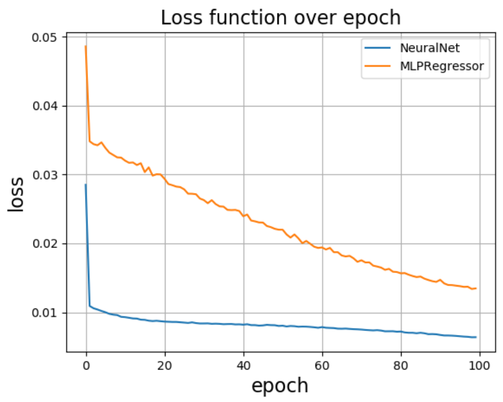
Batch sizes and comparison with Scikit-Learn

As noted earlier, Figure 7 panel (b) showed that the system highly prefers larger minibatches, differing somewhat from how SGD behaved. In order to test this we will compare our results with Scikit-Learn, using mostly the same parameter choices as our best fit. We plot the loss function directly for each epoch, but for some reason adding regularization to the mix seems to cause some scaling difference with how it is added to the loss function, so we disregard it here.

Figure 9 shows the loss curve comparison between our code and Scikit-Learn for batch sizes 1 and 50. We see that for batch size = 1, our own code follows the SKL curve very closely, with some start difference that may stem from SKL using Glorot initialization [8] for their weights and bias. Also, SKL stops before reaching the maximum number of epochs, which we failed to deactivate. For batch size 50 however, our results are extremely different, yielding much better results than SKL. This could be due to how the minibatches are made and drawn for each iteration, but after looking at the code for MLPRegressor I am not sure how much of a difference there is.



(a) Batch size = 1



(b) Batch size = 50

Figure 9: Loss curve plotted against epoch for both our code and Scikit-Learn's MLPRegressor. Left panel shows batch size = 1, while right panel shows batch size = 50. Parameters are the best-fit, except that $\lambda = 0$.

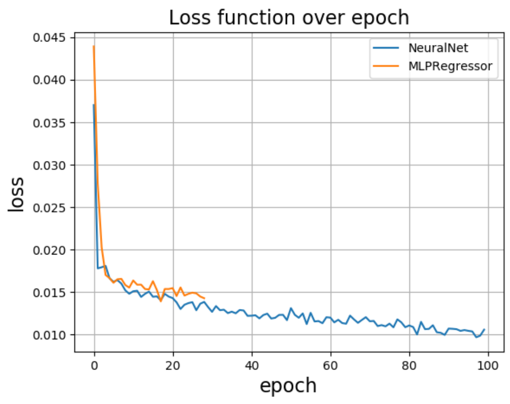
The same results can be seen in Table 9, where we see that both the MSE and R^2 score is better for both of the batch sizes.

Table 9: Comparison between MSE and R^2 for our code and Scikit-Learn, with different batch sizes. Parameters are the best-fit, except that $\lambda = 0$.

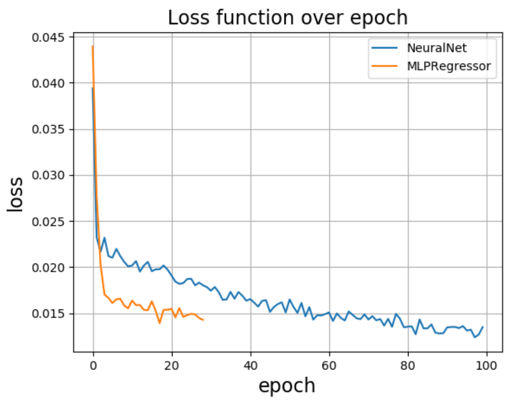
batch size	NN MSE	SKL MSE	NN R^2	SKL R^2
1	0.013	0.037	0.803	0.442
50	0.011	0.030	0.838	0.553

Weights and bias initialization

We briefly look at how the results depend on initialization. Figure 10 shows the loss curve for our code using either random or Glorot initialization. We see that our results are better when using random initialization , and more in line with what MLPRegressor gets (until it stops) despite it using Glorot initialization.



(a) Random initialization



(b) Glorot initialization

Figure 10: Loss curve plotted against epoch for both our code and Scikit-Learn's MLPRegressor, with different weights/bias initialization scheme. Parameters are the best-fit, except that $\lambda = 0$. Both of the SKL plots use Glorot initialization.

Non-constant learning rate

Lastly, we take a look at using a scaling learning rate instead of constant. We use the best fit settings with batch size set to 1 and $\lambda = 0$. Figure 11 shows the loss curve with learning rate set to the one discussed earlier, with $t_0 = 1$ and $t_1 = 10$. We see that until the point that SKL stops, our own code follows, which is similar to what we saw with a constant learning rate. However afterwards, the loss barely changes, which makes sense as the learning rate gets vanishingly small as the iteration count increases, so stopping around the point where SKL stops seems to make sense in that case (despite SKL using a constant learning rate, so not sure why it ends up stopping there). This corresponds to an R^2 score of about 0.66, which is not nearly as good as we got for the constant learning rate.

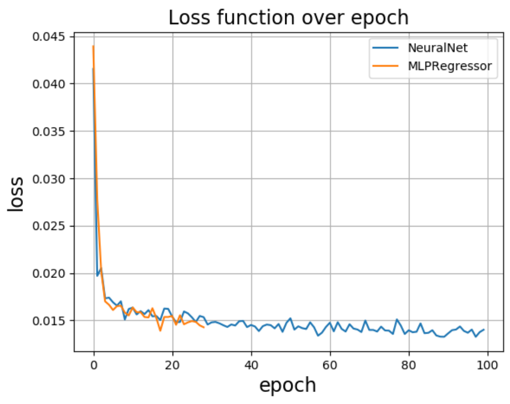


Figure 11: Loss curve plotted against epoch for both our code and Scikit-Learn's MLPRegressor, with non-constant. Parameters are the best-fit, with batch size = 1 and $\lambda = 0$

4.2 MNIST data set

In this section we will be looking at the results from analysing the reduced MNIST data set described in Chapter 3. The design matrix we use for both the NN and SGD results will be all the images, with each pixels as one predictor. This gives us an 1797×64 design matrix.

4.2.1 Classification with Neural Network

Using the same neural network code as for the Franke function, we have the same hyperparameters to adjust. The activation function for the hidden layer is the sigmoid function like with the regression case, but the output activation function is set to the softmax function, and the loss function as cross-entropy. We fit our model using the following parameter sets (the same as for the Franke function)

$$\begin{aligned} N_{\text{epochs}} &\in \{10, 25, 50, 100\} \\ \text{batch size} &\in \{1, 5, 10, 50\} \\ \eta_0 &\in \{0.1, 0.01, 0.001\} \\ \lambda &\in \{0.0, 0.1, 0.01, 0.001\} \\ N_{h,\text{nodes}} &\in \{10, 25, 50\} \\ N_{h,\text{layers}} &= 1 \end{aligned}$$

Table 10 shows the best fit parameters for the neural net with the combinations specified above. Oddly, for some reason based on the parameters it seems like the neural net converges extremely fast given the number of epochs and size of the hidden layer.

Table 10: Best-fit hyperparameters for the MNIST data set using our neural network code. The amount of hidden layers $N_{h,\text{layers}}$ had only one combination to test, and mostly added here for completeness.

Parameter	Value
N_{epochs}	10
batch size	5
η_0	0.1
λ	0.01
$N_{h,\text{nodes}}$	10
$N_{h,\text{layers}}$	1

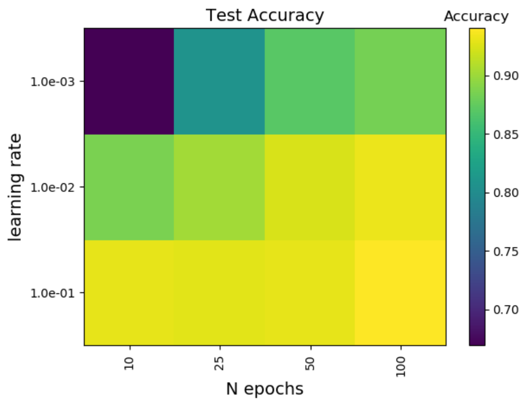
Table 11 shows the accuracy for our NN-code compared to Scikit-Learn using our best-fit parameters. Our NN-code performs extremely well, being even better than Scikit-Learn, with a test accuracy of almost 95%. It is worth noting that SKL complains complains that there is not enought iterations to converge. Setting the max iterations higher allowed changes the test accuracy to 90%, which is still not as high as our own code.

Table 11: Comparison between the accuracy score for our NN code and Scikit-Learn using best-fit parameters.

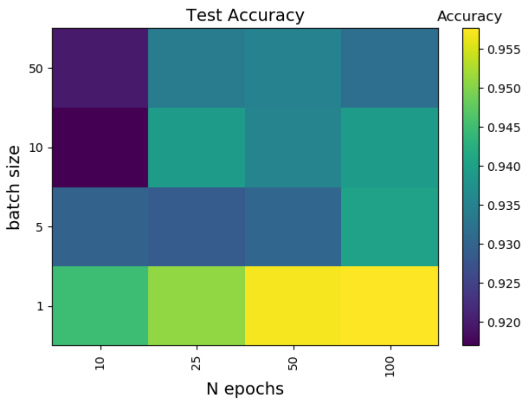
Method	Train accuracy	Test accuracy
NN	0.987	0.949
SKL	0.908	0.850

We then plot the resulting heatmaps to study how they depend on each other. Figure 12 shows the accuracy heatmaps for different combinations of hyperparameters. In panel (a) we see the same relation between N_{epochs} and learning rate that we saw earlier in both SGD and NN for the Franke function. Panel (b) shows the opposite dependency upon batch size, however looking at the z-axis, the difference is only around 3% in the accuracy.

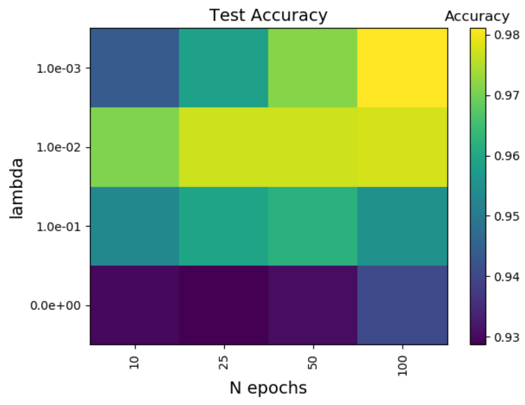
Panel (c) shows that the two lowest λ values are preferred, though no regularization performs the worst. The difference is only around 5% though. Panel (d) shows surprisingly shows that the highest N_{epochs} and $N_{h,\text{nodes}}$ yields the best accuracy, and the worst for the lowest combination. The best fit-parameters however was with he lowest parameter combination, so it shows how interconnected the parameters are, though the difference in that plot is only 3%. Finally, panel (e) shows similar results to earlier.



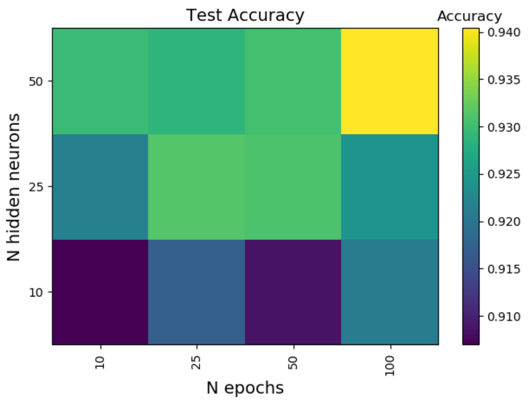
(a) N_{epochs} vs learning rate



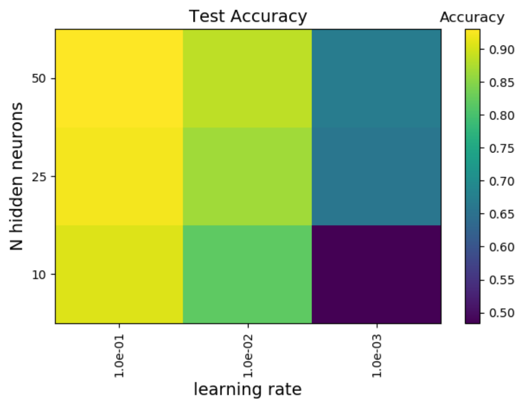
(b) N_{epochs} vs batch size



(c) N_{epochs} vs λ



(d) N_{epochs} vs $N_{h,\text{nodes}}$



(e) Learning rate vs $N_{h,\text{nodes}}$

Figure 12: Accuracy score for different combinations of hyperparameters. All other parameters are set to the best-fit values shown in Table 10.

Testing the amount of hidden layers

We test what happens when we let only the amount of nodes and amount of hidden layers to vary, testing the following combinations. All other parameters are set to the best-fit.

$$N_{h,\text{nodes}} \in \{10, 25, 50\}$$
$$N_{h,\text{layers}} \in \{1, 2, 3, 4\}$$

Figure 13 shows the accuracy score heatmap for $N_{h,\text{nodes}}$ plotted against $N_{h,\text{layers}}$. We see that for each $N_{h,\text{nodes}}$, the accuracy decreases the more layers are added, contrasting how it worked in the regression case. This does however mean that the best-fit $N_{h,\text{layers}}$ would have been 1 had we included more than one value.

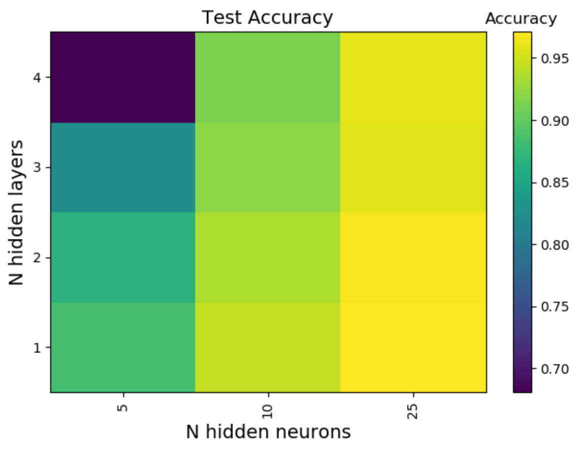


Figure 13: Accuracy score for number of hidden layers vs number of nodes per hidden layer. All other parameters are set to the best-fit values.

Testing different activation functions

We test different activation functions for the hidden layers in order to see how the results depend on it. The results of this is shown in Table 12. Both of the ReLu functions end up with a divide by zero error, causing the results to be horrible. Both the logistic function (sigmoid) and tanh gives similar and good results, with the logistic function being slightly better.

Table 12: Comparison between the train/test accuracy score for different activation functions. The hyperparameters are the best-fit found earlier.

$\sigma(z)$	Train accuracy	Test accuracy
sigmoid	0.987	0.949
ReLu	0.099	0.099
leaky ReLu	0.099	0.099
tanh	0.966	0.926

4.2.2 Logistic regression

Finally, we want to use our SGD code to fit the MNIST data set. We test the following combinations of hyperparameters, setting the learning rate to constant.

$$
\begin{aligned}
N_{\text{epochs}} &\in \{10, 25, 50, 100\} \\
\text{batch size} &\in \{1, 5, 10, 50\} \\
\eta_0 &\in \{0.1, 0.01, 0.001\} \\
\lambda &\in \{0.0, 0.1, 0.01, 0.001\}
\end{aligned}
$$

Table 13 shows the best-fit parameters for this data set and set of hyperparameters. Unlike the NN, the model might take longer to converge as the highest N_{epochs} is chosen, though the learning rate is also the lowest. The regularization factor is also the largest one.

Table 13: Best-fit hyperparameters for the MNIST data set using SGD for logistic regression.

Parameter	Value
N_{epochs}	100
batch size	1
η_0	0.001
λ	0.1

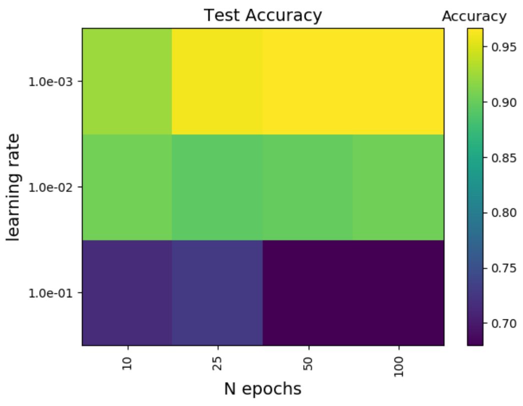
Table 14 shows the accuracy for our SGD-code compared to Scikit-Learn using our best-fit parameters. Like with the neural net, our SGD code performs extremely well, ending up with a test accuracy of 96%, even better than the NN. SKL is also very close, though some of that is due to having enough time to converge unlike the problems it had in the NN case.

Table 14: Comparison between the accuracy score for our SGD code and Scikit-Learn using best-fit parameters.

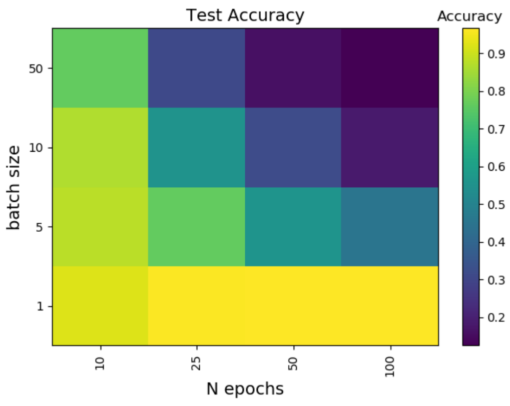
Method	Train accuracy	Test accuracy
NN	0.986	0.960
SKL	0.971	0.956

Hyperparameter heatmaps

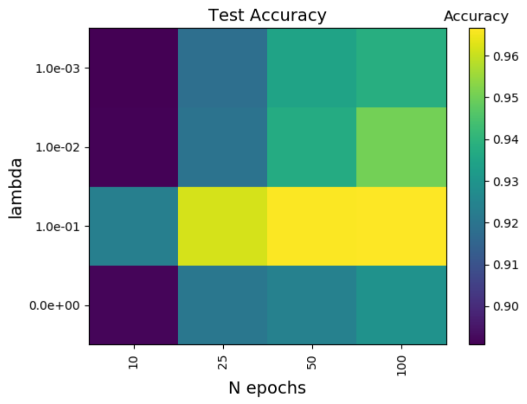
Figure 14 shows the Accuracy heatmaps for some combinations of hyperparameters. In panel (a) we see that for all possible N_{epochs}, the results get better as the learning rate is decreased. Panel (b) shows that the smallest batch sizes gives the best results, which behaves roughly as it did for NN with classification. Panel (c) shows that the best regularization is the largest λ, however the z-axis also shows that the difference in percentage isn't that massive.



(a) N_{epochs} vs learning rate



(b) N_{epochs} vs batch size



(c) N_{epochs} vs λ

Figure 14: Accuracy score for different combinations of hyperparameters. All other parameters are set to the best-fit values shown in Table 13.

Non-constant learning rate

Finally, we test with a non-constant learning rate. We set $N_{\text{epochs}} \in \{10, 25, 50, 75, 100, 200\}$ as well as the different lambdas we tested for earlier. For $t_0 = 1$ and $t_1 = 10$, we get Figure 15. We see that in this case, the largest λ value is still preferred, similar to when we did the same for the Franke function SGD model (though then, with a constant learning rate, the preferred λ was either the lowest or zero). The accuracy is best for $N_{\text{epochs}} = 100$, and decreases slightly going above that. Testing other values of t_0 and t_1 would have been a good idea.

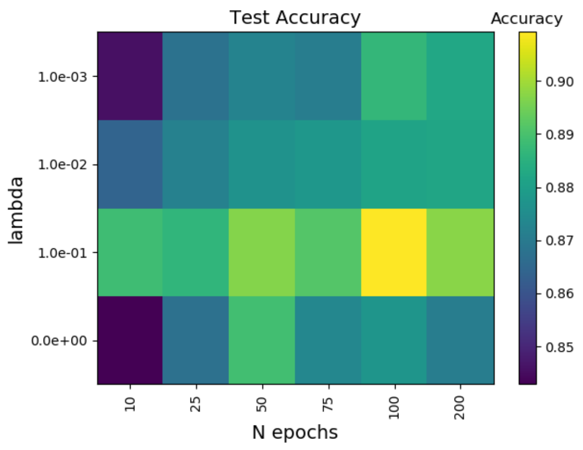


Figure 15: Accuracy score for number of epochs plotted against regularization λ with a learning rate that decreases over time ($t_0 = 1$, $t_1 = 10$). All other parameters are set to the best-fit values.

4.3 Training time

One of the big problems with methods like the stochastic gradient descent and neural networks is the high amount of hyperparameters to tweak. In this project, we used grid-search in order to try to find the optimal values for all our hyperparameters. This is a method that lends itself very well to parallelization as each parameter combination is independent of each other, though you have to go through a lot of combinations. Sadly, there was no time to implement parallelization for this project.

For the neural net results in this project, we looked at a set of 576 combinations when trying to find the optimal fit. For the Franke function, this did not take too long, but for the classification problem, this took 15.5 hours to complete. Which is why we ended up avoiding to run with more than one hidden layer in the main fit procedure, as a time estimate was made when deciding upon which parameters to try.

Imagine then if we were to use the full MNIST data set instead of the reduced one we used here, we would end up with a design matrix with almost 55 million elements compared to the 115 thousand elements we had in this report. At the very least, having a functional stopping criteria for the gradients would be beneficial. It also shows the importance of better methods for searching for hyperparameters (if you even have time to fit the models more than once), as well as doing research about what others have found.

5 Conclusion

In this project, we set out to investigate how stochastic gradient descent and feed-forward neural networks work on both regression and classification problems. We used cross-validation in order to assess which of the hyperparameter combinations gave the best fit.

For the Franke function, we found that both SGD and FFNN performed worse than OLS and

Ridge, though the neural net got much closer, with an R^2 score of 0.93, while SGD got 0.84, none of which are bad. For both, we found that the largest number of epochs $N_{\text{epochs}} = 100$ and the largest learning rate $\eta = 0.1$ gave the best results. SGD preferred a batch size of 5, with no regularization for $p = 8$, and $\lambda = 0.001$ at $p = 15$, showing similar behavior with how OLS and Ridge performed for those polynomial degrees. The neural net yielded much better results as the batch size is increased. Based on tests, it seems that using either one of the ReLu family activation functions would have improved the results.

For the classification case, where we studied a reduced version of the MNIST data set, both the SGD and Neural net performed exceedingly well, outperforming the equivalent Scikit-Learn implementation. The SGD implementation gave an accuracy of 96%, which starts to approach what you would expect from human accuracy. SKL was within a percent point away from it. The NN code yielded an accuracy of 95%, but doing so with only 10 epochs and 10 nodes in the singular hidden layer.

References

[1] GitHub repository, Project 1. `https://github.com/simennb/fysstk4155-project1`.

[2] Richard Franke. A critical comparison of some methods for interpolation of scattered data, 1979.

[3] MNIST handwritten digit database, Yann LeCun Corinna Cortes and Chris Burges. `http://yann.lecun.com/exdb/mnist/`.

[4] Hastie et al. *The Elements of Statistical Learning - Data Mining, Inference, and Prediction.* Springer, second edition edition, 2017.

[5] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow.* O'Reilly, first edition edition, 2017.

[6] Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

[7] Michael A. Nielsen. *Neural Networks and Deep Learning* - Chapter 2. `http://neuralnetworksanddeeplearning.com/chap2.html`.

[8] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. JMLR Workshop and Conference Proceedings.

[9] Dan Ciresan, Ueli Meier, and Juergen Schmidhuber. *Multi-column Deep Neural Networks for Image Classification*, 2012.

[10] GitHub repository, Project 2. `https://github.com/simennb/fysstk4155-project2`.