Classification and Regression
From Linear and Logistic Regression to Neural Networks

Nicholas Karlsen and Thore Espedal Moe
University of Oslo
(Dated: November 14, 2020)

In this paper we implement from scratch a fully connected Feed-Forward Neural Network (FFNN) and investigate its application to function approximation and classification. To wit, we study the prediction/reconstruction of a set of terrain data from a subset of data-points as an example of function approximation, and we study the classification of hand-written digits 0-9 from the MNIST-dataset. As it is in a sense the motor of our neural network, we also look in some detail at the properties of the Stochastic Gradient Descent (SGD) method and its variant siblings. The performance of our network on the reconstruction of terrain data is compared and contrasted with previous results obtained via linear-regression methods. For the classification case the comparison is made with results from a logistic-regression based classifier that we also implement from scratch. We find that our FFNN fairly easily gives excellent results for the classification problem, and that it is significantly more accurate than our logistic-regression classifier; at least without further fine-tuning of the logistic classifier . For the terrain-predictions we observe, for some setups, sizable improvements over the previous results from linear-regression, but only after rather extensive fine-tuning of the network's hyper-parameters.

I. INTRODUCTION

Neural networks are among the most hyped techniques in modern machine learning. Being that they are extremely flexible and applicable to a wide range of problems from general function approximations to all sorts of classification tasks, this is hardly surprising. The blessing of flexibility can, however, be a curse in disguise as it gives rise to veritably vast range of hyper-parameters that must be tuned to the specific problems at hand. They have great potential, but the art of applying them optimally can at times appear akin to black magic.

In the present work we implement from scratch a Feed-Forward Neural Network (FFNN), aiming to explore its application both to linear-regression type problems and to classification tasks. We hope to gain some sense for the relative importance and the general effects of the various hyper-parameters the network can take in.

In somewhat simple terms, one can consider neural networks to be a collection of various weighting parameters that combine the inputs to the network in complicated non-linear ways to produce a predictive output. In order for the network to make useful predictions, it must be trained. That is, its weights must be optimized in some problem-specific way. In our present case, this optimization/training is driven by Stochastic Gradient Descent (SGD) methods. In a certain sense the neural network can be thought of as a car frame, translating the work of the SGD engine into movement/improved predictions. As such we will begin with testing the tuning of SGD (and some of its myriad siblings) applied to a simple linear-regression type problem where we have analytical solutions for the optimal model parameters. The goal is to gain insights into the properties of the SGD-type engines, which hopefully can guide our investigations of the full networks.

Armed with insights from the investigation of the SGD-methods, we turn to study the application of our neural network to a linear-regression type problem, namely the terrain-prediction considered in [1]. We vary a select set, guided by the results for the SGD-methods, of the network's hyper-parameters and try to find combinations which give better predictions than those obtained in [1]. The flexibility of neural networks can be expected to better predict such highly non-linear functions than regular (polynomial) linear-regression; but finding the combinations of hyper-parameters which actually perform better than plain old linear-regression is not trivial.

Subsequently we test the application of our network on a classification problem, specifically the classification of hand-written digits 0-9 from the MNIST data set. Once again we tune our network in an effort to obtain the best possible performance we can manage. In order to compare the neural network's results with another machine learning method, we then implement a simple multinomial logistic regression and apply it to the same classification problem.

Finally we summarize and compare how the different methods perform, both in terms of accuracy and computational speed, but also in terms of the time cost and difficulty for the user to find reasonable hyper-parameters. We do not aim to make definite statements as to which method is "best" since that very much varies from case-to-case; both in terms of how the various aspects like accuracy and computational speed are valued, but also in terms of how the various hyper-parameters relate to the problem at hand. Instead we wish point out some tendencies of a hopefully somewhat general nature, which can serve as a guidelines for which methods one might prefer to try first for a given case.

II. THEORY

A. Gradient Descent

Consider a cost function in the form

$$C(\boldsymbol{Y}, \tilde{\boldsymbol{Y}}(\boldsymbol{w})) = \frac{1}{N}\sum_{i=1}^{N} c_i(\boldsymbol{y}_i, \tilde{\boldsymbol{y}}(\boldsymbol{w})_i) \tag{1}$$

which quantifies the error for some data set $\boldsymbol{Y} = \{\boldsymbol{y}_1, \ldots, \boldsymbol{y}_N\}$ with respect to a corresponding set of modeled data $\tilde{\boldsymbol{Y}}(\boldsymbol{w}) = \{\tilde{\boldsymbol{y}}_1(\boldsymbol{w}), \ldots, \tilde{\boldsymbol{y}}_N(\boldsymbol{w})\}$, where $\boldsymbol{w}$ denotes the free parameters of the model.

Since we can not in general expect to have a way of minimizing the cost analytically, we may in stead optimize it by gradient descent (GD), where the set of free parameters $\boldsymbol{w}$ are initialized in some way, and then incremented by

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \eta \nabla_w C\Big(\boldsymbol{Y}, \tilde{\boldsymbol{Y}}(\boldsymbol{w})\Big) \tag{2}$$

for either a set number of epochs or until the weights has converged such that $\left\|\boldsymbol{w}^{(k+1)} - \boldsymbol{w}^{(k)}\right\|_2 < \varepsilon$, some tolerance. We also have the parameter $\eta \in \mathbb{R}_{>0}$, often called the learning rate, which can either be constant, or change wrt. k.

A notable, and very substantial drawback of GD is that it will always converge to a local minima for a given data set & initial weights, which for a particularly complicated high dimensional parameter space can not be expected to correspond to the global minima in general. This issue, along with many others [2, pp.15-16] motivates modifications to the GD method.

1. Stochastic Gradient Descent

One attempt at remedying the problems of GD is the so-called *Stochastic Gradient Descent* (SGD), in which stochasticity is introduced to the gradient descent by instead of performing GD on the entirety of $\boldsymbol{Y}$, it is performed on individual, randomly sampled points $\boldsymbol{y}_i$ with replacement, and updating the weights for each individual sample. It turns out that doing SGD in this way generally leads to faster, and better convergence compared to the regular GD, with the added benefit of being computationally faster. It is worth noting however that it is often observed that doing SGD without replacement can yield faster convergence, and is in practice often done this way. The exact reasoning behind this remaining an open question [3][4]. For simplicity, we have opted to conform to this norm and will be performing SGD without replacement.

2. SGD with Mini-Batches

Yet another modification of the SGD consists of performing the SGD on random subsets $\boldsymbol{Y}_{MB} \subset \boldsymbol{Y}$, rather than individual points $\boldsymbol{y}_i$. These subsets are often called *Mini-Batches* (MB), and their introduction may again improve the performance of SGD. To elaborate, this is performed by randomly splitting $\boldsymbol{Y}$ into a set of mini-batches where each subset has $\approx N_B$ elements[1]. GD is then performed on each of the subsets, updating the weights each time. Once the subsets has been exhausted, $\boldsymbol{Y}$ is again randomly sampled to construct another similar set of mini-batches, where each run-through of all the mini-batches constitutes an epoch of the algorithm.

The choice of N_B is somewhat decided by trial and error and is also strongly linked to the choice of learning rate η. However, it is usually chosen as relatively small number relative to the full dataset as to yield the benefits of the SGD method. One may also, as discussed in this excellent lecture by Carleo [5] intuitively link SGD to statistical mechanics, with the ratio $\eta/N_B \propto T$; representing an "effective temperature". Thus having small mini-batch sizes corresponds to a high effective temperature; where the system will be able to explore a much larger portion of the parameter space. Then by either increasing N_B, or decreasing η one may slowly lower the effective temperature; annealing the system into the global minima. For the full justification of this view; we refer the reader to Carleo [5].

Common choices for N_B is often in the order of $\sim 10 - 100$, with larger batches often yields diminishing returns in terms of the performance gain along with a larger computational cost. Further, choosing batch sizes as powers of 2 may yield slight performance gain by virtue of how computer hardware works [6].

3. Learning Rate Adjustment

As somewhat motivated by the suggested intuition in the previous section, it is often quite useful to be able to adjust the learning rate of SGD over the epochs, as this will in principle allow the algorithm to initially very rapidly explore the parameter space, prior to relaxing slowly into the (hopefully) global minima as the learning rate is decreased.

One such scheme is the inverse decay, where the learning rate evolves like

$$\eta(t) = \frac{\eta_0}{1 + \gamma t} \tag{3}$$

where η_0 is the initial learning rate, t the current epoch and the decay rate γ which controls the rate at which the learning rate will decrease. Similarly; we may also evolve the learning rate by as exponential decay as

$$\eta(t) = \eta_0 e^{-\gamma t} \tag{4}$$

[1] If $\boldsymbol{Y}$ is not exactly divisible, one may simply distribute the extra $\boldsymbol{y}_i$ equally among the batches

where the appropriate choice must be determined on a per-dataset basis as the convergence of SGD will vary wildly. An additional, much more involved technique is to run the SGD for a certain number of epochs, then manually readjusting the learning rate, either as a constant or by the above methods, then running for another set of epochs. Whilst this sort of technique where one manually hand-tunes the hyper-parameters requires much attention from the user, it may yield good results and even enable some of the simpler schemes to outperform some of the more sophisticated, adaptive schemes [7].

4. SGD Variants

Beyond the basic iteration defined by Eqn. 2, there exists a large number of alterations beyond just scaling the learning rate which in some way aim to improve the convergence of SGD. Some of which generally yield performance gains, whilst other may yield performance gains for particular types of problems. Perhaps the simplest of which is SGD with momentum (SGDM), which as the name suggests adds some parameter to the update rule which somehow attempts to respect the current rate at which the algorithm is traversing parameter space; speeding past shallow dips and slowing down for deep, narrow ones. Much akin to the physical quantity from which it is named. The update scheme may then be summarized by the following equations

$$\begin{aligned} \Delta \boldsymbol{w}^{(k)} &= p\Delta \boldsymbol{w}^{(k-1)} - \eta \nabla_w C(\boldsymbol{Y}, \tilde{\boldsymbol{Y}}(\boldsymbol{w})) \\ \boldsymbol{w}^{(k+1)} &= \boldsymbol{w}^{(k)} + \Delta \boldsymbol{w}^{(k)} \end{aligned} \tag{5}$$

where the momentum p is usually set in the interval $[0, 1]$. Whilst simple, this scheme often yields significant performance gains over the standard constant learning rate methods, which in practice is rarely used [2]. There also exists a plethora of other sophisticated schemes that in some way aims to increase the convergence rate of SGD, like the adaptive gradient (AdaGRAD), in which the learning rate is down-scaled by a cumulative history of the magnitude gradient. There is also the Root mean squared propagation (RMSProp) which works in a similar way to AdaGRAD by adapting the learning-rate to the gradient, but has an additional hyper-parameter which lets it "forget" older gradients. Common to many of these schemes however; is that they are often domain specific, and while they may yield exceptional performance for certain problems, there is no universal best choice.

B. Neural Networks

Neural Networks are a class of algorithms in which a set of nodes, often referred to as Neurons, are connected as a weighted graph. Which as the name may suggests, aims to emulate behaviour similar to that of the human brain. Each of the nodes in this graph is then activated by an activation function, which takes the weighted input of all of its connected graphs like

$$a = \sigma(w_1 a_1' + w_2 a_2' + \cdots + w_n a_n' + b) \tag{6}$$

where w_i, a_i' denotes the weights and activations of the connected nodes, and σ the activation function of the node, which is chosen differently depending on the problem at hand. Lastly, we have also have a bias, b, which simply shifts the activation of the Node in the case of a binary activation, or in the case where the activation function is chosen to be the identity function, the bias is simply the intercept.

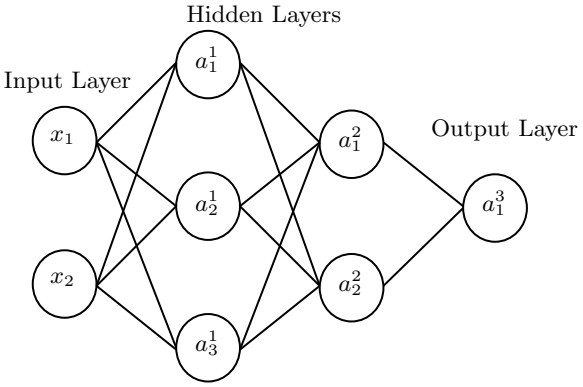


FIG. 1: Visual representation of a simple neural network with 2 inputs, 2 hidden layers with 3, 2 neurons respectively and an output layer with a single neuron.

The simplest type of neural network is the *Feed-Forward Neural Network* (FFNN), which is a directed graph, consisting of layers where every neuron in the preceding layer is connected to every neuron in the following layer as depicted in Fig. 1, which is directed going from left to right.

As the name may suggest, the input layer is where the input data starts out, before being fed forward through the hidden layers of the network, weighted in-between each node before reaching the output layer, the final model. In a similar fashion to other methods of supervised learning, the neural network must first undergo a process of training to obtain a suitable set of weights and biases. This is done by optimizing the all of the weights and biases in the network wrt to a chosen cost function usually by SGD. However, due to the costly nature of evaluating all of these cost functions; an algorithm which cleverly exploits the chain rule of differentiation has been developed to very efficiently compute all of the gradients in the network in a process which is called backpropagation.

1. Backpropagation

We base our explanation of the backpropagation algorithm following more or less directly Nielsen [8] and

Mehta *et al.* [2], referring the reader to those texts for the finer details and derivations of how the algorithm actually works, focusing instead on the computational aspects in the present texts. However, in short; the algorithm may be summarized as a clever use of the chain rule of differentiation.

Before we start, we first define a few quantities central to the network. First, we have the set of weights connecting each node in the network

$$\{w_{jk}^1, \dots, w_{jk}^L\} \tag{7}$$

where in a somewhat sloppy notation the j, k indices denote the number of neurons in the current and previous layers respectively, thus spanning a different range for each l. We also define the set of inputs

$$\{z_j^1, \dots, z_j^L\} \tag{8}$$

which again adheres to the same convention of j denoting the number of neurons in the current layer. These inputs then gets fed into the activation function $\sigma(z_j^l)$ of their corresponding neurons yielding the set of activations

$$\{a_j^1, \dots, a_j^L\} \tag{9}$$

where notably the activation function used in the output layer may differ from the one used in the hidden layers. We denote this special, output activation function as $\tilde{\sigma}(z_j^L)$.

We then start the backpropagation algorithm by first computing the response of each of the neurons in the network by the Feeding the input data forward in the network, starting with the first hidden layer, which is activated directly by the input data X_p like

$$\begin{aligned} z_j^1 &= w_{jk}^1 X_k + b_j^1 \\ a_j^1 &= \sigma(z_j^1) \end{aligned} \tag{10}$$

where we adopt the Einstein summation convention by summing over repeated indices. We then similarly compute

$$\begin{aligned} z_j^l &= w_{jk}^l a_k^{l-1} + b_j^l \\ a_j^l &= \sigma(z_j^l) \end{aligned} \tag{11}$$

for hidden layers $l = 2, \dots, L$. Then, for the output layer we compute

$$\begin{aligned} z_j^L &= w_{jk}^L a_k^{L-1} + b_j^L \\ a_j^L &= \tilde{\sigma}(z_j^L) \end{aligned} \tag{12}$$

where a_j^L is the predicted response, and $\tilde{\sigma}$ is the activation function for the output layer, which as mentioned may differ from the activation function used in the hidden layers.

We then compute the error of the output as

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \odot \tilde{\sigma}'(z_j^L) \tag{13}$$

where $\odot$ denotes the Hadamard product, which is a form of element-wise multiplication of vectors and matrices. It is also worth to note that Eqn. 13 implicitly assumes that the derivatives of the output activation function $\frac{\partial}{\partial z_j}\sigma(z_i^l) = 0 \ \forall \ i \neq j$. Whilst this holds true for a wide variety of activation functions; it does not hold for the SoftMax, which we will return to later on when we look at classification.

We then continue on by backpropagating the error like

$$\delta_j^l = \left(\delta_k^{l+1}\left(w^{l+1}\right)_{kj}^T\right) \odot \sigma'(z_j^l) \tag{14}$$

for all $l = L-1, \dots, 1$.

We may then easily compute the gradients of the cost function wrt. the weights & biases as

$$\begin{aligned} \frac{\partial C}{\partial w_{jk}^l} &= \delta_j^l a_k^{l-1} \\ \frac{\partial C}{\partial b_j^l} &= \delta_j^l \end{aligned} \tag{15}$$

which are then used to update the weights and biases via gradient descent. Whilst this is in principle all there is to backpropagation, the way in which this method is performed requires some further modifications when solved on a computer.

2. *Backpropagation with mini-batches*

In order to efficiently perform backpropagation simultaneously across several inputs at once, we need to slightly adjust our algorithm such that we may take advantage of the fast and efficient linear algebra libraries that are available, like in for example Numpy.

We then structure our input and output matrices adhering to the row-major storage of Numpy arrays by letting $X \in [M \times P]$, $Y \in [M \times Q]$ where M denotes the number of data points in the mini-batch and P, Q the dimensionality of the input and output respectively. Explicitly, our data then undergoes the structure change

$$X = \begin{bmatrix} X_1 \\ \vdots \\ X_P \end{bmatrix} \to X = \begin{bmatrix} X_{11} & \dots & X_{1P} \\ & \vdots & \\ X_{M1} & \dots & X_{MP} \end{bmatrix} \tag{16}$$

similarly, we let $z_j^l \to z_{mj}^l$ and $a_j^l \to a_{mj}^l$ such that they are in accordance with X and Y.

In order to adhere to this new form, we transpose Eqn. 10 which yields

$$w_{jk}X_K \to (w_{jk}X_k)^T = X_k^T\left(w^1\right)_{kj}^T \tag{17}$$

Thus, we may write the initial step as

$$\begin{aligned} z_{mj}^1 &= X_{mk}\left(w^1\right)_{kj}^T + b_j^1 \\ a_{mj}^1 &= \sigma\left(z_{mj}^1\right) \end{aligned} \tag{18}$$

where the transposed bias is implicitly[2] added element-wise to each row in the resultant matrix. In a similar fashion, we feed forward for $l = 2, \dots L-1$, the last hidden layer as

$$\begin{aligned} z^l_{mj} &= a^{l-1}_{mk} \left(w^l\right)^T_{kj} + b^L_j \\ a^l_{mj} &= \sigma(z^l_{mj}) \end{aligned} \tag{19}$$

and for the output layer

$$\begin{aligned} z^L_{mj} &= a^{L-1}_{mk} \left(w^L\right)^T_{kj} + b^L_j \\ a^L_{mj} &= \tilde{\sigma}\left(z^L_{mj}\right) \end{aligned} \tag{20}$$

Then, we compute the error of the output error as

$$\delta^L_{mj} = \frac{\partial C}{\partial a^L_{mj}} \odot \tilde{\sigma}'(z^L_{mj}) \tag{21}$$

which we then backpropagate throughout the layers for $l = L-1, \dots 1$

$$\delta^l_{mj} = \delta^{l+1}_{mk} w^{l+1}_{kj} \odot \sigma'(z^l_{mj}) \tag{22}$$

and finally for the output layer. We the compute the derivatives of the cost functions wrt. the weights and biases for the input layer

$$\begin{aligned} \frac{\partial C}{\partial w^1_{jk}} &= \left(\delta^1\right)^T_{jm} X_{mk} \\ \frac{\partial C}{\partial b^1_j} &= \sum_m \delta^1_{mj} \end{aligned} \tag{23}$$

and similarly for layers $l = 2, \dots, L$ as

$$\begin{aligned} \frac{\partial C}{\partial w^l_{jk}} &= \left(\delta^1\right)^T_{jm} a^l_{mk} \\ \frac{\partial C}{\partial b^l_j} &= \sum_m \delta_{mj} \end{aligned} \tag{24}$$

The derivatives are then used to update the weights and biases by gradient descent in the same way as before. One may also following the same logic optimize for column-major storage, as is used in languages like Fortran, MATLAB, Julia, etc.

3. Activation Functions

As mentioned, at each neuron within the neural network we have an activation function σ, for which there exists many choices with different properties and domains in which they excel. For convenience, we have compiled a list of some of the commonly used ones in Table. I.

[2] Matching the behaviour of Numpys addition operator

Historically, the Sigmoid[3] and tanh have been popular choices, however in recent years ReLU and other similar activations have gained in popularity [6].

However, for the output activation function $\tilde{\sigma}$ one is a bit more restricted in choice, as an important requirement is that the output activation function must map to the entire range of the target data.

4. Weight & Bias Initialization

For neural networks, particularly larger ones, the way in which the weights are initialized may contribute significantly to the convergence of the Network. Where in particular the somewhat naive approach by sampling all the weights in a constant fashion from some normal, or uniform distribution may lead to poor convergence for larger and deeper networks. Where by virtue of the number of nodes; the product of the weights in one layer may blow up, which in turn will blow up the activations of the Network, or conversely, the weights may be initialized as being too small for some parts of the network. Thus, several clever schemes are employed too initialize the weights in such a way that avoid this issue, often specialized to some particular type of Neural Network. One such method is the Xavier initialization, proposed by Glorot and Bengio [9]. In this method, the initial weights are sampled individually for each layer from a uniform distribution[4] bounded by $\pm\sqrt{6/(j+k)}$, where j, k denotes the number of neurons in the current and previous layers respectively. Thus, the magnitude of the initial weights are scaled down with respect to the size of w^l_{jk}, thus avoiding the aforementioned problem. Another type of initialization proposed by He *et al.* [10] is observed to perform particularly well for ReLU type of activation functions. In He's initialization method, the weights are sampled from a standard normal distribution $\mathcal{N}(0,1)$ scaled down by a factor $\sqrt{2/k}$, where k denotes the number of neurons in the previous layer.

For the initialization of the bias', a standard choice is to set them all to zero Aggarwal [6]. However, for ReLU type activations it may in some cases be beneficial to add a small bias 0.01 to ensure that the neurons activate from the beginning, however, whether this is beneficial in general is not clear [11]. There are of course again a wide range of different techniques that may or may not increase the performance of your network, and as such, experimentation seems to be the key.

[3] Sometimes also referred to as the logistic function

[4] As a side-note, some sources suggest it as being sampled from a Gaussian instead. As is seemingly quite common in the ML literature there is a lack of standard practice with a lot of competing conventions. We thus opted for the safe route and follow the original paper.

TABLE I: Various Activation functions used in Neural Networks

Name	Activation Function	Derivative	Range
Identity	$\sigma(x) = x$	$\sigma'(x) = 1$	$(-\infty, \infty)$
Sigmoid	$\sigma(x) = \frac{1}{1+e^{-x}}$	$\sigma'(x) = \sigma(x)(1-\sigma(x))$	$(0, 1)$
SoftMax	$\sigma(x_j) = \frac{e^{x_j}}{\sum_k e^{x_k}}$	$\frac{\partial \sigma(x_k)}{\partial x_j} = \sigma(z_k)(\delta_{kj} - \sigma(z_j))$	$(0, 1)$
Tanh	$\sigma(x) = \tanh(x)$	$\sigma'(x) = 1 - \tanh^2(x)$	$(0, 1)$
ReLU	$\sigma(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } 0 < x \end{cases}$	$\sigma'(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ 1 & \text{for } 0 < x \end{cases}$	$[0, \infty)$
LeakyReLU	$\sigma(x) = \begin{cases} 0.01x & \text{for } x \leq 0 \\ x & \text{for } 0 < x \end{cases}$	$\sigma'(x) = \begin{cases} 0.01 & \text{for } x \leq 0 \\ 1 & \text{for } 0 < x \end{cases}$	$(-\infty, \infty)$

C. Application of Neural Networks

Having presented the general description of how neural networks work, it is now time to briefly look at the specifics of how they are applied to different types of problems. In this article we deal with two distinct tasks:, the approximation of continuous functions, and classification. These tasks are both amenable to treatment by neural networks, but require somewhat different setups for the input and output layers.

For function approximations the task is simply to map, for sample m, a set of input dependent variables $\mathbf{x}_m$ to their corresponding function-value $f(\mathbf{x}_m) = y_m$. The input to the neural network will then, for N samples with p dependent variables, just be the $[N, p]$-dimensional matrix of the dependent variables. The output will equally simply be the $[N, 1]$-dimensional matrix of predicted values $\tilde{y}_m$, and the activation function for the output layer can be taken as the identity function, i.e. $\tilde{y}_m = a^L_{m,j=1}$. This corresponds to p input nodes and a single output node in the network architecture. The most natural cost-function for this kind of output is, of course, the mean-squared error (MSE), where y_m are the actual response-values of the samples:

$$C = \frac{1}{N}\sum_m (\tilde{y}_m - y_m)^2 = \frac{1}{N}\sum_m (a^L_{m,j=1} - y_m)^2 \quad (25)$$

which is straightforward to differentiate with respect to the output activation functions when computing the error in the output layer δ^L.for the backpropagation-algorithm. It turns out that FFNN with at least one hidden layer is in principle capable of modeling any type of functional outcome, as proved by Hornik *et al.* [12]. Where the basic idea comes from the fact that neurons may form complicated, nonlinear terms as opposed to standard regression techniques like the ones we studied previously in [1].

In classification the task is, for sample m, to go from some p predictor variables x_{mp} to determining whether or not the sample belongs to a certain class k. The input to the neural network will then be, for N samples with p predictors, the $[N, p]$-dimensional matrix of the predictors. Thus requiring p input nodes in the network.

In the present work we restrict ourselves to data-sets where the samples belong to one of K distinct classes. When the samples belong to one, and only, of K classes, the output can be represented as a $[N, K]$-dimensional matrix $\tilde{\mathbf{Y}}$ representing the model's confidence $\tilde{Y}_{mk}$ that sample m is a member of class k. This means the number of output nodes should be K.

A very convenient way to represent the actual class-membership of the samples for these kinds of problems is the so-called "one-hot vector-form". For each sample m belonging to one of K classes we can write a K-dimensional target vector $\mathbf{y}^{target}_m$ whose elements are, using the Kronecker-delta, $y^{target}_{mi} = \delta_{ik}$ where k is the sample's actual class-number.

As will be motivated in the subsequent discussion of logistic regression, there are two different activation functions commonly used for the output-layer in classifying applications: the sigmoid function and the SoftMax function.

$$C = -\frac{1}{N}\sum_m \sum_k y^{target}_{mk} \ln\Big(\tilde{Y}_{mk}\Big) - (1 - y^{target}_{mk}) \ln\Big(1 - \tilde{Y}_{mk}\Big) \quad (26)$$

while for SoftMax the preferred cost-function is:

$$C = -\frac{1}{N}\sum_m \sum_k y_{mk}^{target} \ln\left(\tilde{Y}_{mk}\right) \tag{27}$$

which we will call the "SoftMax-loss". The reason for using different cost-functions is a practical one. While the sigmoid function activating the node k in the ouput layer is independent of all the other inputs $z_{j\neq k}$ to the nodes in the output layer, the SoftMax function DOES depend on all the other $z_{j\neq k}$. We will not show this, it comes from a simple but rather long and tedious chain of chain-rule applications, but merely state as [8] that this choice allows us to simply compute the error in the output layer as:

$$\delta_m^L j = \tilde{Y}_{mj} - y_{mj}^{target} \tag{28}$$

On the theoretical side, one might be given pause by the impression that eq. 26 penalizes both "underestimating" the correct class and "overestimating" the wrong classes, while eq. 27 only appears to punish "underestimating" the correct class. In this case one should remember that the SoftMax-activation contains the "normalization sum" over the exponentials of all the z_j inputs to the output-layer. This implicitly penalizes giving confidence to the wrong classes, so the explicit penalty in from the right-hand term in eq.26 becomes somewhat redundant for the SoftMax case.

D. Classification with Logistic Regression

We have previously described how neural networks can be used to do classification problems. A different method for performing classification is (Multinomial) Logistic Regression. As opposed to linear regression where the aim is to project data-points into some linear basis as to model some continuous function, logistic regression instead aims to project data unto some binary response: 0,1. I.e. whether or not some particular data point fits into some category k. Thus, we wish to fit our data unto some function $f : \mathbb{R} \to [0, 1]$ for each of the categories K under consideration.

As in linear regression one may start by assuming, for a given category k and a given example, a response S_k to be a linear combination of the data-features x_p:

$$S_k = \beta_{0k} + \sum_p \beta_{pk} x_p \tag{29}$$

where the β are the regression coefficients including an intercept β_{0k}. The departure from linear regression is that the responses S_k are further input to a function f whose range is $[0, 1]$. so that for each category k the regression output becomes:

$$\tilde{y}_k = f(S_k) \tag{30}$$

The output $\tilde{y_k}$ can be interpreted as the model's confidence that the example belongs to the class k. In the case of single-class problems (yes/no for membership in a single category/class) the use of the logistic function (also known as the sigmoid function) as f allows one to directly interpret $\tilde{y}_k$ as a probability of class membership for a given set of features and regression weights. That is:

$$P(y_{target} = 1|x_p, \beta) = \frac{e^{S_k}}{1 + e^{S_k}} = \tilde{y}_k \tag{31}$$

For multiclass exclusive problems (i.e. the example belongs to one and only one of the K considered classes k) one can achieve the same probability interpretation by using the SoftMax-function:

$$P(y_{target}^k = 1|x_p, \beta) = \frac{e^{S_k}}{\sum_k e^{S_k}} = \tilde{y}_k \tag{32}$$

It should be noted that the single-class binary problem is equivalent to a two-class exclusive problem. Thus the generalization in 32 can be equally well used for single-class binary problems, if the target y_{target} is reformulated according to:

$$y_{target} = 0 \to y_{target}^{k=0} = 1, \quad y_{target} = 1 \to y_{target}^{k=1} = 1 \tag{33}$$

The above equations describe the (multinomial) logistic predictions for a single example $\mathbf{x}_n, \mathbf{y}_n$ where the vector $\mathbf{x}_n$ contains the p predictors for that example and the vector $\mathbf{y}_n$ is the K-dimensional one-hot form of the target.. These equations can be extended to deal with N samples belonging to one of K different classes, and rewritten in terms of matrices (including an intercept column of ones in the design matrix $\mathbf{X}$):

$$\tilde{Y}_{nk} = \frac{e^{[\mathbf{X}\beta]_{nk}}}{\sum_k e^{[\mathbf{X}\beta]_{nk}}} \tag{34}$$

where $\tilde{\mathbf{Y}}$ is the $[N, K]$ matrix of the probabilities that example n belongs to the class k, $\mathbf{X}$ is the $[N, p + 1]$ design matrix and β is the $[p+1, K]$ matrix of regression coefficients.

By using the SoftMax-loss as cost function:

$$C(\beta) = -\frac{1}{N}\sum_n \sum_k y_{nk}^{target} \ln\left(\tilde{Y}_{nk}\right) \tag{35}$$

one can obtain (after some simple, but tedious, applications of the chain rule) the following expression for the gradient of the cost function with respect to the regression parameters β :

$$\nabla_\beta C(\beta) = \frac{1}{N}\mathbf{X}^T(\tilde{\mathbf{Y}} - \mathbf{Y}) \tag{36}$$

which is a matrix of dimensions $[p+1, K]$. Having obtained the gradient, one can then optimize the regression parameters β by minimizing the cost-function using e.g. SGD. When the regression parameters have been optimized, the model 34 is ready to predict the class memberships of the input data. If one wishes, one can easily include l_2-regularization by simply adding a term $\sum_i \sum_j \frac{\lambda}{2}\beta_{i,j}$ to the cost-function; that simply corresponds to adding the matrix $\lambda\beta$ to equation 36. As a technical note, without further tweaking this will include the potentially undesirable penalizing of the intercept coefficients β_{0k}. So long as the penalty λ is sufficiently small we do not expect this to produce any problems, but we do acknowledge that our current implementation does not avoid this intercept penalization when regularization is turned on.

Finally, we remark that our performance metric for the classification problem is not the values of the cost-functions, as it is for function-approximation. Rather we take, for both the neural network and logistic method, the so-called accuracy-score. The accuracy score is very simply defined as the number of correctly classified samples in a set divided by the total number of samples in the set. We, quite intuitively, interpret the class k corresponding to the largest the probability $\tilde{Y}_{nk}$ for a given sample n to be the sample's class predicted by our models.

III. RESULTS & DISCUSSION

A. Testing hyper parameters in SGD

In order to get an understanding of the way in which the wide variety of hyper-parameters work in SGD, and by extension FFNN, we studied the evolution of the mean squared error (MSE) of a 6th degree polynomial modeling the Franke Function wrt. the number of epochs the SGD had ran for. As a benchmark, we used the results found in our previous work [1] where the Franke function was studied rigorously using ordinary least squares (OLS), Ridge and LASSO regression, where in particular the OLS solution was found to perform optimally for our model of choice, and will thus be used as the primary benchmark.

In order to facilitate a direct comparison, we prepared our data in a similar way by sampling 500 random points from a uniform distribution spanning $[0, 1]$, the domain of the Franke function. We then centered the response of the data by subtracting the mean, and scaled the data using Sci-Kits StandardScaler() utility [13]. The dataset was then randomly split into a training and testing set at a $80:20$ ratio, where the testing set was set aside to measure the performance of the model for each epoch of the SGD using the MSE, which we previously found to be the most useful metric [1].

We begin our analysis by turning to Fig. 2, in which we have studied the effects of learning rates, as well as the addition of momentum and varying the size of mini-batches. Firstly, we look at all 4 plots in isolation, ignoring any comparison. We very clearly the effect of adjusting the learning rates η, leading to a quicker convergence. But notably; for $\eta = 10^{-2}$ in the case of $p = 0.75$ and MB size $= 32$, we see that once converged; the MSE becomes somewhat unstable, likely fluctuating around a minima. Here, the intuition is that whilst the SGD has been able to locate a local minima, it is unable to dive deeper into the minima due to its fixed learning rate, which if we draw the comparison of a ball rolling up and down in a quadratic well with a frictionless surface, it will never be able to settle further down the well for a given, constant energy. Where in this case, the learning rate is analogous to the energy of the ball. Therefore; once converged in such a way, lowering the learning rate is a necessity in order to converge to a lower MSE.

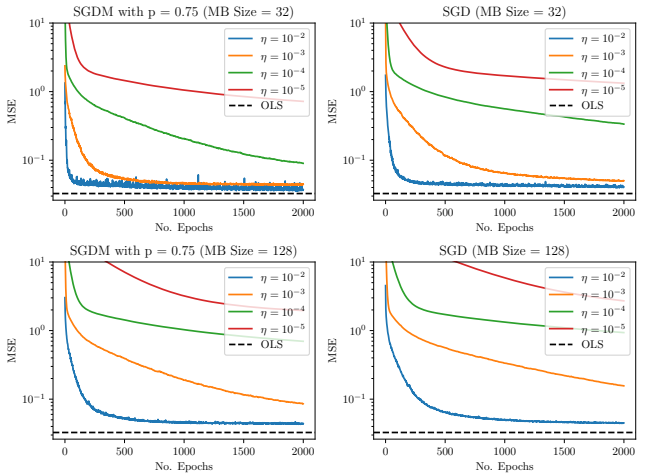


FIG. 2: Franke Function modeled as a 6th degree polynomial using an OLS type cost function optimized using Stochastic Gradient Descent with momentum for varying learning rates benchmarked against the analytic solution

We then compare the vertically neighboring figures in Fig. 2 by which may observe directly the result of varying the mini-batch size from 32 to 128, where the latter yields a more rapid initial convergence, but also a slightly more unstable exploration of the parameter space once converged, as exemplified by the SGDM plots on the left. More or less the same effect that we observe from varying increasing the learning rates, somewhat increasing our faith in the intuition proposed by Carleo [5].

We compare the horizontally neighboring figures in Fig. 2, where we observe the effect of adding momentum. We see observe that the general effect of adding momentum is an increase in the convergence rate. In particular, for $\eta = 10^{-3}$ in the cases with $MB = 32$, we see the clear benefit; by yielding a quick and relatively stable convergence compared to the same learning rate without momentum.

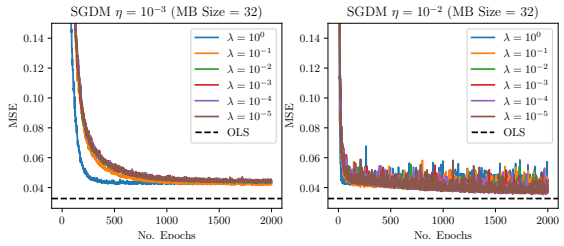


FIG. 3: Franke Function modeled as a 6th degree polynomial using an OLS type cost function optimized using Stochastic Gradient Descent with momentum $p = 0.75$ for varying L^2 penalties λ benchmarked against the analytic solution

We then turn our attention to Fig. 3, where we observe the effect of adding a penalty parameter λ to the cost function which punishes the weights, $\boldsymbol{w}$ for having a large L^2 norm akin to Ridge regression [1]. We curiously enough observe behaviour much akin to that of increasing the learning rate, ν, where the MSE very quickly converges to a floor before it starts to destabilize, oscillating rather rapidly. We may understand this by considering the way in which the L^2 penalty enters into the cost function like

$$C(\boldsymbol{Y}, \tilde{\boldsymbol{Y}}) = \frac{1}{N} \sum c_i(\boldsymbol{y}_i, \tilde{\boldsymbol{y}}(\boldsymbol{w})_i) + \lambda \|\boldsymbol{w}\|_2 \qquad (37)$$

which will affect the update rule in the GD given by Eqn. 2 somewhat. Firstly, it will fulfill its job by preventing the L^2 norm of $\boldsymbol{w}$ of blowing up by adding an extra negative term in the update rule which will grow proportionally with $\|\boldsymbol{w}\|_2$. However, this additional term will not completely vanish in the cases where $\boldsymbol{w}$ is "well behaved", and will thus serve as a roughly constant term continuously perturbing he weights irrespective of the gradient of the cost function. As such, we may expect a similar effect as we found in our previous report [1], in which the penalty serves as an excellent tool in preventing over-fitting thus yielding a stable MSE, but also does not yield the most precise models. As such; we are tempted to believe that the penalty parameters are mostly suited for unsupervised methods, in which one does not closely monitor the model as we have been doing. Lastly we discuss the usage of variable learning rates. During our experimentation of these, with both SGD and later on with FFNN, we found little gain in our performance. Quite the opposite, we did not find it worthwhile tweaking yet another hyper-parameter and ultimately opted for a hand-tuning strategy where in we manually adjusted the learning rate depending based on MSE/Epoch plots as we have looked at in this section. However, qualitatively the benefit of decaying the learning is clear based on the results we have seen in this section, an

Lastly, we discuss the usage of variable learning rates. During our experimentation both with the SGD and later on with FFNN, we did not find it particularly beneficial to employ either the inverse or exponential decay schemes as discussed in the theory section. We believe this is ultimately comes down to the fact that we aren't training our models for a particularly high number of epochs, as this would make the analysis much more computationally expensive and we are ultimately interested in the overarching behaviour of the models. However, based on our findings in this present section with the SGD, we can definitely appreciate the benefit of training a network for at an initially higher rate, gradually lowering the rate as the model converges leading to and overall faster training process. This also lead to us employing a hand-tuning strategy for trying to optimize our models in the following sections, where we let the model train for a set number of epochs, monitoring its performance and tweaking the learning rate up/down depending on the change recent change in MSE.

B. Building a FFNN model for terrain data

As a way to explore the performance of our FFNN on a regression problem, we opted to revisit the terrain data which we studied previously in [1]. In short summary, we studied a 1801×1801 section of terrain data which we down-sampled to every 40th pixel. We then parametrized the horizontal and vertical pixels as $x, y \in [0, 1]$, split the dataset into a 80:20 split of training and testing data. Our final models fitting the data as a 25th degree polynomial using OLS, Ridge and LASSO regression is reproduced here for convenience in Fig.12, where the best MSE scores where found to be in the range $\approx 2 \cdot 10^4 - 3 \cdot 10^4$. As our first approach, we performed a grid search over various combinations of penalties and learning rates with $k = 5$ fold cross validation, training the network for 500 epochs each time for Sigmoid, ReLU and LeakyReLU activation functions for 50×50 hidden layers, with the identity function as the activation of the output. We found this approach to be rather uninformative in constructing a good model with the behaviour across the different learning rates for a given penalty being rather uniform. Given that this approach was rather computationally expensive, we opted to abandon it in favor hand tuning the learning rate. Nevertheless, our efforts are documented in the

We ended up instead using a network of $100 \times 100 \times 100$ hidden layers where we hand-tuned the learning parameter after each 100 number of epochs for the three different activation functions Sigmoid, ReLU and LeakyReLU. We started with using reasonably high learning rates that gave fairly immediate decreases in the test-MSE. We continued using these learning rates until we plateaued, at which point we turned them up and/or down until we got out of the plateau. For the Sigmoid and ReLU functions this lead, after quite a bit of tuning and absolutely no tuning respectively, to test-MSE's a factor 2 to 3 lower than the best OLS-result from [1] after several thousand epochs. For the LeakyReLU we were not yet able to get

better than the best OLS-results, though we did manage to match them; as the test-MSE was varying wildly over our 100-epoch cycles, even for very small learning rates. Some small experimentation with adding a regularizaton penalty did not seem to improve matters, and was abandoned when it became apparent that the reconstructed terrain became very "blocky". Fig. **??** shows the MSE vs number of epochs ran for our FFNN using the ReLU activation function, a learning rate of 10^{-5}, $100 \times 100 \times 100$ hidden layers and no momentum or regularization parameters. This setup led to the 3-d plots in Fig. **??**. These should be compared with the best OLS results from [1], which can be seen reproduced in Fig. 12. Qualitatively the improvement is visible, but not overwhelming. Further results, including both LeakyReLU and Sigmoid can be found in the notebooks in the notebook-folder on our Github-repository. As a final note, we tried to do a similar analysis for the Franke-function; however we found that more extensive optimization of the hyper-parameters where needed to yield predictions comparable to those from OLS. That is not to say that our FFNN is incapable of producing better predictions; we merely haven't yet been able to tune it to do so.

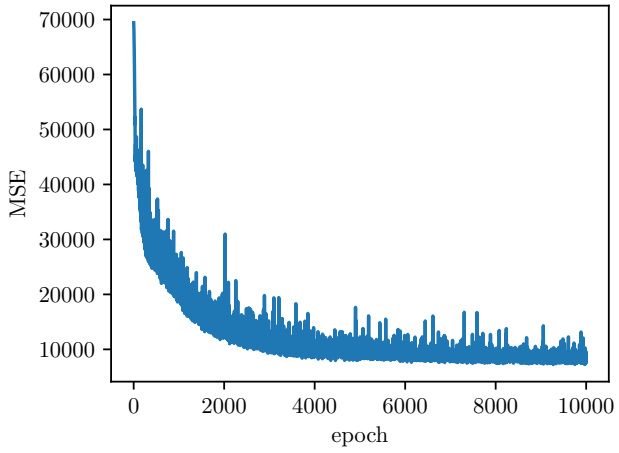


FIG. 4: MSE versus number of epochs run for our best FFNN prediction of the terrain data, using ReLU activation functions and a learning rate of 10^{-5}.

C. Classification of the MNIST dataset

We now turn to our analysis of the MNIST dataset [14], consisting of 70 000 images each 28×28 pixels of labeled handwritten numbers, a small subset of which is shown in Fig. 5. We wish to develop a model which can classify and assign a label to a previously unseen image as one of the 10 possible classes

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

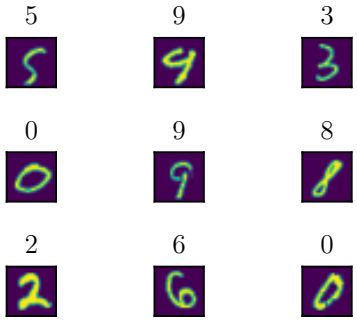
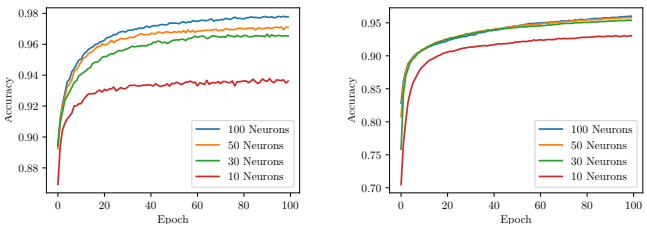


FIG. 5: Examples from the MNIST data set

by the use of a FFNN and logistic regression, then to compare the performance of these two approaches. We randomly split the complete dataset into sets of training and testing data at a ratio of 80:20 respectively. Further, the labels are converted to one-hot form.

1. *FFNN Classification*

We started by building a FFNN based classification model using the ReLU and Sigmoid activation functions for a single-layer neural network with varying number of neurons and the SoftMax activation function on the output layer, with its corresponding cost function Eqn. 27 as discussed in Sect. II C. We then trained the network for 100 epochs with $\eta = 0.01$, mini-batch size of 32 and no penalty or momentum, monitoring the MSE wrt. testing data for each epoch. The result is seen in Fig. 6. The



(a) ReLU (b) Sigmoid

FIG. 6: FFNN trained on the MNIST dataset for a single-layer NN of various sizes for using both the ReLU (a) and Sigmoid (b) activation functions in the hidden layer and SoftMax in the outer layer. Notably, the final score for ReLU with 100 neurons was ≈ 0.978

first thing we observe is the relatively large jump in accuracy score when going from 10 neurons to 30, then much less so in the subsequent steps. We postulate then that 10 neurons in a single layer structure simply isn't able to recreate the full feature set of the data set, but as the network grows, there is diminishing returns from adding

further neurons. To verify this suspicion, we performed the same training procedure using the ReLU activation function for $5, 10, 20$ and 30 layer networks, as seen in Fig. 7, which seems to confirm our postulate. We then

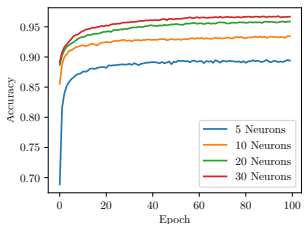
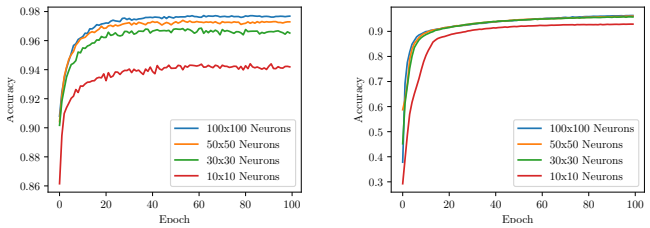


FIG. 7: FFNN trained on the MNIST data set with a single-layer of various sizes using the ReLU activation function in the hidden layer and the SoftMax in the outer layer.

explored the effects of adding additional layers to the network as summarized by Fig. 8 where we observe that there is no significant gain in adding an extra layer to the network of equal size, particularity for the larger sizes. We do however observe a minor increase in accuracy going from 10 neurons to 10×10 which is to be expected, given the increased flexibility of the network. However, for the larger networks, particularly the 100 neuron compared to 100×100 network, the difference in accuracy score where negligible. As a final curiosity, we attempted



(a) ReLU (b) Sigmoid

FIG. 8: FFNN trained on the MNIST dataset for a two-layer NN of various sizes for using both the ReLU (a) and Sigmoid (b) activation functions in the hidden layer and SoftMax in the outer layer.

to hand-tune a network towards a higher accuracy score by running the network for a small number of epochs, tuning the learning rate up/down according to the evolution of the accuracy in those epochs, similarly to what we did for the terrain data. We also experimented with tweaking the momentum and the addition of an L^2 regularization. However, much to our surprise we where not able to improve the accuracy score past 97.8%. Which speaks volumes of how well the FFNN is able to model the MNIST data with a very simple single layer network using only the basic SGD, no regularization and a constant learning rate in a relatively small number of epochs. This becomes all the more impressive when one considers that the human-level of accuracy in classifying this dataset is estimated to be $\approx 98\%$ [15].

As an additional note in regards to the way in which we initialized the weights of the network. During initial testing, when sampling our weights from the standard normal distribution we experienced rather slow convergence of our network, especially compared to Sci-Kit learns implementation. Reaching only an accuracy score of 90% after 100 epochs using the 100 single layer network. This motivated us to seek out alternative initialization schemes in the literature, which lead to us settling on the He [10] and Xavier [9] initialization schemes for ReLU and Sigmoid respectively, which greatly improved the performance of our networks.

2. *Classification with Logistic Regression*

For our logistic-regression implementation we do a very similar analysis of the MNIST data. We once again use a 80/20 train/test-split and investigate the effect of various hyper-parameters on the obtained accuracy score for 100 epochs. Fig. 9 shows the score of the test set for the indicated choices of hyper-parameters, with the notable absence of any regularization penalty. The only noteworthy change in setup from the neural network is how we initialize the regression parameters β. In the interest of simplicity we simply initialize them according to a standard normal distribution.We once again see that, for the better combinations of parameters, the performance fairly rapidly reaches fairly good scores, around 0.90 in these cases, before plateauing. Even the best combinations have not yet fully converged, but the convergence rate rapidly slows down significantly; indicating that many, many more iterations are required to match the performance of our neural network, if it can be matched at all. Though not shown here (but see the logreg notebook in the test folder at [16]), we have compared our results with the default logistic-regression classifier in Sci-Kit, and found nearly identical performance. Furthermore, the small discrepancy we did find vanishes completely when we downscale our β-initialization, similar to the He [10] scheme used in the FFNN, yielding a score of ≈ 0.92 as the best case after 100 epochs. It is eminently imaginable that further tuning could improve on this, but as it stands this is significantly worse than what we achieve with our neural network. We also take a look at the effects of adding a regularization parameter to our better choices of learning rates and mini batch-sizes. Fig. 10 shows the accuracy score on the test set for 100 epochs, using the indicated hyper-parameters such as the regularization parameters λ and learning rates η, notably not adding a momentum hyper-parameter. We see that the regularization penalty does a remarkable job of killing the learning, keeping the results very stable around the result of the first epoch. Not shown here (but available for verification in the logreg notebook in [16]), we find that

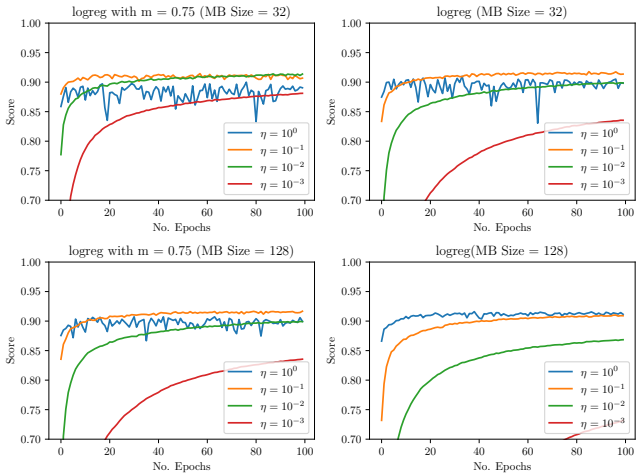


FIG. 9: Accuracy score on the test set for our logistic-regression method applied to the MNIST data set as a function of epochs run, for the indicated hyper-parameters and no regularization penalty.

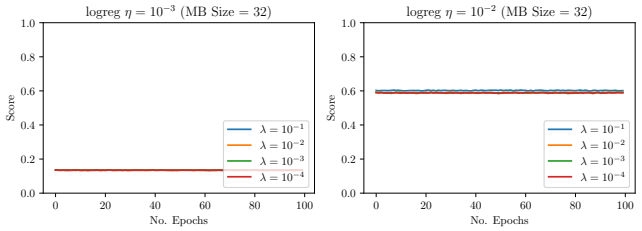


FIG. 10: Accuracy score on the test set for our logistic-regression method applied to the MNIST data set as a function of epochs runs for the indicated hyper-parameters and no momentum.

the addition of a momentum term of 0.75 does make our best combinations immediately (i.e. already after just one epoch) start, and stay stable, at an accuracy score of $\approx$ 0.90-0.92, consistent with the default performance of Sci-Kit's logistic-regression classifier.

In summary, we find that logistic-regression classifiers are capable of very quickly getting quite reasonable (accuracy score on test set of ≈ 0.91) performance, but that further refinement is slow as molasses without further fine-tuning. Meanwhile our neural network can fairly easily achieve a significantly better performance in the same number of epochs. The only advantage of the logistic-regression classifier seems to be that it can pretty much start (after one epoch of training) at close to its best performance when using regularization and momentum, while the neural network does require some epochs of training before it really gives superior performance.

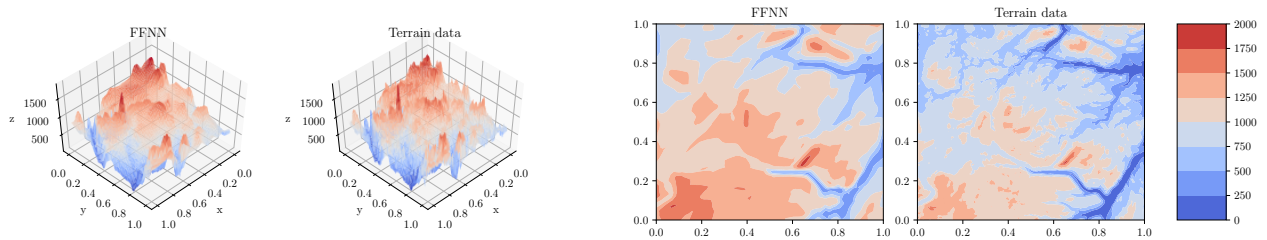
IV. CONCLUSION

With the massive flexibility and freedom granted by the direct dearth of hyper-parameters to consider, it is very difficult to make definite statements about which method is the optimal one for each problem. There may very well be far superior combinations of hyper-parameters, both in terms of speed and accuracy, than those we have tested for all the methods we have investigated in this paper. We can, however, highlight some considerations and tendencies based on our experiences throughout this work.

In summary, if you are pressed for time and want something that very easily just works, and works pretty well, your best bet seems to be linear-regression methods for predicting terrain-type data. There is improvement to be had from neural networks, especially our FFNN using ReLU as activation functions achieved a 2-3 times better MSE than our previously best result from OLS with some, though not too much, hassle; but the cost in time spent tuning is in general steep. The one clear advantage of FFNN for these linear-regression type problems is that they forego the need for constructing a model. This is, however, a double-edged advantage because often one is not only interested in the hands-down "best" prediction in terms of accuracy, but one might rather be interested in the model behind the predictions, which for neural networks often can be described as a "black box". For the classification case both the neural network and logistic regression are fairly easy to get going well, but the neural network does have a marked performance advantage over the logistic-regression classifier, with minimal mucking around required.

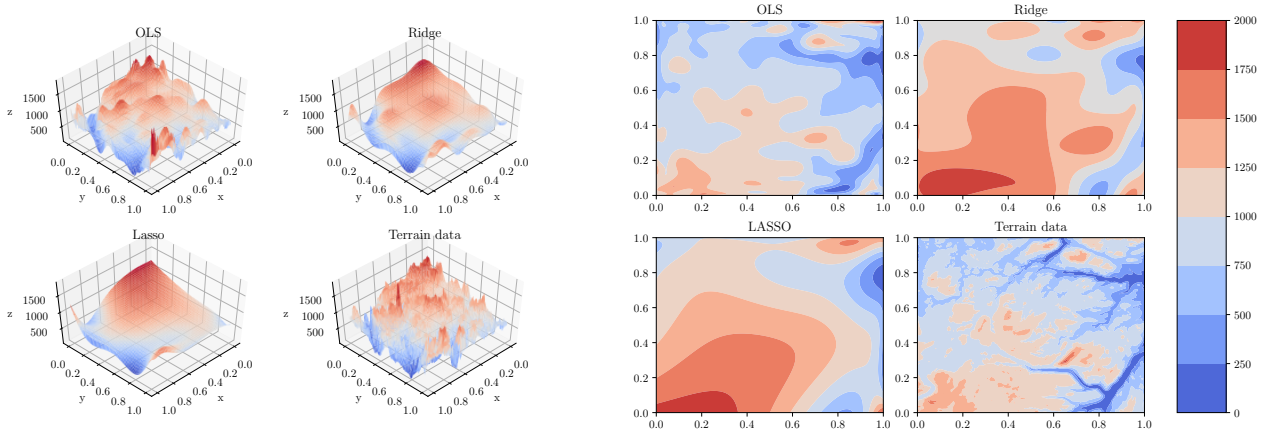
[1] N. Karlsen and T. Espedal Moe, Regression analysis and resampling methods (2020).
[2] P. Mehta, M. Bukov, C.-H. Wang, A. G. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, A high-bias, low-variance introduction to machine learning for physicists, Physics Reports **810**, 1–124 (2019).
[3] O. Shamir, Without-replacement sampling for stochastic gradient methods: Convergence results and application to distributed optimization (2016), arXiv:1603.00570 [cs.LG].
[4] D. Nagaraj, P. Jain, and P. Netrapalli, SGD without replacement: Sharper rates for general smooth convex functions (2019) pp. 4703–4711.
[5] G. Carleo, Machine learning techniques for quantum many-body physics - lecture 1 (2017).
[6] C. C. Aggarwal, *Neural Networks and Deep Learning* (Springer, 2018).
[7] J. Zhang and I. Mitliagkas, Yellowfin and the art of momentum tuning (2018), arXiv:1706.03471 [stat.ML].
[8] M. A. Nielsen, Neural networks and deep learning (2015).

[9] X. Glorot and Y. Bengio, Understanding the difficulty of training deep feedforward neural networks (JMLR Workshop and Conference Proceedings, Chia Laguna Resort, Sardinia, Italy, 2010) pp. 249–256.
[10] K. He, X. Zhang, S. Ren, and J. Sun, Delving deep into rectifiers: Surpassing human-level performance on imagenet classification (2015), arXiv:1502.01852 [cs.CV].
[11] Cs231n convolutional neural networks for visual recognition.
[12] K. Hornik, M. Stinchcombe, and H. White, Multilayer feedforward networks are universal approximators, Neural Networks **2**, 359 (1989).
[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, Scikit-learn: Machine learning in Python, Journal of Machine Learning Research **12**, 2825 (2011).
[14] Y. LeCun, C. Cortes, and C. Burges, Mnist handwritten digit database, ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist **2** (2010).
[15] Week 41 tensor flow and deep learning, convolutional neural networks.
[16] Source of terrain data.



(a) (b)

FIG. 11: 3-d plots and contour plots for our best achieved prediction of the terrain data using FFNN with ReLU, contrasted with the actual terrain data.



(a) (b)

FIG. 12: Main results for the terrain data in Project one, where the terrain data was modeled as a 25th order polynomial using OLS, Ridge and LASSO regression with optimized penalty parameters compared with the true terrain data