Draft version November 17, 2020
Preprint typeset using LaTeX style emulateapj v. 12/16/11

# FYS-STK4155 - PROJECT 1

Dahl, Jon Kristian
Bekkevik, Marit
Reaz, Fardous

*Draft version November 17, 2020*

## ABSTRACT

In this report we explore the the possibilities of neural networking, and how we can use such a device to accurately predict classification and regression problems. We give a critical evaluation of a neural networks performance by tweaking all possible parameters, for then to study the accuracy of the network in the form of mean squared error, $R^2$ score, and classification score. We also implement logistic regression as a zero-layer neural network, and benchmark its abilities to predict against our self-developed neural network classifier. For all parts of the analysis we compare our tools to one of the industry standards of machine learning, namely scikit-learn. We look at how different activation functions affect the predictive powers of the network. We implement stochastic and mini-batch gradient descent and see how changing the number and size of the mini-batches affect the result. We tweak the learning rate, $\eta$, the regularization parameter $\lambda$, and the number of epochs. We found that the logistic activation function is easy to work with in regards of learning rate and supports a large range of learning rates, compared to ReLU and leaky ReLU which are quite hard to make behave properly and demand small and precise learning rates. We found that increasing the number of hidden layers, both for regression and classification, proved to decrease the accuracy, as well as increasing the computation time, and that 1-2 hidden layers were best suited for this study. Logistic regression proved to be superior to a multi-layer neural network in classification problems, yielding larger scores at a lower computation time.

## 1. INTRODUCTION

Neural networks dates back as far as to the 1940's with McCulloch and Pitts' first computational model for a neural network McCulloch & Pitts (1943), and its popularity has only increased with the increasing availability of computing power. Today, neural networks and machine learning are very popular topics for programmers and statisticians alike, due to its enormous potential in technological advancements. The concept of a neural network is inspired by the network of neurons in the animal brain, both in the architectural construction and the way biological neurons send signals in response to specific stimuli. One can imagine how a neural network's abilities of learning from previous experience and correcting mistakes are very useful in the development of artificial intelligence. Neural network uses range from pattern recognition, mathematical optimization and data mining, to mention a few.

We give a description of how we implement a stochastic mini-batch gradient descent method, using ordinary least squares and ridge regression, without any involvement of a neural network. Mini-batch gradient descent will later be included in the neural network. We analyse and discuss the results with respect to different learning rates, number of mini-batches, number of epochs, and for ridge regression, the penalty parameter. Moreover, we compare our results to what is obtained from Scikit-Learn's SGD-methods scikit-learn developers (2020a).

Further, we implement our own feedforward neural network (FFNN) with an adjustable number of layers, number of nodes per layer, and types of activation functions. This network implementation is used for three main tasks: Linear regression, logistic regression, and classification. For linear regression we use data from Franke's function. The linear regression network is trained, and its performance is benchmarked using the mean squared error and $R^2$, by varying the learning rate, $\eta$, and the regularization parameter, $\lambda$. We also use different activation functions in the study.

The classification neural network is analysed by classifying images of hand-written numbers from the UCI Machine Learning Repository Dua & Graff (2017). The precision of the predictions are measured with a classification accuracy score. The result are again compared to the scikit-learn implementation of classification in a neural network scikit-learn developers (2020b). Finally, we use the neural network as a logistic regressor to classify the same data from the UCI Machine Learning Repository.

The concepts of the methods and tools we use, are outlined in the theory section. A desciption of our approach and implementations can be found in the method section, while the fruits of our investigations, together with a discussion, are represented in the results and discussion section. Lastly it all comes together in a conclusion in the end, where a list of references can also be found.

All code used to generate data for this report is available at the GitHub repository `https://github.uio.no/jonkd/FYS-STK4155_H20/tree/master/projects/project2`.

## 2. THEORY

The theory section is mainly based upon Morten Hjorth-Jensen's lecture notes in the Applied Data analysis and Machine learning course (FYS4155) at the University of Oslo Hjorth-Jensen (2020a). We have also used two other sources for deeper understanding of the topics, namely StatQuest Starmer (2011) and 3Blue1Brown

Sanderson (2015).

### 2.1. *Logistic regression and classification problems*

For the workings of linear regression, see our project 1 report.

For classification problems, be it a logistic regressor or a classifier, just like with linear regression we want to find relationships between explanatory variables and response variables, to be able to predict an outcome from some input data. with classifiers and logistic regressors there's not a continuous spectrum of results, but discrete values. Often it is a binary problem, for example: yes/no, win/loose, alive/dead, 0/1, -1/1. We can also add more classes, like numbers from zero to nine ($y_i \in [0, 9]$) as long as their probabilities add up to 1.

Like with linear regression, the a classification model is represented by parameters $\boldsymbol{\beta}$, allthough the curve is not a polynomial but a $\log_e$-type function given by

$$p = \frac{e^{\beta_0+\beta_1 x}}{1 - e^{\beta_0+\beta_1 x}}, \tag{1}$$

if there is one predictor $x$. With more predictors the function becomes

$$p = \frac{e^{\beta_0+\beta_1 x_1+\beta_2 x_2 \ldots+\beta_p x_p}}{1 - e^{\beta_0+\beta_1 x_1+\beta_2 x_2+\ldots+\beta_p x_p}}. \tag{2}$$

These function produce S-shaped curves. The functions can be derived from the logit- or log(odds)-function:

$$\log(\text{odds}) = \log \frac{p}{1-p} = \beta_0 + \beta_1 x_1 \tag{3}$$

$$\frac{p}{1-p} = e^{\beta_0+\beta_1 x_1} \tag{4}$$

$$p = (1-p)e^{\beta_0+\beta_1 x_1} \tag{5}$$

$$p = e^{\beta_0+\beta_1 x_1} - pe^{\beta_0+\beta_1 x_1} \tag{6}$$

$$p + pe^{\beta_0+\beta_1 x_1} = e^{\beta_0+\beta_1 x_1} \tag{7}$$

$$p = \frac{e^{\beta_0+\beta_1 x_1}}{1 - e^{\beta_0+\beta_1 x_1}}. \tag{8}$$

To find $\boldsymbol{\beta}$ it is practical to consider the graph of the logit function. Its codomain is given by

$$\lim_{p\to 0} \log\left(\frac{p}{1-p}\right) = -\infty$$

$$\lim_{p\to 1} \log\left(\frac{p}{1-p}\right) = \infty$$

With $p = e^{\beta_0+\beta_1 x_1}$ we obtain a straight line with the $\beta_0$ and $\beta_1$ representing the intercept and slope, respectively. With values running from $\pm\infty$, we can no longer find the residuals and consequently neither the mean squared error. Instead, we fit the model by using the maximum likelihood estimator.

### 2.2. *Maximum likelihood estimator*

Just like fitting a line in linear regression by minimizing the mean squared error, we maximize the likelihood in logistic regression with the maximum likelihood estimator (MLE) given by

$$p(\mathbf{y}|\mathbf{X}\boldsymbol{\beta}) = \prod_{i=0}^{n-1} p_i^{y_i}(1-p_i)^{1-y_i}. \tag{9}$$

This gives a cost function $C(\boldsymbol{\beta})$

$$C(\boldsymbol{\beta}) = -\sum_i \left[y_i \log p_i + (1-y_i)\log(1-p_i)\right]. \tag{10}$$

The minus sign comes from us wanting to minimize the value. To find the minimum we differentiate the cost function with respect to $\beta_i$:

$$\frac{\partial C}{\partial \beta_0} = -\sum i = 0^{n-1}(y_i - p_i) = \sum_i (p_i - y_i) \tag{11}$$

$$\frac{\partial C}{\partial \beta_1} = \sum_i x_i(p_i - y_i), \tag{12}$$

$$\vdots \tag{13}$$

which in matrix form equals

$$\frac{\partial C}{\partial \boldsymbol{\beta}} = \mathbf{X}^T(\mathbf{p} - \mathbf{y}) = g(\boldsymbol{\beta}), \tag{14}$$

where $g(\boldsymbol{\beta})$ is the gradient. Following the gradient descending to a minimum is the quest of the gradient descent method.

### 2.3. *Activation functions*

In a neural network, every hidden node receives an input from nodes in the previous layer. It is beneficial to pass the output from a node through whats called an *activation function* before it is fed to the next node. One can use different activation functions to tweak a neural network for a specific task.

The input, $z$, to a node is the result of the output, $a$, from all or some nodes in the previous layer, a weight $w$ and a bias $b$. The input, $z$, is passed thorugh an activation function before entering the input node. The activation function adds complexity and variability to a model, and without an activation function the model would be reduced to a pure linear regression case. There are many different activation functions, and their advantages and disadvantages are mainly found from applying the function to a neural network model. Some frequent used function are represented below.

#### 2.3.1. *Binary step function*

This simple function is also called the Heaviside function, and has the form

$$a(z) = \begin{cases} 0 & z < 0 \\ 1 & z \geq 0 \end{cases}. \tag{15}$$

This results in the neuron being active (1) when the input is over a threshold or not active (0) if not. It may not be useful when the task is a multiple classification case. Furthermore, the gradient is zero, so the weight and biases would not be updated in the backpropagation process, which we look at in 2.8.

#### 2.3.2. *Logistic function*

A Sigmoid function is a function with a characteristic S shaped curve. In this study we use the logistic function which is a Sigmoid type function. It has the form

$$a(z) = \frac{1}{1 - e^{-z}}. \tag{16}$$

The logistic function has a domain of $(-\infty, \infty)$ and a codomain of $(0, 1)$. It can thus map any value lay between zero and one. Since the gradient is not zero or constant, the update of weights and biases are flexible to learning during backpropagation, as we will see in section 2.8.

#### 2.3.3. *Hyperbolic tangent*

The hyperbolic tangent, or tanh, can be considered to be a shift of the logistic function to be symmetric around zero, giving outputs of both positive and negative sign between -1 and 1. It's on the form

$$\tanh(z) = 2\,\text{logistic}(2z) - 1. \tag{17}$$

Except for that its gradient is steeper, the tanh has similar properties as the logistic function.

#### 2.3.4. *Softmax function*

The softmax function

$$a(\mathbf{z}) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, ..., K \tag{18}$$

is sometimes described as a combination of multiple sigmoid functions, and is often used in classification problems with multiple classes. Softmax is often used as the output layer activation function since it normalizes the input vector to a probability distribution.

#### 2.3.5. *ReLU*

ReLU stands for rectified linear unit and is widely used, mainly due to the fact that it does not activate all the neurons at the same time. It is defined as

$$a(z) = \max(0, z). \tag{19}$$

For the neurons receiving outputs below zero, the gradient is zero, resulting in the weight and biases not getting updated (dead neurons). This problem is taken care of by the Leaky ReLU.

#### 2.3.6. *Leaky ReLU*

The leaky rectified linear unit is similar to ReLU, but it has a non-zero gradient, avoiding dead neurons Gupta (2020).

$$a(z) = \begin{cases} 0.01z & z < 0 \\ z & z \geq 0. \end{cases} \tag{20}$$

### 2.4. *Gradient descent*

In linear regression we have an exact expression for the optimal $\boldsymbol{\beta}$, but that doesn't exist in logistic regression. What we do have is an expression for the gradient, which is the differentiated cost function with respect to $\boldsymbol{\beta}$. One can for example use the mean squared error or the L2-norm as a cost function, to mention some. In the OLS case the cost functions are

$$C^{\text{MSE}}(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}(\mathbf{X}_i\boldsymbol{\beta} - y_i)^2 \tag{21}$$

$$C^{\text{L2}}(\boldsymbol{\beta}) = \sum_{i=0}^{n-1}(\mathbf{X}_i\boldsymbol{\beta} - y_i)^2. \tag{22}$$

Differentiating the $C^{\text{MSE}}$ using the chain rule gives

$$\frac{\partial C}{\partial \boldsymbol{\beta}} = \frac{2}{n}\sum_{i=0}^{n-1}(\mathbf{X}_i\boldsymbol{\beta} - y_i)\frac{\partial \mathbf{X}_i\boldsymbol{\beta}_j}{\partial \boldsymbol{\beta}_j} \tag{23}$$

$$= \frac{2}{n}\sum_{i=0}^{n-1}\mathbf{X}_i^T(\mathbf{X}\boldsymbol{\beta} - y_i) \tag{24}$$

$$\nabla C(\boldsymbol{\beta}) = \left[\frac{\partial C}{\partial \beta_0}\frac{\partial C}{\partial \beta_1}\cdots\frac{\partial C}{\partial \beta_{n-1}}\right] \tag{25}$$

$$= \frac{2}{n}\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}). \tag{26}$$

In ridge regression the MSE cost function and the gradient is

$$C^{\text{ridge}}(\boldsymbol{\beta}) = \frac{1}{n}\sum_{i=0}^{n-1}(\mathbf{X}_i\boldsymbol{\beta} - y_i)^2 + \lambda\sum_{j=0}^{n-1}(\boldsymbol{\beta})^2 \tag{27}$$

$$\frac{\partial C^{\text{ridge}}}{\partial \boldsymbol{\beta}} = \frac{2}{n}\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) + 2\lambda\boldsymbol{\beta}\mathbf{I}, \tag{28}$$

where $\mathbf{I}$ is the identity matrix.

To outline the process, imagine the cost function as a convex curve. The minimum represents the optimal parameter $\hat{\boldsymbol{\beta}}$. Picture yourself walking down this convex curve towards the minimum. Ideally, you want to be taking big steps when you are far from the goal and small steps when you are close. First, you try a set of parameters and insert them into the gradient. For each time the gradient is calculated, it is multiplied by a learning rate $\eta$. The learning rate can be constant or dependent on the iteration. The new and better parameter is calculated by

$$\boldsymbol{\beta}^{(n+1)} = \boldsymbol{\beta}^{(n)} - \eta g(\boldsymbol{\beta}^{(n)}). \tag{29}$$

Since the gradient is always pointing in the direction of greatest increment, we use the negative gradient to find the fastest path to the minimum.

Choosing the right learning rate is important. A too small learning rate can cause the process to not converge to a minimum within the specified number of iterations. A too large learning rate might make the process diverge, as we will see in the results of this study. Gradient descent, be it a powerful method to find a minimum of a function, can also be very computationally costly with a large dataset. There is also a risk that we might find a local minimum instead of the desired global minimum, making the process sensitive to initial conditions. In addition, gradient descent might get stuck in a saddle point. To overcome these issues, we use the stochastic gradient descent method, or rather a variant called mini-batch gradient descent.

### 2.5. *Stochastic gradient descent*

Stochastic gradient descent (SGD) circumvents the problems of gradient descent by calculating the gradient based on a randomly drawn data point instead of using the entire dataset. By drawing random numbers, the risk of stopping at a local minimum is lowered, and the computation time of a single data point is much lower than calculating the gradient on the entire dataset. The process of drawing a random data point and calculating the gradient is repeated until desired accuracy is achieved.

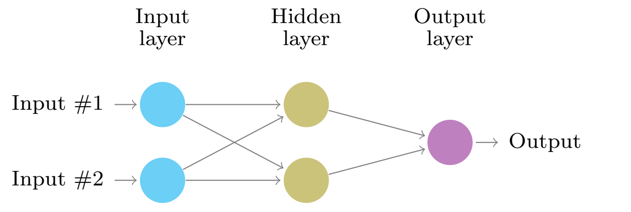


FIG. 1.— Architecture of a simple feed forward neural network. [a]

[a]The code for the tikz picture is a modified version found at https://texample.net/tikz/examples/neural-network/.

The data points may or may not be drawn with replacement. When all the data points have been drawn (without replacement), or a number of data points corresponding to the total number of data points have been drawn (with replacement) one says that an *epoch* has passed. To achive the desired accuracy, typically several epochs have to be performed.

## 2.6. *Mini-batch gradient descent*

Mini-batch gradient descent is a mix of GD and SGD, and it tries to mimic the accuracy of GD and the effectiveness of SGD. Instead of randomly choosing a single data point, a set of random data points are chosen from the full data set. This set of randomly drawn data points is called a *mini-batch*, and the gradient is calculated on all the data in the mini-batch. Then, a new mini-batch is drawn, and the gradient is calculated again. This process is repeated until desired accuracy is achieved.

## 2.7. *Feedforward neural network*

Neural networks provide many opportunities to predict outcomes from a set of input data, be it a problem of identifying images of hand-written digits, or fitting a curve. The general architecture of the network is fairly simple and can be outlined by the input layer at the start of the network, to the hidden layers at the core of the network, to the output layer at the end. The characteristic of a *feedforward* neural network is that the nodes in the network are not connected in a cycle, but only to the nodes in the succeeding layer. One is free to choose the depth of the network, that is the number of hidden layers, and the width of the network, which is the number of nodes per layer. To explain what happens in the network, lets look at a very simplified version with two inputs, one hidden layer with two nodes and one output layer, as seen in figure 1. The input nodes, $I_1$ and $I_2$, can both be connected to the two hidden nodes $H_1$ and $H_2$. By connection, we mean that the output from one node is sent to a next node. The output from the input nodes, which we denote $a_1$ and $a_2$, enter the hidden layer by being multiplied by weights $w_{1,1}$, $w_{1,2}$, $w_{2,1}$and $w_{2,2}$ and added biases $b_{1,1}$, $b_{1,2}$, $b_{2,1}$, $b_{2,2}$. Here $w_{i,j}$ is the weight multiplied by the output of $I_i$ as it enters $H_j$, and equivalently for $b_{i,j}$. It is also normal to use the same bias on both inputs to a node. In addition, the output from a node, multiplied by a weight and added a bias, is passed through an activation function, as we talked about in section 2.3. We then have the complete expression,

$$y_j = f\left(\sum_{i=1}^{n} w_{i,j}a_i + b_{i,j}\right), \tag{30}$$

where $y_j$ is the output of hidden node $j$, $f$ is an activation function, and $n$ is the number of nodes in the input layer which in this example is $n = 2$. As you may see, the system is not limited to having two nodes per layer, or a single hidden layer. We may implement an arbitrary number of hidden layers, and each layer may not have the same amount of nodes. The more hidden nodes and connections, the more advanced the output curve can be which may or may not improve the accuracy. The structure of the neural network strongly depends on what type of input data is being handled. To make the best possible output, all the weights and biases must be fine-tuned. This is done in a process called backpropagation.

## 2.8. *Backpropagation*

Backpropagation is the process where the weights and biases in the neural network are adjusted and fine-tuned. As the name implies, the weights and biases are updated backwards, layer by layer, going from the output layer and backwards through the hidden layers. Considering figure 1, we backpropagate in the opposite direction than that of the feedforward process. The biases and weights are optimized by using a variant of gradiant descent, as we discussed in sections 2.4, 2.5, and 2.6.

A single backpropagation starts by calculating the cost by the desired cost function,

$$\boldsymbol{\delta}_h = \text{cost}(\mathbf{y}_{\text{prediction}}, \mathbf{y}_{\text{actual}}). \tag{31}$$

To obtain $y_{\text{prediction}}$, we must first make a prediction by performing one feedforward with the weights and biases we already have. This means that the weights and biases must be assigned an initial value. It is common to initialize the biases with a small constant value, like 0.01, and to initialize the weights by drawing random numbers from the normal distribution. One often uses the standard normal distribution. When a cost is aquired, we can calculate the gradient of the weights and of the biases by

$$\nabla \mathbf{W}_h = \mathbf{X}^T \boldsymbol{\delta}_h, \tag{32}$$
$$\nabla \mathbf{b}_h = \sum \boldsymbol{\delta}_h. \tag{33}$$

In this backpropagation example, we assume a single hidden layer, where the subscript $h$ indicates the hidden layer.

## 2.9. *Accuracy*

For classification problems we use an accuracy score to study how precise the predictions are. It is defined as

$$\text{accuracy} = \frac{\sum_{i=1}^{n} I(y_{\text{actual},i} = y_{\text{prediction},i})}{n}, \tag{34}$$

where $I$ is the indicator function, $n$ is the number of predictions, $\mathbf{y}_{\text{prediction}}$ are the predictions made by the network and $\mathbf{y}_{\text{actual}}$ are the true outputs. Since the function is normalized, 0 is the worst result and 1 is the best.

## 2.10. *Learning rate*

There are several ways to decide the learning rate, $\eta$. It can be constant or it can be a function of time. One possible schedule is

$$\eta_j(t;t_0,t_1) = \frac{t_0}{t+t_1}, \tag{35}$$

$$t = em + i, \tag{36}$$

where $t_0$ and $t_1$ are two fixed values, $e = 1, 2, ...$ is the current epoch, $m$ is the amount of data in each mini-batch, and $i = 0, ..., m-1$. Another approach is the inbuilt 'invscaling' found in scikit-learns neural_network.MLPClassifier. This decreases the effective learning rate for each time step $t$ in the following manner,

$$\eta_e = \frac{\eta}{t^{\text{power\_t}}}, \tag{37}$$

where power_t is by default = 0.5.

## 3. METHOD

### 3.1. *Mini-batch gradient descent*

When performing the mini-batch gradient descent, we first calculate how much data goes into each mini-batch by dividing the total number of data points by the number of mini-batches. Any rest is discarded. We then generate an array of integer corresponding to the total number of data points in the training data set, meaning we get an array of indices. We then loop over the desired amount of epochs. For every epoch, we iterate over the total number of mini-batches. For every loop in the mini-batch iteration we draw $m$ random indices from the index array previously described, and use these to fetch $m$ random data points from the training set, where $m$ is the number of data points in each mini-batch. Then, we initiate a feedforward step using initial weights and biases, to generate a prediction. Next up, the backpropagation step uses the prediction from the feedforward step to update the weights and biases. The weights and biases are updated using gradient descent with the selection of data, the mini-batch. Now, we draw a new random mini-batch and continue the process where the previous mini-batch left off. This process continues until we have performed the desired amount of epochs. The implementation can be found in the file `neural_network.py`, class `_FFNN`, methods `train_neural_network`, `feedforward`, and `_backpropagation`.

We have also implemented stochastic descent without any connection to a neural network. In this case, the mini-batch sizes are set to 1, and the gradients are calculated by using the derivative of the mean squared error. A ridge addition is added, and the gradient is multiplied by the learning rate, and finally a momentum term is added. The process is repeated until the desired amount of epochs have been performed. The learning rate can either be constant, or using the schedule 35. The implementation can be found in the file `common.py`, class `_StatTools`, method `mini_batch_gradient_descent`.

We analyse the mean squared error (MSE) as a function of varying number of mini-batches and epochs, different learning rates as well as how the learning rate is calculated. We test both a constant learning rate and the time dependent, equation 35. For mini-batch gradient descent with ridge, we also assess how MSE varied with the penalty hyperparameter. We also compare our implementation of SGD with scikit-learn's `SGDRegressor`.

### 3.2. *FFNN and linear regression*

We have implemented a feed forward neural network with a flexible numbers of hidden layers and nodes per layer. See section 3.5 for details of the implementation. We have implemented the functionality to change the activation function for the hidden layers and the output layer, as well as the cost function. The different activation functions have been tested and compared. We used data from the Franke function with 1000 data points and analyzed the results using a grid search. We varied the regularization parameter, the step size, the number of mini-batches, the number of epochs, as well as using the logistic activation function, the ReLU and Leaky ReLU. The data was compared using MSE and $R^2$.

### 3.3. *FFNN and classification*

To do a classification of 1797 handwritten digits from `sklearn.datasets.load_digits` we used the `sklearn.neural\_network.MLPClassifier`. It uses the Softmax activation function for the output layer. We tried different numbers of hidden layers, batch sizes, learning rates, penalty parameters, and hidden layer activation functions. We also tried constant learning rates versus a learning rate that is updated with each time step, equation 35. The results was evaluated with an accuracy score, see section 2.9.

### 3.4. *Logistic regression*

Our logistic regression was implemented by using the classification feedforward neural network with zero hidden layers. Easy peasy. See section 3.5 for details of the implementation. We compare the logistic regression with the feedforward neural network classifier by considering the classification score as a function of epochs. We compare it with three neural network configurations: (50,), (50, 25), and (50, 25, 25), the numbers indicating the number of nodes in each hidden layer. We also compare the computation time of the single-layer feedforward neural network to the computation time of the logistic regressor.

In addition, we compare the logistic regressor to scikit-learn's feedforward neural network classifier with zero hidden layers scikit-learn developers (2020b). We configure the thwo implemetations with as similar parameters as possible and look at the classification score as a function of the number of epochs. We also do a comparison of the computation time as a function of the number of epochs.

### 3.5. *The program*

All the experiments are coded in Python. The code is available at `https://github.uio.no/jonkd/FYS-STK4155_H20/tree/master/projects/project2`.

The total project is split into several files. `common.py` incorporates most of the functionality which is common for the different tasks. For instance, it contains a class `Regression` which contain the linear regression code from project 1. The class `_StatTools` includes the mini-batch gradient descent code which is compared to the linear regression functionality in `Regression`. Also present in `common.py` are functions for calculating the mean squared error and the R score. One also finds code for setting up the design matrix, and the Franke function itself.

The file `neural_network.py` contains all the specifics of the feedforward neural network implementation. A base class `_FFNN` contains the common neural network functionality, like setting up the initial state of weights and biases, the feedforward algorithm, the backpropagation algorithm, the training method which includes the selection of mini-batches, and functionality for making a prediction. The class `FFNNClassifier` inherits from `_FFNN` and implements the classification functionality by setting up a slightly different initial state, and by implementing a different way of measuring the score. The class `FFNNRegressor` inherits from the class `_FFNN` and sets up a slightly different initial state. The class `LogisticRegressor` inherits from `FFNNClassifier`, since logistic regression in effect is classification by a neural network with zero hidden layers. It implements a different initial state and backpropagation algorithm to accomodate for the lack of hidden layers.

The file `activation_functions.py` contains all the different activation functions. They are split into a separate file as to not have too much code in `neural_network.py`

Further, all code specifics for solving the different subtasks are split into separate files named `task_x.py` for x = a, b, ..., e. Run the file with name corresponding to the desired task to generate the correct data.

Finally, `test.py` contains simple testing which compares a hard-coded single layer feedforward neural network (from Hjorth-Jensen (2020b)), `FFNNSingle`, with the multi-layer feedforward neural network, `_FFNN`, in `neural_network.py`. The testing performs a single feedforward and a single backpropagation, and checks that all the weight, biases, and pretty much every single value in every single vector in our implementation corresponds to the known good values from the referenced single-layer feedforward neural network. Run it with `pytest`.

## 3.6. *Data*

### 3.6.1. *Franke's function*

For the linear regression problem we use data obtained from Franke's function. Franke's function is a function of two variables which we choose to be a linear set of numbers from 0 to 1. We add normally distributed noise to the output of Franke's function. The data is gathered into a design matrix of a specified polynomial degree for the linear regression case. For the neural network, the data is structured as a matrix with two columns, where each column are the pair of inputs to Franke's function.

### 3.6.2. *Classification data*

For the classification and logistic regression case we use data from the UCI Machine Learning Repository Dua & Graff (2017), available through scikit-learn by `sklearn.datasets.load_digits()` which contains 1797 8x8 pixels images of handwritten digits.

## 4. **RESULTS AND DISCUSSION**

### 4.1. *Stochastic gradient descent*

We compare our own stochastic gradient descent method to scikit-learns `SGDRegressor`. The comparison is run for both ordinary least squares (OLS) and ridge

TABLE 1
The table shows the mean squared error for ordinary least squares with and without ridge regression using stochastic gradient descent. The calulations loop over number of epochs, learning rates and ridge parameters. The calculations found that no ridge regression yields the best results.

| | $MSE_{\text{OLS}} = 0.02$ | $MSE_{\text{ridge}} = 0.02$ |
|---|---|---|
| # epochs | 18 | 20 |
| Learning rate | 0.03 | 0.07 |
| $\lambda$ | - | 0 |

regression with 400 data points and 15 repetitions to the random data. We use a constant learning rate from $10^{-3}$ to 0.1 and penalty parameter ($\lambda$) from 0 to 0.1. For each combination, the predictions are evaluated by the mean squared error. The lowest MSE for OLS and ridge are found in table 4.1. As we can see from the table, ridge regression provides no significant improvement over regular OLS for the set of data we are considering. We do find the minimum MSE at a bit different number of epochs and learning rates, but we believe that is due to the statistical randomness in the shuffling of the data we analyse. The data is shuffled for every time we change a parameter, to start off fresh every time.

See figure 2 for a graphical representation of how the accuracy of the network varies as a function of the number of epochs. For our implementation of stochastic gradient descent the MSE seems to decrease with increasing numbers of epochs as one would expect. More epochs means that the calculation of the gradient will be done with greater precision, and hence a lower MSE. For the scikit-learn implementation, the MSE seems more or less constant for different epoch number after approximately 5 epochs, which might be due to some sort of tolerance or limit we have yet to discover. Though, the comparison with scikit-learn proves useful to assess that we have a working framework. We see a similar trend in figure 3 where the two implementations are shown with MSE as a function of learning rate, $\eta$. Here we use 20 epochs, and as we see in the plot, that is not enough iterations to land at the lowest MSE for the smallest learning rates. With inreasing learning rate, the systems converge on a lower MSE more quickly.

From table 4.1 we see that the system achieves the lowest MSE without any ridge regression. This is also seen in figure 4 where we force an increasing ridge regression penalty parameter, $\lambda$. Both our implementation and the implementation of scikit-learn have a strictly rising MSE for larger $\lambda$, indicating that ridge regression is not something to use with the Franke dataset.

### 4.2. *FFNN*

We implement different activation functions for the hidden layers of our FFNN, and calculate the MSE and $R^2$ for the Franke function data set. The results are analyzed using a grid search, where we represent the measure metric as the color / heat values and vary a pair of parameters.

For the logistic activation function, we varied $\eta$ between $10^{-2}$ to $10^{-1}$ and $\lambda$ between 0 to $10^{-3}$. As shown in figure 5, for the logistic activation function, the smallest MSE and largest $R^2$ are produced for $\eta = 6.4 \times 10^{-2}$ and $\lambda = 0$. The smallest observed value of MSE is 0.14 and the largest $R^2$ is 0.84. For ReLU and Leaky ReLU,

TABLE 2
THE MSE AND $R^2$ SCORES FOR THE FRANKE FUNCTION DATASETS USING THE FFNN WITH $N_{epoch} = 1000$, $M = 10$. DIFFERENT FUNCTIONS ARE IMPLEMENTED AS ACTIVATION FUNCTION FOR HIDDEN LAYERS

| Act. function | $\lambda$ | $\eta$ | MSE | $R^2$ |
|---|---|---|---|---|
| Sigmoid | 0 | $6.4 \times 10^{-2}$ | 0.14 | 0.84 |
| ReLU | 0 | $30.0 \times 10^{-2}$ | 0.18 | 0.80 |
| Leaky Relu | 0 | $24.2 \times 10^{-2}$ | 0.16 | 0.83 |

TABLE 3
THE MSE AND $R^2$ SCORES FOR THE FRANKE FUNCTION DATASETS USING THE FFNN WITH $N_{epoch} = 1000$, $M = 10$ AND SIGMOID AS ACTIVATION FUNCTION. WEIGHTS AND BIASES ARE INITIALIZED FOLLOWING DIFFERENT METHODS

| Act. function | $\lambda$ | $\eta$ | MSE | $R^2$ |
|---|---|---|---|---|
| Normal $w_{ij}$, $b_i = 0.01$ | 0 | $6.4 \times 10^{-2}$ | 0.14 | 0.84 |
| Normal $w_{ij}$, $b_i = 0.1$ | 0 | $2.8 \times 10^{-2}$ | 0.15 | 0.83 |
| $w_{ij} = 0.1$, $b_i = 0.01$ | 0 | $2.3 \times 10^{-2}$ | 0.23 | 0.79 |
| $w_{ij} = 0.1$, $b_i = 0.1$ | 0 | $2.3 \times 10^{-2}$ | 0.18 | 0.80 |

we varied $\eta$ between $10^{-5}$ to $3 \times 10^{-4}$ and $\lambda$ between 0 to $10^{-3}$. Comparison between different activation functions are given in table 2. With regularization parameter = 0, all activation functions produced best prediction for the test data. However, the corresponding optimal learning rate was different for the different activation functions. We experience that ReLU and leaky are very sensitive to learning rates, inparticular large learning rates, and quickly produce overflow. The logistic function on the other hand is much nicer to work with, and can handle a wide range of learning rates. This is due to the logistic function constraining the output to be no larger than 1 and no smaller than 0, while ReLU and leaky may pass through arbitrary large function values which can quickly accumulate in size and lead to overflows. The MSE and $R2$ values obtained by using different activation functions are very close. These values can be tweaked by increasing the number of epochs.

We follow different ways to initialize the weights and biases of our neural network and quantify their effects using MSE and R2. The results are summarized in table 3. The variation in MSE and $R^2$ for different initialization of weights and biases are not very significant, which did do not coincide with our anticipation. In our Franke function dataset, we include standard normally distributed random fluctuations, multiplied by a factor of 0.1. This random fluctuation makes it harder for the FFNN to make accurate predictions. The added random fluctuation in the data can dominate the error over the initialization of weights and biases, but at smaller fluctuations we experience that the FFNN can produce precise predictions with normally distributed weights and constant initial biases of 0.01.

### 4.3. *FFNN and linear regression*

From the implemetation of our own neural network with 3 hidden layers of 50, 25 and 25 nodes respectively, 1000 epochs and mini-batch size of 50, we obtained a mean squared and R$^2$ as seen in table 4. Here we also see results form linear regression with bootstrapping with OLS and ridge penalty parameter $\lambda = 0.001$. When taking the ratio of MSE and R$^2$ from the neural network and linear regression see that linear regression has a lower error than neural network in all cases except for R$^2$ of training data, see table 5. The MSE for FFNN was several folds higher than the regression methods for both test and train data. This shows that greater complexity does not necessarily mean greater predictive power.

TABLE 4
MEAN SQUARED ERROR AND R$^2$ FOR LINEAR REGRESSION WITH 50 BOOTSTRAP RESAMPLES FOR OLS AND RIDGE (WITH PENALTY PARAMETER $\lambda = 0.001$) AND FOR A NEURAL NETWORK WITH 3 HIDDEN LAYERS SIZES WITH 50, 25 AND 25 NODES, EPOCHS=1000 AND MINI-BATCH SIZE OF 50.

| Method | Lin. reg, OLS | Lin. reg, ridge | Neural network |
|---|---|---|---|
| MSE train | 0.01288 | 0.01413 | 0.1093 |
| MSE test | 0.01128 | 0.01278 | 0.1351 |
| R$^2$ train | 0.8734 | 0.8610 | 0.8907 |
| R$^2$ test | 0.8799 | 0.8640 | 0.8663 |

TABLE 5
COMPARISON OF MSE AND R$^2$SCORES FOR LINEAR REGRESSION WITH BOOTSTRAP (LRB) AND NEURAL NETWORK (NN).

| | |
|---|---|
| MSE train nn/lrb | 8.4841 |
| MSE test nn/lrb | 11.9787 |
| R$^2$ train nn/lrb | 0.8627 |
| R$^2$ test nn/lrb | 1.1134 |

TABLE 6
ACCURACY SCORE OF SCIKIT-LEARN NEURAL NETWORK

| | Accuracy score=0.9778 |
|---|---|
| # of hidden layers | 1 |
| # of nodes | 9 |
| Learning rate | 0.01437 |
| $\lambda$ | 0.0005995 |
| Activation function | Identity |

### 4.4. *FFNN and classification*

With `sklearn.neural_network.MLPClassifier` a sample of 1797 handwritten numbers was classified to an accuracy score of 0.9778. The highest score was obtained from a combination of parameters found in table 6.

How the accuracy varies with the different parameters is shown in figure 7, 8 and 9. In figure 7 we see the regularization parameter, $\lambda$, and learning rate, $\eta$, varying while we use a single hidden layer of 50 nodes, a mini-batch size of 20, and 50 epochs. The logistic function is used as hidden layer activation function, while the output is linear. The system favors a low learning rate of about $\eta = 0.02$ while the regularization parameter gives a fairly constant accuracy at this learning rate. Though, we do see that the system favors a larger regularization of approximately $\lambda = 10^{-4}$ for the other learning rates.

In figure 8 we vary the number of hidden layers and the number of nodes in each layer. We use an equal number of nodes in each hidden layer with the logistic function as activation, and linear for the output. A batch size of 20 is used, with 50 epochs. We find the maximum score at two hidden layers, though it is only a tiny bit larger than at a single layer. For all numbers of hidden layers we see no significant connection between score and the number of nodes per layer, in the range of 10 to 170 hidden layer nodes. One must then conclude that it is wiser to use a smaller number of nodes due to the reduce in computation time.

Consider now figure 9, where we vary the hidden layer activation functions, along with the number of hidden layers. Due to the differing behaviour of the three activation functions, we must use different learning rates to make them behave properly. We use $\eta = 0.02, 0.001, 0.001$ for logistic, tanh, and ReLU, respectively. ReLU in particular, quickly leads to overflows if we are not careful with choosing a sufficiently small learning rate. The logistic function delivers good results for all hidden layer configurations, but both tanh and ReLU has a dramatic drop in accuracy with increasing number of layers. With more layers, more hidden nodes need activation, which means that an already difficultly behaving function like ReLU get more room to be naughty. The trend for all functions is to use a single hidden layer, which is fortunate for us with regards to computation time.

### 4.5. *FFNN and logistic regression classification*

Since the logistic regression implementation in effect is the same as the neural network, but with zero hidden layers, we expect them to behave similarly. For an equal amount of epochs, logistic regression should have a shorter computation time than the neural network due to the difference in hidden layers, but without running a simulation it is not apparent which one of the methods will have greater accuracy. One can speculate that the greater complexity of the neural network with one or several hidden layers will yield better results, as we saw in figure 8, but does greater complexity always yield better results?

Consider figure 10, where logistic regression is compared to the neural network. All implementations use mini-batch gradient descent with a mini-batch size of 20 and a learning rate of $\eta = 4.8 \cdot 10^{-3}$, and the data is averaged over 30 runs. The neural network is plotted with three different hidden layer configurations, of 1, 2, and 3 hidden layers, with number of nodes in the plot legend. Considering the given parameters, the implementations are on equal footing except for the nodes in the hidden layers of the neural networks. For a small number of epochs, the logistic regression performs a bit better than the neural networks, up to approximately 40 epochs. From 50 epochs and above, the neural network with a single hidden layer yields a bit better results, save for at a few points where the neural network single layer score dips below the score of the logistic regression, which we believe is due to statistical fluctuations. Although the neural network does perform better, the increase in score is not large, maximum 0.5% better than logistic regression, and that is only at fairly large amounts of epochs.

We see from this comparison that increasing the complexity of the neural network, increasing the number of hidden layers, is not beneficial for this type of calculation. By adding a second and a third hidden layer, the score gets consistently lower for all numbers of epochs, in addition to increase the computational cost and thus computational time.

In figure 11 we see the classification score as a function of computation time. It is apparent that the neural network is much more computationally heavy than the logistic regression, needing far longer computation times to reach the same scores. As indicated in the title of the plot, the max classification score for the two implementations are equal, but logistic regression uses over 1s less computation time to reach the same score. This may not seem like a lot, but the handwritten numbers data set from UCI Dua & Graff (2017), is not particularly large, and real-life applications can easily exceed this amount of data by many orders of magnitude. Adding more hidden layers to the neural network will only increase the computation time further, in addition to lowering the score, as we saw in figure 10.

We thus conclude that for classification problems logistic regression seems to be the better alternative, due to the equal score to the neural network, and in particular due to the lower computational time.

### 4.6. *Logistic regression and scikit-learn*

We now compare our logistic regression implementation to the equivalent tool in scikit-learn scikit-learn developers (2020b). The scikit-learn implementation is also a zero hidden layer neural network classifier. Both implementations use the logistic function as activation function, mini-batch gradient descent with a batch-size of 20, and no l2 penalty. Our logistic regression implementation use a learning rate of $\eta = 4.8 \cdot 10^{-3}$, and scikit-learn uses $\eta = 0.21$. These learning rates are decided by varying the learning rate of both implementation and finding what learning rate yields the greatest classification score. The data is averaged over 30 runs.

Consider figure 12 where we see the classification score as a function of the number of epochs. The scikit-learn implementation starts at a high score, at only 10 epochs, and has a small rise in score to approximately 80 epochs. For the subsequent number of epochs, the score doesnt alter much, indicating that the scikit-learn implementation quickly yields stable results.

Our implementation starts out at a much worse score, at 10 epochs. It quickly rises, and at 100 epochs and above, it consistently has a larger score than scikit-learn. Now, bear in mind that to perform this comparison, we have forced both implementations to equal footing in regards of, including but not limited to, batch size and type of solver. The scikit-learn implementation has a lot of other parameters we can adjust to increase the score, in our tests up to $\approx 0.98$. But on equal footing, it seems that our implementation is able to squeeze out approximately half a percent better classification score. The exact reason as to why our score is larger at approximately the same parameters is difficult to establish, since these implementations are fairly complex, and demands a study of its own to pinpoint. For now, we only observe and describe the differences.

Now, I'm sure that the scikit-learn developer team would have us join them until we consider figure 13. Our implementation has a more or less linear rise in computation time as a function of epochs, while the scikit-learn implementation caps at approximately 0.8 seconds. Recognizing computation time as a very important resource in machine learning, scikit-learn has a clear advantage in reaching a good score quickly. A subject for a future study is to implement these calculations in a low-level language, like C and Fortran, and try to tweak the performance further.

## 5. CONCLUSION

We have studied the SGD and NN for Franke function and MNIST data set using our code and Scikit-Learn. Evaluation of the MSE on the Franke data indicates no advantage of using the ridge penalty parameter when the self-produced SGD replaced the matrix inversion algorithm. In comparison with scikit-learn, our own code produced an improved result in most respects, save for the computational time. The consistency of MSE for Scikit-Learn with the number of epochs is still an open question to us. The activation function assessment for hidden layers was carried out for logistic, ReLU, and Leaky ReLU. Even though the logistic function was better with the least MSE of 0.14, the result did not facilitate us with a clear choice. The logistic function proved to be more user friendly, while ReLU and leaky ReLU were quite picky with the learning rate. Alltough, for correct learning rate, ReLU proved to perform well with few hidden layers in the neural network. Most surprisingly, the initialization of weights and biases did not influence the statistical parameters significantly. The value of $R^2$ stayed within 5% for various modes of initialization for the weights and biases.

The comparative study between the logistic regression and neural network produced close results. With a smaller number of epochs (up to 40), logistic regression performed better than the neural network. Beyond 50 epochs, the neural network did slightly better. Thus a complex model can not always ensure improved performance, but it will always causes a longer computation time. As we benchmarked our logistic regression code against Scikit-Learn, it exposed our code's underperformance at a smaller number of epochs; however, it suppressed the Scikit-Learn at a larger number of epochs. The neural network has the advantage of reaching higher scores, but at the cost of computational time, while logistic regression is comparatively lightweight.

All code used to generate data for this report is available at the GitHub repository https://github.uio.no/jonkd/FYS-STK4155_H20/tree/master/projects/project2.

## REFERENCES

Dua, D., & Graff, C. 2017, UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

Gupta, D. 2020, Fundamentals of Deep Learning – Activation Functions and When to Use Them?, https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/

Hjorth-Jensen, M. 2020a, Overview of course material: Data Analysis and Machine Learning. https://compphysics.github.io/MachineLearning/doc/web/course.html

—. 2020b, Week 41 Tensor flow and Deep Learning, Convolutional Neural Networks. https://compphysics.github.io/MachineLearning/doc/pub/week41/html/week41.html

McCulloch, W. S., & Pitts, W. 1943, The bulletin of mathematical biophysics, 5, 115, doi: 10.1007/BF02478259

Sanderson, G. 2015, 3Blue1Brown. https://www.youtube.com/c/3blue1brown/about

scikit-learn developers. 2020a, sklearn.linear_model.SGDRegressor, https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html

—. 2020b, sklearn.neural_network.MLPClassifier, https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

Starmer, J. 2011, StatQuest. https://www.youtube.com/c/joshstarmer/about

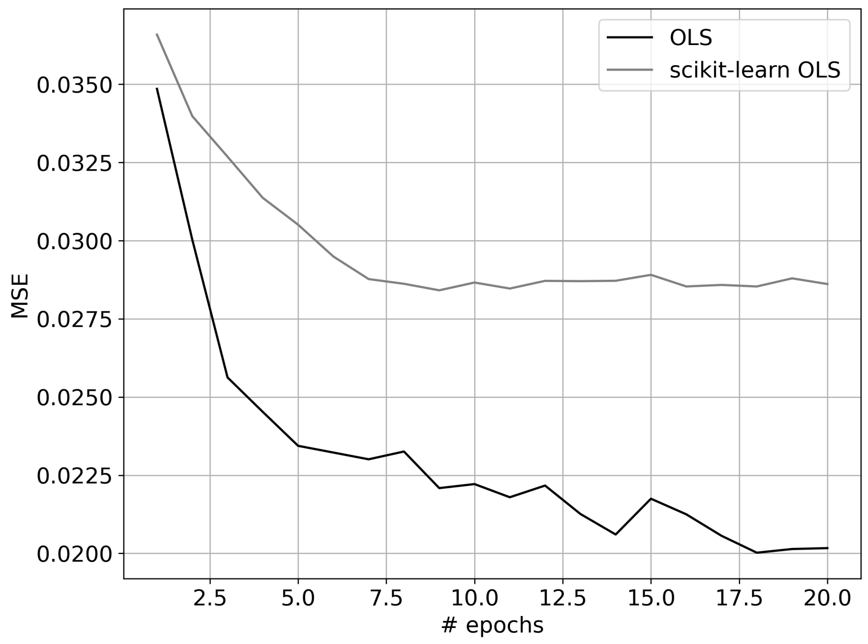


FIG. 2.— Stochastic gradient descent with our implemetation and for the scikit-learn SGDRegressor. The MSE as a function of the numbers of epochs, while the other parameters are set to the optimal values decided by our grid search.

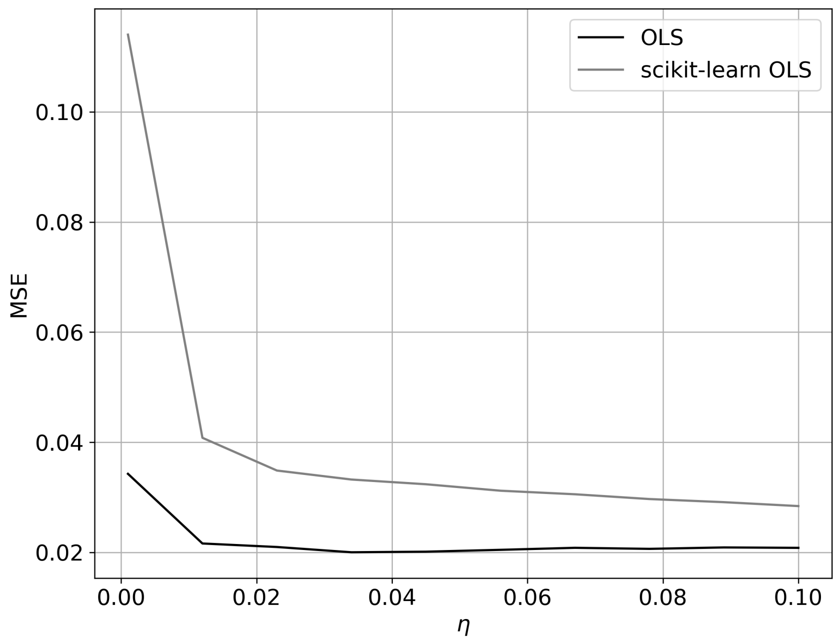


FIG. 3.— Stochastic gradient descent with our implemetation and for the scikit-learn SGDRegressor method. The MSE as a function of the learning rate $\eta$, while the other parameters are set to the optimal values decided by our grid search.

## APPENDIX

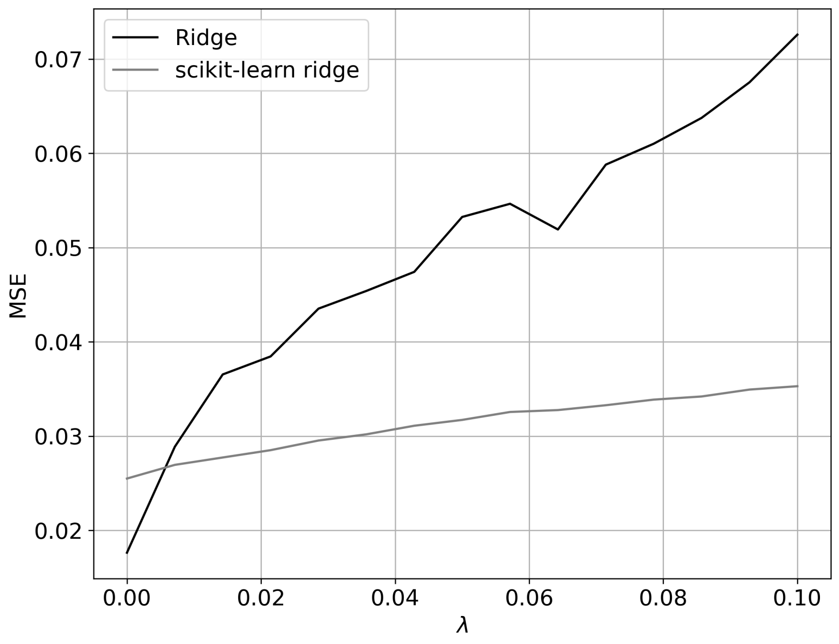
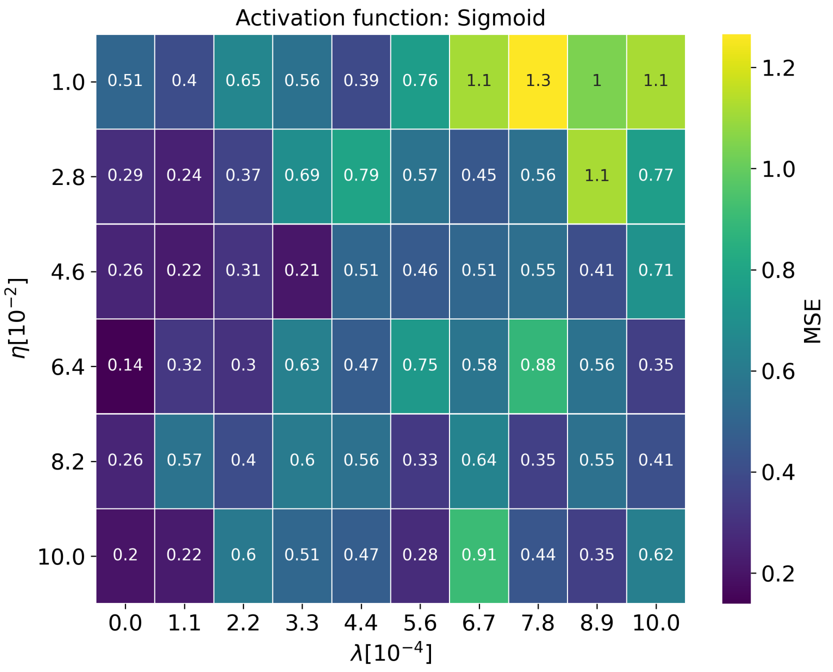


FIG. 4.— Stochastic gradient descent with our own implemetation and for the scikit-learn SGDRegressor method. The MSE as a function of the ridge penalty parameter, while the other parameters are set to the optimal values decided by our grid search.



| $\eta[10^{-2}]$ \ $\lambda[10^{-4}]$ | 0.0 | 1.1 | 2.2 | 3.3 | 4.4 | 5.6 | 6.7 | 7.8 | 8.9 | 10.0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.0 | 0.51 | 0.4 | 0.65 | 0.56 | 0.39 | 0.76 | 1.1 | 1.3 | 1 | 1.1 |
| 2.8 | 0.29 | 0.24 | 0.37 | 0.69 | 0.79 | 0.57 | 0.45 | 0.56 | 1.1 | 0.77 |
| 4.6 | 0.26 | 0.22 | 0.31 | 0.21 | 0.51 | 0.46 | 0.51 | 0.55 | 0.41 | 0.71 |
| 6.4 | 0.14 | 0.32 | 0.3 | 0.63 | 0.47 | 0.75 | 0.58 | 0.88 | 0.56 | 0.35 |
| 8.2 | 0.26 | 0.57 | 0.4 | 0.6 | 0.56 | 0.33 | 0.64 | 0.35 | 0.55 | 0.41 |
| 10.0 | 0.2 | 0.22 | 0.6 | 0.51 | 0.47 | 0.28 | 0.91 | 0.44 | 0.35 | 0.62 |

FIG. 5.— The MSE for regression of Franke function data for a range of learning rate $\eta$ and regularization parameter $\lambda$ with Sigmoid activation function.

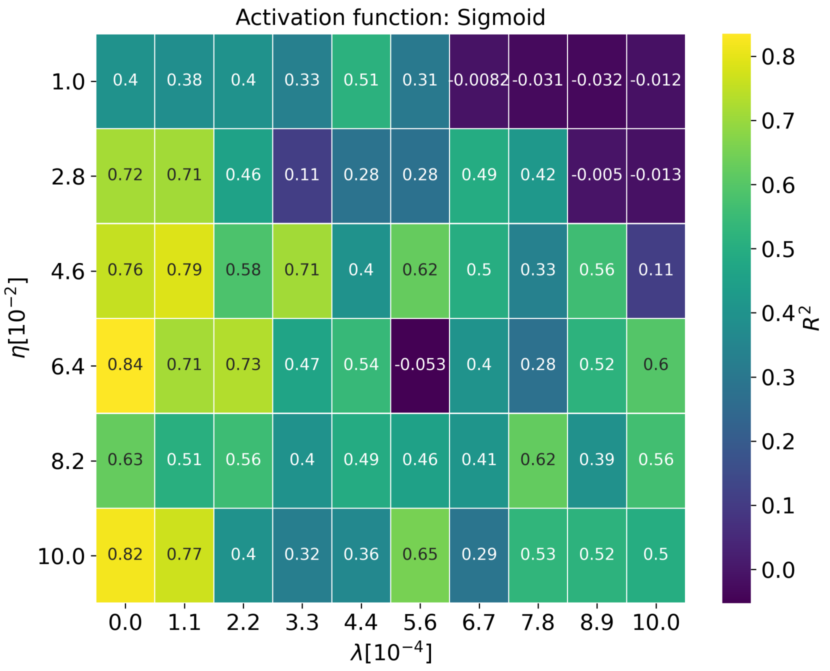


FIG. 6.— The $R^2$ for regression of Franke function data for a range of learning rate $\eta$ and regularization parameter $\lambda$ with Sigmoid activation function.

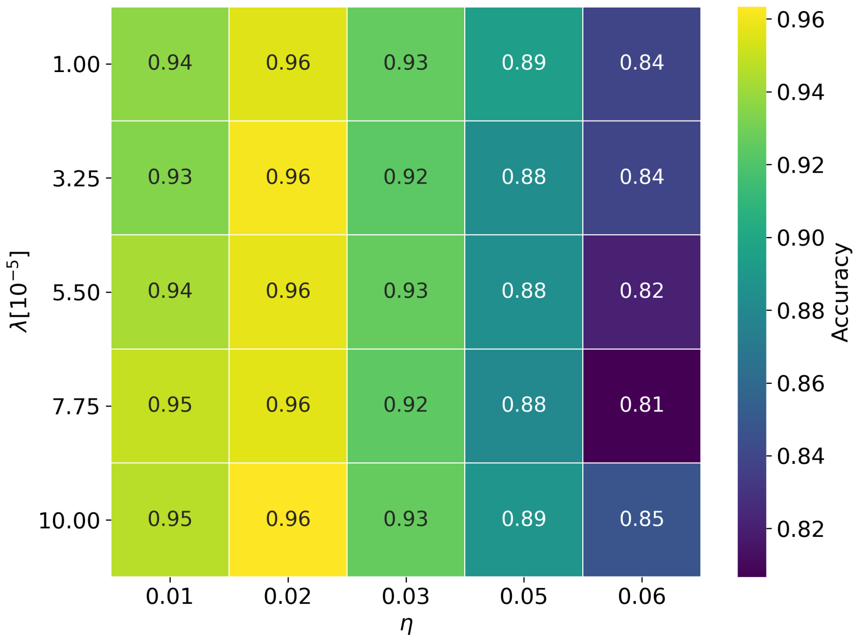


FIG. 7.— Heatmap of classification accuracy as a function of regularization parameters, $\lambda$, and learning rates, $\eta$. The hidden layer activation function is the logistic function. The neural network has a single hidden layer of 50 nodes, a batch size of 20, and 50 epochs.

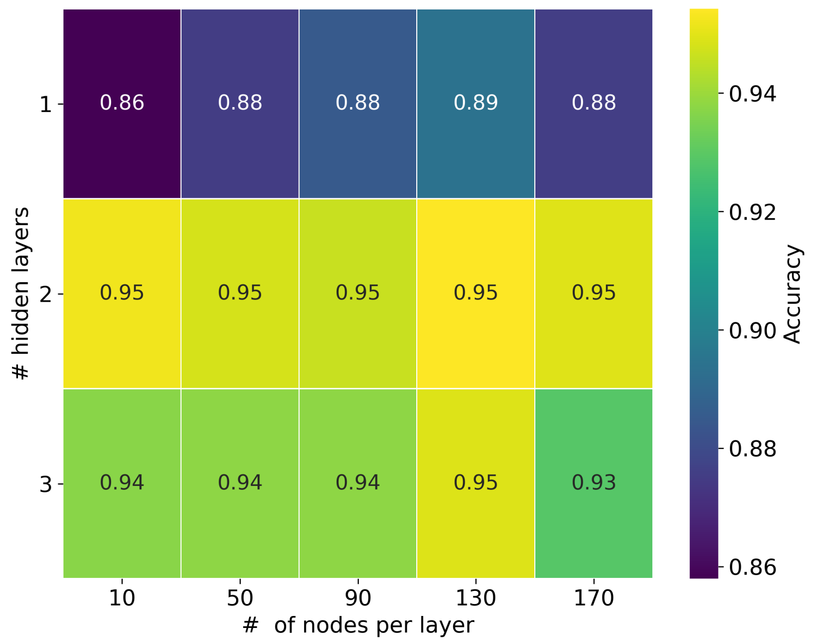


Fig. 8.— Heatmap of classifiaction accuracy as a function of the number of hidden layers and the number of nodes in each hidden layer. Here we use an equal number of nodes in every hidden layer. The hidden layer activation function is the logistic function. We use a batch size of 20 and 50 epochs.

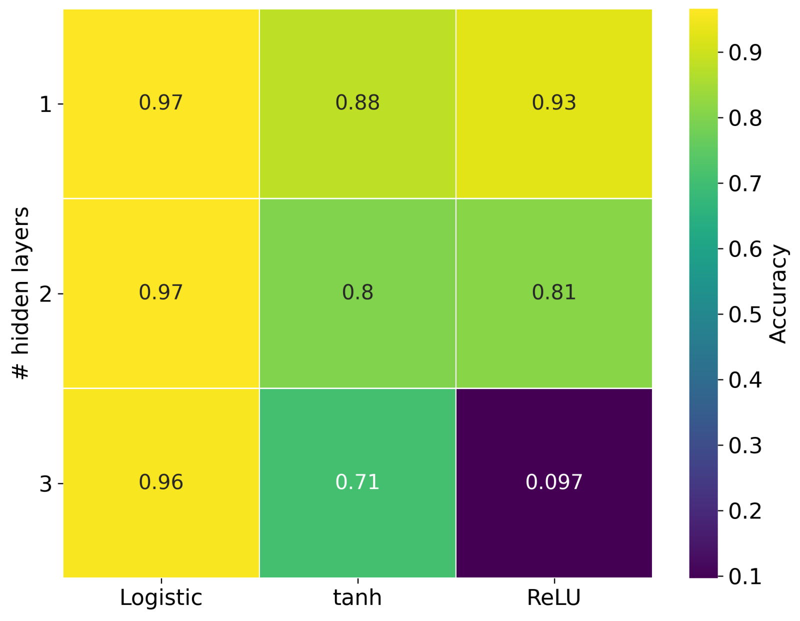


Fig. 9.— Heatmap of classification accuracy as a function of hidden layer activation functions and the numer of hidden layers. We use 50 nodes per hidden layer. The learning rates used are 0.02, 0.001 and 0.001 for logistic, tanh and ReLU respectively. We use 50 epochs and a batch size of 20.

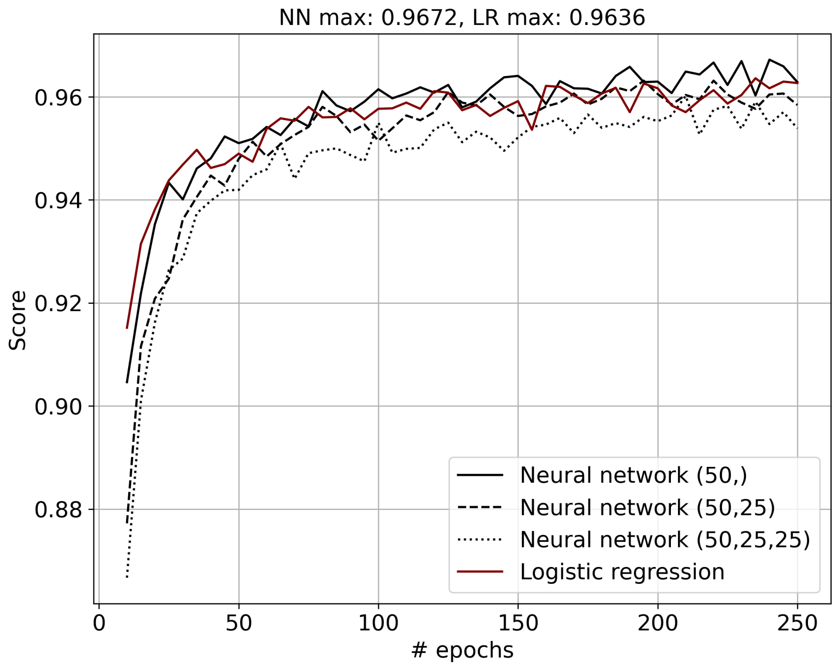


Fig. 10.— Classification score as a function of number of epochs for logistic regression and neural network on the data set of handwritten digits from UCI Dua & Graff (2017). Both use a mini-batch size of 20 and a learning rate of $\eta = 4.8 \cdot 10^{-3}$, and the data is averaged over 30 runs. The neural network is plotted for three different hidden layer configurations: 1st layer of 50 nodes; 1st layer of 50 nodes, 2nd layer of 25 nodes; 1st layer of 50 nodes, 2nd layer of 25 nodes, 3rd layer of 25 nodes. The hidden layer activation function is the Sigmoid function, and the output activation is softmax.

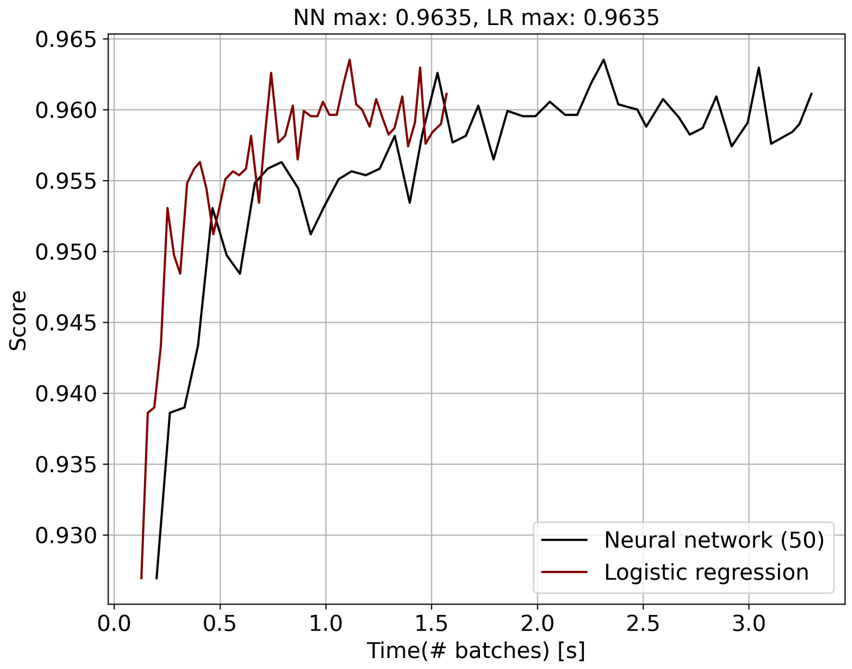


Fig. 11.— Classification score as a function of computation time for logistic regression and neural network on the data set of handwritten digits from UCI Dua & Graff (2017). Both use a mini-batch size of 20 and a learning rate of $\eta = 4.8 \cdot 10^{-3}$, and the data is averaged over 30 runs. The neural network has a single layer with 50 nodes. The computation time is a function of the number of epochs, as seen in figure 10. The hidden layer activation function is the Sigmoid function, and the output activation is softmax.

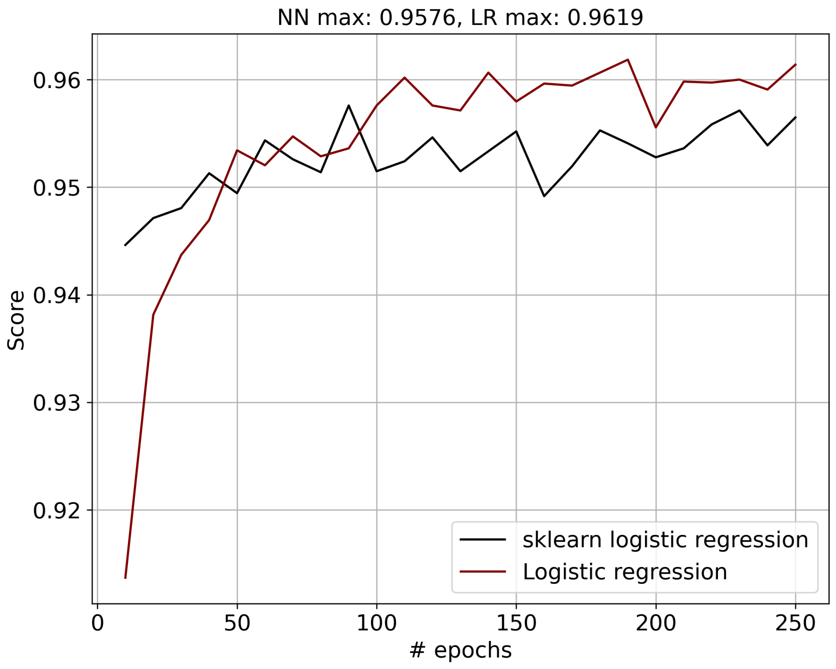


Fig. 12.— Classification score as a function of number of epochs for logistic regression and sckikit learn logistic regression scikit-learn developers (2020b) on the data set of handwritten digits from UCI Dua & Graff (2017). Both use a mini-batch size of 20 and the data is averaged over 30 runs. Our logistic regression implementation use a learning rate of $\eta = 4.8 \cdot 10^{-3}$, and scikit-learn uses $\eta = 0.21$.

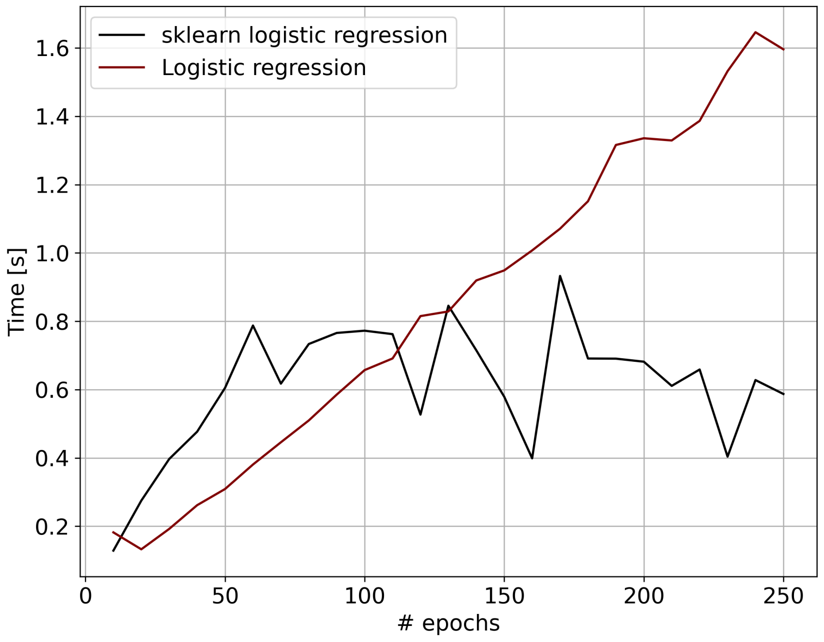


Fig. 13.— Here we see computation time as a function of the number of epochs. Our logistic regression implemetation has a clearly linear rise in computation time, while scikit-learn stabilises at 50 epochs and on.