Classification and Regression, from linear and logistic regression to neural networks

---

by
Pascal Sado
November 2020

Abstract

We apply Stochastic Gradient Descent to the same Franke Function as in the first report, to find ideal regression parameters. We evaluate the performance based on the polynomial degree and learning rate. We find that although the mean squared error is slightly lower than for OLS in the previous report, this comes at the cost of having to choose the learning rate carefully as well as long computation times. We find however that this method is more robust towards higher polynomial degrees than OLS. In our case Ridge Regression therefore cannot lead to significant improvements, because the parameters were already well regularized.
Evaluating the neural network, we find that the performance of the neural network strongly depends on the complexity of the network, its number of layers and neurons. We also test the influence of different activation functions.
For the classification we find that it works well with the mean squared error as a cost function but less good with cross entropy.

Contents

1 Introduction

Depending on the problem, analytical solutions cannot always be found or may be too complex to yield a stable solution. Ordinary Least Squares and Ridge Regression are suitable analytical solvers for linear problems, Lasso extends this towards numerical solutions. More complex problems, for example classification problems, are usually not linear problems and the previously employed methods will not work any more.
If for example features for the classes can be identified, classification can still be done analytically using gaussian classifiers. If there are no obvious underlying features or the correlation between the features and their classes is unknown or too complex, neural networks can be employed to extract features from a given set of data.
In a neural network, the model is replaced by one or several layers of neurons. Each neuron performs a simple non-linear mathematical operation, where the input depends on the output of the neurons of the previous layers. Combining several layers with several neurons each makes it possible to model complex and non-linear relationships between the input and output data.
Similar to polynomials, neural networks can be made to be trivially simple or arbitrarily complex.
First, I will give an introduction into the theory behind neural networks. I will focus only on the parts relevant to the project. The theory behind neural networks can be expanded upon much further than that.
At first I will employ gradient descend methods to our data from the first project and find optimal regression parameters. This is to show that the method works and build a comparison between everything. Then we will build a neural network architecture from scratch, including variable numbers of layers and arbitrary numbers of neurons per layer. The network will be able to handle different activation functions in the layers and different cost functions, depending on the problem it has to solve.
The neural network will be used to solve the same problem again, this time based purely on the neurons without finding regression parameters. I will then use Tensorflow with Keras to compare the performance of my own neural network implementation to that of a prebuilt module.
At last I will use my own network to classify digital images of digits into categories corresponding to their numerical values.
An overview over the results will be given in 5.

2 Basics

To find a set of parameters for any given problems, these parameters need to be optimized such that the feature vector applied to the parameters converges towards the solution for any given vector of features under one set of parameters. We will later define different cost functions, that measure how good the parameters fit the problem. For now our knowledge of ordinary least squares is sufficient. In this case, the squared differences between the solution and feature vectors applied to the parameters is to be minimized.

2.1 Gradient Descent Methods

2.1.1 Gradient Descent

The simplest way to optimise parameters is by using gradient descend methods. To find the minimum of a function you follow the function along its negative gradient until a minimum is reached.

$$x_{t+1} = x_t - \eta * \nabla_x f(x_t) \tag{1}$$

Here we evaluate the gradient of the function f at our current position x_t and move a step of size η in the direction if descent. This is our new position, where the next gradient is evaluated. The procedure repeats for a given amount of steps or until the gradient has vanished.
A simple example is finding the bottom of a hill. You descend the side of the hill by following its slope downwards until the ground levels. One of the problems of gradient descend can already become apparent here:
Is this ground level the lowest level? Minimising the cost function in our context means that we want to find the lowest possible value, namely the global minimum, not a local minimum. If the cost function is convex, it can be shown that the minimum has to be a local minimum.[1] In the case for a linear problem this is the case, for complex neural networks this will not be true.
Another problem is also if we take too large steps. We might overshoot the minimum, land on the other side of the slope and bounce our way outwards, away from the minimum. On the other hand, choosing very small steps will lead to convergence, but may take a very long time. The stepsize has therefore always to be chosen carefully.
Often we also move in higher-dimensional space. We have to optimize not for one parameter or dimension but for tens, hundreds of even thousands of parameters. For this the amount of samples has to increase and we might see tens of thousands or millions of datapoints. Evaluating the slope of the underlying function

towards the parameters is of course possible and would lead to a solution but is also incredibly expensive computationally.

2.1.2 Stochastic Gradient Descend

Instead of therefore calculating the gradient over the whole set of data, we divide our samples into a fixed amount of mini-batches. We calculate the gradient for each of these batches and update the parameters as if we had calculated the full gradient over everything. We then cycle through all of the batches one at a time and update accordingly. After every batch and every sample has been processed exactly once, we have completed one epoch. Both methods are repeated over several epochs, the computer therefore sees the same data multiple times.
The second method is called **Stochastic** Gradient Descend, because we assume that by choosing the samples and divisions into batches randomly but from the same distribution, the average gradient is the same as if every sample was processed at once. We also solve the problem of dropping into a local minimum, because due to the randomness of SGD, we might be accidentally kicked out of the minimum and descend further towards the global solution.

2.1.3 Momentum

To increase learning, it is possible for the optimiser to remember its previous direction.

$$\begin{aligned} v_t &= \gamma * v_{t-1} + \eta \nabla_x f(x_t) \\ x_{t+1} &= x_t - v_t \end{aligned} \tag{2}$$

Here we do not update the parameter right away, but calculate a velocity with which the new position in parameter space is calculated. The parameter $0 \leq \gamma \leq 1$ determines the momentum, i.e. how much of the velocity stays over the iterations and η is again the stepwidth of the gradient. For $\gamma = 0$ this reduces to standard SGD.
This so called "momentum" lets the optimizer follow larger differences more easily because random perturbations are less influential.
This method can be expanded upon by introducing another term which physically would correspond to viscous drag.[2]

2.1.4 Adaptive Learning Rate

Another way of finding the global minimum is by updating the learning rate adaptively. At the beginning it is likely to be far away from the global minimum. The learning rate can therefore be high. Fur further iterations, when moving towards

the minimum, a high learning rate could lead to overshoot and the learning rate should therefore be decreased.
The simplest way would be to just decrease the learning rate with increasing batch and epoch number.

$$\eta_t = \frac{\eta_0}{\text{current epoch} * \#\text{ of batches per epoch} + \text{current batch count}} \tag{3}$$

In this example we start with a learning rate η_0 that decreases for every batch that is analysed. This works well if convergence towards a minimum takes long steps at the beginning and smaller steps towards the end. This may however also mean that the learning rate could drop to a small number before a minimum is reached. Increasing the initial learning rate could mean that the calculation explodes in the beginning without ever having a change to converge.

2.1.5 RMSProp

In RMSProp this is taken into account and the learning rate changes adaptively based on the gradient.

$$\begin{aligned} g_t &= \nabla_x f(x_t) \\ s_t &= \beta s_{t-1} + (1-\beta) g_t^2 \\ x_{t+1} &= x_t - \eta_0 \frac{g_t}{\sqrt{s_t + \epsilon}} \end{aligned} \tag{4}$$

Similar to the velocity we introduce a regularization parameter s_t that has a memory introduced by $0 \leq \beta \leq 1$ over the last iterations, ϵ is a small number to avoid division by zero.
For RMSProp the learning rate is reduced towards directions of small g_t and mostly unchanged in other dimensions. This allows for quick convergence in dimensions where no convergence has yet been achieved while allowing for smaller steps in dimensions that are close to finding a solution.

2.2 Neural Networks

Neural Networks are a way for a computerized system to learn and classify independently. Instead of having a fixed problem (e.g. a polynomial of fixed grade with a fixed and small number of parameters) the neural network is given a less constrained environment with more parameters. By optimizing these parameters it can solve previously difficult to solve problems and detect correlations without us having to know the exact relation behind those properties.

2.2.1 Feed Forward Neural Network

A feed forward neural network consists of several layers of neurons stacked behind each other. The first layer takes the feature data as input and computes an output in the last layer that is supposed to equal the data matching the feature data. In this kind of neural network data only flows into one direction and it is the simplest of its kind. Because only FFNNs are used in the project, only they will be explained here.

2.2.2 Dense Layer

We also only used the simplest form of a layer, the dense layer. Each layer in a neural network consists of a fixed amount of neurons. Those neurons are connected between the layers but not to each other in a layer. How the neurons are connected determines the type of the layer. In a dense layer, every neuron of a layer is connected to every neuron in the next layer.

$$x_{li} = \sum_{j}^{N} w_{lji} * z_{kj} + b_{li} \tag{5}$$

Here, x_{li} is the i-th neuron in the l-th layer. It's input is equal to the sum over every neuron's output z_{kj} in the previous layer k multiplied by a weight w_{lji} which is unique for every connection and an additional bias b_{li}. This can also be written in matrix notation

$$\vec{x}_l = \underline{w}_l \vec{z}_k + \vec{b}_l \tag{6}$$

Here, $\vec{x}_l$ is the vector representing the inputs for the l-th layer, $\underline{w}_l$ represents the weights towards the l-th layer, $\vec{z}_k$ is the output of the previous layer k and $\vec{b}_l$ is the bias vector added to the neurons' inputs.
This type of layer is sufficient for simple problems but often layers have to be supported e.g. by convolutional layers which excel in image processing.

2.2.3 Activation Functions

If every layer was left like this, in a FFNN every layer could be described by a matrix multiplication and the whole network could be described by a composition of linear functions. This way we could only describe linear problems and would therefore be very limited in the types of problems to solve.
To work around this, every neuron is associated with an activation function. This function takes the previously calculated input and calculates the neurons output, which can then be used to calculated the next neurons' inputs. It is important that the activation function is not purely linear but it also has to be differentiable

for reasons that become apparent later. Because the output of this function (and later its derivative) has to be computed very often, it is also useful to choose a computationally cheap function.

2.2.3.1 Sigmoid Function

Looking at nature and real neurons, one popular activation function is the sigmoid function:

$$\sigma(x) = \frac{e^x}{1+e^x} = \frac{1}{1+e^{-x}} \tag{7}$$

This function is zero for very negative values of x, one for very positive values and almost linear around zero. Its derivative is also easy to compute:

$$\frac{d\sigma(x)}{dx} = \frac{e^x}{(1+e^x)^2} = \sigma(x)(1-\sigma(x)) = \sigma(x)\sigma(-x) \tag{8}$$

2.2.3.2 ReLU, ELU and Leaky RELU

Rectified Linear Unit, Exponential Linear Unit and Leaky Rectified Linear Unit all share the same basic property. If the neuron input value is larger than zero, they pass the value on to the output without change. They all achieve non-linearity by handling the negative part differently.
In the case for ReLU, the negative part is zero:

$$\begin{aligned} \text{RELU}(x) &= \begin{cases} x, & x >= 0 \\ 0, & x < 0 \end{cases} \\ \frac{d\text{RELU}(x)}{dx} &= \begin{cases} 1, & x >= 0 \\ 0, & x < 0 \end{cases} \end{aligned} \tag{9}$$

Leaky Relu returns the value of the input for values smaller than zero but multiplied with a small constant α:

$$\begin{aligned} \text{Leaky RELU}(x) &= \begin{cases} x, & x >= 0 \\ \alpha * x, & x < 0 \end{cases} \\ \frac{d\text{Leaky RELU}(x)}{dx} &= \begin{cases} 1, & x >= 0 \\ \alpha, & x < 0 \end{cases} \end{aligned} \tag{10}$$

At last, ELU is a middle ground between the sigmoid function and RELU. It treats the negative part with an exponentially decaying function.

$$\begin{aligned} \text{Leaky RELU}(x) &= \begin{cases} x, & x >= 0 \\ \alpha * (e^x - 1), & x < 0 \end{cases} \\ \frac{d\text{Leaky RELU}(x)}{dx} &= \begin{cases} 1, & x >= 0 \\ \alpha * e^x, & x < 0 \end{cases} \end{aligned} \tag{11}$$

Here the negative part goes to wards one for large negative values with the gradient vanishing.

2.3 Optimizing

It this point we have all the tools to build a simple neural network than can successfully complete a forward pass through the network. The network is however not able to "learn" yet. Since the weights and biases would be initialised in some way but not changed afterwards, every input would produce the same output every time.
To teach the neural network, we need to measure how wrong its prediction was, for how much of the error each layer, its nodes with weights and biases are responsible for and then change the weights and biases accordingly. Because we start with the last layer and move backwards through the neural network, this process is called backpropagation.

2.3.1 Cost Functions

The cost function measures how "wrong" the prediction the neural network made was. This cost function is what we later want to minimise and is the reason why we looked into gradient descend methods earlier. For different problems different cost functions might be useful. We will also introduce another activation function which is in general only useful for the output layer.

2.3.1.1 Mean squared error

The mean squared error is already known from ordinary least squares regression. It measures the squared differences between the predicted output and the true output.

$$R_{mse} = \frac{1}{2} \sum_{i}^{N} (y_i - \hat{y}_i)^2 \tag{12}$$

Here and in the following, the true value that is to be predicted is denoted by y and the value predicted by the neural network by $\hat{y}$. Depending on the preference of the applicator one might find the factor 1/2 omitted or replaced with 1/N or a combination of both. As long as the cost function is not switched in the calculation and the gradient calculated accordingly, the only difference is a constant factor. Because we are not looking for the actual value of the minimum just its position, we also do not care about this factor.
The gradient of the cost function is

$$\frac{dR_{mse}}{dy} = y - \hat{y} \tag{13}$$

Because y and $\hat{y}$ are vectors with one element for each output, the gradient also has N components. If $y = \hat{y}$ then $R_{mse} = \frac{dR_{mse}}{dy} = 0$ and we have found the optimum condition.

2.3.1.2 Ridge Regression

Similar to ridge regression, which still had an anlytical solution, we can define our cost function and derivative accordingly. We again take another scaling parameter λ that reduces the size of our parameters and prohibits growth.

$$R_{ridge} = \frac{1}{2}\sum_{i}^{N}(y_i - \hat{y}_i)^2 + \lambda * \sum \beta^2 \tag{14}$$

$$\frac{dR_{ridge}}{dy} = y - \hat{y} + \lambda * \beta \tag{15}$$

The cost function is expanded by another parameter β which is equal to the sum of the weights and bias that influences the input of a neuron. The cost function therefore tries to not only minimise the error but also minimises the parameters depending on the weight λ. The gradient is again a vector of the size of the number of neurons.

2.3.1.3 Logistic Regression & Cross Entropy

The mean squared error is useful when predicting continuous numerical values. If we were to predict probabilities belonging to discrete classes we will use a different activation function for the last layer and calculate the cost function differently.
In the simplest case we want to make a binary prediction, i.e. our output is in

either of two classes 0 or 1. As an output we can use the sigmoid function.

$$\begin{aligned} p(y=1|x) &= \sigma(x) = \frac{1}{1+e^{-x}} \\ p(y=0|x) &= 1 - p(y=1|x) = \frac{1}{1+e^{x}} = \sigma(-x) \end{aligned} \tag{16}$$

Here y represents the output class which is either 1 or 0 and x is the input vector into the last layer. The probability of finding the output then depends on the input vector which itself depends on the parameters and values of the output functions of the previous layers. For large negative values the function returns 0, for large positive values it returns 1. It also fulfils $\sigma(-x) = 1 - \sigma(x)$ so it is a useful discriminator between two classes where the probability of one class is 1 minus the probability of the other class occurring. We then want to maximise the likelihood of finding the correct probability over many y_i which is the product of finding the correct probability for each y_i

$$P = \prod_{i}^{N} p(y_i = 1|x)_i^y * p(y_i = 0|x)^{1-y_i} \tag{17}$$

To derive the cost function we first take the logarithm before differentiating. Because the logarithm is a monotonically increasing function, applying it to the likelihood function does not change the position of the extremum. Because we want to find the maximum of the probability, we also multiply by -1 to instead find the minimum of the cost function.

$$\begin{aligned} R_P &= -\sum_{i}^{N} \left(y_i \log(p(y_i = 1|x)) + (1 - y_i) * \log(p(y_i = 0|x))\right) \\ &= \sum_{i}^{N} (y_i * \log(1 + e^{-x}) + (1 - y_i) * \log(1 + e^{x})) \\ &= \sum_{i}^{N} (y_i * x - \log(e^{x} + 1)) \end{aligned} \tag{18}$$

We can also differentiate with respect to the input giving another vector with entries for every output value:

$$\frac{dR_P}{dx} = y_i - \frac{e^x}{e^x + 1} = y_i - \sigma(x) \tag{19}$$

2.3.1.4 Softmax

This type of classification may be useful when classifying between just two classes, e.g. "cat" and "dog", "sick" and "not sick". If we want to add another class (e.g.

"horse") we have to expand our output. This is called the softmax function:

$$P(c_j|x)_{softmax} = \frac{e^{x_j}}{\sum e^{x_i}} \tag{20}$$

Here, $c_j \in C$ is the probability that the result obtained based on x represents the class c_j. Calculating this for every class, instead of having a single output neuron giving 0 ("cat") or 1 ("dog") we would now have for example three neurons, each with an output between 0 and 1, summing to 1, where the first neuron gives the probability for cat, the second for dog and the third for horse. The output for different classes c_j depends on the observations x_j. This function can again be differentiated towards our input:

$$\frac{dP_{softmax}}{dx_j} = x_j \frac{\sum e^{x_j} - x_j}{\sum e^{x_i}} \tag{21}$$

We can then expand the cross-entropy cost function from two classes to several class:

$$R_{ce,multiclass} = -\frac{1}{N} \sum_{i}^{N} y_i * \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \tag{22}$$

With the derivative:

$$\frac{dR_{ce,multiclass}}{dx_j} = \frac{y_j - \hat{y}_j}{N\hat{y}_j(1 - \hat{y}_j)} \tag{23}$$

From the cross entropy term we can see the following things: Because our class variable y_j either takes only values of 0 or 1 (true or false), we have two terms in the sum. The first term is important if our class variable takes the value 1. In this case, the predicted value is supposed to converge to 1 as well to indicate the probability of this class. The logarithm if $\hat{y}_i$ subsequently vanishes. The second term becomes zero, but is important for the other predictions, where the class variable is equal to 0. Here we want the predictor to go to zero as well, again resulting in the logarithm to vanish. Depending on how the neural network is initialized, we might see predictors being 0 or 1, if the class variable is 1 or 0 respectively. This would blow up the logarithm, which is why I add a small value $\epsilon = 10^{-15}$ to the predictor.
For the derivative we see that if predictor and class variable are equal, the derivative vanishes. I again add a small ϵ to avoid division by zero. If predictor and class variable are not equal, the derivative scales with $\frac{1}{1-\hat{y}_i}$ for $y_i = 0$ and $\frac{1}{\hat{y}_i}$ for $y_i = 1$. In both cases the derivative goes towards infinity for the predictor being opposite of the class value and vanishes for the correct result.
Cross entropy therefore ensures, that the neural network not only increases the output of the correct class neuron, but also decreases the outputs of the incorrect classes until it converged to the correct solution.

2.3.2 Backpropagation

After knowing how wrong the result is, the network can begin to tweak its parameters to adjust its output closer to the correct result. Let us create a small example of a neural network with 3 total layers j, k and l. The layer j is the input layer, l is the output layer. w_{jki} denote the weights between layers j and k that connect from the neurons of layer j to the i-th neuron in layer k. Similarly we define the weights w_{kli} between the layers k and l. The weights are denoted b_{jki} for the weights between layers j and k influencing the input of the i-th neuron in layer k. Again b_{kli} for the weights between layers k and l. The input of layer k is then calculated as $x_{ki} = y_j * w_{jki} + b_{jki}$ where x_{ki} is the i-th input and x_k the whole input vector. Similarly the input x_l is calculated. The output of each layer is denoted as y and depends on the activation functions of the layers. For the input layer, there is no activation function as we just forward the given data. For layer k we get $y_k = f_k(x_k)$ and $y_l = f_l(x_l)$ for layer l. We also define y as the true data to test our predictions against.
For simplicity, lets first look only at the last two layers k and l and how the weights and biases change for the i-th neuron based on the error between the predictions and the true data. Here, $E = R(y_l, y)$ is the output of the cost function.

$$\begin{aligned} \frac{dE}{dw_{kli}} &= \frac{\partial E}{\partial y_{li}} \frac{dy_{li}}{dw_{kli}} \\ &= \frac{\partial E}{\partial y_{li}} \frac{\partial y_{li}}{\partial x_{li}} \frac{dx_{li}}{dw_{kli}} \end{aligned} \tag{24}$$

Similarly we get for the biases:

$$\begin{aligned} \frac{dE}{db_{kli}} &= \frac{\partial E}{\partial y_{li}} \frac{dy_{li}}{db_{kli}} \\ &= \frac{\partial E}{\partial y_{li}} \frac{\partial y_{li}}{\partial x_{li}} \frac{dx_{li}}{db_{kli}} \end{aligned} \tag{25}$$

We see that the error has an easy to follow and calculate derivative with respect to the weights and biases. This means that if we know how each of these derivatives behave, we can calculate how to change the weights and biases in order to decrease the error.

Now, we calculate the derivative towards the weights and biases in layer j.

$$\begin{aligned}\frac{dE}{dw_{jki}} &= \frac{\partial E}{\partial y_l}\frac{dy_l}{dw_{jki}} \\ &= \frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial x_l}\frac{dx_l}{dw_{jki}} \\ &= \frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial x_l}\frac{\partial x_l}{\partial y_k}\frac{dy_k}{dw_{jki}} \\ &= \frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial x_l}\frac{\partial x_l}{\partial y_k}\frac{\partial y_{ki}}{\partial x_{ki}}\frac{dx_{ki}}{dw_{jki}}\end{aligned} \tag{26}$$

$$\begin{aligned}\frac{dE}{db_{jki}} &= \frac{\partial E}{\partial y_l}\frac{dy_l}{db_{jki}} \\ &= \frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial x_l}\frac{dx_l}{db_{jki}} \\ &= \frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial x_l}\frac{\partial x_l}{\partial y_k}\frac{dy_k}{db_{jki}} \\ &= \frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial x_l}\frac{\partial x_l}{\partial y_k}\frac{\partial y_{ki}}{\partial x_{ki}}\frac{dx_{ki}}{db_{jki}}\end{aligned} \tag{27}$$

We can see that in the derivations the first terms are very similar. When backpropagating through several layers, every layer will add another term of the same shape to the total derivative for the next layer. This can be used to an advantage when writing the code for backpropagation. Because these numerical values do not change they can be communicated to the next layer where they are used for calculating the derivative, before multiplying with the layer-dependent constant and passing it on to the next layer.
It should now also become apparent why it is so important to choose functions for the loss and activation functions that are easily differentiable. We can also see when a neural network stops adapting certain weights and biases. The first case is when the derivative of the error is zero. In this case we have either found the solution or a local minimum but no adjustment will be made. Ideally only this will happen and the other derivatives keep producing outputs. The derivative of the input towards the weights or biases has a purely linear form and depends at most on the input.
The derivative of the activation function can however cause problems. For the sigmoid function, the derivative vanishes for very large positive or negative values. In this case the error is not propagated backwards to the weights and biases, they are not adapted and the network stops learning in this part. The output of the sigmoid function can only change if the weights and biases are adapted, which will

not happen in this case. The weights are therefore locked in until somewhere further upstream the values change enough to make this particular neuron produce a new derivative.
A similar problem arrives for the ReLU function, because values below 0 produce no derivative and therefore also reduce the learning chance. Both ELU and Leaky ReLU attempt to fix this, once by producing an output below 0 similar to the sigmoid function and once by continuing linearly but with a lower value. Because the Leaky ReLU follows a linear pattern above and below zero just with different derivatives it also faces the problem that it might fail to describe non-linear problems efficiently.

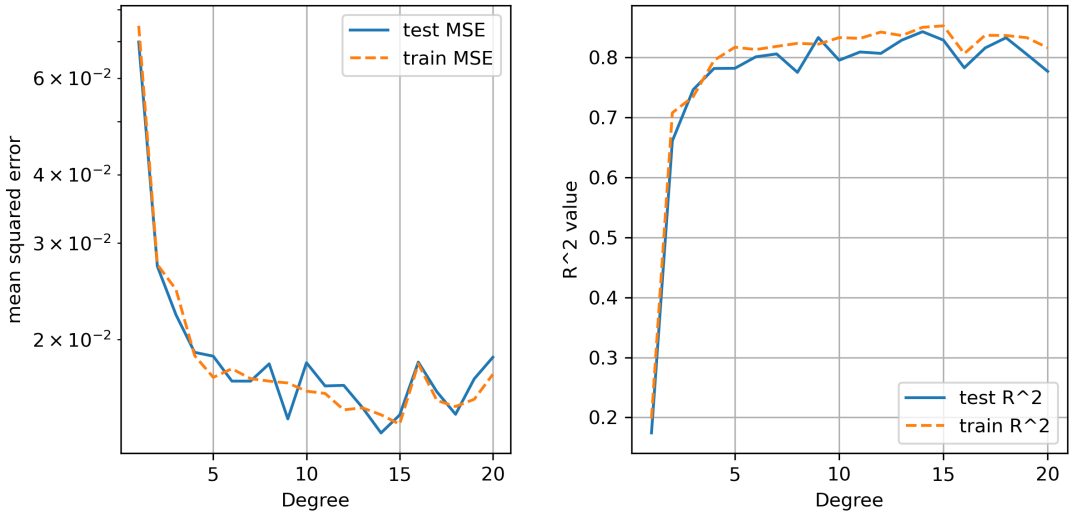
3 Tasks

3.1 Stochastic gradient descent

For this task I replaced the matrix inversion for ordinary least squares regression with an algorithm that implements stochastic gradient descent with RMSProp. The data is generated using the franke function with x and y values between zero and one, forming a grid with a resolution of 0.02. This gives 2500 points of data. Onto the value of the franke function randomly distributed noise with an amplitude of 0.1 is added. The data is randomly split into 80% for training and 20% for testing. These parameters are the same as for the last project to make comparisons between the different algorithms easier.

3.1.1 Choosing a polynomial degree

Figure 1 shows how the mean squared error and R^2 value for different degrees of the polynomial behave. I chose an initial learning rate of $\eta_0 = 10^{-3}$, split the data into 25 batches of equal size ($n_{\text{batch}} = 80$) and trained the algorithm for 150 epochs. As we can see the algorithm produces good results already for a polynomial degree of 5. The lowest mean squared error is reached for a degree of 15, where also the best R^2 value is obtained. Contrary to ordinary least squares in the previous project we do not see a separation between testing and training data for higher degrees which is an indicator for overfitting, but rather the errors for test and training data both increase.
In fig. 2 we also see how the fit parameters behave for higher order polynomials. Contrary to ordinary least squares with matrix inversion the parameters belonging to higher orders do not dominate but the parameters are all within the same range and share similar values. For this range of polynomials we therefore do not yet see



(a) (b)

Figure 1: Mean squared error (left) and R^2 value (right) for different degrees of a polynomial

overfitting.
When performing further analysis I have chosen to use a degree of $n = 10$. For a degree of this kind there will be 66 parameters to optimise, so it is reasonably complex without having problems for being too complex.

3.1.2 Evaluating the learning rate

After having decided on a subset of data to evaluate, the learning rate is the next parameter to optimise. I keep my choice of batch size and epochs the same and vary the learning rate equidistant in logarithmic space between 10^{-8} and 10^{2}. Figure 3 shows how different learning rates influence the mean squared error and goodness of the fit after 150 epochs. A learning rate of $\eta_0 \approx 10^{-3}$ seems to be the best choice for an initial learning rate. Learning rates much lower show no convergence and much larger learning rate show divergence. All of these evaluations were still performed with a fixed amount of batches and only the metrics for the last epoch were taken into account. Next I therefore perform a gridsearch over different amounts of training epochs, amount of batches and learning rates to see how different batch sizes might influence the result and whether a lower amount of epochs can be used of a higher will yield significantly better results. This evaluation takes place for up to 200 epochs, 1-20 batches ($n_{\text{batch}} = 100 - 2000$) and learning rates between 10^{-4} and 10^{-1}. The training data achieves its lowest mean squared error for 200 epochs,

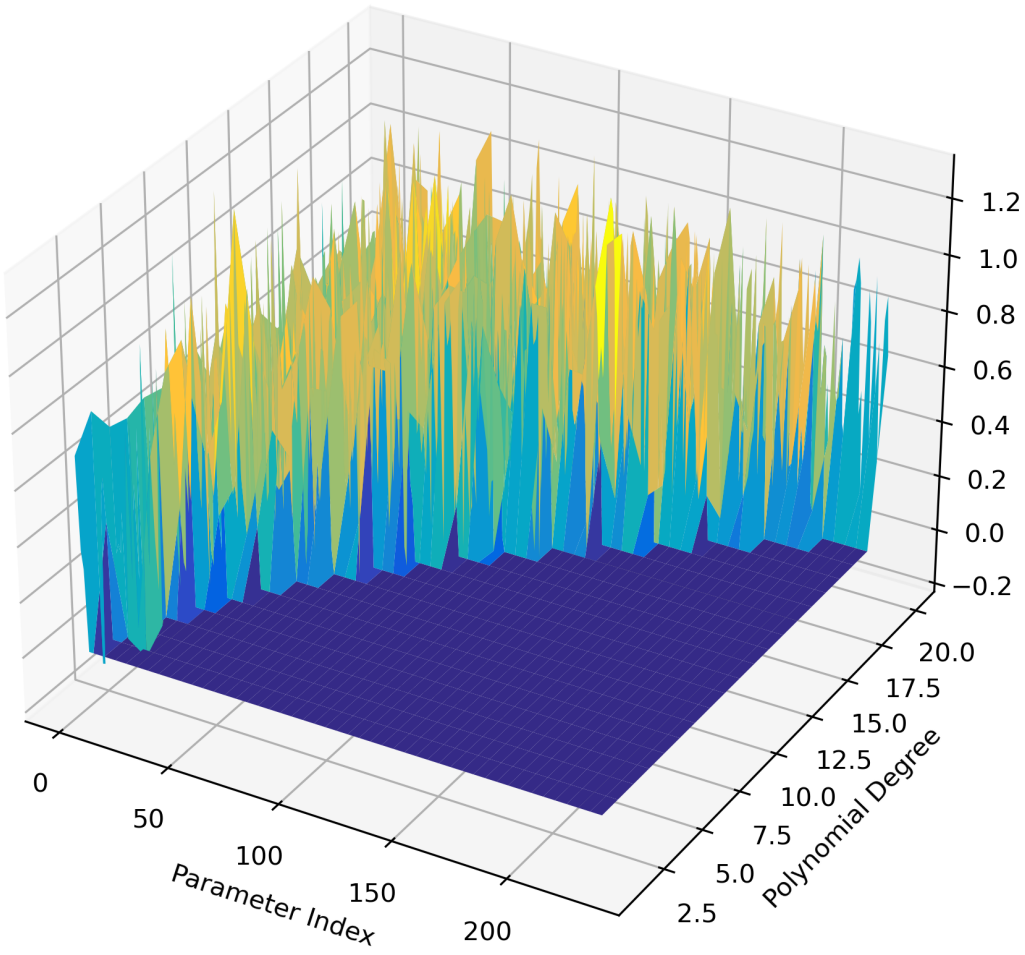
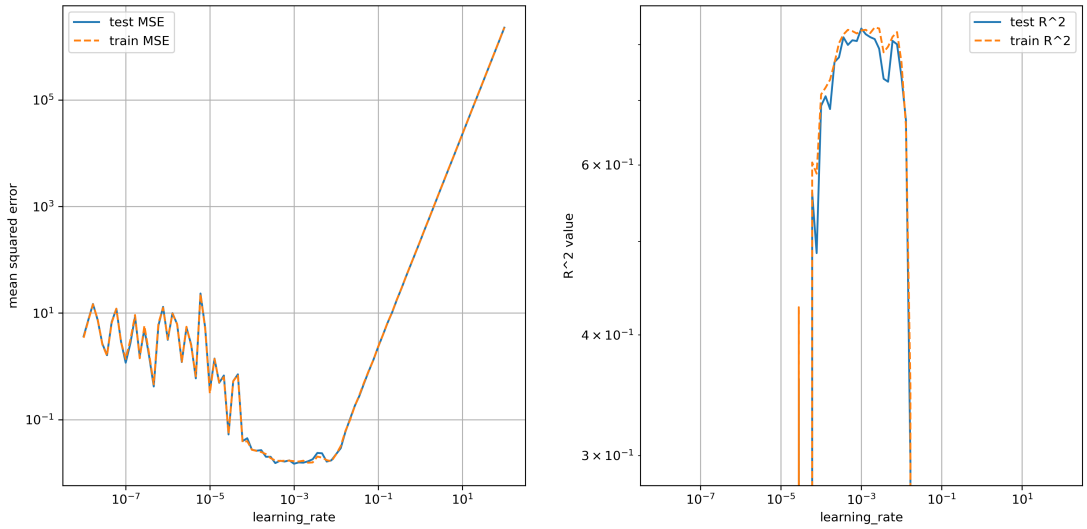


Figure 2: Values for regression parameters for ordinary least squares for 150 epochs and 25 batches.



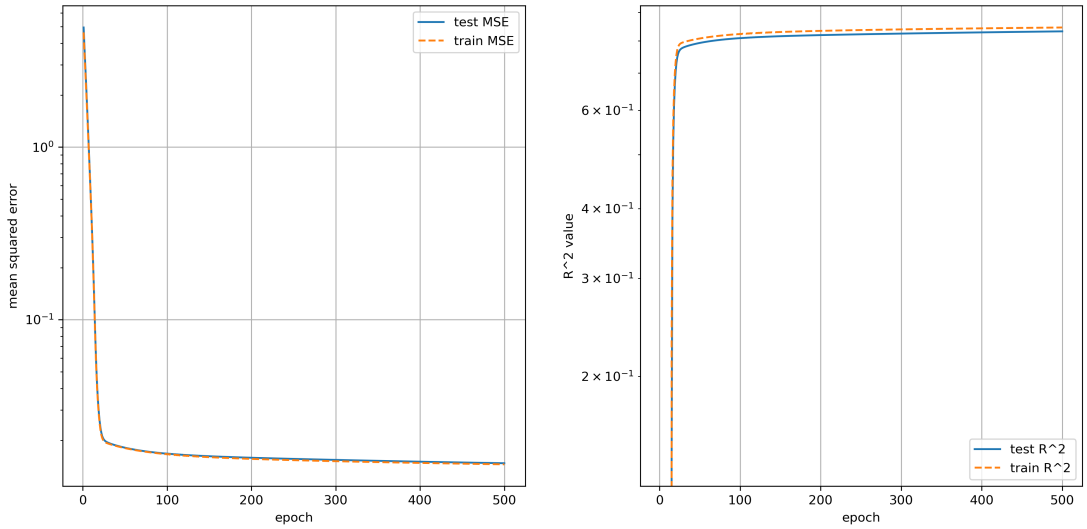
(a) (b)

Figure 3: Mean squared error (left) and R^2 value (right) for different learning rates

20 batches and a learning rate of $\eta_0 = 1.38 * 10^{-3}$. The learning rate is equal to the previously found learning rate. That the training data prefers the highest amount of epochs and batches is to be expected because this means that as long there is no overfitting, the model will learn albeit slowly. For the testing data the ideal amount of epochs is reached at 199, with 12 batches ($n_{\text{batches}} = 167$) and a learning rate $\eta_0 = 1.20 * 10^{-3}$. This shows that training and testing data agree about the initial learning rate. The batch size should be adapted to fit the testing data better. In fig. 4 we see how the mean squared error and goodness of the fit behave for increasing amounts of epochs with an initial learning rate of $\eta_0 = 1 * 10^{-3}$ and 12 batches. For the first ≈ 50 epochs we see a strong decrease in the mean squared error, afterwards the mse begins to stagnate. Here the mean squared error reaches its lowest value of MSE $\approx 1.5 * 10^{-2}$. Values of MSE $\approx 2.0 * 10^{-2}$ or less could already be achieved in the initial testing run where the degree of the polynomial was selected but only training up to 150 epochs was performed. This is however a slight improvement over ordinary least squares in the last project, where the lowest achieved mean squared error was MSE $\approx 2 * 10^{-2}$

3.1.3 Ridge regression

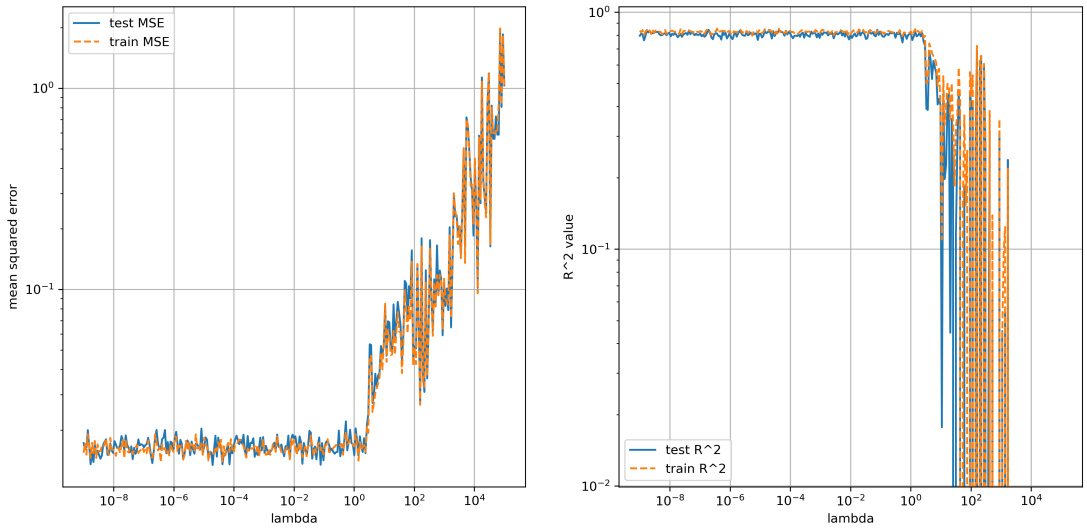
Next I apply the previously found hyperparameters to ridge regression. I choose amounts of batches of 10, 12, 14, 16 and 18 each, with learning rates $\eta_0 = 1.2 * 10^{-3}$ and search for an ideal regularisation parameter λ between 10^{-7} and 10^2. The



(a) (b)

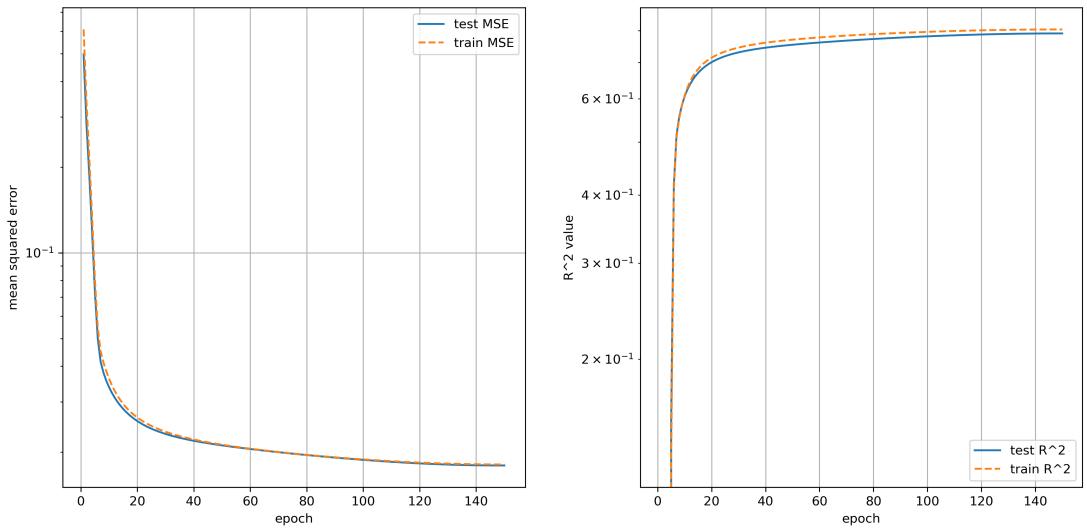
Figure 4: Mean squared error (left) and R^2 value (right) for increasing training epochs

evaluation is performed over a maximum of 150 epochs, based on my previous observations that most of the learning has been completed at this point.
In fig. 5 we can see how the mean squared error and goodness of the fit behave for different parameters λ for ridge regression for 18 batches ($n_{\text{batch}} = 111$) and $\eta_0 = 1.2 * 10^{-3}$. For high values above 10^1 the regularisation parameter starts to suppress the parameters too much and the mean squared error increases for higher values. For values of 10^0 or lower the effect is lower and the mean squared error converges towards values we have already seen for ordinary least squares regression. The model converges within about 100 epochs to its final value, similar to ordinary regression without a regularisation parameter of $\lambda = 1$. The values for the mean squared error and R^2 value for the fit are within what we already have seen for least squares regression. This convergence is shown in fig. 6.
As already shown in fig. 2, the values for the regression parameters are all very close and take values between -0.2 and 1.2. These values are already well regularized, there are no outliers or very large parameters that need to be lowered and the overall model is not overfitting. Because none of these cases appear, there is nothing for ridge regression to penalise and we therefore see no significant improvement over ordinary least squares. If for example the model data was more sparse or errors were higher, we could see improvement here.



(a) (b)

Figure 5: Mean squared error (left) and R^2 value (right) for different lambdas for 18 batches ($n_{\text{batch}} = 111$) and $\eta_0 = 1.2 * 10^{-3}$



(a) (b)

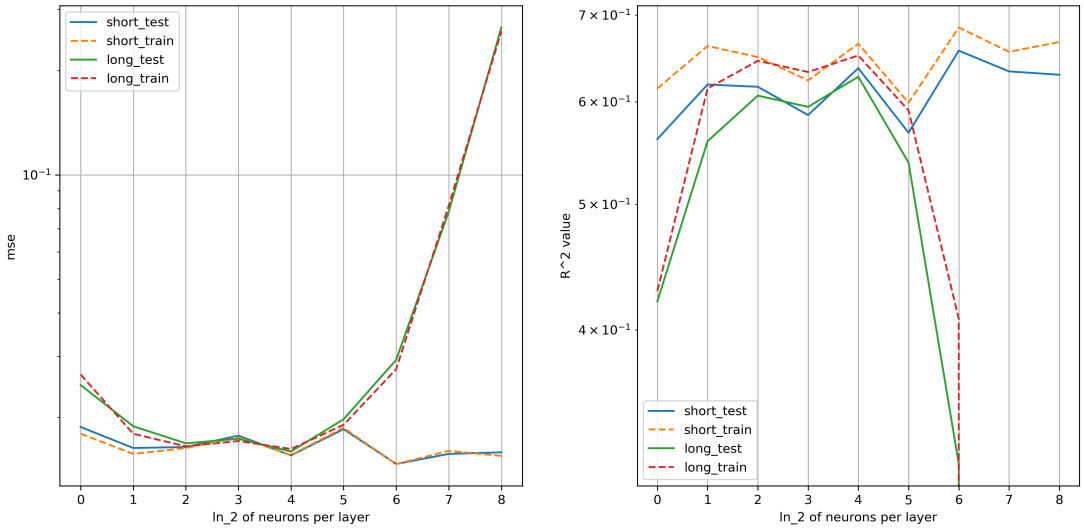
Figure 6: Mean squared error (left) and R^2 value (right) for increasing epochs for ridge regression with $\lambda = 1$

4 Neural Network for regression

In this part I have replaced fitting parameters of the polynomial with a neural network that takes the coordinates of the grid (x and y values between 0 and 1) and fits those to the output value of the franke function. The franke function takes values between -0.2 and +1.4, which limits the available output functions or necessitates preprocessing of the data produced by the franke function.
Because I wanted to keep the original data intact, the output function has to be able to describe the whole range of possible outputs. The ReLU function cannot produce negative outputs and the sigmoid function can only produce outputs between 0 and 1. This leaves the two variations of the ReLU function, the ELU and leaky ReLU and a linear output function, which passes the input directly to the output after applying weights and biases.
I will test different shapes and numbers for the middle layers, different activation functions for the middle layers and compare my results to how a neural network created by keras[3] and implemented in tensorflow[4] performs.
The results will be presented based on the mean squared error and R^2 value.

4.1 Own network code with different sizes and layers

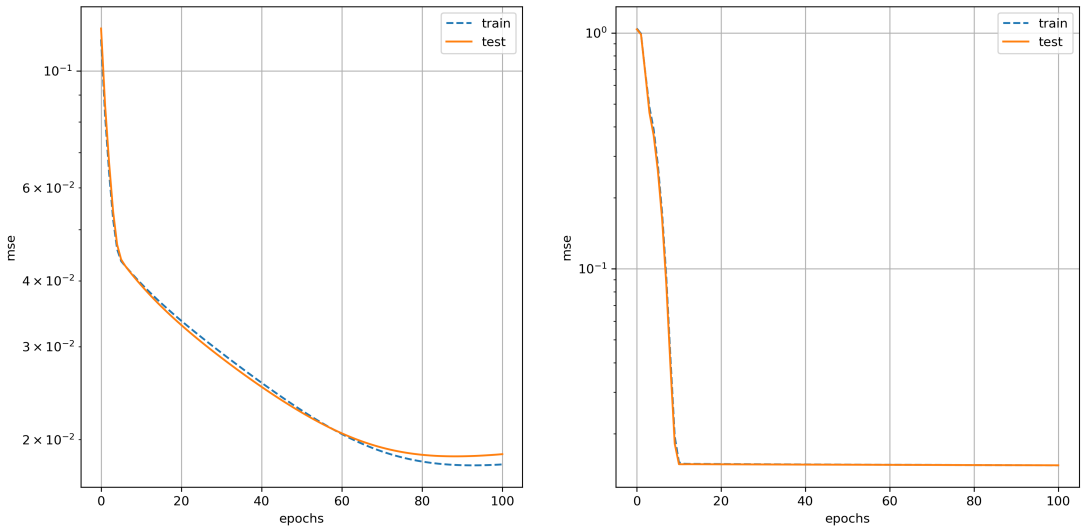
My own network is a feed forward neural network where layers can easily be added. Activation functions, learning rates and cost functions can be chosen differently per layer if necessary. When working on this type of problem the shape of the inputs will always be equal to the batch size in one dimension and equal to 2 for the two coordinates in the other dimension and the output layer will be of the shape of the batch size in one dimension and shape 1 in the other for the single output value of the franke function.
I will vary the size of the layer in the middle and test a single layer and two middle layer neural networks on this problem. This middle layer will use the sigmoid function function as its activation function. The different tested networks will have between 1 and 256 neurons in steps equal to the power of 2 in between. This gives nine different shapes for the middle layers and two combinations of those middle layers each, resulting in 18 total configurations. Each of these configurations will be tested with learning rates between 10^0 and 10^{-4}. The networks containing one middle layer will be called "short" and the networks containing two layers will be called "long".
I will show some of the produced graphs here and mention the result of other test runs. The remaining figures can be found in the accompanying github repository.



(a) (b)

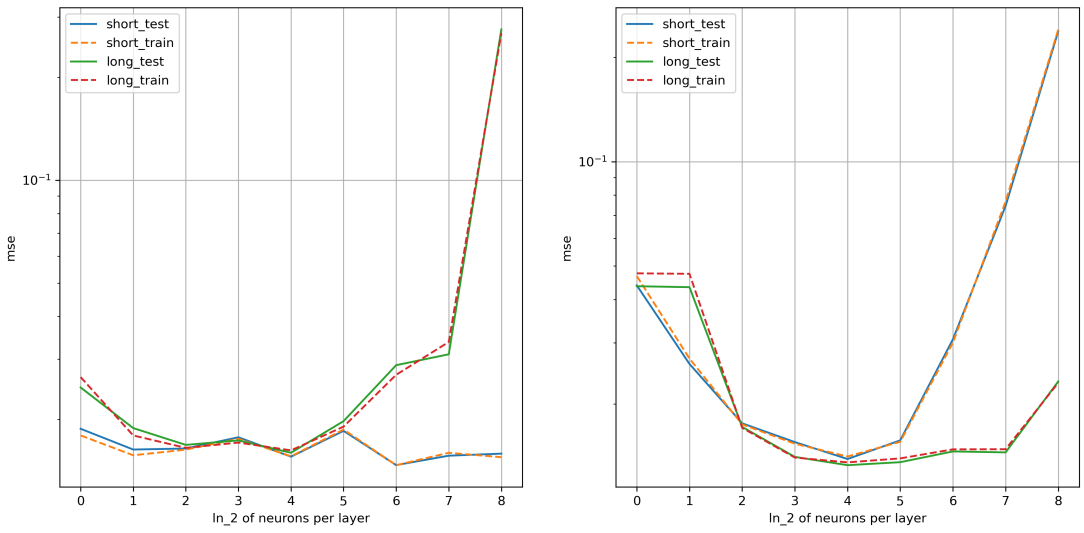
Figure 7: Mean squared error (left) and R^2 value (right) depending on the amount of neurons in the middle layer(s) for $\eta_0 = 10^{-3}$ and ELU as activation function for the output layer

In fig. 7 we see how the different networks perform for varying sizes. Here, ELU has been chosen as the activation function for the output layer with $\alpha = 1$ and the initial learning rate was set to $\eta_0 = 10^{-3}$. Best performance is achieved for the short neural network with $2^6 = 64$ neurons. This is comparable to the amount of parameters that can be tweaked for a polynomial of degree 10, which has 66 individual parameters. This configuration has achieved an MSE of $\approx 3 * 10^{-2}$ with an R^2 value ≈ 0.65. Both of these values are comparable to what we have seen previously when performing a fit onto the polynomial of degree 10.
Comparing the short neural networks to the long networks, we can see that the short networks always performed equally well or better than the long ones. Even the short neural network with just one neuron in the middle layer performs well and outperforms the long neural network with $2^8 = 256$ neurons.
As shown in fig. 8 the neural network with 1 middle neuron (left) converges much slower than the network with 64 neurons (right). The smaller network takes approximately 90 epochs, whereas the larger network converges after approximately 10 epochs.
It is also interesting to replace the output function in the output layer with a linear output function or increase the learning rate by a factor of 10. This is shown in fig. 9a. For the short networks, the linear activation function performs similarly



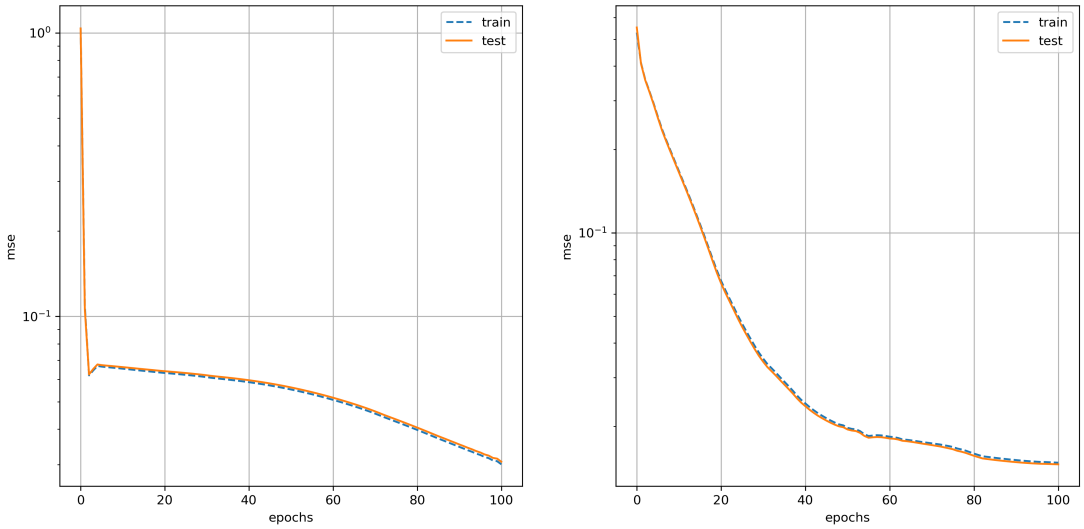
(a) (b)

Figure 8: Mean squared error of the short networks with 2^0 (left) and 2^6 (right) neurons with ELU activation function in the last layer and $\eta_0 = 10^{-3}$.



(a) (b)

Figure 9: Mean squared error for a linear activation function in the output layer (left) and a higher learning rate of $\eta_0 = 10^{-2}$ (right)



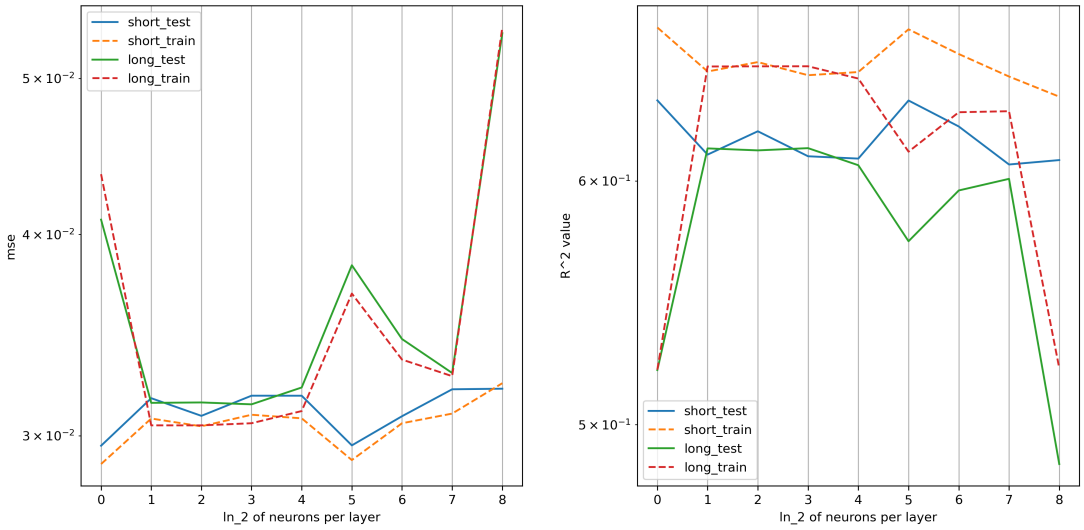
(a) (b)

Figure 10: Mean squared error for short network (left) and long network (right) with 64 neurons in the middle layer and an increased learning rate of $\eta_0 = 10^{-2}$

but it performs slightly better for the long networks. This difference is notably mostly for the layers with higher neuron count.
Increasing the learning rate significantly increases the error for the networks with lower amounts of neurons and increases the performance for the long networks with higher neuron counts.
This shows itself when investigating how the neural network performs over the different epochs. Figure 10 shows the performance for a short network with 64 middle neurons (left) and the same long network, with an increased learning rate of $\eta_0 = 10^{-2}$. The long network converges slowly, whereas the short network sees a rapid convergence in the beginning, then levels out but starts learning again towards the end of the destined amount of epochs. Allowing this network to train for longer could have enabled it to catch up to the performance of the long network.

4.2 Comparison to Tensorflow / Keras

For comparison between my own flow and tensorflow / keras I perform an analysis of a similar network structure. The networks consist of an input layer of the same shape, an output layer of the same shape albeit only with the ELU function as activation function for the outer layer since we saw the best performance for this function for our network.
First, fig. 11 shows how the mean squared error and R^2 value behave depending



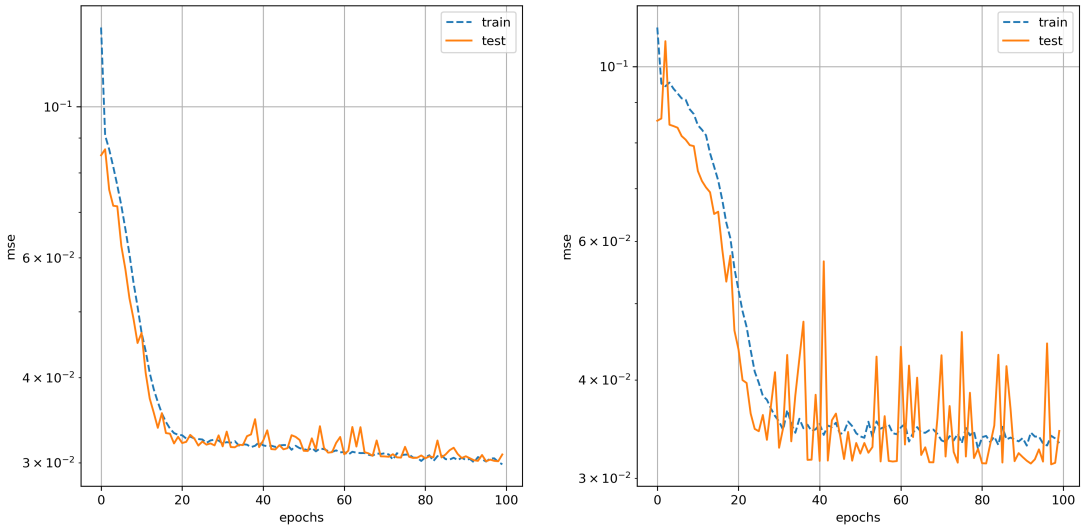
(a) (b)

Figure 11: Mean squared error (left) and R^2 value (right) depending on the amount of neurons in the middle layer(s) for $\eta_0 = 10^{-3}$ and ELU as activation function for the output layer, computed by keras

on the amount of neurons in the middle layer and the amount of middle layers. This fig is the same as fig. 7 but computed with keras. The figures for the mean squared errors and R^2 values show similar shapes, which shows that the computation performed by my own neural network code for the same network structure works out well.
The mean squared error computed by keras also levels out at $\approx 3 * 10^{-2}$ with $R^2 \approx 0.65$. For keras the long neural networks are similarly difficult and perform worse for higher amounts of neurons, but for neuron counts between 2 and 32 the structures are almost equal with the short networks again performing better. While the cost function for the training data converges smoothly in keras, the testing data is more prone to vary around its mean as shown in fig. 12. This is especially pronounced for the longer networks and shows itself not only in the here shown figure but also in other configurations.

4.3 Varying output activation functions

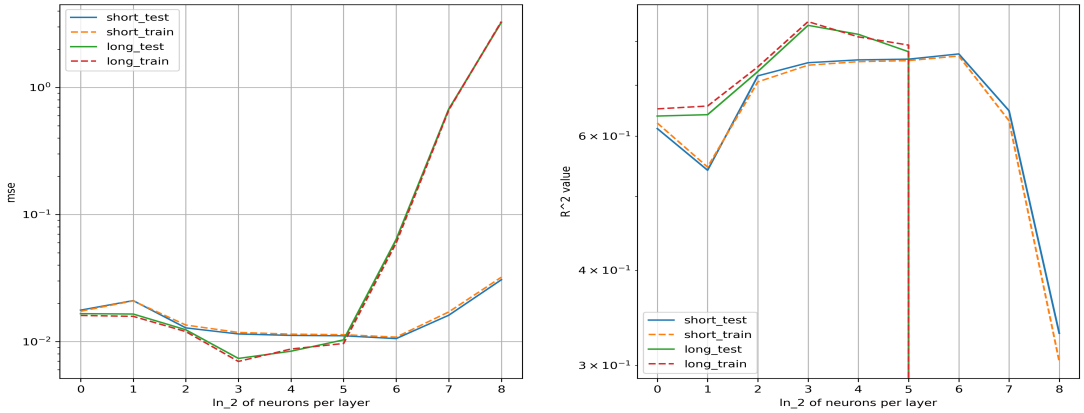
So far we have found out that for the output layer ELU works well as an activation function and that an initial learning rate of $\eta_0 = 10^{-3}$ is a good choice. I will now test the ReLU, leaky ReLU with $\alpha = 0.02$ and linear activation functions for the inner layer(s).



(a) (b)

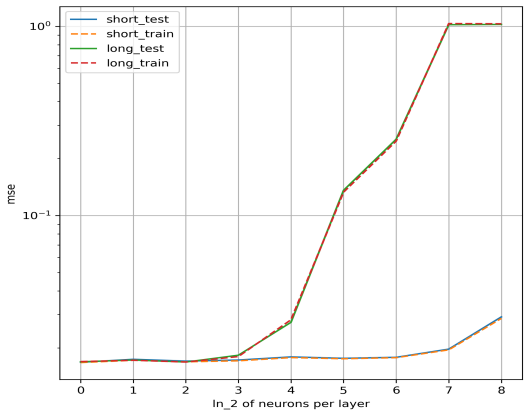
Figure 12: Mean squared error for a short network (left) and a long network (right) with 2^6 middle neurons, learning rate $\eta_0 = 10^{-3}$, computed by keras

The results are shown in fig. 13. The left column shows the mean squared errors, the right column the R^2 values. The top row corresponds to leaky Relu with $\alpha = 0.02$, the middle row is the linear function and the bottom row ReLU.
The leaky ReLU function performs well for middle layers with neurons counts between 2^2 and 2^5 for both short and long networks and even well for short networks with higher neuron counts, but breaks for those long networks. The mean squared error is lowest at $\approx 7 * 10^{-3}$ for the long network with 2^3 neurons and reaches an R^2value ≈ 0.8, which is higher than for the previous tests.
The linear function only performs well up for neuron counts up to 2^3 for the long networks and 2^7 for the short networks. The mean squared error is higher than in the previous cases and the R^2 value is lower. This is not surprising because the franke function is a non-linear function. Using mostly linear functions to describe it, will not be sufficient.
The basic ReLU has difficulties with the networks with just one middle neuron, both long and short and cannot fit the long networks with 2^6 or more neurons. For the other cases it performs consistently well with mean squarer errors at lower than $\approx 10^{-2}$ and R^2value of up to ≈ 0.85 for the long network with 2^4 neurons. The short network of same neuron count performs only slightly worse.

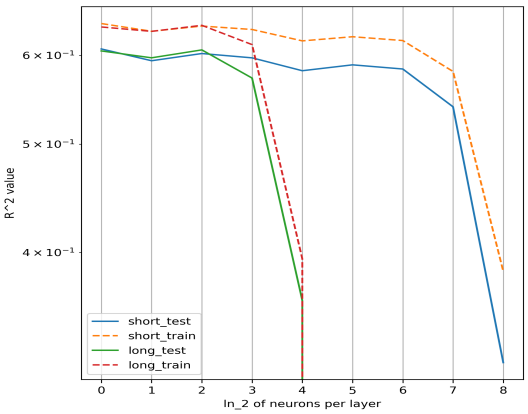


(a)

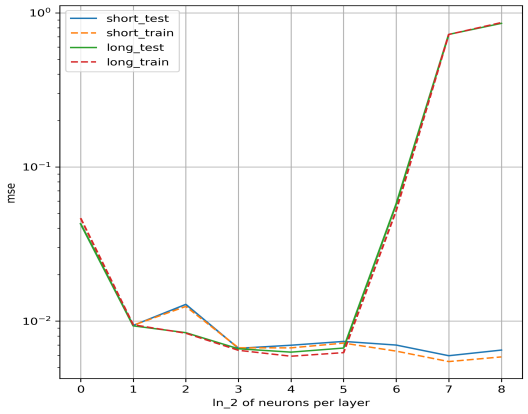
(b)



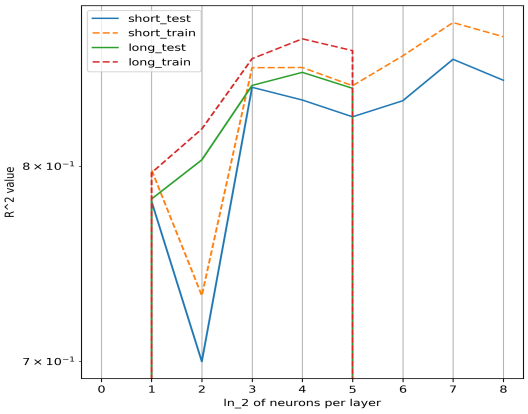
(c)



(d)



(e)



(f)

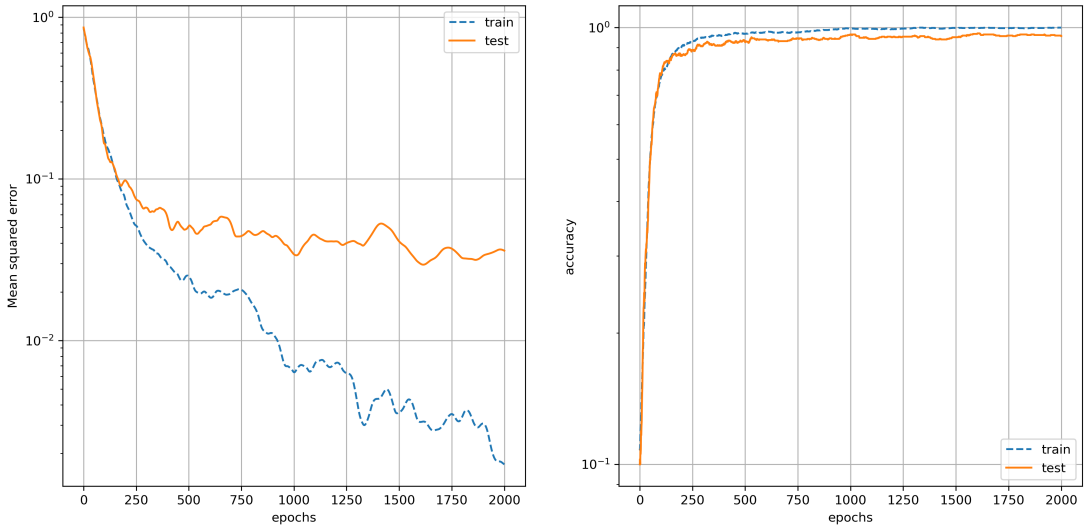
Figure 13: Mean squared errors (left column) and R^2 values (right column) for different activation functions in the middle layer(s). Top row shows leaky Relu with $\alpha = 0.02$, middle row is the linear function and bottom row ReLU.

4.4 Classification of images of digits

After having successfully shown the applications and limits of different neural network architectures the learned processes were applied to a different problem. Here we want to classify images of digits into categories which represent the numerical value of said digits. The images are produced from scikit-learn load_digits function which returns a numerical representation of up to 1797 digits. [5] A single image is represented by a label Y containing the value of the digit it represents and a numerical vector X of length 64, which is a representation of the flattened 8x8 px image. The image vector takes values in the range of 0-16. This large range of values sparks the possibility to saturate the sigmoid function and decrease any chances of the network learning successfully. To scale the values, I subtract half the maximum and divide by the same value. The resulting vector then takes values between -1 and +1. The mean value of these over the whole set of data is calculated to be -0.389, therefore there is a small bias towards negative numbers, but saturation after initialisation should be less likely.
Each of the labels are converted from its numerical value into a one-hot vector, which represents the class of the label. This means that instead of the label having the value i, the label is represented by a vector which is zero everywhere, except for its i-th position, where it takes the value one. This representation can be interpreted as the probability of the label being in any class but i to be zero, and the probability to be in class i as one.
For this reason we will use the softmax function as the activation function for the last layer. For every possible digit this will give us the probability calculated by the classifier for the digit to be in that category.

4.4.1 Mean squared error as cost function

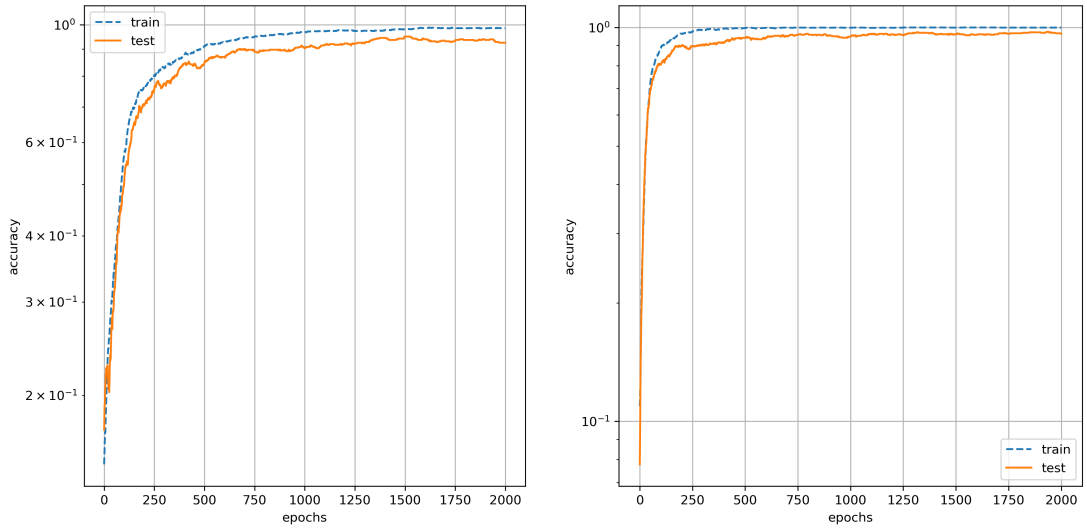
At first we want to stay close to our latest test and only change the function of the output layer. The cost function and therefore the error which will be propagated backwards is calculated as the mean squared error. The used network will consist of three layers. The first layer will take 64 values representing the scaled image pixels as input. I have chosen the ELU function as activation function because it showed previously to scale well over the different problems and might even work when the network's architecture is not ideally suited. The middle layer will consist of 256 neurons with the sigmoid function as the activation function. Although I have shown that a small number of neurons can be sufficient, I have also shown that a kind of network like this reaches its limitations faster. I will also need more neurons because there is more input data to be processed and there are more outputs. This is not the first value I have tried and not the lowest I had success with, but this amount worked consistently over different random initialisations.



(a) (b)

Figure 14: Mean squared error and accuracy of prediction digit data.

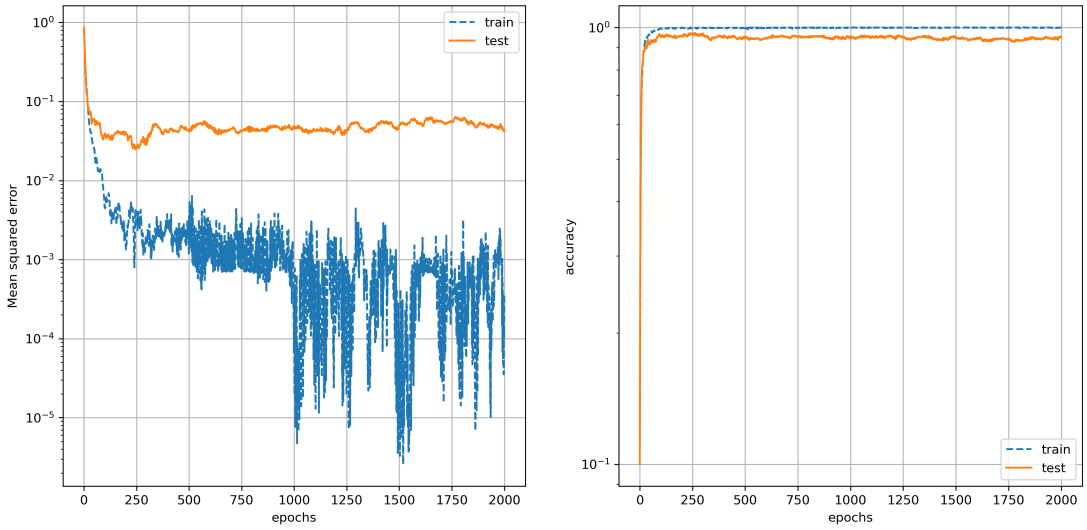
I chose the sigmoid function, because it has a maximum output. Because the softmax function relies on calculating the exponential of its input first, passing too high values into the last layer could lead to overflow in the softmax function. If the sigmoid function restricts this, the weights would have to increase to huge values to cause this.
The last layer has 10 output neurons where each represents the network's prediction for the given digit to be in the corresponding class. This is calculated by the softmax function as mentioned before. Because the network is very simple and training therefore very cheap, I chose a lower initial learning rate of $\eta_0 = 10^{-4}$. Even if convergence does not occur quickly, my code allows for the network to remember its status and keep training.
Figure 14 shows the mean squared error and accuracy of this training run. After 2000 epochs this network reached 99.93% precision on the training data and predicted 95.56% of the testing data correctly. Most of the training was done after the first 500 epochs and the last few promille of accuracy were achieved in the following epochs. This can also be tracked via the mean squared error, which decreased about equally until 500 epochs, where after only the training data profited and the testing data's error only decreased slightly.
Decreasing the amount of neurons in the middle layer to 128 lead to a decrease in testing accuracy to 92.50%. Increasing the amount of neurons to 512 increased the training time significantly but only lead to an increase in testing accuracy to



(a) (b)

Figure 15: Accuracy of prediction digit data for 128 in the middle layer (left) and 512 neurons (right)

96.39%. When inspecting the smoothness of the prediction graph for the testing data, we can see small bumps up and down, where a change in the network caused rapid change in its predictive ability, both increasing and decreasing it. These bumps were less likely to occur or less distinct for the neural network with 512 neurons and even more likely for the neural network with 128 neurons.
This is shown in fig. 15, where the left figure shows the accuracy for 128 neurons in the middle layer and 512 neurons in the middle layer on the right. The network with 512 neurons in the middle layer also converges quicker.
At last I want to look at how different learning rates may affect the outcome. I have therefore increased the learning rate by a factor of 10 to $\eta_0 = 10^{-3}$ but kept the amount of middle-neurons the same at 256.
In fig. 16 we can see how the mean squared error for the training data can be very volatile and vary by orders of magnitude between two training runs. RMSProp limits how this can effect the backpropagation between several batches, but we can still see how the testing error starts to increase after reaching its minimum after 250 epochs. This might be due to the rapid changes in the cost function or just the result of overfitting to the data. Still, this shows how important it is too choose the learning rate efficiently.



(a) (b)

Figure 16: Mean squared error and accuracy of prediction digit data for increased learning rate $\eta_0 = 10^{-3}$

4.5 Logistic Regression

At last we want to replace the cost function with logistic regression. Logistic regression has been developed to fit categorization problems and should fit problems like this therefore better. Because the architecture has worked so far, I will keep the same neural network with 256 neurons in the middle layer, a learning rate $\eta_0 = 10^{-4}$ and the same activation functions with the exception of the last layer, where the costfunction will be replaced by logistic regression.
Figure 17 shows that for these parameters the network converges towards 85% accuracy. When inspecting the mean squared error and cross entropy for this type of network, we see that both level out as well and that the network has stopped learning. Since we have not used cross entropy as a metric in the previous case, I use the mean squared error here to draw a comparison. Although we do not optimize for a low mean squared error, a correct prediction still results in a low mean squared error. The mean squared error achieved with logistic regression is almost an order of magnitude higher than in the previous cases.

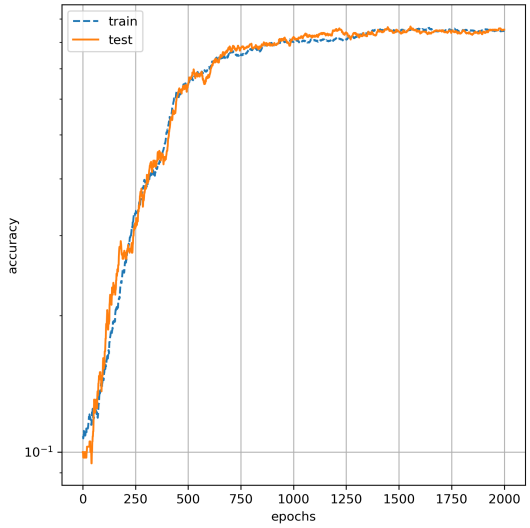
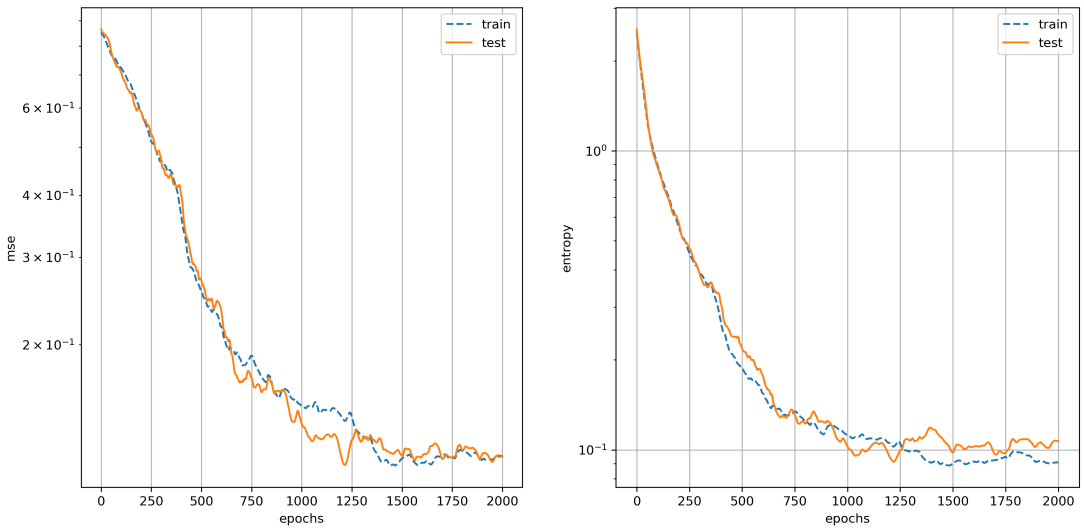


Figure 17: Prediction accuracy of the model using logistic regression



(a) (b)

Figure 18: Mean squared error and cross entropy of prediction digit data when using cross entropy as a loss function

5 Conclusion

When searching for a numerical solution to a problem that cannot be solved analytically, one might be tempted to just use an overly complex neural network, train it over night and return to a perfect solution the next morning. When analysing neural networks of different sizes (and therefore complexity) in this report, we have however shown that the performance of neural networks tends to degrade when the network becomes too complex for the problem it is intended to solve. For the same problem of fitting data onto the Franke function, the neural network performed similarly well with a neuron count similar to the amount of parameters one would have when fitting with OLS.
Comparing my own implementation of a neural network to that by Keras, I see similar performance, confirming that my implementation is working as desired.
Varying the activation functions in the neural network we can see essential differences. Both Leaky ReLU and the linear output function struggle to fit the data well, whereas the basic ReLU and ELU perform better for these cases.
Classifying digits works well when the cost function is chosen to be the mean squared error between the one-hot encoded class vectors. 95% accuracy in the training data can be achieved with a simple neural network after already about 250 epochs. Training time for this kind of network is therefore low. Variations in the complexity of the network again shows that a more complex network does not necessarily lead to an increase in performance and that the increase in training time might not be worth it. The learning rate also has to be chosen carefully, as an increase in the learning rate might lead to fast convergence but also overfitting. Replacing the cost function with logistic regression we find that the network is not able to learn as well as for the previous case and only an accuracy of 85% can be reached.

References

[1] Morten Hjorth-Jensen. Week 39: Optimization and gradient methods. https://compphysics.github.io/MachineLearning/doc/pub/week39/html/week39.html (accessed: 2020-11-03).

[2] Morten Hjorth-Jensen. Week 40: From stochastic gradient descent to neural networks. https://compphysics.github.io/MachineLearning/doc/pub/week40/html/week40.html (accessed: 2020-11-03).

[3] Francois Chollet et al. Keras, 2015.

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[5] sklearn.datasets.load_digits. https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html(accessed: 2020-10-26).

[6] jmlb. Implement a custom metric function in keras. https://jmlb.github.io/ml/2017/03/20/CoeffDetermination_CustomMetric4Keras/ (accessed: 2020-10-23).

[7] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn:

Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[9] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

Python Code

The python code used to create the data this report is based on can be found in the following github repository: https://github.uio.no/Pascalsa/FYS4155 or at the end of this report. Each task has its own function that creates figures put into subfolders labeled according to the task they belong to.
[6] was used to create "coeff_determination" in "main.py". The usage of numpy[7], Scikit-learn[8] and matplotlib[9] was essential for this project.
The code for fitting the parameters using stochastic gradient descend has been built upon the code for the first project. This did not yield the most efficient or clean code for the task, considering it was never meant to do this in the first place, but was easier than to start from scratch.

```
1  from tensorflow.python.keras.utils.vis_utils import
       plot_model
2  from tensorflow import keras
3  from tensorflow.keras import layers
4
5  from regression import task_a, create_grid, RESOLUTION,
       franke, NOISE_LEVEL, mse_error, r2_error, \
6      print_errors
7  from sklearn.model_selection import train_test_split
8  import matplotlib.pyplot as plt
9  from network import LayerDense, Costfunctions,
       ActivationFunctions, Metrics, Network
10 import numpy as np
11
12 from sklearn import datasets
13
14 from keras import backend as K
15
16
17 # https://jmlb.github.io/ml/2017/03/20/
       CoeffDetermination_CustomMetric4Keras/
18 def coeff_determination(y_true, y_pred):
19     r = (K.square(y_true - y_pred)) / K.sum(K.square(y_true
           - K.mean(y_pred)))
20     return 1 - K.sum(r, axis=0)
21
22
23 def create_terrain_data():
```

```
24      split_size = 0.8
25      x, y = create_grid(RESOLUTION)
26      z = np.expand_dims(franke(x, y, NOISE_LEVEL).flatten(),
            axis=1)
27      feature_mat = np.zeros(shape=(x.size, 2))
28      feature_mat[:, 0] = x.flatten()
29      feature_mat[:, 1] = y.flatten()
30      x_train, x_test, y_train, y_test = train_test_split(
            feature_mat, z, train_size=split_size)
31      return x_train, x_test, y_train, y_test
32
33
34  def print_metrik_by_network_and_epoch(num_epochs, metric,
        task, name, metric_name):
35      fig = plt.figure(figsize=(6, 6), dpi=300)
36      if len(metric) == 2:
37          plt.plot(np.linspace(0, num_epochs, num_epochs + 1)
                , metric[0], label="train",
38                   linestyle="--")
39          plt.plot(np.linspace(0, num_epochs, num_epochs + 1)
                , metric[1], label="test", linestyle="-")
40      else:
41          plt.plot(np.linspace(0, num_epochs, num_epochs + 1)
                , metric, linestyle="--")
42      plt.legend()
43      plt.grid()
44      ax = fig.gca()
45      ax.set_xlabel("epochs")
46      ax.set_ylabel(metric_name)
47      plt.yscale('log')
48      fig.tight_layout()
49      fig.savefig("images/" + task + "/" + name + ".png", dpi
            =300)
50      plt.close()
51
52
53  def create_network_array(n_in, n_out, learning_rate, cf, af
        , laf, start, stop, mf):
54      networks = []
55      for i in range(start, stop + 1):
```

```
56          li = LayerDense(n_in, 2 ** i, "ldi" + str(i),
                learning_rate, cf, af)
57          lm = LayerDense(2 ** i, 2 ** i, "ldm" + str(i),
                learning_rate, cf, af)
58          lo = LayerDense(2 ** i, n_out, "lo" + str(i),
                learning_rate, cf, laf)
59          networks.append(Network([li, lo], str(i) + "short",
                 mf))
60          networks.append(Network([li, lm, lo], str(i) + "
                long", mf))
61      return networks
62
63
64  def create_keras_network_array(n_in, n_out, learning_rate,
        cf, af, start, stop, task):
65      models = []
66      for i in range(start, stop + 1):
67          model1 = keras.Sequential(name=str(i) + "short")
68          model1.add(
69              layers.Dense(n_in, input_shape=(n_in,),
                    activation=af, kernel_initializer='
                    he_uniform'))
70          model1.add(layers.Dense(2 ** i, activation=af,
                kernel_initializer='he_uniform'))
71          model1.add(layers.Dense(n_out, activation='elu',
                kernel_initializer='he_uniform'))
72          plot_model(model1, to_file='images/b/' + model1.
                name + '.png', show_shapes=True,
73                     show_layer_names=True)
74          # model1.summary()
75          model1.compile(optimizer=keras.optimizers.RMSprop(
                learning_rate=learning_rate),
76                         loss=cf, metrics=['MeanSquaredError'
                               , coeff_determination])
77          model2 = keras.Sequential(name=str(i) + "long")
78          model2.add(
79              layers.Dense(n_in, input_shape=(n_in,),
                    activation=af, kernel_initializer='
                    he_uniform'))
80          model2.add(layers.Dense(2 ** i, activation=af))
```

```
81          model2.add(layers.Dense(2 ** i, activation=af))
82          model2.add(layers.Dense(n_out, activation='elu'))
83          plot_model(model2, to_file='images/' + task + '/' +
                model2.name + '.png', show_shapes=True,
84                     show_layer_names=True)
85          # model2.summary()
86          model2.compile(optimizer=keras.optimizers.RMSprop(
                learning_rate=learning_rate),
87                         loss=cf, metrics=['MeanSquaredError'
                               , coeff_determination])
88          models.append(model1)
89          models.append(model2)
90          del model1
91          del model2
92      return models
93
94
95  def prepare_digit_data():
96      digits = datasets.load_digits()
97      x = digits.data
98      t = digits.target
99      n_out = 10
100     y = np.zeros(shape=(x.shape[0], n_out))
101     y[np.arange(t.size), t] = 1
102     split_size = 0.8
103     x = (x - np.max(x) / 2) / (np.max(x) / 2)
104     x_train, x_test, y_train, y_test = train_test_split(x,
            y, train_size=split_size)
105     return x_train, x_test, y_train, y_test
106
107
108 def full_network_test(cf, af, laf, learning_rate, task,
        epochs=100):
109     x_train, x_test, y_train, y_test = create_terrain_data
            ()
110     start = 0
111     stop = 8
112     n = stop - start + 1
113     networks = create_network_array(x_train.shape[1],
            y_train.shape[1], learning_rate, cf, af, laf,
```

```
114                                         start, stop, [Metrics.
                                               mse, Metrics.
                                               coeff_determination
                                               ])
115     batch_size = 50
116     mse_errors = []
117     r2_scores = []
118     for network in networks:
119         network.train(x_train.T, y_train.T, epochs,
               batch_size, x_test.T, y_test.T)
120         mse_test = network.get_test_met()[:, 0]
121         mse_train = network.get_train_met()[:, 0]
122         r2_test = network.get_test_met()[:, 1]
123         r2_train = network.get_train_met()[:, 1]
124         print("Metrics after initialization, after first
               epoch and last epoch ")
125         print(mse_train[0], r2_train[0], mse_train[1],
               r2_train[1], mse_train[-1], r2_train[-1])
126         print_metrik_by_network_and_epoch(epochs, [
               mse_train, mse_test], task,
127                                           "_".join(["own",
                                                 network.name,
                                                 laf.__name__,
                                                 str(
                                                 learning_rate)
                                                 , af.__name__,
                                                  "mse"]),
128                                           "mse")
129         print_metrik_by_network_and_epoch(epochs, [r2_train
               , r2_test], task,
130                                           "_".join(["own",
                                                 network.name,
                                                 laf.__name__,
                                                 str(
                                                 learning_rate)
                                                 , af.__name__,
                                                  "r_squared"])
                                                 ,
131                                           "r2")
132         mse_errors.append([mse_test[-1], mse_train[-1]])
```

```
133             r2_scores.append([r2_test[-1], r2_train[-1]])
134         mse_errors = np.array(mse_errors).reshape(n, 4).T
135         r2_scores = np.array(r2_scores).reshape(n, 4).T
136         print_errors(np.linspace(start, stop, n), mse_errors, ["short_test", "short_train", "long_test",
137                                                                "long_train"],"_".join(["own","mse"]),laf.__name__,str(learning_ra),af.__name__
```

```
                                                                    ])
                                                                    ,

138                     logy=True, xlabel="ln_2 of neurons per
                            layer",
139                     ylabel="mse", task=task)
140     print_errors(np.linspace(start, stop, n), r2_scores, ["
            short_test", "short_train", "long_test",
141                                                                 "
                                                                    long_train
                                                                    "
                                                                    ],

                                                                    "

                                                                    _
                                                                    "

                                                                    .
                                                                    join
                                                                    ([
                                                                    "
                                                                    own
                                                                    "
                                                                    ,

                                                                    "
                                                                    r_squared
                                                                    "
                                                                    ,

                                                                    laf
                                                                    .
                                                                    __name__
                                                                    ,

                                                                    str
                                                                    (
                                                                    learning_ra
                                                                    )
                                                                    ,
```

```
af
.
__name__
])
,

logy
=
True
,

xlabel
=
"
ln_2

of

neurons

per

layer
"
,

ylabel
=
"
R
^2

value
"
,
```

```
                    task=task)
    return networks


```

```
146 def full_network_test_keras(learning_rate, epochs, task):
147     x_train, x_test, y_train, y_test = create_terrain_data
        ()
148     start = 0
149     stop = 8
150     n = stop - start + 1
151     batch_size = 50
152     models = create_keras_network_array(x_train.shape[1],
        y_train.shape[1], learning_rate,
153                                         'MeanSquaredError',
                                              'sigmoid',
                                            start, stop, "b"
                                            )
154     mse_errors = []
155     r2_scores = []
156     for model in models:
157         print(model.name)
158         history = model.fit(x_train, y_train, epochs=epochs
            , batch_size=batch_size,
159                             validation_data=(x_test, y_test
                                ), verbose=0)
160         mse_errors.append([model.evaluate(x_test, y_test)
            [1], model.evaluate(x_train, y_train)[1]])
161         r2_scores.append([model.evaluate(x_test, y_test)
            [2], model.evaluate(x_train, y_train)[2]])
162         print_metrik_by_network_and_epoch(epochs - 1,
163                                           [history.history[
                                              'loss'],
                                              history.
                                              history['
                                              val_loss']],
164                                           task, "_".join(["
                                              keras", model.
                                              name, "
                                              learning_rate"
                                              ,
165                                                           str
                                                              (
                                                              learning_rat
                                                              )
```

```
])
,
"
mse
"
)
```

```
166
167     mse_errors = np.array(mse_errors).reshape(n, 4).T
168     r2_scores = np.array(r2_scores).reshape(n, 4).T
169     print_errors(np.linspace(start, stop, n), mse_errors,
170                  ["short_test", "short_train", "long_test",
                         "long_train"],
171                  "_".join(["keras_mse", "learning_rate",
                         str(learning_rate)]), logy=True,
172                  xlabel="ln_2 of neurons per layer", ylabel
                         ="mse", task=task)
173     print_errors(np.linspace(start, stop, n), r2_scores,
174                  ["short_test", "short_train", "long_test",
                         "long_train"],
175                  "_".join(["keras_r_squared", "
                         learning_rate", str(learning_rate)]),
176                  logy=True, xlabel="ln_2 of neurons per
                         layer", ylabel="R^2 value", task=task)
177
178
179 def task_b_own():
180     learning_rate = 0.0001
181     n1 = full_network_test(Costfunctions.mse,
            ActivationFunctions.sigmoid, ActivationFunctions.
            linear,
182                            learning_rate, "b", 100)
183     n2 = full_network_test(Costfunctions.mse,
            ActivationFunctions.sigmoid, ActivationFunctions.elu
            ,
184                            learning_rate, "b", 100)
185     learning_rate = 0.001
186     n3 = full_network_test(Costfunctions.mse,
            ActivationFunctions.sigmoid, ActivationFunctions.
```

```
        linear,
187                         learning_rate, "b", 100)
188     n4 = full_network_test(Costfunctions.mse,
        ActivationFunctions.sigmoid, ActivationFunctions.elu
        ,
189                         learning_rate, "b", 100)
190     learning_rate = 0.01
191     n5 = full_network_test(Costfunctions.mse,
        ActivationFunctions.sigmoid, ActivationFunctions.
        linear,
192                             learning_rate, "b", 100)
193     n6 = full_network_test(Costfunctions.mse,
        ActivationFunctions.sigmoid, ActivationFunctions.elu
        ,
194                             learning_rate, "b", 100)
195     learning_rate = 0.1
196     n7 = full_network_test(Costfunctions.mse,
        ActivationFunctions.sigmoid, ActivationFunctions.
        linear,
197                         learning_rate, "b", 100)
198     n8 = full_network_test(Costfunctions.mse,
        ActivationFunctions.sigmoid, ActivationFunctions.elu
        ,
199                         learning_rate, "b", 100)
200     learning_rate = 1
201     n9 = full_network_test(Costfunctions.mse,
        ActivationFunctions.sigmoid, ActivationFunctions.
        linear,
202                         learning_rate, "b", 100)
203     n10 = full_network_test(Costfunctions.mse,
        ActivationFunctions.sigmoid, ActivationFunctions.elu
        ,
204                         learning_rate, "b", 100)
205
206
207 def task_b_keras():
208     learning_rate = 0.001
209     epochs = 100
210     task = "b"
211     full_network_test_keras(learning_rate, epochs, task)
```

```
212     learning_rate = 0.01
213     full_network_test_keras(learning_rate, epochs, task)
214
215
216 def task_c():
217     learning_rate = 0.001
218     full_network_test(Costfunctions.mse,
            ActivationFunctions.leaky_relu,
219                       ActivationFunctions.elu,
                              learning_rate, "c", epochs=100)
220     full_network_test(Costfunctions.mse,
            ActivationFunctions.linear,
221                       ActivationFunctions.elu,
                              learning_rate, "c", epochs=100)
222     full_network_test(Costfunctions.mse,
            ActivationFunctions.relu,
223                       ActivationFunctions.elu,
                              learning_rate, "c", epochs=100)
224
225
226 def digit_prediction_mse(learning_rate, nodes):
227     n_middle = nodes
228     x_train, x_test, y_train, y_test = prepare_digit_data()
229     n_out = 10
230     n_in = x_train.shape[1]
231     epochs = 2000
232     batch_size = 100
233     l_in = LayerDense(n_in, n_middle, "lin", learning_rate,
            Costfunctions.mse,
234                       ActivationFunctions.elu)
235     l_mi = LayerDense(n_middle, n_middle, "lmi2",
            learning_rate, Costfunctions.mse,
236                       ActivationFunctions.sigmoid)
237     l_ou = LayerDense(n_middle, n_out, "lou", learning_rate
            , Costfunctions.mse,
238                       ActivationFunctions.softmax)
239     network = Network([l_in, l_mi, l_ou], "mnist", [Metrics
            .mse, Metrics.accuracy])
240     network.train(x_train.T, y_train.T, epochs, batch_size,
            x_test.T, y_test.T)
```

```
241     print("Training metric after initialization, after
            first epoch and last epoch")
242     print(network.get_train_met()[0, :], network.
            get_train_met()[1, :],
243           network.get_train_met()[-1, :])
244     print("Testing metric after initialization, after first
            epoch and last epoch")
245     print(network.get_test_met()[0, :], network.
            get_test_met()[1, :],
246           network.get_test_met()[-1, :])
247     print_metrik_by_network_and_epoch(epochs,
248                                       [network.
                                          get_train_met()[:,
                                           0], network.
                                          get_test_met()[:,
                                          0]],
249                                      "d", "mse_own_" + "_"
                                          .join(
250               [network.name, "epochs", str(2000), "lr", str(
                      learning_rate), "nodes", str(n_middle),
251                "batch_size", str(batch_size)]), "Mean squared
                       error")
252     print_metrik_by_network_and_epoch(epochs,
253                                       [network.
                                          get_train_met()[:,
                                           1], network.
                                          get_test_met()[:,
                                          1]],
254                                      "d", "acc_own_" + "_"
                                          .join(
255               [network.name, "epochs", str(2000), "lr", str(
                      learning_rate), "nodes", str(n_middle),
256                "batch_size", str(batch_size)]), "accuracy")
257
258
259 def task_d():
260     digit_prediction_mse(0.0001, 256)
261     digit_prediction_mse(0.0001, 128)
262     digit_prediction_mse(0.0001, 512)
263     digit_prediction_mse(0.001, 256)
```

```
264
265
266 def task_e():
267     x_train, x_test, y_train, y_test = prepare_digit_data()
268     n_out = 10
269     n_in = x_train.shape[1]
270     n_middle = 256
271     epochs = 2000
272     batch_size = 100
273     learning_rate = 0.0001
274
275     l_in = LayerDense(n_in, n_middle, "lin", learning_rate,
            Costfunctions.mse,
276                       ActivationFunctions.elu)
277     l_mi = LayerDense(n_middle, n_middle, "lmi",
            learning_rate, Costfunctions.mse,
278                       ActivationFunctions.sigmoid)
279     l_ou = LayerDense(n_middle, n_out, "lou", learning_rate
            , Costfunctions.cross_entropy,
280                       ActivationFunctions.softmax)
281     network = Network([l_in, l_mi, l_ou], "mnist", [Metrics
            .ce, Metrics.accuracy,
282                                                    Metrics
                                                        .
                                                    ce_grad
                                                    ,
                                                    Metrics
                                                    .mse
                                                    ])
283     network.train(x_train.T, y_train.T, epochs, batch_size,
            x_test.T, y_test.T)
284     print("Metric after initialization, after first epoch
            and last epoch")
285     print(network.get_train_met()[0, :], network.
            get_train_met()[1, :],
286           network.get_train_met()[-1, :])
287     print_metrik_by_network_and_epoch(epochs,
288                                       [network.
                                           get_train_met()[:,
                                            0], network.
```

```
                                                get_test_met()[:,
                                                0]],
289                                           "e", "cre_own_" + "_"
                                                .join(
290             [network.name, "epochs", str(epochs), "lr", str
                  (learning_rate), "nodes", str(n_middle),
291              "batch_size", str(batch_size)]), "entropy")
292     print_metrik_by_network_and_epoch(epochs,
293                                           [network.
                                                get_train_met()[:,
                                                 1], network.
                                                get_test_met()[:,
                                                1]],
294                                           "e", "acc_own_" + "_"
                                                .join(
295             [network.name, "epochs", str(epochs), "lr", str
                  (learning_rate), "nodes", str(n_middle),
296              "batch_size", str(batch_size)]), "accuracy")
297     print_metrik_by_network_and_epoch(epochs,
298                                           [network.
                                                get_train_met()[:,
                                                 2], network.
                                                get_test_met()[:,
                                                2]],
299                                           "e", "ceg_own_" + "_"
                                                .join(
300             [network.name, "epochs", str(epochs), "lr", str
                  (learning_rate), "nodes", str(n_middle),
301              "batch_size", str(batch_size)]), "ent grad")
302     print_metrik_by_network_and_epoch(epochs,
303                                           [network.
                                                get_train_met()[:,
                                                 3], network.
                                                get_test_met()[:,
                                                3]],
304                                           "e", "mse_own_" + "_"
                                                .join(
305             [network.name, "epochs", str(epochs), "lr", str
                  (learning_rate), "nodes", str(n_middle),
306              "batch_size", str(batch_size)]), "mse")
```

```
307
308
309 task_a()
310 task_b_own()
311 task_b_keras()
312 task_c()
313 task_d()
314 task_e()
```

```
1  import numpy as np
2  from typing import List
3  from console_progressbar import ProgressBar
4
5  RANDOM_SEED = 1337
6
7
8  class Metrics:
9      """
10     Class to provide different functions that will be
           accepted by the neural network as a metric.
11     Some of the functions just forward their input to their
           corresponding Costfunctions.
12     Accuracy is only available as a metric, not as a cost
           function.
13     """
14     @staticmethod
15     def __pred_to_class(y):
16         """
17         Returns the predicted class as integer, based on a
               vector of probabilities.
18         :param y: probability vector
19         :return: class with highest probability
20         """
21         r = np.argmax(y, axis=0)
22         return r
23
24     @staticmethod
25     def accuracy(y_true, y_pred):
26         """
27         Calculates the accuracy of the prediction vs the
               true result.
```

```
28          :param y_true: Data to test against
29          :param y_pred: Predicted result
30          :return: probability of agreement
31          """
32          y_true = Metrics.__pred_to_class(y_true)
33          y_pred = Metrics.__pred_to_class(y_pred)
34          return np.sum(y_true == y_pred) / len(y_pred)
35
36      @staticmethod
37      def mse(y_true, y_pred):
38          """
39          Wrapper for mean squared error cost function. The
               cost function accepts batches with n>1,
40          the evaluated metric is therefore the mean over the
               different mses. This is no different
41          than calculating the mse over one large batch where
               all the data is somehow conglomerated.
42          :param y_true: Data to test against
43          :param y_pred: Predicted result
44          :return: Mean squared error metric
45          """
46          return np.sum(Costfunctions.mse(y_true, y_pred,
               None)) / y_pred.shape[1]
47
48      @staticmethod
49      def ce(y_true, y_pred):
50          """
51          Wrapper for cross entropy. Just forwards to CE as
               costfunction and calculates mean over
52          batch.
53          :param y_true: Data to test against
54          :param y_pred: Predicted result
55          :return: Cross Entropy metric
56          """
57          return np.sum(Costfunctions.cross_entropy(y_true,
               y_pred, None)) / y_pred.shape[1]
58
59      @staticmethod
60      def ce_grad(y_true, y_pred):
61          """
```

```
62          Average value of the absolute value of the gradient
                of the cross entropy. Used mostly for
63          debugging and to determin whether the nn has
               converged to a solution.
64          :param y_true: Data to test against
65          :param y_pred: Predicted result
66          :return: Cross entropy gradient metric
67          """
68          y_pred = y_pred + Costfunctions.eps
69          return np.mean(np.abs((y_pred - y_true) / (y_pred *
                (1 - y_pred)) / y_true.shape[0]))
70
71      @staticmethod
72      def coeff_determination(y_true, y_pred):
73          """
74          R^2 value but adapted to fit into my code
75          :param y_true: Data to test against
76          :param y_pred: Predicted result
77          :return: R^2 value metric
78          """
79          r = (np.square(y_true - y_pred)) / np.sum(np.square
               (y_true - np.mean(y_true)))
80          return 1 - np.sum(r)
81
82
83  class Costfunctions:
84      """
85      Class that defines my Costfunctions that are accepted
           by the neural network as valid cost
86      functions. Every cost function needs a gradient
           associated with it.
87      """
88
89      """la is the parameter for ridge regression"""
90      la = 0.1
91      """small value for cross entropy to avoid log(0) or
           division by zero"""
92      eps = 1e-15
93
94      @staticmethod
```

```
95      def mse(y_true, y_pred, l):
96          """
97          Mean squared error
98          :param y_true: Data to test against
99          :param y_pred: Predicted result
100         :param l: sum of values of weights and biases. Not
                used here
101         :return: mean squared error per batch and node
102         """
103         return 1/2*np.sum((y_true - y_pred)**2, axis=1)
104
105     @staticmethod
106     def __mse_grad(y_true, y_pred, l):
107         """
108         Gradient of mean squared error
109         :param y_true: Data to test against
110         :param y_pred: Predicted result
111         :param l: sum of values of weights and biases. Not
                used here
112         :return: gradient of mse per batch and node
113         """
114         return y_pred - y_true
115
116     @staticmethod
117     def ridge(y_true, y_pred, l):
118         """
119         Ridge error based on the parameter la.
120         :param y_true: Data to test against
121         :param y_pred: Predicted result
122         :param l: sum of values of weights and biases
123         :return: ridge error per batch and node
124         """
125         return np.sum((y_true - y_pred)**2) + Costfunctions
                .la*np.sum(l**2)
126
127     @staticmethod
128     def __ridge_grad(y_true, y_pred, l):
129         """
130         Gradient of ridge error
131         :param y_true: Data to test against
```

```
132         :param y_pred: Predicted result
133         :param l: sum of values of weights and biases
134         :return: ridge error gradient per batch and node
135         """
136         return -(y_true - y_pred + Costfunctions.la*l)
137 
138     @staticmethod
139     def cross_entropy(y_true, y_pred, l):
140         """
141         Cross entropy error
142         :param y_true: Data to test against
143         :param y_pred: Predicted result
144         :param l: sum of values of weights and biases. Not
                used here.
145         :return: cross entropy error per batch and node
146         """
147         y_pred = y_pred + Costfunctions.eps
148         return - np.nanmean(y_true * np.log(y_pred) + (1-
                y_true)*np.log(1-y_pred), axis=0)
149 
150     @staticmethod
151     def __cross_entropy_grad(y_true, y_pred, l):
152         """
153         Gradient of cross entropy
154         :param y_true: Data to test against
155         :param y_pred: Predicted result
156         :param l: sum of values of weights and biases. Not
                used here.
157         :return: gradient of cross entropy per batch and
                node
158         """
159         y_pred = y_pred + Costfunctions.eps
160         return (y_pred - y_true)/(y_pred*(1-y_pred))/y_true
                .shape[0]
161 
162 
163 class ActivationFunctions:
164     """
165     Class that defines the activation functions that are
        accepted by the layers my NN consists
```

```
166     out of. Each activation function needs an accompanying
           gradient function.
167     """
168
169     """alpha is the small value for leaky relu"""
170     alpha = 0.02
171     """beta is the value for exponential linear unit"""
172     beta = 1
173
174     @staticmethod
175     def sigmoid(x):
176         """
177         The basic sigmoid activation function
178         :param x: input
179         :return: output
180         """
181         return 1 / (1 + np.exp(-x))
182
183     @staticmethod
184     def __sigmoid_grad(x):
185         """
186         Gradient of the sigmoid function
187         :param x: input
188         :return: output
189         """
190         return ActivationFunctions.sigmoid(x)*(1-
               ActivationFunctions.sigmoid(x))
191
192     @staticmethod
193     def softmax(x):
194         """
195         The softmax function. The result is the probability
               of a class based on the input vector.
196         :param x: input
197         :return: output
198         """
199         z = np.exp(x)
200         return z / np.sum(z, axis=0, keepdims=True)
201
202     @staticmethod
```

```
203     def __softmax_grad(x):
204         """
205         Gradient of the softmax function
206         :param x: input
207         :return: output
208         """
209         z = np.exp(x)
210         a = np.sum(z)
211         return x * (a-x)/a
212 
213     @staticmethod
214     def relu(x):
215         """
216         The ReLU function
217         :param x: input
218         :return: output
219         """
220         return (x > 0)*x
221 
222     @staticmethod
223     def __relu_grad(x):
224         """
225         Gradient of the ReLU function
226         :param x: input
227         :return: output
228         """
229         return (x > 0)*1
230 
231     @staticmethod
232     def elu(x):
233         """
234         The ELU function
235         :param x: input
236         :return: output
237         """
238         return (x > 0) * x + (x < 0) * ActivationFunctions.
                beta * (np.exp(x)-1)
239 
240     @staticmethod
241     def __elu_grad(x):
```

```
242             """
243             Gradient of the ELU function
244             :param x: input
245             :return: output
246             """
247             return (x > 0) * 1 + (x < 0) * np.exp(x)*
                    ActivationFunctions.beta
248
249         @staticmethod
250         def leaky_relu(x):
251             """
252             The leaky ReLU function
253             :param x: input
254             :return: output
255             """
256             return (x > 0) * x + (x < 0) * x *
                    ActivationFunctions.alpha
257
258         @staticmethod
259         def __leaky_relu_grad(x):
260             """
261             Gradient of the leaky ReLU function
262             :param x: input
263             :return: output
264             """
265             return (x > 0) * 1 + (x < 0) * ActivationFunctions.
                    alpha
266
267         @staticmethod
268         def linear(x):
269             """
270             The linear output function
271             :param x: input
272             :return: output
273             """
274             return x
275
276         @staticmethod
277         def __linear_grad(x):
278             """
```

```
279             Gradient of the linear output function
280             :param x: input
281             :return: output
282             """
283             return 1
284 
285 
286 class Layer:
287     """
288     The basic layer class. Other layer classes can be
            abstracted from the basic layer class.
289     It currently provides the sceleton for errors, learning
            rate, inputs and outputs, cost- and
290     activation functions.
291     """
292     def __init__(self, n_outputs, name, learning_rate, cf,
            af):
293         """
294         The constructor for the base layer.
295         :param n_outputs: Amount of neurons in this layer
296         :param name: Name of the layer. Strictly speaking
                not necessary, but makes identification
297             easier
298         :param learning_rate: Learning rate of the layer
299         :param cf: Costfunction of the layer
300         :param af: Activationfunction of the layer
301         """
302         # Initializing the different variables
303         # The final output of the layer
304         self.output = np.zeros((n_outputs, 1))
305         # Output before being passed through the activation
                function
306         self.z = np.array([])
307         # Input of the layer
308         self.x = np.array([])
309         # Error for each neuron, based on backpropagation
310         self.error = np.array([])
311         # Keeping track of the learning rate in case it is
                adapted
312         self.initial_learning_rate = learning_rate
```

```
313             self.learning_rate = learning_rate
314             # variables for RMSProp
315             self.beta = 0.9
316             self.weight_s = 0
317             self.bias_s = 0
318             self.eps = 1e-8
319             self.name = name
320             # Setting cost and activation functions as well as
                    their gradients
321             self.cf = cf
322             self.cf_grad = getattr(Costfunctions, "__"+cf.
                    __name__+"_grad")
323             self.af = af
324             self.af_grad = getattr(ActivationFunctions, "__"+af
                    .__name__+"_grad")
325
326         """
327         The Layer class is used to abstract the actual layers
                from. Depending on the layers,
328         forwards and backwards passes will look differently.
                Because Every layer will have those
329         functions, they are defined here, but each layer will
                overwrite them with their own.
330         """
331
332         def forward(self, x):
333             pass
334
335         def backwards(self, error):
336             pass
337
338         """
339         The activation and cost functions are defined in the
                constructor. Every layer will have one
340         but since they are interchangeable and don't depend on
                the layer, they can be defined here.
341         """
342         def activation(self, x):
343             return self.af(x)
344
```

```
345     def activation_grad(self, x):
346         return self.af_grad(x)
347 
348     def cost_function(self, y, yt, l):
349         return self.cf(y, yt, l)
350 
351     def cost_grad(self, y, yt, l):
352         return self.cf_grad(y, yt, l)
353 
354 
355 class LayerDense(Layer):
356     """
357     The class for a dense layer. It is abstracted from the
          basic layer class.
358     """
359     def __init__(self, n_inputs, n_outputs, name,
          learning_rate, cost, activation):
360         """
361         The constructor passes most of its arguments on to
              the constructor for the base class.
362         :param n_inputs: Amount of inputs into the layer
363         :param n_outputs: Amount of neurons in this layer
364         :param name: Name of the layer. Strictly speaking
              not necessary, but makes identification
365             easier
366         :param learning_rate: Learning rate of the layer
367         :param cost: Costfunction of the layer
368         :param activation: Activationfunction of the layer
369         """
370         Layer.__init__(self, n_outputs, name, learning_rate
              , cost, activation)
371         # initializing weights and biases for the dense
              layer.
372         self.biases = 0.1*np.ones((n_outputs, 1))
373         self.weights = 1*np.random.randn(n_outputs,
              n_inputs)
374         # initializing for RMSProp
375         self.bg = []
376         self.wg = []
377         # keeping track of weights and biases after
```

```
            initialization
378         self.initial_weights = self.weights
379         self.initial_biases = self.biases
380
381     def forward(self, x) -> None:
382         """
383         Calculates the forwards pass for a dense layer. It
                does not return anything, but saves
384         the result as the output of the layer.
385         :param x: input
386         :return: None
387         """
388         self.x = x
389         self.z = np.dot(self.weights, self.x) + self.biases
390         self.output = self.activation(self.z)
391
392     def __weight_grad(self):
393         """
394         Calculates the gradient of the weights for
                backpropagation, based on the error returned
395         from the previous layer and the values of its
                current inputs.
396         :return: Gradient of the weights
397         """
398         return np.matmul(self.error, self.x.T)
399
400     def __bias_grad(self):
401         """
402         Calculates the gradient of the biases for
                backpropagation, based on the error returned
403         from the previous layer and the values of its
                current inputs.
404         :return: Gradient of the biases
405         """
406         return np.expand_dims(np.sum(self.error, axis=1),
                axis=1)
407
408     def backwards(self, error):
409         """
410         Calculates the backwards pass for the layer
```

```
411             :param error: The error backpropagated from the
                    previous layer or the cost function in
412             case this is the output layer
413             :return: The error that needs to be propagated
                    backwards to the next layer
414             """
415             # saving the current error for this layer and
                    calculating the backpropagated error
416             self.error = error
417             next_error = np.matmul(self.weights.T, self.error)
                    * self.activation_grad(self.x)
418             # calculating gradients for weights and biases
419             b_grad = self.__bias_grad()
420             w_grad = self.__weight_grad()
421
422             # Applying RMSProp to update weights and biases
423             self.bias_s = self.bias_s * self.beta + (1-self.
                    beta)*b_grad**2
424             self.weight_s = self.weight_s * self.beta + (1-self
                    .beta)*w_grad**2
425
426             self.bg = b_grad / np.sqrt(self.bias_s + self.eps)
427             self.wg = w_grad / np.sqrt(self.weight_s + self.eps
                    )
428
429             self.biases = self.biases - self.learning_rate *
                    self.bg
430             self.weights = self.weights - self.learning_rate *
                    self.wg
431             return next_error
432
433
434 class Network:
435     """
436     The class that fully defines a neural network
437     """
438     def __init__(self, layers: List[Layer], name: str, mf:
                []):
439         """
440         The constructor for the neural network. It takes a
```

```
441             list of already initialized layers as
                an input, where the first layer in the list is the
                    input layer and the last layer is the
442             output layer. The constructor does not perform any
                    checking for errors, so it will just
443             break if the amount of outputs of one layer are not
                     equal to the amount of inputs for the
444             next layer.
445             Due to the way the Network is built, it will be
                    possible to change layers, activation or
446             cost functions, inspect values within the layers of
                     the network or just continue training
447             the network after a certain stage.
448             The network also takes in a list of metric
                    functions. Before training begins and after
449             each epoch, the metric functions will be applied to
                     training and testing data,
450             to keep track of how the neural network performs.
451             :param layers: The layers the neural network
                    consists of.
452             :param name: Name of the network
453             :param mf: List of metric functions as defined in
                    the above class
454             """
455             self.layers = layers
456             self.train_M = None
457             self.test_M = None
458             self.name = name
459             self.mf = mf
460             # initializing epoch count
461             self.epoch = 0
462             np.random.seed(RANDOM_SEED)
463
464         def __forward_pass(self, x):
465             """
466             A full forwards pass through the whole network. The
                     outputs of the network are stored in
467             each layer.
468             :param x: input of the first layer
469             :return: Output of the last layer
```

```
470             """
471             for layer in self.layers:
472                 layer.forward(x)
473                 x = layer.output
474             return x
475
476         def predict(self, x):
477             """
478             Public function that fulfills the same
                    functionality as the forward pass
479             :param x: input of the first layer
480             :return: Output of the last layer
481             """
482             return self.__forward_pass(x)
483
484         def __backward_pass(self, error):
485             """
486             Goes backwards through the layers and calls the
                    backwards function for each layer,
487             with the error calculated from the previous layer
488             :param error: initial error for the output layer
489             :return: None
490             """
491             for layer in reversed(self.layers):
492                 error = layer.backwards(error)
493
494         def __inc_epoch(self):
495             """
496             Increases the epoch count
497             :return: None
498             """
499             self.epoch += 1
500
501         def train(self, x, y, epochs, batch_size, x_test,
                y_test):
502             """
503             Training the neural networ
504             :param x: input vector of training data
505             :param y: output vector of training data
506             :param epochs: number of epochs to be trained
```

```
507             :param batch_size: size of batches the training
                    data is split into
508             :param x_test: input vector of testing data
509             :param y_test: output vector of testing data
510             :return: None
511             """
512             batches = int(np.floor(x.shape[1] / batch_size))
513             p = 0
514             pb = ProgressBar(total=epochs, prefix='', suffix=''
                    , decimals=3,
515                              length=50, fill='=',
516                              zfill='>')
517             # Calculating metrics for training and testing data
                    before training the network
518             self.__app_metrics(x, x_test, y, y_test)
519             pb.print_progress_bar(p)
520             # going through batches and epochs
521             for i in range(epochs):
522                 for j in range(batches):
523                     xn = x[:, j * batch_size:(j + 1) *
                            batch_size]
524                     yn = y[:, j * batch_size:(j + 1) *
                            batch_size]
525                     self.__forward_pass(xn)
526                     l = (np.sum(np.abs(self.layers[-1].weights)
                            , axis=1) + np.abs(self.layers[-1].
                            biases.T)).T
527                     self.__backward_pass(self.layers[-1].
                            cost_grad(yn, self.layers[-1].output, l)
                            )
528                 # Calculating metrics are every epoch
529                 self.__app_metrics(x, x_test, y, y_test)
530                 self.__inc_epoch()
531                 p += 1
532                 pb.print_progress_bar(p)
533
534     def adapt_learning_rate(self, lr):
535         """
536         Adapts the learning rate for every layer
537         :param lr: new learning rate
```

```
538             :return: None
539             """
540             for layer in self.layers:
541                 layer.learning_rate = lr
542 
543         def __metric(self, y_true, y_pred):
544             """
545             Calculates every metric in the previously defined
                    list of metrics for the given data
546             :param y_true: Data to test against
547             :param y_pred: Predicted result
548             :return: The result for every metric
549             """
550             ms = []
551             if self.mf is None:
552                 return ms
553             for m in self.mf:
554                 ms.append(m(y_true, y_pred))
555             return np.expand_dims(np.array(ms).T, axis=1)
556 
557         def __app_metrics(self, x_train, x_test, y_train,
                y_test):
558             """
559             Calculates metrics for test and training data and
                    appends to the preinitialized list
560             :param x_train: training input
561             :param x_test: testing input
562             :param y_train: known true result for training data
563             :param y_test: known true result for testing data
564             :return:
565             """
566             if self.mf is None:
567                 return
568             if self.train_M is None:
569                 self.train_M = self.__metric(y_train, self.
                        predict(x_train))
570             else:
571                 self.train_M = np.append(self.train_M, self.
                        __metric(y_train, self.predict(x_train)),
572                                          axis=1)
```

```
573             if self.test_M is None:
574                 self.test_M = self.__metric(y_test, self.
                        predict(x_test))
575             else:
576                 self.test_M = np.append(self.test_M, self.
                        __metric(y_test, self.predict(x_test)), axis
                        =1)
577 
578     def get_train_met(self):
579         """
580         :return: metrics for training data
581         """
582         return self.train_M.T
583 
584     def get_test_met(self):
585         """
586         :return: metrics for testing data
587         """
588         return self.test_M.T
```

```
1  import sys
2  from typing import Tuple
3  
4  import matplotlib.pyplot as plt
5  import numpy as np
6  import pandas as pd
7  import itertools as it
8  
9  from console_progressbar import ProgressBar
10 from matplotlib.figure import Figure
11 from sklearn.preprocessing import scale
12 from sklearn.utils import shuffle
13 
14 import colormaps as cmaps
15 
16 
17 def franke(x: np.array, y: np.array, noise_level: float) ->
       np.array:
18     """
19     Calculates the value of the franke function for x and y
           coordinates and adds random noise based
```

```
20      on the noise level
21      :param x: x-coordinate
22      :param y: y-coordinate
23      :param noise_level: noise level
24      :return: Values of the franke function for the x-y
            pairs
25      """
26      if x.any() < 0 or x.any() > 1 or y.any() < 0 or y.any()
            > 1:
27          # Breaks if the interval of x and y is outside
                [0,1]
28          print("Franke function is only valid for x and y
                between 0 and 1.")
29          return None
30      term1 = 0.75 * np.exp(-(0.25 * (9 * x - 2) ** 2) - 0.25
            * ((9 * y - 2) ** 2))
31      term2 = 0.75 * np.exp(-((9 * x + 1) ** 2) / 49.0 - 0.1
            * (9 * y + 1))
32      term3 = 0.5 * np.exp(-(9 * x - 7) ** 2 / 4.0 - 0.25 *
            ((9 * y - 3) ** 2))
33      term4 = -0.2 * np.exp(-(9 * x - 4) ** 2 - (9 * y - 7)
            ** 2)
34      noise = np.random.normal(0, 1, term4.shape)
35      return term1 + term2 + term3 + term4 + noise_level *
            noise
36
37
38  def r2_error(y_data: np.array, y_model: np.array) -> float:
39      """
40      Calculates R squared based on predicted and true data.
41      """
42      r = ((y_data - y_model) ** 2) / np.sum((y_data - np.
            mean(y_data)) ** 2)
43      return 1 - np.sum(r, axis=0)
44
45
46  def mse_error(y_data: np.array, y_model: np.array) -> float
        :
47      """
48      Calculates mean squared error based in predicted and
```

```
        true data
49      """
50      r = (y_data - y_model) ** 2
51      return np.sum(r, axis=0) / np.size(y_model, 0)
52
53
54  def coords_to_polynomial(x: np.array, y: np.array, p: int)
        -> np.array:
55      """
56      Calculates the feature matrix based on input
            coordinates x and y and the given degree of the
57      polynomial
58      :param x: x vector
59      :param y: y vector
60      :param p: degree of polynomial
61      :return: feature matrix
62      """
63      if x.shape != y.shape:
64          # Breaks if x and y are of different shapes
65          print("mismatch between size of x and y vector")
66          return None
67      # Preparing x and y and pre-allocating the feature
            matrix X
68      x = x.flatten()
69      y = y.flatten()
70      feat_matrix = np.zeros(shape=(len(x), max_mat_len(p)))
71      k = 0
72      for i in range(p + 1):
73          for j in range(i + 1):
74              feat_matrix[:, k] = x ** (i - j) * y ** j
75              k += 1
76      return feat_matrix
77
78
79  def solve_lin_equ(y: np.array, x: np.array, solver: str = "
        ols", epochs: int = 0, batches: int = 0,
80                    la: float = 1, g0: float = 1e-3) -> Tuple
                          [np.array, np.array]:
81      """
82      Solves linear equation of type y = x*beta. This can be
```

```
        done using ordinary least squares (OLS),
83      ridge or lasso regression. For ridge and lasso, an
        additional parameter l for shrinkage /
84      normalization needs to be provided.
85      :param y: solution vector of linear problem
86      :param x: feature vector or matrix of problem
87      :param solver: solver. Can be ols, ridge or lasso
88      :param la: shrinkage / normalization parameter for
        ridge or lasso
89      :param g: learning rate factor gamma
90      :return: parameter vector that solves the problem &
        variance of parameter vector
91      """
92      if epochs <= 0 or batches <= 0:
93          print("epoch or batch data invalid")
94          sys.exit(-1)
95      beta = np.random.random(size=(x.shape[1], epochs))
96      cost = np.zeros(epochs)
97      g = g0
98      bet = 0.9
99      s = 0
100     eps = 1e-8
101     bl = int(np.floor(y.shape[0] / batches))
102     currb = beta[:, 0]
103     for e in range(epochs):
104         for b in range(batches):
105             yb = y[b * bl:(b + 1) * bl - 1]
106             xb = x[b * bl:(b + 1) * bl - 1]
107             c = cost_function(yb, xb, currb, solver, la)
108             gd = cost_grad(yb, xb, currb, solver, la)
109             s = bet * s + (1 - bet) * gd ** 2
110             currb = currb - g * gd / (np.sqrt(s + eps))
111         beta[:, e] = currb
112         cost[e] = c
113     return beta, cost
114
115
116 def cost_function(y: np.array, x: np.array, beta: np.array,
    solver: str = "ols",
117                   la: float = 1) -> float:
```

```
118     """
119     Calculates cost function for ordinary least squares or
            ridge regression
120     :param y: desired output
121     :param x: feature matrix
122     :param beta: parameters
123     :param solver: solver. Can be ols or ridge
124     :param la: regression parameter for ridge regression
125     :return: Cost value
126     """
127     if solver == "ols":
128         return 1 / 2 * np.sum((y - x @ beta) ** 2)
129     elif solver == "ridge":
130         r = 1 / 2 * np.sum((y - x @ beta) ** 2) + 1 / 2 *
                la * np.sum(beta ** 2)
131         return r
132     elif solver == "logistic":
133         return -np.sum(y * np.log(x @ beta) + (1 - y) * np.
                log(1 - x @ beta))
134
135
136 def cost_grad(y: np.array, x: np.array, beta: np.array,
        solver: str = "ols",
137               la: float = 1) -> np.array:
138     """
139     Calculates gradient of cost function in parameter
            direction
140     :param y: desired output
141     :param x: feature matrix
142     :param beta: parameters
143     :param solver: solver. Can be ols or ridge
144     :param la: regression parameter for ridge regression
145     :return: gradient of cost function
146     """
147     if solver == "ols":
148         return - (y - x @ beta) @ x
149     elif solver == "ridge":
150
151         res = - (y - x @ beta) @ x + la * np.abs(beta)
152         if np.isnan(np.sum(res)):
```

```
153                 print(y)
154                 print(x)
155                 print(beta)
156                 sys.exit()
157             return res
158         elif solver == "logistic":
159             return (x @ beta - y) / (x @ beta - x @ beta ** 2)
160 
161 
162 def create_grid(res: float) -> Tuple[np.array, np.array]:
163     """
164     Creates a parameter grid for x and y in [0,1] with a
            given resolution
165     :param res: resolution of the grid
166     :return: Parameter grid
167     """
168     x = np.arange(0, 1, res)
169     y = np.arange(0, 1, res)
170     x, y = np.meshgrid(x, y)
171     return x, y
172 
173 
174 def create_data(deg: int, x: np.array = None, y: np.array =
        None, z: np.array = None) \
175             -> Tuple[np.array, np.array, np.array, np.array]:
176     """
177     Creates the data that is used for the regression.
            Depending on the given degree of the
178     polynomial a feature matrix will be created. Also data
            for the franke function for this feature
179     matrix is calculated. If cross validation is to be
            performed, the parameter with a value larger
180     than one has to be set. For a value equal 1, the data
            is split into 80:20 training and test
181     data. For cross validation, the data will be split into
             equal folds depending on the number
182     given.
183     If values for x,y and z are provided, no feature matrix
             is created but instead the given values
184     are passed on. The function also performs scaling of
```

```
        the data, if necessary.
185     :param deg: Degree of the polynomial
186     :param x: x-vector
187     :param y: y-vector
188     :param z: z-vector
189     :return: arrays containing feature and result vectors
            split into training and test
190     """
191     # creating coordinate vectors if none were provided
192     if x is None or y is None:
193         x, y = create_grid(RESOLUTION)
194         z = franke(x, y, NOISE_LEVEL).flatten()
195     # calculating feature matrix
196     feature_mat = coords_to_polynomial(x, y, deg)
197     # scaling
198     if SCALE_DATA:
199         feature_mat = scale(feature_mat, axis=1)
200     # calculation z-vector if none was provided
201     if z is None:
202         z = franke(x, y, NOISE_LEVEL).flatten()
203     # dimensions of the feature matrix
204     n = int(feature_mat.shape[0])
205     # test_l is the length of the test split
206     # 80:20 split in train and test data
207     test_l = int(np.floor(n * 0.2))
208
209     # randomizing order of feature matrix and result vector
            and allocating split variables
210     feature_mat, z = shuffle(feature_mat, z, random_state=
            RANDOM_SEED)
211     feature_mat_train = np.delete(feature_mat, np.arange(0,
            test_l, 1, int), axis=0)
212     feature_mat_test = feature_mat[:test_l, :]
213     z_train = np.delete(z, np.arange(0, test_l, 1, int),
            axis=0)
214     z_test = z[:test_l]
215     z_test = np.expand_dims(z_test, 1)
216     z_train = np.expand_dims(z_train, 1)
217     # reshaping to make it easier in the coming functions.
            The first two indeces are left in their
```

```
218     # previous relative and are the length and height of
            the features. The now last index represents
219     # the fold.
220     return feature_mat_train, feature_mat_test, z_train,
            z_test
221
222
223 def predict(beta: np.array, feature_mat: np.array) -> np.
    array:
224     """
225     Predicts result of linear equation: y = x*beta
226     :param beta: parameter matrix
227     :param feature_mat: feature matrix
228     :return:
229     """
230     return feature_mat @ beta
231
232
233 def err_from_var(var: np.array, sample_size: int) -> np.
    array:
234     """
235     Calculates the error from its variance
236     :param var: variance
237     :param sample_size: Sample size
238     :return: error based on variance and sample size
239     """
240     return 1.97 * np.sqrt(var) / np.sqrt(sample_size)
241
242
243 def train(deg: int, solver: str = "ols", la: float = 1,
    epochs: int = 0, batches: int = 0,
244           x: np.array = None, y: np.array = None, z: np.
                array = None, g0: float = 1e-3) \
245         -> Tuple[np.array, np.array, np.array, np.array, np
              .array, np.array, np.array]:
246     """
247     Calculates fit based on given degree. If no data for x,
            y and z are given, data will be
248     created from / for the franke function, otherwise the
            provided data will be used
```

```
249     :param deg: Degree of the polynomial to be fitted onto
250     :param solver: Algorithm for solving. Can be ols, ridge
            or lasso
251     :param la: parameter for ridge and lasso regression
252     :param x: x-vector
253     :param y: y-vector
254     :param z: z-vector
255     :return: Regression parameter vector, error and
            variance of parameter vector, test & training
256          data R squqared value and mean squared error
257     """
258     # creates feature matrix and result vector
259     train_x, test_x, train_z, test_z = create_data(deg, x=x
            , y=y, z=z)
260     # declares variables for results
261     # solving the linear equation, calculating error for
            parameter vector
262     beta, cost = solve_lin_equ(train_z.flatten(), train_x,
            solver=solver, epochs=epochs, batches=batches,
263                                la=la, g0=g0)
264     # Calculate errors based on the chosen samples
265     test_r = r2_error(test_z, predict(beta, test_x))
266     test_m = mse_error(test_z, predict(beta, test_x))
267     train_r = r2_error(train_z, predict(beta, train_x))
268     train_m = mse_error(train_z, predict(beta, train_x))
269     test_cost = cost_function(test_z, test_x, beta, solver=
            solver, la=la)
270     return test_r, train_r, test_m, train_m, beta, cost,
            test_cost
271
272
273 def max_mat_len(maxdeg: int) -> int:
274     """
275     Calculates the size of the polyonmial based on its
            degree
276     :param maxdeg: degree of polynomial
277     :return: size of the polynomial
278     """
279     return int((maxdeg + 2) * (maxdeg + 1) / 2)
280
```

```
281
282 def train_degs(maxdeg: int, solver: str = "ols",
283                 la: float = 1, epochs: int = 0, batches: int
                        = 0,
284                 x: np.array = None, y: np.array = None, z:
                        np.array = None, g0: float = 1e-3) \
285         -> Tuple[np.array, np.array, np.array, np.array, np
                .array, np.array, np.array]:
286     """
287     This is a wrapper for the function "train", that loops
            over all the degrees given in maxdeg
288     :param maxdeg: highest order degree to be trained
289     :param solver: Algorithm for solving. Can be ols, ridge
            or lasso
290     :param la: parameter for ridge and lasso regression
291     :param x: x-vector
292     :param y: y-vector
293     :param z: z-vector
294     :return: Regression parameter vector, error and
            variance of parameter vector, test & training
295         data R squqared value and mean squared error
296     """
297     # declare variables so that the values can be assigned
            in the loop
298     beta = np.zeros(shape=(maxdeg, max_mat_len(maxdeg),
            epochs))
299     test_r = np.zeros(shape=(maxdeg, epochs))
300     train_r = np.zeros(shape=(maxdeg, epochs))
301     test_m = np.zeros(shape=(maxdeg, epochs))
302     train_m = np.zeros(shape=(maxdeg, epochs))
303     test_cost = np.zeros(shape=(maxdeg, epochs))
304     train_cost = np.zeros(shape=(maxdeg, epochs))
305     # looping over every degree to be trained
306     for i in range(maxdeg):
307         print("training degree: " + str(i + 1))
308         t = train(i + 1, solver=solver, epochs=epochs,
                batches=batches, la=la, x=x, y=y, z=z, g0=g0)
309         b = t[4]
310         beta[i, 0:len(b)] = b
311         test_r[i, :] = t[0]
```

```
312             train_r[i, :] = t[1]
313             test_m[i, :] = t[2]
314             train_m[i, :] = t[3]
315             train_cost[i, :] = t[5]
316             test_cost[i, :] = t[6]
317         return test_r, train_r, test_m, train_m, beta,
                train_cost, test_cost
318
319
320 def print_errors(x_values: np.array, errors: np.array,
        labels: list, name: str, logy: bool = False,
321                  logx: bool = False,
322                  xlabel: str = "Degree", ylabel: str = "
                         error value", d: int = 6, task: str = "
                         a") -> Figure:
323     """
324     Helper function to create similar looking graphs. All
            the graphs where mean squared errors or
325     the r_squared value are plotted and shown in the report
             are plotted using this function
326     :param x_values: values for x axis
327     :param errors: values for y axis
328     :param labels: plot labels
329     :param name: filename
330     :param logy: plot y logarithmically?
331     :param logx: plot x logarithmically?
332     :param xlabel: label for x
333     :param ylabel: label for y
334     :param d: size of plot
335     :return: the created figure
336     """
337     # creating new figure
338     fig = plt.figure(figsize=(d, d), dpi=300)
339     # plotting every given value
340     for i in range(len(errors)):
341         label = labels[i]
342         # linestyle depends on test or train data
343         if label.__contains__("train"):
344             linestyle = "--"
345         else:
```

```
346             linestyle = "_"
347         plt.plot(x_values, errors[i], label=labels[i],
                linestyle=linestyle)
348     plt.legend()
349     plt.grid()
350     ax = fig.gca()
351     ax.set_xlabel(xlabel)
352     ax.set_ylabel(ylabel)
353     if logy:
354         plt.yscale('log')
355     if logx:
356         plt.xscale('log')
357     fig.tight_layout()
358     fig.savefig("images/" + task + "/" + name + ".png", dpi
            =300)
359     plt.close()
360     return fig
361
362
363 def print_cont(x: np.array, y: np.array, data: np.array,
        name: str, x_label: str = "", y_label: str = "", z_label
        : str = "", task: str = "a") -> Figure:
364     """
365     Similar as above but for contour data where the axes
            depend on the degree of the polynome
366     :param x:
367     :param y:
368     :param data: z-values
369     :param name: filename
370     :param z_label: label for z axis
371     :return: the create figure
372     """
373     x, y = np.meshgrid(x, y)
374     fig = plt.figure(figsize=(6, 6), dpi=300)
375     ax = fig.gca(projection='3d')
376     ax.plot_surface(x, y, data, cmap=cmaps.parula)
377     ax.set_xlabel(x_label)
378     ax.set_ylabel(y_label)
379     ax.set_zlabel(z_label)
380     fig.tight_layout()
```

```
381     fig.savefig("images/" + task + "/" + name + ".png", dpi
            =300)
382     return fig
383
384
385 def read_data_file() -> pd.DataFrame:
386     """
387     Reads file with own data into pandas data frame
388     :return: File content as dataframe
389     """
390     data_file = DATA_FILE
391     return pd.read_csv(data_file, delimiter=",")
392
393
394 def get_data() -> Tuple[np.array, np.array, np.array]:
395     """
396     I used some data that I had left over from my previous
            position. This is calculated magnetic
397     field data from some permanent magnet configuration.
            The data does not follow a strictly
398     integer-polynomial trend but can be described not too
            badly by a polynom within the boundaries.
399     I made the data more sparse by randomly removing 80% of
             the original data.
400     :return: prepared data
401     """
402     data = read_data_file()
403     x = np.array(data["x"])
404     y = np.array(data["y"])
405     z = np.array(data["B"])
406     cutoff = int(np.floor(len(z) * .2))
407     x, y, z = shuffle(x, y, z, random_state=RANDOM_SEED)
408     x = x[:cutoff]
409     y = y[:cutoff]
410     z = z[:cutoff]
411     z = z / np.max(z)
412     return x, y, z
413
414
415 def print_data() -> Figure:
```

```
416         """
417         Prints my data to image file
418         :return: The created figure
419         """
420         x, y, z = get_data()
421         fig = plt.figure(figsize=(8, 8), dpi=300)
422         ax = fig.gca(projection='3d')
423         ax.scatter(x, y, z, c='y', marker='o')
424         ax.set_xlabel("x")
425         ax.set_ylabel("y")
426         ax.set_zlabel("B")
427         fig.savefig("images/data.png", dpi=300)
428         return fig
429
430
431 def train_print_hyperparameter(deg: int, parameter: [str],
        solver: str = "ridge", x: np.array = None, y: np.array =
        None,
432                                z: np.array = None, epochv:
                                    [int] = np.array([0]),
                                    batchv: [int] = np.array
                                    ([0]), lambdav: [float] =
                                     np.array([1]),
433                                lrv: [float] = np.array([1e
                                    -3]), task: str = "a", d:
                                     int = 6) -> None:
434         """
435         Wrapper function around "train_degs", that is able to
              handle different parameters for ridge and
436         lasso and automatically prints the resulting diagrams.
437         :param deg:
438         :param parameter:
439         :param values:
440         :param solver:
441         :param x:
442         :param y:
443         :param z:
444         :param epochv:
445         :param batchv:
446         :param lambdav:
```

```
447         :param lrv:
448         :param task:
449         :param d:
450         :return:
451         """
452 
453         """batchv = np.array([])
454         epochv = []
455         lambdav= []
456         lrv = []"""
457         #lrv = np.array([1e-3, 2e-3])
458         ba_n = batchv.size
459         ep_n = epochv.size
460         la_n = lambdav.size
461         lr_n = lrv.size
462         b_c = 0
463         lr_c = 0
464         la_c = 0
465         N = ba_n*la_n*lr_n
466         test_r = np.zeros(shape=(ep_n, ba_n, lr_n, la_n))
467         train_r = np.zeros(shape=(ep_n, ba_n, lr_n, la_n))
468         test_m = np.zeros(shape=(ep_n, ba_n, lr_n, la_n))
469         train_m = np.zeros(shape=(ep_n, ba_n, lr_n, la_n))
470         beta = np.zeros(shape=(max_mat_len(deg), ep_n, ba_n,
                lr_n, la_n))
471         cost = np.zeros(shape=(ep_n, ba_n, lr_n, la_n))
472         test_cost = np.zeros(shape=(ba_n, lr_n, la_n))
473 
474         print("testing " + str(N) + " combinations of
                parameters for " + str(ep_n) + " epochs")
475         e = ep_n
476         i = 0
477         pb = ProgressBar(total=N, prefix='', suffix='',
                decimals=3,
478                          length=50, fill='=',
479                          zfill='>')
480         pb.print_progress_bar(i)
481         for b, l, s in it.product(batchv, lrv, lambdav):
482             p1, p2, p3, p4, p5, p6, p7 = train(deg=deg, solver=
                    solver, la=s, epochs=e, batches=b, x=x,
```

```
                                                    y=y, z=z, g0=l)
        test_r[:, b_c, lr_c, la_c] = p1
        train_r[:, b_c, lr_c, la_c] = p2
        test_m[:, b_c, lr_c, la_c] = p3
        train_m[:, b_c, lr_c, la_c] = p4
        beta[:, :, b_c, lr_c, la_c] = p5
        cost[:, b_c, lr_c, la_c] = p6
        test_cost[b_c, lr_c, la_c] = p7
        la_c += 1
        if la_c == la_n:
            la_c = 0
            lr_c += 1
            if lr_c == lr_n:
                lr_c = 0
                b_c += 1
        i += 1
        pb.print_progress_bar(i)
    ref = ["epoch", "batch", "learning_rate", "lambda"]
    if len(parameter) == 1:
        parameter = parameter[0]
        i = ref.index(parameter)
        if i == 0:
            values = np.linspace(1, ep_n, ep_n, dtype=int)
        elif i == 1:
            values = batchv
        elif i == 2:
            values = lrv
        elif i == 3:
            values = lambdav
        if i != 0:
            errors_m = np.transpose(np.squeeze(np.append(np
                .moveaxis(test_m[-1, :, :, :], i-1, 0)[:],
                np.moveaxis(train_m[-1, :, :, :], i-1, 0)
                [:], axis=1)))
            errors_r = np.transpose(np.squeeze(np.append(np
                .moveaxis(test_r[-1, :, :, :], i-1, 0)[:],
                np.moveaxis(train_r[-1, :, :, :], i-1, 0)
                [:], axis=1)))
        else:
            errors_m = np.transpose(np.squeeze(np.append(
```

```
                test_m[:], train_m[:], axis=1)))
517             errors_r = np.transpose(np.squeeze(np.append(
                test_r[:], train_r[:], axis=1)))
518         labels_m = ["test MSE", "train MSE"]
519         labels_r = ["test R^2", "train R^2"]
520         # plotting mean squared error
521         if parameter == "lambda" or parameter == "
            learning_rate":
522             logx = True
523         else:
524             logx = False
525         print_errors(values, errors_m, labels_m,
526                      solver + "_mse_parameter_" + parameter
                         , logx=logx, logy=True,
527                      xlabel=parameter, d=d, task=task,
                         ylabel="mean squared error")
528         print_errors(values, errors_r, labels_r,
529                      solver + "_r_squared_parameter_" +
                         parameter, logx=logx, logy=True,
530                      xlabel=parameter, d=d, task=task,
                         ylabel="R^2 value")
531     elif len(parameter) == 2:
532         par1 = parameter[0]
533         par2 = parameter[1]
534         i = ref.index(par1)
535         val1 = []
536         if i == 0:
537             val1 = np.linspace(1, ep_n, ep_n, dtype=int)
538         elif i == 1:
539             val1 = batchv
540         elif i == 2:
541             val1 = lrv
542         elif i == 3:
543             val1 = lambdav
544         j = ref.index(par2)
545         val2 = []
546         if j == 0:
547             val2 = np.linspace(1, ep_n, ep_n, dtype=int)
548         elif j == 1:
549             val2 = batchv
```

```
550             elif j == 2:
551                 val2 = lrv
552             elif j == 3:
553                 val2 = lambdav
554             if i != 0 and j != 0:
555                 data_m_train = np.squeeze(np.moveaxis(np.
                        moveaxis(train_m[-1, :, :, :], j-1, 0), i-1,
                         0), axis=(2))
556                 data_m_test = np.squeeze(np.moveaxis(np.
                        moveaxis(test_m[-1, :, :, :], j-1, 0), i-1,
                        0), axis=(2))
557                 data_r_train = np.squeeze(np.moveaxis(np.
                        moveaxis(train_r[-1, :, :, :], j-1, 0), i-1,
                         0), axis=(2))
558                 data_r_test = np.squeeze(np.moveaxis(np.
                        moveaxis(test_r[-1, :, :, :], j-1, 0), i-1,
                        0), axis=(2))
559             else:
560                 data_m_train = np.squeeze(np.moveaxis(np.
                        moveaxis(train_m[:, :, :, :], j, 0), i, 0),
                        axis=(2, 3))
561                 data_m_test = np.squeeze(np.moveaxis(np.
                        moveaxis(test_m[:, :, :, :], j, 0), i, 0),
                        axis=(2, 3))
562                 data_r_train = np.squeeze(np.moveaxis(np.
                        moveaxis(train_r[:, :, :, :], j, 0), i, 0),
                        axis=(2, 3))
563                 data_r_test = np.squeeze(np.moveaxis(np.
                        moveaxis(test_r[:, :, :, :], j, 0), i, 0),
                        axis=(2, 3))
564             print_cont(x=val2, y=val1, data=data_m_train, name=
                    "_".join(parameter), z_label="train_Data_mse",
                    x_label=par2, y_label=par1, task=task)
565             print_cont(x=val2, y=val1, data=data_m_test, name="
                    _".join(parameter), z_label="test_Data_mse",
                    x_label=par2, y_label=par1, task=task)
566             print_cont(x=val2, y=val1, data=data_r_train, name=
                    "_".join(parameter), z_label="
                    train_Data_r_squared", x_label=par2, y_label=
                    par1, task=task)
```

```
567             print_cont(x=val2, y=val1, data=data_r_test, name="
                    _".join(parameter), z_label="test_Data_r_squared
                    ", x_label=par2, y_label=par1, task=task)
568         else:
569             pass
570         min_ind_train = np.unravel_index(np.argmin(train_m),
                train_m.shape)
571         min_ind_test = np.unravel_index(np.argmin(test_m),
                test_m.shape)
572         print("training mean squared error:" + str(train_m[
                min_ind_train]))
573         print("epochs: " + str(epochv[min_ind_train[0]]))
574         print("batches: " + str(batchv[min_ind_train[1]]))
575         print("learning rate: " + str(lrv[min_ind_train[2]]))
576         print("lambda: " + str(lambdav[min_ind_train[3]]))
577         print("testing mean squared error:" + str(test_m[
                min_ind_test]))
578         print("epochs: " + str(epochv[min_ind_test[0]]))
579         print("batches: " + str(batchv[min_ind_test[1]]))
580         print("learning rate: " + str(lrv[min_ind_test[2]]))
581         print("lambda: " + str(lambdav[min_ind_test[3]]))
582         print("Cost function for this training run: "+str(cost[
                min_ind_train[0],min_ind_train[1],min_ind_train[2],
                min_ind_train[3]]))
583
584
585         print("class by cost function")
586         min_ind_train = np.unravel_index(np.argmin(cost), cost.
                shape)
587         min_ind_test = np.unravel_index(np.argmin(test_cost),
                test_cost.shape)
588         print("training cost function:" + str(cost[
                min_ind_train]))
589         print("epochs: " + str(epochv[min_ind_train[0]]))
590         print("batches: " + str(batchv[min_ind_train[1]]))
591         print("learning rate: " + str(lrv[min_ind_train[2]]))
592         print("lambda: " + str(lambdav[min_ind_train[3]]))
593         print("testing cost function:" + str(test_cost[
                min_ind_test]))
594         print("epoch: "+ str(ep_n))
```

```
595     print("batches: " + str(batchv[min_ind_test[0]]))
596     print("learning rate: " + str(lrv[min_ind_test[1]]))
597     print("lambda: " + str(lambdav[min_ind_test[2]]))
598     print("Mean squared error: "+str(train_m[min_ind_test
            [0],min_ind_train[1],min_ind_train[2],min_ind_train
            [3]]))
599
600
601 def train_print_single_lambda(deg: int, epochs: int,
        batches: int, la: float = 0,
602                               solver: str = "ridge", x: np.
                                  array = None, y: np.array
                                  = None,
603                               z: np.array = None, task: str
                                   = "a", d: int = 6):
604     """
605     Same as "train_print_diff_lambdas" but only for a
            single lambda
606     :param deg: maximum degree for which to train
607     :param batches:
608     :param epochs:      :param la: lambda for which to plot
609     :param solver: ridge or lasso
610     :param x: x vector
611     :param y: y vector
612     :param z: z vector
613     :param task: task this belongs to, to sort the figures
            into the correct folder
614     :param d:
615     :return: None
616     """
617     if task == "f" or task == "e":
618         logy = False
619     else:
620         logy = True
621     test_r, train_r, test_m, train_m, beta, cost, test_cost
            = train_degs(maxdeg=deg, solver=solver, la=la, x=x,
            y=y,
622                                                          z=z
                                                          ,
```

```
                                                                epochs
                                                                =
                                                                epochs
                                                                ,
                                                                batches
                                                                =
                                                                batches
                                                                )
623     errors = [test_m[:, -1], train_m[:, -1]]
624     labels = ["test MSE", "train MSE"]
625     print_errors(np.linspace(1, deg, deg), errors, labels,
626                  solver + "_mse_deg" + str(deg) + "_epochs_
                         " +
627                  str(epochs) + "_batches_" + str(batches) +
                         "_lambda_" + str(la),
628                  logy=True, ylabel="mean squared error", d=
                         d, task=task)
629     errors = [test_r[:, -1], train_r[:, -1]]
630     labels = ["test R^2 ", "train R^2 "]
631     print_errors(np.linspace(1, deg, deg), errors, labels,
632                  solver + "_R_squared_deg" + str(deg) + "
                         _epochs_" +
633                  str(epochs) + "_batches_" + str(batches) +
                         "_lambda_" + str(la),
634                  logy=logy, ylabel="R squared value", d=d,
                         task=task)
635
636
637 def task_a():
638     deg = 20
639     epochs = EPOCHS
640     batches = BATCHES
641     g0 = LEARNING_RATE
642     test_r, train_r, test_m, train_m, beta, train_cost,
            test_cost = train_degs(deg, epochs=epochs, batches=
            batches, g0=g0)
643     x = np.linspace(1, max_mat_len(deg), max_mat_len(deg))
644     y = np.linspace(1, deg, deg)
```

```
645     print_cont(x, y, beta[:, :, -1],
646                "ols_betas_epochs_" + str(epochs) + "
                       _batches_" + str(batches),
647                z_label="Parameter values", x_label="
                       Parameter Index", y_label="Polynomial
                       Degree")
648     errors = [test_m[:, -1], train_m[:, -1]]
649     labels = ["test MSE", "train MSE"]
650     print_errors(np.linspace(1, deg, deg), errors, labels,
651                  "ols_mse_epochs_" + str(epochs) + "
                         _batches_" + str(batches),
652                  logy=True, ylabel="mean squared error", d
                         =4, task="a")
653     errors = [test_r[:, -1], train_r[:, -1]]
654     labels = ["test R^2", "train R^2"]
655     print_errors(np.linspace(1, deg, deg), errors, labels,
656                  "ols_R_squared_epochs_" + str(epochs) + "
                         _batches_" + str(batches),
657                  ylabel="R^2 value", d=4, task="a")
658     epochs = np.linspace(1, 150, 150, dtype=int)
659     batches = np.array([batches])
660     lrs = np.logspace(-8, 2, num=9*10+1)
661     deg = 10
662     print("Finding ideal learning rate with estimated batch
            numbers and epochs")
663     train_print_hyperparameter(deg=deg, parameter=["
           learning_rate"], lrv=lrs,
664                                solver="ols", epochv=epochs,
                                       batchv=batches, task="a"
                                       , d=6)
665     epochs = np.linspace(1, 200, 200, dtype=int)
666     batches = np.linspace(1, 20, 30, dtype=int)
667     lrs = np.logspace(-4, -1, num=5*10+1)
668     print("Narrowing overall parameters down")
669     train_print_hyperparameter(deg=deg, parameter=["
           learning_rate", "batch", "epoch"], lrv=lrs,
670                                solver="ols", epochv=epochs,
                                       batchv=batches, task="a"
                                       , d=6)
671     print("large epochs still produce (probably only
```

```
672             slightly) better results. Testing for large "
                "epochs")
673         lrs = np.array([1.20e-3])
674         batches = np.array([12])
675         epochs = np.linspace(1, 500, 500, dtype=int)
676         train_print_hyperparameter(deg=deg, parameter=["epoch"
                ], lrv=lrs,
677                                    solver="ols", epochv=epochs,
                                        batchv=batches, task="a"
                                       , d=6)
678         print("The mean square error still decreases even after
                500 epochs, but the most improvement is"
679               "done after 100 epochs. For the sake of computing
                      time I limit myself to 150 epochs")
680         print("Choosing batches and learning rate that matches
                testing data")
681         epochs = np.linspace(1, 150, 150, dtype=int)
682         batches = np.array([18])
683         lrs = np.array([1.2e-3])
684         lambdas = np.logspace(-9, 5, num=15*20+1)
685         print("Finding ideal lambda with estimated batch
                numbers and epochs")
686         train_print_hyperparameter(deg=deg, parameter=["lambda"
                ], lrv=lrs, lambdav=lambdas,
687                                    solver="ridge", epochv=
                                       epochs, batchv=batches,
                                       task="a", d=6)
688
689         epochs = np.linspace(1, 150, 150, dtype=int)
690         batches = np.array([18])
691         lrs = np.array([1.2e-3])
692         lambdas = np.array([1])
693         print("Finding ideal lambda with estimated batch
                numbers and epochs")
694         train_print_hyperparameter(deg=deg, parameter=["epoch"
                ], lrv=lrs, lambdav=lambdas,
695                                    solver="ridge", epochv=
                                       epochs, batchv=batches,
                                       task="a", d=6)
696
```

```
697
698 NOISE_LEVEL = 0.1
699 MAX_DEG = 30
700 RESOLUTION = .02
701 RANDOM_SEED = 1337
702 SCALE_DATA = True
703 DATA_FILE = "files/test.csv"
704 np.random.seed(RANDOM_SEED)
705 EPOCHS = 150
706 BATCHES = 25
707 LEARNING_RATE = 1e-3
```