FYS-STK3155 Project 2

Didrik Sten Ingebrigtsen & Severin Schirmer

October 2020

GitHub repo: `https://github.com/didriksi/FYS-STK3155_project2`

Abstract

The aim of this project was to study regression and classification problems and to gain a deeper understanding of neural networks by developing our own feed forward neural network (FFNN). The regression problem was to approximate a function that takes in the coordinates and returns the height of terrain. For classification we studied a sample from the famous MNIST handwritten digits data set where the model predicts the digit based on an 8x8 monochrome image. The neural networks were compared to linear models and the classification had an additional Keras benchmark as comparison. We achieved good results on the classification problem with 98% correct predictions on unseen data, same as the Keras model though our implementation was significantly slower. We did not achieve good solutions to the regression problem due to the procedures which did not allow resampling without high computational cost. The regression models turned out to be too simple to make visually similar models to the terrain though we achieved $R^2 = 0.87$.

1 Introduction

Neural networks are a class of non linear statistical methods that can be used for a wide range of applications which fall under classification and regression. They have taken off since the breakthrough in image recognition in 2012 when convolutional neural networks were used in the ImageNet competition [1]. However, much of the underlying theory has existed for decades. Today neural networks are easily implemented using Pytorch or Keras, however, this does not provide much insight into how they actually work. Therefore, we have written our own code with the aim to understand how the models work and compared ours to the models provided by Keras.

In addition to our implementation of neural networks and Keras we do linear regression with stochastic gradient descent and logistic regression as a basis for comparison with the neural networks .

This report analyses three data sets. The first set is elevation data from an area in Ireland which provides us with a regression problem. The second is a collection of images of handwritten digits native to scikit-learn. This is a classification problem where we try to find out which digit was written.

The report starts with a an overview of the theory behind stochastic gradient descent, neural networks and logistic regression. This is followed by explanations of the code implementation. Finally, we present the results along with our discussion and conclusion.

2 Theory

2.1 Gradient descent

In our earlier works [4] [8], we have only looked at regression problems where there is an analytical solution for the parameters β given some hyper parameters, and some data. Finding optimal hyper parameters was still hard, and we had issues with overfitting to the seen data, but at least gauging the performance of a set of hyper parameters was trivial due to the training process being very short. In this report, we will look at cases where such an analytical solution does not exist, and where we therefore have to find some good parameters through iterative methods.

To investigate how to find a good β iteratively, let's start by defining a cost function $C : \mathbb{R}^{2\times n} \to \mathbb{R}^n$. It takes in our estimate $\tilde{\mathbf{y}}$ and the actual data $\mathbf{y}$, and returns some error metric. Mean squared error, $MSE = \frac{1}{n}(\tilde{\mathbf{y}} - \mathbf{y})^T(\tilde{\mathbf{y}} - \mathbf{y})$, is a common option for such a function. Then, since each $\tilde{\mathbf{y}}$ is dependent only on our parameters β and our design matrix X, for a given data set $\{\mathbf{x}, \mathbf{y}\}$, we can say that the cost function really only depends on β. Each possible β is associated with a cost, where the cost can now be interpreted as how well these parameters work on the given data set. Searching for the lowest cost, by changing the β parameters, can be done in several ways. When there's no analytical solution, the most common is gradient descent.

Gradient descent works by calculating the gradient of our cost function, with regards to our parameters, and then changing our parameters in the direction of the gradient, by some distance proportional to the norm of the gradient.

$$\beta_j + 1 = \beta_j - \gamma \nabla C_{\beta_j} \tag{1}$$

Here, γ is the learning rate, a number between 0 and 1 indicating how fast we should be moving downwards in the direction of the gradient. If our cost function is the mean squared error, and our β is defined through ordinary least square linear regression, our $\nabla C^{OLS}_{\beta_j}$ becomes

$$\begin{aligned}
\nabla C_\beta^{OLS} &= \frac{\partial}{\partial \beta}\frac{1}{n}(\tilde{\mathbf{y}}-\mathbf{y})^T(\tilde{\mathbf{y}}-\mathbf{y}) && (2)\\
&= \frac{1}{n}\frac{\partial}{\partial \beta}(X\beta-\mathbf{y})^T(X\beta-\mathbf{y}) && (3)\\
&= \frac{2}{n}X^T(X\beta-\mathbf{y}) && (4)\\
&= \frac{2}{n}X^T(\tilde{\mathbf{y}}-\mathbf{y}) && (5)
\end{aligned}$$

For Ridge the cost function is given by

$$C_\beta^{Ridge} = C_\beta^{OLS} + \lambda\beta^T\beta \tag{6}$$

And the derivative is

$$\nabla C_\beta^{Ridge} = \frac{2}{n}X^T(\tilde{\mathbf{y}}-\mathbf{y}) + 2\lambda\beta \tag{7}$$

2.2 Stochastic gradient descent

In equation 5, X, $\tilde{\mathbf{y}}$ and $\mathbf{y}$ can all get pretty large if we include the entire data set. Matrix multiplication can quickly get quite expensive, which is often very undesirable. We need some way of lowering the amount of computations here, and the easiest way to do that is to include less data in each iteration. Stochastic Gradient Descent (SGD) is one way of doing that. SGD works by randomly dividing all the data into mini-batches, and perform each step by calculating the cost on just one of them. This introduces stochasticity, hence the name.

The idea is that for a subset of the data the gradient of the cost function will point in the general direction of the gradient for the whole set[7].

$$\frac{\sum_{j=1}^{m}\nabla C_\beta}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C \tag{8}$$

where the first sum is over the mini batch and the second is over the entire set of training data.

2.3 Momentum gradient descent

A problem with gradient descent that SGD doesn't fix entirely is the possibility of converging in small local minima of the cost function. Unless the cost function is constantly convex, there are many small holes a gradient can get stuck in. Stochasticity can bump our model out of the hole, but there are several other techniques that can be layered on top to add some more stability. One of them, and the one we implemented in this project, is momentum gradient descent. It adds something analogous to real world momentum to the gradient descent, both making the descent faster, and letting it more frequently skip minima for small spans of parameters. Mathematically, momentum works by replacing the gradient descent equation 1 with this

$$v_t = v_{t-1} * \mu + \gamma * \nabla C \beta_t = \beta_{t-1} - v_t \tag{9}$$

where $\mu \in [0, 1]$ is a hyper parameter dictating how much of the previous momentum should be kept. If $\mu = 0$, there is no momentum, and everything acts like before, and for $\mu = 1$, the rate of change is basically summed up over time, and can for many cases tend towards infinity. There is therefore a trade-off here, between skipping local minima and doing it quickly, but also possibly skipping the actually likely candidates for global minima, and having to try more initial conditions. The optimum momentum hyper parameter $\hat{\mu}$ varies between different models and data sets, and should therefore be tuned.

2.4 Adaptive learning rates

When initialising our parameters, they are likely very far away from their optimal versions. The best strategy for getting towards the optimal parameters from there is probably to make big leaps in whatever direction leads down the gradient. This will both save on computations, and reduce the risk of getting stuck in small local cost minima. However, after some time, if everything works, they should get closer to better minima, and we want to reduce the risk of overshooting, and also just oscillating back and forth without ever descending into the lowest part of this convex area of the cost function.

To be able to do both these things, a fixed learning rate will work suboptimally. We therefore need adaptive learning rates. Adaptive learning rates can be very complicated, and be a function of for example loss and gradient. In this project, we will only look at a simple class of adaptive learning rates, namely the kind that only takes in the step number. More specifically, we will be using step-based decay, with a function like this

$$\gamma(i; \gamma_0, \tau) = \frac{\gamma_0}{1 + i \cdot \tau}$$

,

where γ is learning rate, i is step number, γ_0 is the initial learning rate, and τ is some decaying time constant.

2.5 Classification and regression

In project 1, we worked with a regression problem. The goal was to estimate how high the terrain was at any point, based on the coordinates. In this project, we will continue with the same regression problem, but we will also be trying to solve a classification problem. Classification problems are problems where we want to say not how much of some quantity that is, but to describe a target qualitatively. A related classification problem to our regression terrain problem in project 1 would be a problem where we get terrain data, but are supposed to predict whether the terrain represents for example a mountain, a plain, or a valley.

To evaluate how well our model performs we have to use different metrics depending on what type of problem we are trying to solve. In the terrain regression problem, mean squared error ($MSE = (\tilde{\mathbf{y}} - \mathbf{y})^2$) described our error well. It did this because we want to penalise large errors more than we want to penalise small errors - we'd rather want our model constantly 10 meters off of the target, than for them to match perfectly 9 out of 10 times, but then shoot of for an error of 100 meters at the last one. All of our estimates are also continuous, so subtracting the expected from the computed gives us a number that itself tells us a lot about the error.

If we on the other hand look at a classification problem, it isn't even obvious that subtracting the expected from the computed is a valid operation. How can we subtract a mountain from a valley? Two options crystallise: Either, we can give each class a unique number. So, subtracting mountain (2) from a valley (1), would result in an error of 1. However, this implies that there exists some simple relationship between the classes, that a mountain is twice of a valley, and that a mountain added with a valley is an ocean. This is rarely the case, and therefore, one-hot encoding is the very practical second option. It works by making every class a vector instead of a scalar, with a length equal to the amount of classes. All of the numbers in the vector are 0, except for one, which is labeled as 1. This might seem very similar since all the classes get assigned a unique number, but since that number is used as an index in a vector an not just on its own, there is no implied relation between the different classes. The set of one-hot encoded vectors become an orthonormal basis, actually the standard basis, for $\mathbb{R}^n$, where n is the amount of classes.

If our prediction is then made as a vector of the same length, we can now measure distance to the correct solution in an intuitive way. Taking the 2-norm of the vector between the one-hot vector of the actual class we wanted the model to predict, and the one it actually did, would be the same as MSE defined above here. This works decently, but isn't actually the best way. This is because when we descend the gradient of MSE, it isn't really that steep when the error is large. MSE has a squared relationship between 1-norm error and cost, so its gradient has a linear relationship with it. In one-hot encoded classification, the numbers are small, and the distance the output has to change is therefore small. However, the parameters still have to change quite a lot. We'll get back to what kind of activation functions work well for this later, but we need a cost function that penalises the comparatively small 1-norm errors in the output layer, harder than MSE. A good alternative, is $cross - entropy$ loss [7]. It is defined as

$$C_{CE} = -\frac{1}{n}\sum_i (y_i \ln(\tilde{y}_i) + (1 - y_i)\ln(1 - \tilde{y}_i)) = -\frac{1}{n}\sum_i y_i \ln(\tilde{y}_i),$$

Here, since $y_i = 0 \vee y_i = 1$ for all i, there is only one non-zero term in each sum and that term has a high value if the estimate is far from the target. The function comes from information theory, and describes the amount of change that needs to be done to one data set to make another [6]. However, the important quality it has that we're looking for, is that $|\nabla C| \to \infty$ when the difference between an element of $\tilde{\mathbf{y}}$ and $\mathbf{y}$ goes towards 1. Therefore, we will use cross-entropy as the loss function for our classification problems.

2.6 Neural networks[1]

The theory described until now builds up to neural networks, a powerful machine learning strategy that can handle systems of higher complexity than linear models. In this project we study feed forward neural networks (FFNN). A feed forward neural network is built up of layers of nodes where the first layer, the input layer, receives X and it passes through a number of hidden layers until it reaches the output layer where regression models often have one output node as in our case where we have $\mathbb{R}^2 \to \mathbb{R}$. A classification network on the other hand will have K output nodes corresponding to the number of possible classes [3].

Each layer in the network, excluding the input layer, consists of initially randomly distributed weights w, biases b, and a chosen activation function σ. The weights and biases create a linear combination of the output from the previous layer and the activation function provides the non linearity of the model allowing it to fit to more complex structures. One of the most common activation is the sigmoid function, which squishes the input to the function into a number between 0 and 1.

$$\text{sigmoid } \sigma(z) = \frac{1}{1+e^{-z}} \tag{10}$$

[1]When talking about neural networks, we will refer to the input vector as a layer, even though it doesn't have activation functions, weights or biases. We still count them as a layer mostly due to the way we have implemented them. Also, where as we before have used bold types to indicate a vector, this becomes impractical when writing about neural networks, so we will abstain from it from here on and out.

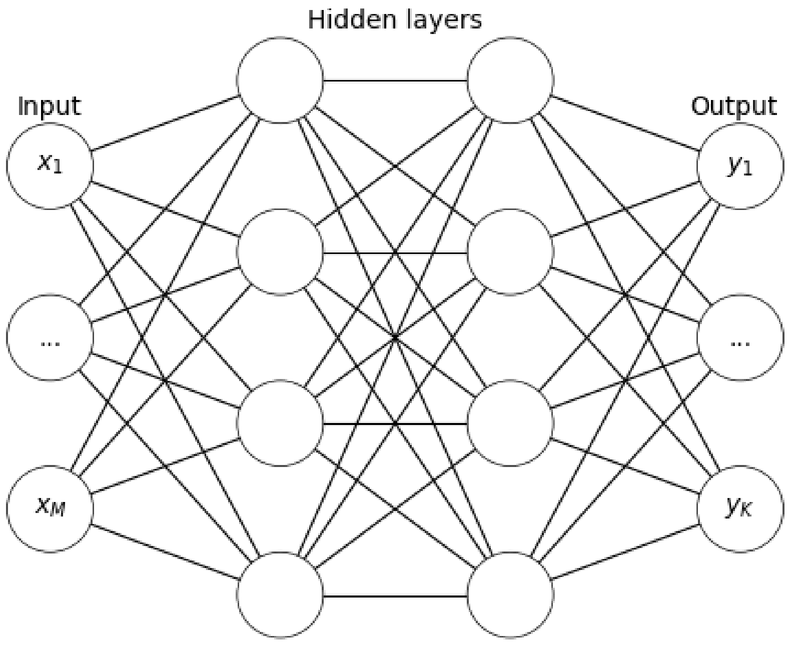


Figure 1: A 4-layer general neural network with M inputs and K outputs.

Each node in a layer receives a vector a of activations from the previous layer or from the input if it is the first layer. The activation of the j^{th} node in a layer is given by

$$z_j = w_j^T \cdot a + b_j \tag{11}$$

$$a' = \sigma(z_j) = \frac{1}{1 + e^{-z_j}} \tag{12}$$

where the first element of w corresponds to weight between the first node in the previous layer and the j^{th} node in the current layer. This operation can be vectorized for the whole layer. Then we have a matrix w containing every w_j and such that the element w_{jk} of the matrix contains the weight between the k^{th} node of the previous layer and the j^{th} node in the current layer [7]. The vector of activations for the current layer is now quite simple

$$a' = \sigma(wa + b) \tag{13}$$

Due to the sigmoid every element in a' is in the range $[0, 1]$. For the output layer the activation function depends on the type of network. For linear regression we use a simple linear activation function $f(a) = a$. Sending the an x through a network is called feed forward. And an example with three layers, $L = 3$, could look like

$$\begin{aligned} a^{(1)} &= \sigma(w_1 x + b_1) \\ a^{(2)} &= \sigma(w_2 a_1 + b_2) \\ a^{(3)} &= f(w_3 a_2 + b_2) \end{aligned}$$

For the general case the last layer is denoted by $a^{(L)}$. Now that we are able to send data through the network we have to find the error of each prediction and tune the parameters w and b accordingly.

2.7 Backpropagation

The goal when training the network is to understand how we need to change the weights and biases to reduce the value of the cost function. The parameters are initially randomly distributed and using backpropagation and SGD the goal is that the cost function converges towards a global minimum. For the networks used in this report we used the mean squared error (MSE) and the cross entropy cost functions for regression and classification respectively. Both can be considered as functions of the parameters because there are a fixed x and y for each iteration.

$$C_{MSE} = (y - a^{(L)})^2 \tag{14}$$

$$C_{CE} = -\frac{1}{n} \sum_i y_i \ln(a_i^{(L)}) \tag{15}$$

The first step is to find how to change the weights in the last layer to affect the cost. We need $\frac{\partial C}{\partial w^{(L)}}$. C depends on $a^{(L)}$ which depends on $z^{(L)}$ which in turn depends on the weights $w^{(L)}$. According to the chain rule we have

$$\frac{\partial C}{\partial w^{(L)}} = \frac{\partial C}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial w^{(L)}} \tag{16}$$

With C being the mean squared error and considering a single iteration k we have

$$\frac{\partial C_k}{\partial a^{(L)}} = 2(y - a^{(L)})$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)})$$

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = \frac{\partial}{\partial w^{(L)}} \left(w^{(L)} a^{(L-1)} + b^{(L)} \right) = a^{(L-1)}$$

$$\frac{\partial C_k}{\partial w^{(L)}} = a^{(L-1)} \sigma'(z^{(L)}) 2(y - a^{(L)}) \tag{17}$$

We define $\frac{\partial C_k}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} = \delta^{(L)}$ which simplifies notation going forward. With stochastic gradient descent we average over multiple iterations so we have

$$\frac{\partial C_b}{\partial w^{(L)}} = \sum_{k=0}^{m-1} \frac{\partial C_k}{\partial w^{(L)}} \tag{18}$$

Where m is the number of iterations in a mini batch and b is a single mini batch. Thus we know the direction we need to change the weights in in order to decrease the cost. Similarly for the bias we have

$$\frac{\partial C}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C}{\partial a^{(L)}} = \sigma'(z^{(L)}) 2(y - a^{(L)}) = \delta^{(L)} \tag{19}$$

We now have that both the weights and the biases are defined by δ. Therefore we just need to calculate δ for each layer and for the weights we multiply with the activation in the previous layer which we already have from the feed forward procedure.

If a different cost function is used we simply switch $2(y - a^{(L)})$ in δ with the derivative of the new cost function. For cross-entropy the derivative is given by

$$\frac{\partial C_{CE}}{\partial z^{(L)}} = a^{(L)} - y = \delta \tag{20}$$

Same goes for different activation functions where $\sigma'(z)$ can be switched for the derivative of another activation function.

We have now looked at the case for the output layer and it being a single neuron. Going backwards through the whole system just means finding $\frac{\partial C_k}{\partial a^{(L-k)}}$ which is just a continuation of the chain rule. For the output layer L, δ looks the same but for a hidden layer l we have

$$\delta^{(l)} = \left[w^{(l+1)} \delta^{(l+1)T} \right]^T \sigma'(z^{(l)}) \tag{21}$$

where each element is a vector and the weights are a matrix. For applying stochastic gradient descent we iterate through the layers with

$$w^{(l)}_{new} = w^{(l)}_{current} - \gamma \delta^{(l)} a^{(l-1)} \tag{22}$$

$$b^{(l)}_{new} = b^{(l)}_{current} - \gamma \delta^{(l)} \tag{23}$$

where γ is the learning rate. For SGD with momentum equations 22 and 23 can simply be considered as v_t in the equation for momentum 9.

2.8 Activation functions

Depending on the application of the neural network different activation functions are used. For regression we usually just output the value of the output layer

$$f(z) = z \tag{24}$$

$$f'(z) = 1 \tag{25}$$

where f can produce all real numbers. However, in classification we want to be able to interpret the output as a probability for each class. Therefore the sum of the activation of each of the K output nodes corresponding to the K classes must sum to 1 as there is a 100% probability of getting one of the classes. For this we use the softmax function

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{26}$$

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)) \tag{27}$$

In the hidden layers we use either the sigmoid, σ, ReLu or Leaky ReLu. The sigmoid is discussed above, but the ReLu produces values in $[0, \infty\rangle$

$$ReLu(z) = \begin{cases} z, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases} \tag{28}$$

$$ReLu'(z) = \begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{if } z < 0 \end{cases} \tag{29}$$

An issue that is most often encountered in deep learning, where the number of layers becomes relatively big, is that of vanishing gradients [10]. When $|z|$ becomes very big the gradient of the sigmoid gets close to 0 or for ReLu the gradient is 0 for $z < 0$. If we have n hidden layers using the sigmoid function we have n small derivatives multiplied together which means the gradient decreases exponentially as we propagate back through the layers [10]. For ReLu the issue is called dying ReLu and is not necessarily caused by a big network. It can be solved with the Leaky ReLu function for which the derivative returns a small constant for $z < 0$ instead of 0 due to the function having a slight slope before for $z < 0$.

$$Leaky\ ReLu(z) = \begin{cases} z, & \text{if } z \geq 0 \\ \alpha z, & \text{if } z < 0 \end{cases} \tag{30}$$

where α is small constant.

$$Leaky\ ReLu'(z) = \begin{cases} 1, & \text{if } z > 0 \\ \alpha, & \text{if } z < 0 \end{cases} \tag{31}$$

2.9 Logistic regression

Logistic regression is used for classification problems. We wish to calculate the probabilities of the K classes and that they sum to 1 and are in $[0, 1]$. The model is given by

$$log\left(\frac{P(G = k|X = x)}{P(G = K|X = x)}\right) = \beta_{k0} + \beta_k x, \quad \text{for } k = 1, ...K - 1 \tag{32}$$

and we have

$$P(G = K|X = x) = \frac{1}{1 + \sum_{l=1}^{K-1} \exp(\beta_{l0} + \beta_l^T x} \tag{33}$$

The probability of K is defined the by the probability of the other classes as we have $P(K|x) = 1 - \sum_{k=1}^{K-1} P(k|x)$ [3]. Therefore we have one set of β less to find when training making it less computationally expensive. We can then easily rewrite equation 32 to

$$P(G = k|X = x) = \frac{\exp(\beta_{k0} + \beta_k x)}{1 + \sum_{l=1}^{K-1} \exp(\beta_{l0} + \beta_l^T x} \tag{34}$$

Logistic regression is basically a neural network with only an input layer and an output layer where the activation function is the softmax function

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{35}$$

This is because we can interpret $\beta_{k0} + \beta_k x$ as

$$\beta_{k0} + \beta_k x = b + wx = z \tag{36}$$

However, this approach does not utilise the reduced number of parameters, but can be easier to implement.

3 Implementation

3.1 Stochastic gradient descent

To implement SGD, we have used this basic algorithm:

Algorithm 1 Pseudocode for stochastic gradient descent

```
procedure SGD($x_{train}, x_{test}, y_{train}, y_{test}, model, epochs, mini\_batch\_size$)
    $indexes = \{k|k = 0, 1, 2, \dots, size(y)\}$
    $errors =$
    for $epoch \in [1, epochs]$ do
        $these\_errors = MSE(model.predict(x_{test}), y_{test})$
        $errors+ = these\_errors$
        if $mean(these\_errors) > prev\_errors$ then
            end procedure
        end if
        $prev\_errors = these\_errors$
        $shuffle(indexes)$
        $mini\_batches = reshape(indexes, mini\_batch\_size)$
        for $mini\_batch \in mini\_batches$ do
            $\tilde{\mathbf{y}} = model.predict(x_{train}[mini\_batch])$
            $\nabla C = model.gradient(x_{train}[mini\_batch], y_{train}[mini\_batch], \tilde{\mathbf{y}}[mini\_batch]))$
            $model.update\_parameters(\nabla C)$
        end for
    end for
    return $errors$
end procedure
```

As is evident in the pseudocode, our early stopping algorithm works by comparing the validation error for the last epochs, with the current one, and stop if it has gotten worse. For computational efficiency and for making it a bit more stable, in our actual implementation we only do this every n epochs, and then compare the mean of the error of the last n epochs, with the mean of the n epochs before that again. The strengths of this algorithm include low computational cost, and an effective tool against overfitting. On the other side, we don't save models and revert back to them if we start doing worse, so we still might overfit some. We also run the risk of converging at a set of parameters, and not improving (or getting worse) on validation data for thousands of epochs without stopping, something checking the norm of the gradient could solve.

We have also created a series of helper functions that allow us to easily set up and plot errors for different combinations of hyper parameters:

1. `make_models` takes in model classes, parameters that should be the same for all of them, and parameters that should change, and returns a list of models.
2. texttt_sgd_on_model takes in x and y data, a list of models, as well as the corresponding parameters for the sgd function, and put all the errors into a big dataframe. The dataframe has indexes decided by texttt_filter_dicts, that finds similarities and differences between models.
3. texttt_plot_sgd_errors takes in the dataframe or errors, and plots them using the texttt_side_by_side function from the plotting module.

This series of functions makes it so that it is possible to very flexibly and with few lines of code test almost any set of models against each other. However, the code for doing that, is unintuitive, and became bloated. This is certainly one part of the code base that could benefit from more work. Because the interface has gotten a bit complicated, we will quickly go through some example code to use it, so that it is possible to for the reader to understand the code in the repository later.

```python
def example_regression(*data):
    common_kwargs = {'momentum': 0.6}
    subplot_uniques = [{'layers': [{'height': 2}, {'height': h_height}, {'height': 1}} for h_height in [1, 2, 3]]
    subsubplot_uniques = [{'learning_rate': l_rate} for l_rate in np.logspace(-3, -1, 3)]

    unique_sgd_kwargs = [{'mini_batch_size': 10}, {'mini_batch_size': 20}]

    subplot_models = helpers.make_models(
        neural_model.Network,
        common_kwargs,
        subplot_uniques,
        len(unique_sgd_kwargs),
        subsubplot_uniques
        )

    subplots = [(models, sgd_kwargs) for models, sgd_kwargs in zip(subplot_models, unique_sgd_kwargs*len(subplot_models))]

    errors, subtitle, subplots, metrics_string = sgd.sgd_on_models
    (*data, *subplots, epochs=300)

    title = ['Example', 'example', subtitle]

    sgd.plot_sgd_errors(errors, title, metrics_string)
```

This code will train models with all possible combinations of the given mini batch sizes (10, 20), learning rates $(10^{-3}, 10^{-2}, 10^{-1})$, heights of hidden layers (1, 2, 3), but will share a momentum of 0.6. It will be divided into $len(subplot\_uniques) \cdot len(unique\_sgd\_kwargs)$ amount of subplots, each having $len(subsubplot\_uniques)$ amount of plots in them.

3.2 Tuning

To find the optimal hyper parameters, we made a tuning class, that takes in a list of models with different hyper parameters, uses SGD to train them, and then makes heatmaps of errors on validation data. This code does something similar to the function `sgd_on_models` discussed above and could possibly be merged.

Another problem with the tuning code is that it is very unstable. In the previous report, we used resampling techniques to test different hyper parameters, and get a good estimate of errors as a function of them. This time, training is so much more computationally expensive, so we can't really do resampling. This means that the variance between different tuning runs is probably large. As the variance is dependent on the algorithms used for training as well as the data, we ironically have no way of estimating it properly without actually resampling, so we don't know how big of an issue our lack of resampling is.

3.3 Models

We made the SGD code quite general, so that it would later be easy to compare not only models with different parameters, but also entirely different models. This required a similar interface to be added to all our models. An option would be to add a super class that all the model classes inherited from, and at first this seemed reasonable. However, even though the class structures were similar, there were really no methods that could share anything meaningful between the neural models and the logistic model on one side, and the Ridge and OLS models on the other side. That means our Ridge model has the OLS class as a super class, and that we implement logistic regression as a special case of the neural network. Strictly speaking, we could have implemented the Ridge and OLS models as special cases of the neural model, with linear activation functions, only one layer, and some other small adjustments, but we found this just made the code more difficult to read, and not necessary when everything is already working.

3.4 OLS and Ridge models

Let us start by discussing the implementations of the linear models. We started with some of the code from [4] and developed it further. The new additions include a more expansive constructor taking in hyper parameters like the momentum z, the learning rate γ, and the method for initialising β. We introduced a method for updating the parameters in a step, that uses the momentum eq 9 to do a step down a gradient. This method is shared between all of the linear models, while the method for calculating the gradient varies between them based on their cost functions explained in the theory section.

3.5 Neural and logistic models

The neural network is implemented as a class `Network` and three layer classes, `Input`, `Dense`, and `Output` which are sub-classes of `Layer` and `Output` is a subclass of `Dense`. Using these we build sequential networks with fully connected layers. Feed forward, prediction, backpropagation and updating parameters are implemented as methods in `Network` and the training of the network uses the SGD algorithm explained above.

The activation function of each layer and the cost function can be defined when adding the layers which makes testing of different model architectures trivial. Networks for regression and classification can easily be set up without changing the instances of the classes significantly. However, we encountered problems when we combined the softmax activation function with cross entropy loss when the network had one or more hidden layers. In the end we opted for Nielsen's approach with cross entropy and sigmoid activation in the final layer [7]. This results in a smoothly running network but we lose softmax's advantage of interpreting the output as probabilities. This is discussed further in the results.

For logistic regression the problem described in the previous paragraph did not occur and it was implemented as explained in the final part of the theory. We simply used an input and an output layer with the softmax activation and cross entropy loss to produce a logistic regression algorithm.

3.6 Learning rate

To make our adaptive learning rate functions easy to construct and reuse, we made a parametrised class, that can take in τ and γ_0, and return a function.

Because we had some issues with models occasionally performing so poorly that they ended up diverging if the learning rate was too high in the beginning, we also included a linear ramp up in the beginning of our learning rate function. This way, if the model was off by a lot, it would still make reasonable steps the first few times, and then hopefully be close enough to the relevant parameter area by the time the learning rates got high.

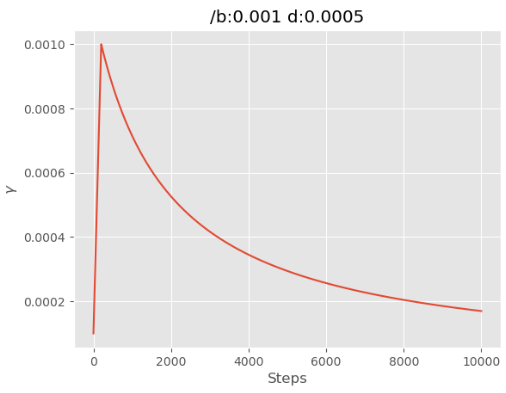


Figure 2: Example learning rate function with $\gamma_0 = 0.001$, $decay = 0.0005$, and $ramp\_up\_steps = 200$, shown for the first 10000 steps.

3.7 Data

Since we want to apply models to both classification and regression problems, we need to have two different data sets.

3.8 Regression

For our regression problem, we decided to use terrain data, since we have experience with that from project 1 [4] [8]. This data is pulled from US Geological Studies [9]. The task at hand is to use some polynomial basis expansion of our coordinates to approximate the height.

We split the data with the same algorithm as we did in the last project, by picking training and validation data on two offset and regular grids, and then putting the rest of the data off as testing data. The advantage with this is that we ensure that we don't have any big gaps in our training and validation sets, making the chance that our data is unbiased higher. In table 1 we have shown the relationship between the grid size $x_{sparsity}$, and the ratios of dataset sizes. Since we have plenty of data, we set $x_{sparsity} = 20$. This makes our datasets look as they do in 3, and still have a training set of 400 data points. We also divide all our data on 400 to get more manageable sizes for both x and y, preventing overflow. in the end, $x \in [0, 1]$ while $y \in [0, 0.6]$.

3.9 Classification

The data we use for the classification problem is the MNIST dataset [5]. This is a set of images of hand drawn digits with resolution 8x8, and the task here is to classify which number they are meant to represent. In fig 4 we have displayed a random sample from each of the classes, to show how they look.

Whereas it is trivial in the regression problem to see that there should exist a

Table 1: Portion of total data that is divided into the training, validation and testing sets respectively for different values of $x_{sparsity}$.

$x_{sparsity}$	**Training**	**Validation**	**Testing**
1	1	0	0
2	1/4	1/4	1/2
3	1/9	1/9	7/9
4	1/16	1/16	7/8
5	1/25	1/25	23/25
10	1/100	1/100	98/100
20	1/400	1/400	398/400

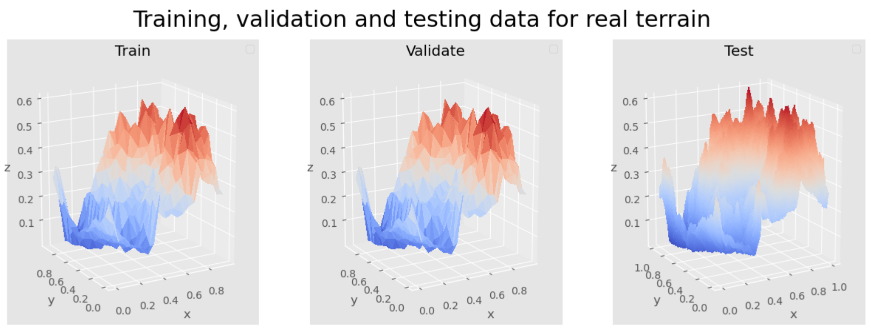


Figure 3: The terrain data, split into training, validation and testing sets.

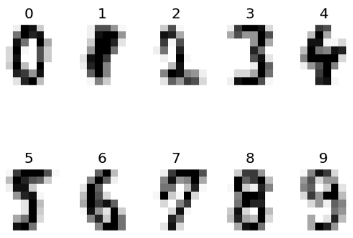


Figure 4: Samples of the different classes

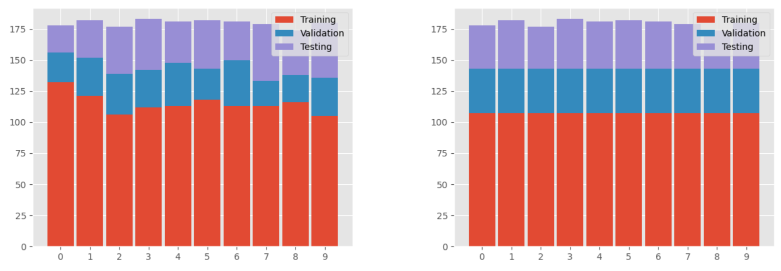


Figure 5: Distribution of different classes between the training, validation and testing sets. The left plot is straight up random, while the second is divided with an equal share of samples from each class in the training and validation sets.

polynomial function that approximates the terrain data pretty well, it is certainly not trivial to see it in this case.

In order to select what should be our training, testing and validation datasets, selecting it randomly is an obvious option. There isn't an easily defined distance in the predictor space that lets us select an unbiased sample without stochastisity, so randomness is wise. However, a completely random selection could easily lead to a skewed distribution of what classes are in which sets. This could again lead to a biased model, that for example guessed the number 0 more often than it should, because an disproportional amount of samples in the training and validation set were 0's. This is shown in fig 5. To combat this bias, we select which samples of each class should be in which set randomly, but do it on a class by class basis. We can therefore ensure that we have a representative amount of samples for each class.

However, since there isn't an equal amount of samples for each class in the dataset, we have to make a choice about how representative we think the dataset is of the real world, and what we actually want our model to do. We have decided that we won't look into distributions of numbers in different real world applications, and want our model to perform as well on each class. This means that if the MNIST set is representative, and there are slightly more 3's than 8's, we still don't want our model to guess 3 slightly more often than 8. To implement this idea, we have as many of each class in the training and validation sets, and let the testing set be biased. This isn't unproblematic, and an option would be to truncate the data so that even the testing set could have as many datapoints from every class, or to resample some of the datapoints to achieve the same goal. We have however not had the time to study the effetcs of this choice.

4 Results

4.1 Testing SGD for regression, and analysis of hyper parameters

To try out our SGD implementation, verify that it works, and explore how it differs from the analytical solutions we looked at in project 1, we begin by using the same problem, data and solution model. Namely, the regression problem, real terrain data, and our familiar linear models. In fig 6, we compare different models with different mini-batch sizes, and see how their errors change over epochs as it is trained. On all of them, we let the maximum degree of polynomials be 8, since that's a hyper parameter our first project demonstrated to work well [4]. We also use training data for training, and validation data for testing. Note that epochs aren't an entirely fair way to compare different mini-batch sizes, as there are fewer mini-batches for large mini-batches, but the step taken is an average of all the gradients in the mini-batch. We have not looked into how much time is spent computing for different mini-batch sizes, but we reason that an epoch with mini-batch size 40 should run faster than one with mini-batch size 10. We believe this because the operations with the mini-batches are vectorised making the operations of finding and applying the gradient not that expensive. The only operation that would indicate otherwise is the matrix multiplication in both the linear and the neural models, which quickly becomes expensive as it's an $\mathcal{O}(n^3)$ operation. Since error per computation is the real metric we want to optimize here, it is difficult to be absolutely sure of how to interpret the comparisons between different mini-batch sizes, but there are some general and rough conclusions this points toward.

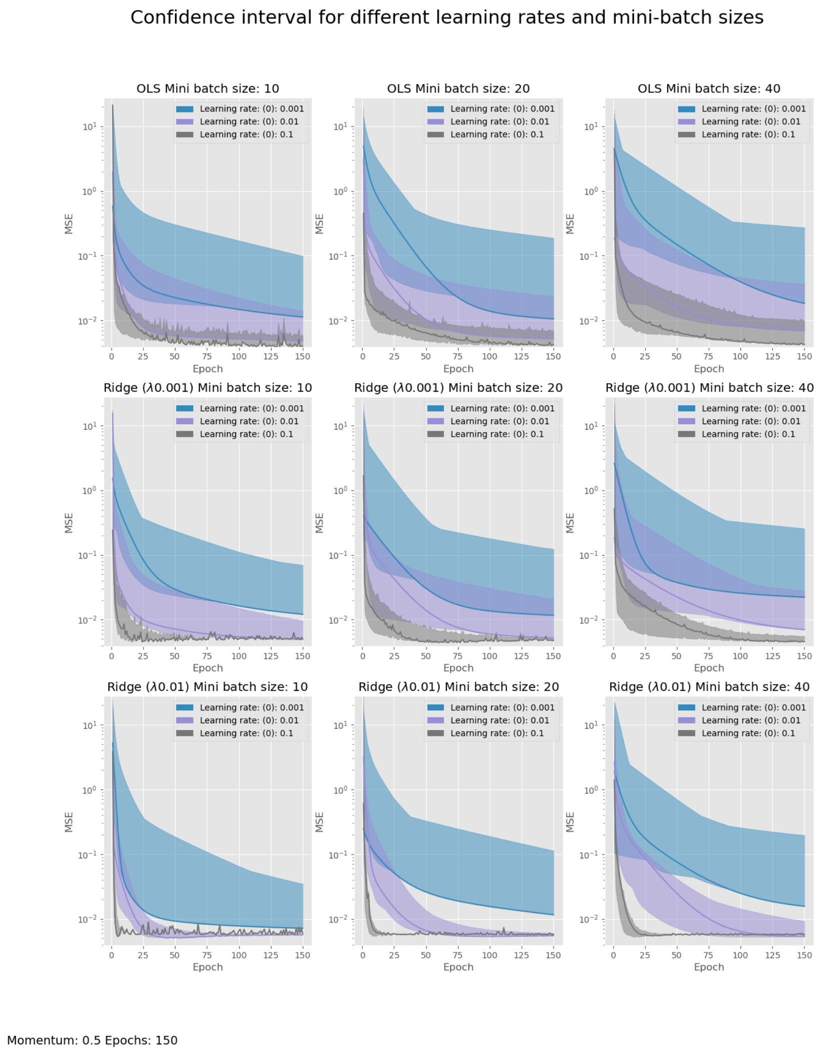
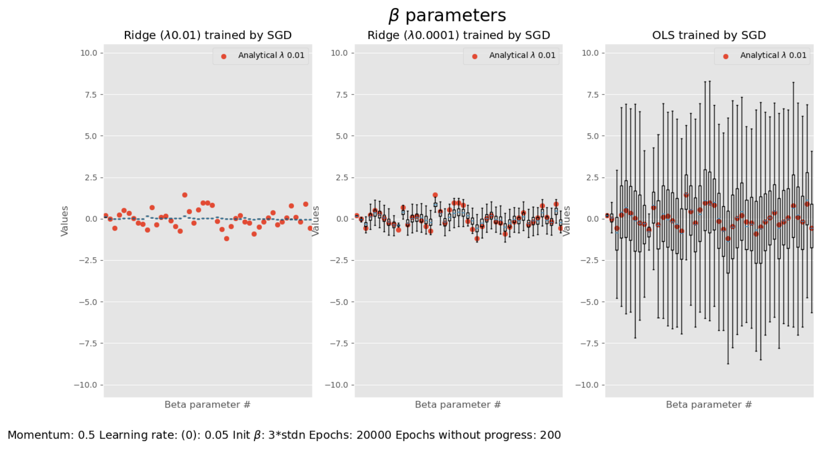


Figure 6: Confidence intervals for errors for different linear models with different mini-batch sizes, while using SGD to train them. The errors are calculated based on predictions on unseen validation data. The slightly transparent filled in areas indicate the empirical confidence intervals retrieved from testing the models with different initial conditions, while the hard line is the model that ended up with the lowest error.

Firstly, the relationship between stability, mini-batch sizes, and learning rates, is as expected. A higher mini-batch size and a lower learning rate leads to less stochastisity and higher stability. This is evident by the plots with low mini-batch sizes and high learning rates where the error oscillate rapidly, and appears as anything but smooth, and opposite for high mini-batch sizes and low learning rates. The dependence on initial conditions however is the opposite. At least for this case where there seems to be one pretty clear set of parameters all the different models convergence on, the unstable models seem to reach it quickly regardless of initial conditions, while as the stable ones take time, and the time varies greatly with initial conditions. It is possible that for problems with a less convex cost, the very stable models might converge at different minima, while the unstable models all converge at the same one. However, this data set cannot confirm this hypothesis.

Another observation from figure 6, is that the models with the initially lowest errors, aren't necessarily the ones that end up doing the best. Looking at the subplot in the lower right corner, there is clearly at least two times where the currently best model is surpassed. This indicates that even though our cost function seems convex, there are certainly areas that are steeper than others. Lastly, these models have analytical solutions, giving the lowest possible MSE. Because all our models converged on parameters with the same error, and did so pretty quickly, our initial assumption was that the parameters they had converged on were the same as the analytically proven best models. However, this was not the case.

When trained on the training set, and tested on the validation set, the models SGD converges on have an average MSE of roughly 0.0057, while as the analytical solutions get an MSE of around 0.0037 [4]. This leaves us with two questions. Firstly, do the SGD trained models converge on the same parameters? Phrased differently; is there an obvious cost minima that SGD leads to, regardless of initial conditions? Secondly, why don't they converge at the same parameters as the analytical model?



Momentum: 0.5 Learning rate: (0): 0.05 Init β: 3*stdn Epochs: 20000 Epochs without progress: 200

Figure 7: Box plot for β parameters of linear models trained on SGD for 40000 epochs. In the left, the small boxes indicate the confidence interval for Ridge models ($\lambda = 0.01$) trained by SGD, and in the right for OLS models trained by SGD. The red dots are the analytical solution for Ridge with $\lambda = 0.01$.

To answer both questions, let us look at the distributions of β parameter values, plotted in fig 7. The left subplot demonstrates clearly, by the boxes being so small, that different initial conditions and even λ values converge on the same solutions. The latter is a bit surprising, but we believe it to happen because our

λ values quickly brings $|\beta|$ down close to 0, before any real fitting can happen. This is a clear indication of too high *lambda* values. At the same time, the plot on the right shows that we can't really get OLS models to converge tightly to anything, so some regularisation is probably good. Something more interesting however, is the median β parameter values of the OLS models trained by SGD. They appear to match up with the analytical solution for Ridge regression very well. From these two plots, it seems as if SGD in itself almost regularises some, at least with the initialisation of β parameters we have used here, which is $\beta_i \sim 3 \cdot \mathbb{N}(0, 1)$. This makes sense, since it is unlikely that there's a straight gradient going from somewhere close to 0, to somewhere very far from it. β stops at a local minimum sometime before that.

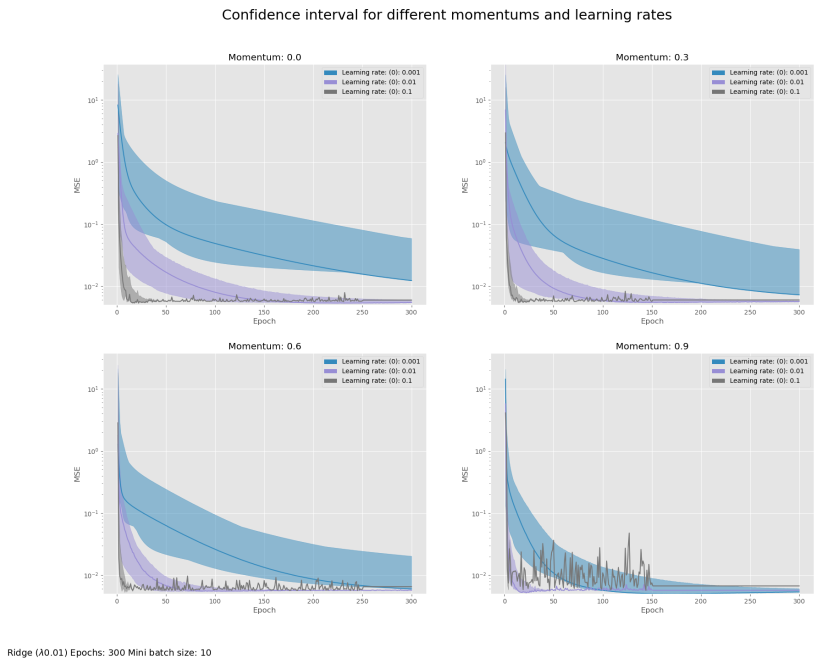


Figure 8: Empirical confidence intervals for validation errors for Ridge model trained by SGD on terrain data, with different momentum paramaters.

Lastly, in this discussion of the results from SGD, let us discuss the momentum hyper parameter, with its effect demonstrated in fig 8. Here, even clearer than in fig 6, it is possible to see momentums impact. Other than the stability aspects already noted, it is interesting to note how the model that in the end performs best, has a high momentum, and low learning rate. It (the blue model in the lower right hand subplot), only does marginally better, and this is of course no general rule, but it does at the very least show how momentum and learning rate aren't just two parameters that do the same thing.

4.2 Using SGD to train a neural network for regression

Neural models have a more involved set of hyper parameters than linear regression techniques, with network architecture and activation functions being the main contributors to the extra complexity. We know that the terrain data isn't very complicated since we in project 1 have shown that the fit linear models did was pretty good, so this means our network probably doesn't benefit from being very big. We therefore choose to narrow our search down to models with maximum two layers, and where each layer has no more than eight neurons. When it comes to activation functions, we want to try all these models with sigmoid, ReLu and leaky ReLu, three popular alternatives, in the hidden layer. For the output layer, it makes sense to have the identity function as activation function (if you can even call it an activation function), since we want to regress to data that could be any real number. Strictly speaking, our scaled terrain data lies between 0 and 1, so we could use something like a sigmoid function for the output layer as well, but doing that would be making assumptions about our terrain data that we couldn't make in general. The tuning results can be seen in fig 9

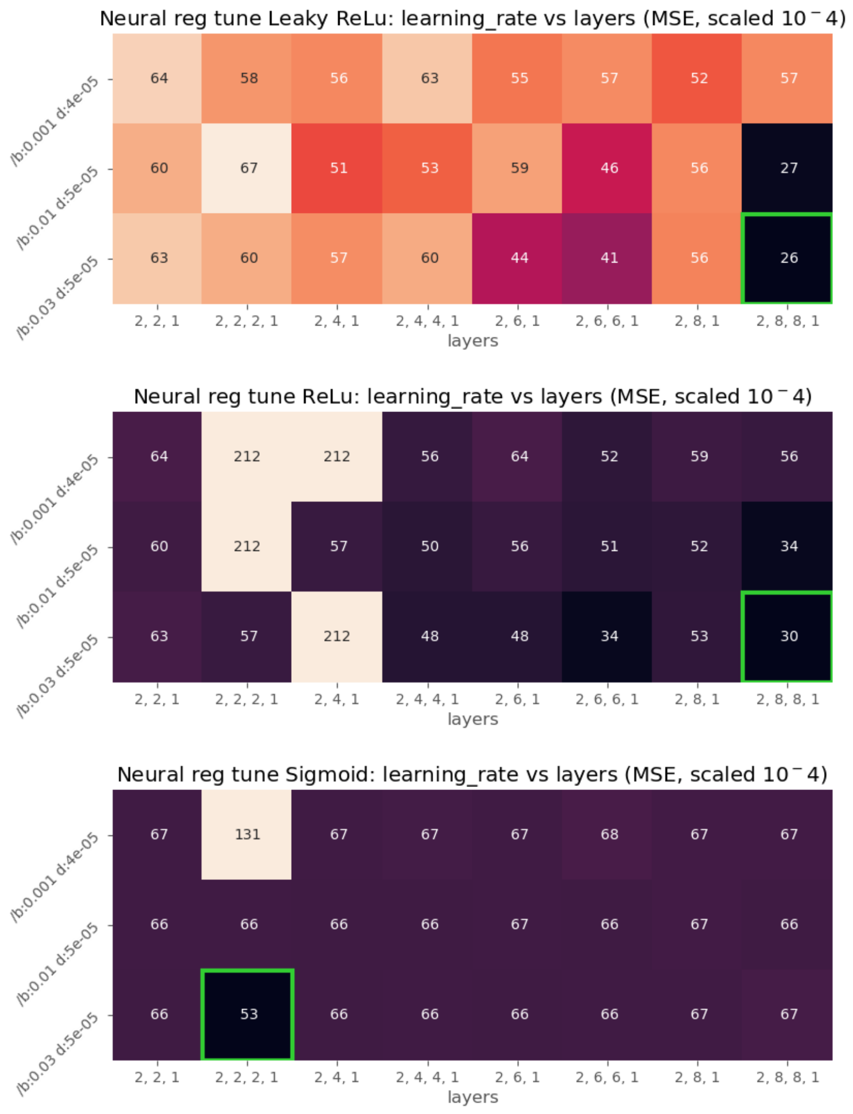


Figure 9: Heatmaps made by the tuning class, for validation errors for different network architectures and learning rates, with either the ReLu, the leaky ReLu or the sigmoid activation function in the hidden layer. Learning rate is on the y-axis.

Looking at the tuning results, it is, as discussed before, very possible that the parameter variance is so large for training that the calculated errors associated with each set of hyper parameters in fig 9 is more determined by chance than by these hyper parameters. However, it is unlikely that at least the good results for ReLu and leaky ReLu for models of size 2x8x8x1 are entirely random. Running the best models once again, this time alongside the best ridge model, gives us fig 10. Here, we can see that both the leaky and the unleaky variants of ReLu perform best, and since this is a new run, we can be more sure than we were before that they do well. Ridge makes it to a third in performance in the end, but sigmoid converges faster. What all the neural models share, is a learning rate and momentum combination that seems to be too high, evident in the uneven lines. If we had more time, we would have made them decay faster, and experiment more with combinations of momentum and learning rate.

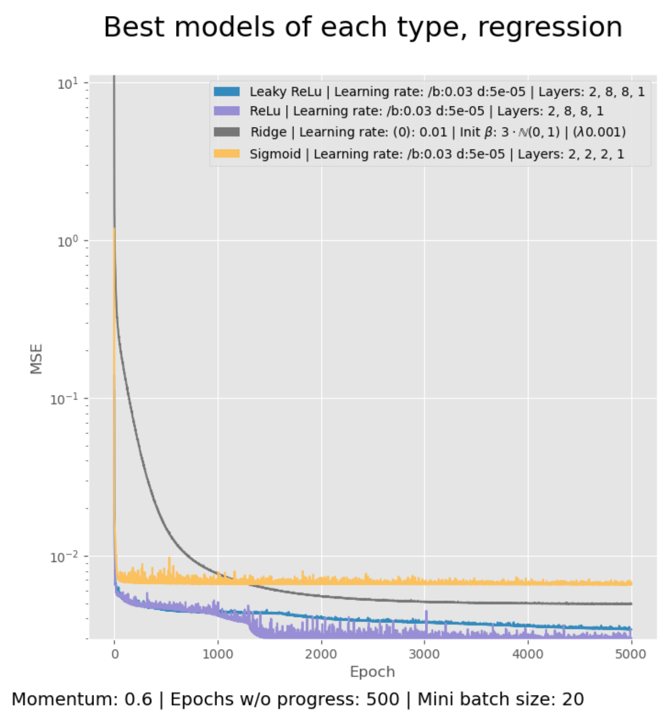


Figure 10: Different network architectures attempting to fit to the terrain data.

Even though ReLu and leaky ReLu turned out to be the best activation functions for the hidden layers, they also had the most complicated architectures. This shows that for this data set, sigmoid can't take advantage of as big layers as ReLu, which isn't a general rule. This demonstrates the power of tuning well. In some cases, we also won't want our model to be more complicated than it has to, and can happily sacrifice some error for explainability or less computations. Depending on what we prioritise, it is therefore also possible to argue that sigmoid was the best activation function, and also that ridge was the best model. Now that we've decided which models we believe are best, let's see how well they do on completely unseen testing data:

Table 2: Mean squared error and R^2 score of our different models on the testing terrain data.

Model name	MSE	R^2
Ridge	0.00636	0.691
ReLu	0.00277	0.866
leaky ReLu	0.00323	0.843
Sigmoid	0.00479	0.768

The results in table 2 look decent. Firstly, it shows that our validation data does give a pretty good estimate of how well our models does. The only aspect that has really changed from validation to testing is that the neural model with sigmoid activation function actually does better than the linear ridge model now. We don't have enough data to be sure if this is actually the case, but it seems to indicate that the simple neural model generalised better than the linear model did. The R^2 score doesn't look great, but we also don't have a good intuition for what a good score for a problem like this is.

To understand more how the models perform better, we have plotted the model predictions on test data in fig 11. Here, it is obvious that the neural models aren't complex enough. The model with the sigmoid activation function is just a plane, and the ReLu and leaky ReLu models are too jagged and coarse. This also points in the direction of too high learning rates, because it seems as if the jagged lines doesn't even align that well with the real data - the complexity it has could be used in a better way. Later attempts at fitting to data of this kind should include more complex models in the tuning space.

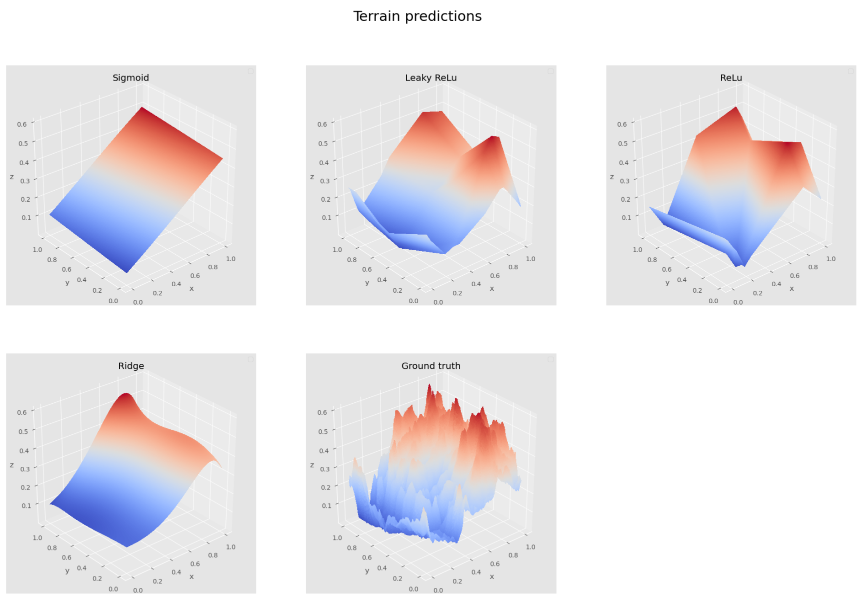


Figure 11: The terrain predicted on test data by the best models. This is a worse fit than we expected.

4.3 Classification

For classification we compared our neural networks' ability to recognize the handwritten digits to a Keras implementation and logistic regression. Tuning of the models worked similarly to regression. We compared multiple network architectures combined with different learning rates to find the optimal model. In figure 12 where the sigmoid activation function is used in every layer we see that model with architecture 64x32x10 and learning rate given by $\gamma_0 = 0.003$ and $decay = 2.5e{-}5$ has the best prediction accuracy on the validation set. However, if we find the optimal models looking at the three learning rates separately we see that the architecture 64x32x10 actually performs worst for the first and third learning rate. A weakness in our tuning is that the initial parameters for a specific architecture changes for the different learning rates. This creates an unstability that it is hard to account for without multiple runs of each model for each set of hyper parameters and averaging the error rates as discussed in the analysis of the regression results.

As predicted there is variance across runs. In a larger second run of the tuning of the neural network on the classification the results changed as predicted and the optimal model was 64x16x10 with $\gamma_0 = 0.005$ and $decay = 0.001$. However, it also achieved an error of 2.78% which might suggest an upper limit for the performance of our neural network across hyper parameters on this data set. This suggests that initial conditions are almost as important as hyper parameters, as long as the hyper parameters stay within a reasonable space.

Apart from two outliers the worst performing model of the second run had a validation error of 9.44%, slightly worse than the first run in fig 12. This suggests that the variation in performance isn't that large after all, but we don't have enough data to be sure.

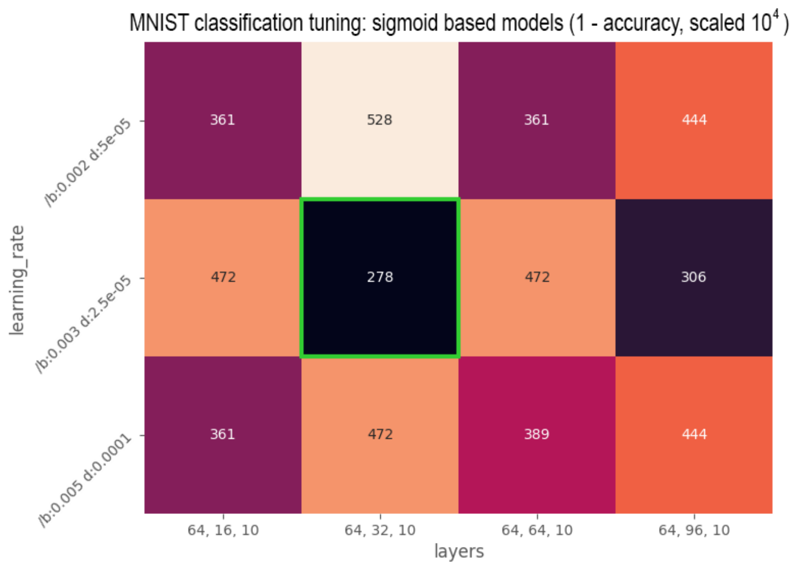


Figure 12: Errors of neural networks with sigmoid activation in both layers for different learning rates and architectures on the MNIST classification problem. The best performing model is marked with the green box and has an error of 2.78%.

Neural networks implemented with ReLu activation in the hidden layer performed better than sigmoid activation as seen in figure 13. The output layer still uses sigmoid but we see that the effect of ReLu is that all but three out of twelve models outperform the second best model in figure 12. The best two models with ReLu also improve upon the previous best by 0.84%. The first model has 64x32x10 structure and the second 64x64x10. They both have an accuracy of 98.6% on the validation set and the first achieves 97.1% on the test set and the second achieves 98.0%. The confusion matrix for the second model is shown in figure 3. Notice that the amount of samples in each class is not equal due to the equal distribution of digits in the validation and test set visualised in figure 5.

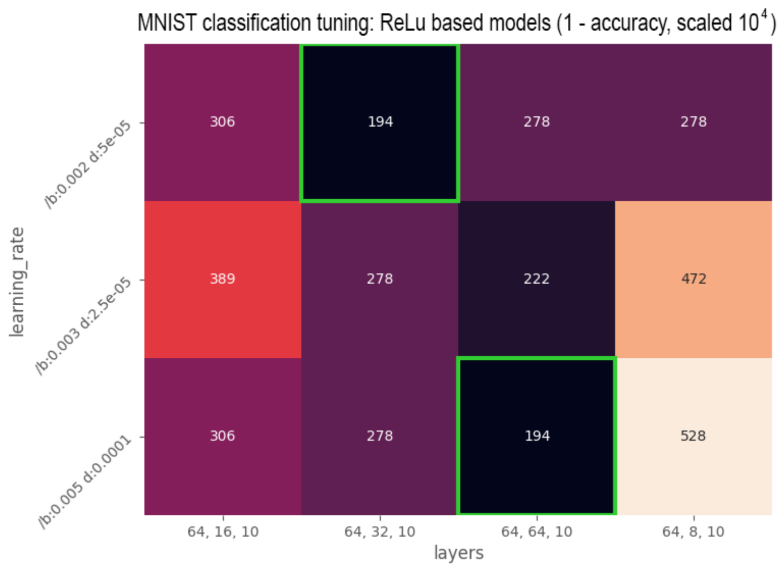


Figure 13: With ReLu activation in the hidden layer the neural network outperforms the fully sigmoid activated model.

Table 3: Confusion matrix for the best performing model from figure 13 on the MNIST data set with 98.0% accuracy on the test set. Predicted class on the x-axis, and actual class on the y-axis.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	34	0	0	0	1	0	0	0	0	0
1	0	38	0	0	0	0	0	0	1	0
2	0	0	34	0	0	0	0	0	0	0
3	0	0	0	40	0	0	0	0	0	0
4	0	0	0	0	38	0	0	0	0	0
5	0	1	0	0	0	37	0	0	0	1
6	0	0	0	0	0	0	38	0	0	0
7	0	0	0	0	0	0	0	35	1	1
8	0	2	0	0	0	0	0	0	29	0
9	0	0	0	0	0	1	0	0	0	20

Logistic regression was able to use our implementation of the softmax function however it did not measure up to the neural networks. It did seem as the softmax function was able to converge faster than ReLu as seen in figure 14. However, logistic regression is able to predict with high accuracy as we saw with our scikit learn implementation which achieved 97.7% on the test data.

The Keras benchmark did on average marginally outperform our implementation for accuracy, and substantially for speed and stability. A Keras implementation with 64x32x32x10 i.e two hidden layers and an output layer achieved 98.0% accuracy on the test set and 99.9% on the training set. These numbers are averages over 10 runs which took significantly less time to run than our tune function, which shows to the gap in efficiency between the two models. However, this might not be solely due to pure computational efficiency as the Keras models converge after just 3-6 epochs and they had only 20 epochs total. Our neural network needed about 3000 epochs as seen in figure 14 before it converged.

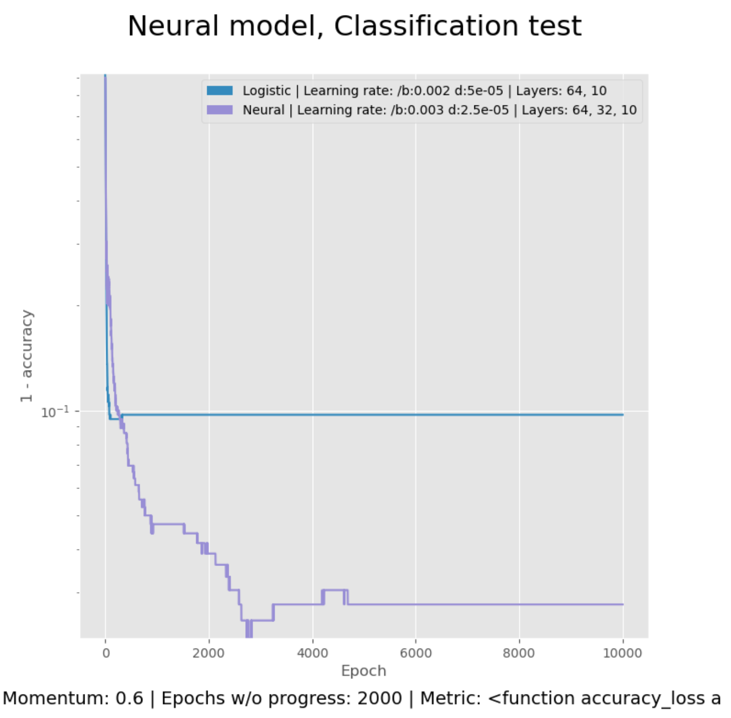


Figure 14: Performance of the tuned neural and logistic models on the MNIST dataset. The metric is $1 - accuracy$, meaning that the logistic model end up with around 10% in validation error, and the neural model around 3%, which is roughly the same as we saw during tuning. The neural model is more complex, and as this plot shows, this added complexity pays off with a higher best accuracy. However, it also makes overfitting a more real threat.

Softmax and sigmoid on output layer

When doing classification, softmax is often the preferred activation function. It works like sigmoid and squishes everything in to values between 0 and 1, but it also normalises everything, making sure the sum of all the numbers is 1. This makes it so that $\tilde{y}$ can be interpreted as a vector of probabilities for the input being in the different classes. It also works very nicely with cross-entropy. However, we had some problems when running our implementation, leading to the models never performing better than chance. We have looked into it being caused by us deriving the wrong expressions, numerical instability, or bugs in our code, but have still not been able to fix it. This lead us to eventually use sigmoid as the activation function in the output layer too, it being very similar, just without the normalisation.

Even though we haven't been able to fix the problem, we know exactly why the models perform poorly, and that is that the output layer explodes, as shown in fig 15.

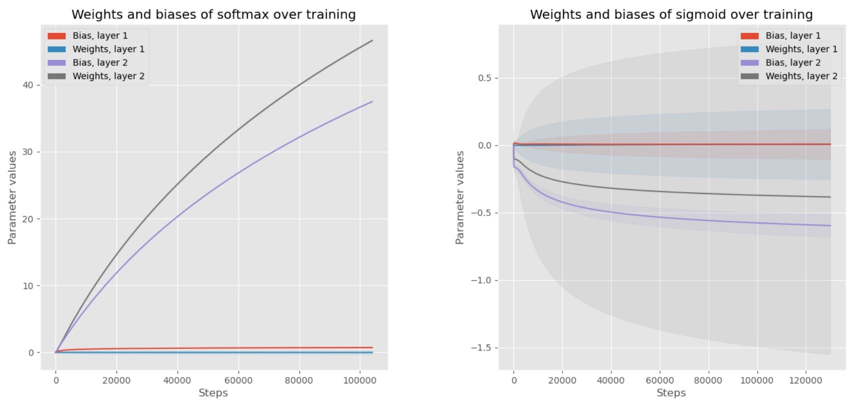


Figure 15: Distributions of weights and biases during training, for networks with softmax and sigmoid as activation functions for the final layer in a 64x32x10 network that classifies on the MNIST data set. The hard lines are the mean values of the 1-norms of the weight-matrix or the bias-vector, and the translucent area around it marks 1 standard deviation in each direction, for parameter values. Note that the y-axis don't have the same scale.

5 Conclusions

The aim of this project was to study classification and regression problems and to gain a deeper understanding of neural networks by developing our own feed forward neural network. We developed a well functioning network that is capable of solving both classification and regression problems.

On the MNIST data set our best performing model achieved 98.0% on the test set equalling the Keras benchmark which also reached 98%. However, our implementation converges more slowly and usually needs 100-fold more epochs than Keras. On the other hand, Nielsen writes that slow models are necessary for understanding what goes on under the hood [7]. Optimized models like the Keras Sequential model are difficult to understand if one where to read through the code behind the function calls.

Most work done on the MNIST data set uses the full set of 60,000 images with 784 pixels each from Lecun. However, we used the reduced set of 1789 64-pixel images available in scikit learn. Assuming that it is harder to classify a high percentage of the 10,000 test images in MNIST correctly we can assume that our model performs well below the best FFNN which has an error rate of 0.35% [5] .

The methods we used to select the ultimate models looked at mainly two hyper parameters, the learning rate and the architecture. An alternative method that might yield better results is deciding on an architecture and then rather optimize learning with the learning rate and momentum.

On our regression problem, where we tried to fit both linear and neural models to terrain data, our models performed less well. The models with the ReLu class of activation functions in their hidden layers seemed to be able to take advantage of the more complex network architectures we tuned it on, but their fits only got up to a test R^2 score of roughly 0.86 on test data. A much higher degree of complexity would probably have worked better, and further studies should be done to look into this.

Another interesting conclusion to come out from our work with the regression problem, is the inherently regularising nature of stochastic gradient descent. On the linear models, our β parameters were harder regularised by the λ hyper parameter than the analytical solutions were. This should be tested with other model classes, and for different data sets, to see if this is a general trend or just coincidence. If the former is true, we also offer an explanation, with the regularisation quickly driving the parameters down before any other fitting really has the time to happen, so that β stops at the cost minima closest to $\mathbf{0}$.

Moving forward it is interesting to study the effect of utilising convolutional neural networks on image classification problems. Our implementation does not take the relative position of the pixels into account, which is an important factor and alpha-omega in how humans recognize digits. The best performing classification models on the MNIST set are convolutional neural networks which boast of superhuman digit recognition abilities [5] [2].

6 Referencing

References

[1] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. “Optimization Methods for Large-Scale Machine Learning”. In: *SIAM Review* 60 (June 2016). DOI: 10.1137/16M1080173.

[2] Dan C. Ciresan, Ueli Meier, and Jürgen Schmidhuber. “Multi-column Deep Neural Networks for Image Classification”. In: *CoRR* abs/1202.2745 (2012). arXiv: 1202.2745. URL: http://arxiv.org/abs/1202.2745.

[3] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning. Data Mining, Inference, and Prediction, Second Edition.* Springer, 2009.

[4] Didrik Sten Ingebrigtsen. *Report, project 1 FYS-STK3155.* 2020.

[5] Yann LeCun and Corinna Cortes. “MNIST handwritten digit database”. In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[6] Ashutosh Nayak. *Cross-entropy: From an Information theory point of view.* June 2019. URL: https://towardsdatascience.com/cross-entropy-from-an-information-theory-point-of-view-456b34fd939d (visited on 11/12/2020).

[7] Micheal A. Nielsen. *Neural Networks and Deep Learning.* Determination Press, 2015.

[8] Severin Schirmer. *Report, project 1 FYS-STK3155.* 2020.

[9] US Geological Services. *Earthexplorer.* URL: https://earthexplorer.usgs.gov.

[10] Chi-Feng Wang. *The Vanishing Gradient Problem.* Jan. 2019. URL: https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484 (visited on 11/11/2020).