# FYS-STK4155: Applied Data Analysis and Machine Learning

## Project 2

Mohamad Mahmoud & Aleksandar Davidov
GitHub - click here -

November 23, 2021

**Abstract**

A gradient descent approach in order to solve various regression related problems. We've implemented a stochastic gradient descent algorithm (SGD) and used it in order to solve the linear regression problem faced in [2]. We expanded on our SGD and implemented it together with a Feed Forward Neural Network. We compared the results for the linear regression using the same kind of data generated in [2], namely the Franke function. We also used SGD to solve a logistic regression problem where we have used the Wisconsin Breast Cancer Data. We then expanded on our FFNN in order to make it so it can solve classification problems. We contrasted the results of both plain SGD and the neural network throughout our work, in order to better understand how they behave. Our finding show that the neural network outperforms the gradient descent in both regression and classification. FFNN: 0.018 in test MSE and SGD: 0.02 in test MSE for regression problems. As for classification problems, FFNN: 0.991 in test accuracy against 0.965 to that of SGD.

# Contents

## I. INTRODUCTION

Linear regression although a powerful tool in and of itself, it tends to preform poorly when the model is introduce to new data that doesn't quite behave the same way as the data the model has been fitted on. This is especially true when the complexity of the model is increased (the polynomial degree $p$), as seen in our findings in [2]. The MSE for the training data decreases significantly with increased $p$ but the model itself preforms poorly. That is why in iteration 2 we dive deeper into more complex, reliable and applicable methods namely gradient descent and the Feed Forward Neural Network.

This is an extension of our work in [2] where we will now utilize the regular and stochastic gradient decent, with and without momentum. In order to solve regression problems using SGD all while we have reference data we can go off of, guiding our analysis for linear regression. We then use the stochastic gradient decent algorithm with momentum to construct a Feed Forward Neural Network and contrast the results to that of linear regression with SGD. All our work for linear regression was done using the dataset generated by the Franke function.

For our next step we implemented a Logistic Regression algorithm using our gradient decent. Where we have used the Wisconsin Breast Cancer data provided via the Scikit-Learn Python module. From that we followed the same step as for linear regression and expanded upon our FFNN in order for it to be able to solve classification problems.

## II. THEORY

Given a dataset $\mathcal{D}(\mathbf{X}, \mathbf{y})$ consisting of a set of inputs/predictors $\mathbf{X}$ that correlate to an output/response $\mathbf{y}$. Our goal is to develop a model $f(\mathbf{X}, \theta)$, where $\theta$ is a set of parameters fitted in accordance to the data. $\theta$ is determined by defining a cost function $\mathcal{C}(\theta)$, also denoted $E(\theta)$, that allows us to judge how well the model describes the data at hand. The model is fitted by finding the values of $\theta$ that minimizes the cost function.

Regression analysis takes advantage of multiple mechanism that minimizes the cost function. One of which is the fact that the first derivative of $\mathcal{C}(\theta)$ wrt. $\theta$ is zero at a minimum whether local or global.

### i. Gradient Descent

A tuning of $\theta$ could be done by utilizing gradient descent (GD). Where the basic idea is that any function for that matter $\mathbf{F}(\mathbf{x})$ decreases the fastest if one goes from $\mathbf{x}$ in the direction of the negative derivative $-\nabla\mathbf{F}(\mathbf{x})$, [1].

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta\nabla\mathbf{F}(\mathbf{x}_k)$$

Here $\eta > 0$ is the learning rate which gauges how much we move in the direction of the negative gradient. Given a sufficiently small $\eta$ the condition $\mathbf{F}(\mathbf{x}_{k+1}) \leq \mathbf{F}(\mathbf{x}_k)$ is fulfilled, which means with each iteration $k$ we move towards a smaller value, i.e a minimum.

So for our case we want to find $\theta$ that minimizes the cost function $\mathcal{C}(\theta)$ so the iteration scheme for the GD is given by

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

where $\nabla_\theta \mathcal{C}(\theta_t)$ is the gradient of the cost function wrt. $\theta$ at an iteration step $t$ and $\eta_t$ is the learning rate. For the gradient descent we start with an initial guess $\theta_0$ and iteratively move towards a minimum. We also need to choose $\eta_t$ wisely, since very small values of $\eta_t$ results in an extensive number of iterations before a minimum is reached and with $\eta_t$ relatively big we risk overshooting the minimum.

Gradient Descent has multiple drawbacks:

- it's sensitive to the initial condition
- it's deterministic, as in if it converges to a minimum it stays there no matter if global or local.
- it's very sensitive to the choice of the learning rate
- it treats all directions in parameter space uniformly, meaning that learning rate is constant

We will now dive into different variations that tackle many of the drawbacks in the standard GD.

### Stochastic Gradient Descent

The Stochastic gradient descent (SGD) takes advantage of the fact that the cost function, which we aim to minimize can, be written as a sum over the number of data points $n$, meaning:

$$\mathcal{C}(\beta) = \sum_i^n c_i(\mathbf{x}_i, \beta)$$

which means that the gradient of the cost function can be computed as a sum of gradients over $n$ data points, [1].

$$\nabla_\beta \mathcal{C}(\beta) = \sum_i^n \nabla_\beta c_i(\mathbf{x}_i, \beta)$$

To avoid evaluating the gradient of the cost function for the entire data set, the SGD iterates over a number of subsets of the data sets, referred to as *mini-batches*. Each batch contain randomly drawn points from the original data set, where the size of the batch, denoted $B$, is significantly smaller than $n$ for large data sets. So given that the size of each batch is $B$ then the total number of batches is $n/B$, where each mini-batch is indexed as $B_k$ where $k = 1, \ldots, n/B$. The goal now is instead of doing the GD iteration scheme for all data points, we do it over one of the $n/B$ mini-batches $B_k$ available, where $k$ is picked at random with equal probability from $[1, n/M] \subset \mathbb{Z}^+$. Meaning:

$$\nabla_\beta \mathcal{C}(\beta) = \sum_i^n \nabla_\beta c_i(\mathbf{x}_i, \beta)$$
$$\rightarrow \ \nabla_\beta \mathcal{C}_{B_k}(\beta) = \sum_{i \in B_k}^n \nabla_\beta c_i(\mathbf{x}_i, \beta) \tag{2}$$

An iteration over the number of mini-batches $n/B$ is an *epoch* and typically this is done repeatedly over a number of epochs. SGD outperforms the standard GD by introducing stochasticity which decreases the likelihood of the optimization scheme getting stuck in a local minimum. Secondly the size of each mini-batch is small relative to the number of data points, $B < n$. Thus the computation of the gradient became significantly less taxing.

A way to avoid overlapping datapoints from being picked over and over, from epoch to epoch and hinder that some datapoints are never picked, is to introduce an indices `array`. The `array` is of length N, whhich is the same as the number of datapoints in $\mathcal{D}$ and ranges from 0 to N-1. All that's left is to create a random indices `array` that contains the shuffled indices and loop through the number of mini-batches $B_k$ as shown below:

**Iteration scheme over batches for each epoch**

```
indices= [0, ..., N-1]
for epoch in epochs
    random_indices = shuffle(indices)
    for b_k in B_k
      batch = random_indices[b_k · B: b_k · B + B]
```

The iteration scheme for the SGD is expressed as

$$v_t = \eta_t \nabla_\theta E_{B_k}(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_t \tag{3}$$

From this we extract the fact that $v_t$ is a running average of recent gradients.

### Momentum based Gradient Descent

Adding a momentum serves as a memory of the direction we are moving in parameter space. The idea is to let the previous iteration depict, to a degree, the change made in the next iteration. This is done by

$$v_t = \gamma v_{t-1} + \eta_t \nabla_\theta E_{B_k}(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_t \tag{4}$$

where $\gamma \in [0, 1]$ is referred to as the momentum parameter. The advantages behind this modification is that this way the GD scheme converges faster and moves persistently towards a minimum in the case of small gradients even in the presence of stochasticity. Furthermore, it suppresses oscillations in high-curvature directions, [1].

### The ADAM Optimizer

In addition of keeping track of the running average of the first momentum, in ADAM (short for ADAptive Moment estimation) the same is done for the second moment of the gradient. By appropriating this into our iteration scheme for the GD, we can update our parameter $\theta$ as follows

$$g_t = \nabla_\theta E_{B_k}(\theta_t)$$
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$s_t = \beta_2 s_{t-1} + (1 - \beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{s}_t = \frac{s_t}{1 - \beta_2^t}$$
$$\theta_{t+1} = \theta_t - \eta_t \frac{\hat{m}_t}{\sqrt{\hat{s}_t} + \epsilon}$$

Where $m_t = \mathbb{E}[g_t]$ and $s_t = \mathbb{E}[g_t^2]$ are the running average of both the first and second moment of the gradient. $\epsilon \sim 10^{-8}$ is a regularization constant to prevent divergence and singularities. Both $\beta_1$ and $\beta_2$ set the memory lifetime of the first and second moment and are typically taken to be 0.9 and 0.99, respectively. In addition the learning rate $\eta_t$ is set to $10^{-3}$, usually. This modification of the scheme insures that the learning rate is reduced in parts of the scheme where the norm of the gradient is persistently large over a time period, i.e. when approaching a minimum. Additionally the scheme allows for larger learning rates for flatter/saddle-like parts, which in turn results in a speedier convergence rate, ch. 7 in [1].

## ii. From matrix inversion to GD: OLS and Ridge

Linear regression problems could be solved by either finding an analytical expression for the parameter $\beta$ through matrix inversion or by finding the derivative of the cost function (the MSE) wrt. $\beta$ and finding the optimal values of $\beta$ through a minimization algorithm such as the gradient descent variations mentioned in the previous section. Here we will contrast our own linear regression analysis in [2] to that of the gradient descent approaches priorly discussed.

The vector parameter $\beta$ for both ordinary least squares (OLS) and Ridge is given by:

$$\beta_{OLS} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{y}$$
$$\beta_{Ridge} = \left(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}\right)^{-1}\mathbf{X}^T\mathbf{y}$$

As for the first derivative of the loss function for both OLS and Ridge

$$\frac{\partial \mathcal{C}(\beta)}{\partial \beta} = \frac{2}{n}\mathbf{X}^T\left(\mathbf{X}\beta - \mathbf{y}\right) \quad \text{(OLS)}$$
$$\frac{\partial \mathcal{C}(\beta)}{\partial \beta} = 2\mathbf{X}^T\left(\mathbf{X}\beta - \mathbf{y}\right) + 2\lambda\beta \quad \text{(Ridge)}$$

Here we make use of our findings in [2] and contrast the result to that produced by the minimization algorithms. This implies preforming an analysis of the various parameters and presets choices and study the results. We look at the optimal choice of polynomial degrees, hyperparameter $\lambda$ for ridge, the learning rate $\eta$ and for the case of SGD the number of *epochs* and *mini-batches*.

Both of the derivatives for the OLS and Ridge, respectively, are convex functions; here we refer to ch. 7.2 in [1] for a thorough mathematical explanation of convex function. In other words a minimum of the first derivatives is guaranteed to be a global minimum.

### iii. Logistic Regression

Logistic regression differs from linear regression in that we are trying to solve a classification problem. Thus, in a logistic regression problem, we aim to find the optimal parameters $\beta$ that maps each input $\mathbf{x}$ into a class $y$. In other words given a class with discrete outputs $y \in \{0, 1\}$ and input $\mathbf{x} \in \mathbb{R}^p$, where $p$ is the number of features/characteristics that describe the model.

The same concept can be expanded to the multitude of $K$ classes, where $y$ a is $K$-long *one-hot* vector. So for the $j$-th class

$$y_j = \begin{cases} 0 & \text{if doesn't belong} \\ 1 & \text{if belongs} \end{cases} \rightarrow y_j = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1_j \\ 0 \\ \vdots \end{bmatrix}$$

Solving a multi-class classification problem is often referred to as a *multinomial* classification problem, making this a multinomial logistic regression problem. Furthermore, there are two types of classification, *Hard* and *Soft* classification.

- Hard classification in which the prediction is hard-set to either a value of 0 or 1.
- Soft classification in which the outputs from the logistic function of the prediction are consequent probabilities summing to 1. And the classification is made by picking the parameter with the highest probability.

In logistic regression we map the parameters $\beta$ and the inputs $\mathbf{x}$ into a probability function by using the Sigmoid function.

$$p(t) = \frac{1}{1 + \exp(-t)} = \frac{\exp t}{1 + \exp t} \tag{5}$$

Making it a soft classifier. For a class $y_i \in K$, where $y_i$ could be either 0 or 1, our polynomial fit of $y_i$ is

$$\widetilde{y}_i = \mathbf{x}\beta = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p \tag{6}$$

The probabilities for the binary outcome in class $y_i$ are given by:

$$p(y_i = 1|\mathbf{x}, \beta) = \frac{\exp\left(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p\right)}{1 + \exp\left(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_p x_p\right)}$$
$$p(y_i = 0|\mathbf{x}, \beta) = 1 - p(y_i = 1|\mathbf{x}, \beta)$$

The total likelihood of all possible outcomes in a dataset $\{\mathcal{D} \mid x_i, y_i \in \mathcal{D}\}$, where all outcomes $y_i$ are drawn independently, is defined by the maximum likelihood estimation (MLE) principle, [1]. Thus, our aim is to determine $\beta$ such that we maximize the probability of getting the observed data $y_i$ from our model $\widetilde{y}_i$. In other words for a specific outcome $y_i$ the likelihood is expressed in terms of the chain of probabilities corresponding to $y_i$, meaning

$$P\left(\mathcal{D}|\beta\right) = \prod_{i=1}^{n} \left[p(y_i = 1|\mathbf{x}, \beta)\right]^{y_i} \left[1 - p(y_i = 1|\mathbf{x}, \beta)\right]^{1-y_i}$$

From this we obtain the log-likelihood and the first iteration of our cost-function

$$\mathcal{C}(\beta) = \sum_{i=1}^{n} \left[y_i \cdot \mathbf{x}\beta - \log(1 + \exp(\mathbf{x}\beta)\right] \tag{7}$$

where $\beta\mathbf{x}$ is as defined in eq. 6. We adapt the cost-function above to a regression model by inverting signs, making it so we now minimize as we iterate and so it's applicable to any of the minimization algorithms discussed earlier. This is key since in logistic regression the minimization of the cost function results in a non-linear equation in the parameter $\beta$, [1]. Leaving us with the variations of gradient descent as our only option to optimize the learner.

$$\mathcal{C}(\beta) = -\sum_{i=1}^{n} \left[y_i \cdot \mathbf{x}\beta - \log(1 + \exp(\mathbf{x}\beta)\right] \tag{8}$$

Eq. 8 is often refereed to as cross entropy. The cross entropy is convex function with respect to the weights $\beta$, meaning that the minimum found by the optimizer is guaranteed to be a global minimum, [1]. Much like what we did for linear regression we need to find the derivative of the loss function wrt. $\beta$, this yields

$$\frac{\partial \mathcal{C}(\beta)}{\partial \beta} = -\mathbf{X}^T(\mathbf{y} - \mathbf{p}) \tag{9}$$

where $\mathbf{X}$ is the set of inputs corresponding to the one-hot vector $\mathbf{y}$ and $\mathbf{p}$ is the Sigmoid logistic function as a function of the model's prediction $\widetilde{\mathbf{y}}$.

It's worth noting that the cross entropy is often used with an L1 or L2 regularization term, adding the L2-norm to our cost function:

$$\mathcal{C}(\beta) = -\sum_{i=1}^{n} \left[ y_i \cdot \mathbf{x}\beta - \log(1 + \exp(\mathbf{x}\beta) \right] + \lambda \|\beta\|_2^2, \ \lambda > 0$$

where $\lambda\|\beta\|_2^2 = \lambda \sum_{i=1}^{n} \beta_i^2$. As for the gradient

$$\frac{\partial \lambda \|\beta\|_2^2}{\partial \beta} = 2\lambda\beta = \lambda'\beta$$

this gives us

$$\frac{\partial \mathcal{C}(\beta)}{\partial \beta} = -\mathbf{X}^T(\mathbf{y} - \mathbf{p}) + \lambda'\beta$$

Logistic regression much like linear regression takes advantage of the fact that the parameters that yield the lowest value cost function are the optimal parameters for the model at hand.

### iv. Artificial Neural Network

Artificial Neural Networks (ANN) is a collection of Machine Learning (ML) algorithms which are inspired and modeled after biological neural networks in the brain. Typically, but not necessarily, an ANN consists of three types of layers, each consisting of their own set of operations, in which the training data $\mathbf{X}$ passes through in order to optimize the learner. These layers consists of *nodes* that are connected together by a set of weights. Usually the layers makeup consists of *input-layers*, *hidden-layers* and *output-layers*. Consequently their set of nodes are referred to as input-nodes, hidden-nodes and output-nodes respectively. The main idea behind ANNs is to calibrate the weights which connect the numerous classes of layers. The way this is done is through a process called *backpropagation*. The calibrated weights are then used to optimize our model $\hat{\mathbf{y}}$.

In our project, we have implemented the simplest type of ANN called the Feed Forward Neural Network (FFNN). In an FFNN, the connections between the nodes, do not form a cycle, unlike recurrent neural networks for example. In other words its input-nodes are only connected to the hidden-nodes and its hidden-nodes only to the output nodes. That means there are no direct connections between the input and output nodes.

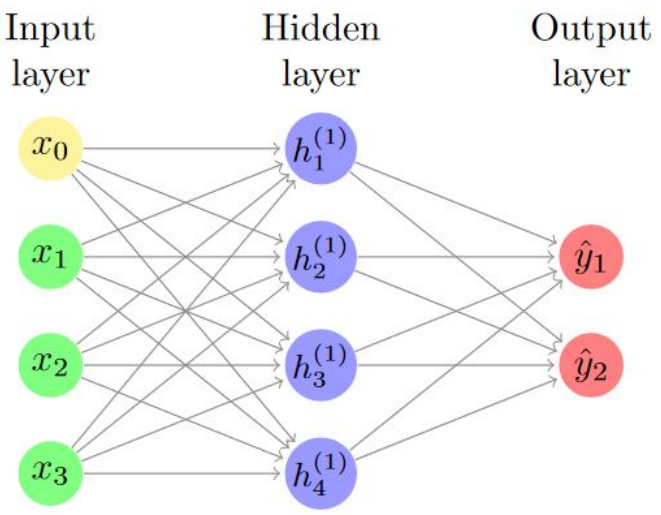


Figure 1. Example of a simple Feed Forward Neural Network with three input, four hidden and two output nodes. The $x_0$ node is the bias term for the input layer. We will have a bias term for each layer in our network.

The number of nodes in the input-layer is determined by the number of features; complexity/dimensionality of the input data. The number of nodes in the hidden-layers can be freely chosen and tweaked to achieve optimal results and lastly the number of nodes in the output-layer is determined by the complexity/dimensionality of the targets in our model.

#### Forward-propagation

Forward propagation is a reference to the scheme (set of operations) the inputs passes through before arriving to a generated weighted output, that is a fit for our model. The input data is fed to each node in each subsequent layer following the input-layer. For each of these individual nodes in our network, the following computation is performed:

$$z_j^l = \sum_{i=1}^{n_{l-1}} = w_{ij}^l a_i^{l-1} + b_j^l \tag{10}$$

$$a_j^l = f(z_j^l) \tag{11}$$

where $z_j^l$ corresponds to the input value in layer $l$ for node $j$. $w_{ij}^l$ corresponds to the matrix-element $w_{ij}$ for layer $l$ in the weights matrix $W^l$, where $w_{ij}^l$ connects node $i$ in layer $l-1$ to node $j$ in layer $l$. We also have $b_j^l$ which is the bias-term for node $j$ in layer $l$. Lastly $a_j^{l-1}$ is the value produced by the *activation function* defined for $z_j^{l-1}$.

#### Backpropagation

Now we arrive at the backpropagation algorithm which is essential for training an FFNN. In essence, backpropagation is used to update each of the weights and biases of

a neural network, in order so that the output $\hat{\mathbf{y}}$ best fits the target $\mathbf{y}$. Hence minimizing the error of the output.

For a regression problem, we might consider the Mean Square Error (MSE) as our cost function

$$\mathcal{C} = \|\hat{\mathbf{y}} - \mathbf{y}\| = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

As for classification consider the Cross Entropy cost function

$$\mathcal{C} = -\sum_{i=1}^{N}\left[\hat{y}_i \log y_i + \log\left(1 - y_i\right)\left(1 - \hat{y}\right)\right].$$

The theory behind the backpropagation algorithm is to compute the gradient of the cost function wrt. the weights and biases and use the various gradient descent schemes discussed in **II THEORY i**. This is done via the chain rule for derivation, one layer at a time, iterating backwards from the last layer $l = L$ to the first $l = l_0$. Hence the name backpropagation.

Recall that the gradient of a function is the vector of all its partial derivatives.

$$\nabla\mathcal{C} = \left\{\frac{\partial\mathcal{C}}{\partial w_{ij}^l}, \frac{\partial\mathcal{C}}{\partial b_i^l}\right\}.$$

For the layer $L$, using the chain rule, we obtain

$$\frac{\partial\mathcal{C}}{\partial w_{ij}^L} = \frac{\partial\mathcal{C}}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^l}{\partial w_{ij}^L}$$

and we can rewrite this as

$$\frac{\partial\mathcal{C}}{\partial w_{ij}^L} = \frac{\partial\mathcal{C}}{\partial a_j^L}\frac{df}{dz_j^L}a_i^{L-1}. \tag{12}$$

As for the derivative wrt. the bias $b_j^L$, we obtain

$$\frac{\partial\mathcal{C}}{\partial b_j^L} = \frac{\partial\mathcal{C}}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial b_j^L}$$

which again can be rewritten as

$$\frac{\partial\mathcal{C}}{\partial b_j^L} = \frac{\partial\mathcal{C}}{\partial a_j^L}\frac{df}{dz_j^L}.$$

Considering what we've arrived to until now we can define the error at layer $l$ for node $j$ as

$$\delta_j^l = \frac{\partial\mathcal{C}}{\partial z_j^l} = \frac{\partial\mathcal{C}}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l} = \frac{\partial\mathcal{C}}{\partial a_j^l}\frac{df}{da_j^l}. \tag{13}$$

Eq. 12 can then easily be rewritten using $\delta_j^L$

$$\frac{\partial\mathcal{C}}{\partial w_{ij}^L} = \delta_j^L a_j^{L-1}$$

The error $\delta_j^l$ gauges the contribution from node $j$ in changing the cost function at layer $l$. Building up on what we obtained we can generally define the error $\delta_j^l$ for $j$-th node at a layer $l$ in terms of the "prior" layer $l+1$ in the backpropagation scheme.

$$\delta_j^l = \sum_{\mathrm{k}}\frac{\partial\mathcal{C}}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} \tag{14}$$

Where

$$\frac{\partial\mathcal{C}}{\partial z_k^{l+1}} = \delta_k^{l+1}$$

Solving for the second derivative term wrt. $z_j$

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{jk}^{l+1}\frac{df}{dz_j^l}$$

and putting this back into eq. 14, we get

$$\delta_j^l = \sum_{\mathrm{k}}\delta_k^{l+1}w_{jk}^{l+1}\frac{df}{dz_j^l} \tag{15}$$

So for the complet backprogation scheme we start with computing $\delta_j^L$ and then iterate backwards starting with layer $l = L - 1$. For each iteration over a layer $l$ we compute $\delta_j^l$ using eq. 15 and update the weights and biases in accordance to the gradient descent scheme in use. The gradient descent in its simplest form would follow

$$w_{ij}^l = w_{ij}^l - \eta\delta_j^l a_i^{l-1}$$
$$b_j^l = b_j^l - \eta\delta_j^l$$

here $\eta$ is the learning rate.

### v. Activation Functions

In the two previous subsections, we mentioned activation functions. The activation function defines the output of a neuron given the inputs. There are many different kinds of activation functions with their unique strength and weaknesses. In our project, we have implemented the *Sigmoid*, *ReLU*, *Leaky-ReLU* and *Softmax* activation functions. The sigmoid activation function is defined as in eq. 5.

The ReLU activation function is defined as

$$f(z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{elsewhere} \end{cases}$$

while the Leaky ReLU is defined as

$$f(z) = \begin{cases} z & \text{if } z \geq 0 \\ \alpha \cdot z & \text{elsewhere} \end{cases}$$

where we have used $\alpha = 0.01$.
Lastly, the Softmax activation function is defined as

$$f(z) = \frac{e^z}{\sum_i e_i^z}$$

## III. TRAINING DATA

### i. Franke Function

We have used the Franke Funcion for our regression analysis. For description, refer to 1 [2] for overview on how we draw data from the function.

### ii. Cancer Data

Four our classification, we have used the Breast Cancer Wisconsin (diagnostic) dataset available through *Scikit-learn*. This dataset contains a total of 569 data points with 30 features. The output value is either 0 or 1 depending on if the cancer is benign (does not spread to the rest of the body) or malignant (spreads to rest of the body).

### Scaling

The breast cancer dataset we are using has values in the range between 0 to 4000 with varying numerical scales/scalibility. Therefore we will apply a standard scaling. This is done easily on the design matrix, $\mathbf{X}$ by subtracting the mean and divide by the standard deviation.

### Deep Neural Networks

To get even better predictions, we will extend our Feed Forward Neural Network to include multiple hidden-layers. For classification problems, we want to analyze discrete data. Therefore, adding more hidden layers will allow for more complex and accurate decision making. We call Neural Networks with more than one hidden layer for Deep Neural Networks (DNN).

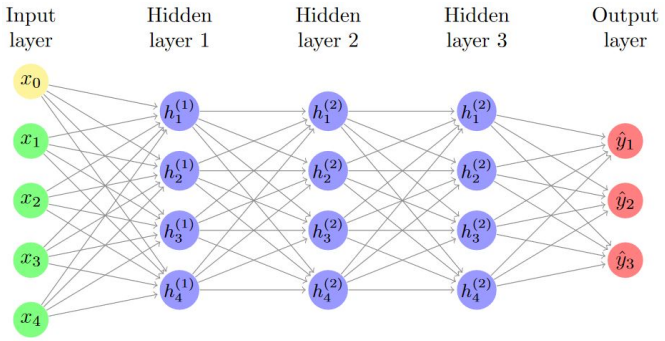


Figure 2. Example of a Deep Feed Forward Neural Network with three hidden layers.

## IV. RESULTS & DISCUSSIONS

### i. From matrix inversion to GD: OLS and Ridge

We studied the efficacy of SGD by comparing the results to our previous findings in [2] of the Franke function. Our findings pointed out that for OLS a design matrix of the polynomial degree $p = 5$ yilded the best MSE results for both training and testing and bias-variance trade-off, fig. 4 & 5 in [2]. As for Ridge the best result was obtained by $p = 5$ and the regularization parameter set to $\lambda = 0.001$. We did the analysis with that in mind and set the complexity of the design matrix to $p = 5$ or `deg=5` in our code. We tested for the learning rate $\eta$, the `batch size`, in addition to the regularization parameter also referred as the *hyperparameter* $\lambda$ when it comes to Ridge. The data generated had $N = 400$ datapoints with a 0.8 split for training and 0.2 for testing. The stagnation of the training MSE from epoch to epoch was the determining factor for our convergence, then the test MSE was calculated as a result of the fitted $\beta$ coefficients.

#### Momentum SGD

We first determined the best value for the momentum parameter $\gamma$ in stochastic gradient descent. We did that by setting up a preset of what we think is a good initial testing preset; where we set $\eta = 0.01$ and `batch size = 40`. Our metrics for determining $\gamma$ are both the training MSE and the number of epochs before convergence as performance gauge.

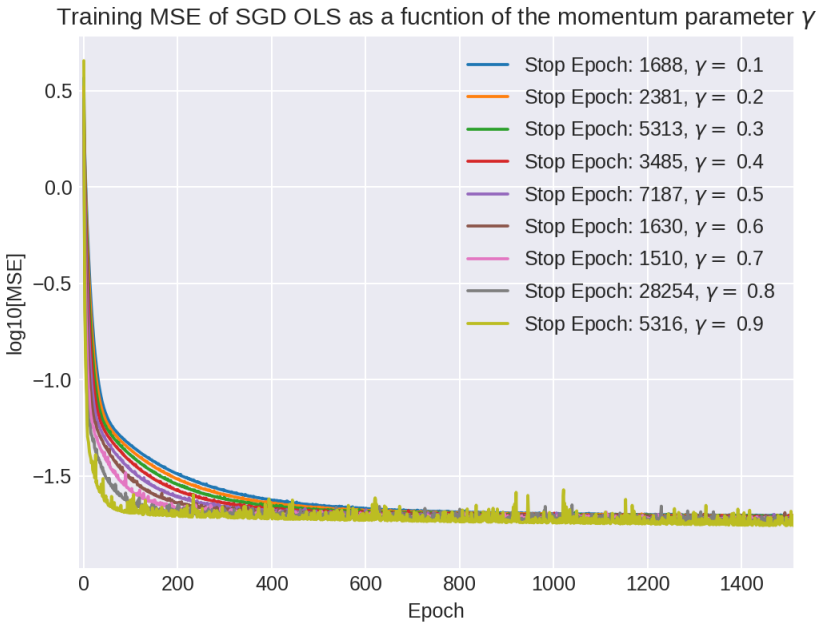


Figure 3. The training MSE as a function of the momentum parameter $\gamma$ with a chosen preset for both the learning rate and the batch size; $\eta = 0.01$ and `batch size = 40`.

The trend we observed in fig. 3 is that the training MSE decreases as $\gamma$ approaches 1, on the other hand $\gamma = 1$ is unstable and leads to overflow. This decrease in MSE comes at the expense of the number of epochs required to converge and stability of the MSE, which in itself explains why it takes more to converge. Lower values of $\gamma$ behaved in a stable and predictable manner without requiring over 10000 epochs to converge. Given that we seek predictability and performance in our analysis we chose to set the momentum parameter to $\gamma = 0.2$, in the upcoming parts composed of SGD.

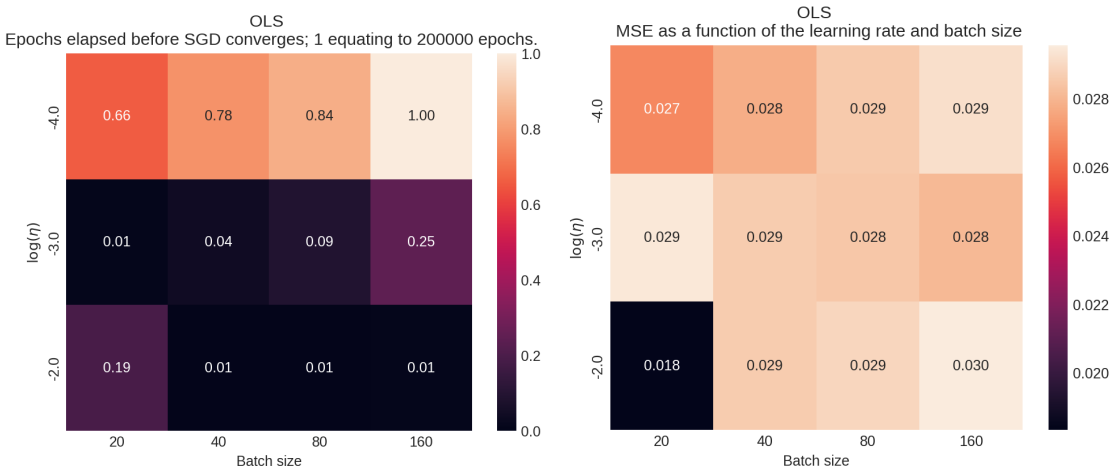


Figure 4. Both graphs are heatmap matrices as a function of both the learning rate $\eta$ and the `batch size`. Number of epochs elapsed before the SGD converged (left); 1.00 equating to 200000 epochs in total. The test MSE of SGD after convergence (right) or no convergence in the case of 1.00.

Fig. 4 shows $\eta$ = 1e−4 yields by far the worst results in terms of both performance and MSE. This value for the learning rate utilized the most number of epochs to converge, at `batch size = 160` it went through all 200000 epochs without converging and without any improvement for the MSE results.

The results of the MSE for OLS-SGD were used to determine the best batch size suited for our Ridge-analysis. This was done by taking the average MSE of each column (in other words the average MSE for each batch size) and picking the one with smallest average MSE. This turned out to be `batch size = 20` in our case. This method of determining the batch size was chosen since the analysis of the learning rate is the same for Ridge and we aimed to make it so every value produced for $\eta$ can contribute in determining the batch size.

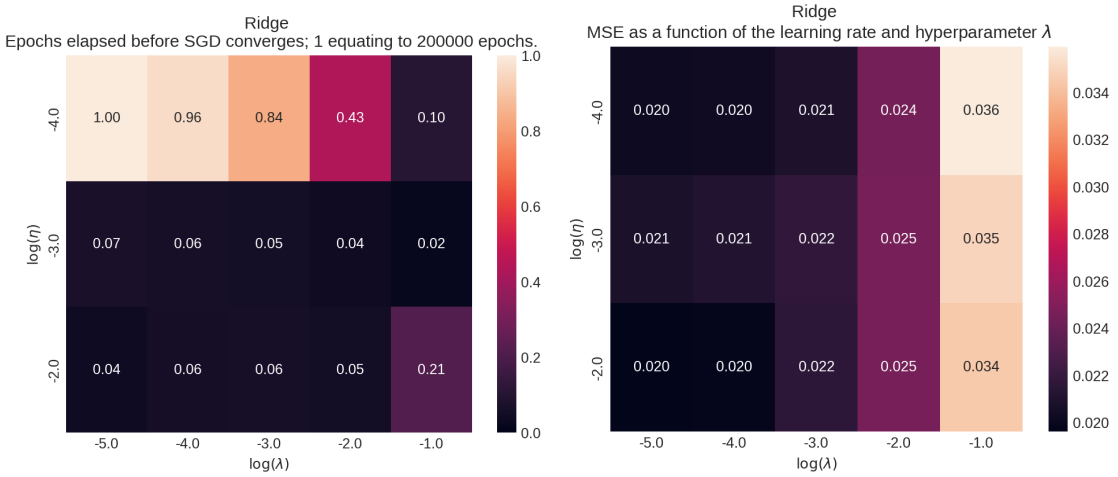


Figure 5. Both graphs are heatmap matrices as a function of both the learning rate $\eta$ and the hyperparameter $\lambda$. Number of epochs elapsed before the SGD converged (left); 1.00 equating to 200000 epochs in total. The test MSE of the SGD after convergence (right) or no convergence in the case of 1.00.

Agreeing with the OLS-analysis fig. 5 shows again that $\eta$ = 1e−4 performed the poorest going as far as not converging for $\lambda$ = 1e−5. From this we can conclude that setting the learning rate to $\eta > 1e{-}4$ results in a faster rate of convergence.

Wrapping up our analysis for linear regression of the Franke dataset. OLS shows that a batch size of 20 in this case yields the best test results without impacting performance too much, only 0.19 of epochs were required for $\eta$ = 1e−2. In terms of MSE and learning rates, $\eta$ = 1e−2 gave out the best results on average in terms of batch sizes and specifically for `batch size = 20`. As for Ridge the best performance as a function of the learning rate agrees with our findings for OLS. Furthermore, $\lambda$ = 1e−1 is the worst offender across the board in terms of MSE, all the while $\lambda$ = 1e−5, 1e−4 resulted in the best performance and test MSE.

### Artificial Neural Network

### ii. FFNN: Franke

We now build upon our linear regression Franke-analysis, by tackling the problem using an FFNN, with the MSE as our cost function and Sigmoid eq. 5 as the activation function. We have tested for different setups of number of hidden layers (NHL) and number of hidden nodes (NHN) and done the same type of analysis as for linear regression.

#### Sigmoid as our activation function

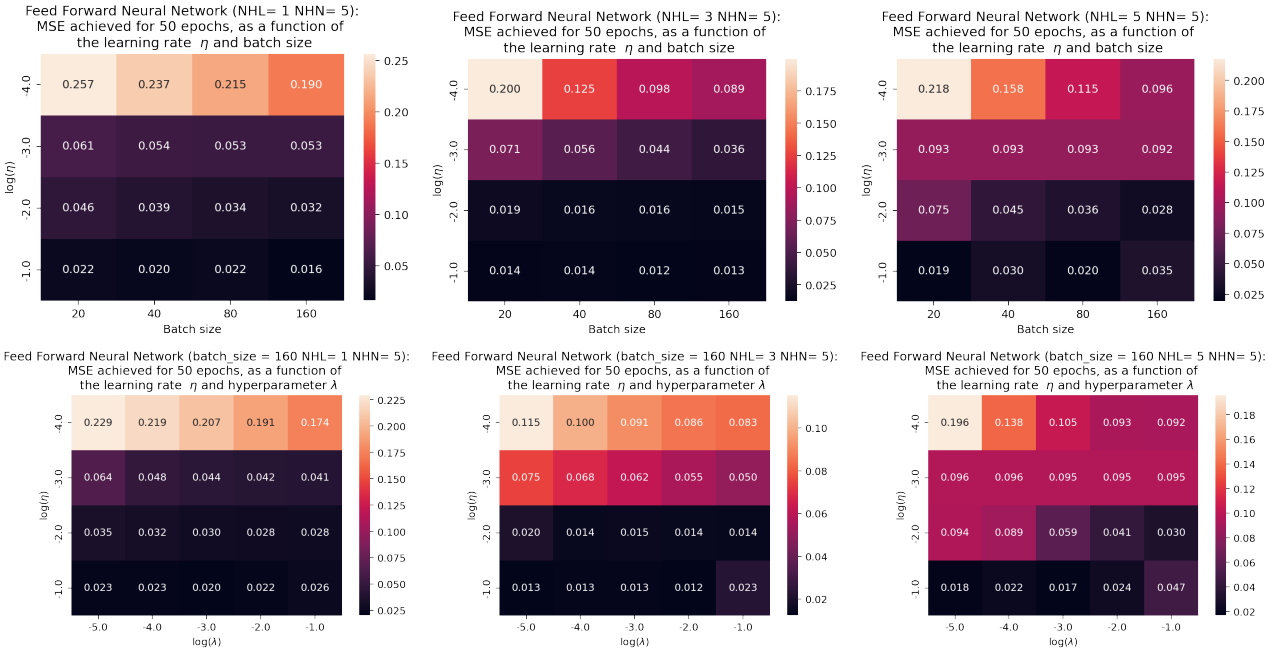


Figure 6. MSE heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs and `NHN= 5`. Batch size analysis (upper row), $\lambda$ analysis (lower row).

The upper row in fig. 6 represents the test MSE from the batch size analysis. Here the hyperparameter was set to $\lambda = 0$ so it won't contribute in deciding the optimal batch size for the following hyperparameter analysis (lower row) in fig. 6. Following the same footsteps as for our regression analysis the optimal batch size was determined by taking the average MSE over each of the columns and picking the batch size that corresponds to the lowest MSE. The optimal batch size for all 3 compositions of number of hidden layers, `NHL= 1, 3, 5` is `batch size= 160`. The heatmaps make it very clear that setting the learning rate to $\eta = 1\text{e}{-4}, 1\text{e}{-3}$ yields the worst results in terms of the test MSE and that $\eta = 1\text{e}{-1}$ is the optimal choice in this case.

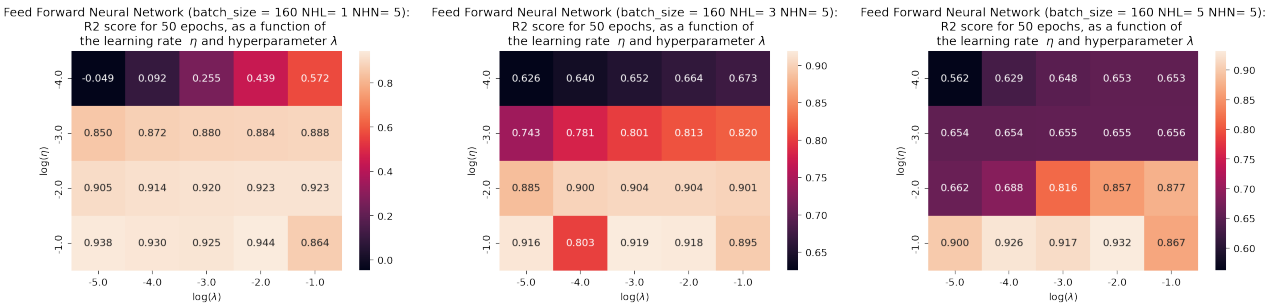


Figure 7. R2 score heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs and `NHN= 5`.

As for the structure of the NN and how it effects the performance in terms of test MSE and R2 score, setting `NHN= 5` gave by far the best result. These are the results we chose to highlight in fig. 6 and 7. The values for the MSE obtained from different setups can be found in **APPENDIX VI**. Which shows how poorly the neural network has preformed

after that the number of hidden nodes has exceeded the complexity of the design matrix $\mathbf{X}$, $p = 5$. The figures show that the results obtained for the MSE are signicantly worse to that of `NHN= 5`.

When it comes to the hyperparameter setting it to $\lambda = 1e{-}1$ improved our results, especially for $\eta = 1e{-}2$. The bullet points we can draw from this analysis is that having the learning in the interval $\eta = [1e{-}4, 1e{-}3]$ results in a higher test MSE and subsequently a lower R2 score. In addition increasing the number of nodes beyond the polynomial degree of the design matrix would generally make matter worse. As for the number of hidden layers and how it affected the test results, choosing `NHL= 3` gave the best test values. Since having multiple layers adjusting the weights for a better fit of our model beats 1, but certainly overdoing it can have a negative impact as is the case for `NHL= 5`.

Thus for our ReLU and Leaky ReLU analysis we are dropping the two smallest values for the learning rate and doing computation for `NHN= 3, 4, 5` and `NHL`$\leq$ `4`.

It's also worth mentioning that we have compared our implementation of the FFNN to that of Scikit's `MLPRegressor` and got

Table I. FFNN vs. `MLPRegressor`.

| | `NHL = 1 NHN = 5` | `NHL = 3 NHN = 5` |
|---|---|---|
| FFNN | 0.10 | 0.09 |
| Scikit | 0.31 | 0.62 |

### ReLU and Leaky ReLU

Again we redid the hyperparameter analysis for both the learning rate $\eta$ and $\lambda$, the optimal batch size is as always determined by the lowest average MSE over batches. This time the leaning values are in the range $\log(\eta) = [-2, -0.5]$ as we chose to disregard smaller values for the reasons we have discussed above, the values for $\lambda$ are the same. The neural network setups we have tested for are `NHL= 2, 3, 4` and `NHN= 3, 4, 5`. For the last activation function we chose Sigmoid, which gave adequate results.

Starting with learning rate analysis the NN tends to perform poorly at the margins, in other words as $\log(\eta) \to -2$ or $-0.5$, as can be shown in fig. 8

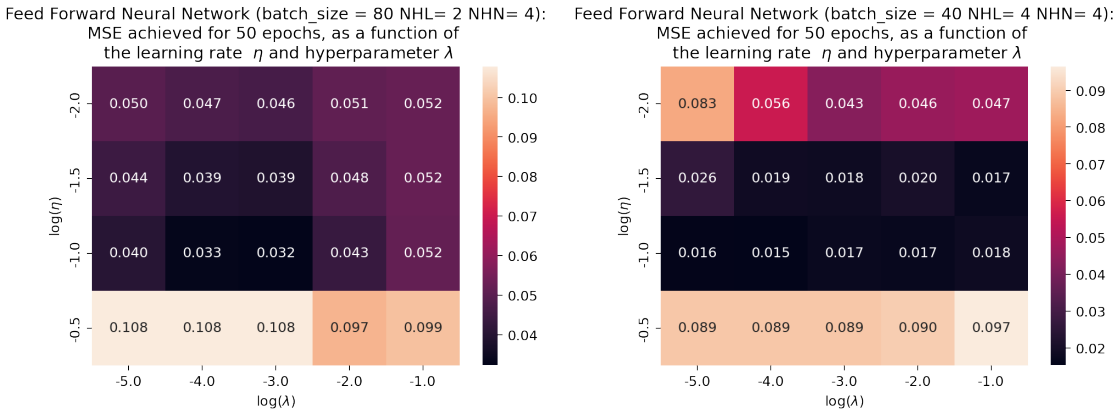


Figure 8. ReLU: MSE heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs, `NHL= 2, 3` and `NHN= 4`.

Replicating what we've got from Sigmoid `NHL= 3` had overall the best results, with `NHN= 5` and $\log(\eta) = -1$ yielding the best values.

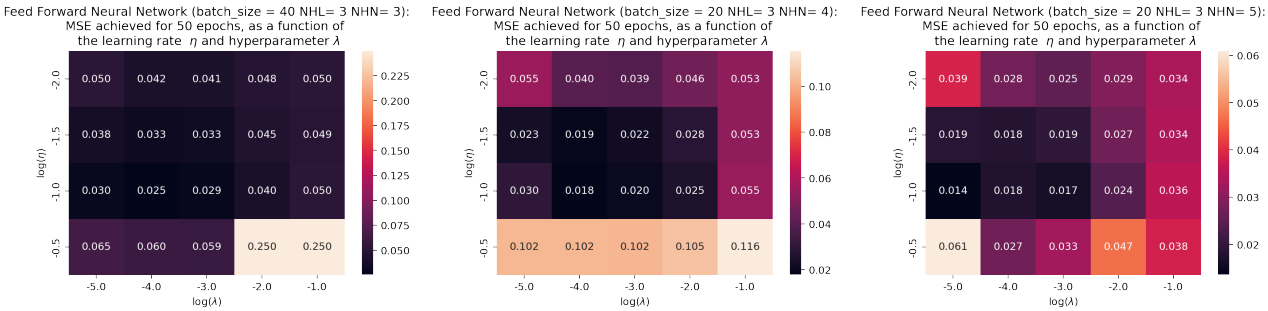


Figure 9. ReLU: MSE heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs and `NHL= 3`.

Increasing the number of the hidden nodes to better match the complexity of the design matrix, in this case it's 5, bettered the test MSE values. Which was we would have expected given our previous results. As for `NHL= 4`, this did not result in any significant or reliable improvements especially for the case `NHN= 5`, where it performed the worst specifically for $\log(\eta) = -0.5$.

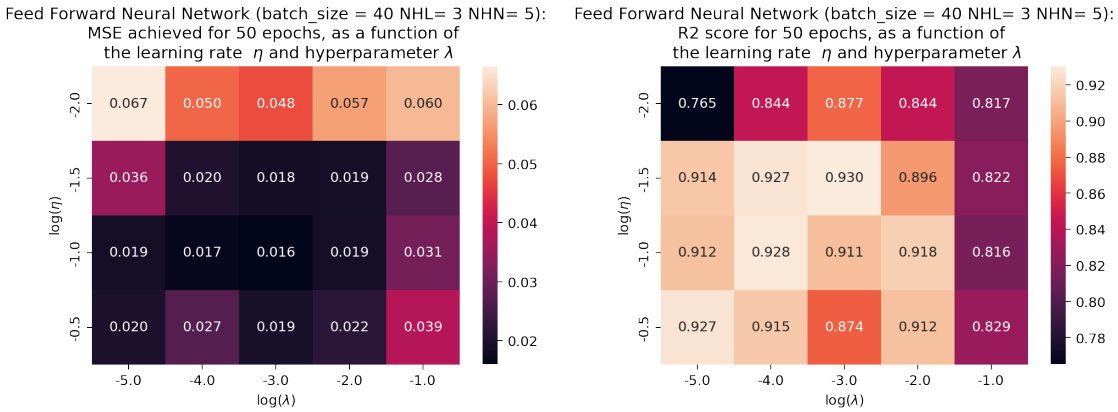


Figure 10. Leaky ReLU: MSE heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs, `NHL= 3` and `NHN= 5`. MSE (left) R2 (right).

What we can draw from this analysis that setting the number of hidden nodes to high or low relative to the polynomial degree of the design matrix, would almost definitively yield worse results. And one should chose the number of hidden layers carefully and tweak it according to which one yields the better test value errors.

### iii. Logistic Regression: Cancer Data

We used the cancer data available through `Scikit-Learn`. The data consist off $N = 569$ datapoints with a 0.8 split for training and 0.2 for testing. Here the response also referred to in this case as targets had binary outputs, either 0 or 1. We did our analysis for the learning rate $\eta$, the `batch size`, in addition to the hyperparameter $\lambda$ when it comes to logistic regression with L2-regularization. The convergence of the accuracy towards 1 for the training data was the determining factor for stopping further iterations. Then the accuracy of the test data was calculated using the fitted weights $\beta$ in our model.

## Momentum SGD

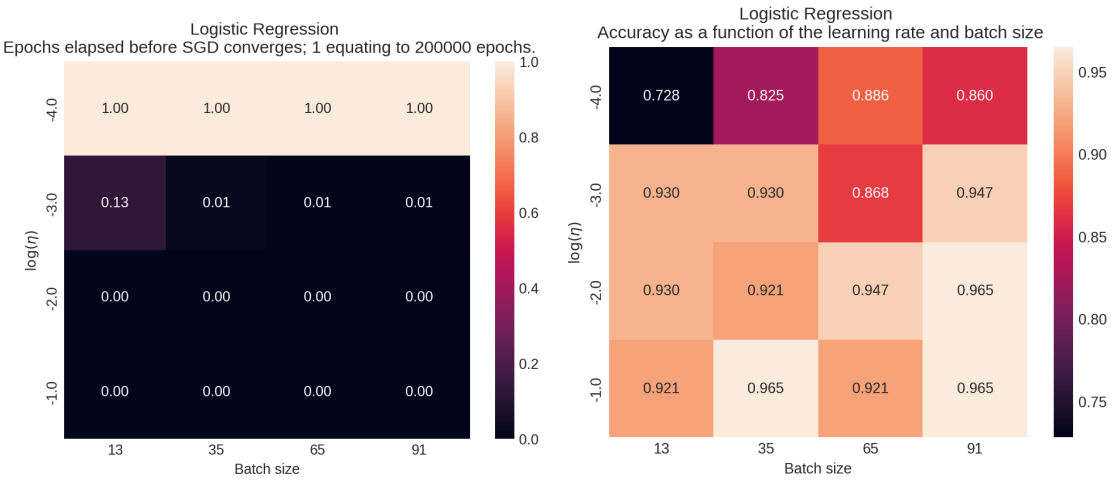


Figure 11. Both graphs are heatmap matrices as a function of both the learning rate $\eta$ and the `batch size`. Number of epochs elapsed before the SGD converged (left); 1.00 equating to 200000 epochs in total. The test *Accuracy* of the SGD after convergence (right) or no convergence in the case of epochs: 1.00.

Following the footsteps of the linear regression case fig. 11 shows $\eta = 1e{-}4$ yielded by far the worst results in terms of both performance and MSE. $\eta = 1e{-}4$ resulted in the most number of epochs to converge, that being 200000 epochs for any batch size, along with the worst accuracy.

Following the same approach as for linear regression, we determined the best batch size suited for L2-regularization by looking at the *Accuracy* results from fig. 11. Our approach is the exact same as for linear regression only this time the average is concerning the accuracy (goes without saying) and the batch size coinciding with the highest average accuracy was picked. This turned out to be `batch size= 91` in our case.

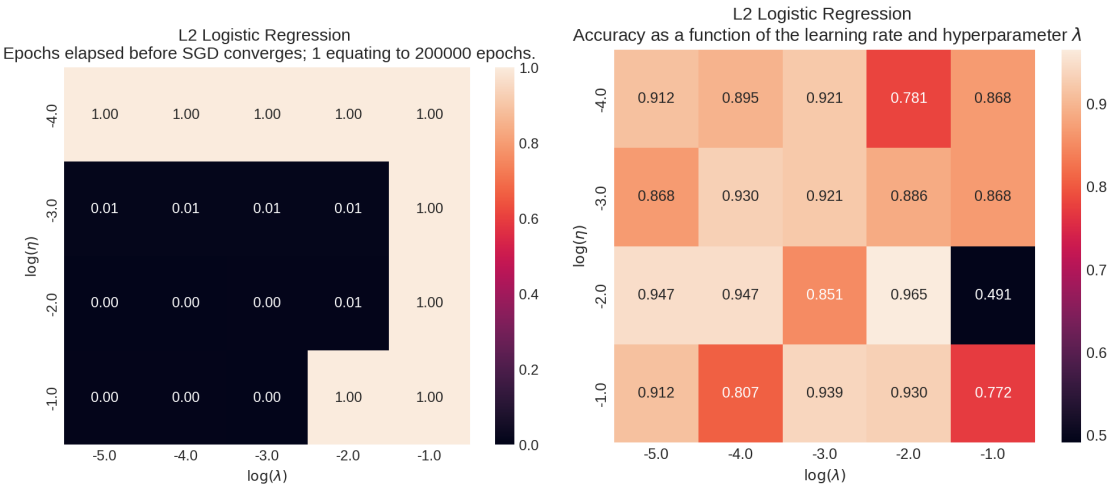


Figure 12. Both graphs are heatmap matrices as a function of both the learning rate $\eta$ and the L2-hyperparameter $\lambda$. Number of epochs elapsed before the SGD converged (left); 1 equating to 200000 epochs in total. The test *Accuracy* of the SGD after convergence (right) or no convergence in the case of epochs: 1.00.

Agreeing with the our initial logistic regression analysis fig. 12 shows that $\eta = 1e{-}4$ was the poorest performer in terms of both convergence rate and accuracy. In addition we observe that setting the hyperparameter to $\lambda = 0.1$ also yields the poorest performance and accuracy. We can draw the conclusion that setting the learning rate to $\eta > 1e{-}4$ and the hyperparameter to $\lambda > 0.1$ results in a faster convergence rate and better accuracy all around.

Drawing our final conclusion our initial logistic regression analysis shows that a batch size of 91 in this case produces the best test results, while being on par with any batch size in terms of performance. As for the *Accuracy* and learning rates, $\eta > 1e{-}2$ results in the best values in terms of accuracy, especially for `batch size = 91`. When it comes logistic

regression with L2-regularization the best performance as a function of the learning rate agrees with our finding for the initial logistic regression analysis. Furthermore, as we have already mentioned setting the hyperparameter to $\lambda = 0.1$ led to the worst results at hand, both in terms of performance and *Accuracy*. All the while the highest accuracy was achieved by setting $\lambda = 1\text{e}{-2}$, but with the caveat that it performed poorly for $\eta = 1\text{e}{-1}$. Leaving us with $\lambda = 1\text{e}{-3}$ as the best performer with more than adequate enough accuracy results.

## Artificial Neural Network

### iv. FFNN: Cancer Data

Continuing with our analysis and testing for the Wisconsin Breast Cancer data, this time for FFNN with the cross entropy eq. 8 as the cost function and Sigmoid eq. 5 as the activation function. We have tested for different setups of number of hidden layers (NHL) and number of hidden nodes (NHN) and done the same type of analysis as for logistic regression.

### Sigmoid as our activation function

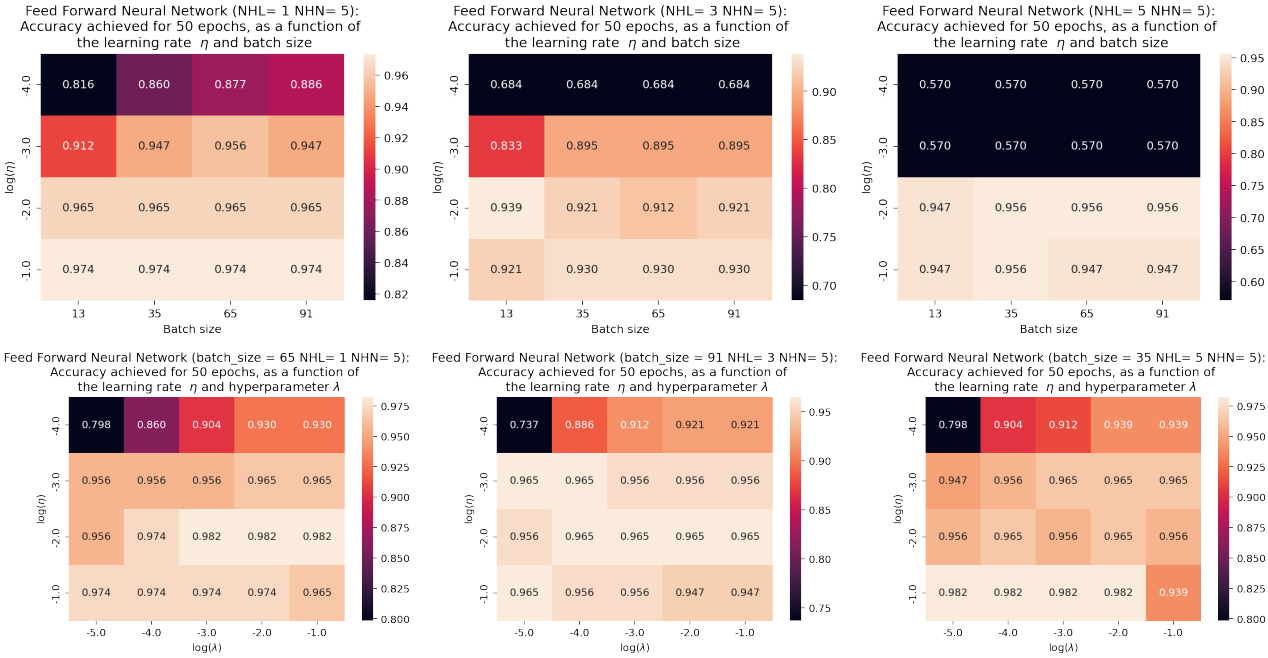


Figure 13. Test accuracy heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs and `NHN= 5`. Batch size analysis (upper row), $\lambda$ analysis (lower row).

The upper row in fig. 13 represents the values obtained of test *Accuracy* from our batch size analysis. Here the hyperparameter was set to $\lambda = 0$ so it won't contribute to the results shows. Following the same footsteps as before the optimal batch size was determined by taking the average accuracy over each column and picking the batch size that corresponds to the best accuracy. The batch sizes we've landed on are `batch size = 65, 35, 35` for `NHL = 1, 3, 5`, respectively.

The batch size analysis makes it very clear that setting $\eta = 1\text{e}{-4}, 1\text{e}{-3}$ resulted in the worst accuracies obtained no matter the batch size. As for the hyperparameter each time the batch size resulting in the most accurate results has been used, in the same manner as before. Small values of $\lambda$ preformed worse in general for this particular setup of NHL and NHN.

For `NHL= 1, 3, 5` setting the number of hidden nodes to 20 gave the best results overall. The overall analysis over the hidden layers for nodes ranging from 5 to 100 per layer is shown in **APPENDIX VI**.

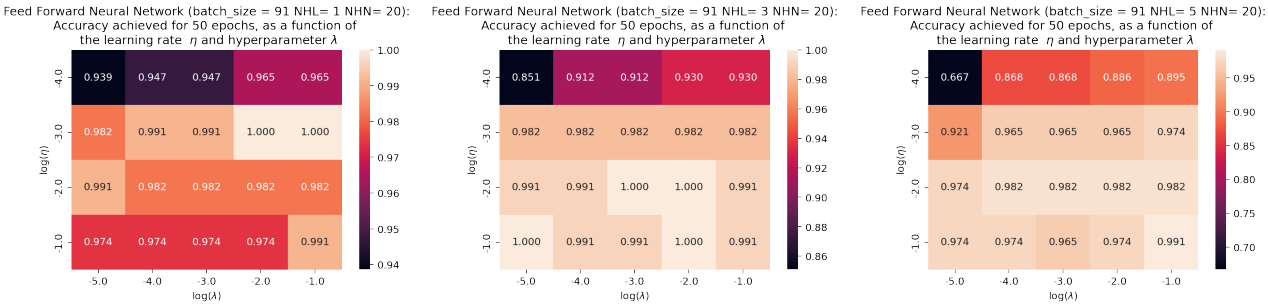


Figure 14. Test accuracy heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs and `NHN = 20`.

The reason for why a higher number of nodes were less accurate could be, and this is hypothesis from our side, related to the number of datapoints available from the dataset. $\mathcal{D}$ for the cancer data was made of 569 datapoint where 80% was used for training. Multiple hidden layers makes it so the tweaking process of the weights and biases goes through multiple stages before arriving to a final answer. As for the number of nodes per layer this determines how much each node is densely populated in terms of datapoints. Setting the number of hidden nodes to a relatively big number (closer to the numder of datapoints) makes it so that in each layer the data is more scattered and thus each tweaking operation/iteration has less data to work with. As for a small number of nodes each node would be very densely populated thus the corresponding weights would overfit for the data at hand, leading to worse results in the testing phase.

Wrapping up our analysis for the Sigmoid function: setting the learning rate to $\eta = 1\text{e}{-4}, 1\text{e}{-3}$ generally yields the worst results. Having the number of hidden nodes set to a relatively high number 40 and 100 for our testing cases, doesn't result in any gained accuracy, matter of fact it only increases the complexity of the neural network resulting in worse performance in terms of runtime.

Thus for our ReLU and Leaky ReLU analysis we are dropping the two smallest values for the learning rate and doing computation up to 30.

It's also worth mentioning that we have compared our implementation of the FFNN to that of Scikit's `MLPClassifier` and got

Table II. FFNN vs. `MLPClassifier`.

| | `NHL = 3 NHN = 20` | `NHL = 3 NHN = 30` |
|---|---|---|
| FFNN | 99.12% | 98.25% |
| Scikit | 99.12% | 85.96% |

### ReLU and Leaky ReLU

We've done a hyperparameter analysis for both the learning rate $\eta$ and $\lambda$, the batch size was determined in the same manner as we have always done. The learning rate values are in the range $\log(\eta) \in [-2, -1)$ as we disregarded smaller values due to risen instabilities and all in all yielding worse results, the values for $\lambda$ stayed the same. The composition of hidden layers we have tested for are the same as before; this being `NHL= 1, 3, 5`. As for the number of hidden nodes we focused on testing for `NHN= 10, 20, 30` but we also looked at `40` and `50` for the sake of hopefully getting the full picture of how ReLU and Leaky ReLU behave. For the `last activation` function we chose Sigmoid, which gave adequate results.

Generally speaking when it comes for the leaning rate, the NN tends to perform badly at the margins, in other words as $\log(\eta) \to -2$ or $-1$, as observed in fig. 15:

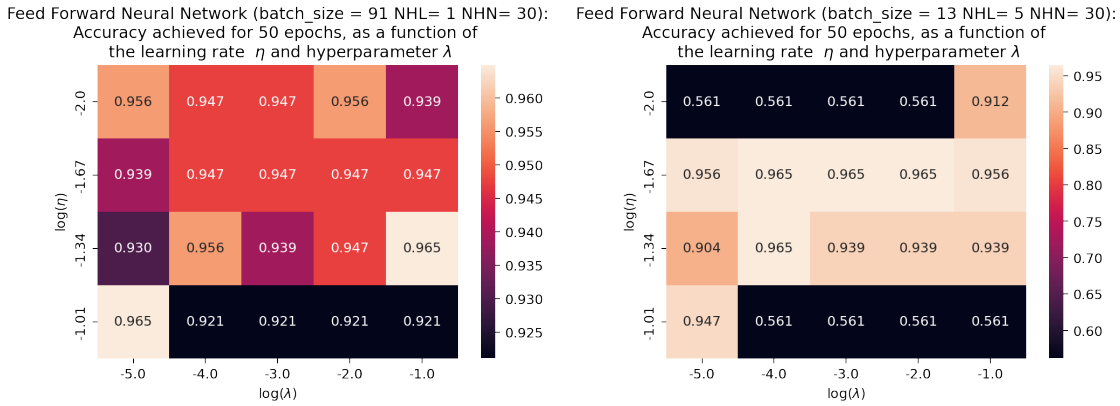


Figure 15. ReLU: Test accuracy heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs; `NHL= 1, 5 NHN = 30`.

Increasing the number of hidden nodes from 10 no matter the number of hidden layers resulted in a worse performance in terms of accuracy for `NHL= 1`. This can be seen throughout our testing where an extensive presentation can be found in **APPENDIX VI**, showing just that. Tying this back to our hypothesis, this could be due to scattering the data across many more nodes than what would be needed for one layer.

The most stable and accurately depicted results came from setting `NHL= 3`, given that fact we chose to explore the behavior of the NN for `NHN= 40` and `50`.

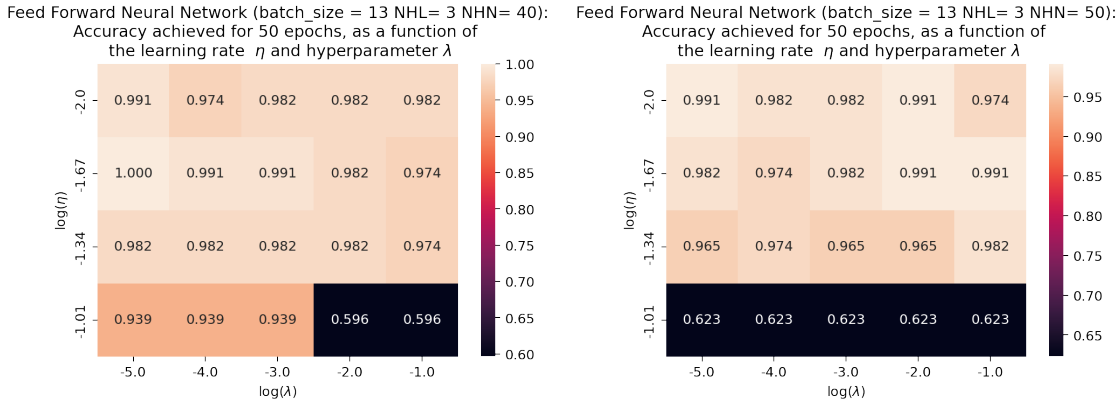


Figure 16. ReLU: Test accuracy heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs; `NHL= 3 NHN = 40, 50`.

Increasing the number of hidden nodes for `NHL= 3` gave better prediction results with the best accuracy being achived for $\log(\eta) \approx -2$; $-1.67$ from our testing. Although we obtained a staggering 100% accuracy for $\log(\eta) = -1.67$ and $\log(\lambda) = -5$, we advise the reader to disregard this result, since it's most likely due to pure luck and most likely can't be replicated. The accuracy has improved though to that of fig. 27: from 0.974, 0.982 (at best) to a steady 0.982 and sometimes 0.991.

The results obtained for `NHL= 5` in fig. 27 proves that the NN performs the worse at the margins for $\eta$ and that increasing the number of layers doesn't guarantee better predictions. The best approach is to tweak the number of hidden layers and nodes so that learner can be properly trained and that every node has the proper amount of datapoints to work with, meaning not too few and not saturated.

The Leaky ReLU analysis agrees with our finding so far. By that we mean setting the learning rate $\eta$ near the margins resulted in the poorest accuracy values, as shown in fig. 17

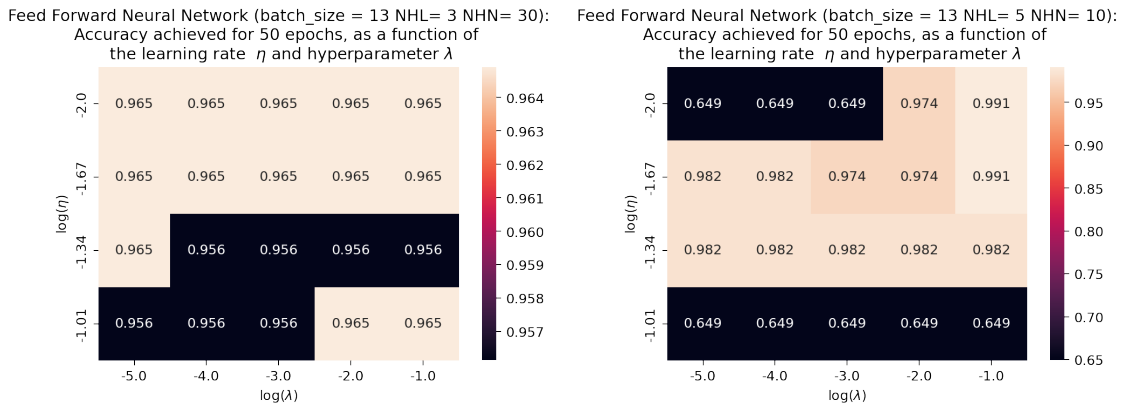


Figure 17. Leaky: Test accuracy heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs; `NHL= 3, 5 NHN = 10, 30`.

And building on what we found to be the case for ReLU, the most stable results were again obtained by setting `NHL= 3`. Given the fact the trend continued for Leaky ReLU we opted for further testing the NN's performance for `NHN= 40` and `50`.

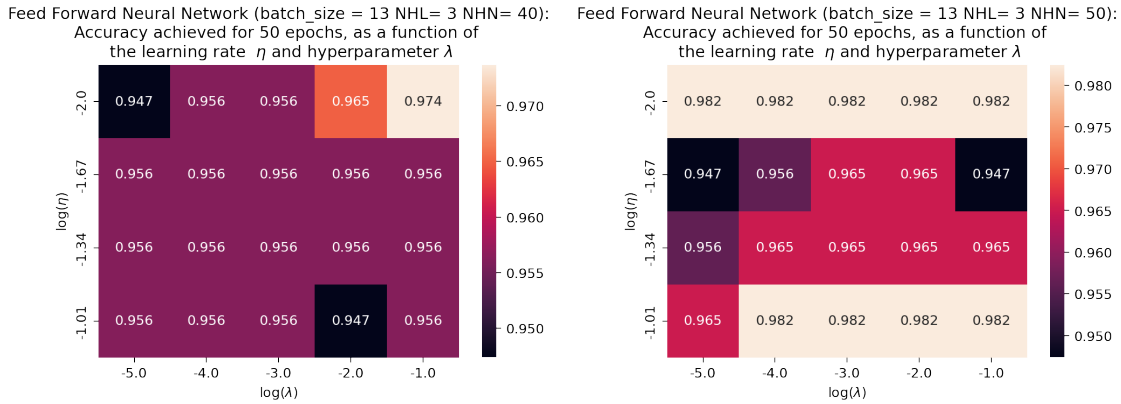


Figure 18. Leaky: Test accuracy heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs; `NHL= 3 NHN = 40, 50`.

Here is where this trend in our testing breaks, fig. 18 shows some doubtful results in comparison to what we've obtained from a lower number of nodes in fig. 28.

As for the results for `NHL= 5` that also didn't follow the footsteps of ReLU, since they were the most accurate of the bunch in terms of number of hidden layers no matter the number of hidden nodes; at least from what we have tested. The most accurate result was obtained by setting `NHN= 20`. With the highest accuracy being 0.991 for a mulitude of combinations of $\eta$ and $\lambda$.

What we can draw from all of this is that our analysis is good in determining the range of the learning rate and the combinations of the number of hidden layers and nodes that perform the worst. Through our analysis we can comfortably decide on what values of $\eta$ and composition of hidden layers and nodes we can carry on with for further testing, as we did. Paving in the way for further analysis to determine the best set of compositions and parameters values. We refrained from drawing any conclusion for the regularization parameter $\lambda$ as our results don't show any conclusive outcome as to what generally works the best. What we recommend moving forward, is to build upon what we have done so far in order to isolate or at least have a solid educated guess as to which values of $\lambda$ work the best. By that we mean, to run the same setup with the concluded best learning rate range and composition of hidden layers and nodes and then taking the running average over the number of multiple test-runs. Cause then the test accuracy results have to regress towards the mean given enough runs. Of course outliers (good or bad) can be dismissed if their behavior doesn't fit previous test-runs. This can be done by comparing the current test-run error to the average test-run error.

## V. CONCLUSION

### i. Franker

Our analysis showed that the SGD approach with momentum is accurate and reliable in terms of our testing and compared to the linear regression approach using OLS and Ridge. The test MSE is withing a margin of error in comparison to our regression analysis in [2]. The advantage with SGD is that it's more viable and SGD is the essential mechanism in any neural network. Which is why we went ahead and built our own Feed Forward Neural Network and contrasted it to the SGD results. The FFNN resulted in a measurable improvement approximately 0.018 for the test MSE with Sigmoid compared to that of Ridge 0.02 and OLS 0.03. Sigmoid was the generally more reliable approach by not differing much for $\lambda$ if $\eta$ was chosen correctly. ReLU and Leaky ReLU were adequate, although less reliable, generally yield on par testing scores or worse and required more tweaking of the parameters. As for the structure of the neural network the number of hidden nodes seemed to correlated with the polynomial degree of the design matrix. The number of hidden layers must tested for generally in order to decide on one, we decided on 3 in our case.

### ii. Cancer Data

The logistic regression approach for classifying the cancer data generally gave accurate and reliable results. For logistic regression with no L2-regularization the test accuracy was 0.965, with regularization for the most part our best result was at 0.947 with one outlier at 0.965 for $\eta = 1e{-}2$ and $\lambda = 1e{-}2$. We also followed the same procedure as for Franke and used the FFNN and compared that to the result produced by our logistic regression analysis. Sigmoid performed the best out of any activation function with accuracies in the range of 0.991-1.000, with stable and reliable results. Relu's accuracies ranged from 0.974 to 0.991 and leaky rely generally had an accuracy of 0.991. With both requiring more tweaking and generally being more sensitive to the choice of $\eta$ and $\lambda$. As for the structure of the neural network choosing a high number of hidden nodes increases the complexity of the neural network with no measurable improvements and at times resulting in a worse prediction. And also we can draw the same conclusion as for Franke, the number of hidden nodes seems to correlate with the number of features in the data. The best values were generally obtained for `NHN= 20, 30`, were in our dataset we had $p = 30$ features.

## REFERENCES


[1] M. Hjort-Jensen. Applied Data Analysis and Machine Learning. Jupyter book, (2021)

[2] M. Mahmoud and A. Davidov. Regression Analysis and Resampling Methodd. PDF, (2021)

# VI. APPENDIX

## i. Figures

### Franke: Sigmoid

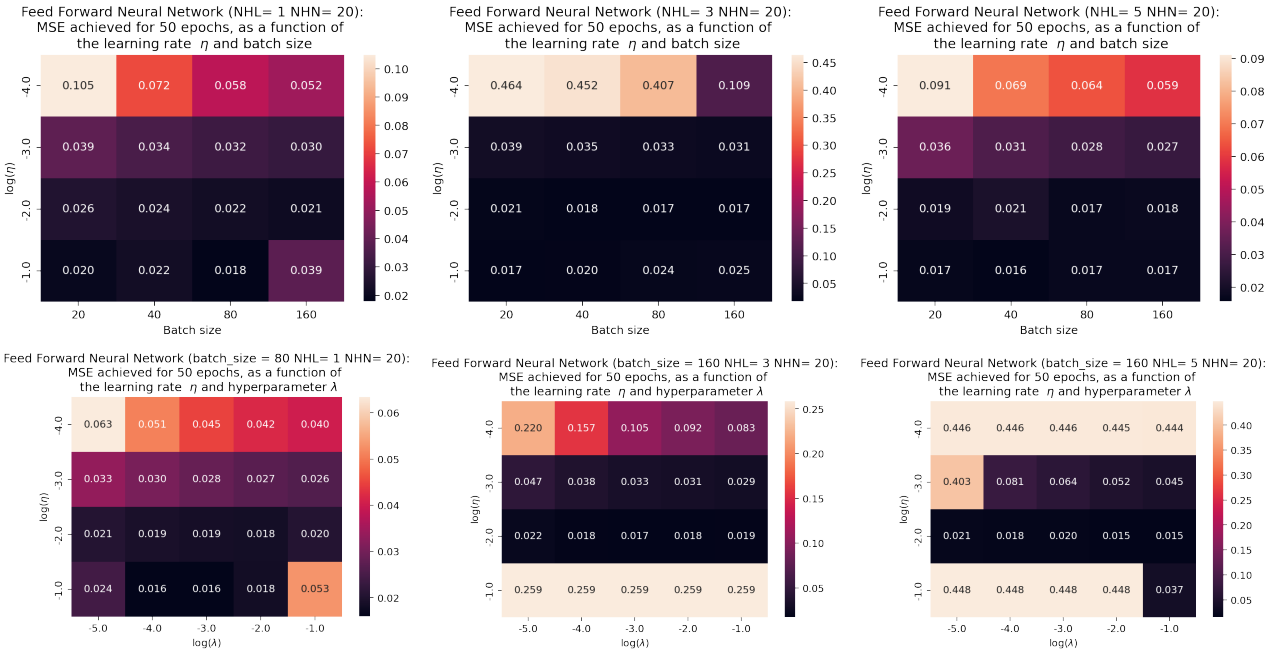


Figure 19. MSE heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs and `NHN= 20`. Batch size analysis (upper row), $\lambda$ analysis (lower row).

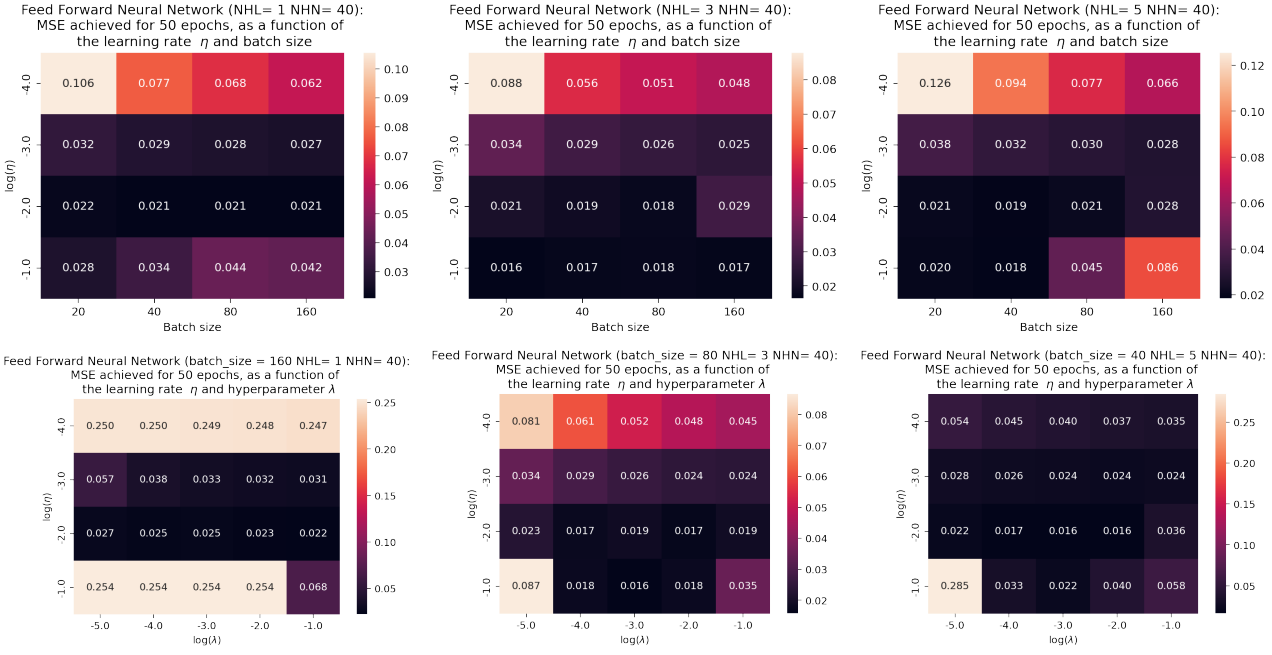


Figure 20. MSE heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs and `NHN= 40`. Batch size analysis (upper row), $\lambda$ analysis (lower row).

## Franke: ReLU

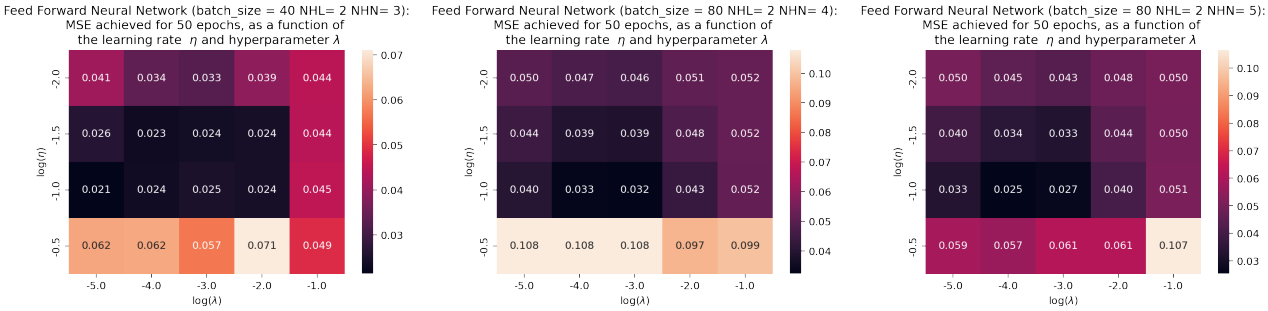


Figure 21. MSE heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs and `NHL= 2`.

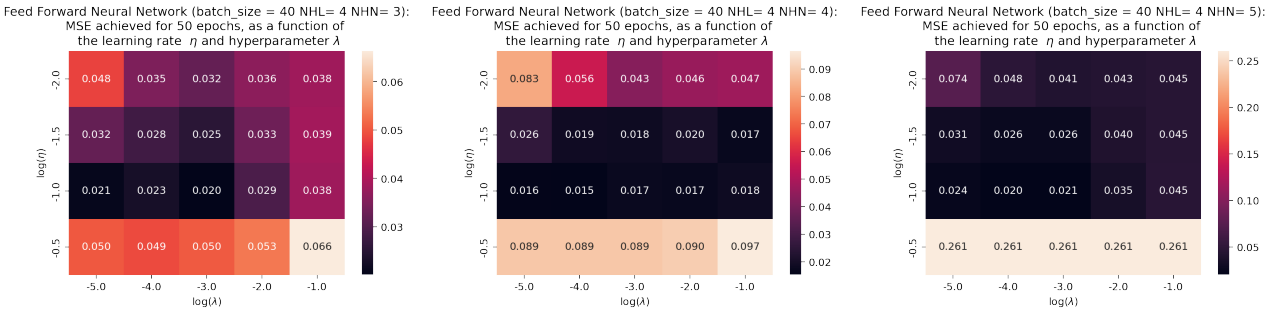


Figure 22. MSE heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs and `NHL= 4`.

## Cancer Data: Sigmoid

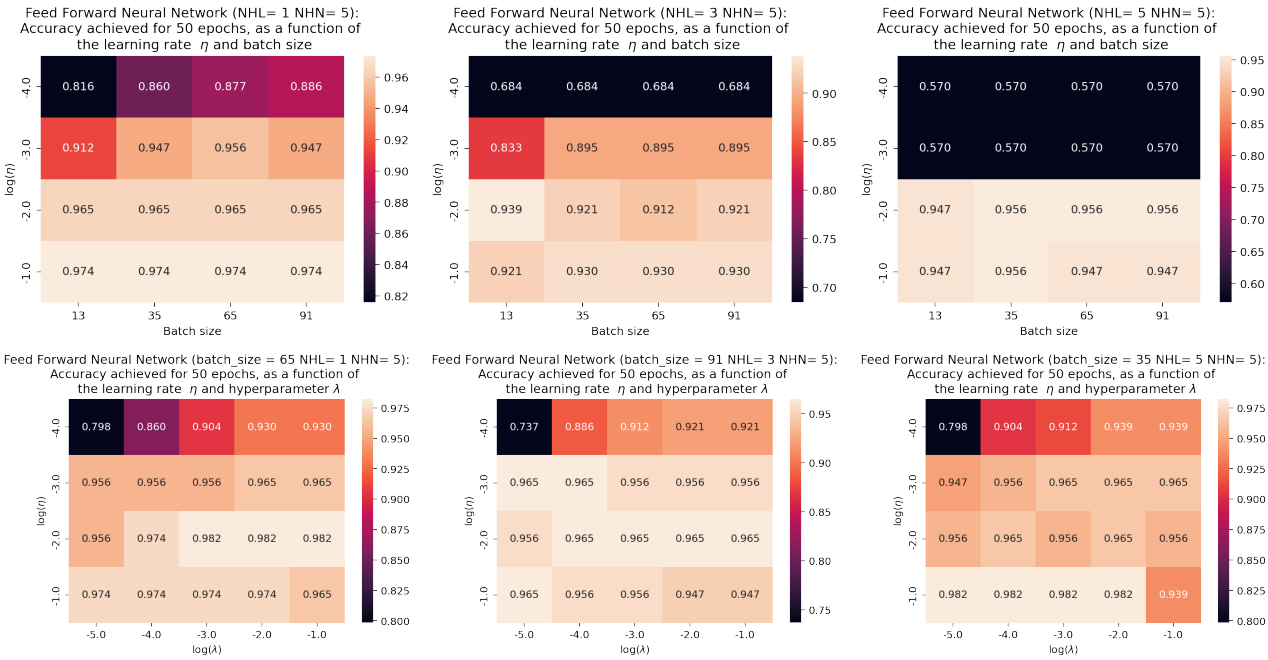


Figure 23. Test accuracy heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs and `NHN= 5`. Batch size analysis (upper row), $\lambda$ analysis (lower row).

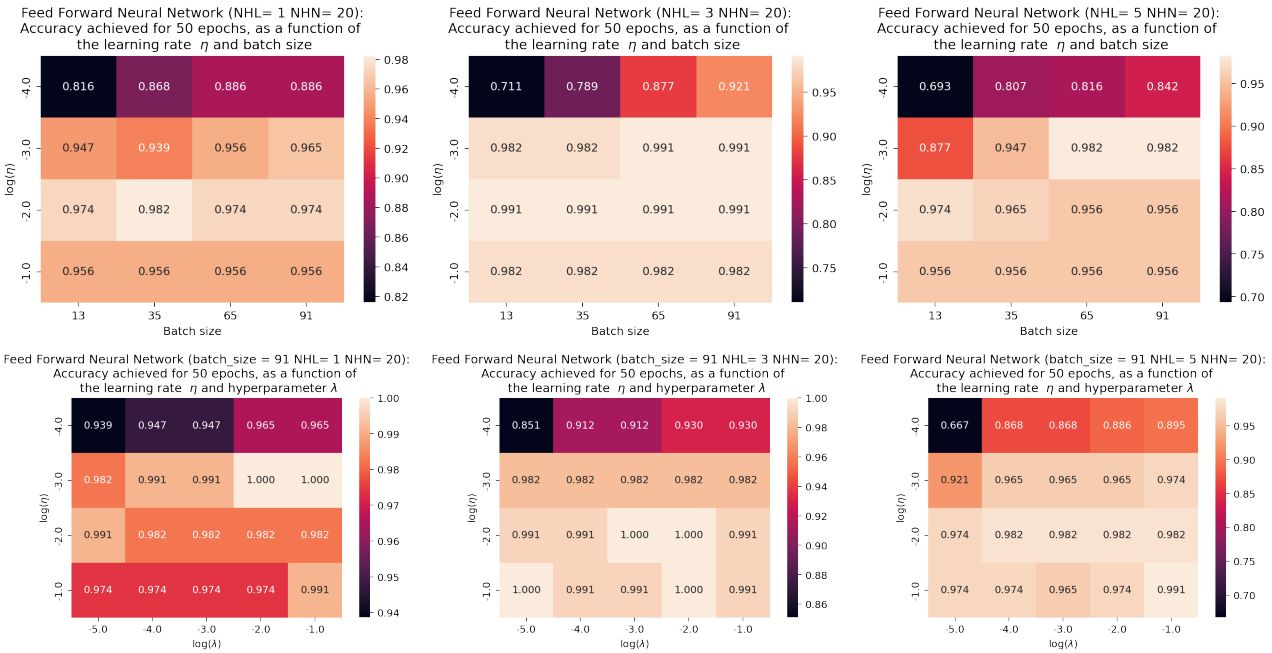


Figure 24. Test accuracy heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs and `NHN= 20`. Batch size analysis (upper row), $\lambda$ analysis (lower row).

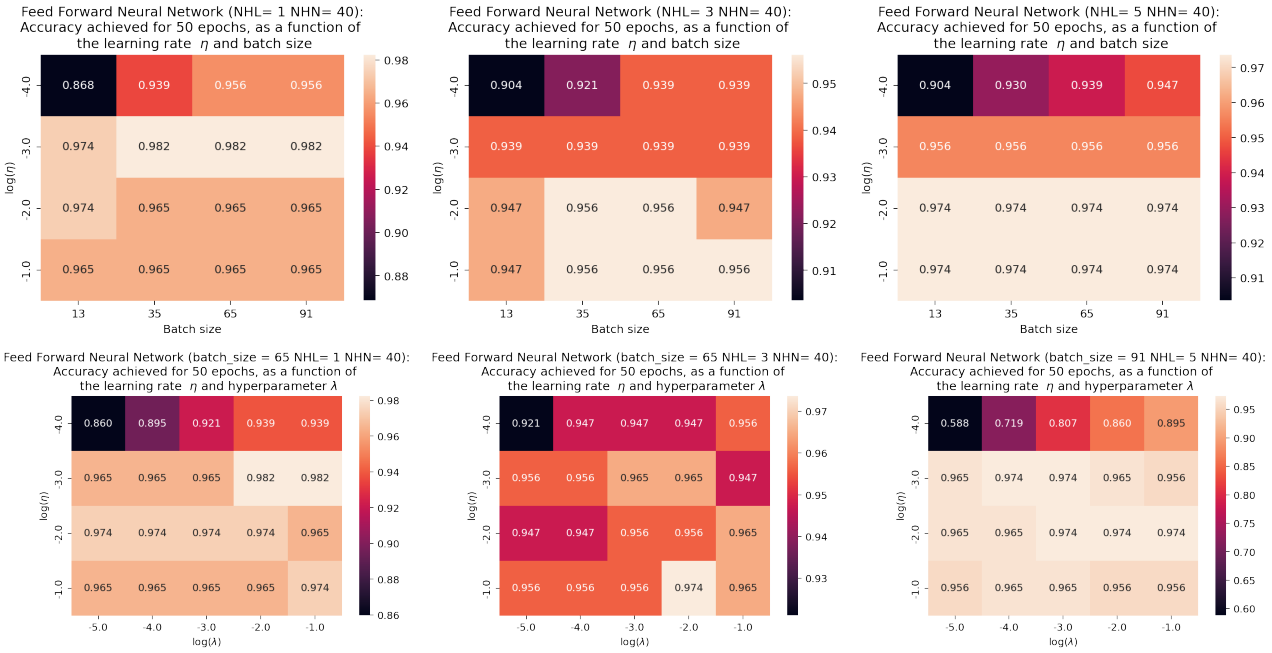


Figure 25. Test accuracy heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs `NHN= 40`. Batch size analysis (upper row), $\lambda$ analysis (lower row).

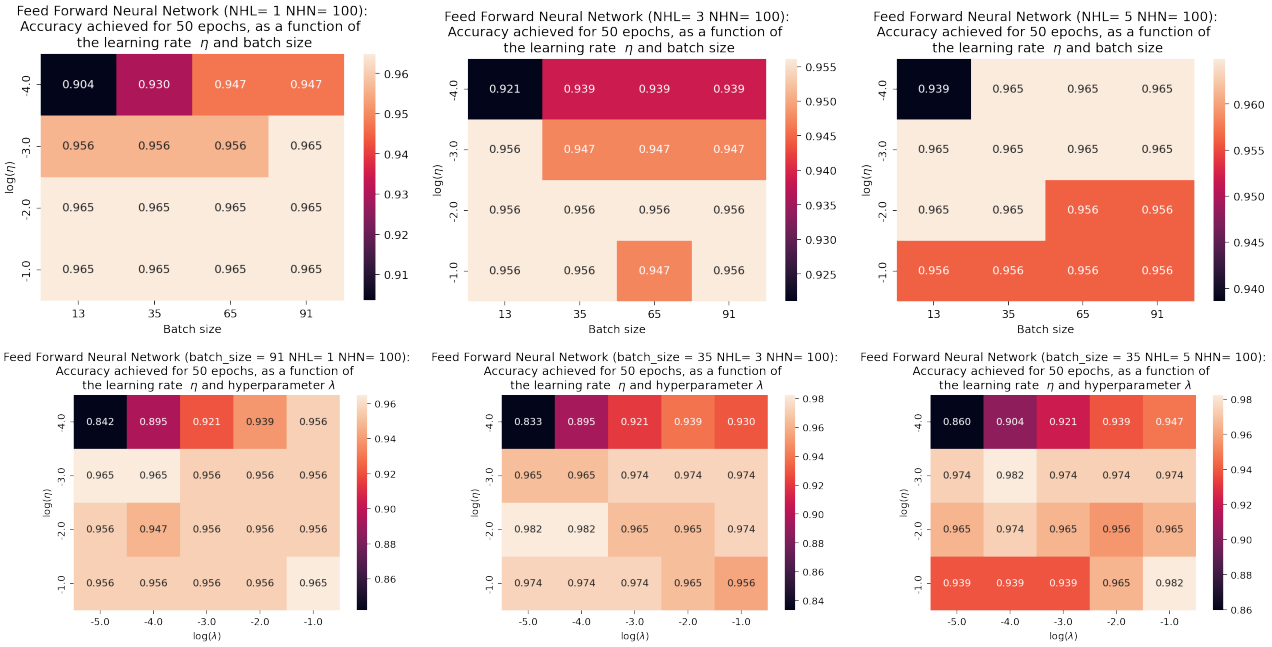


Figure 26. Test accuracy heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs and `NHN= 100`. Batch size analysis (upper row), $\lambda$ analysis (lower row).

## Cancer Data: ReLU and Leaky ReLU

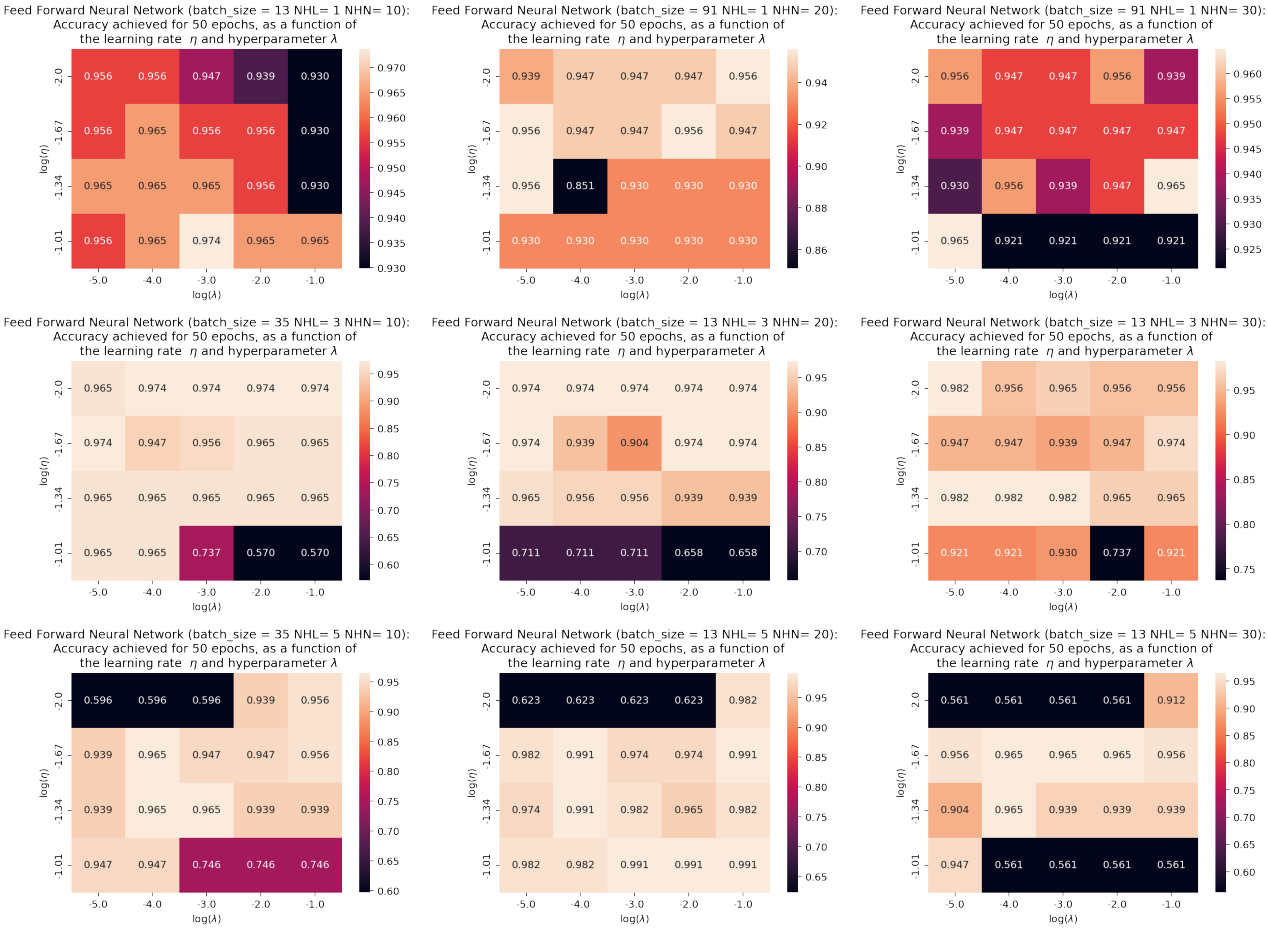


Figure 27. ReLU: Test accuracy heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs; `NHL= 1, 3, 5 NHN = 10, 20, 30`.

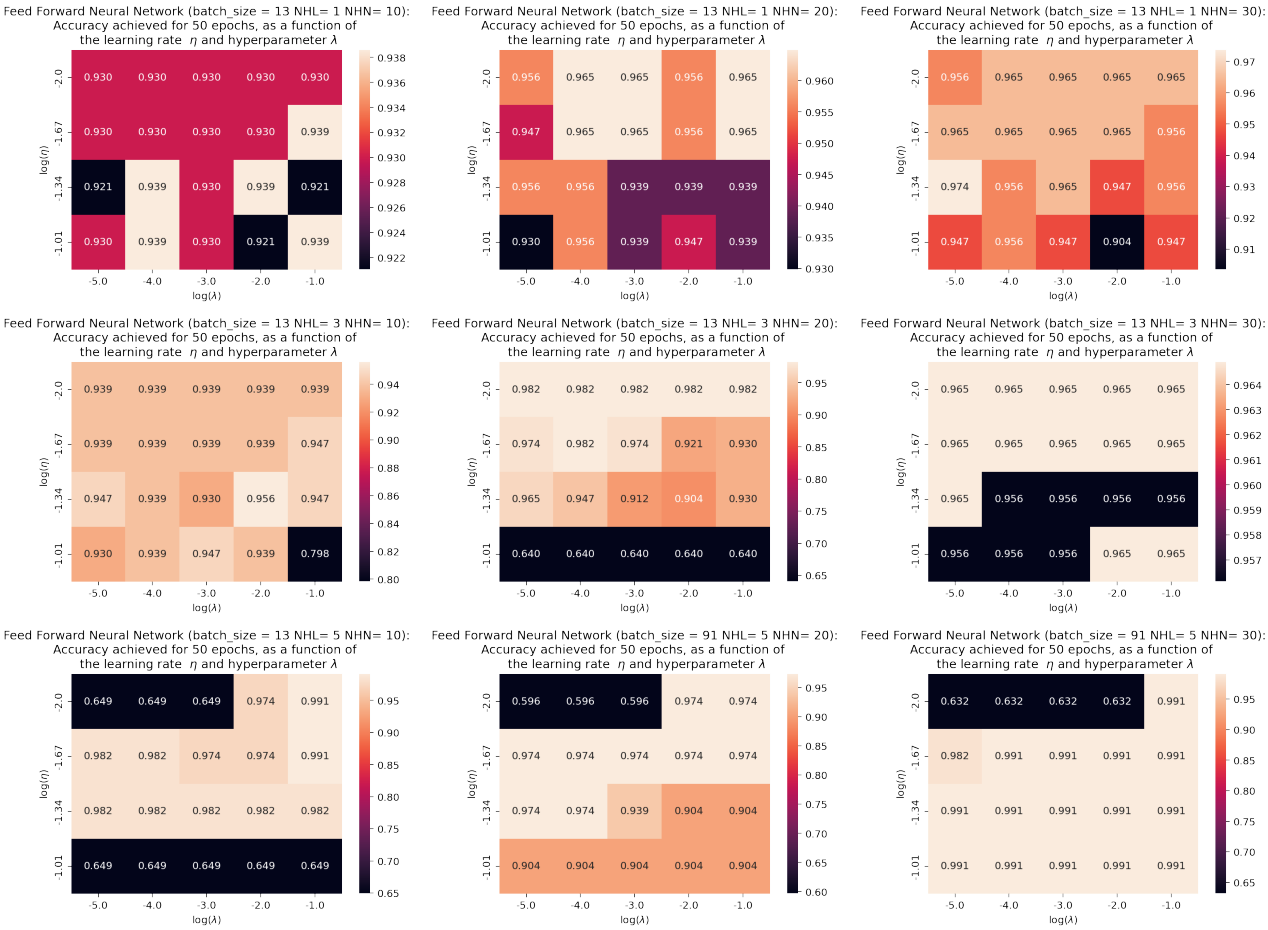


Figure 28. Leaky: Test accuracy heatmaps as a function of both the learning rate $\eta$ and $\lambda$, for a total of 50 epochs; `NHL= 1, 3, 5 NHN = 10, 20, 30`.