# A Comparative Study of Neural Nets and Vanilla Methods in Regression and Classification Tasks

René Ask, Kaspara Skovli Gåsvær & Maria Linea Horgen

(Dated: November 10, 2020)

We present an overview of gradient descent methods which are applied to minimize the cost-function of several supervised machine learning models. We compare matrix inversion and stochastic gradient descent (SGD) with and without momentum in the context of linear regression and find that matrix inversion slightly outperforms SGD. This is due to the fact that the linear models permit closed-form solutions. A model for multinomial logistic regression is presented and tested on the MNIST dataset using SGD with and without momentum, and the Adam optimizer. We find parameters that achieve an accuracy of $\sim 92-93\%$. We present the theory of neural nets. In regression tasks, the neural net is found to perform slightly better than the linear regression models. In the classification tasks, the model outperforms the multinomial logisitic regression model by achieving an accuracy of $\sim 98\%$ on both the validation and test sets.

## I. INTRODUCTION

Linear regression models such as ordinary least-squares (OLS) and Ridge regression are powerful methods to find best fit models to functions. But they may generalize poorly to unseen data as found here [1]. Today, the neural nets or neural networks has become widely popular for solving regression problems efficiently. This report extends the analysis from our previous work with linear regression methods in [1], where the observed data are assumed to stem from an underlying continuous function. The extension comes in the form of performing categorical predictions on a given dataset, which is referred to as classification.

We will study how two different methods of stochastic gradient descent, namely SGD with mini-batches, with- and without momentum, perform on a regression problem by comparing it with our regression analysis in [1]. The article performs OLS and Ridge regression on a dataset of Franke's function [3]. The analysis includes finding optimal values for an assemblage of hyperparameters and comparing performance metrics.

We'll develop and implement a multinomial logistic regression model for classification which we'll train, evaluate and test on the MNIST dataset [5]. We'll fit the model using SGD with and without momentum, and the Adam optimizer. We'll perform several grid searches in hyperparameter spaces to optimize the performance of the model to be compared with the performance of the neural net.

Finally, we'll develop the theory of neural nets. We'll discuss the basic model, choice of cost-function, top layer activation and the backpropagation algorithm where we add $L_2$ regularization and momentum. We'll delve into initialization of the model's parameters and choice if hidden activation functions. We'll methodically perform grid searches through hyperparameter spaces to learn how to pick a model that generalizes well to unseen data. We'll further compare the model to linear regression for regression tasks using Franke's function [3]. We'll also compare the model to the multinomial logistic regression model using the MNIST dataset.

## II. FORMALISM

### A. Gradient Descent

Assume we are given a dataset $\mathcal{D} = (X, y)$ of length $n$, where $X$ is some observed real data and $y$ is the response variable. The goal is to explain $y$ through a model $f(X, \theta)$, which is a function of the parameters $\theta$. To determine how well the model describes the observed data, we define a cost function $E(\theta)$. Choosing the parameters in such a way that they minimize the cost function is the core of many machine learning algorithms. In gradient descent (GD) the parameters are initialized to some value $\theta_0$, and then iteratively updated in the direction of the steepest descent through the updating scheme

$$\begin{aligned} v_t &= \eta_t \nabla_\theta E(\theta_t), \\ \theta_{t+1} &= \theta_t - v_t. \end{aligned} \tag{1}$$

Here $\nabla_\theta E(\theta_t)$ is the gradient of $E$ with respect to the parameters $\theta$ at iteration step $t$ and $\eta_t$ is the learning rate which controls the step size we move in the direction of the gradient.

Determination of the learning rate requires several aspects to be taken into consideration: choosing $\eta_t$ too small is computationally expensive, i. e. the number of iterations needed to converge to the local minimum becomes too large. Depending on the convexity of our cost function $E(\theta)$, the method will usually converge towards a (possible) local or global minimum. With a high learning rate we risk overshooting the minimum, so choosing $\eta_t$ wisely is crucial to obtain satisfactory results.

There are several variants of gradient decent methods. In this report we confine ourselves to three: stochastic gradient descent with mini-batches with- and without momentum and Adam which makes use of the second moment of the gradient as well. There are a few drawbacks with GD: computing the gradient for all datapoints when $n$ is large is highly time consuming. Furthermore, it lacks stochasticity and treats all directions in the parameter space uniformly, i. e. the learning rate is constant.

#### 1. *Stochastic Gradient Descent with mini-batches*

To incorporate randomness in GD the gradient is approximated on subsets of the observed data, called *mini-batches*, rather than being calculated on the full dataset. The batch size is usually significantly smaller than $n$, normally ranging from a ten to a couple of hundred datapoints [6].

Denoting the batch size as $B$, there are $B_k$ mini-batches for $k = 1, ..., n/B$. A full iteration over all the $n/B$ mini-batches, or all the datapoints $n$, is called an *epoch*. The datapoints included in each mini-batch are randomly chosen, either with- or without replacement, and thus introduces the missing stochasticity which decreases the likelihood of our fitting algorithm to get stuck in isolated local minima. For the case with replacement, some datapoints might be picked several times during an epoch, while others may not be picked at all. This approach generally converges more rapidly than the one without replacement [4].

When taking a gradient descent step, the steepest descent direction is now approximated by a mini-batch $B_k$, as opposed to the full dataset, meaning

$$\nabla_\theta E(\theta) = \sum_{i=1}^{n} \nabla_\theta E_i(\theta) \rightarrow \nabla_\theta E^{MB}(\theta) = \sum_{i \in B_k} \nabla_\theta E_i(\theta), \tag{2}$$

with $\nabla_\theta E^{MB}(\theta)$ now being the mini-batch approximation. This tackles the time consumption problem that arises with the regular GD method. The updating scheme becomes

$$\begin{aligned} v_t &= \eta_t \nabla_\theta E^{\mathrm{MB}}(\theta_t), \\ \theta_{t+1} &= \theta_t - v_t. \end{aligned} \tag{3}$$

#### 2. *Adding momentum*

A modification which deals with the constant learning rate $\eta_t$ is to add a momentum term in the updating scheme,

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta_t \nabla_\theta E^{\mathrm{MB}}(\theta_t), \\ \theta_{t+1} &= \theta_t - v_t. \end{aligned} \tag{4}$$

The learning rate is still constant per se, but the momentum term serves as a memory of the previous time step by adding a fraction $\gamma \in (0, 1)$ of the update vector $v_{t-1}$ from the previous iteration step to the current update vector. This can be interpreted as a weighted mean of recent gradients, which means the GD algorithm will move in the direction of small and presistent gradients while suppressing the effect of sudden oscillations which could result from regular SGD due to the stochasticity of the update rule. This will in many cases increase the convergence rate. However, the algorithm can in some cases overshoot the local minimum for some choices of $\gamma$ and $\eta_t$, which makes the fitting algorithm oscillate about some local minima and decreases the convergence rate.

#### 3. *ADAM: Adding the second moment*

Another way to optimize GD is to incorporate the second moment of the gradient as well as the first. Adding these to the updating scheme provides us with an algorithm known as Adam and looks like

$$\begin{aligned} g_t &= \nabla_\theta E^{\mathrm{MB}}(\theta_t), \\ m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t, \\ s_t &= \beta_2 s_{t-1} + (1 - \beta_2) g_t^2, \\ \alpha_t &= \eta_t \frac{\sqrt{1 - (\beta_2)^t}}{1 - (\beta_1)^t}, \\ \epsilon_t &= \epsilon \sqrt{1 - (\beta_2)^t}, \\ \theta_{t+1} &= \theta_t - \alpha_t \frac{m_t}{\sqrt{s_t} + \epsilon_t}, \end{aligned} \tag{5}$$

where $m_t$ is the average first moment of the gradient, $s_t$ is the average second moment of the gradient, $\alpha_t$ is the scaled learning rate at iteration step $t$ and $\epsilon_t$ is a parameter to avoid division by zero. The factors $\beta_1$ and $\beta_2$ controls the decay rates of the averages and are typically set to values close to 1.0 [6]. In few words one can think of Adam as a combination of momentum and adaptive learning rate which in turn leads to faster convergence as the weights are updated more efficiently. The Adam optimizer will only be incorporated into our multinomial logistic regression model when looking at classification problems.

### B. Linear Regression: SGD vs. Matrix Inversion

As mentioned in the introduction, we will compare the results from our regression analysis in [1] for Frankes function with OLS and Ridge regression with the ones obtained with SGD. Which implies that we perform an analysis on the optimal choices of polynomial degree, number of epochs, mini-batch size and learning rate $\eta$. An analysis of the polynomial degree is implicit an analysis on the number of features $p$ included in our model. The main result for comparison is the optimal mean square error (MSE) obtained as a function of polynomial degree, and for Ridge regression also the hyperparameter $\lambda$. For a thorough walk-through on how we preprocess the data and the theory behind linear regression, we refer to our previous work in [1]. After determining the optimal parameters, we also perform a comparison between SGD with and without momentum.

In regression problems, the model $f(X, \theta)$ can be expressed as a matrix-vector product

$$f(X, w) = Xw, \tag{6}$$

with $X$ as the design matrix and $w = \theta$ as the weights in the linear model. The cost function we seek to minimize has two slightly different definitions, given by the regression type, yielding two different expressions for the gradient:

$$\begin{aligned}\nabla_w E_{\text{OLS}} &= \frac{\partial}{\partial w}\left(\frac{1}{2}\|z - Xw\|_2^2\right), \\ &= -X^T z + X^T X w, \end{aligned} \tag{7}$$

$$\begin{aligned}\nabla_w E_{\text{Ridge}} &= \frac{\partial}{\partial w}\left(\frac{1}{2}\|z - Xw\|_2^2 + \frac{\lambda}{2}\|w\|_2^2\right), \\ &= -X^T z + X^T X w + \lambda w. \end{aligned} \tag{8}$$

Both $E_{\text{OLS}}$ and $E_{\text{Ridge}}$ are convex functions, see section VI A for a simple proof. Any local minimum of a convex function is also a global minimum, thus the solution obtained from SGD will necessarily converge towards the global minimum.

## C. Logistic regression

### 1. *Preliminaries on classification*

Logistic regression is a supervised learning method used to predict discrete outputs $y \in \{0, 1\}$ given an input $x \in \mathbb{R}^p$. Problems of this kind is known as classification problems. Classification problems can consist of a single class, where the task is to predict whether $x$ belongs to a category $(y = 1)$ or not $(y = 0)$. More generally, classification problems can consist of $M$ classes such that the output $y \in \{0, 1\}^M$ belongs to a single class $j$ represented by what is known as a *one-hot vector* where

$$y_i = \begin{cases} 1 & \text{if} \quad i = j \\ 0 & \text{else} \end{cases} \tag{9}$$

The task is to predict which class $j$ a given input $x$ belongs to. In the context of logistic regression, a model that solves such a classification problem is called *multinomial.*

Classification problems may be divided further as follows:

- *Hard* classification in which the predicted values are only either 0 or 1. In this case, the model is called a hard classifier.
- *Soft* classification where the predicted values are on the real interval $[0, 1]$. The model is called a soft classifier. A soft classifier will yield a probability of whether $x$ belongs to a class or not.

We'll restrict ourselves to implement a multinomial logicstic regression model (also known as a softmax model) which is a soft classifier.

### 2. *Multinomial Logicistic Regression*

Suppose our classification problem consists of $M$ classes. Our task is to predict which class $m$ a given input $x$ belongs to. The nuts-and-bolts of the model is a single weight matrix $W \in \mathbb{R}^{p \times M}$ and a bias $b \in \mathbb{R}^M$. The output of the model $\hat{y} \in [0, 1]^M$ is modelled by the so-called *Softmax* function

$$\hat{y}_j = \sigma(z_j) = \frac{e^{z_j}}{\sum_{m=1}^M e^{z_m}}. \tag{10}$$

where $z_j = \sum_i W_{ji} x_i + b_j$, where $j$ refers to class $j$.

The corresponding cost-function used with this model is the so-called cross-entropy which for a single datapoint $(x, y)$ with $x \in \mathbb{R}^p$ and $y \in \{0, 1\}^M$, is given by

$$E = -\sum_{m=1}^M y_m \log \hat{y}_m + (1 - y_m) \log(1 - \hat{y}_m). \tag{11}$$

### 3. *Gradients of the cost-function*

The gradients for a single data point are

$$\frac{\partial E}{\partial W_{jk}} = (\sigma_j - y_j) x_k. \tag{12}$$

Similarly, for the bias term, we get

$$\frac{\partial E}{\partial b_j} = \sigma_j - y_j. \tag{13}$$

See section VI B for a more thorough derivation of the gradients.

## D. Neural Nets

Neural nets (or neural networks) are nonlinear machine learning models for supervised learning. The basic building blocks of the model class are a set of layers, each of which has a set of neurons (or nodes). These layers are usually divided into three categories:

- The *input* layer which is where a real observable $x \in \mathbb{R}^p$ is fed to the network.
- *Hidden* layers which is where the processing of the information happens.

- The *output* or *top* layer where a response or activation $y$ associated with the observable is realized.

Between any two adjacent layers, there's a set of weights connecting a neuron from one layer to every neuron in the adjacent layer. Typically, a bias is added to each neuron in every layer to help facilitate activation of the neuron.

### 1. Basic mathematical formalism

Denote a layer in the neural network by $l$. Let there be $L$ layers in total such that $l \in \{1, 2, ..., L\}$. Denote the activation of neuron $j$ in layer $l$ by $a_j^l$ and its corresponding bias as $b_j^l$. Finally, let $W_{jk}^l$ be the weights connecting neuron $k$ in layer $l-1$ to neuron $j$ in layer $l$. The activation $a_j^l$ is modelled through the (in general) nonlinear equation

$$a_j^l = \sigma\left(\sum_k W_{jk}^l a_k^{l-1} + b_j^l\right) \equiv \sigma(z_j^l), \tag{14}$$

where $\sigma(\cdot)$ is some (possibly) nonlinear function colloquially called the *activation function*. We'll come back to a discussion on common choices later on.

For $l = 1$, eq. (14) specializes to

$$a_j^1 = \sigma\left(\sum_k W_{jk}^1 x_k + b_j^1\right), \tag{15}$$

where $x \in \mathbb{R}^p$ is the input vector containing $p$ features.

### 2. Training a neural network with the backpropagation algorithm

A neural network can be represented by a function of the form

$$f = f_1 \circ f_2 \circ \cdots \circ f_L, \tag{16}$$

who's gradient carry a computational complexity which makes a direct calculation intractable, in general. A clever circumvention is to create a set of equations defining a recursive relationship such that knowledge of the error in any layer $l+1$ permits us to deduce of the error in layer $l$. This is known as the *backpropagation algorithm*. We'll simply recite the important equations and their meaning, and refer the reader to a derivation here [6].

The backpropagation boils down to four equations which we'll describe in the following. Suppose $E$ is some cost-function and let $\Delta_j^l$ denote the error at neuron $j$ in layer $l$. The first of the four equations is the error at layer $L$ which is defined as

$$\Delta_j^L = \frac{\partial E}{\partial z_j^L} \tag{17}$$

The second equation is a recursive equation which allows us to compute the error at layers $l = L-1, ..., 1$ from $\Delta_j^L$. The equation is given by

$$\Delta_j^l = \left(\sum_k \Delta_k^{l+1} w_{kj}^{l+1}\right) \sigma'(z_j^l), \tag{18}$$

where $\sigma'(\cdot)$ denotes the derivative of $\sigma(\cdot)$.

The third equation relates the errors to the gradient of the cost-function with respect to the biases which is

$$\frac{\partial E}{\partial b_j^l} = \Delta_j^l. \tag{19}$$

Finally, the fourth equation relates the errors to the gradient with respect to the weights:

$$\frac{\partial E}{\partial W_{jk}^l} = \Delta_j^l a_k^{l-1} \tag{20}$$

Before we can list the complete backpropagation algorithm, we need an explicit expression for $\Delta_j^L$, which the next section delves into.

### 3. Top layer activation and choice of cost-function

The rate at which a neural network learns, is highly dependent on the combination of top layer activation and cost-function. We will consider two cost-functions depending on the problem at hand.

**Classification** For classification, a natural choice for the top layer activation function is the Softmax function in eq. (10), which would make the neural network a soft classifier. A common choice for the cost-function is then the cross-entropy function, which for a neural network becomes

$$E = -\sum_m \left[y_m \log a_m^L + (1 - y_m)\log\big(1 - a_m^L\big)\right]. \tag{21}$$

Starting from eq. (17), the error at layer $L$ can be computed as

$$\begin{aligned}\Delta_j^L &= \frac{\partial E}{\partial z_j^L} = \frac{\partial E}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} \\ &= -\sum_m \left[\frac{y_m}{a_m^L} - \frac{1 - y_m}{1 - a_m^L}\right]\delta_{jm} a_j^L (1 - a_j^L) \\ &= a_j^L - y_j,\end{aligned} \tag{22}$$

meaning the error is simply given by the difference between the prediction $a_j^L$ and the ground truth $y_j$.

**Regression** For regression, a natural choice for top layer activation is simply a linear relationship

$$a_j^L = z_j = \sum_k W_{jk}^L a_k^{L-1} + b_j^L. \tag{23}$$

In this case, choosing the cost-function (for a single data point) to be

$$E = \frac{1}{2} \sum_j (y_j - a_j^L)^2, \tag{24}$$

gives the following error at layer $L$

$$\Delta_j^L = \frac{\partial E}{\partial z_j^L} = \frac{\partial E}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = a_j^L - y_j, \tag{25}$$

which is exactly the same as eq. (22).

Now we're ready to list the backpropagation algorithm in 1 for a single datapoint.

**Algorithm 1** Backpropagation

```
$a_j^0 = x_j$    for    $j = 1, ..., p$                    ▷ Initialize input
Feed-forward
for $l = 1, 2, .., L-1$ do
    for $j = 1, 2, .., n$ do
        $a_j^l \leftarrow \sigma\left(\sum_k W_{jk}^l a_k^{l-1} + b_j^l\right)$
for $j = 1, 2, .., m$ do                                   ▷ $m$ outputs
    $a_j^L \leftarrow \sigma\left(\sum_k W_{jk}^L a_k^{L-1} + b_j^L\right)$
Backward pass
for $j = 1, 2, .., m$ do                                   ▷ $m$ outputs
    $\Delta_j^L \leftarrow a_j^L - y_j$
    $\partial E / \partial b_j^L \leftarrow \Delta_j^L$
    $\partial E / \partial W_{jk}^L \leftarrow \Delta_j^L a_k^{L-1}$
for $l = L-1, ..., 1$ do
    for $j = 1, ..., n$ do
        $\Delta_j^l \leftarrow \left(\sum_k \Delta_k^{l+1} W_{kj}^{l+1}\right) \sigma'(z_j^l)$
        $\partial E / \partial b_j^l \leftarrow \Delta_j^l$
        $\partial E / \partial W_{jk}^l \leftarrow \Delta_j^l a_k^{l-1}$
```

### 4. *Adding $L_2$ regularization*

A common strategy to increase the neural nets' ability to generalize is to add $L_2$-regularization. This can be achieved by defining the new cost-function

$$E' = E + \frac{\lambda}{2} \sum_{lij} (W_{ij}^l)^2, \tag{26}$$

where $E$ is the un-regularized cost-function and $\lambda$ is the regularization parameter. This simply adds an additional term to the update rule for the weights:

$$\frac{\partial E}{\partial W_{jk}^l} = \Delta_j^l a_k^{l-1} + \lambda W_{jk}^l. \tag{27}$$

### 5. *Adding momentum*

Adding momentum to the update rules of the neural net as described by eq. (4). This adds a new hyperparameter $\gamma$ to the model, which modifies the update rules for the weights of the neural net as

$$\begin{aligned} V_{ij,t}^l &= \gamma V_{ij,t-1}^l + \eta \frac{\partial E}{\partial W_{jk}^l} \\ W_{jk}^l &\leftarrow W_{jk}^l - V_{ij,t}^l \end{aligned} \tag{28}$$

and similarly for the biases

$$\begin{aligned} v_{j,t}^l &= \gamma v_{j,t-1}^l + \eta \frac{\partial E}{\partial b_j^l} \\ b_j^l &\leftarrow b_j^l - v_{j,t}^l \end{aligned} \tag{29}$$

where $V_{ij}^l$ and $v_j^l$ are introduced to store the momentum of the weights and biases, respectively. The parameter $t$ keeps track of which iteration the algorithm is at.

### 6. *The complete neural net algorithm*

The backpropagation algorithm with mini-batch size $B$, $L_2$-regularization and momentum is as shown in algorithm 2.

**Algorithm 2** Backpropagation of a mini-batch with $L_2$-regularization and momentum $\gamma$

```
for $b = 1, 2, ..., B$ do                                  ▷ Loop over the mini-batch
    $a_j^0 = x_j$    for    $j = 1, ..., p$                ▷ Initialize input
    Feed-forward
    for $l = 1, 2, .., L-1$ do
        for $j = 1, 2, .., n$ do
            $a_j^l \leftarrow \sigma\left(\sum_k W_{jk}^l a_k^{l-1} + b_j^l\right)$
    for $j = 1, 2, .., m$ do                               ▷ $m$ outputs
        $a_j^L \leftarrow \sigma\left(\sum_k W_{jk}^l a_k^{L-1} + b_j^L\right)$
    Backward pass
    for $j = 1, 2, .., m$ do                               ▷ $m$ outputs
        $\Delta_j^L \leftarrow a_j^L - y_j$
        $\partial E / \partial b_j^L \leftarrow \partial E / \partial b_j^L + \Delta_j^L$
        $\partial E / \partial W_{jk}^L \leftarrow \partial E / \partial W_{jk}^L + \Delta_j^L a_k^{L-1}$
    for $l = L-1, ..., 1$ do
        for $j = 1, ..., n$ do
            $\Delta_j^l \leftarrow \left(\sum_k \Delta_k^{l+1} W_{kj}^{l+1}\right) \sigma'(z_j^l)$
            $\partial E / \partial b_j^l \leftarrow \partial E / \partial b_j^l + \Delta_j^l$
            $\partial E / \partial W_{jk}^l \leftarrow \partial E / \partial W_{jk}^l + \Delta_j^l a_k^{l-1}$
Update weights and biases
for $l = 1, ..., L$ do
    for $j = 1, ..., N$ do
        for $k = 1, ..., M$ do
            $V_{ij,t}^l \leftarrow \gamma V_{ij,t-1}^l + \eta\left(\partial E / \partial W_{jk}^l / B + \lambda W_{jk}^l\right)$
            $W_{jk}^l \leftarrow W_{jk}^l - V_{ij,t}^l$
        $v_{j,t}^l \leftarrow \gamma v_{j,t-1}^l + (\eta / B)\, \partial E / \partial b_j^l$
        $b_j^l \leftarrow b_j^l - v_{j,t-1}^l$
```

#### 7. Hidden activation functions

Hidden activation functions play an important role when it comes to the neural nets' ability to learn. The Sigmoid function was a common choice for a long time, but suffers the from the fact that $\sigma'(z) \approx 0$ for large $z$. Since $\sigma'(z)$ plays an important role in the backpropagation algorithm (see eq. (18)), this may cause a problem famously known as the *the vanishing gradient* problem. Two commonly introduced activation functions can remedy this problem. The first one is the so-called *rectified linear unit* (ReLU) which is given by

$$\sigma(z) = (z)^+ = \max(0, z). \tag{30}$$

The second activation function we'll study the effects of is known as the leaky ReLU given by

$$\sigma(z) = \begin{cases} z & \text{if} \quad z > 0, \\ 0.01z & \text{else.} \end{cases} \tag{31}$$

Both of these avoid the problem of $\sigma'(z) \to 0$ for large $z$.

#### 8. Initialization of weights biases

Initialization of weights has an important effect on performance of the neural net. The fundamental problem is the sum

$$z_j^l = \sum_k W_{jk}^l a_k^{l-1} + b_j^l, \tag{32}$$

which can become very large (in absolute value) if the sum consists of many terms. This in combination with large weights can exacerbate the vanishing gradient problem further. A solution to this problem is to decrease the likelihood that the sum becomes large. To robustly implement this, we can implement the weights according to $\mathcal{N}(0,1)$, but we *scale* all the elements in $W_{jk}^l$ with the squared root of the number of elements in the sum above (which is the same as the square root of the number of columns in the weight matrix). If we loosely consider $z_j^l$ as a distribution of possible values to be fed to the activation function, then we can interpret the scaling of the weights in this way to reduce the width of this distribution, which reduces the likelihood of large (absolute) values of $z$.

The biases can be initialized according to $\mathcal{N}(0,1)$ because they do not present the same scaling problem, i.e they do not scale with the complexity of the model.

#### 9. Brief note on model complexity

The model complexity of a neural net can in a sense be boiled down to how many parameters the model needs. Let $P$ denote how many parameters the model needs. Let $L$ be the number of hidden layers, $n$ be number of nodes in each layer, $p$ be the features and $m$ be number of outputs. Then the model has the following number of parameters:

$$P = n(p+1) + (L-1)n(n+1) + m(n+1), \tag{33}$$

where the first term is the first hidden layer, the second term describes all hidden layers which are simply connected to another hidden layer, and the last term is the parameters need in the top layer. One thing to take notice of is that increasing how many hidden layers the model increases the model complexity more than simply keeping it fixed while increasing how many hidden neurons the model has.

#### 10. Neural nets: Methods

For regression, we'll study how the model complexity of a neural net affects its performance. We'll perform grid searches over hyperparameter spaces to further tune the model and compare its performance to the linear regression models that uses matrix inversion and iterative methods such as SGD with and without momentum.

For classification, the features are determined by the dataset, which sets a slight restriction on the model complexity. This simplifies the study of model complexity, which will be performed in addition to grid searches over hyperparameter spaces. The resulting models will be compared to the performance of the logistic regression model.

## E. Performance metrics

#### 1. Regression

In regression problems, we'll use two different performance metrics. The first one is the mean-squared error defined as

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \tag{34}$$

where $n$ is the number of datapoints, $y_i$ is the response variable and $\hat{y}_i$ is the predicted value. The second is the so-called $R^2$-score defined as

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}, \tag{35}$$

where $\bar{y}$ denoted the arithmetic mean of the responses $y$.

#### 2. Classification

For classification, the natural metric is accuracy which is defined as

$$\text{accuracy} = \frac{\text{correct predictions}}{\text{total predictions}}, \tag{36}$$

# III. DATASETS

## A. Franke's function

The dataset used to study regression problems are generated using Franke's function [3] (denoted $f(x, y)$). The dataset consists of 20000 tuples $(x, y, z(x, y))$ where the response variable is generated using

$$z(x, y) = f(x, y) + \epsilon, \tag{37}$$

where $\epsilon \sim \mathcal{N}(0, 1)$. The dataset is split into 90 % training data and 10 % test data. 5 % of the training data is used as a validation set, while the remaining tuples are used for training.

## B. MNIST hand-written digits

The MNIST dataset [5] consists of hand-written images of numerical characters 0-9. The dataset is made up of 60000 training examples and 10000 examples. In this article, we split the training data into a new training set of 57000 examples and a validations set of 3000 examples. The dataset sizes are summarized in table I.

| Training size | Validation size | Test size |
|---|---|---|
| 57000 | 3000 | 10000 |

TABLE I. The table shows the data sizes of the MNIST dataset used in this article. The original training set of 60000 examples is split into a new traning set of 57000 examples and a validation set of 3000 examples.

# IV. RESULTS & DISCUSSION

## A. Stochastic Gradient Descent

The following results are produced from a dataset of Franke's function with $n = 20000$ and $\sigma = 0.1$, when applying SGD to minimize the $E_{\mathrm{OLS}}$, except in the case were we determined the polynomial degree.

### 1. *SGD vs. Matrix Inversion for OLS*

Determining the polynomial degree $d$ which yielded the lowest MSE on the validation set was done through creating a heatmap for $d \in \{1, ..., 6\}$, training our model for number of epochs in the range $[1, 700]$ and mini-batch sizes between $[1, 100]$, and evaluating the MSE on a validation set, with $\eta = 10^{-5}$. Due to a high number of calculations for each polynomial degree, the analysis was performed on a dataset with $n = 1000$. The minimum MSE values obtained from all of the polynomial degrees are shown in table II.

| Polynomial degree $d$ | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| min(MSE) | 0.342 | 0.272 | 0.188 | 0.147 | 0.131 | 0.129 |

TABLE II. The table shows the minimum mean square error obtained from different polynomial degrees $d$ where we have applied SGD on $E_{\mathrm{OLS}}$ with $\eta = 10^{-5}$. $d = 6$ yields the lowest MSE value.

From table II it is evident that $d = 6$ yielded the lowest MSE, hence the optimal value for $d$ is 6. Ideally the range of polynomial degrees should have been larger, but due to accumulation of high values of the gradient leading to overflow during our calculations, we had to limit ourselves to the given range. A workaround could have been to considerably downscale the initialized weights. The weights were divided by the number of features for the results presented.

After determining the polynomial degree, a more extensive heatmap, i. e. number of epochs in the range $[1, 1000]$, were produced to find optimal values for epochs and mini-batch size. Figure 1 depicts the heatmap.

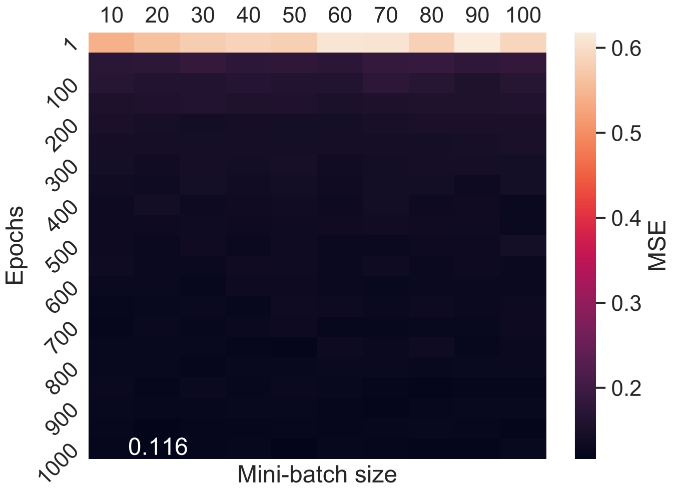

FIG. 1. The figure shows number of epochs and mini-batch size as a function of MSE for OLS regression, with $\eta = 10^{-5}$. The lowest MSE value is obtained from 1000 epochs and mini-batch size 20.

The optimal choice of epochs and mini-batch size is $(1000, 20)$, when $\eta = 10^{-5}$. Given the convexity of the cost function, a higher number of iterations, i. e. epochs, combined with a sufficiently small learning rate will always lead to better convergence. Meaning that it is no surprise that the optimal number of epochs corresponds to the maximum value on the given interval.

Due to lack of computing power leading to highly time consuming calculations, we estimated the number of epochs and mini-batch size for only one value of $\eta$. This is a weakness related to our analysis, and is proposed as a point of improvement for further work.

We applied the optimal values for epochs and mini-batch size to determine the preferred learning rate, which is depicted in figure 2.

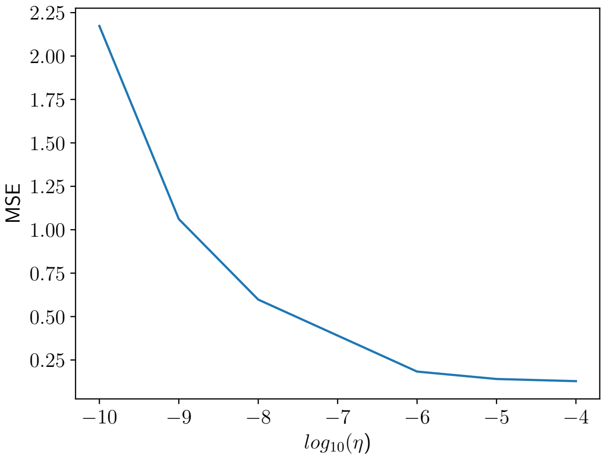

FIG. 2. The figure shows the MSE as a function of $\log_{10}(\eta)$ for $d = 6$, 1000 epochs and batch size of 20, when applying SGD on $E_{\text{OLS}}$ for the validation set. $\eta = 10^{-4}$ yields the lowest MSE.

The lowest MSE score was obtained with $\eta = 10^{-4}$. We summarize the optimal values for the various hyperparameters in table III before introducing the MSE value from the test set.

| Hyperparameter | Optimal value |
|---|---|
| Polynomial degree $d$ | 6 |
| Epochs | 1000 |
| Mini-batch size | 20 |
| Learning rate $\eta$ | $10^{-4}$ |

TABLE III. The table displays optimal values for the hyperparameters polynomial degree $d$, number of epochs, mini-batch size and learning rate $\eta$ when applying SGD on $E_{\text{OLS}}$.

Table IV displays the MSE value obtained from our regression analysis in [1], and the one found from applying SGD on $E_{\text{OLS}}$ for unseen test data.

| Method | MSE |
|---|---|
| Matrix inversion | 0.112 |
| SGD | 0.122 |

TABLE IV. The table displays MSE obtained from performing ordinary least squares regression and SGD on $E_{\text{OLS}}$ when applied to unseen test data for Franke's function. The lowest MSE values is reached when applying linear regression.

In our regression analysis the weights were found by solving eq. (6) for $w$, i. e. they are found from inverting a matrix. With this in mind, we expected the results from [1] to slightly outperform the results from SGD, which is exactly the behaviour observed. Given that SGD is an iterative method, and despite $E_{\text{OLS}}$ being a convex function, there is no guarantee that we will reach the exact solution during our numerical simulations. That is not to say that we will not converge towards it.

A note on the choice of performance metrics in our comparison with [1]: in our previous work we only evaluated the mean square error, and not the $R^2$ score. In general the $R^2$ provides a measure of how well the model replicate the observed values, and is thus more akin to accuracy. The MSE values let us only compare the two methods relative to each other, and not make general claims about the methods in itself. Thus we see that matrix inversion approach is in this particular case better at determining the weights $w$, compared to SGD.

To make a comparison between SGD and SGD with momentum, we determine the optimal choice of $\gamma$ as a function of the optimal parameters in table (III). Figure 3 displays the $R^2$ score as a function of $\gamma$.

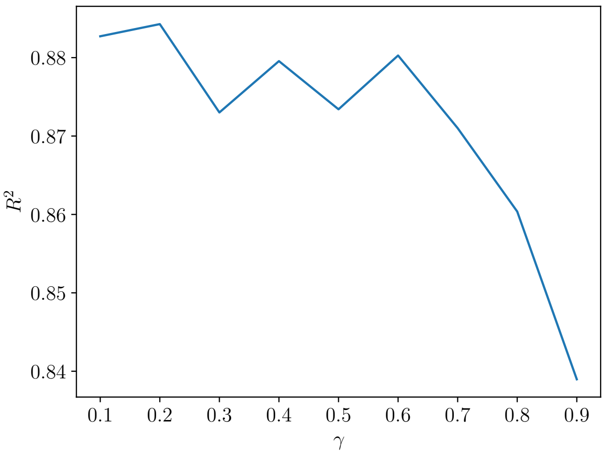

FIG. 3. The figure shows the $R^2$ score for SGD with momentum as a function of the momentum parameter $\gamma$. The highest $R^2$ score is obtained for $\gamma = 0.2$.

The highest $R^2$ score was reached for $\gamma = 0.2$. From the theory we expect SGD with momentum to dampen oscillations and converge at a faster rate. A comparison on how the cost function evolves with the number of epochs for the two SGD methods are displayed in fig. 4.

From fig. 4 we observe a confirmation of the theory, which is also reflected in the obtained value for the mean square error,

$$\text{MSE}_{\text{mom}} = 0.117, \tag{38}$$

where the subscript is related to the fact that we now look at SGD with momentum. A comparison with the MSE from regular SGD in table IV yields a 4% relative decrease in MSE when applying momentum.

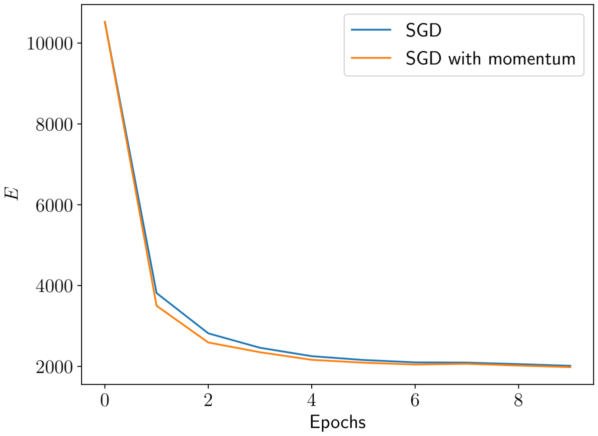

FIG. 4. The figure shows how the cost functions for SGD and SGD with momentum behave as a function of number of epochs. We observe a faster convergence rate for SGD with momentum.

### 2. *SGD vs. Matrix Inversion for Ridge Regression*

Since Ridge regression introduces a new hyperparamter $\lambda$, we restrict ourselves to only determine the optimal value of the above-mentioned hyperparameter. For Ridge regression we studied the $R^2$ score as a function of ($\eta$, $\lambda$), with the optimal values for number of epochs and mini-batch size found in section IV A 1. Figure 5 shows the described heatmap.

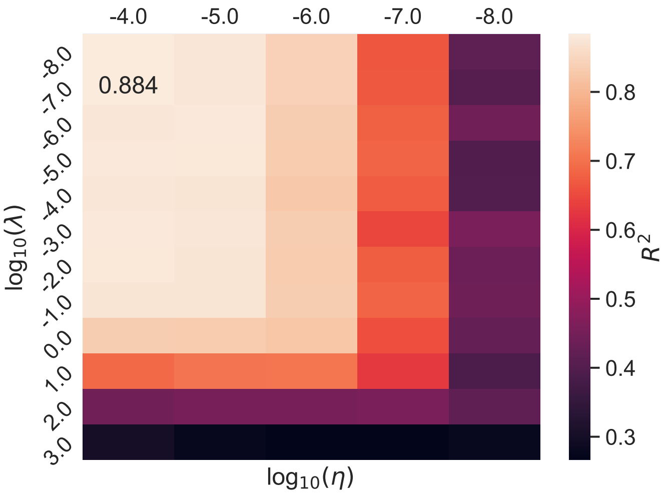

FIG. 5. The figure depicts the $R^2$ score as a function of ($\eta$, ) when applying SGD to $E_{\text{Ridge}}$. The highest $R^2$ score is obtained for $\lambda = 10^{-7}$ and $\eta = 10^{-4}$. Here the batch size was 20, with 1000 epochs for $d = 6$.

The highest $R^2$ score is found when ($\eta$, $\lambda$) = $(10^{-4}, 10^{-7})$. As $\lambda \to \infty$ the $R^2$ score goes to zero. Interpreting $\lambda$ as a penalty parameter for $\|w\|_2$, the weights will approach zero for increasing values of $\lambda$, which is exactly the behaviour the plot exhibits.

The computed MSE value for unseen test data with the parameters in table III and $\lambda = 10^{-7}$, compared to the MSE value found in [1], is shown in table V.

| Method | MSE |
|---|---|
| Matrix inversion | 0.112 |
| SGD | 0.116 |

TABLE V. The table displays MSE obtained from performing matrix inversion and SGD for Ridge regression when applied to unseen test data for Franke's function. It is clear that the matrix inversion approach yields the lowest MSE.

We observe the same behaviour as we did for OLS regression in table IV. The existence of an analytical expression gives the matrix inversion approach an advantage, which is the expected outcome. Comparing the mean square errors obtained with SGD, Ridge regression slightly outperforms OLS. Generally we know that adding $L_2$ regularization prevents overfitting, and ensures that the matrix $X^T X$ in eq. (8) is invertible. This can contribute to the difference in the observed performance.

## B. Logistic Regression

All results in this section is acquired using the MNIST dataset as listed in table I. To obtain optimal values of batch size and epochs we trained our logistic regression model using the SGD optimizer without momentum or regularization by performing a grid search over their parameter space and tested it on a validation set. The resulting heatmap is presented in figure 6.

The figure shows the model predicts with a relatively high accuracy, $91 - 92\%$, already for 10 epochs and with a batch size of 100. For larger values of both parameters the model seems stable and continues to predict with accuracy within this range. This could indicate that the model is good at generalizing to unseen data, an is not yet overfitting from the training set. For further work on this project it could be interesting to see how long the model predicts accurately, i. e. how much larger the number of epochs have to become before the model starts to make poor predictions on unseen datsets/starts to overfit the data.

To identify the optimal values for $\lambda$ and $\gamma$ when applying SGD with momentum and $L_2$-regularization, we utilized the optimal values from figure 6, batch size of 500 and 100 epochs. Figure 7 depicts the accuracy as a function of $(\gamma, \lambda)$ for the validation set.

In figure 7 there is a large region where the accuracy does not fluctuate significantly, meaning the model predicts with a stable accuracy around 92% for a number

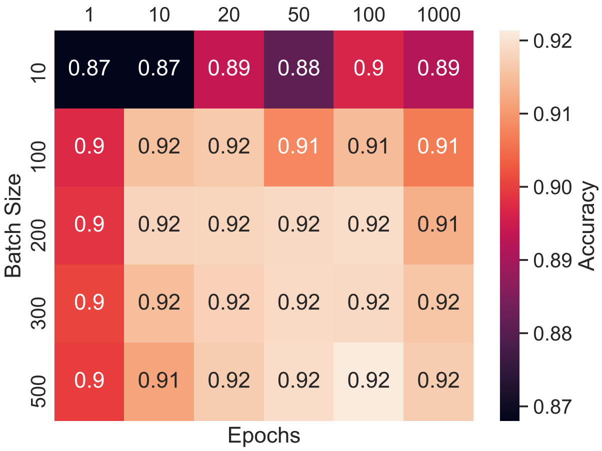

FIG. 6. The figure shows accuracy as a function of epochs and batch size when our logistic regression model is trained on a training set using the SGD optimizer with learning rate $\eta = 0.1$ and no regularization term on a validation set. Batch size of 500 and 100 epochs yielded the highest accuracy.

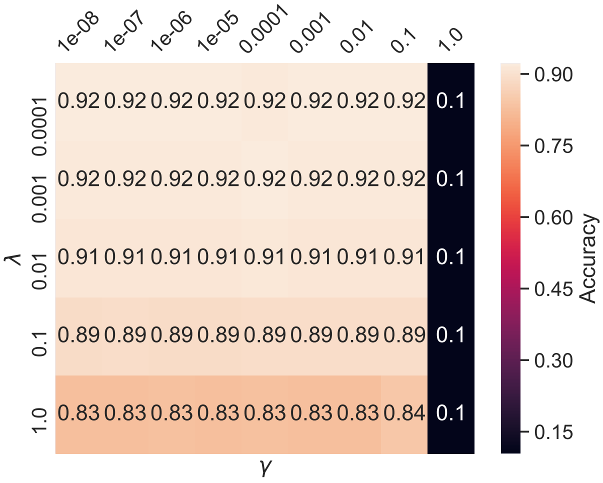

FIG. 7. The figure shows accuracy as a function of momentum $\gamma$ and regularirzation parameter $\lambda$ when our Logistic Regression model is trained on a dataset using learning rate $\eta = 0.1$, ephochs = 100 and batch size = 500 and then tested on a validation set.

of combinations of $(\gamma, \lambda)$. This leaves a number of of suitable choices for $\gamma$ and $\lambda$.

We chose $\lambda = 10^{-4}$ and $\gamma = 10^{-7}$ for further testing. As long as $\gamma$ lies in the interval $[10^{-8}, 10^{-1}]$, and $\lambda \in [10^{-3}, 10^{-4}]$, cf. figure 7, its specific value should not be of great significance to our results. As with the results from figure 6 the quite large region of parameters that enable the model to predict with high accuracy indicates a model that generalizes well to unseen data.

We performed a grid search in the parameter space $(\beta_1, \beta_2)$ with the Adam optimizer using the optimal parameters of 100 epochs, batch size of 500, $\lambda = 10^{-4}$ and $\eta = 0.1$. The resulting heatmap is presented in figure 8.

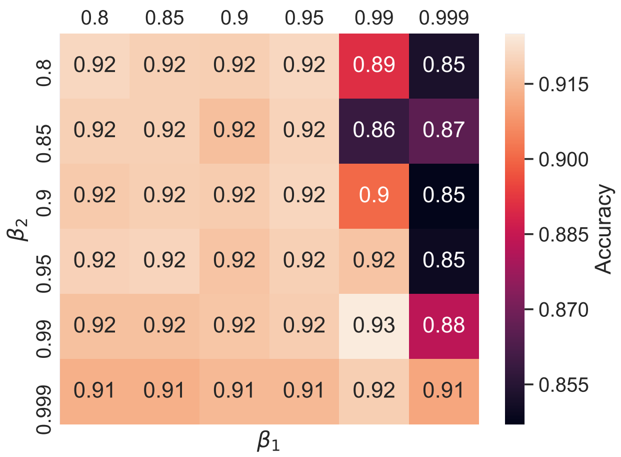

FIG. 8. The figure shows accuracy as a function of the decay rates $\beta_1$ and $\beta_2$ when our logistic regression model is trained with $\eta = 0.1$, 100 epochs, batch size of 500 and $\lambda = 10^{-4}$ and then tested on a validation set. The highest score of accuracy was achieved with $\beta_1 = \beta_2 = 0.99$.

We observe an area of accuracy of $= 90 - 93\%$ spanned by the parameter values $\beta_1 \in [0.8, 0.99]$ and $\beta_2 \in [0.90, 0.999]$. This follows well from the typical initialization values of $\beta_1, \beta_2 \sim 1.0$ seen in literature. We also observe that for our model it seemingly looks like a good choice to make sure that especially the decay rate of the second moment of the gradient, $\beta_2$, is initiliazied to a value as close to 1.0 as possible to ensure accurate predictions.

Finally we trained our model with the parameters found above for different optimization algorithms. The acquired model were then applied to an unseen dataset. Table VI shows the resulting accuracy scores.

| Method | Accuracy (%) |
|---|---|
| SGD | 92.61% |
| SGD with momentum | 92.69% |
| Adam | 92.80% |

TABLE VI. The table shows the maximally attained validation accuracy achieved with the multinomial logistic regressor on the MNIST dataset for the various optimizers.

As we chose to find some optimal hyperparameters using one method of optimization and then use these as we try to identify other optimal hyperparameters for a different method of optimization we do not get to see the difference of the methods in full action. For further work on this subject it would be of interest to see if methods like Adam really does converge towards a minimum faster than regular gradient descent by seeing if one could achieve the same level of accuracy by using fewer epochs than the 100 we estimated from SGD. This sadly requires a lot of time and computing power as a grid-search (at best) has an exponential growth of time usage when adding more parameters.

## C. Neural Network and Regression

### 1. *The effect of model complexity*

In figure 9, we show the validation $R^2$-score as a function of hidden neurons and polynomial degree. The parameters used were: mini-batch size of 100, $\eta = 0.1$, 1 hidden layer and 100 epochs. One should especially note that the $R^2$-score is reduced severely once 500 hidden neurons is used, which is indicative of overfitting. Similarly, with 1 hidden neuron, the $R^2$-score is generally a bit worse than what models with more hidden neurons achieve which is a sign of underfitting. The result itself does not indicate a serious preference for any specific model, but the polynomial degree can be chosen between $d = \{2, 3, ..., 9\}$ and hidden neurons should be between 10 and 30 to make sure the validation score does not dip below 0.9.

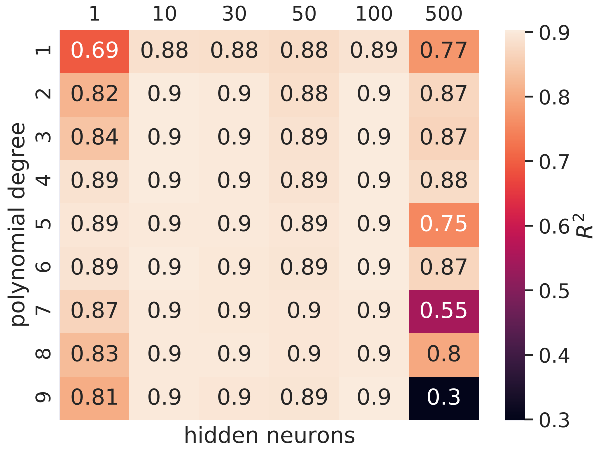

FIG. 9. The figure shows the $R^2$ score on validation data from the Franke's function dataset as a function of hidden neurons and polynomial degree. The parameters used were: mini-batch size of 100, $\eta = 0.1$, 1 hidden layer and 100 epochs. The hidden activation function was the Sigmoid function.

Another way to increase model complexity, is to vary the number of hidden layers. In figure 10, we show the validation $R^2$-score as a function of number of hidden layers and polynomial degree. The number of hidden neurons per layer were fixed to 30, otherwise the parameters are the same as in figure 9. From the figure, we can observe that the model performance decays as the complexity of the model becomes large. There's in fact fewer neurons in the model with the largest set of hidden layers, but because the number of parameters needed in the model scales more strongly with hidden layers than with number of neurons, it's not sufficient to simply consider how many hidden neurons the model has, which figure 10 clearly give evidence for.

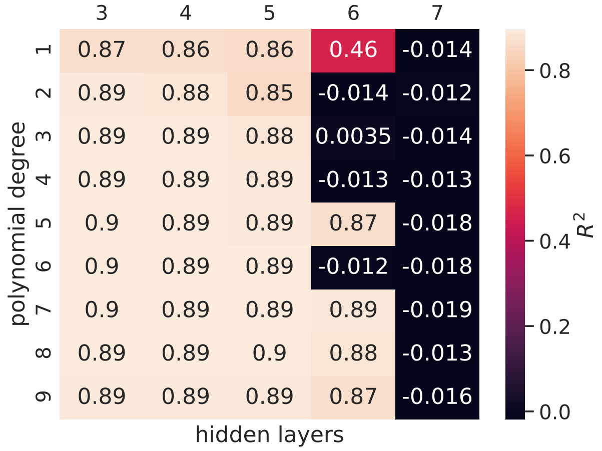

FIG. 10. The figure shows the $R^2$ score on validation data from the Franke's function dataset as a function of hidden layers and polynomial degree. The parameters used are shown in table (no ref table just yet). The parameters used were: 1 hidden layer, 30 hidden neurons, 100 epochs, mini-batch size of 100, $\eta = 0.1$ and $d = 5$. The hidden activation function was the Sigmoid function.

### 2. *Adding regularization and momentum*

With a polynomial degree $d = 5$ and 30 hidden neurons, we performed a grid search over the hyperparameter space $(\lambda, \gamma)$ which gave the results shown in figure 11. Clearly, neither regularization nor momentum has any significant effect on the validation score, indicating that there's some other bottleneck that keeps the model from improving. One possible bottleneck is the size of the training data set as neural nets benefit greatly from large data sets.

### 3. *Comparison of hidden activation functions*

When investigating the different hidden activation functions, we look at the $R^2$ score as a function of the learning rate $\eta$ and polynomial degree $d$ with the following values for the remaining hyperparameters: Mini-batch size of 100, 1 hidden layer, 100 epochs and 30 hidden neurons.

Figure 12 shows the $R^2$ score for the ReLU activation function, while figure 13 depicts the $R^2$ score for leaky ReLU.

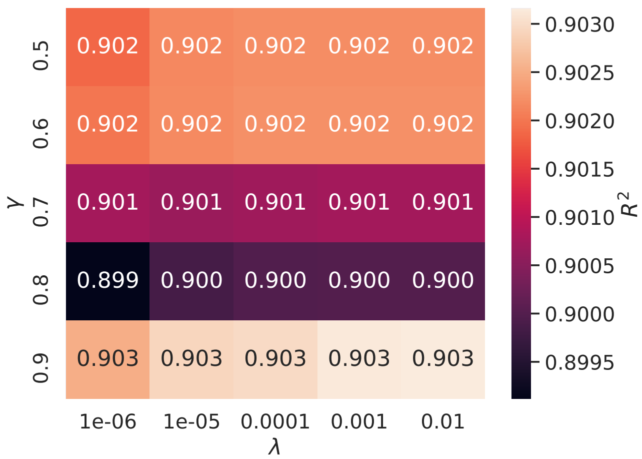

FIG. 11. The figure shows the $R^2$ score on validation data from the Franke's function dataset as a function of $\lambda$ and $\gamma$. The parameters used were: a mini-batch size of 100, $\eta = 0.1$, 1 hidden layer, 100 epochs, 30 hidden neurons and $d = 5$. The Sigmoid function was the hidden activation function.

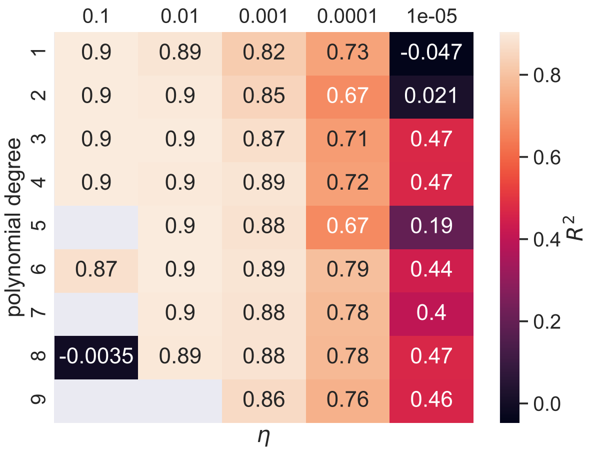

FIG. 12. The figure shows the $R^2$ score on validation data from the Franke's function dataset as a function of $\eta$ and polynomial degree. The parameters used were: a mini-batch size of 100, 1 hidden layer, 100 epochs and 30 hidden neurons. The ReLU function was the hidden activation function. The blank spaces corresponds with NaN-values.

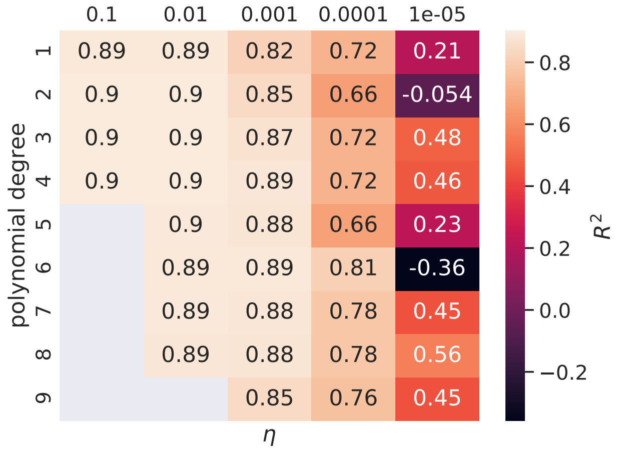

FIG. 13. The figure shows the $R^2$ score on validation data from the Franke's function dataset as a function of $\eta$ and polynomial degree. The parameters used were: a mini-batch size of 100, 1 hidden layer, 100 epochs, 30 hidden neurons. The hidden activation function was the leaky ReLU. The black spaces correspond with NaN-values.

The same behaviour is observed for the $R^2$ score in figures 12 and 13. By applying either one of the two activation functions we evade the problem of the vanishing gradient, but encounter a new obstacle related to high values of the gradient. Studying the figures we see that a high learning rate give rise to a number of NaN-values, which is a result from overflow in the numerical calculations. This is a direct result of high values of the gradient, and thus the optimal optimal values for the learning rate are one order of magnitude smaller than the preferred $\eta$ when applying Sigmoid as the hidden activation function.

The obtained $R^2$ scores are not noticeably higher than the one aquired with the Sigmoid function, leading to the following conclusion: in the case of regression, the different activation functions does not influence the generalizability of our model.

### 4. *Comparison with linear regression*

From the previous sections, a set of parameters that optimizes the validation score can be chosen to see how well the model fares in comparison to the linear regression models studied earlier. Using the parameters $d = 5$, 30 hidden neurons, 1 hidden layer, $\lambda = 10^{-4}$, 100 epochs, mini-batch size of 100, $\gamma = 0.5$, $\eta = 0.1$ with the Sigmoid function as the hidden activation function, we achieved MSE values on validation and test sets as shown in table VII. Comparing these values with the once shown in table V and table IV, we can note that the neural net performs slightly better than the linear models. In addition, the MSE scores are relatively close in value, suggesting the model generalizes well.

| Data set | MSE |
|---|---|
| Validation set | 0.103 |
| Test set | 0.108 |

TABLE VII. The table shows the MSE values achieved by the neural net model with parameters $d = 5$, 30 hidden neurons, 1 hidden layer, $\lambda = 10^{-4}$, 100 epochs, mini-batch size of 100, $\gamma = 0.5$, $\eta = 0.1$ with the Sigmoid function as the hidden activation function.

### D. Neural Network and Classification

In figure 14 the accuracy on the test set of MNIST is shown as a function of training data size and learning rate $\eta$. The main conclusions to draw from this result is that neural nets is a data hungry model that thrives on large amounts of training data. The figure also indicates that

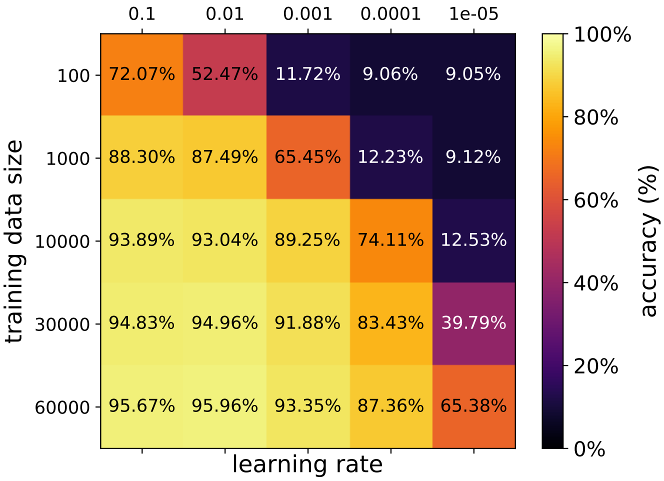

FIG. 14. The figure shows the accuracy achieved by the neural network on the MNIST dataset for as a function of traning data size and learning rates. The accuracy was computed on $n_{\text{test}} = 10000$ test examples. The parameters used were batch size = 10, 1 hidden layer, 30 neurons and 30 epochs. No regularization or momentum was added. The hidden activation function was the Sigmoid function and the top layer activation was the Softmax function.

#### 1. *Comparison of hidden activation functions for classification*

In figure 15, we show the accuracy achieved by the neural net on the validation set of the MNIST data as a function for hidden neurons and epochs. The parameters chosen were a mini-batch size of 10, 1 hidden layer and $\eta = 0.1$. We can note a general tendency for the model to increase accuracy levels as we increase the complexity of the model. It also appears that the model is not in need of a large set of epochs in order to learn from the data.

In figure 16, an identical form of grid search was performed using $\eta = 0.01$ with the ReLU activation function. The lower learning rate was chosen based on the results found with regression as shown in figure 12. Comparing this to figure 15, we mostly achieve a better validation accuracy using the ReLU function. Especially in the case with only a single hidden neuron, we're able to significantly increase the performance of the model.

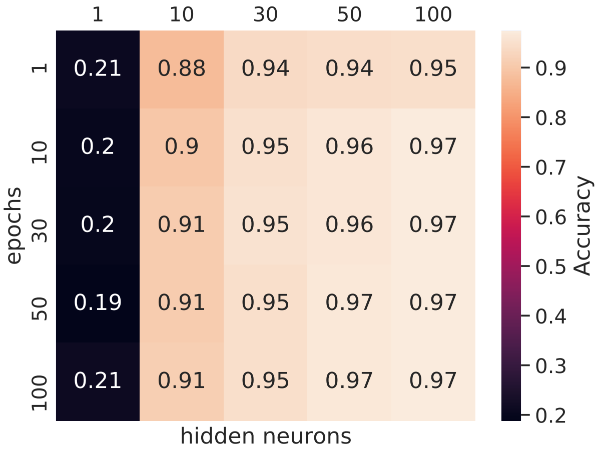

FIG. 15. The figure shows the accuracy achieved by the neural network on the MNIST dataset for as a function of hidden neurons and epochs. The accuracy was computed on $n_{\text{validation}} = 3000$ validation examples. The parameters used were mini-batch size 10, 1 hidden layer, $\eta = 0.1$, $n_{\text{train}} = 57000$. No regularization or momentum was added. The hidden activation function was the Sigmoid function and the top layer activation was the Softmax function.

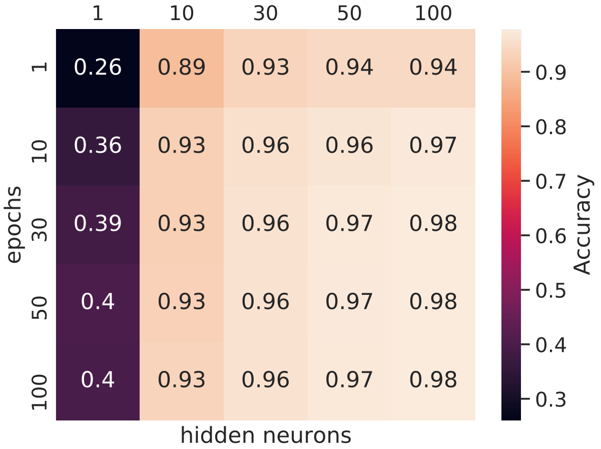

FIG. 16. The figure shows the accuracy achieved by the neural network on the MNIST dataset for as a function of hidden neurons and epochs. The accuracy was computed on $n_{\text{validation}} = 3000$ validation examples. The parameters used were mini-batch size 10, 1 hidden layer, $\eta = 0.01$, $n_{\text{train}} = 57000$. No regularization or momentum was added. The hidden activation function was ReLU and the top layer activation was the Softmax function.

#### 2. *Generalizability of the model*

To understand how well the model generalizes, it's necessary to compare the accuracy on the validation set to the accuracy achieved on the test set in which a small difference between the two indicates that the model generalizes well. In figure 11, we show the validation accuracy as a function of $\lambda$ and $\gamma$. Observe that ReLU does not seem to benefit from large momentum, since the accuracy dips a little as $\gamma \rightarrow 1$.

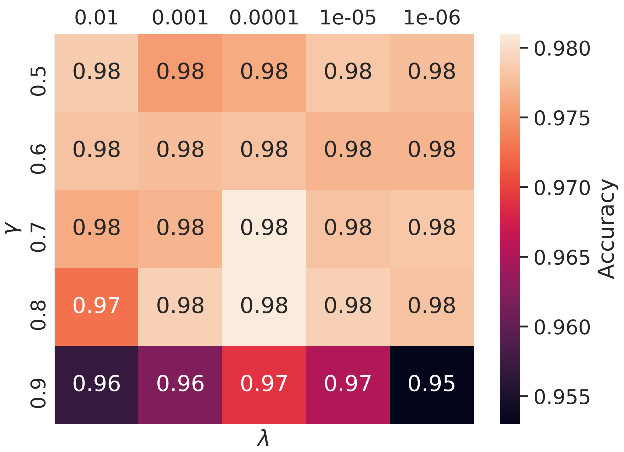

FIG. 17. The figure shows the accuracy achieved by the neural network on the MNIST dataset for as a function of $\lambda$ and $\gamma$. The accuracy was computed on $n_{\text{validation}} = 3000$ validation examples. The parameters used were mini-batch size 10, 1 hidden layer, $\eta = 0.01$, $n_{\text{train}} = 57000$, 100 hidden neurons and 30 epochs. The hidden activation function was ReLU and the top layer activation was the Softmax function.

The test accuracy on the MNIST test set consisting of 10000 test examples as a function of $\lambda$ and $\gamma$ is shown in figure 18. The main thing to note is that for any choice $(\lambda, \gamma)$, the accuracy is practically the same, suggesting that the model generalizes well for most such choices. In reality we'd pick a single tuple $(\lambda, \gamma)$ along with the rest of the parameters chosen as our model which we'd use in some production environment or similar to perform a given task. These results indicate that we could simply pick the parameters that maximizes the accuracy on the validation set to achieve a model that performs well on unseen data.

### *3. Neural nets as a superior classifier*

When comparing any of the accuracy results achieved on the MNIST dataset compared to the best ones achieved by the multinomial logistic regression model, it's clear that the neural net performs much better. This is likely due to the fact that the neural net has a larger viable set of parameters at its disposal (for instance a variable number of hidden neurons and layers) that allows it to be complex enough to capture the information encoded by the dataset it trains on and extract which features matters for a given response. The logistic regression model, on the other hand, has a fixed set of weights given by the feature set (at least in this case since we're studying images from the MNIST dataset) which gives the model a fixed number of parameters it must seek to optimize in order to predict unseen data. This means the model will plateau at some point and for this particular dataset it does so at a validation accuracy of about 92−93%, while the neural net comfortably achieves $\sim 98\%$ on both the validation and test sets.

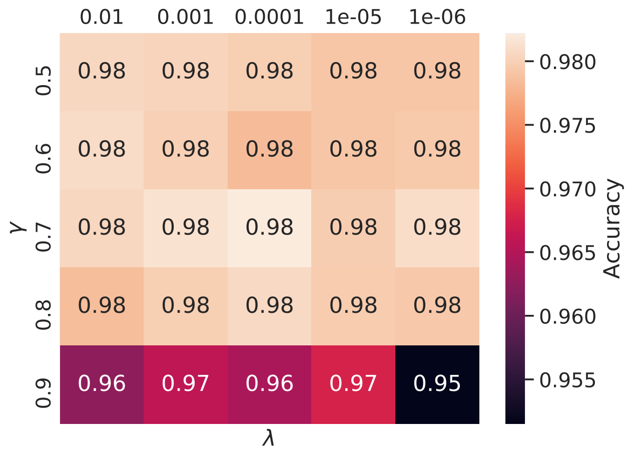

FIG. 18. The figure shows the test accuracy achieved by the neural network on the MNIST dataset for as a function of $\lambda$ and $\gamma$. The accuracy was computed on $n_{\text{test}} = 10000$ validation examples. The parameters used were mini-batch size 10, 1 hidden layer, $\eta = 0.01$, $n_{\text{train}} = 57000$, 100 hidden neurons and 30 epochs. The hidden activation function was ReLU and the top layer activation was the Softmax function.

## V. CONCLUDING REMARKS

We compared the performance of the linear regression models and found that matrix inversion in general performed better than using SGD with and without momentum to find the global minimum.

We found that the neural net performed slightly better than the linear models in the regression tasks, which we think is likely due to the fact that neural nets are non-linear models that are able to capture information from data that the linear models simply can't.

We compared three different gradient descent methods in the context of logistic regression, and found that SGD with momentum performed slightly better than SGD without momentum, while the Adam outperforms them both by a pinch. The difference in convergence rate for the three methods would be a highly relevant topic of investigation and is left as a task for further work on this subject.

We found that the neural net was superior in the classification task compared to the multinomial logicistic regression model. We conclude this is due to the fact that neural nets can vary its parameter set and create more complex models that can account for more subtle details in the data.

The conclusions drawn in both the regression and clasification tasks may be data dependent, so we suggest this as a topic for further investigation to understand whether the conclusions drawn in this article are generalizable. Another topic of interest might be to increase the dataset size used in regression to see whether the neural net can increase its performance since the neural nets' performance appear to benefit from data size.

## VI. APPENDIX

### A. Convexity

There are several definitions for a convex function. To prove that the cost functions $E_{\text{OLS}}$ and $E_{\text{Ridge}}$ are two convex functions, we rely on the following definition: A function $f$ is convex if its Hessian is positive semi-definite [2]. A symmetric matrix $A$ is said to be positive semi-definite if the relation

$$u^T A u \geq 0, \tag{39}$$

is fulfilled for all vectors $u \neq 0$. The Hessians' of the cost functions in question are respectively

$$\nabla^2 E_{\text{OLS}} = X^T X, \qquad \nabla^2 E_{\text{Ridge}} = X^T X + \lambda I. \tag{40}$$

Noting that the matrix $X^T X$ is symmetric, we insert these expressions into the inequality in eq. (39) yields

$$u^T\left(\nabla^2 E_{\text{OLS}}\right)u = u^T X^T X u = \|Xu\|^2 \geq 0, \tag{41}$$

$$u^T\left(\nabla^2 E_{\text{Ridge}}\right)u = u^T(X^T X + \lambda I)u = \|Xu\|^2 + \lambda\|u\|^2 \geq \lambda\|u\|^2 > 0. \tag{42}$$

Given that the penalty parameter $\lambda > 0$, both of the expressions fulfils eq. (39), hence the cost functions are indeed convex. Eq. (42) is strictly larger than zero, hence $E_{\text{Ridge}}$ is strictly convex.

### B. Gradient of Cross-entropy

Suppose the output of our model $\hat{y} \in \{0,1\}^M$ is made up of $M$ classes. Then the cross-entropy for a single datapoint $(x, y)$ with $x \in \mathbb{R}^p$ and $y \in \{0,1\}^M$, is given by

$$E = -\sum_{m=1}^{M} y_m \log \hat{y}_m + (1 - y_m)\log(1 - \hat{y}_m). \tag{43}$$

Let $\hat{y}_m = \sigma(z_m) \equiv \sigma_m$, where $\sigma(z)$ is assumed to obey $\partial\sigma_m/\partial z_j = \sigma_m(1-\sigma_m)\delta_{mj}$. Let $z_m = \sum_i W_{mi}x_i + b_m$, for some weight matrix $W$ and bias vector $b$. Then the gradients for a single data point are

$$\begin{aligned}\frac{\partial E}{\partial W_{jk}} &= \frac{\partial E}{\partial \hat{y}_m}\frac{\partial \hat{y}_m}{\partial z_j}\frac{\partial z_j}{\partial W_{jk}} = \frac{\partial E}{\partial \sigma_m}\frac{\partial \sigma_m}{\partial z_j}\frac{\partial z_j}{\partial W_{jk}} \\ &= -\sum_{m=1}^{M}\left[\frac{y_m}{\sigma_m} - \frac{1-y_m}{1-\sigma_m}\right]\sigma_m(1-\sigma_m)\delta_{jm}\delta_{ki}x_i \\ &= (\sigma_j - y_j)x_k.\end{aligned} \tag{44}$$

Similarly, for the bias term, we get

$$\frac{\partial E}{\partial b_j} = \sigma_j - y_j. \tag{45}$$

[1] René Ask, Kaspara Gåsvær, and Maria Horgen. Regression and resampling techniques applied to franke's function and terrain data. https://github.com/reneaas/fys-stk4155/blob/master/project1/report/Linear_Regression.pdf, 2020.

[2] Geir Dahl and Michael Floater. Optimization. https://www.uio.no/studier/emner/matnat/math/MAT3110/h20/pensumliste/lecture12.pdf, 2020.

[3] Richard Franke. A critical comparison of some methods for interpolation of scattered data. 1979.

[4] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems*. O'Reilly, 2019.

[5] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2323, 1998.

[6] Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre Day, Clint Richardson, Charles Fisher, and David Schwab. A high-bias, low-variance introduction to machine learning for physicists. *Physics Reports*, 810:16, 03 2018.