From linear and logistic regression to neural networks

FYS-STK4155 – Project 2

Stjepan Salatovic
github.com/stipesal/FYS-STK4155

November 16, 2021

Abstract

Using (mini-batch) stochastic gradient descent, we compare iterative linear regression methods with fully connected feedforward neural networks and perform hyperparameter tuning of all relevant variables. Closed-form solutions for OLS and ridge regression are the best performing ones, also in terms of computation time. Activation functions, weight matrix intializations, and network architecture have a very strong impact on a feedforward neural networks performance. The results show that highly optimized implementations of logistic regression outperform neural networks on relatively simple classification tasks. We support our results using experiments on Franke's function as well as datasets on breast cancer, digits and fashion MNIST.

Contents

1 Neural networks **2**
1.1 Architecture . . . 2
1.2 Interpretation . . . 2
1.3 Regularization . . . 3
1.4 Training . . . 3

2 Optimization **4**
2.1 Gradient descent methods . . . 4
2.1.1 (Batch) gradient descent . . . 4
2.1.2 (Mini-Batch) stochastic gradient descent . . . 5
2.2 Backpropagation . . . 5
2.3 Weight initialization . . . 6

3 Logistic regression **7**
3.1 Model assumption . . . 7
3.2 Cross-entropy loss . . . 7
3.3 Design matrix . . . 7
3.4 Derivative . . . 7

4 Data & Results **8**
4.1 Franke's function . . . 8
4.2 Breast cancer data . . . 14
4.3 Digits MNIST . . . 15
4.4 Fashion MNIST . . . 16

5 Discussion **17**

1 Neural networks

Machine learning can be broken down into several broad categories. In supervised learning the algorithm builds a mathematical model to approximate a function given labeled training data, i. e. data with inputs and desired outputs. The main goal is to map unseen instances with the learned function approximation to a reasonable output. Supervised learning tasks include classification and regression, which we deal with in particular in this report. In unsupervised learning, on the other hand, only unlabelled data is available and the algorithm tries to organise itself and find a mathematical pattern. Tasks of this type include clustering and dimensionality reduction. Typical machine learning algorithms include decision trees, support vector machines, and neural networks. The idea of an artificial neural network was firstly mentioned by McColloch and Pitts in 1943 (see [7]). Since then, artificial neural networks have achieved outstanding accuracy in certain tasks, surpassing even human cognition. Over the last couple of years, many neural network architectures have emerged to tackle problems in a wide range of domains. It should be noted that despite it draws inspiration from biological brains, neural networks do not operate the same way our brains do.

1.1 Architecture

Mathematically speaking, a neural network is a directed graph, where nodes correspond to neurons holding values, and edges represent weighted connections. In a fully connected feedforward neural network, each neuron in a layer is connected to every neuron in the previous and following layer. More general, feedforward neural networks do not have any cycles between neurons. Thus, feedforward neural networks are directed, acyclic graphs.

To introduce neural networks more formally, let $L \in \mathbb{N}_0$ and $p = (p_0, \dots, p_{L+1})^\top \in \mathbb{N}^{L+2}$. A *neural network* with architecture (L, p) and *activation function* $\sigma : \mathbb{R} \to \mathbb{R}$ is a function $g : \mathbb{R}^{p_0} \to \mathbb{R}^{p_{L+1}}$ with

$$g(x) = W^{(L+1)} \cdot (f_L \circ f_{L-1} \circ \cdots \circ f_1)(x),$$

where

$$f_l(x) = \sigma\left(W^{(l)} \cdot x + b^{(l)}\right), \quad l = 1, \dots, L$$

are intermediate transformations between layers.[1] By $\mathcal{F}(L, p)$ we denote the set of all such neural networks g with architecture (L, p) (and activation function σ). The non-linearity σ is necessary so that the network g does not collapse into a linear model. The most popular activation is the *rectified linear unit* (ReLU) function given by

$$\text{ReLU}(x) = \max\{0, x\}$$

The parameters of the neural network g are given by *weights*

$$W^{(l)} \in \mathbb{R}^{p_l \times p_{l-1}}$$

for $l = 1, \dots, L+1$ and *biases*

$$b^{(l)} \in \mathbb{R}^{p_l}$$

for $l = 1, \dots, L$. Further, we call L the *depth* and p the *width vector* of the neural network g. To illustrate the last two terms, a small fully connected feedforward network with $L = 1$ and width vector $p = (3, 4, 2)^\top$ is depicted in Figure 1. The first layer is called the *input layer* and the last is called the *output layer*. All layers in between are called *hidden layers*. Sometimes L is also referred to as the number of hidden layers. One speaks of a *deep neural network* if L is sufficiently large, although there are no precise definitions.

In order to complete the example, the network in figure 1 is given by $g : \mathbb{R}^3 \to \mathbb{R}^2$ with

$$g(x) = W^{(2)} \cdot \sigma\left(W^{(1)} x + b^{(1)}\right),$$

and weights $W^{(1)} \in \mathbb{R}^{4\times 3}$, $W^{(2)} \in \mathbb{R}^{2\times 4}$, bias $b^{(1)} \in \mathbb{R}^4$, and an activation function σ.

1.2 Interpretation

Due to the abstract introduction of neural networks, an interpretation into the problem types regression and classification is easily possible as follows. An element $g \in \mathcal{F}(L, p)$ is called neural network for regression if $p_{L+1} = 1$. Otherwise, if $p_{L+1} = K > 1$ then g is called neural network for classification with K classes. The *decision rule* (or prediction) associated with the network g is then

[1] The application of the activation σ is to be understood component-wise.

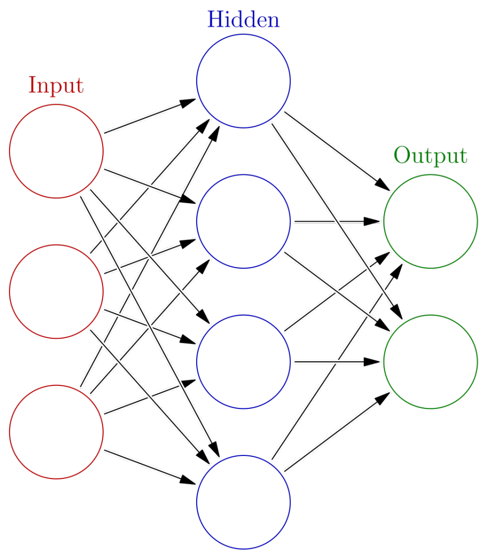

Figure 1. An example fully connected feedforward neural network (see [8]). The underlying architecture is given by $L = 1$ and $p = (3, 4, 2)^\top$.

given by $f = g$ in the regression case, and in the classification case by

$$f(x) = \underset{k \in \{1,\dots,K\}}{\arg\max}\; M_k\left(g(x)\right),$$

where $M : \mathbb{R}^K \to \mathbb{R}^K$ with

$$M_k(x) = \frac{e^{x_k}}{\sum_{l=1}^K e^{x_l}}, \qquad k = 1, \dots, K$$

is the so-called *softmax*-function. This function has the property that $M_k > 0$ for all $k \in \{1, \dots, K\}$ and $\sum_{i=1}^K M_k \equiv 1$ for all $x \in \mathbb{R}^K$, which is why it can be regarded as an approximation to a probability density function. So in the regression case, the prediction is given directly by the network output, whereas in the classification case one still applies a final activation function (usually softmax) to the output layer and chooses the class corresponding to the maximum there as the prediction.

1.3 Regularization

To prevent overfitting to training data, a penalty is usually introduced for the parameters in the network, as in ridge regression. For this purpose let $q \geq 1$ and the q-norm of a matrix $A \in \mathbb{R}^{n \times m}$

$$\|A\|_q = \left(\sum_{i=1}^n \sum_{j=1}^m |A_{ij}|^q \right)^{1/q}.$$

Then, for a network $g \in \mathcal{F}(L, p)$ we denote by

$$J(g) := \sum_{l=1}^{L+1} \|W^{(l)}\|_2^2.$$

the *penalty term* for g. The biases $b^{(l)}$ are not penalised here, although this is generally possible.

1.4 Training

As with linear regression, a suitable loss function must be chosen that firstly correctly maps the error between training data and current network output, and secondly is sufficiently smooth. Of course, this differs depending on the problem type.

Given training data (x_i, y_i) for $i = 1, \dots, n$ we impose the usual *mean squared error* (MSE)

$$L(g) = \frac{1}{n} \sum_{i=1}^n \left(y_i - g(x_i)\right)^2$$

for regression problems, and the *(categorical) cross-entropy loss*

$$L(g) = -\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^K \mathbb{1}_{\{Y_i = k\}} \log M_k\left(g\left(x_i\right)\right)$$

for classification problems. The latter is the result of a maximum likelihood approach for the softmax output, which can be regarded as a probability due to its properties mentioned above. If further a regularization parameter $\lambda \geq 0$ is given, then the optimization problem at hand is to find the neural network

$$g^* = \underset{g \in \mathcal{F}(L,p)}{\arg\min} \left[L(g) + \lambda \cdot J(g)\right]$$

where $J(g)$ denotes the penalty term for $g \in \mathcal{F}(L, p)$. To solve the optimization problem and thus determine g^*, iterative methods such as the stochastic gradient method are used.

2 Optimization

In this Section we deal with the question of how to determine g^*, the solution of the stated optimization problem. The answer is: not at all. In general, only a sufficiently good approximation to g^* is determined through modifications of gradient methods, which we will introduce further down below.

Parameters θ of a neural network

In order to iteratively update the parameters of a network we would like to have one vector of all parameters and hence need a formal definition. For this, first define the column-wise vectorization of a matrix $A \in \mathbb{R}^{n\times m}$ by

$$\operatorname{vec}(A) = (A_{11}, \dots, A_{nm})^\top \in \mathbb{R}^{n\cdot m}.$$

Let $g \in \mathcal{F}(L,p)$. The parameters of layer $l \in \{1,\dots,L\}$ are then given by a vertical concatenation with the bias

$$\theta^{(l)} = \begin{pmatrix} \operatorname{vec}\left(W^{(l)}\right) \\ v^{(l)} \end{pmatrix} \in \mathbb{R}^{p_l(p_{l-1}+1)},$$

and the total parameters of $g \in \mathcal{F}(L,p)$ by the super vector

$$\theta = \begin{pmatrix} \theta^{(1)} \\ \vdots \\ \theta^{(L)} \\ \operatorname{vec}\left(W^{(L+1)}\right) \end{pmatrix}.$$

Note that this now also allows the unambiguous notation $g = g_\theta$, if θ is the parameter vector belonging to $g \in \mathcal{F}(L,p)$.

The determination of $g^* \in \mathcal{F}(L,p)$

The calculation of g^* proves to be difficult because, firstly, for a fixed architecture (L,p) a neural network $g \in \mathcal{F}(L,p)$ lives in a relatively large parameter space. To be more precise, the size of the super vector $\theta \in \mathbb{R}^D$ that uniquely defines a neural network $g = g_\theta$ is given by

$$D = \sum_{l=1}^{L+1} p_l p_{l-1} + \sum_{l=1}^{L} p_l.$$

Note that if the number of hidden layers $L \in \mathcal{O}(n)$ and the number of neurons in each layer $p_l \in \mathcal{O}(n)$ for all $l = 1,\dots,L$, then the number of total parameters $D \in \mathcal{O}(n^3)$. The second reason for difficulty in determining g^* is that the function

$$\theta \mapsto [L(g_\theta) + \lambda \cdot J(g_\theta)]$$

is in general not convex and has many local minima. In practice, one therefore limits oneself to the determination of a local minimum with the hope of similar properties as those of the global minimum. Moreover, one can ask whether the determination of the global minimum $g^* \in \mathcal{F}(L,p)$ makes sense at all with regard to an overfitting to the training data.

2.1 Gradient descent methods

The overall purpose of any neural network is generally minimizing its loss function by converging into a (local) minimum by iteratively calculating the gradient of the loss function at the current point and adjusting the weights of the neural network in the direction of its largest negative slope. A crucial piece of this procedure is the optimization algorithm during the iterative process of convergence. Before we (can) proceed with the training of neural networks, we need to know how these iterative optimization methods work.

2.1.1 (Batch) gradient descent

Given are a sufficiently smooth function $F : \mathbb{R}^N \to \mathbb{R}$, a starting vector $x_0 \in \mathbb{R}^N$ and a step size $\gamma > 0$. Then the iterations for the so-called *gradient descent* are given by

$$x_{t+1} = x_t - \gamma \nabla_x F(x_t)$$

for $t = 0,\dots,n$. The idea is that if the step size $\gamma > 0$ is small enough, then $F(x_{t+1}) < F(x_t)$.

In the context of machine learning, there is usually a loss function of the form

$$L(g) = \frac{1}{n}\sum_{i=1}^{n} C(x_i, y_i)$$

averaging over training data (x_i, y_i) for $i = 1,\dots,n$, e.g. $c(x,y) = (y - g_\theta(x))^2$ for regression. Due to this additive structure of the loss function, the determination of the derivative $\nabla_\theta L(g_\theta)$

is reduced to the determination of the derivative of $\nabla_\theta C(x_i, y_i)$ for each $i \in \{1, \dots, n\}$. Since the entire training data *batch* is used for the parameter update, the vanilla gradient descent is also often called *batch gradient descent*. Another renaming in the context of machine learning is that of step size. The parameter γ is often referred to as *learning rate* and is a so-called *hyperparameter* for the neural network. If the learning rate is chosen too large, minima may be skipped, while if it is chosen too small, the procedure may not converge. Note that $\gamma = \gamma_t$ can also be chosen depending on the epoch, e.g. a choice motivated from stochastic optimization for γ_t is

$$\sum_{t=1}^{\infty} \gamma_t = \infty \quad \text{and} \quad \sum_{t=1}^{\infty} \gamma_t^2 < \infty.$$

These summability conditions are more theoretical in nature and are needed for stochastic convergence proofs. The iterative reduction of the γ_t is also called *learning rate shrinkage*.

2.1.2 (Mini-Batch) stochastic gradient descent

Gradient descent is the most natural algorithm to use for optimizing the loss function. The deterministic version, however, is computationally quite slow, due to the necessity to calculate the gradient on the whole training set with $n \in \mathbb{N}$ observations. *(Mini-Batch) stochastic gradient descent* (SGD) calculates a gradient on the loss of a mini batch $\mathcal{B} \subset \{1, \dots, n\}$ with size $|\mathcal{B}| < N$. That is the true gradient

$$\nabla_\theta L(g_\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta C(x_i, y_i),$$

is estimated by

$$\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \nabla_\theta C(x_i, y_i).$$

Because the training set is separated randomly into mini batches for each epoch, the gradient for each mini batch is an estimation of the true gradient. Calculating gradients with less data improves computing time and therefore enables faster convergence in time. For this method to work it is essential to guarantee that each mini-batch has a similar distribution, else the method might not lead to convergence. Ways to mitigate this problem include increasing the mini-batch size or considering some data-transformations. With each mini batch we calculate a gradient and an iteration passes. When the whole training set was used during this process, we call it an epoch. So an *epoch* denotes the period of time in which all training data contributed once to update the parameters.[2] Despite SGD performing way better than normal gradient descent, it is not perfect either. It has the problem of "Zigzagging" and does not have much chance to overcome (bad) local minima or saddle points, which are a huge problem in high dimensional problems. *Adam* (adaptive moment estimation, see [6]) further improves SGD by mitigating these problems. Using a first momentum term it is more likely to overshoot saddle points and local minima, while the second momentum term[3] in the denominator, helps the function to adaptively change the learning rate in different directions. Other variations on SGD exist and are used in practise such as SGD with momentum and *RMSprop*.

If the data-set is small and the network is shallow with little weights to train, one could also consider using quasi-Newton methods such as *L-BFGS*. This method is deterministic, but due to random weight initialization, one might still end up in different local minima for each training.

2.2 Backpropagation

The main iterative scheme for optimzing the loss function of a neural network is hence given by

$$\theta^{[t+1]} = \theta^{[t]} - \gamma \cdot \nabla_\theta \left[L\left(g_{\theta^{[t]}}\right) + \lambda \cdot J\left(g_{\theta^{[t]}}\right) \right]$$

for $t = 0, 1, \dots$ and so on. We recall that θ is the vectorization of all weight matrices and biases. Despite the omission of proofs, this Subsection is very math-heavy, but because of its central importance it is indispensable. Due to the recursive structure of the network $g \in \mathcal{F}(L, p)$, the relevant gradients can be calculated systematically, which yields an algorithm called *backpropagation*. We explain the structure in 4 steps:

[2] Note that an epoch with batch gradient descent is given by exactly one iteration, i. e. from t to $t+1$.

[3] second moment refers to elementwise squared gradient

1. The following canonical decomposition of a network $g \in \mathcal{F}(L,p)$ is essential. Let $x^{(0)}(x) = x$, as well as

$$z^{(l)}(x) = v^{(l)} + W^{(l)} \cdot x^{(l-1)}(x),$$
$$x^{(l)}(x) = \sigma\left(z^{(l)}(x)\right),$$

the intermediate transformations between layers. Then $g(x) = W^{(L+1)} \cdot x^{(L)}(x)$.

2. The following quantities are central for the derivative of $C_i = C(x_i, y_i)$ w.r.t the parameters $b^{(l)}$ and $W^{(l)}$ of the neural network $g \in \mathcal{F}(L,p)$. We denote by

$$\delta^l = \frac{\partial C_i}{\partial z^{(l)}}$$

the so-called influence of the l-th layer on C_i.

3. Then one can show the recursion

$$\delta^L = \frac{\partial C_i}{\partial x^{(L)}} \odot \sigma'\left(z^{(L)}\right),$$
$$\delta^l = \left[\left(W^{(l+1)}\right)^\top \delta^{l+1}\right] \odot \sigma'\left(z^{(l)}\right)$$

for $l = L-1, \dots, 1$, where $\odot$ denotes the Hadamard product.[4]

4. Finally, the relationship to the parameter derivatives is given by

$$\frac{\partial C_i}{\partial b^{(l)}} = \delta^l, \quad \frac{\partial C_i}{\partial W^{(l)}} = \delta^l \cdot x^{(l-1)^\top}.$$

Backpropagation thus consists of calculating the derivatives of the parameters for the respective layers of the network and successively updating them, which is possible backwards due to the recursion structure of the network $g \in \mathcal{F}(L,p)$.

2.3 Weight initialization

Favourable starting parameters $\theta^{[0]}$ are crucial due to the non-convexity of the loss function

$$\theta \mapsto [L(g_\theta) + \lambda \cdot J(g_\theta)].$$

There exist local minima of which we don't know which is best for our application. Unfavourable start parameters can lead to gradients that vanish or explode. Weights are usually initialized with random values from a distribution with zero mean, while the biases are initialised to 0 (or a $\varepsilon \ll 1$). As a remedy different starting weights are considered, which usually lead to different end results. Initializing all weights to 0 is problematic, as it could make all neurons to be updated the same way, making symmetry breaking essential. So how to initialise the weights $W^{(l)} \in \mathbb{R}^{p_l \times p_{l-1}}$ best? There exist specific initialization methods such as *Xavier* initialization (see [3]) which aims to reduce covariate shift in other layers. It is given by

$$W_{ij}^{(l)} \sim \mathcal{U}\left[-c_l, c_l\right], \quad \text{where } c_l = \frac{6}{\sqrt{p_l + p_{l-1}}}.$$

Another method is known as *Kaiming* initialization (see [5]) with

$$W_{ij}^{(l)} \sim \mathcal{N}\left(0, \sigma_l^2\right), \quad \text{where } \sigma_l^2 = \frac{2}{p_{l-1}}.$$

This result is obtained after demanding variances of the weight matrices to be approximately 1, even in deep layers. Kaiming initialization in conjunction with the ReLU activation are particularly popular for deep neural networks, i. e. when $L \gg 1$.

Exemplary resulting distributions for the two initialization methods are illustrated in Figure 2 where the weights of a matrix with size 30×30, connecting two layers of size 30, are plotted in a histogram. One can see that both distributions are centred around 0, as well as the uniform and normal distribution assumptions respectively.

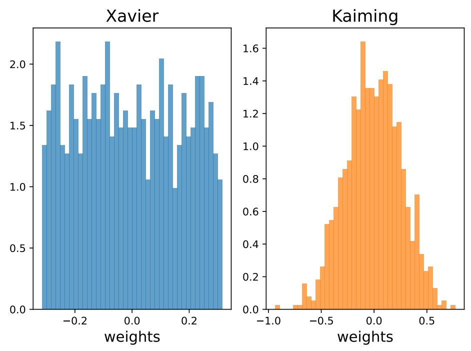

Figure 2. Distribution of weights in an exemplary weight matrix $W \in \mathbb{R}^{30 \times 30}$ with the Xavier intialization (left) and the Kaiming initialization (right).

[4]The Hadamard product of $a, b \in \mathbb{R}^p$ is given by $a \odot b = (a_j b_j)_{j=1,\dots,p}$.

3 Logistic regression

In simple terms, *logistic regression* is, contrary to the name, a generalization of linear regression to binary classification problems. In a binary classification problem, the labels belong to one of only two categories, that is $y_i \in \{0, 1\}$ for all $i = 1, \dots, n$, or, $K = 2$, using the notation from Section 1.

3.1 Model assumption

While linear regression produces continuous values in a potentially infinite range, predicting categories requires rethinking. The idea is to first squeeze the output of a simple linear combination $x^\top \beta$ into the value range (0, 1), e.g. by using the logistic (or sigmoid) function $\sigma : \mathbb{R} \to (0, 1)$ with

$$\sigma(x) = \frac{e^x}{1 + e^x},$$

of which a plot is shown in Section 4, Figure 14.

The intuition behind the 0-1 range is, as with the softmax function, the possibility for interpretation as a probability. The model assumption for logistic regression is

$$\mathbb{P}(y_i = 1 \mid x_i, \beta) = \sigma(x_i^T \beta),$$

and therefore

$$\mathbb{P}(y_i = 0 \mid x_i, \beta) = 1 - \sigma(x_i^T \beta).$$

3.2 Cross-entropy loss

Since $y_i \in \{0, 1\}$ we can summarize the last two formulas to be the likelihood for one sample

$$\mathcal{P}(y_i \mid x_i, \beta) = \sigma(x_i^\top \beta)^{y_i} \left(1 - \sigma(x_i^\top \beta)\right)^{1 - y_i}.$$

Building the product over the training batch, taking the logarithm, and diving by the batch size yields the log-likelihood function

$$L(\beta) = -\frac{1}{n} \sum_{i=1}^{n} \left[y_i \log \sigma\left(x_i^\top \beta\right) + \right. \\ \left. (1 - y_i) \log\left(1 - \sigma\left(x_i^\top \beta\right)\right) \right]$$

which is often referred to as the *cross-entropy loss* function.[5] A non-binary classification scenario with multiple categories $K > 2$ can also be represented by extending the cross entropy loss.

[5]Note the negative sign in the definition of L to formulate a minimization instead of maximization problem.

3.3 Design matrix

Due to the similar model assumption as in linear regressions, we first start with the construction of the design matrix. Note here that since we want to have a non-trivial model, we add a regressor column for the intercept. Thus given training data (x_i, y_i) with $x_i \in \mathbb{R}^k$ for $i = 1, \dots, n$ denote by

$$X = \begin{pmatrix} 1 & x_{11} & \dots & x_{1k} \\ \vdots & \vdots & & \vdots \\ 1 & x_{n1} & \dots & x_{nk} \end{pmatrix} \in \mathbb{R}^{n \times (k+1)}$$

the design matrix. Further we concatenate the vector of labels y_i by

$$y = (y_1, \dots, y_n)^\top \in \mathbb{R}^n.$$

Also in logistic regression, we would like to add an L2 regularization (cf. ridge regression) so that the underlying optimization problem is thus

$$\beta^* = \underset{\beta \in \mathbb{R}^k}{\arg\min} \left[L(\beta) + \lambda \|\beta\|_2^2 \right]$$

with a regularization parameter $\lambda \geq 0$.

3.4 Derivative

In contrast to ordinary least squares or ridge regression, it is not possible to find a closed-form expression for β^*, the maximizer of the log-likelihood function L. Thus, it is necessary to switch to iterative solvers, such as the gradient methods from Section 2. For this to work, the derivative of the loss function $\beta \mapsto L(\beta) + \lambda \|\beta\|_2^2$ w.r.t to the parameters $\beta$is necessary. After a few algebraic transformations we get

$$\frac{\partial}{\partial \beta} L(\beta) = -\frac{1}{n} X^\top \left(y - \sigma\left(X\beta\right)\right) + 2\lambda\beta.$$

An important question to ask is how to get from the still continuous outputs of the sigmoid (logistic) function to categorical predictions for label 0 or 1. Here we use a simple approach, namely a 0.5 threshold. If the output is larger than the threshold, we choose 1, otherwise 0.[6]

[6]Note that $\sigma(x) > 0.5 \iff x > 0$.

4 Data & Results

4.1 Franke's function

We dedicate ourselves again to Franke's function [2], which we have already dealt with intensively in previous projects, and would like to draw a comparison to linear regression methods. As a reminder, Franke's function is defined as $f : \mathbb{R}^2 \to \mathbb{R}$ with

$$\begin{aligned} f(x,y) = & \frac{3}{4}\exp\left\{-\frac{1}{4}\left[(9x-2)^2+(9y-2)^2\right]\right\} \\ & +\frac{3}{4}\exp\left\{-\frac{1}{49}(9x+1)^2+\frac{1}{10}(9y+1)^2\right\} \\ & +\frac{1}{2}\exp\left\{-\frac{1}{4}\left[(9x-7)^2+(9y-3)^2\right]\right\} \\ & -\frac{1}{5}\exp\left\{-\frac{1}{4}\left[(9x+4)^2+(9y-7)^2\right]\right\}. \end{aligned}$$

Figure 3 shows Franke's function on the unit square, which is the (pre-scaling) domain we will fit our models on.

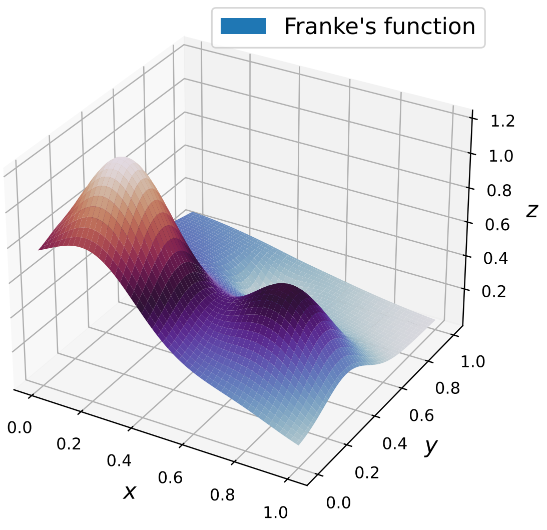

Figure 3. Franke's function on the unit square, that is $x, y \in [0, 1]$. The colors indicate different values in height.

We sample $N \in \mathbb{N}$ uniformly distributed points on the unit square and add white noise $\varepsilon \sim N(0, \sigma^2)$. Figure 4 shows Franke's function with $N = 200$ sampled train and test data points generated by adding noise with $\sigma^2 = 0.1$.

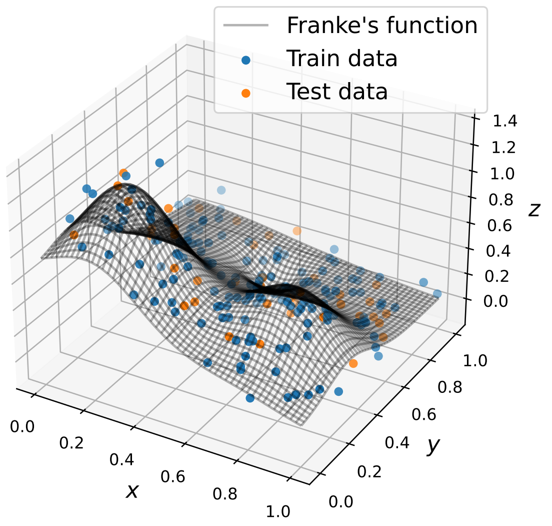

Figure 4. Franke's function on the unit square, as well as $N = 200$ train (blue) and test (orange) data points, generated by adding a noise level of $\sigma^2 = 0.1$.

Unlike the previous project, this time we are using scaling of the Franke's function features because we have seen improvements, especially in the iterative optimization algorithms. Normalization was used, and no transformation to the interval $[0, 1]$, because the data is already contained there.

Closed-form vs. iterative linear regression

First, we look at the viability of our implemented stochastic gradient descent method. To do this, we compare the closed-form solutions we derived for OLS and ridge regression in the previous project against an iterative SGD fit. To do this, we only need the derivative of the loss function, which is given by

$$L(\beta) = \frac{2}{n} X^\top (X\beta - y) + 2\lambda\beta,$$

where $\lambda \geq 0$ is the regularization.[7]

Subsequently we sampled $N = 200$ random and uniform data points in the unit square $[0, 1]^2$, splitted them with a train-test ratio of 0.25, and applied normalization. The hyperparameters are a regularization of $\lambda = 1 \times 10^{-3}$, learning rate $\gamma = 1 \times 10^{-2}$,

[7]The OLS loss is revived by using $\lambda = 0$.

500 epochs, and a batch size of 16. The results are shown in Figure 5, where the loss (MSE) history over the 500 epochs is shown. Closed-form solver do not have a history, such that their test MSEs are shown as horizontal, dotted lines.[8]

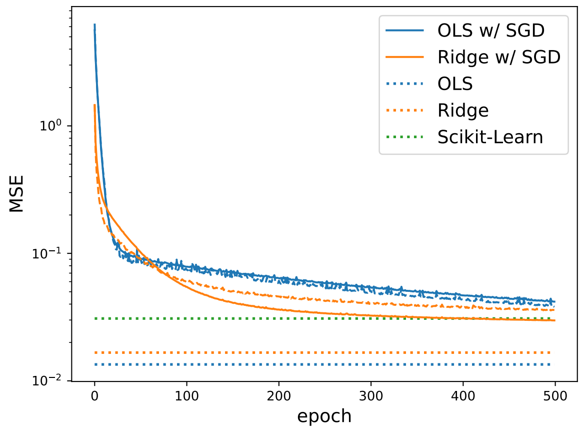

Figure 5. Closed-form and SGD solution loss for OLS (blue) and ridge (orange) regression. Train MSEs are shown as solid lines and test MSEs as dashed lines. The three dotted horizontal lines correspond to test MSEs for the closed-form solvers. Scikit-Learn's (iterative) SGDRegressor test MSE is shown as a green dotted line.

OLS has the best score (lowest MSE), which can be surpassed by Ridge regression after tuning the regularization parameter λ, or having a smaller sample size $n \in \mathbb{N}$. Overall, it can be said that the closed-form methods provide better results than the iterative methods, and also faster in computation time. This behaviour is to be expected: since OLS and Ridge Regression are convex optimization problems, iterative algorithms must in theory reach the global minimum (the closed-form solution), with appropriate step size control $\gamma = \gamma_t$. We are convinced that the loss curves would continue to fall even after 500 epochs if a step size is incorporated that decreases over epochs.

Hyperparameter tuning

The choice of the above parameters is the result of the following analyses. Hyperparameter tuning was established for a ridge model by performing a grid search on the regularization parameter and learning rate. We used 30 linearly spaced learning rates γ between 1×10^{-3} and 1×10^{-2}, as well as 30 linearly spaced regularization parameters λ between 1×10^{-5} and 1×10^{-2}. To reduce computation time each of the $30 \times 30 = 900$ Ridge models were fitted using 50 epochs and a batch size of 128.[9] To make the results more meaningful, each of the 900 parameter combinations were used to fit a Ridge model $N_{SIM} = 10$ times and the resulting test MSEs were averaged (cf. bootstrapping). The resulting test MSE grid is shown in Figure 6.

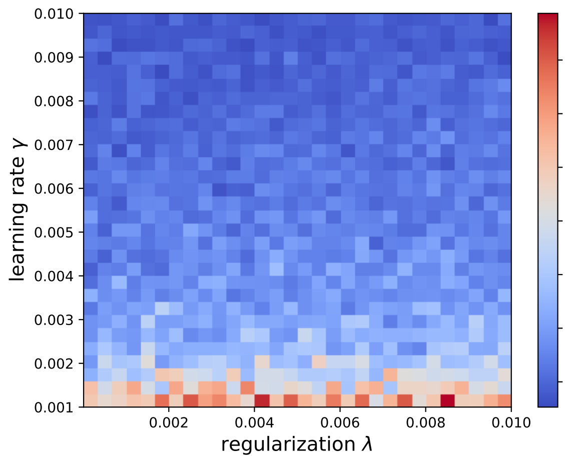

Figure 6. Grid search for the learning rate γ and regularization λ in a ridge regression, each combination trained for 50 epochs and batch size of 128.

One can see a strong influence of the learning rate γ on the test MSE. The learning rate seems to correlate negatively with the loss, although this relationship cannot persist for much larger learning rates $\gamma > 1 \times 10^{-2}$. However, for the regularization parameter λ, the results look relatively the same for a fixed learning rate γ. This can be justified by the fact that $N = 200$ data points are too many for the benefits of regularization to be felt. So the picture suggests mainly the use of a learning rate $\gamma = 1 \times 10^{-2}$.

In another hyperparameter tuning, only the learning rate γ is varied, while the regularization of $\lambda = 1 \times 10^{-2}$ is fixed. 200 logarithmically

[8]For simplicity, Scikit-Learn's (iterative) SGDRegressor is assumed to be a closed-form solver.

[9]Larger batchsizes achieve faster training times due to vectorization in our implementation.

spaced learning rates between 1×10^{-3} and 1×10^{-2} are chosen, and for each $N_{SIM} = 10$ times a ridge model with 100 epochs and batch size 128 is trained. The averaged train and test MSEs are shown in Figure 4.

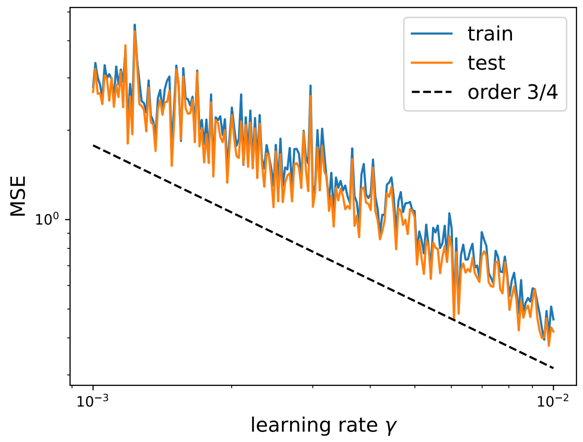

Figure 7. Hyperparameter tuning of 200 logarithmically spaced learning rates. Train MSE shown in blue and test MSE shown in orange. The

As seen in the previous grid Search in Figure 6, the MSE decreases in this range of the learning rate. A reference line with slope $-3/4$ helps to estimate an experimental order of convergence. If we double the learning rate, the train and test MSE is expected to decrease by approximately 75%.

In a third hyperparameter tuning, we examine the number of epochs and the batch size in a grid search. We used every fourth epoch number between 10 and 100, and every fourth batch size between 1 and 100. For each epoch - batch size combination, a ridge model was again trained $N_{SIM} = 10$ times, now with a fixed learning rate $\gamma = 1\times10^{-2}$ and regularization $\gamma = 1\times10^{-3}$. The averaged test MSE grid can be seen in Figure 8.

Roughly speaking, the more epochs one trains, the better; and the smaller the batch size, the better. Of course, the latter means more stochasticity in the iterative optimization. Note that a smaller batch size leads to more parameter updates in the SGD. Coupled with the constant learning rate $\gamma \equiv 1\times10^{-2}$ and the convex loss function of the ridge regression, a lower test MSE is to be expected there.

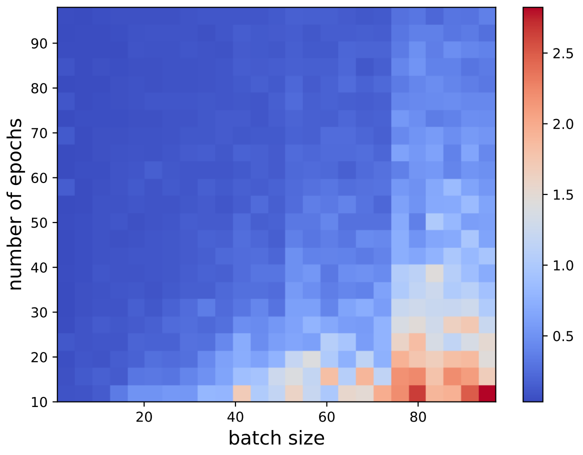

Figure 8. Grid search for the number of epochs and batch size in a ridge regression, each combination trained with a learning rate of $\gamma = 1\times10^{-2}$ and a regularization of $\gamma = 1\times10^{-3}$.

Feedforward neural network

Now we compare linear regression (closed-form as well as iterative) with feedforward neural networks introduced in Section 1. We use a network $g \in \mathcal{F}(L,p)$ with architecture $L = 2$ and $p = (2,50,50,1)^\top$. Further we use a *leaky ReLU* activation with $\alpha = 0.1$ (see Section **??**) and Xavier weight initialization. Thus, the network g has

$$\sum_{l=1}^{L+1} p_l p_{l-1} + \sum_{l=1}^{L} p_l = 2751$$

parameters, i. e. $\theta \in \mathbb{R}^{2751}$, while the linear regression models operates with inhomogeneous polynomial features of degree $d = 5$, that is

$$\frac{1}{2}(d+1)(d+2) = 21$$

parameters, i. e. $\beta \in \mathbb{R}^{21}$. Further, a learning rate of $\gamma = 1\times10^{-2}$, a regularization of $\gamma = 1\times10^{-4}$ are used, while training is performed using 1000 epochs and a batch size of 8. The loss history on a train and validation set against the epochs is shown in Figure 9 down below.

The iterative SGD linear regression methods do not reach the reference of the closed-form solutions OLS and Ridge even after 1000 training epochs with a relatively small batch size of 8. The neural

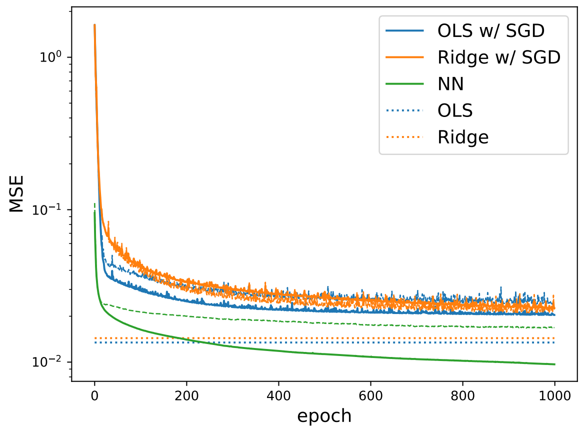

Figure 9. Closed-form and iterative OLS (blue) and ridge (orange) regression, as well as neural network (green) loss history. Train MSEs are shown as solid lines and test MSEs as dashed lines. The two dotted horizontal lines correspond to test MSEs for the closed-form solvers OLS and Ridge.

network g learns, as expected, much faster due to the oversized architecture, but does not reach the minimum of the closed-form solutions either. Although not shown in the figure, the network reaches the train MSE of the closed-form solution after about 600 epochs. This is for us a classical example of overfitting with respect to the number of parameters, so proper hyperparameter tuning for the regularization parameter λ would make sense.

Figures 10 and 11 show the predictive ability of the models. The first Figure 10 shows an actual vs. predicted scatter plot of the three iteratively fitted models OLS, Ridge, and the neural network g corresponding to the loss history in Figure 9. For every model the predictions for the 50 test data points is shown. A perfect model, with test MSE of 0, would produce dots on the dashed black line. Figure 11 shows Franke's function, as well as the prediction for the unit square $[0, 1]^2$ made by the neural network g. Further, the 50 test data points are shown.

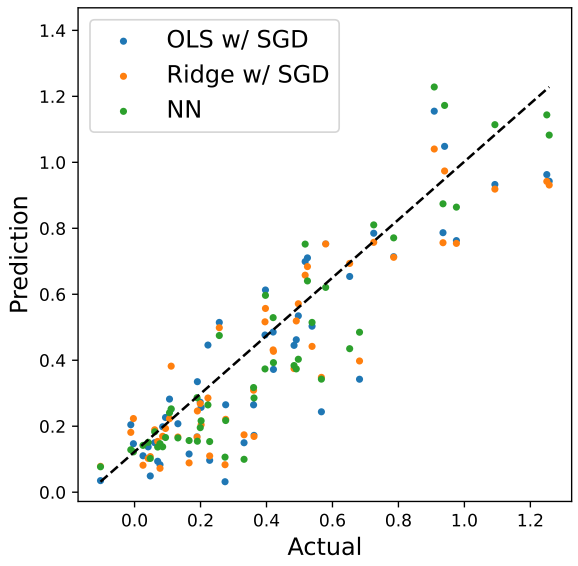

Figure 10. Scatter plot of the ground truths (x-axis) and predictions (y-axis) made by the three iteratively fitted models OLS (blue), Ridge (orange), and neural network (green) on the test set of 50 points.

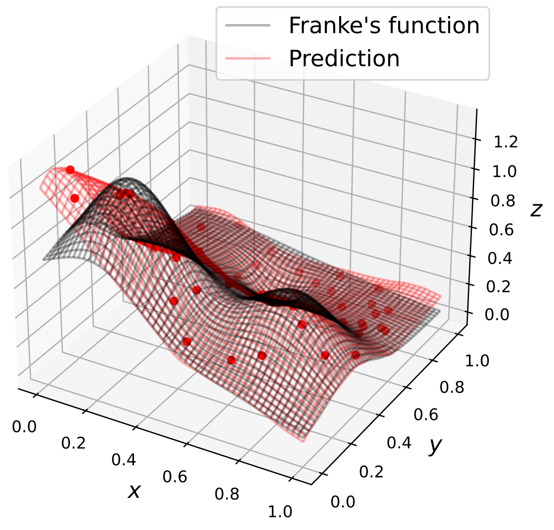

Figure 11. Shown are Franke's function (grey grid), 3D prediction (red grid) of the fitted neural network g, and 50 test data points (red dots).

Hyperparameter tuning

Furthermore, we also performed a grid search for the learning rate γ and regularization λ, and obtained very similar results as with the iterative ridge regression in Figure 6. However, due to long computation times, only 20 linearly spaced learning rates γ between 1×10^{-4} and 1×10^{-2} and

20 linearly spaced regularization parameters γ between 1×10^{-5} and 1×10^{-1} are selected. Each of the $20 \times 20 = 400$ combinations will be used in training a neural network $g \in \mathcal{F}(1, [2, 50, 1]^\top)$ for $N_{SIM} = 10$ times with 50 epochs and batch size 64. The averaged test MSE grid is shown in Figure 12.

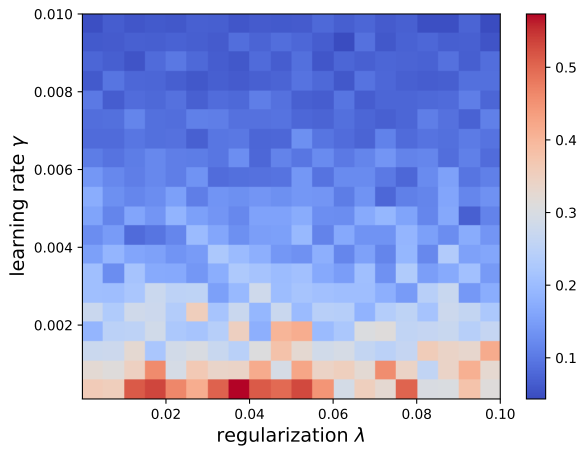

Figure 12. Grid search for the learning rate γ and regularization λ in a neural network $g \in \mathcal{F}(1, [2, 50, 1]^\top)$, each combination trained for 50 epochs and batch size of 128.

Besides the falling test MSE for larger learning rates γ, another area of regularization parameters λ leads to large test MSE when the learning rate is too small.

Linear or polynomial features?

When comparing the performance of the neural network $g \in \mathcal{F}(2, [2, 50, 50, 1]^\top)$ against linear regression, only the (linear) x and y coordinate features of Franke's function were used for the network, i. e. $p_0 = 2$ input neurons. Figure 13 below compares the train and test MSEs for two networks, one trained on the 2 features $g_1 \in \mathcal{F}(1, [2, 50, 1]^\top)$, and one using the 21 polynomial features $g_2 \in \mathcal{F}(1, [21, 50, 1]^\top)$.

The loss with the network g_2 (orange) falls initially much faster, because more information of the data is available (and of course more parameters). After about 200 training epochs, the (linear) network g_1 has caught up with the more complex network and falls even faster than the latter in both,

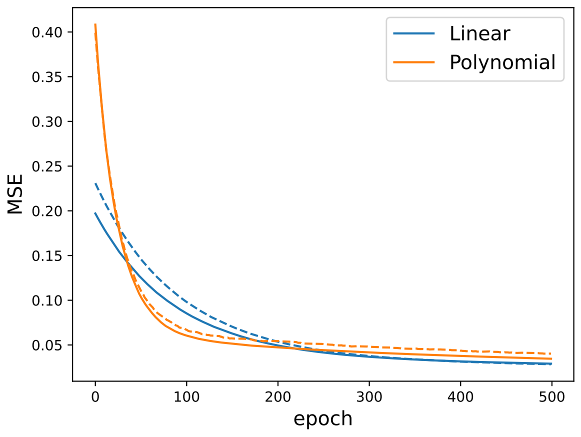

Figure 13. Train (solid) and test (dashed) MSEs for $g_1 \in \mathcal{F}(1, [2, 10, 1]^\top)$ (blue) trained on linear features and $g_2 \in \mathcal{F}(1, [21, 50, 1]^\top)$ (orange) trained on polynomial features.

train and test MSE. Finally, the simpler network g_1 has the better train and test MSE. This could be explained by the fact that the model constructs its own, perhaps more suitable features, while the polynomial features are more of a hurdle for g_2. Here, the motto that the simpler model is to be preferred is apparently valid.

Activation functions

In the following, we will define four activation functions, all of which can be seen in Figure 14.

Sigmoid

$$\sigma(x) = \frac{e^x}{1 + e^x}$$

This function is a classic example of a continuously differentiable activation function. The value range is in $(0, 1)$ which allows the interpretation as an activation rate of a neuron. However, the derivative is very small for many $x \in \mathbb{R}$, so that one sometimes needs a lot of parameter updates to train a neural network.

Hyperbolic tangent (tanh)

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

This activation function also has a sigmoidal shape, but the range of values extends over $(-1,1)$. Sometimes a better performance can be achieved by using this activation, because the function is more similar to the identity, that is a neuron with no input signal is also not activated, i. e. $\tanh(0)=0$.

ReLU

$$\mathrm{ReLU}(x)=\max\{0,x\}$$

This is by far the best known activation function. It produces sparse outputs in an efficient manner. One problem is the non-differentiability in 0, but this can be fixed (or ignored) without much concern. A much bigger problem is given by the *Dying ReLU* phenomenon: If the parameter distribution is unfavorable, some neurons reach a state where no more activation is passed on, because the input signal $x<0$. The following activation function often provides a remedy.

Leaky ReLU

$$\text{Leaky ReLU}(x)=\max\{\alpha x,x\}$$

The structure is the same as in the above ReLU activation, except that the main problem of this is to be bypassed with a parameter $\alpha>0$. Due to the small negative activation for values $x<0$, no more sparse outputs are produced, so that the probability for vanishing gradients almost disappears.

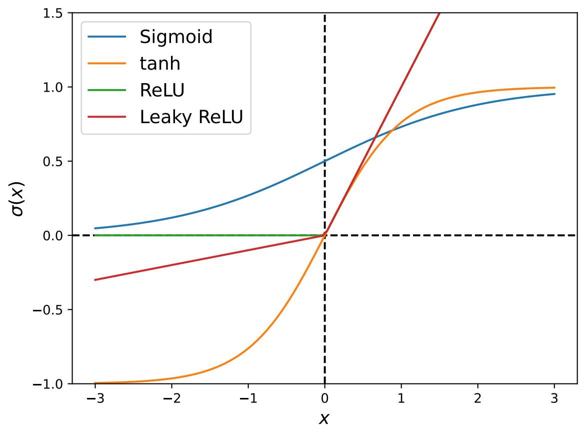

Figure 14. Caption

One thing that activation functions have in common is that they increase monotonically, i.e. $\sigma'(x)\geq 0$ for all $x\in\mathbb{R}$. This is inspired by the stimulus transmission from biological neurons, a neuron fires rather at larger stimulus input. Nevertheless, it can be problematic if the relayed activation is none, i.e. zero, as in the ReLU function $\sigma_{ReLU}(x)$ for $x\leq 0$. In this way, the so-called *vanishing gradient* problem can occur, where weights are no longer updated due to 0 gradients, so that training is no longer possible or effective.

In the following, the four presented activation functions are tested. For this purpose, a neural network with architecture $g\in\mathcal{F}(3,[2,50,50,50,1]^\top)$, which is large for this work, is fitted with each activation. The parameters used are a learning rate of $\gamma=0.1$, regularization $\lambda=1\times10^{-3}$, 200 epochs and a batch size of 16. The loss (MSE) on train and test set is shown in Figure 15. The solid lines represent train MSE, while the dashed lines represent test MSE.

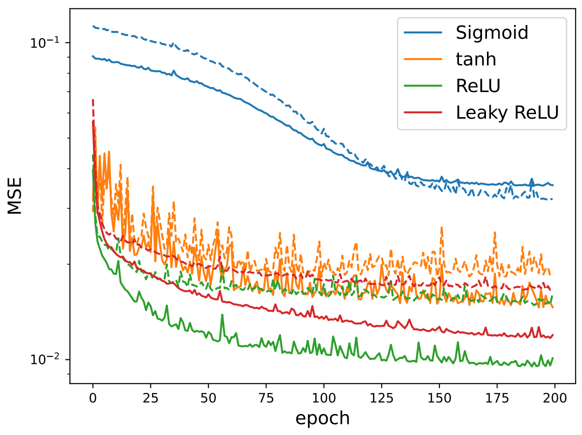

Figure 15. Train (solid) and test (dashed) MSE history for the network $g\in\mathcal{F}(3,[2,50,50,50,1]^\top)$ fitted with each of the four activation functions.

One can see very different behavior at first glance. To explain this, we refer to the recursion of the intermediate variables for the backpropagation algorithm from chapter 2, that is

$$\delta^l=\left[\left(W^{(l+1)}\right)^\top\delta^{l+1}\right]\odot\sigma'\left(z^{(l)}\right)$$

So the derivative of the activation function is essential, because if σ' is very small for many $x\in\mathbb{R}$, the parameters change only very slowly. This can be seen from the MSE curves for the sigmoid activation (blue). Since the derivative of the sigmoid activation is often very small, one would need a larger learning rate $\gamma > 0.1$ to still get acceptable parameter changes. However, if the derivative is too large, oscillations like for the Tanh activation (orange) may occur. There one would have to counteract this with a smaller learning rate $\gamma < 0.1$. It seems that the chosen learning rate of $\gamma = 0.1$ works well for both of the ReLU functions (green and red), and since the mesh is not very deep with $L = 3$, there is no vanishing gradient problem, so that the leaky ReLU (red) activation cannot play its cards. All in all, a parallel tuning of all hyperparameters is necessary, and the network architecture, activation and learning rate have to be coordinated together.

Weight initializations

Another essential parameter besides those mentioned above is that of weight initialization. In Section 2 two were presented, Xavier and Kaiming, for which now a neural network $g \in \mathcal{F}(2, [2, 30, 30, 1]^\top)$ was fitted respectively. We use a learning rate of $\gamma = 1 \times 10^{-2}$, a regularization $\lambda = 1 \times 10^{-5}$, 50 epochs and a batch size of 4. The resulting loss (MSE) curves are shown in Figure 16 below.

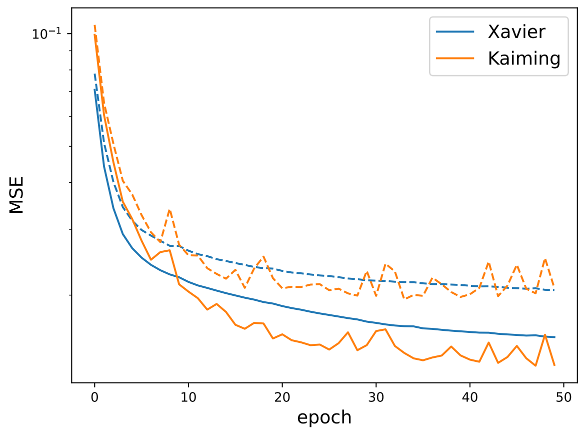

Figure 16. Train (solid) and test (dashed) MSE history for the network $g \in \mathcal{F}(2, [2, 30, 30, 1]^\top)$ fitted with Xavier (blue) and Kaiming (orange) weight initialization.

4.2 Breast cancer data

Now we switch from regression to classification by changing the last entry of the network architecture $p \in \mathbb{N}^{L+2}$ from one neuron $p_{L+1} = 1$ to $p_{L+1} = 2$ two neurons in the output layer. In addition to that we apply the softmax activation introduced in Section 1 to the output layer. We are concerned here with the Wisconsin Breast Cancer data set in which the two categories 0 or 1, the presence of breast cancer, should be predicted. The data consists of $n = 569$ samples, each having $k = 30$ regressor variables. A train-test ratio of 0.25 is used.

Now we would like to measure the performance of neural networks against that of logistic regression in this simple binary classification example. Furthermore, we will consider the respective results of the Scitkit-Learn implementation. We use the cross-entropy loss for optimization as discussed in Section 3. The metric used to monitor performance is the intuitive *accuracy*, which basically just counts how many times the correct category is predicted and divides by the sample size. Given fixed training data (x_i, y_i) for $i = 1, \dots, n$, the accuracy of a model $\mathcal{M}$ is given by

$$\mathrm{acc}(\mathcal{M}) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}_{\mathcal{M}(x_i)=y_i}$$

where $\mathcal{M}(x_i)$ denotes the category predicted by the model $\mathcal{M}$ for the sample x_i.

Throughout, we will use the following parameters. The network to be fitted has a structure of $g \in \mathcal{F}(1, [31, 20, 2])$, while we use a learning rate of $\gamma = 5 \times 10^{-2}$ and a regularization of $\lambda = 1 \times 10^{-3}$. Note the number of neurons in the first layer $p_0 = 31$ since we add an intercept to the $k = 30$ regressors as discussed in Section 3. All the models are trained using 50 epochs with a batch size of 8. Table 1 down below shows the resulting accuracy of the trained models; the neural net g, the logistic regression model, as well as Scikit-Learn's *MLPRegressor*[10] and *LogisticRegression* (orange).

The first three models, i.e. FFNN, logistic regression, and Scikit-Learn's MLPRegressor, all achieve the same accuracy value of about 97.9% . The logistic regression model of Scikit-Learn performs best with an accuracy of about 98.6%. For simple binary classification problems, logistic regression can

[10]multi-layer perceptron

Table 1. Accuracy results on the test data set for the FFNN g and the logistic regression, as well as Scikit-Learn's version.

model $\mathcal{M}$	acc
FFNN	0.979021
Log. regression	0.979021
FFNN (Scikit)	0.979021
Log. regression (Scikit)	0.986014

thus be much more effective than a complex neural network.

Figure 17 shows the accuracy against the training epochs, for the feedforward neural network g (blue) and the logistic regression model (orange). Solid lines represent accuracy on the train set while dashed lines represent accuracy on the test set. The dotted lines are the test accuracies of Scikit-Learn's MLPRegressor (blue) and LogisticRegression (orange) respectively.

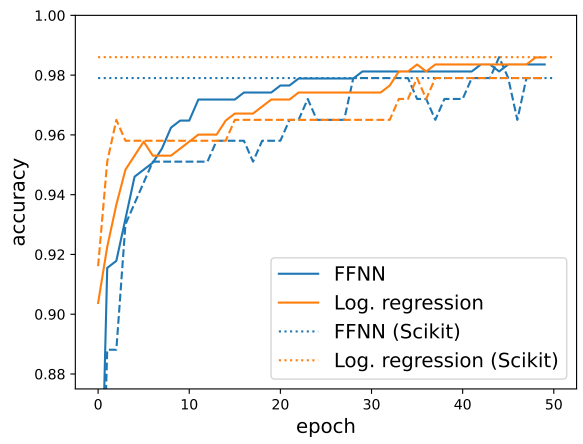

Figure 17. Accuracies during the training process for the FFNN g (blue) and the logistic regression model (orange). Solid (dashed) lines represent accuracy on the train (test). The dotted lines are the test accuracies of Scikit-Learn's models.

Finally, we will address the question of how the number of neurons p_l in a layer l affects the accuracy. For this purpose, we use neural networks of architecture

$$g_i \in \mathcal{F}(1, [31, 2i, 2])$$

for $i = 1, \ldots, 9$. Each of these networks is trained $N_{SIM} = 10$ times in 100 epochs with a batch size of 32, using a learning rate of $\gamma = 5 \times 10^{-2}$ and a regularization of $\lambda = 1 \times 10^{-4}$. The averaged train (blue) and test (orange) accuracies are shown as a function of the number of neurons p_l in Figure 18.

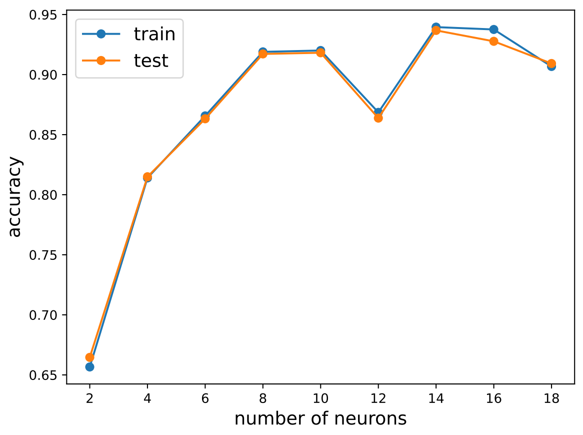

Figure 18. Accuracy against the number of neurons $p = 2i$ where the neural net $g \in \infty, [\ni\infty, \in\rangle, \in]$ is fitted for $i = 1, \ldots, 9$. The train (test) accuracy is shown in blue (orange).

As expected, accuracy increases with the number of neurons in the hidden layer. The Figure also shows that 8 neurons are sufficient to achieve an average accuracy of above 90% on the test data set.

4.3 Digits MNIST

In this Subsection we consider the MNIST[11] dataset of handwritten digits which is commonly and widely used for training and testing in the field of machine learning (see [1]). The dataset we use is available in Scikit-Learn's datasets module and consists of $n = 1797$ handwritten digits $x_i \in \{0, 1, \ldots, 16\}^{8\times 8}$ paired with hand-labeled labels $y_i \in \{0, 1, \ldots, 9\}$ for $i = 1, \ldots, n$. Figure 19 down belows shows 18 examples for handwritten images paired with their true label, that is (x_i, y_i).

The column-wise vectorization of the images $x_i \in \{0, 1, \ldots, 16\}^{8\times 8}$ suggests the use of $8\times 8 = 64$ input neurons. Thus the neural network

$$g \in \mathcal{F}(3, [64, 10, 10, 10, 10]^\top)$$

[11]Modified National Institute of Standards and Technology database

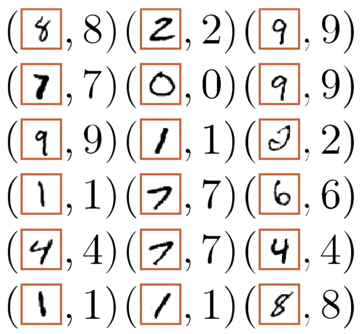

Figure 19. 18 training data samples (x_i, y_i) with $x_i \in \{0, 1, \dots, 16\}^{8\times 8}$ and $y_i \in \{0, 1, \dots, 9\}$.

was trained using the tanh activation, a learning rate of $\gamma = 1$, regularization $\lambda = 1 \times 10^{-4}$, 500 epochs with a batch size of 32. The train (blue) and test (orange) accuracy during training is shown in Figure 20 down below. The network achieves a precision of 93.2% percent on the test data set, while Scikit-Learn's LogisticRegression achieves 97.2%.

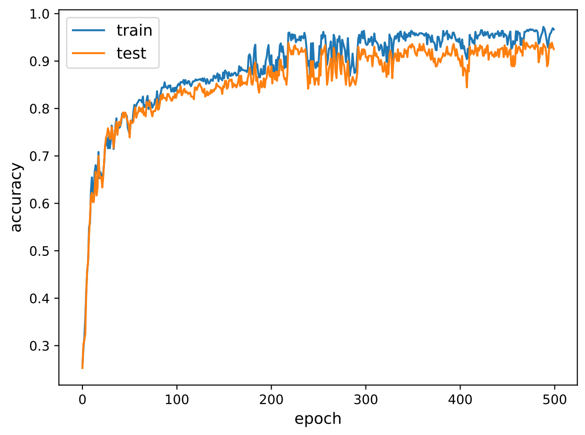

Figure 20. Train (blue) and test (orange) accuracy history for the neural network $g \in \mathcal{F}(3, [64, 10, 10, 10, 10]^\top$

4.4 Fashion MNIST

Last, we would like to try a slightly more complicated classification task with our feedforward neural networks. For this, we again use an MNIST dataset, namely the Fashion dataset from Zalando (see [9]). As with the handwritten digits, we are concerned here with $n = 60000$ images $x_i \in \{0, \dots, 255\}^{28\times 28}$ of garments and associated labels belonging to one of 10 classes $y_i \in \{0, \dots, 0\}$. The number labels figure is given by Table 2.

Table 2. Label-description mapping for the labels.

Label	Description
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

Again, column-wise vectorization suggests the use of $28 \times 28 = 784$ input neurons. Thus the neural network

$$g \in \mathcal{F}(1, [784, 50, 10]^\top)$$

was trained using the tanh activation, a learning rate of $\gamma = 5 \times 10^{-2}$, regularization $\lambda = 1 \times 10^{-3}$, 100 epochs with a batch size of 128. The train (blue) and test (orange) accuracy during training is shown in Figure 21 down below. The network achieves a precision of 76.3% percent on the test data set, while Scikit-Learn's LogisticRegression achieves 79.7%.

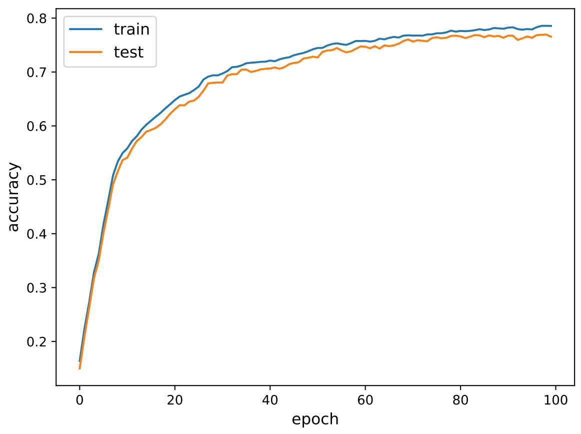

Figure 21. Train (blue) and test (orange) accuracy history for the neural network $g \in \mathcal{F}(1, [784, 50, 10]^\top$

5 Discussion

After we have put neural networks, as well as the necessary iterative optimization algorithms, on a mathematically secure foundation, we have drawn a performance comparison to ordinary least squares (OLS) and ridge regression using Franke's function. In particular, the stochastic gradient method was also used for the linear regression methods, to establish iterative versions of the two regression methods. To be more precise, we used a (inhomogeneous) polynomial regression of degree 5, which is the best model as shown by earlier works.

For regression problems, the closed form OLS regression method still proved to be the best, closely followed by ridge regression. The neural network performs better than the iterative forms of OLS and Ridge, but also takes longer to train. We have seen that the neural networks with ReLU activation function is best suited for our regression problem, but only if it is equipped with the right learning rate. Due to the perhaps too many 200 data points, i. e. 50 test data points, regularization did not pay off very well.

The classification experiments have shown that logistic regression is the most appropriate. However, this is only due to the relatively simple data sets and cannot be generalized. For complicated classification tasks, sophisticated neural network architectures are superior to a logistic regression. The superiority of Sicikit-Learn's LogisticRegression on the MNIST datasets digit and fashion over our neural networks is most likely due to a not ideal implementation from our side.

References

[1] Li Deng. "The mnist database of handwritten digit images for machine learning research". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[2] Richard Franke. *A Critical Comparison of Some Methods for Interpolation of Scattered Data.* Calhoun, 1979. URL: https://calhoun.nps.edu/handle/10945/35052.

[3] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics.* 2010.

[4] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning.* Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[5] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *CoRR* abs/1502.01852 (2015). arXiv: 1502.01852. URL: http://arxiv.org/abs/1502.01852.

[6] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2017. arXiv: 1412.6980 [cs.LG].

[7] Warren S McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[8] *Own work, Derivative of File:Artificial neural network.svg.* Glosser.ca, 2021. URL: https://commons.wikimedia.org/w/index.php?curid=24913461.

[9] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms.* Aug. 28, 2017. arXiv: cs.LG/1708.07747 [cs.LG].