Data Analysis and Machine Learning, Project 2

Francesco Pogliano

November 2019

Department of Physics, University of Oslo, Norway

Abstract

In this report I explore gradient descent, stochastic gradient descent with the use of varying step length and random minibatches and apply it to both logistic regression (classification problems) and neural networks (both classification and continuous regression). A focus has been given to the Credit Card dataset as analysed by Yeh et al. in [1] as a classification benchmark, and some stress has been given on the nature of the dataset, its parameters and undefined values.
Both the logistic regression techniques and the neural network scripts here developed reproduce the results from [1], and they do an exceptional job also in modelling the Franke function analysed in project one [3], outperforming linear regressions with OLS, Ridge and LASSO. Although being better, neural networks are because of the unlimited freedom in choosing the right architecture when it comes to number of layers, number of nodes for each layer, the choice of hyperparameters and activation function between the hidden layers. This has also to be weighted against the potential pitfalls of numerical instability.

Contents

1 Introduction

Whether you are scrolling your Facebook feed, trying to choose which music to listen to on Spotify or browsing your recommended videos on YouTube, you are in contact with machine learning algorithms. A part for recommending music or picking which advertisements to show on your phone, machine learning algorithms may be used to find out whether a person is likely to develop a certain health issue given their background, or whether a bank client is likely to default on his credit card debt given his financial history. These last two problems are classification tasks, where a model is built around a certain dataset containing many earlier results, and try to predict in base of these the likelihood for a new patient to develop the health issue, or a new client to default on their debt.
Classification does not necessarily have to be carried out by neural network algorithms, though. Logistic regression is the first step to understand how this kind of tasks work, and here we try to find the best parameters that give us the best likelihood for the past result to occur from the given training dataset, and use these parameters to predict new cases. In order to achieve such result we need to find the lowest point for our cost function, and this has to be done by the gradient descent method family, where we use the gradient of the cost function in order to reach the global minima of the function.
Neural networks work in a similar way, as we also have a cost function that have to be minimized by using gradient decent methods. In this project we will see how a Multilayer Perceptron (a kind of neural network) may be built and used for both classification problems and regression problems similar to the ones done in the previous project, and how it can outperform both of them.

2 Formalism

This project will continue the study on regressions by taking into account concepts like logistic regression and gradient descent first, and then the more complex topic of neural networks. In this section I will introduce the main theoretical points to be discussed later in the report, and is mostly based on the lecture notes on Logistic regression, Gradient descent and Neural networks by Morten Hjorth-Jensen [2].

2.1 Classification methods

As linear regression is more useful for finding the best fit for a continuous function, when it comes to classification problems we are more interested yes-no, (thus discrete) answers. This is the aim of classification algorithms, which may take as input a database, might it be the background of earlier hospital patients, or of past credit card users, (as we will see later) and give a discrete prediction for a given problem, like whether future patients will develop complications, or if new credit card users will default on their debt.

As we will see, there are different ways to solve a classification problem, where the first stepping stone in that sense is logistic regression.

2.1.1 Logistic regression

Logistic regression is the simplest algorithm for classification, and it is based on the sigmoid (or *logistic*) function,

$$sigmoid(x) = \frac{1}{1+e^{-x}}. \tag{1}$$

This function takes as input all the real numbers, and returns a number between 0 and 1. This is ideal when trying to evaluate the probability for a certain event to occur, and when predicting, a threshold at 0.5 might be given, for which all the outputs bigger than 0.5 will be assigned to 1, and the ones smaller, assigned to 0. This will be a hard classification, while a soft classification will retain the original, continuous output and interpret it as a probability.

When interpreting the result of the sigmoid function as the probability for a certain event to occur (p), we see that it has some useful properties. If our job were to classify between two classes (like yes/no, true/false and so on), the sum of the probabilities for the two classes will give one, meaning that we can express the second probability as $1-p$, and

$$\begin{aligned} p &= sigmoid(x) = \frac{1}{1+e^{-x}} \\ 1-p &= 1-sigmoid(x) = 1-\frac{1}{1+e^{-x}} = \frac{e^{-x}}{1+e^{-x}} = \frac{1}{1+e^{x}} = sigmoid(-x). \end{aligned} \tag{2}$$

Our aim when trying to fit a model to a database, will be to maximize the likelihood for the model to give the same result as for the training data, by tweaking the coefficients. The maximum likelihood is given as

$$P(\mathcal{D}|\boldsymbol{\beta}) = \prod_{i=1}^{n} \left[p(y_i = 1|x_i, \boldsymbol{\beta})\right]^{y_i} \left[1 - p(y_i = 1|x_i, \boldsymbol{\beta})\right]^{1-y_i}. \tag{3}$$

This is the usual likelihood distribution for the result of a binary experiment with probability p of success. For example, when throwing a coin y_i with probability for head $p = 0.5$ n times, the equation will give a likelihood distribution P telling us which result will be most likely to occur. We will for example have a very small probability for all coins ending up with head (and, respectively, for tail), while the probability is highest where the number of heads and tails even out. In the case

where we don't know the probabilities p and $1-p$, we will try to find a model that give us a P that most resembles the training data. As p depends on β, our aim is to find the βs that maximize P.

The logarithmic version of equation (3) will have a maximum in the same place, and is

$$\mathcal{P}_{\log}(\boldsymbol{\beta}) = \sum_{i=1}^{n} \left(y_i \log p(y_i = 1|x_i, \boldsymbol{\beta}) + (1 - y_i) \log \left[1 - p(y_i = 1|x_i, \boldsymbol{\beta})\right]\right), \tag{4}$$

and this might be thought as an inverse cost function. Since we are more interested in finding the minimum of a function, our cost function will be, (after reordering the logarithms,)

$$\mathcal{C}(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \left(y_i(\beta_0 + \beta_1 x_i) - \log\left(1 + \exp\left(\beta_0 + \beta_1 x_i\right)\right)\right). \tag{5}$$

This cost function is also known as *cross entropy*, and our job is to find its minimum with respect of the parameters $\boldsymbol{\beta}$. This will be done by finding its derivatives,

$$\frac{\partial \mathcal{C}(\hat{\beta})}{\partial \hat{\beta}} = -\hat{X}^T \left(\hat{y} - \hat{p}\right), \tag{6}$$

and using for example a gradient descent method to find the minima.

2.1.2 Gradient descent

Once we have a cost function $\mathcal{C}(\boldsymbol{\beta})$, finding a minima is an optimization exercise, and it can be summed up conceptually in finding the direction where the cost function decreases the most, for then taking a (small) step in that direction, and repeating the above until a minima is met.
Mathematically we know that the steepest ascent of a function $f(\boldsymbol{x})$ is given by its gradient $\nabla f(\boldsymbol{x})$. A way to find the steepest descent is then to move in the opposite direction: $-\nabla f(\boldsymbol{x}$. If we were in the coordinates $\boldsymbol{\beta}_k$ and moved in the direction where the cost function decreases the most with a step η, we would end up in:

$$\boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k - \eta \nabla \mathcal{C}(\boldsymbol{\beta}_k). \tag{7}$$

An algorithm implementing this regression method will then repeat the above step until a certain number of iterations is done. The loop might also be stopped by evaluating the cost function for each iteration, and registering when a lowest point is met.
Gradient descent (GD) might work fine for convex cost functions where there is only one minimum, but in other more rugged landscapes we encounter the problem of getting stuck in a local minimum. The algorithm is also very sensible to the choice of starting point of our search and the step length, other than being computationally expensive for many-dimensional landscapes. Some of these problems might be addressed by adding random elements to it, as we will see next.

2.1.3 Stochastic gradient descent

Some of the major problems of the GD might be the computational expense in having to evaluate the derivative of the cost function for each step, or the danger of getting stuck in a local minimum. These two might be addressed by evaluating the gradients for only a random set of database points. Such a random set is called a minibatch, and by introducing this step both makes the landscape different for each iteration (and thus diminish the danger of getting stuck in a local minimum) and makes the computation lighter by reducing the number of points to be fitted. Such a method is called Stochastic gradient decent (SGD).
If we were to divide our dataset in M minibatches m_j, where $0 \leq j < M$, $j = 0, 1, 2...$, then for each loop we would evaluate a randomly picked minibatch j. After having looped through M random minibatches (so not necessarily ordered) we say we have looped through one *epoch*.

Another way to improve the GD is to adjust the learning rate η to depend on how far in the algorithm we have come. Ideally, we would like to take big steps in the beginning and minimize the steps the closer we get to a minima, in order to avoid zig-zagging around the bottom of our landscape without ever reaching it because we are taking too large steps. This is reached by making η inversely proportional to the amount of iterations, such that:

$$\eta_t = \gamma_t \eta_0, \tag{8}$$

$$\gamma_t = \frac{t_0}{t + t_1}, \tag{9}$$

$$t = e \cdot M + i, \tag{10}$$

where e is the number of the current epoch, M is the total number of minibatches, while $i = 0, 1, 2...M - 1$.

2.2 Neural Networks

An artificial neural network is a computing system inspired by biological neural networks, which is able to "learn" how to do a task without being programmed specifically to perform such task beforehand. For example, a neural network may be taught to recognize patterns such as digits or images, or which song or series to recommend next based on your listening or watching habits or history.
there are many kinds of neural networks (NN), such as Feed Forward Neural Networks (FFNN) where information only flows in one direction, and Recurrent Neural Networks, where neurons may create computational cycles. Among the FFNNs we have also Convolutional Neural Networks, these exploit the spatial correlation of the inputs, making them ideal when it comes to image recognition. In this project we will focus on FFNNs, more specifically Multilayer Perceptrons (MLPs).
The basic structure of a MLP can be described in this way: The information from a dataset is sent to the first layer of the NN, this is processed by the layer's nodes according to their associated weights and biases, and then sent to the output layer again processed by the output weights and biases. There might be only one hidden layer, or more, each with an arbitrary number of nodes. The result is then compared to the target, and the error in the prediction will tell us how good or bad the weights and biases were, and by doing that it gives us the information in order to build a cost function, which can be used in the *backpropagation* algorithm in order to update the weights and better the prediction for the next round.
Neural networks can be used both for classification and regression tasks, as we will see in these report.

2.2.1 The node

The node is the building block of a neural network, and its task in a MLP is to receive information from all the nodes in the previous layer and make it ready to be read by the next layer (or, if the node is in the output layer, make it ready to be shown). This process is done by scaling each input from the previous nodes a_i^{l-1} by a node-specific weight w_{ij} and a bias b_j:

$$z_j = \sum_{i=0}^{N-1} w_{ij} a_i^{l-1} + b_j, \tag{11}$$

where N is the number of nodes in the $l-1$ layer (or, equivalently, the number of lines connecting to the j-th node of layer l from behind). Now that the node j in level l has collected and weighted all the inputs from the previous layer, the result z_j has to be compressed and made ready to be

read by the next layer. This is done by the activation function f, which usually compresses to a value between *0* and *1*:

$$a_j = f(z_j). \tag{12}$$

A natural candidate is the previously discussed *sigmoid*, but an activation function might be whichever function that increases monotonically, is continuous, non-constant and bounded. Other examples of activation functions are for example $f(x) = \tanh x$, the Heaviside step function or the ReLU function, defined as:

$$ReLU(x) = \begin{cases} 0 & x < 0, \\ x & x \geq 0. \end{cases} \tag{13}$$

This last one is faster to compute, but is not bound from above, meaning that it might give overflow problems and numerical instability, and that it might shut down some nodes when the values go to zero.

2.2.2 The layers

The MLP is a Feed Forward Neural Network, meaning that information is sent forward from the input to the output through the in-between layers. The first hidden layer takes the design matrix as input, so that the output of that layer will be, in vector notation:

$$\boldsymbol{a_h} = f\left(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}\right). \tag{14}$$

When we arrive at the last layer, we might want to make the output readable, and for this reason we might want to convert the output to probabilities. If we are trying to train our network for a task of classification, (e.g. recognize a number of handwritten digits,) we may for example want to normalize the result so that the sum of the probabilities gives one. This can be done using the *softmax* function:

$$f(x_i) = \frac{\exp\left(x_i\right)}{\sum_{m=1}^{K} \exp\left(z_m\right)}. \tag{15}$$

This normalization process is important for the last part of our MLP training, namely the *back-propagation* algorithm, which from the errors between the prediction and the target tells us how to modify the weights in order for them to match the result the next time the inputs are fed forward.

2.2.3 Backpropagation

As written before, the backpropagation algorithm takes the difference between the predicted values and the target in order to build a cost function. For a classification problem this will have the same form as the cross entropy as seen in equation (5), just in its more general version where more than two classes are considered:

$$\mathcal{C}(\boldsymbol{a}_h) = -\sum_{i=1}^{n} \left(t_i(a_{h,i}^L) + (1 - t_i)\log\left(1 - a_{h,i}^L\right)\right), \tag{16}$$

where $\boldsymbol{a}_h$ is the vector containing the outputs, and $\boldsymbol{t}$ the corresponding target values. Our aim is to find how this cost function changes with respect to the output weights $\boldsymbol{W}$, and for this we can use the chain rule in this way:

$$\frac{\partial \mathcal{C}}{\partial w_{jk}^L} = \frac{\partial \mathcal{C}}{\partial a_{h,j}^L}\frac{\partial a_{h,j}^L}{\partial w_{jk}^L} = \frac{\partial \mathcal{C}}{\partial a_{h,j}^L}\frac{\partial a_{h,j}^L}{\partial z_{h,j}^L}\frac{\partial z_{h,j}^L}{\partial w_{jk}^L}, \tag{17}$$

where the two last partial derivatives are respectively the derivative of the activation function with respect to its argument,

$$\frac{\partial a_{h,j}^L}{\partial z_{h,j}^L} = \frac{\partial f(z_{h,j}^L)}{\partial z_{h,j}^L} = f'(z_{h,j}^L), \tag{18}$$

and the last is simply the output from the previous layer:

$$\frac{\partial z_{h,j}^L}{\partial w_{jk}^L} = \frac{\partial \left(\sum_k w_{jk}^L a_k^{L-1} + b_j\right)}{\partial w_{jk}^L} = a_k^{L-1} \tag{19}$$

The derivative of the cost function in equation (16) with respect to $\boldsymbol{a}_h$ is then

$$\frac{\partial \mathcal{C}}{\partial a_{h,j}^L} = -\frac{\partial \sum_{j=1}^n \left(t_j(a_{h,j}^L) + (1-t_j)\log\left(1-a_{h,j}^L\right)\right)}{\partial a_{h,j}^L} = -\frac{t_j - a_j^L}{a_j^L(1-a_j^L)}. \tag{20}$$

If we were to use a sigmoid (or a softmax) as the activation function for the last layer, then equation (18) would become

$$f'_{sigmoid}(z_{h,j}^L) = f_{sigmoid}(z_{h,j}^L)(1 - f_{sigmoid}(z_{h,j}^L)) = a_j^L(1-a_j^L) \tag{21}$$

meaning that our gradient of the cost function for a MLP classification problem with sigmoid/softmax output function with respect to the output weights will be

$$\frac{\partial \mathcal{C}}{\partial w_{jk}^L} = \frac{a_j^L - t_j}{a_j^L(1-a_j^L} a_j^L(1-a_j^L)a_k^{L-1} = (a_j^L - t_j)a_k^{L-1}. \tag{22}$$

We have now propagated the error from the output layer to the weights leading to it. In order to propagate it even further, we look at

$$\frac{\partial \mathcal{C}}{\partial a_{h,j}^L}\frac{\partial a_{h,j}^L}{\partial z_{h,j}^L}\frac{\partial z_{h,j}^L}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}, \tag{23}$$

where we have defined

$$\delta_j^L = \frac{\partial C}{\partial z_j^L}. \tag{24}$$

By changing the superscript from L to l, we find that

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l). \tag{25}$$

This is our last equation, telling us how the error propagates through the layers, giving for each a gradient which, tuned by the step length η, tell us how to change every weight.
When it comes to the change of the biases, we can see how equation (26) can also be written as

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial \mathcal{C}}{\partial b_{h,j}^L}\frac{\partial b_{h,j}^L}{\partial z_{h,j}^L} = \frac{\partial \mathcal{C}}{\partial b_{h,j}^L} \tag{26}$$

meaning that the rate of change of the cost function relative to the biases is the same as the previously found error.

This is true when it comes to classification. When doing a linear regression, the cost function and the final activation function may be different, so that the result in equations (16) and (18) may

not be used, and the resulting equation in (23) may be different. For a linear regression we might want to use the MSE cost function, that is

$$\mathcal{C} = \frac{1}{2}\sum_j \left(a_j^L - t_j\right)^2, \tag{27}$$

were a linear activation function in the form $f(x) = x$ is implied (when doing a regression, results may not necessarily be bound between 0 and 1).
Using the chain rule as before, we find out that

$$\frac{\partial \mathcal{C}}{\partial a_{h,j}^L}\frac{\partial a_{h,j}^L}{\partial z_{h,j}^L}\frac{\partial z_{h,j}^L}{\partial w_{jk}^L} = (a_j^L - t_j) f'_{linear}(z_j^L) a_k^{L-1} = (a_j^L - t_j) a_k^{L-1}, \tag{28}$$

which is surprisingly the same result as for the classification algorithm.
So whether we are doing a classification with a softmax final activation function, or a linear regression with a linear final activation function, we may use the same equation in order to find the errors to propagate back. If we are using other activation or cost functions for some reason, then we would have to calculate the derivatives in equation (23) from scratch.

2.2.4 Regularization

As a final addendum to the above discussion, the cost function may be supplied by a regularizing hyperparameter in order to avoid overfitting and that the weights grow too much. This might be done by adding the *L2-norm* of the weights modulated by the hyperparameter λ in a similar fashion as to the Ridge regression studied in the previous project. The L2-norm means in practice adding the following term to the cost function:

$$\lambda ||\hat{w}||_2^2 = \lambda \sum_{jk} w_{jk}^2. \tag{29}$$

The rate of change of this term with respect to the weights w_{jk} is easy to find:

$$\frac{\partial \lambda \sum_{jk} w_{jk}^2}{\partial w_{jk}} = \frac{1}{2}\lambda \sum_{jk} w_{jk} \tag{30}$$

where the factor of one half might as well be absorbed by the hyperparameter.

2.3 Metrics

How do we measure how well our algorithm is doing? This depends on which task it is given, as the MSE or the R2 score might not be the first choice when we are dealing with discrete predictions and targets.
One way to measure how a classification algorithm is doing is the *accuracy score*, which simply measures the ratio between right guesses against the total number of predictions, in other words

$$\text{Accuracy} = \frac{\sum_{i=1}^n I(\hat{y}_i = y_i)}{n}, \tag{31}$$

where a result close to one means a very good algorithm, while a score around 50 means that the model has zero clue what the right result might be. A score less than 0.5 would mean that the algorithm is guessing wrong all the time.
Despite telling us exactly how well our neural network is good at classifying, the accuracy score

does not tell us how sure the algorithm is in making the decisions. This would mean comparing the solution to the *probability* associated to that solution, so that a 0.99 certainty on a guess would count more than a mere 0.51 in our classification task. Such a metric would be for example the ROC-AUC score. This tells us that the algorithm is very good when it is close to one, while a score at around 0.5 means that the guesses are completely random, and the guessing power is zero.
Another such metric I will use in this report is the *area ratio* as presented in the Yeh and Lien paper [1], defined as

$$\text{Area ratio} = \frac{\text{area between model curve and baseline curve}}{\text{area between theoretically best curve and baseline curve}} \tag{32}$$

where for "model curve" is meant the cumulative gain curve. The result from this metric are interpreted such that the algorithm is good as long as this metric stays positive, and possibly as close to one as possible.

In order to find the best architecture for our neural network, other than some educated guessing and trial-and-error, a grid search with varying step lengths η and regularization hyperparameters λ might be useful in order to fine-tune the model.

3 Code, implementation and testing.

3.1 Program structure

The program written is this project is object oriented, meaning that the different algorithms are put together in classes, which makes it easier to structure and organize more advanced algorithms. The software is run from different run*.py-files that imports and calls upon other different instances of the program files. Since the code in many cases is the same (or a slight variation) from the one written for project one, I will simply refer to that report when certain algorithms have already been explained.
In this section I will describe the main classes and their methods, for then illustrating their functioning in some selected run*.py files.

3.2 Classes

3.2.1 Datasets

The `dataset_objects.py` file contains two classes, `dataset` and `credit_card_dataset`, where the second is a child class of the first, with some hard-coded, ad-hoc methods for the Credit Card dataset to be analysed in this project.
First I will introduce the `dataset` class. This is nothing else than a slight adaptation of the `data_generate` class from project one, already described in that report. The same algorithms for k-fold cross-validation, Franke-function generation and normalization have been adopted, with some small modification I will go through now.
Firstly, it is important to remember how this class takes care of the train-test sorting. While normally a routine is used to divide a dataset into two distinct training and test sets, when doing the division this class *replaces* the main "input data" attributes to contain only the training set data, while the full dataset is stored away in a backup file. This is done so that the same methods that worked for the whole set would continue working, just now only on the training set. For this reason, after the `fill_array_test_training` method is called, attributes such as `self.X`, `self.x_1d` or `self.y_1d` actually refer to the training set, and not the dataset as a whole.
After this brief recap, we can focus on the main differences between the original `data_generate` class, and this one.

One of the first ones is clearly the possibility to import `.xlsx` files as datasets. This is simply done via the Pandas package by writing the file address in the initialization method of the class, together with the eventual rows or columns to skip. If `0` is given instead of the filename, a blank `dataset` object is instantiated, which may be filled by generating a Franke function with its proper function, or filled with some other dataset.
The `polish_and_divide` method also is new. This does nothing else than dividing the dataset into inputs and targets.
`normalize_dataset` is mostly the same, except for the fact that the scikit-Learn `StandardScaler` has been replaced with a `MinMaxScaler`, as this was shown to give more stable results.
Another small change has been done to the `sort_into_k_batches` method, where minibatch sorting was implemented. This was just a modification to the old k-fold CV sorting, where the same algorithm is run, and a list of indices is then given to either the `self.k_idxs` or the `self.m_idxs` attribute, depending on whether we are sorting for a *k-fold* CV, or in minibatches for a SGD algorithm.

The `credit_card_dataset` class inherits all the methods from its parent class, and adds some dataset-specific ones.
The `CreditCardPolish` method for instance runs the parent class' `polish_and_divide` function but also allow for the option of dropping some of the dataset values (more discussion on this in the Analysis part of the report).
Otherwise, the `plot_setup` method prepares the dataset for a visualization of its features by for example sorting between continuous and discrete values, and the saving of the difference in the age span between the respondents in the dataset, so that a histogram plot of the age distribution is binned correctly.

3.2.2 Fits

This class `fit` mostly correspond to the same one written for project one CITEOLDREPORT, with some few additions. The first is the implementation for the creation of design matrices of not polynomial nature. This was done by updating the `create_design_matrix` method to point to the old algorithm for polynomial design matrices when `deg>0`, while it would point to the new method `design_simple_design_matrix` when `deg=0`. In this case the input dataset (saved as `self.inst.x_1d` in the object) is used as a design matrix.
The main update to this class though is the implementation of the logistic regression algorithms based on the discussions in section 2.1. `fit_design_matrix_logistic_regression` is the method taking care of this, taking as input the desired descent method (might it be normal Gradient Descent `'GD'`, Stochastic Gradient Descent with varying step length and minibatches `'SGD'` or the scikit-learn version of the SGD algoritm with `'skl-SGD'`), the step length η in `eta`, the number of iterations in `Niterations` and the number of desired minibatches for the `'SGD'` algorithm. The code follows the same steps described in the Formalism section and is easy to understand its functioning. The Gradient descent is looped as many times as given as input in `Niterations` for the `'GD'` descent method, while for the `'SGD'` algorithm the same number of iterations is divided by the number of minibatches, looping then through $Niterations/m$ epochs, for then returning the parameters `self.betas` which are our fit.

3.2.3 Sampling methods

A thorough description of the `sampling` class is again given in the report for project one. The only updates are the possibility to call the `'logreg'` regression method from the `fit` class, and the possibility to save and write the appropriate metrics for when we are doing a classification instead of a regression.

3.3 Functions

Most of the functions in the library `functions` are self-explanatory or well commented.

3.3.1 Statistical evaluations of results

The main additions to this project are the evaluation metrics for the classification algorithms. Since the results are not continuous, the MSE and the R2 cannot be used as sensible metrics, and other have to be implemented.
One is *accuracy*, explained in the Formalism section. The algorithm written in the `statistical_functions.py` file is easy to understand, and in our code we will use it interchangeably with the scikit-learn version.
The other metric we use is the ROC-AUC score in the `calc_rocauc` function, which simply calls scikit-learn function evaluating the same metric.
The last one is the *area ratio* as explained in the Yeh and Lien paper in [1]. This one will calculate the ratio between area under the cumulative gain graph and the baseline, and the maximal area for the ideal solution. This is calculated in `calc_area_ratio`, which in turn calls `calc_cumulative_auc`, a functions that makes use of a utility from the package `scikit-plot`.

3.3.2 Neural Network

This is the main class of the project, and the newest. Although mainly based on the `NeuralNetwork` class found in the lecture notes of Morten Hjorth-Jensen in [2], the class has been expanded to be able to support MLPs of more layers, different activation functions and different cost functions.
Most of the algorithms have been discussed in the Formalism section and the code is otherwise well commented and understandable. The only thing I will explain in this section is the interplay between the `NeuralNetwork` class and the `layer` class, introduced to be able to have architectures of different layers.
When generating a `NeuralNetwork` instance, a hidden layer is automatically generated with the given activation function and number of nodes. When calling the `add_layer` method in the class, a `layer` instantiation is generated with the given number of nodes and activation function. A `layer` object will contain the weights pointing to its nodes, the corresponding biases and an attribute telling which activation function we should use in our feed forward algorithm, and correspondingly which derivative in the backpropagation.
Something different happens in the last layer, which gets also the weights and the biases leading to the output layer. These attributes are created by the `NeuralNetwork`'s `close_last_layer` method, which is called at the beginning of the `train` function, waiting effectively until this last function is called before the whole architecture is finally built. A visual representation of the `layer` class is given in figure 1.

3.4 Example runs

The runnable codes in the source are `run_a.py`, `run_b.py`, `run_c.py` and `run_d.py`, where each correspond to each exercise in the project paper. While `run_a.py` simply is plotting the different features of the Credit Card dataset, `run_b.py`, `run_c.py` and `run_d.py` run a logistic regression, a classification with neural networks and a linear regression, again, with neural networks. Since `run_c.py` and `run_d.py` are quite similar, I will only describe one of them, and I will start with describing `run_b.py`.

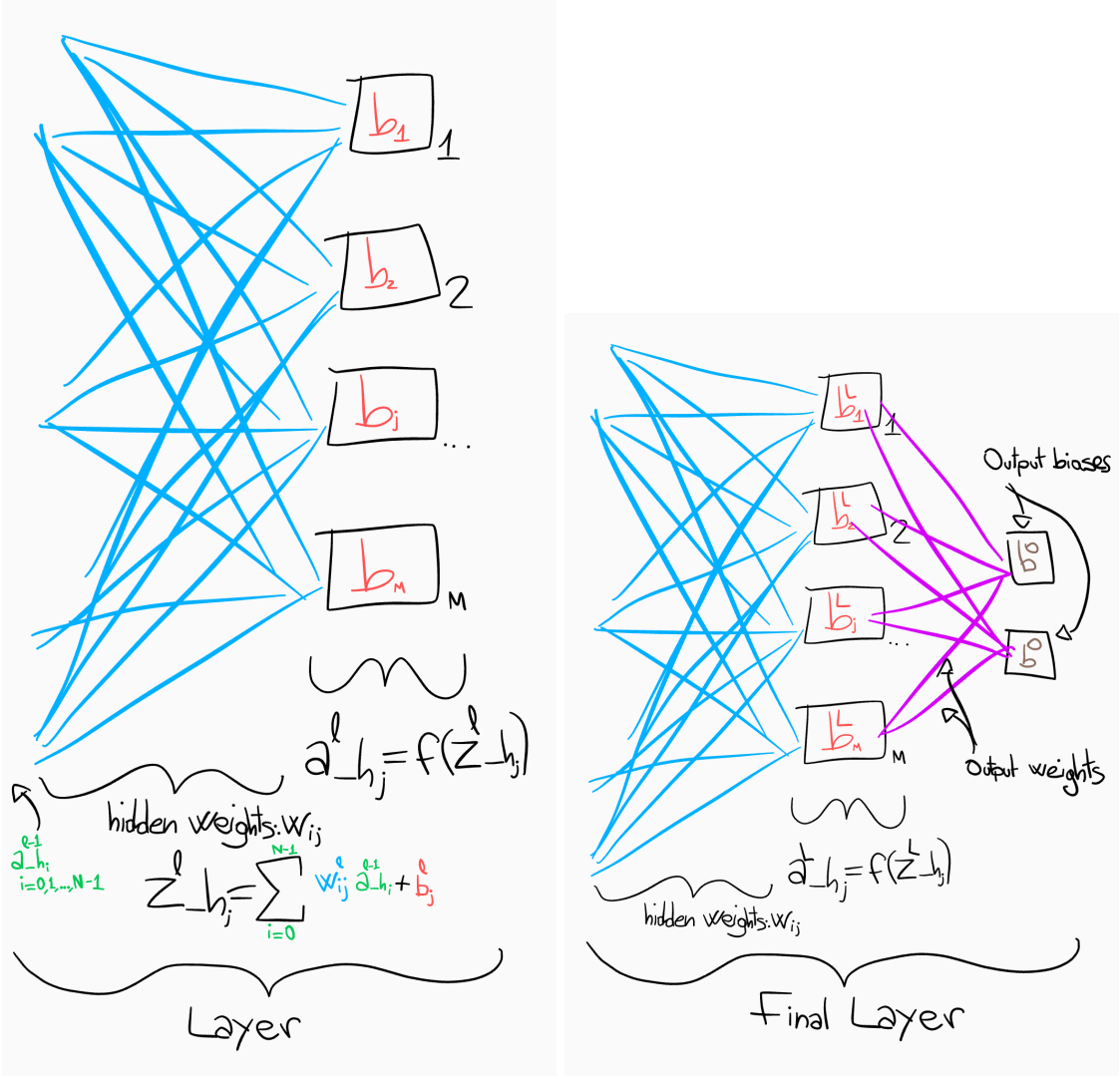

Figure 1: sketch of the structure of the `layer` class. All layers will look like the one on the left except the last one, which will also include the output weights and biases.

3.4.1 Logistic regression

`run_b.py` does a logistic regression in a much similar way to the regressions done in the previous project. From line 1 to 37 parameters are just being initialized, and in this project specifically we will only be varying the descent method, whether we are doing a cross validation (`CV`) and, sometimes, the `input_eta` parameter. Line 37 enforces the random functions to yield the same results, while in line 34 we might choose to test the script with a random dataset generated by the scikit-learn package. Between the lines 50 and 63 the dataset is prepared for fitting and divided either in k batches for the cross validation, or just sorted into training and test sets. In lines 66 and 69 the fit is created, and the model is then evaluated in the following if-statement, asking whether we are doing a cross validation or not. If we are doing a CV, then a `sampling` object is instantiated and the different arguments carried over (such as the `eta`, the descent method and so on). In the end, an output is printed giving us the metrics that evaluate the classification task.

3.4.2 Neural networks

`run_c.py` and `run_d.py` run a neural network regression, and are similar in this regard. In this section we will analyse `run_c.py`, but many of the considerations are valid also for the other code. Between lines 0 and 48 we mostly have initialization of different parameters in a similar way as in `run_b.py`. From line 52 and onward, a grid search is done by first initializing all the lists with all the values of interest to be plotted, and from line 72 the proper search start. Between lines 76 and 87 a neural network is created, with two hidden layers of 20 neurons each, is trained in line 90 and the predictions are collected between lines 93 and 96, saved in the respective matrices between lines 99 and 104, and which after the loop are exported to the plotting script in the `visualization.py` file.

3.5 Testing

Some testing has been done with a randomly generated dataset from the scikit-learn datasets library. This is activated by setting `randomdataset = True` in the scripts. Although not reproducing known values since made of random values, it ensures that the algorithms are doing a good job if we get a good regression for sensible parameters.

4 Analysis and results

After having introduced both the theory and the program structure, I will here present the results of the scripts, together with an analysis of them.

4.1 The Credit Card Dataset

Problem a) focused on getting known with the Credit Card dataset. The dataset has 30000 rows, each of them corresponding to a client for of a major Taiwan bank. After a period of over-issuing of credit cards and the default of many of the clients, there was an interest in developing a model that could predict whether a client would default on his debt or not, given its personal and financial background.
This background consists of 23 parameters:

- The given credit, in NT dollars (continuous)
- The gender of the client: 1 for male, 2 for female (discrete)
- Education: 1 for graduate school, 2 for university, 3 for high school, 4 for others (discrete)
- Marriage: 1 for married, 2 for single, 3 for others (discrete)
- Age, in years (discrete)
- History of payments: This is six variables, each corresponding, in increasing order, to how many months before the measurement the history refers to. -1 stands for pay duly, 1 for payment delay for one month, 2 for payment delay for two months and so on, until 9 for payment delay for nine months and above (discrete)
- Amount of bill statement in the corresponding month, following the order of the previous parameter. NT dollars (continuous)
- amount of payment the previous month, again in the same order as the parameters above, in NT dollars (continuous).

This description was taken from the research paper in [1], but it is incomplete. The dataset has in fact undefined results when it comes to education (undocumented values of 0, 5 and 6) and marriage (undocumented values of 0). In addition, there are also undocumented values of -2 and 0 to be found in all the six "history of payments" categories. Deleting all these values would give us a too small dataset, and these values appear not to be random noise, so the approach taken in this analysis is to leave the data almost as it is (deleting only the clients who did not use the credit card, meaning those with both bill statements and amount of payments like 0) and test the prediction power of the different algorithms with this (possibly incomplete) dataset. After taking out those who did not use the credit card, we are left with 28497 clients, of whose 78.69% did not default on their debt, and 21.31% did.
A visual representation of the data is given in figure 2, together with a correlation plot in figure 3 where we see the occurrences of the undocumented discrete values are dominant in the `PAY_ *` categories. The correlation plot shows that there is high correlation between the six bill statement categories (meaning that people tend to use their credit card consistently through the months) as well as, as we should expect, between the insolvence in the past months to the final default. The correlation matrix show in addition no correlation between sex, education, civil status or age with the possibility of defaulting on a debt.

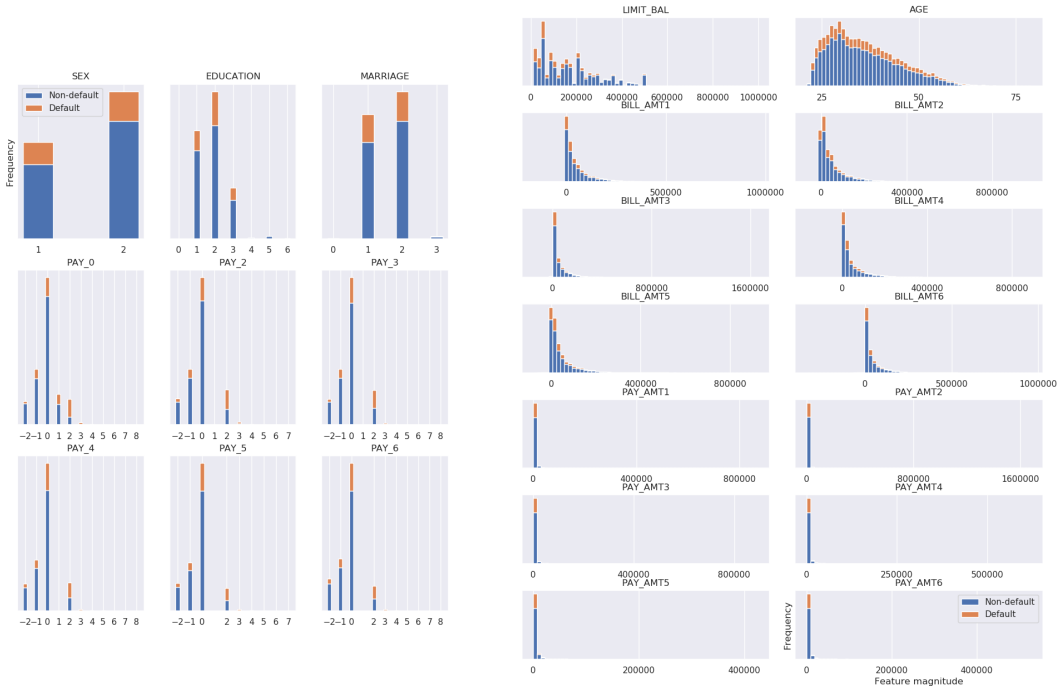

Figure 2: Histograms of the discrete and the continuous categories, on the left and the right respectively. We see the presence of undocumented values in most of the histograms on the left, where we should not expect values of -2 and 0 for the `PAY *` categories, values of 0 in marriage, or values other than 1, 2, 3 and 4 in education.

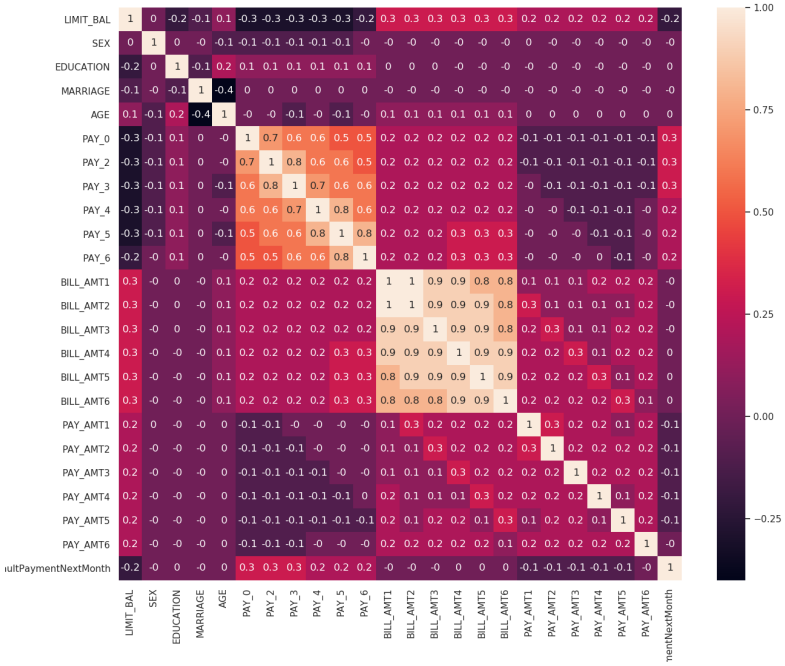

Figure 3: Plot of the correlation between the different categories. Here we see that how some of the categories are highly correlated, for example the six "payment status" and the six "bill amount" categories.

4.2 Logistic regression

The code in `run_b.py` gives the results for our classification using logistic regression methods. I used different algorithms, all of them on the same random division between training and test set enforced by a a `numpy` seed. These are:

- A standard gradient descent algorithm with fixed step length (GD)
- A stochastic gradient descent algorithm with varying step length and minibatches, this for three different starting step lengths (SGD)
- The scikit-learn provided algorithm for stochastic gradient descent (skl-SGD)
- The three above, now with 5-fold cross validation.

The results are shown in table 1. All of them are generated with 5000 iterations. For the SGD algorithm, the dataset was divided into 20 minibatches (giving thus 250 epochs). For all of them, the test set size was 20% of the whole dataset, and the starting step length was 0.1 (except for the SGD algorithm, where also 1.0 and 0.01 were tested). The metrics considered are the accuracy score, the ROC-AUC score and the area ratio, all of these explained in the Formalism section.
First it should be noticed that the testing algorithm for the calculation of the SGD method did not work so well, being highly dependent on the dataset generated: sometimes it would give reasonable results, while other times an accuracy of around 0.5, thus useless. The seed chosen for this report gave these results, but it has to be said that the way of splitting between test and training data

would change the accuracy score dramatically. This is also seen in the difference in the accuracy score between the `skl-SGD` with and without cross validation, where one should expect quite similar results.
These results for the SGD algorithm are scarily close to the percentage of zero values in the target, and the randomness of these results is somehow confirmed by the ROC-AUC scores around 0.5.
The ROC-AUC score and area ratio results for both the GD and the skl-SGD are though what we should expect, giving us exactly the same values found in the Yeh et al. paper in [1]. The accuracy of the skl-SGD method with CV is also in accordance to the paper which makes me believe that the weird result gotten with the same method without CV is a statistical fluke.

Method, non CV	accuracy	ROC-AUC	Area ratio
skl-SGD	0.3556?	0.7239	0.4477
SGD, $\eta = 0.1$	0.7626	0.5641?	0.1282?
SGD, $\eta = 0.01$	0.6419	0.5693?	0.1386?
SGD, $\eta = 1.0$	0.7903?	0.5667?	0.1334?
GD	0.8140	0.7233	0.4466
Method, with CV			
skl-SGD	0.7958	0.7325	0.4650
SGD, $\eta = 0.1$	0.7909?	0.5794?	0.1587?
SGD, $\eta = 0.01$	0.7701?	0.5860?	0.1718?
SGD, $\eta = 1.0$	0.7878?	0.5810?	0.1619?
GD	0.8046	0.7262	0.4523

Table 1: The results from the logistic regression for different methods. Question marks mean bad values.

4.3 Neural Network: Classification

The code `run_c.py` gives the results for our classification analysis using neural networks. The MLP's architecture for such a task require a cross entropy cost function discussed in the Formalism section, this because of the binary nature of the output. For this task I chose to have two layers of 20 neurons each after having tried some combinations and settling for this because of the perceived accuracy and the non-prohibitive running time. I found out that 100 epochs was a good compromise between precision and running time. I tried this architecture with both the sigmoid and the tanh activation function, and a grid search for varying step length `eta` and hyperparameter lambda yielded the results shown in figures 4 and 5 respectively, in the Appendix. Although being more precise than the sigmoid, the tanh looks like being more numerically unstable, breaking down seemingly an order of magnitude of `eta` before the sigmoid one. The tanh in figure 5 gives a top Area ratio score of 0.52, better than all logistic regressions seen in table 1 and very close to the one of 0.54 found in the Yeh et al. paper [1], for a step length of 0.001 and a hyperparamter lambda of 1e-6 (although very little variation was found with varying lambdas for different random seeds). The accuracy of around 0.78 found in both figures is not representative, as it would be attainable by just guessing all zeros since it is the percentage of clients not defaulting.
The results for other random seeds are also attached, another for the tanh activation function in figure 6 where it is shown how we can reach the same metrics as in the 2009 Yeh at al. study [1] for certain parameters, and onther random seed grid search for the ReLU activation function in figure 7, showing the increased instability of such an architecture.

4.4 Neural Network: Regression

When it comes to the task of continuous regression, the cost function must be modified, as now we are not looking for probabilities, but how close we can get to a specific, non-discrete value. The MSE cost function is ideal for this the task, and as seen in the Formalism section, quite applicable to MLP neural networks as well. The internal activation function might be a sigmoid or tanh, but the output should be something unbound in order to match the nature of the target. A linear activation function for the output layer could for example be chosen.
For this task I expanded the previous neural network with another layer of 20 nodes, making in total three hidden layers. The set up is almost identical as for the classification problem, except for the activation function for the output layer (now set to `'linear'` for the reason explain above) and the cost function set to `'MSE'`. The metrics are also different now, and we will evaluate how good the neural network is at continuous regression by looking at the MSE and the R2 score. A grid search is shown in picture 8 for the sigmoid activation function between the hidden layers, and in picture 9 for the tanh activation function. The figures are found in the Appendix.
A Franke function was generated, with 150 points for each coordinate, randomly chosen, and a normally distributed noise of 0.05 around the output. We observe how, especially using the tanh activation function, the MSE and the R2 score yield much better results both for the train and the test sets than a simple OLS or Ridge regression. By running a simple regression in the file `Franke_linear_regression` (which reproduces the same results as in project one, just now with the same seed as for the neural network code for statistical coherence), we see how with a grid search we are able to train a network to outperform an OLS or a ridge regression when it comes to predictions even for the same generating parameters, as shown in table 2. A plot of the best fit to the Franke function is given in 10 in the Appendix.

Method	MSE	R2 score	Param. values
FFNN, sigmoid	0.0045	0.91	$\lambda = 0$, $\eta = 0.001$
FFNN, tanh	0.0042	0.97	$\lambda = 0$, $\eta = 0.0001$
OLS	0.0062	0.8720	
Ridge	0.0063	0.8703	$\lambda = 0.01$
Lasso	0.0080	0.8360	$\lambda = 0.0001$
Lasso	0.0065	0.8776	$\lambda = 0.00001$

Table 2: The results from the regression analysis for both neural network and linear regressions. We see how the neural network outperforms the other methods.

5 Conclusion

In this report we have explored logistic regression, a classification method that builds one the same concepts as the linear regression seen in the previous report: building a cost function, and attempting to find its global minimum. When it is not possible to find an analytical solution, gradient descent methods are to be applied, and apart from the naive one there are improvements in the stochastic gradient descent that solve the problems of numerical heaviness and the danger of getting stuck in a local minimum.
Another way to do a classification is the use of neural networks. We use a Multilevel Perceptron for this task, and analyse its predicting power against the logistic regression methods, to find that they are better when we find the right architecture and parameters.
As a test for its predicting power, except simply checking its accuracy or ROC-AUC metrics, I used the Credit Card Dataset already analyzed by Yeh et al. in [1], where after some discussion, I showed how my code is able to reproduce the same results published in the 2009 paper both for the

logistic regression and for the neural network code.
An analysis on the predicting power for regression problems was also done, showing how a neural network could outperform simple OLS, ridge and LASSO linear regressions.
In conclusion, it seems like a neural network is able to outperform simple logistic regressions for classification problems, and OLS, ridge an LASSO for continuous regression tasks. This though comes with a drawback, which is having to nudge on many different parameters like the step length, the regularization parameter, the choice of activation function and so on, and having to deal with the danger of numerical instability when it comes to continuous regression.

6 Appendix

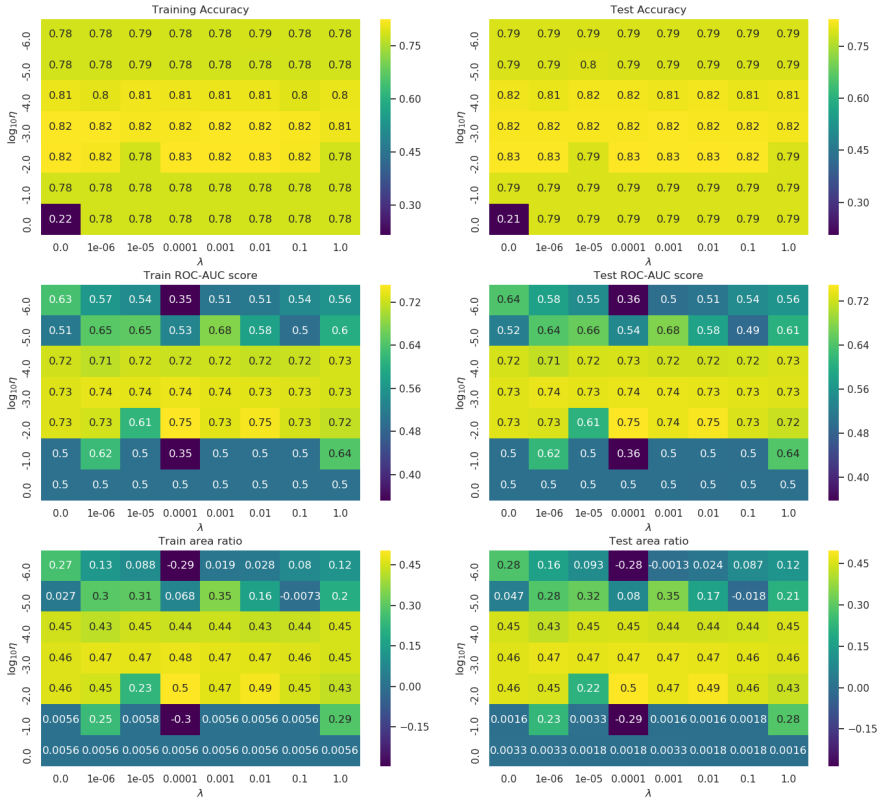

Figure 4: Grid search with the current random seed. Varying etas and lambdas for the classification task using a MLP of two hidden layers of 20 nodes each. Here are the accuracy, the ROC-AUC score and the Area ratio shown for varying values for both the training and the test sets with the sigmoid activation function between the layers.

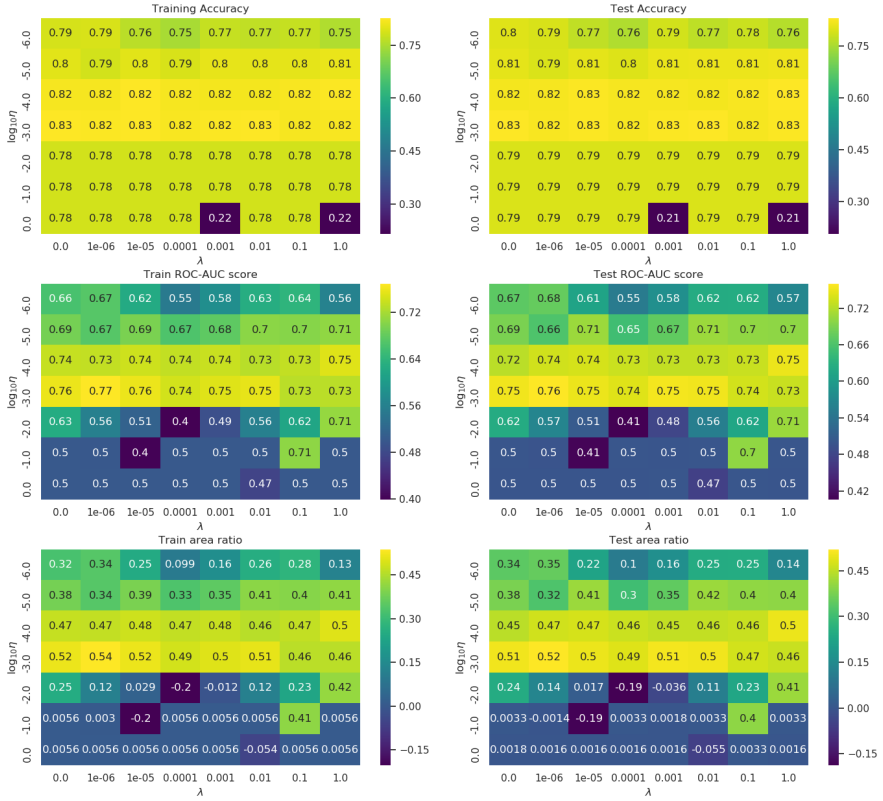

Figure 5: Grid search with the current random seed. Varying etas and lambdas for the classification task using a MLP of two hidden layers of 20 nodes each. Here are the accuracy, the ROC-AUC score and the Area ratio shown for varying values for both the training and the test sets with the tanh activation function between the layers.

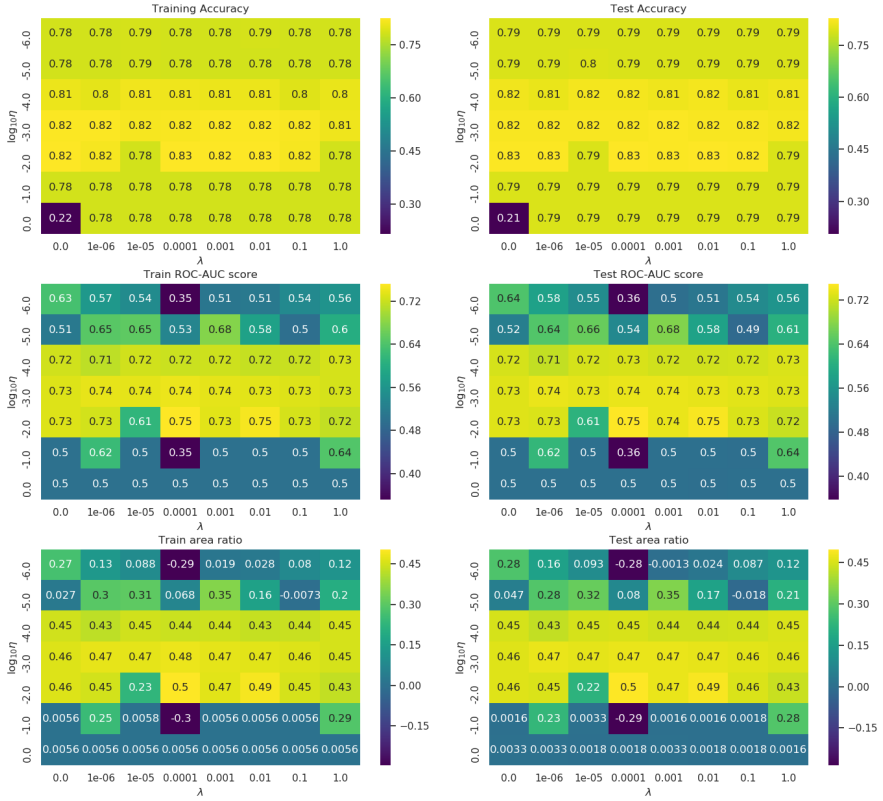

Figure 6: Grid search for another random seed for the same architecture as above, again, with the tanh activation function. Here we see how the area ratio value for the test set reaches 0.54 for certain values of `eta` and lambda, exactly what obtain in [1]. The accuracy also seem to correspond.

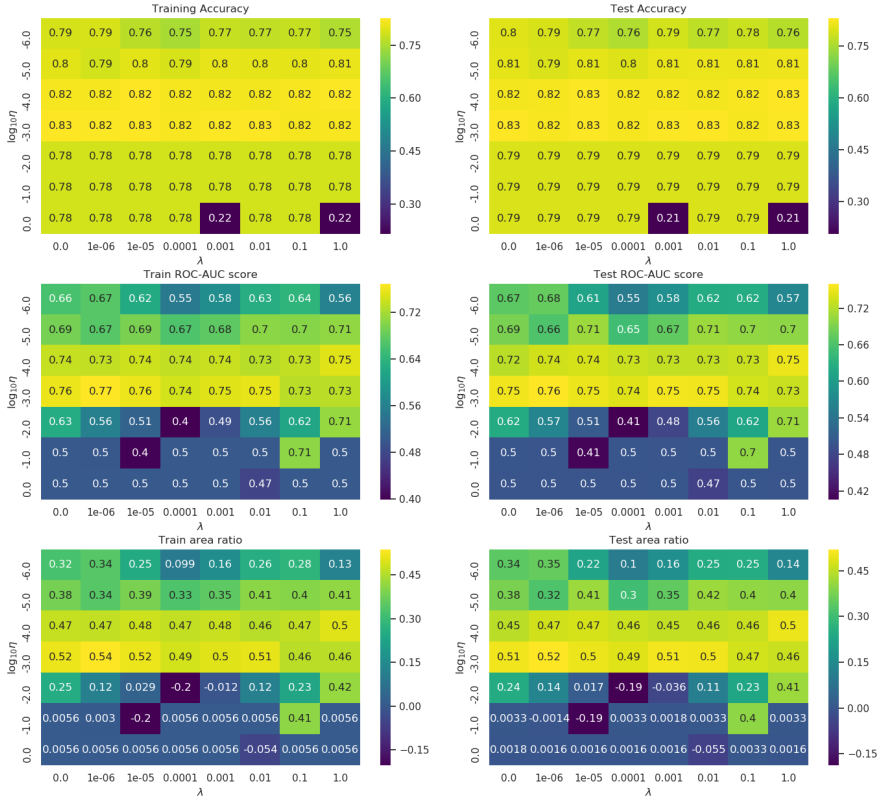

Figure 7: Grid search for another random seed for the same architecture as above, again, with the ReLU activation function. Here we see how the network becomes more unstable as the activation function is not bounded from above.

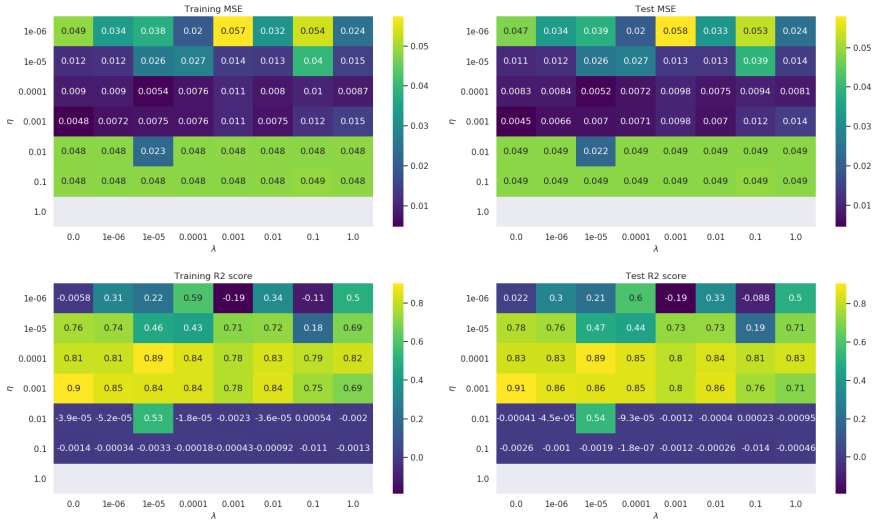

Figure 8: Grid search for the continuous regression using the sigmoid activation function between the three 20-nodes hidden layers, MSE cost function and linear output activation function, for the 1234 seed.

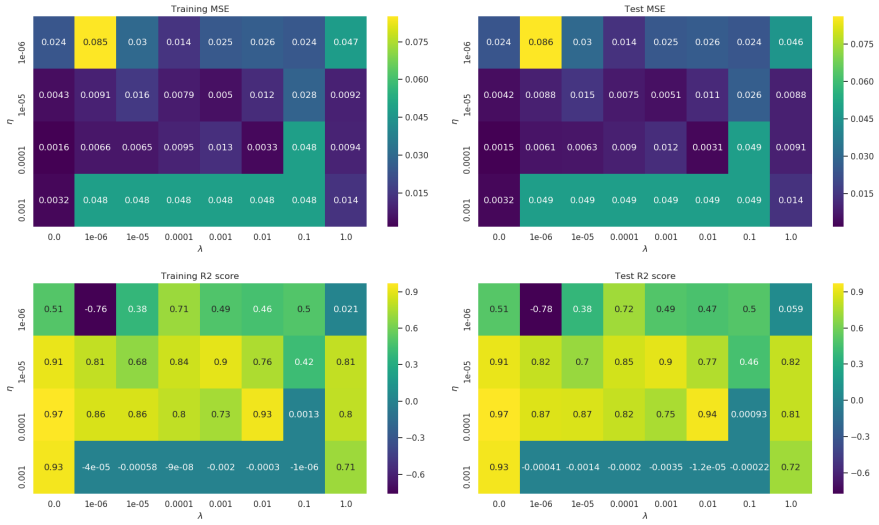

Figure 9: Grid search for the continuous regression using the tanh activation function between the three 20-nodes hidden layers, MSE cost function and linear output activation function, for the 1234 seed. The code would become computationally unstable for `eta` bigger than 0.001, and was stopped. We see how we manage to get better results than with using the sigmoid, although we pay for more numerical instability.

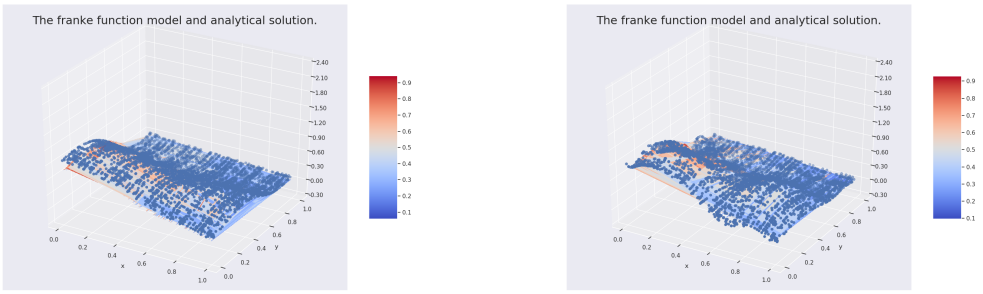

Figure 10: Plots of the best fits to the Franke function by the regression FFNN: sigmoid activation function on the left, and tanh on the right.

References

[1] Ivy Yeh and Che-Hui Lien. "The comparisons of data mining techniques for the predictive accuracy of probability of default of credit card clients". In: *Expert Systems with Applications* 36 (Mar. 2009), pp. 2473–2480. DOI: 10.1016/j.eswa.2007.12.020.

[2] Morten Hjorth-Jensen. *Lecture notes, course FYS-STK4155*. compphysics.github.io/MachineLearning/doc/web/course.html. Department of Physics, University of Oslo et al., 2019.

[3] Francesco Pogliano and Marianne Bjerke. *Data Analysis and Machine Learning, Project 1*. https://github.com/Loopdiloop/fys-stk4155/blob/master/ML.pdf. Department of Physics, University of Oslo and Department of Physics, 2019.