Artificial Neural Networks

FYS-STK4155: Project 2

Ghadi Souheil Al Hajj, Maria Nareklishvili & Ulrik Seip
github.com/MariaRevili/FYS-STK4155

November 11, 2020

Abstract

Using synthetic and real MNIST handwritten digit database (enumerated from 0 up to and including 9) we propose the regression and the classification analysis in this paper. We implement neural networks and the multinomial logistic regression for the classification analysis, whereas the regression study comprises of a comparative investigation of neural networks, the linear and the ridge regression. The standard stochastic gradient descent algorithm solves for the unknown parameters of interest. To measure the performance of these methods, we employ the *mean squared error* (MSE) and the *accuracy* metrics. Our findings indicate that in the regression settings, the standard linear regression outperforms the other methods (0.8% MSE). In the classification analysis neural networks leads to marginally better prediction performance over the traditional methods (97.2% accuracy score), but at a much higher computational cost. Finally, we discuss the possible implications and the results thereof.

I. INTRODUCTION

Regression methods have been excessively popular in the field of statistics for many decades. These methods capture patterns in the unknown data and predict the quantities of interest. In particular, linear regression, with all its variants, has been a traditional workhorse in practically all scientific fields due to their simplicity and strong predictive performance. However, the streaming complex data has made the task of prediction rather difficult. *Artificial neural networks* (ANN) is an extension of the linear methods specifically designed to fit complex and ill-defined structures. Originally the method was intended to model how the human brain processes visual data and learns to recognize objects. However, since then, it has been expanded and adapted to study various complex problems. Examples of complex structures where ANN can potentially outperform linear methods include image recognition with highly non-linear pixel structures (Zheng et al. (2017)), designing solar systems (Kalogirou et al. (2014)), and predicting supreme court decisions (Sharma et al. (2015)).

The present paper aims to compare linear methods to artificial neural networks in, both, regression and classification settings. In particular, for regression, we compare conventional linear regression, ridge regression and ANN's by illustrating the prediction performance of each model. For classification tasks, we illustrate the predictive performance of neural networks and the *softmax regression* method (a.k.a the multinomial logistic regression).

To solve for the parameters of interest, we use the ordinary least squares (OLS) and the stochastic gradient descent (SGD) algorithms. The SGD belongs to the family of gradient descent algorithms and aims to minimize the loss/-cost under an iterative updating scheme.

The data we use in this project for the regression analysis consists of the target/outcome variable generated by the *franke* function. Whereas the classification task is based on the real MNIST handwritten digit database. The data is split into three parts: *train, validation* and *test* sets. We use the first partition to solve for the optimal parameters of interest. The second part is best suited for finding the best possible combination for the unknown *hyper-parameters*. Finally, on the test data, we measure the prediction performance of each method and compare.

The results in this paper inform that the standard linear regression outperforms the other methods (with the MSE of 0.8% on the test data). On the contrary, classification results imply the superiority of neural networks relative to the multinomial logistic regression (with the accuracy score of 96.4% on the test data). However, while neural networks perform better than logistic regression, this comes at the expense of a higher computational cost.

The report is split into sections designed to raise subjects for discussion and lead the reader to conclusions similar to those presented. In the *Theory* section we describe the methods employed in the project mathematically. Then in the *Data* section we illustrate the synthetic data generating process as well as the real data. The *Results* section depicts the estimation results for different parameters of interest (i.e. layers, neurons, stochastic gradient descent decay, learning rate etc). Finally, the choice of the methods and the implications are discussed.

II. REGRESSION STUDY

The ANN intuitively can be thought of as the extension of the simple linear regression methods. The Project 1 by Saanum et al. (2020) discusses linear regression models in-depth, however, a short reminder of the notation is appropriate.

A. Linear Regression

Linear regression is a conventional statistical learning method that approximates an outcome variable as a linear combination of some set of features/inputs (e.g. pixels). For each item i, the outcome of interest can be written as follows:

$$\widehat{y}_i = \sum_{j=0}^{p-1} x_{i,j}\boldsymbol{\beta}_j \tag{1}$$

This is a system of linear equations which involves predicting an outcome $\widehat{y}$ for a given matrix of features $\mathbf{X} = [x_0, x_1, \ldots, x_{p-1}]$ where p is the number of inputs in this matrix. Re-writing this in terms of the linear system of equations yields:

$$\begin{aligned}
\widehat{y}_0 &= \beta_0 x_{0,0} + \beta_1 x_{0,1} + \ldots + \beta_{p-1} x_{0,p-1} \\
\widehat{y}_1 &= \beta_0 x_{1,0} + \beta_1 x_{1,1} + \ldots + \beta_{p-1} x_{1,p-1} \\
&\vdots \\
\widehat{y}_{n-1} &= \beta_0 x_{n-1,0} + \beta_1 x_{n-1,1} + \ldots + \beta_{p-1} x_{n-1,p-1}
\end{aligned}$$

where n is the *sample size* of the data. In the case of MNIST handwritten digit database, the number of pixels. This system of equation can be written in a matrix-vector multiplication:

$$\widehat{\mathbf{y}} = \mathbf{X}\boldsymbol{\beta} \tag{2}$$

The primary goal is to solve equation (2) with respect to unknown parameters, $\boldsymbol{\beta}$.

B. Ridge Regression

In a high-dimensional setup where the number of predictors is relatively large (and possibly highly correlated), the estimated coefficients $\hat{\boldsymbol{\beta}}$ exhibit high variability. Such coefficients may end up *overfitting* the outcome variable we seek to model: Seeking to explain away variability in the training set by adopting (possibly extreme) coefficient values, our estimated model may generalize poorly to the test data, with the coefficients being highly tailored to the training data. Hoerl and Kennard (1976) introduced the additional term λ which regularizes (i.e. penalizes) large values of $\boldsymbol{\beta}$ and thus overcomes the issues associated with the standard linear regression:

$$\widehat{y}_i = \sum_{j=0}^{p-1} x_{i,j}\boldsymbol{\beta}_j + \lambda \sum_{j=1}^{p-1} \beta_j^2 \tag{3}$$

λ is a *regularization/shrinkage* parameter. In general the intercept is omitted from the regularization. Re-writing (3) in a matrix-vector notation leads to the following expression:

$$\widehat{\mathbf{y}} = \mathbf{X}\boldsymbol{\beta} + \lambda\|\boldsymbol{\beta}\|^2 \tag{4}$$

As in the ordinary linear regression, the primary aim is to find unknown parameters of interest, β, and the additional *hyper-parameter*, λ.

C. Ordinary least squares

Several different approaches can be considered to solve for the unknown parameter vector, β. We employ two methods in this paper. In particular, we make use of the ordinary least squares (OLS) method and the stochastic gradient descent (SGD) algorithm for the linear and ridge regressions; and the SGD for neural networks and the softmax regression. The project 1 provides an overarching description of the OLS method (see Saanum et al. (2020)), but to remind, we briefly outline the main points.

The Ordinary Least Squares regression method (OLS) is the simplest and most intuitive to implement. This scheme involves minimizing a cost function, something which is generally the goal of most linear or nonlinear methods. The cost function is in general a measure of how well a calculated prediction $\widehat{y}$ performs in relation to the 'true' data y. In the case of OLS, the cost function is the *Mean Squared Error* (MSE) (Hastie and Tibshirani (2001)):

$$C(\boldsymbol{\beta}) = \|\mathbf{y}-\widehat{\mathbf{y}}\|_2^2 = \|\mathbf{y}-\mathbf{X}\boldsymbol{\beta}\|_2^2 = \sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j}^{p}X_{ij}\beta_j\right)^2 = (\mathbf{y}-\mathbf{X}\boldsymbol{\beta})^T(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}) \tag{5}$$

Minimizing (5) with respect to the parameter vector leads to the least squares solution:

$$\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}}\{C(\boldsymbol{\beta})\} \tag{6}$$

$$0 = \frac{\partial C(\boldsymbol{\beta})}{\partial\boldsymbol{\beta}} = \frac{\partial}{\partial\boldsymbol{\beta}}(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})^T(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})$$

$$-2\mathbf{X}^{\mathbf{T}}(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}) = 0 \Rightarrow$$

$$\boldsymbol{\beta}^{\text{OLS}}_{\text{estimator}} = \left(\mathbf{X}^{\mathbf{T}}\mathbf{X}\right)^{-1}\mathbf{X}^{\mathbf{T}}\mathbf{y}$$

D. Gradient Descent

Almost every question of interest in machine learning and statistics starts with the data, X and y, a model $f(\beta)$, which depends on the parameters β and a cost/error function $C(X, f(\beta))$ that allows us to evaluate how well the model $f(\beta)$ approximates the true outcome of interest y. As mentioned above, the primary goal is to find unknown parameters β by minimizing the cost function. Ideally, we would like to have an analytic solution for β, however this is not possible in many cases. We therefore adopt numerical approximation methods, such as gradient descent family of the algorithms to minimize the cost function.

The Steepest descent

The basic underlying idea of the gradient descent algorithm is rather intuitive. We iteratively update the initial 'guess' for the parameters by taking derivatives with respect to the given cost function.

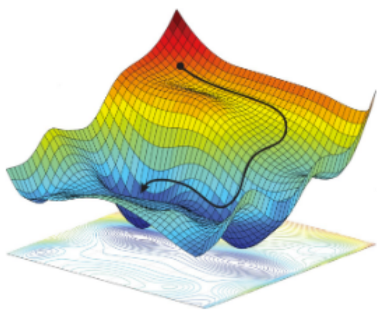


FIG. 1. Cost function $C(\beta)$ where the black line illustrates the iterative updating of the initial guess for the parameters. *Source:* Azizan and Hassibi (2018)

Figure 1 depicts the cost function with respect to the two-dimensional parameter space. First, we initialize parameters β with some small values. After plugging these values in the cost function, we can see that they are located (i.e. the black dot on the function) far from the *global* minimum. How to update the initial guess? Perhaps the intuitive answer is to find the change (i.e. slope) of the cost function at each given point. If the change is positive, the initial parameters are located in the *convex* part of the cost function, thus we need to subtract this change from the parameters. This process continues until we (possibly) reach the global minimum of this cost function. The change (slope) is given by the derivatives of the cost function with respect to the parameters. Hence, mathematically we can express this scheme as follows:

$$\beta_{k+1} = \beta_k - \eta_k\nabla C(\beta_k), \quad \eta_k > 0 \tag{7}$$

for the *learning rate* η_k small enough,

$C(\beta_{k+1}) \leq C(\beta_k)$. This means that for a sufficiently small η_k we are always moving towards smaller function values, i.e the minimum. k simply counts the number of iterations, i.e. the number of times we subtract the derivative from the cost function.

Ideally the sequence $\{\beta_k\}_{k=0}^{K}$ converges to a *global* minimum of the function C. In general we do not know if we reach a global or a local minimum. In the special case when C is a *convex function, all local minima are also global minima*, so in this case gradient descent can converge to the global solution. The advantage of this scheme is that it is conceptually simple and straightforward to implement.

However several important limitations characterize this method:

- The GD algorithm strongly depends on the initial guess of the parameters. In practice, the function of interest is not always fully convex. It normally represents a mix between a convex and concave shapes, in addition to the *saddle* points (i.e. neither the local or the global minima). If the initial guess falls in the local minimum region of the cost function, the algorithm will be stuck at this point.

- The gradient is a function of $\mathbf{x} = (x_1, \cdots, x_n)$; that is, the gradient is computed for all the items (observations) at the same time. This makes the iterative scheme computationally expensive to implement.

Stochastic Gradient Descent

Stochastic gradient descent (SGD) and variants thereof address some of the shortcomings of the Gradient descent method discussed above.

The primary idea of SGD comes from the fact that the cost function can be written as a sum of the cost functions over n datapoints:

$$C(\beta) = \sum_{i=1}^{n} c_i(\mathbf{x}_i, \beta). \tag{8}$$

This in turn means that the gradient can be computed as a sum over i-gradients

$$\nabla_\beta C(\beta) = \sum_{i}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta). \tag{9}$$

Now, stochasticity/randomness is introduced by taking the gradient only on a subset of the data called *minibatches*. If there are n datapoints and the size of each minibatch is M, there will be n/M minibatches. We denote these minibatches by B_k where $k = 1, \cdots, n/M$.

As an example, suppose we have 10 observations $(\mathbf{x}_1, \cdots, \mathbf{x}_{10})$ and we choose to have $M = 5$ minibathces, then each minibatch contains two observations. In particular, we have $B_1 = (\mathbf{x}_1, \mathbf{x}_2), \cdots, B_5 = (\mathbf{x}_9, \mathbf{x}_{10})$. Note that if we choose $M = 1$ we have just a single batch with all datapoints (i.e. the standatd GD algorithm) and on the other extreme, we may choose $M = n$ resulting in a minibatch for each datapoint, i.e $B_k = \mathbf{x}_k$. In this study we use the latter.

The aim is now to approximate the gradient by replacing the sum over all observations with a sum over one of the randomly chosen minibatches in each gradient descent step (possibly after reshuffling the data).

$$\begin{aligned}\nabla_\theta C(\beta) &= \sum_{i=1}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta) \\ &\to \sum_{i \in B_k}^{n} \nabla_\beta c_i(\mathbf{x}_i, \beta).\end{aligned} \tag{10}$$

Thus a gradient descent step now looks as follows:

$$\beta_{k+1} = \beta_k - \eta_k \sum_{i \in B_k}^{n} \nabla_\theta c_i(\mathbf{x}_i, \beta) \tag{11}$$

where k is picked at random with the equal probability from the interval $[1, n/M]$. An iteration over the number of minibatches n/M is commonly referred to as an *epoch*. Thus it is typical to choose a number of epochs and for each epoch iterate over the number of minibatches.

When we assume the cost function with L_2 norm regularization (i.e. as in the ridge regression), the stochastic gradient descent becomes:

$$\beta_{k+1} = \beta_k - \eta_k \sum_{i \in B_k}^{n} \nabla_\theta c_i(\mathbf{x}_i, \beta) - \lambda \beta_k \tag{12}$$

where λ is the regularization (i.e. shrinkage) hyper-parameter.

Taking the gradient only on a subset of the data has two important advatanges. First, it introduces randomness which decreases the chance that our optimization scheme gets stuck in a local minima. Second, if the size of the minibatches are small relative to the number of observations ($M < n$), the computation of the gradient is much cheaper since we sum over the observations in the k-th minibatch and not all n datapoints.

A natural question to ask is how many iterations do we need to converge to the minimum? A possibility is to evaluate the cost function after a number of iterations and stop at iteration for which we get the lowest value. Another way to check is to observe gradients after a given number of epochs. If gradient is close to zero, then we are sufficiently close to the minimum.

Another approach is to let the step length η depend on the number of epochs in such a way that it becomes very small after a reasonable time such that we do not move at all. In this paper we evaluate the cost function and stop the iterations when the value of the cost function increases.

E. Neural Networks

Artificial neural networks (sometimes just neural networks) are computational models with the ability to *learn* from examples it is shown. The structure of the networks are inspired by biological networks constituting animal brains. Artificial neural networks fall under the category of machine learning—a subfield of artificial intelligence—and we will, in the following, expose the precise mechanism of the model learning.

Such neural networks can be created in numerous ways, but we will focus exclusively on the most common architecture, namely *multilayer perceptrons* (MLP). The MLP neural networks are built from *layers* of connected *neurons.* In the artificial network, an input value (possibly a vector) is fed into the network model and then propagated through the layers, being processed through each neuron in turn. We will deal only with *feed forward* ANNs, meaning information always flows through the net in one direction only—essentially there are no loops. The entire ANN produces an output value (possibly a vector), which means we can think of it as a complicated function $\mathbb{R}^n \mapsto \mathbb{R}^m$. As we will see, it is possible to write down a closed form expression and to calculate the gradient of the entire cost function with respect to the parameter vector.

Neurons and layers

A neuron is simply a mathematical function for propagating information through the network. Inspired by biological neurons, the artificial neuron "activates" if it is stimulated by a sufficiently strong signal. The artificial neuron receives a vector of input values $\mathbf{x}$. If the neuron is part of the very first hidden layer (denoted by l which is simply a connection of neurons), the input is simply the input (feature) value(s) to the NN. If one or more layers preceded the current one, $\mathbf{z^1}$ is a vector of outputs from the neurons in the previous layer.

The neuron is connected to the previous layers' neurons, and the strength of the connection is represented by a vector of weights, $\mathbf{w}$. Let us now consider a neuron which we will label by the index k. The output from neuron i (of the preceding layer), z_i, is multiplied by the weight corresponding to the i—k connection, w_i. The combined weight vector multiplied by the input vector gives part of the total activation of the neuron,

$$\begin{aligned} \text{First layer: } &\sum_{i=1}^{N^1} w_i x_i = \mathbf{w}^T\mathbf{x} = z^1 \qquad (13) \\ \text{Second layer: } &\sum_{i=1}^{N^2} w_i z_i^1 = \mathbf{w}^T\mathbf{z^1} = z^2 \\ &\vdots \\ \text{Output layer: } &\sum_{i=1}^{N^{l-1}} w_i z_i^{l-1} = \mathbf{w}^T\mathbf{z^{l-1}} = a \end{aligned}$$

Note that here a is the final predicted output of a particular neuron. This notation is introduced for convenience. Also the number of neurons can differ in each subsequent layer, therefore, the total number of neurons —N^l is indexed by a corresponding layer l. The remaining part is known as the bias, b_k. This is a single real number. There is one for each neuron, and it acts as modifier making the neuron more or less likely to activate independently of the input.

The total input is passed to an activation (or transfer) function, which transforms it in some specified way, yielding the neuron *output* $\hat{z}_k$. This in turn becomes input for the neurons in a subsequent layer.

Various different activation functions f are used for different purposes. The function may be linear or non-linear, but should vanish for small inputs and *saturate* for large inputs. Numerous different transfer functions are in popular use today, and we will outline them shortly after.

In total, the action of a single neuron k in layer l can be written as follows:

$$\text{input} \;\to f\left(\mathbf{w}^T\mathbf{z} + b\right) = z_k^l \;\to\; \text{output}. \quad (14)$$

A schematic representation of a layer consisting of three artificial neurons in a fully connected ANN is shown in Figure 2.

Activation functions

Without introducing the functions that transform the original input, i.e. $f_l(x) = x$ for all layers l, the transformation would be linear. To capture possible non-linear patterns in the data, we transform the input by a large set of *activation* functions. These functions are required to be continuous and differentiable, in order for the SGD scheme to work.

We outline part of the activation functions that we employ in ANNs. The simplest possible activation, the identity transformation $f_\mathrm{I}(x) = x$, is commonly used for the output layer in regression settings. A simple step function, $f_\mathrm{H}(x)$, with

$$f_\mathrm{H}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}, \quad (15)$$

is sometimes used in the output layer of classification networks, but seldom in hidden layers because of the vanishing gradient making it impossible to train with backpropagation. The sigmoid function,

$$f_\mathrm{S}(x) = \frac{1}{1+\mathrm{e}^{-x}}, \quad (16)$$

is commonly used for activating hidden layers along with the similar hyperbolic tangent function:

$$f_\mathrm{t}(x) = \tanh(x) = \frac{\mathrm{e}^x - \mathrm{e}^{-x}}{\mathrm{e}^x + \mathrm{e}^{-x}}. \quad (17)$$

The tangent activation is a simple rescaling of the sigmoid, with $f_\mathrm{t}(x) = 2f_\mathrm{s}(2x) - 1$.

Simpler than the sigmoid, the family of activations known as *rectifiers* consist of piecewise linear functions which are popular nowadays. The basic variant, the rectified linear unit (ReLU) is defined as

$$f_\mathrm{ReLU}(x) = \max(0, x), \quad (18)$$

and is popular mostly because of its application to training *deep* (many, large layers) neural networks. Several of the most well-known variants of the ReLu are the leaky ReLU,

$$f_\text{leaky ReLU}(x;\alpha) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0, \end{cases} \quad (19)$$

and the exponential linear unit (ELU)

$$f_\mathrm{ELU}(x;\alpha) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(\mathrm{e}^x - 1) & \text{if } x < 0. \end{cases} \quad (20)$$

Backpropagation

For every complete iteration of the feed forward process, i.e. the transformation from input to output, we also perform an iteration of back propagation. Here a cost function is applied to $\hat{y}$, and the weights are tuned through the stochastic gradient descent method.

The process starts out by calculating the cost function and its derivative w.r.t. the weights in the output layer W^L,

$$\begin{aligned} \frac{\partial C(W^L)}{\partial w_{jk}^L} &= \frac{\partial C(W^L)}{\partial a_j^L}\left[\frac{\partial a_j^L}{\partial w_{jk}^L}\right] \\ &= \frac{\partial C(W^L)}{\partial a_j^L}\left[\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L}\right] \\ &= \frac{\partial C(W^L)}{\partial a_j^L} f_L'(z_j^L) a_k^{L-1}, \end{aligned} \quad (21)$$

where we used that (note $y_j = a_j^L$, i.e. the activations of the final layer)

$$\begin{aligned} \frac{\partial C(W^L)}{\partial a_j^L} &= \frac{\partial}{\partial a_j^L}\left[\frac{1}{2}\sum_{i=1}^{N_\mathrm{O}}(a_j^L - t_j)^2\right] \\ &= a_j^L - t_j, \end{aligned} \quad (22)$$

and

$$\begin{aligned} \frac{\partial z_j^L}{\partial w_{jk}^L} &= \frac{\partial}{\partial w_{jk}^L}\left[\sum_{p=1}^{N_L} w_{jp}^L a_j^{L-1} + b_j^L\right] \\ &= a_j^{L-1}. \end{aligned} \quad (23)$$

We define the quantity in Eq. (21) (apart from a_k^{L-1} as δ_j^L, meaning $\partial C/\partial w_{jk}^L = \delta_j^L a_k^{L-1}$. Applying the chain rule to δ_j^L yields the derivative of the cost function w.r.t. the output layer biases

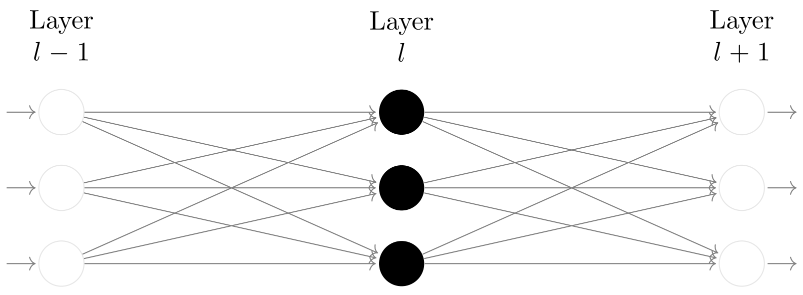


FIG. 2. Schematic representation of a single ANN layer. Each neuron of the layer indexed l is connected from behind to all neurons in layer $l-1$. The connection weights can be organized into a matrix, W^{l-1}, and the action of layer l can be succinctly stated as $f(W^l\mathbf{z}^{l-1}+\mathbf{b}^l)$.

as

$$\delta_j^L = \frac{\partial C(W^L)}{a_j^L}\frac{\partial f^L}{\partial z_j^L} = \frac{\partial C(W^L)}{a_j^L}\frac{\partial a_j^L}{\partial z_j^L}$$
$$= \frac{\partial C(W^L)}{\partial z_j^L} = \frac{\partial C(W^L)}{\partial b_j^L}\frac{\partial b_j^L}{\partial z_j^L} \tag{24}$$
$$= \frac{\partial C(W^L)}{\partial b_j^L}, \tag{25}$$

where we utilized the fact that

$$\frac{\partial b_j^L}{\partial z_j^L} = \left[\frac{\partial z_j^L}{\partial b_j^L}\right]^{-1}$$
$$= \left[\frac{\partial}{\partial b_j^L}\sum_{i=1}^{N_{L-1}} w_{ij}^L a_i^{L-1} + b_j^L\right]^{-1}$$
$$= 1. \tag{26}$$

We have thus found the derivatives of the cost function w.r.t. both the weights and biases in the output layer, W^L and $\mathbf{b}^L$.

The equation

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{27}$$

holds for any layer, not just the output as in Eq. (24). Relating this to derivatives w.r.t. the layer $l+1$ z_js, we find

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{z_j^l}$$
$$= \sum_k \delta_k^{l+1}\frac{\partial z_k^{l+1}}{z_j^l}$$
$$= \sum_k \delta_k^{l+1} w_{kj}^{l+1}\frac{\partial f^l}{\partial z_j^l}, \tag{28}$$

with

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = \frac{\partial}{\partial z_j^l}\left[\sum_{i=1}^{N_l} w_{ik}^{l+1} a_k^l + b_k^{l+1}\right]$$
$$= \frac{\partial}{\partial z_j^l}\left[\sum_{i=1}^{N_l} w_{ik}^{l+1} f^l(z_k^l) + b_k^{l+1}\right]$$
$$= w_{jk}^{l+1} f^l(z_j^l). \tag{29}$$

The rest of the backpropagation scheme is essentially iterating Eq. (28), and computing—for each layer—the gradients $\partial C/\partial w_{ij}^l = \delta_i^l a_j^{l-1}$ and $\partial C/\partial b_i^l = \delta_i^l$. Once the gradients are known, updating the weights and biases to improve the performance of the network (making the cost function smaller) can be done by the aforementioned gradient descent schemes.

Training the Full Network

Overall, training a NN is conceptually simple, and involves only three steps:

Assume input x and corresponding *correct* output y is known.

(1) Compute output $\mathrm{NN}(x) = \hat{y}$ of the artificial neural network, and evaluate the *cost* function, $C(\hat{y}, y)$.

(2) Compute the gradient of $C(\hat{y})$ w.r.t. all the parameters of the network, w_{ij}^l and b_j^l.

(3) Adjust the parameters according to the gradients, yielding a better estimate $\hat{y}$ of the true output value y.

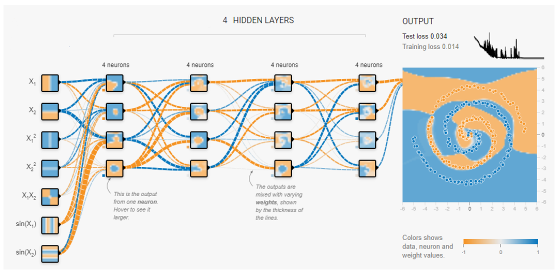
(4) Repeat (1)—(4).

The training scheme is known as *supervised learning*, because the NN is continually presented with x, y pairs, i.e. an input and an expected output. The cost (or objective or loss) function determines how happy the network is with it's own performance. In general, the output of the neural network is a vector of values, $\mathbf{y}$, and the cost function is taken across all outputs. In Eq. (5), the network produces N_{O} outputs for each input (which itself may be a vector).

Step (3) is easy to understand, but complex in practice. In order to update the network weights and biases, a measure of the expected change in the total output is needed. Otherwise, any change would just be done at random[1]. This means we need to compute the set of derivatives

$$g_{ij}^{l} \equiv \frac{\partial C(\hat{\mathbf{y}})}{\partial w_{ij}^{l}}, \quad \text{and} \quad h_i^k \equiv \frac{\partial C(\hat{\mathbf{y}})}{\partial b_i^l}. \tag{30}$$

The most common algorithm for computing these derivatives is the backpropagation algorithm as mentioned above (see Figure 3 for graphical visualisation of the process).



4 HIDDEN LAYERS
OUTPUT
Test loss 0.034
Training loss 0.014
4 neurons
4 neurons
4 neurons
4 neurons
X_1
X_2
X_1^2
X_2^2
X_1X_2
$\sin(X_1)$
$\sin(X_2)$
This is the output from one **neuron**. Hover to see it larger.
The outputs are mixed with varying **weights**, shown by the thickness of the lines.
Colors shows data, neuron and weight values.
-1
0
1

FIG. 3. Neural network classification visualization created at https://playground.tensorflow.org/. The leftmost column represents the input variables, while the remaining columns are hidden layers, each consisting of 4 neurons. The thicknesses of the lines connecting the neurons represent the weights of the connection, and the final and rightmost image is the output.

Vanishing gradients and weight initialization

In the neural network context, the phenomenon of saturation refers to the state in which a neuron predominantly outputs values close to the asymptotic ends of the bounded activation function. Saturation damages both the information capacity and the learning ability of a neural network. This implies that a fully *saturated* neuron with input $z_j \gg 1$, or a *dead* neuron with input $z_j \ll -1$ will most likely exhibit very small gradients and change very little during training (i.e. gradients vanish during training). This by itself indicates that some of the neurons to become a constant number; a job already performed by the bias b_{j+1}. To avoid a saturated behaviour, we initialize weights slightly better than a common heuristic (i.e. $W_{i,j} = \mathcal{U}(-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}})$ where $\mathcal{U}$ denotes the uniform distribution.)

Typically, activation functions are constant in their domain except in the small range close to 0, where they change appreciably. In this region, we may assume that the activation functions are roughly linear. In order for the signal to propagate through the network usefully, we essentially want the mean value of the z_j for each j to vanish, and the variance to be on the order of 1. Taking into account the linearity around in region close to 0, we get the activation:

$$A = W_1X_1 + W_2X_2 + \cdots + W_nX_n, \tag{31}$$

which has variance

$$\begin{aligned} \mathrm{Var}(A) &= \mathrm{Var}(W_1X_2 + \cdots + W_nX_n) \\ &= n\mathrm{Var}(W_i)\mathrm{Var}(X_i), \end{aligned} \tag{32}$$

where X is the input vector with n components. Here we assumed that the inputs and the weights are all independent, identically distributed, with vanishing mean. If we want the activation to have variance on the order of 1, then we must require that the variance of the weights is

$$\mathrm{Var}(W_i) = \frac{1}{n}. \tag{33}$$

Performing the same analysis with the backpropagated input entering the activation function yields the same result,

$$\mathrm{Var}(W_i) = \frac{1}{n'}, \tag{34}$$

with n' being the amount of weights in the *next* network layer.

With this in mind, Glorot and Bengio (2010) suggested initializing weights with average variance:

$$\mathrm{Var}(W_i^l) = \frac{1}{n_l + n_{l+1}}. \tag{35}$$

[1]This is a possible approach, yielding a class of *genetic* optimization algorithms. We will not discuss such schemes in the present work.

In the case of sigmoid or hyperbolic tangent transfer functions, we may realize this variance by initializing $w = \mathcal{U}(-r, r)$ with

$$r_{\text{sigmoid}} = \sqrt{\frac{6}{n_l + n_{l+1}}},$$

$$r_{\tanh} = 4\sqrt{\frac{6}{n_l + n_{l+1}}},$$

where $\mathcal{U}(a, b)$ denotes a uniform distribution between a and b and n_l (n_{l+1}) is the number of neurons in the current (next) layer.

With rectifying linear units, the weight initialization scheme changes slightly. As the ReLU transfer function vanishes across half of the coordinate system space, He et al. (2015) suggest doubling the variance of the weights in order to keep the propagating signal's variance constant, i.e.

$$\text{Var}(W) = \frac{2}{n_l}. \tag{36}$$

We may realize this by initializing weights $w = \mathcal{N}(0, 1)\sqrt{2/n_l}$, where $\mathcal{N}(\mu, \sigma)$ denotes the normal distribution with mean μ and standard deviation σ.

III. CLASSIFICATION STUDY

F. The Softmax Regression

Until now, our discussion has been limited to regression problems where the model learns to output a numerical value. However, in some cases, we are interested in classifying data points into two or more classes based on a set of features. Linear regression would not be suitable for this purpose as the range of outputs is from negative infinity to positive infinity. Even if a threshold, which can be learned or manually set, is used as a cutoff value for classifying data points, the model would still be incompetent as it would suffer from instability when new data points are added to the training data set.

One way to circumvent this problem is to use a specific activation function, namely the logistic function, also called the sigmoid function. This function has two advantages over the hard-classification threshold approach. Firstly, the sigmoid function is differentiable over all its domain which makes it more favorable when it comes to numerical optimization methods such as gradient descent. Secondly, the range of this function is bounded between zero and one; an output that can be interpreted as the probability of a data point belonging to one class or another. It turns out that performing such an operation is equivalent to modeling the odds of the two classes as a linear combination of the features, i.e. a dot product of the weights vector and the input/feature vector. This benefits from linear regression's ability to approximate linear and non-linear models to produce a numerical output.

$$a(x_i) = \sigma(\beta x_i) = \frac{1}{1 + \exp(-\beta x_i)}$$

where σ is the sigmoid function and $a(x_i)$ is the activation function output of the i^{th} data point.

Performing such an operation gives rise to the Logistic regression technique. In fact, a logistic regression model can be seen as a specific case of a neuron from a neural network with the sigmoid function being its activation function. Moreover, this model can be extended from the binary case of two possible classes to the multiple-class scenario. In such a case, the activation function should be able to accommodate for these additional classes. For this purpose, an extension of the sigmoid function, called the softmax can be used in replacement.

$$a(x_i)_k = \frac{\exp(\beta_k x_i)}{1 + \sum_{k=1}^{K-1} \exp(\beta_k x_i)}$$

$$a(x_i)_K = \frac{1}{1 + \sum_{k=1}^{K-1} \exp(\beta_k x_i)}$$

where k is the class for which the probability is being calculated, K is the last class, i is the data point index, and β_k is the weights vector for class k.

For this task, the MSE cost function is replaced by the negative log likelihood function, also known as the cross entropy, where the predicted class is compared against the true label. In the binary classification case, this is equal to:

$$C = \sum_{i=0}^{n-1} -\Big(y_i{*}log(a(x_i)){+}(1{-}y_i){*}log(1{-}a(x_i))\Big)$$

where y_i is the true label for the i^{th} data point, and n is the number of data points.

In the multi-class case, the cost function becomes:

$$C = \sum_{i=0}^{n-1} \left(- \sum_{k=1}^{K} y_i * log(a(x_i)_k) \right)$$

G. Artificial neural networks

In addition to using neural networks for regression problems, they can also be extended to classification problems by using a similar approach to the one used in Logistic Regression. Simply, instead of using an identity activation function in the regression analysis(i.e. an activation function that outputs its input), we modify the output activation function as a sigmoid (for a binary classification setup) or a softmax (for a multiclass classification analysis) functions. Analogously to logistic regression, the negative log likelihood cost function is used instead of the MSE cost function.

IV. PREDICTION PERFORMANCE MEASURES

In order to evaluate the performance of the trained models, we have used two different performance metrics, namely the accuracy for classification problems and the mean square error, MSE, for regression problems.

The accuracy is simply the percentage of the data points that we were correctly classified by the model among all the possible classes.

$$Accuracy = \frac{\#Correctly\ classified\ data\ points}{\#All\ data\ points}$$

In the specific case of binary classification, accuracy is equal to the ratio of the sum of the true positives, TP, and the true negatives, TN, to the total number of data points:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

where FP represents the false positives, and FN represents the false negatives.

MSE, on the other hand, represents the average squared distance between the predicted values and the true ones. Mathematically, it is calculated using the following equation:

$$\frac{1}{n} \sum_{i=0}^{n-1} (y_i - \hat{y}_i)^2$$

where n is the number of data points, y_i is the true output value for the i^{th} input, and $\hat{y}_i$, is the predicted value for the i^{th} input.

V. DATA PRE-PROCESSING

One of the crucial parts is data generating process and the reasonable transformation of the variables of interest for achieving better prediction performance.

H. Split and standardize

In this report, we split the data into three parts: *train, validation and test* sets. On the train data (80% of the total data), we perform the designed model fitting (i.e. we train the model). Since the additional hyper-parameters come into play (i.e. shrinkage parameter λ, learning rate η etc), additional validation data (20% of the total data) is introduced. We choose these hyper-parameters on the validation data (i.e. we *tune* the hyper-parameters) based on the performance measures. After learning parameters β on the train data and choosing hyper-parameters on the validation data set, we compare the prediction performance of different methods on the test data (20% of the total data). This process is more "data greedy" as it requires larger amount of data for splitting, though it produces more reliable accuracy measures.

After having split the data, we use the *min-max standardization* for the outcome variables in the regression analysis. That is, from the outcome variable we subtract its maximum value and divide by the difference between its maximum and minimum values. Not only this ensures avoiding numerical stability problems in the exponential function, but, in most cases, improves/boosts prediction performance of the models.

$$\mathbf{y} := \frac{\mathbf{y} - \max(y)}{\max(y) - \min(y)} \tag{37}$$

I. Regression - Franke function

The goal with the regression study portion of the project is to use neural networks and linear regression methods to fit *Franke's Function.* The Franke function, originally developed by Franke (1979), is a function with two Gaussian peaks. It is normally used in surface interpolation problems. Specifically, the Franke function $f_{\mathrm{F}}(x,y)$ takes the full form:

$$\begin{aligned} f_{\mathrm{F}}(x,y) &= \frac{3}{4}\exp\left\{\frac{-1}{4}\left[(9x-2)^2+(9y-2)^2\right]\right\} \\ &+ \frac{3}{4}\exp\left\{\frac{-1}{49}(9x+1)^2+\frac{1}{10}(9y+1)^2\right\} \\ &+ \frac{1}{2}\exp\left\{\frac{-1}{4}\left[(9x-7)^2+(9y-3)^2\right]\right\} \\ &- \frac{1}{5}\exp\left\{\frac{-1}{4}\left[(9x+4)^2+(9y-7)^2\right]\right\} \end{aligned} \tag{38}$$

This function is generally evaluated for the inputs $0 \le x, y \le 1$.

To set up a design matrix, we will employ a homogeneous polynomial mapping.[2] Polynomial P defines the mapping $(x_1, x_2) \mapsto P(x_1, x_2)$. P can be thought of a function which maps (x_1, x_2) to the design matrix X of the dimension $n \times p$. The Franke function is a 2D function, thus we will consider x_1 and x_2 in all possible homogeneous combinations up to and including degree p ($p = 5$ in this paper). Polynomial of degree p consists of the $(\frac{p(p+3)}{n})$ terms and is formally given as follows (see Project 1 by Saanum et al. (2020)) ($p = 4$ in the present project):

$$\sum_{i=1}^{p}\sum_{j=0}^{i} x_1^{i-j} x_2^{j} \tag{39}$$

In this paper, we consider the data generated by the Franke function without additional noise:

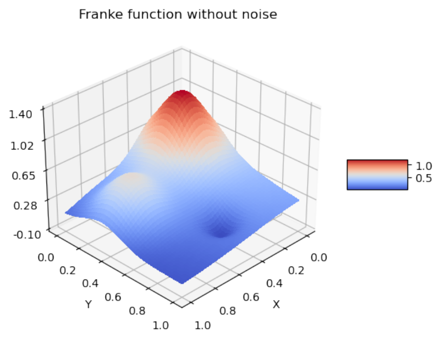


FIG. 4. The Franke function without noise plotted for $0 \le x_1, x_2 \le 1$.

[2] A homogeneous polynomial is a polynomial in which all terms have the same total degree, e.g. $x_1^2 + x_2^2 + x_1 x_2$ is a homogeneous polynomial of degree 2.

J. Classification - MNIST

In the classification analysis, we use the dataset provided by MNIST database. [3] The MNIST database was constructed from NIST's Special Database 3 and Special Database 1 which contain binary images of handwritten digits from 0 up to and including 9. In this study we use the scikit-learn dataset which consists of 1797 images of size $8 \times 88 \times 8$ collected and processed from the original database. An example of the hand-written image is attached:

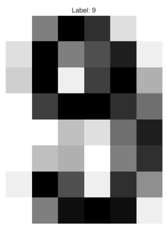


FIG. 5. Example of a hand-written digit 9.

The aim in the classification analysis is to classify hand-written images to their corresponding numbers with high precision.

One-hot encoding

It is common to transform the raw outcome variable as a separate binary encoded variables in the classification study. I.e. we have ten classes (digits enumerated from 0 to 9), therefore, outcome variable will have a dimension $n \times 10$ with n being the number of pixels:

$$\mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ \vdots \\ 9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} \tag{40}$$

VI. RESULTS

K. Regression

This section compares three different methods: the OLS, Ridge and Neural Networks by computing Mean Squared Error on the validation and test data sets. Moreover, we present test data predictions of each method.

[3] see the website: http://yann.lecun.com/exdb/mnist/

Hyper-parameter tuning (ANN)

Before turning to the comparison of the three methods, we present the hyper-parameter tuning on the validation data for neural networks. We carry out the grid-search to find the optimal combination.

Figures 6, 7 and 8 depict the MSE for different small values of the stochastic gradient descent decay and the learning rate when the Sigmoid, ReLu and leaky ReLu are used as activation functions correspondingly. In this instance, $\text{MSE}_{\text{sigmoid}} < \text{MSE}_{\text{leaky ReLu}} < \text{MSE}_{\text{ReLu}}$ **on average** for sufficiently small values of the learning rate and the shrinkage parameter. One of the important observations is that when the complexity of the neural networks architecture increases (i.e. more layers) we need smaller learning rates for the optimal prediction performance. A possible explanation lies in the architecture itself, when we increase the complexity, the network learns the information rather quickly by itself, thus without a great need of high parameter learning values.

To summarize the results, Sigmoid function learns rather efficiently relative to the rectifying functions. The optimal learning rate, $\eta = 0.001$ and the optimal shrinkage, $\lambda = 0.001$

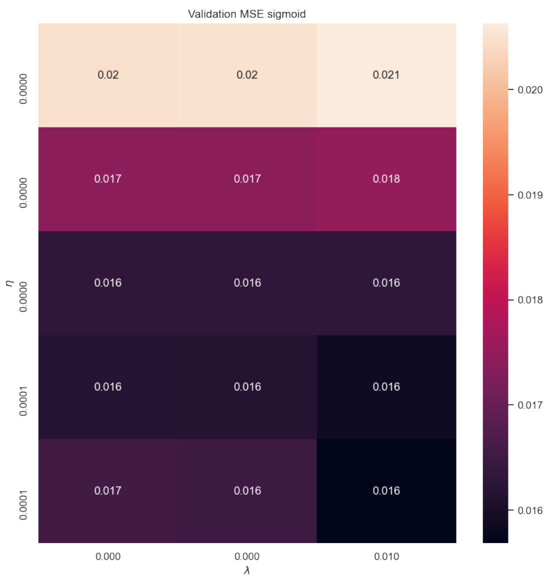


FIG. 6. MSE of the neural networks on the validation data for different stochastic gradient descent decays λ and learning rates η. We use 2 hidden layers and 50 neurons in each hidden layer activated by the sigmoid function.

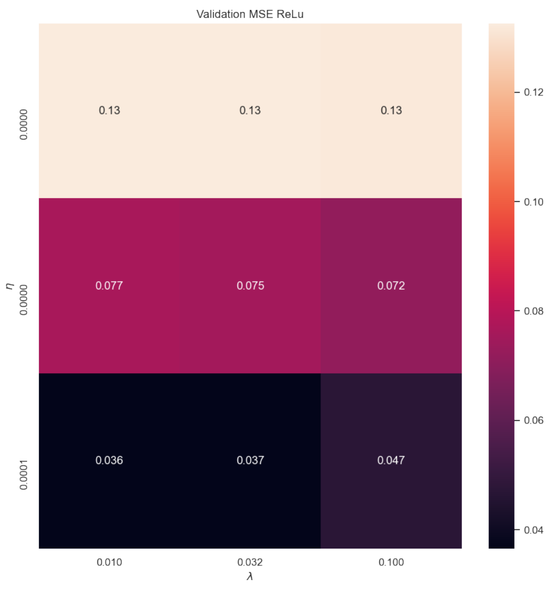


FIG. 7. The MSE of the neural networks on the validation data for different stochastic gradient descent decays λ and learning rates η. We use 2 hidden layers and 50 neurons in each hidden layer activated by the ReLu function.

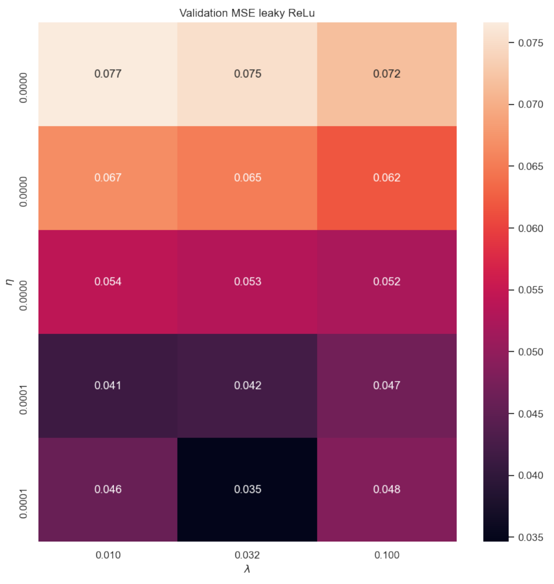


FIG. 8. The MSE of the neural networks on the validation data for different stochastic gradient descent decays λ and learning rates η. We use 2 hidden layers and 50 neurons in each hidden layer activated by the leaky ReLu function.

Hidden Neurons (ANN)

Another important quantity of interest is the number of neurons in the neural networks system. Figure 10 demonstrates the decreasing pattern of the MSE when the number of neurons substantially increases. This trend is perhaps not surprising, as the higher amount of neurons lets the network extract more information from the data, thus learn in a more efficient way.

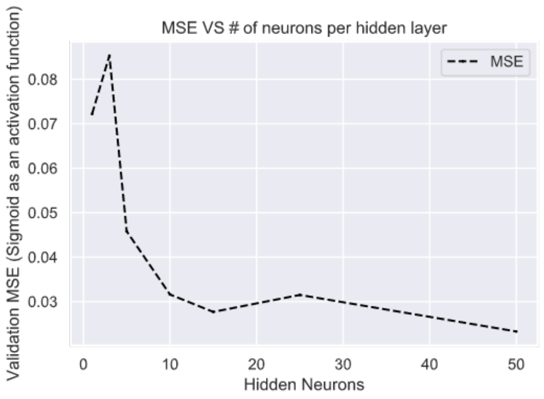


FIG. 9. The MSE of the neural networks on the validation data for different number of neurons. We use 2 hidden layers. $\eta = 10^{-4}$, $\lambda = 10^{-2}$.

Linear methods vs Neural Networks

We implement a similar analysis on the validation data for the Ridge and the linear regression schemes. The results for the linear regression and the ridge regression are very similar to those illustrated in Project 1 (Saanum et al. (2020)).

We can see that for different values of the learning rate η within 10^{-5} and 10^{-3}, the conventional linear regression starts out with a very high value of the MSE, then the OLS prediction performance improves gradually for higher learning rates. The results show that $\text{MSE}_{\text{NN}} < \text{MSE}_{\text{linear regression}} < \text{MSE}_{\text{Ridge}}$.

However, the improvement of the neural networks is rather small relative to the standard linear regression models. In the Franke function example, visual inspection verifies that the regular linear regression performs well. Therefore, when it comes to the choice of the method, neural networks would be sub-optimal because of the high computational cost that it entails. Ridge regression performs the worst in this instance, the fact that is verified by the Project 1 as well.

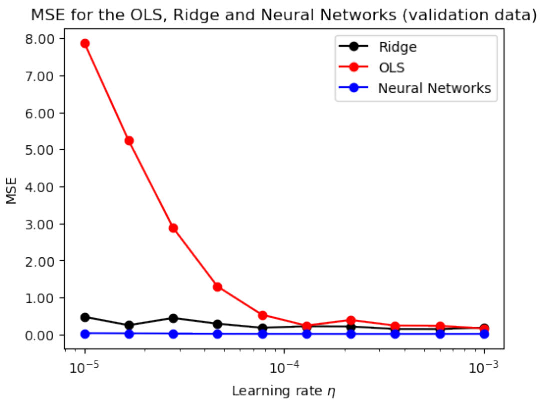


FIG. 10. The MSE of the linear regression, ridge regression and neural networks on the validation data for different number of neurons. We use 2 hidden layers and 18 neurons. Shrinkage parameter $\lambda = 10^{-2}$ for both, neural networks and the Ridge regression.

Prediction: linear methods & neural networks

After tuning hyper-parameter values on the validation data, we compare linear methods and neural networks on the test data set.

Figure 11 illustrates prediction performance of the linear, ridge regression and neural networks correspondingly. The scatter-plot consists of the true true outcome values (the blue straight line points) and the predicted outcome (red dots around the line). The closer the predicted outcome lies around the true outcome values, the better the performance of the method. Visually, the plot clarifies that the standard linear regression is the winner in the regression analysis. Neural networks comes in its distant second, and the Ridge regression takes its last place.

As a robustness analysis, we plot the predictions of the manually setup neural networks and the ANN designed using *tensorflow*. The predictions are similar. Tensorflow implementation uses *Adaptime moment estimation*(Adam) [4] as

[4]Instead of recomputing the gradient at each iteration, Adam keeps a part of the change at the previous time step, essentially giving the optimization a momentum—accelerating the minimization in the *parameter space directions* in which the gradient is not steep, but consistently has a small value aimed steadily in one direction. It also hampers the rapid oscillating solutions in *parameter space directions* in which we are close to the optimum, and the steep gradient makes the SGD

an optimizer instead of the stadard SGD algorithm, therefore leads to the improved prediction performance.

L. Classification

This section discusses hyper-parameter selection on the validation data and compares the softmax regression with neural networks.

Hyper-parameter tuning (ANN)

Figures 13 and 14 depict the accuracy scores for different combinations of the learning rate, the SGD decay and activation functions. When we use the Sigmoid function to activate the incoming inputs (i.e. signals) from the previous layer, the combination $(\eta, \lambda) = (10^{-3}, 10^{-3})$ lead to the highest accuracy (92%). The same values of these hyper-parameters lead to the highes accuracy for the rectifying functions (ReLu, leaky ReLu), though the accuracy is much lower (82% for both, ReLu and leaky ReLu). Tanh performs the same as the sigmoid function (91% accuracy for $(\eta, \lambda) = (10^{-3}, 10^{-1})$). Elu performs better than ReLu and the leaky ReLu for the optimal choice of the parameters (89% accuracy for $(\eta, \lambda) = (10^{-4}, 10^{-1})$).

Hidden neurons (ANN)

Just as in the linear regression case, we illustrate the improvement of the accuracy score for increasing values of the hidden neurons:

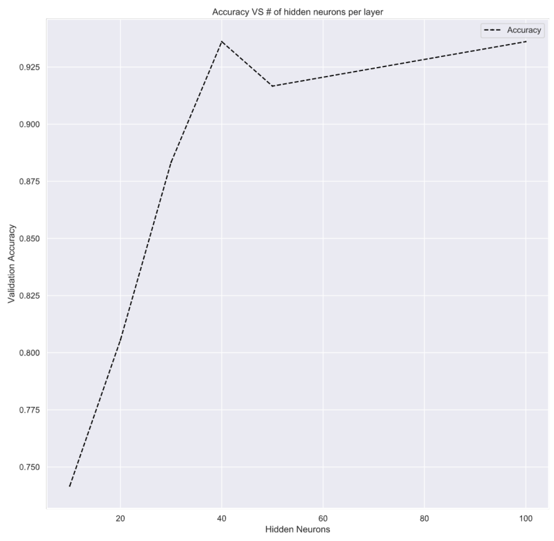


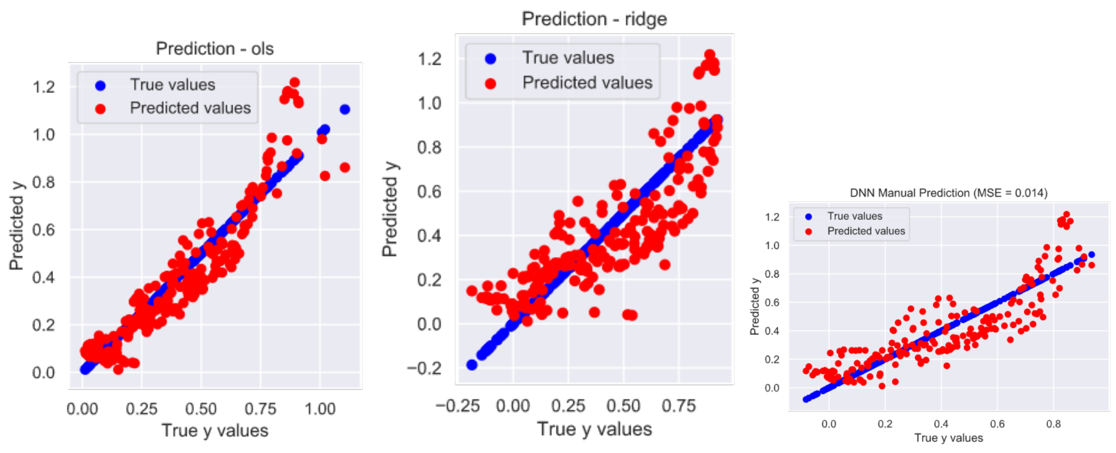
FIG. 15. The accuracy of the NN on the validation data for increasing number of neurons in each layer. Sigmoid activation function is used in each layer. $(\eta, \lambda) = (10^{-3}, 10^{-3})$. The number of SGD iterations (i.e. epochs) is 100.

The plot verifies that for increasing amount of neurons in each layer, the prediction accuracy also increases. In the classification analysis the optimal number of neurons is 40.

Softmax Regression vs Neural Networks

A crucial part of the analysis is to compare the neural networks with the softmax regression (a.k.a the multinomial logistic regression). In certain cases neural networks improves upon the prediction methods substantially. To check whether this applies to the MNIST dataset, we present the prediction performance of the softmax regression on the validation data.

over/under-shoot the solution at every other iteration (Qian (1999)).



OLS Ridge Neural Networks

FIG. 11. Ordinarly least squares, Ridge and neural network fits on the test data. Blue line: True outcome, $\mathbf{y}$ values. Red dots: Predicted outcome $\widehat{\mathbf{y}}$ values. For neural networks we use: 2 layers, 50 neurons, $\eta = 10^{-4}$, $\lambda = 10^{-2}$, 100 SGD iterations, Rectifying function parameters: $\alpha = 0.01, \lambda = 0.1$. For the standard linear regression: $\eta = 0.07$, 300 SGD iterations. For the Ridge regression: $\eta = 10^{-4}, \lambda = 0.01$, 300 SGD iterations.

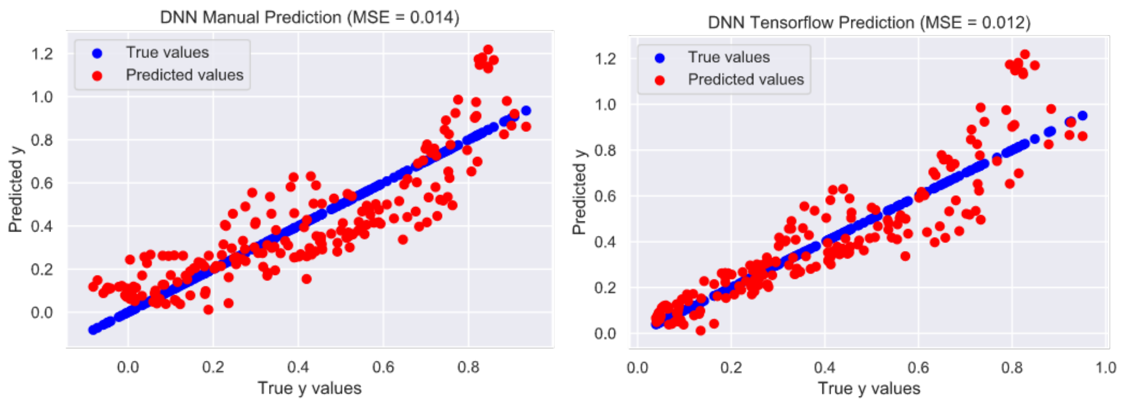


FIG. 12. Neural networks prediction on the test data set using the manual and tensorflow setup. Tensorflow setup includes Adam instead of the standard SGD optimizer.For the hyper-parameter values see Figure 11.

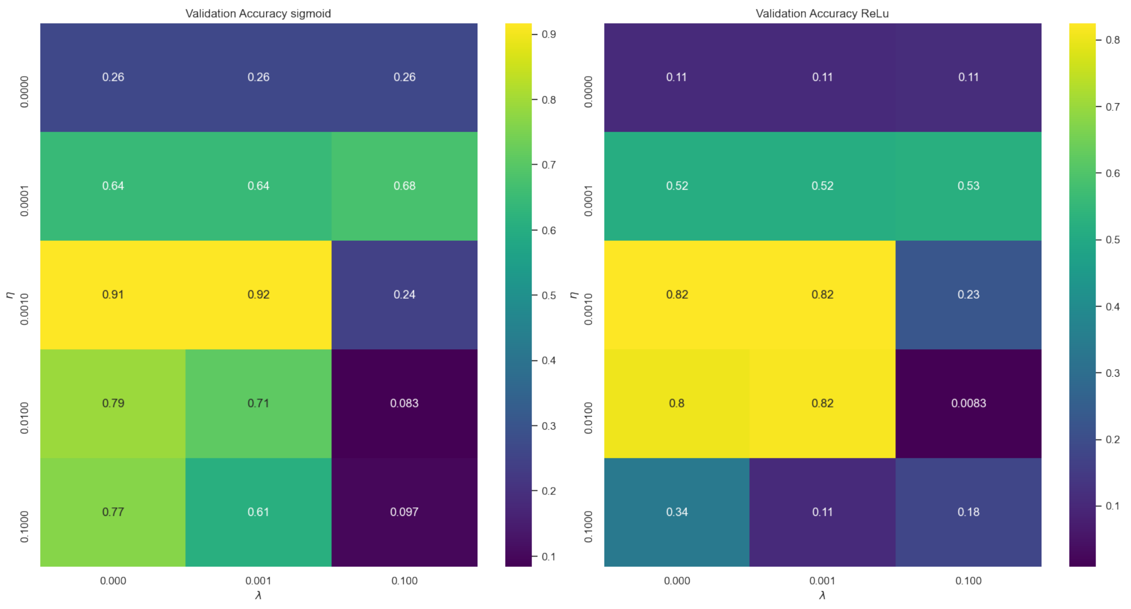


FIG. 13. Accuracy of neural networks on the validation data for different combinations of η and λ. Sigmoid and ReLu activation functions are used correspondingly. The number of iterations (i.e. epochs) in SGD is 100.

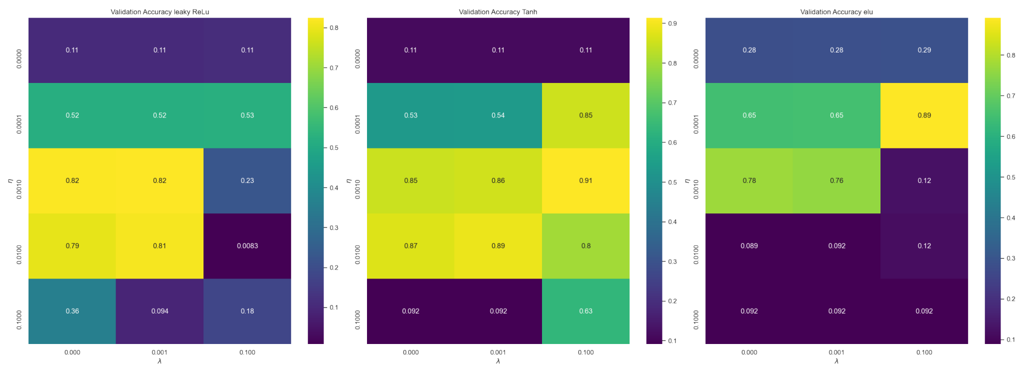


FIG. 14. The accuracy of the NN on the validation data for different combinations of η and λ. leaky ReLu, Tanh and Elu activation functions are used correspondingly. The number of iterations (i.e. epochs) in SGD is 100.

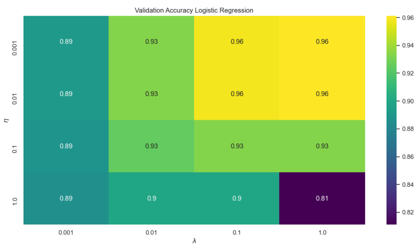


FIG. 16. The accuracy of the softmax regression on the validation data. The number of SGD iterations (i.e. epochs) is 1000.

Figure 16 indicates that the Softmax regression on the validation data achieves higher accuracy than neural networks when the learning rate, $\eta = 10^{-3}$ and shrinkage parameter $\lambda = 1$ (96% accuracy score). One of the potential reasons behind this behaviour lies in the number of iterations. Due to computational intensity of neural networks, we use only 100 iterations (epochs) in neural networks. Increasing the number of iterations will improve the accuracy of neural networks (as seen in Section VII).

VII. SUMMARY OF THE RESULTS

To outline and illustrate main findings of the paper, we present prediction performance measures for all the aforementioned methods in Table I.

The table clarifies that, in general, the methods achieve their best prediction performance on the train data.

In the regression analysis, the standard linear regression outperforms neural networks and the Ridge regression with the MSE of 0.8% (on the test data). Neural Networks comes in the distant second place, while the Ridge regression method does not lead to desirable results. This is a well known observation from Project 1 (Saanum et al. (2020)); In the present settings linear regression with the SGD algorithm also outperforms Ridge and even neural networks.

Even though the ordinary linear regression outperforms the other methods, our implementation of neural networks is based on the standard SGD algorithm. Changing the optimizer from the SGD to e.g. *adaptive moment estimation* (ADAM) improves the performance of neural networks. We observe this on the right side of Figure 12.

On the other hand, results in table I imply that neural networks is better tailored for classifying the numbers in the image dataset. This observation is perhaps not surprising; images are typically characterized with highly non-linear structure in which case we observe the enhanced prediction performance of neural networks. Neural networks outperforms the multinomial logistic regression, however this improvement is quantitatively small (1.4%). The softmax regression performs fairly well in this case. This behaviour can plausibly explained by the fact that the data at hand is not complex enough. Increasing the complexity further will speak in favor of neural networks.

VIII. CONCLUSION

In this work we extended the study of regression problems from Project 1 (Saanum et al. (2020)) by applying the linear and the Ridge regression to the franke function. We have also applied a neural network to the same problem. We found that for this particular problem the conventional linear regression has the over-all best performance in terms of the MSE and the computational cost.

Furthermore, we have studied the multi-class classification problem. In particular we considered the multinomial logistic regression and NN's applied to the classification of MNIST handwritten digits where these digits vary from 0 up to and including 9. For this problem it was clear that the flexibility provided by the neural network outperformed the linear logistic model. In particular we found that NNs achieved an accuracy of $\sim$ 97.2% when doing predictions on the test data, while the multinomial logistic model achieved an accuracy of $\sim$ 95.1%. This improvement is quantitatively small, though increasing the complexity of the data will further enhance superiority of neural networks.

We conclude that while neural networks, in general, provide high flexibility it is not necessarily always the best choice. Specifically, there are cases where we can get comparable performance using simpler models, as shown in the regression problem using the franke function. Also, the computational cost of applying NN's might be prohibitive if the dimensionality of the problem is large.

TABLE I. Prediction performance measures for trained linear regression methods and neural networks. The optimal learning rate $\eta = 10^{-4}$ and the shrinkage parameter $\lambda = 10^{-2}$ is used in all the cases in this table. We implement the SGD with 220 iterations in neural networks (classification), 1000 iterations in the softmax regression, 300 iterations for the linear and ridge regression, 200 iterations in neural networks (regression).

	MSE (%)		
Method	Training	Validation	Test
Neural Networks	1.3	1.6	1.4
Linear Regression	0.7	0.8	0.8
Ridge Regression	7.9	7.9	7.9
	Accuracy (%)		
Softmax Regression	95.1	93.3	95
Neural Networks	97.2	95.3	96.4

REFERENCES

Azizan, N. and Hassibi, B. (2018). Stochastic gradient/mirror descent: Minimax optimality and implicit regularization. *arXiv preprint arXiv:1806.00952.*

Franke, R. (1979). A critical comparison of some methods for interpolation of scattered data. Technical report, NAVAL POSTGRADUATE SCHOOL MONTEREY CA.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256.

Hastie, T. and Tibshirani, R. (2001). Friedman. *The Elements of Statistical Learning," Springer*, page 52.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034.

Hoerl, A. E. and Kennard, R. W. (1976). Ridge regression iterative estimation of the biasing parameter. *Communications in Statistics-Theory and Methods*, 5(1):77–88.

Kalogirou, S. A., Mathioulakis, E., and Belessiotis, V. (2014). Artificial neural networks for the performance prediction of large solar systems. *Renewable Energy*, 63:90–97.

Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151.

Saanum, T., Nareklishvili, M., and Seip, U. (2020). Linear regression methods and applications.

Sharma, R. D., Mittal, S., Tripathi, S., and Acharya, S. (2015). Using modern neural networks to predict the decisions of supreme court of the united states with state-of-the-art accuracy. In *International Conference on Neural Information Processing*, pages 475–483. Springer.

Zheng, H., Fu, J., Mei, T., and Luo, J. (2017). Learning multi-attention convolutional neural network for fine-grained image recognition. In *Proceedings of the IEEE international conference on computer vision*, pages 5209–5217.