UNIVERSITY
OF OSLO

Master's thesis

Localization and identification of Neural Sources from simulated EEG Signals

Kamilla Ida Julie Sulebakk

Biological and Medical Physics
60 ECTS study points

Department of Physics
Faculty of Mathematics and Natural Sciences

Spring 2023

Kamilla Ida Julie Sulebakk

Localization and identification of Neural Sources from simulated EEG Signals

Acknowledgements

Massive thank-yous to my supervisor Gaute Einevoll and my co-supervisor Torbjørn Ness.

Contents

Acknowledgements **i**

Introduction **v**

1 Electroencephalography **3**
1.1 Generating an EEG data set 4
1.2 The Inverse Problem and Source Localization 8

2 Machine Learning and Neural Networks **9**

3 Single Dipol Source Localization **25**
3.1 The dataset 25
3.2 Feed-Forward Neural Network Approach for localizing single dipole sources 26
3.2.1 Training, testing and evaluation 26
3.3 Convolution Neural Network Approach for localizing single dipole sources 27
3.4 Region of Active Correlated Current Dipoles 27

4 Multiple Dipoles Source localization **33**

Introduction

Electroencephalography (EEG) is a method for recording electric potentials stemming from neural activity at the surface of the human head, and it has important scientific and clinical applications. An important issue in EEG signal analysis is so-called source localization where the goal is to localize the source generators, that is, the neural populations that are generating specific EEG signal components. An important example is the localization of the seizure onset zone in EEG recordings from patients with epilepsy. A drawback of EEG signals is however that they tend to be difficult to link to the exact neural activity that is generating the signals.

Source localization from EEG signals has been extensively investigated during the last decades, and a large variety of different methods have been developed. Source localization is very technically challenging: because the number of EEG electrodes is far lower than the number of neural populations that can potentially be contributing to the EEG signal, the problem is mathematically under-constrained, and additional constraints on the number of neural populations and their locations must therefore be introduced to obtain a unique solution.

For the purpose of analyzing EEG signals, the neural sources are treated as equivalent current dipoles. This is because the electric potentials stemming from the neural activity of a population of neurons will tend to look like the potential from a current dipole when recorded at a sufficiently large distance, as in EEG recordings. Source localization is therefore typically considered completed when the location of the current dipoles has been obtained. However, an exciting possibility is to try to go one step further and identify the type of neural activity that caused a localized current dipole. For example, the type of synaptic input (excitatory or inhibitory) to a population of neurons, and the location of the synaptic input (apical or basal) will result in different current dipoles (Ness et al., 2022). It has also been speculated that dendritic calcium spikes can be detected from EEG signals, which could lead to exciting new possibilities for studying learning mechanisms in the human brain (Suzuki & Larkum, 2017). Identifying different types of neural activity from EEG signals would however require knowledge of how different types of neural activity are reflected in EEG signals. Tools for calculating EEG signals from biophysically detailed neural simulations

have however recently been developed, and are available through the software LFPy 2.0 (Hagen et al., 2018; Næss et al., 2021). This allows for simulations of different types of neural activity and the resulting EEG signals, opening up for a more thorough investigation of the link between EEG signals and the underlying neural activity.

The past decade has seen a rapid increase in the availability and sophistication of machine learning techniques based on artificial neural networks, like Convolutional Neural Networks (CNNs). These methods have also been applied to EEG source localization with promising results. However, it has not been investigated if CNNs can also identify the neural origin of EEG signals, in addition to localizing neural sources. In this Master's thesis, the aim will be to investigate the possibility of using CNNs to not only localize current dipoles but also identify the neural origin of different types of neural activity, based on simulated data of different types of neural activity and the ensuing EEG signal.

Chapter 1

Notes

EEG mean: 1.513363748891642e-18
EEG std: 2.0134384686030296

Chapter 2

Electroencephalography

Electroencephalography (EEG) is a non-invasively technique for studying electrical potentials in the human brain. The technique was developed almost a century ago making it among the oldest methods for examining the brain's activity. Today, EEG remains among the most important techniques being used to study electrical activity in the brain, with important applications in both neuroscientific and clinical research (Nunez Srinivasan 2006, Lopes da Silva 2013, Biasiucci et al. 2019, Ilmoniemi Sarvas 2019).

According to the World Health Organization (WHO), nearly 50 million people suffer from epilepsy worldwide. Further there were more than 300 000 new cases of cancer related to brain tumors in 2020. The two mentioned diseases can arise at any age. They affect the cortical neural activity, and can be studied utilizing electroencephalography (EEG) [**?**].

EEG are among the most important techniques for studying cognition and certain stimuli in the human brain, without having to make any break in the skin [**?**]. Generally, information gained from EEG signals plays an important role for detecting and localizing abnormalities and diseases in the brain [**?**], such as epilepsy and tumors among many. By accurately locating the source that causes abnormal signals, EEG scans enable surgery for removal of diseases with minimal damage of healthy brain tissues.

EEG signals arise from cortical neural activity and are typically described in terms of current dipoles [**?**]. In this project we generate data by modeling current dipoles and EEG signals arising from abnormalities in the brain. Using a feed forward neural network we want to study the fluctuations in neural activity and train our model to localize the position of the disease causing unusual brain activity. While training our model we will explore different types of machine learning techniques, such as principal component analysis and cross-validation resampling method, in order to maximize the accuracy of the model.

2.1 Generating an EEG data set

EEG signal is believed to originate from large numbers of synaptic inputs to populations of geometrically aligned pyramidal neurons (Nunez Srinivasan 2006, Pesaran et al. 2018).

Electroencephalography (EEG) is one of the most important techniques for studying cognition and disease in the brain non invasively [**?**]. EEG is a method used to measure brain waves. In practice, this involves electrodes consisting of small metal disks connected to the surface of the scalp. The electrodes detect electrical charges that result from activity of the brain cells. In cases where certain areas of the brain come out as more active than others, it might indicate abnormalities, in which can be signs of disease. In other words, the EEG technique can be used to evaluate multiple types of brain disorders, such as lesions of the brain, Alzheimer's disease, epilepsy or brain tumors [**?**].

An illustration of the typical EEG measurement setup is depicted in Figure 1.1. Abnormality in EEG shows either as an increased amount of slow activity with a frequency below 8 Hz or as the appearance of abnormal waveforms, such as waves with a pointed shape and the occurrence of special patterns[**?**] . In this theses I will train a model to predict the localization of such abnormalities in the brain of patients. However, in lack of real data in consistent with the defined problem, I will be generating data by utilizing work done by external developers.

The signals received from EEG are known to originate from cortical neural activity, which are often described by using current dipoles [**?**]. It is therefor reasonable to implement current dipoles in the brain for when generating a biophysical modeling of EEG signals. The brain model used in this thesis is called the New York Head model [**?**], and is based on high-resolution anatomical MRI-data from 152 adult heads. The model utilizes the software tool LFPy, which is a Python module for calculation of extracellular potentials from multicompartment neuron models [**?**]. This model takes into account that electrical potentials are effected by the geometries and conductivities of various parts of the head [**?**].

An optimal model of EEG signals would have consisted of multiple dipole moments. However, as such a model is complicated and computationally expensive, we will in this project only introduce one single dipole approximation $\mathbf{p}(t)$ for each multicompartmental neuron simulation. In this context, by multicompartmental modelling we refer to the widely used models within neuroscience, which acurately manufactures electical properties of single neurons. The sigle-dipole approximation might sound like a substantial simplification of the real biophysical properties, nevertheless it actually turns out to give a realistic modelling of EEG signals, when handling the single dipole moment, as an abnormality in the brain. We will be thinking of the abnormality as an epileptic seizures or a tumor in the brain, which

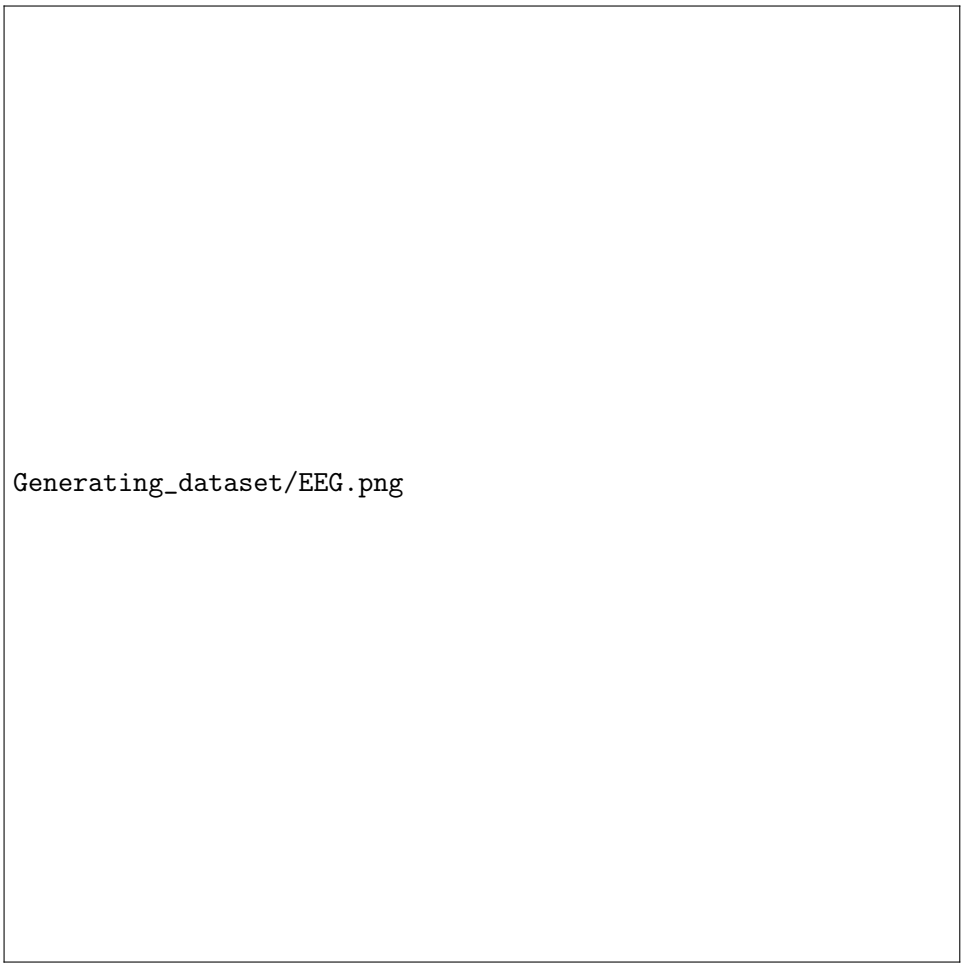

Figure 2.1: Illustration of the EEG method [**?**].

among normal activity in the brain would have stuck out. The single-dipole approximation is implemented by summing up the multiple current dipole moments, $\mathbf{p}(\mathrm{t}) = \sum_{k=1}^{M} \mathbf{p}_k(t) = \sum_{k=1}^{M} I_k^{axial}(t)\mathbf{d}_k$, where I^{axial} is the current flowing along the neurite with the distance vector $\mathbf{d}_k$ and M denotes the number of axial currents [**?**]. The data set we will be using in this project will consist of measures of different EEG signals at a given time from 1000 patients. This means that we for each patient pick a random location (at $t = 0$) for the single current dipole.

EEG signals are generated from synaptic inputs to cells in the cortex. Synaptic inputs are electrical (or chemical) signals that are being transmitted from one neuron to another, causing changes in the membrane potential of the neurons. In other words, neurons are specialized to pass signals, and synapses are the structures that make this transmission possible [**?**].
Imagining a small part of the cortex, all of these cells will have dendrites pointing upwards in the same direction (lets say the z-direction). Due to rotational symmetry around the z-axis, the contributions in the x- and y-direction will cancel. This is illustrated in Figure 1.2. What we see is that the extracellular potential is configured in all sorts of weird ways (A-C) when there is only one synaptic input, while the extracellular potential reminds more of a dipole when we have multiple synaptic inputs (D-F). We can therefore argue that the total contribution to the extracellular potential can be modelled as a dipole in the z -direction for the case where we have multiple synaptic inputs. Hence, for each dipole moment in our simulations, we pass multiple synaptic inputs, and make sure to rotate the positions of the dipoles such that it is orientated along the depth of the cortex.

The cortex matrix consists of 74382 points, which refer to the number of possible positions of the dipole moment in the cortex. When generating our data set, we will for each sample randomly pick the position of the dipole moment, such that one sample corresponds to one patient. In our head model we are considering 231 electrodes uniformly distributed across the cortex, meaning that each EEG sample will consist of this many signals for each time step. However, we are not interested in the time evolution of the signals as this does not affect nor say anything about the position of the dipole moment, and we therefor simply pick out the EEG signals for when $t = 0$ (note that the choice of time step could have been randomly picked). Our final design matrix will then consist of 1000 rows, corresponding to each patient, and 231 columns also refereed to as features, representing the signal of each electrode. The final output we are trying to predict is then the one dimensional vector with length 1000, where each element consists of the x-, y- and z- position of the dipole moment. An example of how the input EEG signals may look like is given in appendix A, where we also have marked the dipole moment with a yellow star.

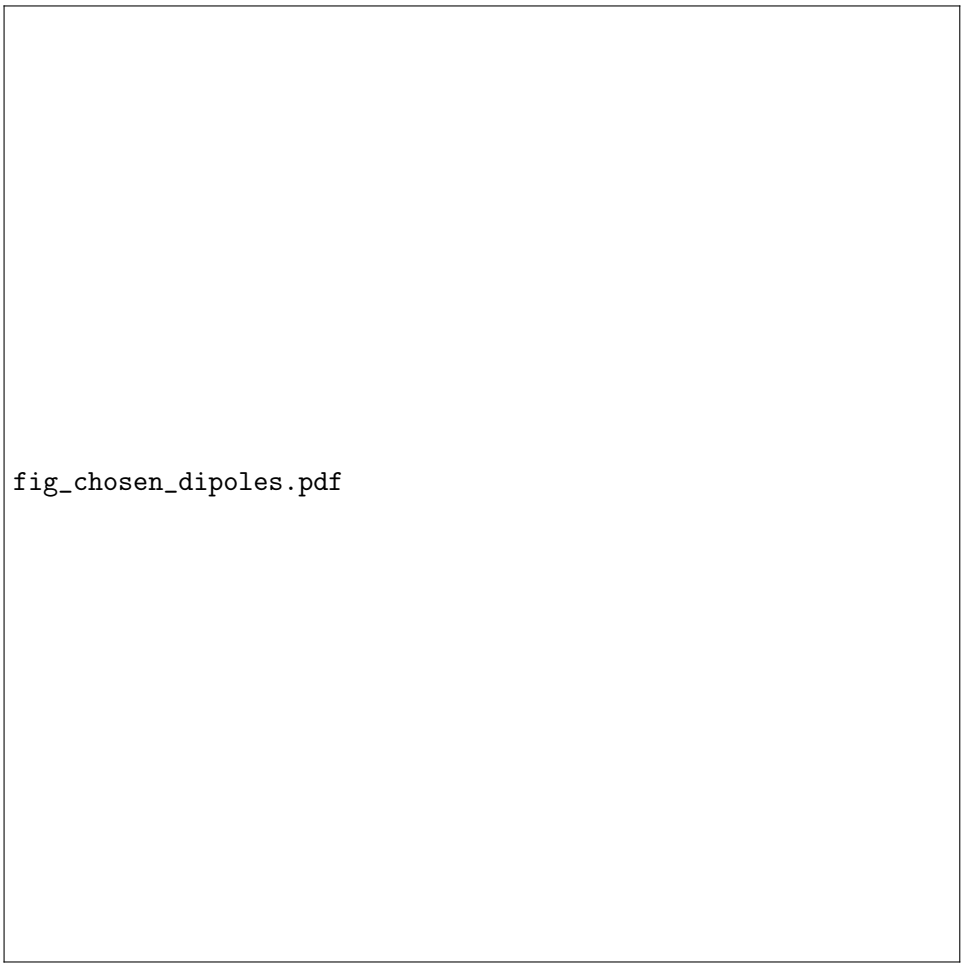

Figure 2.2: Extracellular potential from lonely synaptic input (A-C) and extracellular potential from multiple synaptic inputs (D-F).

2.2 The Inverse Problem and Source Localization

Chapter 3

Machine Learning and Neural Networks

Machine Leaning

The fit is found using the method of least squares using polynomials x and y up to the eighth order. The mean square error (MSE) can be used to evaluate how well the model fit our data, which is given by:

$$MSE(\hat{y}, \hat{\tilde{y}}) = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - \tilde{y}_i)^2 \tag{3.1}$$

Where $\tilde{y}_i$ is the predicted value and y_i is the corresponding true value. Having a MSE of zero, would mean that the estimator $\hat{y}$ predicts observations of the parameter $\hat{\tilde{y}}$ with perfect accuracy. This is obviously ideal, but is however, not typically possible.

The predicted value $\tilde{y}_i$ can be rewritten as $\tilde{y}_i = x_i^T \beta$, which we can use to find the residual sum of the squares:

$$RSS(\beta) = \sum_{i=1}^{N} (y_i - x_i^T \beta)^2 \tag{3.2}$$

The ordinary least squares method (OLS) wants to minimize the sum of the residuals, which is given by the RSS value (see equation 2.2). We want to find the β which minimize this function. If we define a $N \times p$ matrix $\mathbf{X}$ with each row as an input vector and a N-vector $\mathbf{y}$ of the true values, we can represent our solution $\hat{\beta}$ as a 1-dimensional array of size p using matrix notation:

$$\hat{\beta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \tag{3.3}$$

where $\mathbf{X}^T\mathbf{X}$ is non singular [?]. The size of $\hat{\beta}$ will depend on the degree of our polynomial fit. For example polynomial degree 5 we will give us a $\hat{\beta}$ length of p=21. The length N is given by the number of datapoints.

Just like MSE, the coefficient of determination R^2 is a measure for how precise a model is. To be more precise, it provides a measure of how well future samples are likely to be predicted by the model. The score normally ranges between 0 and 1, where 1 is the best possible score. In general, the higher the R-squared, the better the model fits the data. The coefficient of determination R^2 is given as follows:

$$R^2(\hat{y}, \tilde{\hat{y}}) = 1 - \frac{\sum_{i=0}^{n-1}(y_i - \tilde{y}_i)^2}{\sum_{i=0}^{n-1}(y_i - \bar{y})^2}, \tag{3.4}$$

Where the mean value of y_i is defined by $\bar{y}$:
$\bar{y} = \frac{1}{n}\sum_{i=0}^{n-1} y_i$.

Creating a perfect model for a set of data, can easily be done by using a high polynomial degree when fitting. However, creating a foolproof model that only fits one specific data set is not very beneficial. We want our model to predict correlated data, not slavishly follow the path of least square error. With this in mind we should split our data into training and testing sets in order to make the model compatible with multiple data sets. By splitting our data we can test how well our model predicts. In this exercise we have chosen to train on 80% of our data, and test on the remaining 20%. By training, in this context, we mean to find the β that minimize the cost function, using the training data. Having obtained $\hat{\beta}$ we test our model on both sets, expecting that it fits well for both the training and test data.

A much used approach before starting to train the data is to scale our data. The data may be very sensitive to extreme values, and scaling the data could render our inputs more suitable for the algorithms we want to use.

There are several methods for scaling data sets. In this project we have scaled our data by calculating the mean and standard deviation for each feature. For each observed value of the feature, we subtract the mean and divide by the standard deviation. The first feature (first column in our design matrix) was kept untouched, in order to avoid singularities when calculating β. This sort of scaling is called “Standardizing”. When standardizing data one obtain a "standard normal" random variable with mean 0 and standard deviation 1 [?].

If any, there are very few data sets in the real world which have ideal properties without noise. When generating our own data set for the Franke function we therefore explore the addition of an added normal distributed $N(0, 1)$ stochastic noise, in order to make the data set more realistic.

Gradient Descent

When training an algorithm one wants to minimize the cost function to reach its global minimum. Instead of using matrix inversion to reach this minimum we now want to use gradient descent. This is useful due to the fact that it sometimes is hard to perform matrix inversion on data sets.

Gradient Descent is an optimization algorithm for finding a local minimum of a derivable function. The underlying idea of the algorithm is that a function $F(\mathbf{x})$ decreases fastest if we take repeated steps in the opposite direction of the negative gradient $-\nabla F(\mathbf{x})$ of the function at $\mathbf{x}$. We have that:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta_k \nabla F(\mathbf{w}_k) \tag{3.5}$$

which leads to $F(\mathbf{w}_{k+1}) \leq F(\mathbf{w}_k)$, when $\eta_k > 0$. This means that for sufficiently small step lengths η_0 we are always moving towards a minimum. In the Gradient Descent method we do an initial guess on weights $\mathbf{w}_0$ and compute new approximations according to equation 2.5.

Gradient descent then starts at a point and takes steps in the steepest downside direction until it reaches the point where the cost function is as small as possible.

It is important to note that GD is very sensitive to the chosen initial condition. In machine learning we often deal with non-convex high dimensional cost functions with many local minima. This means, that unless we have a very good initial guess, we risk getting stuck in a local minimum when the gradient converges. Moreover, the algorithm is sensitive to the step length η_0, also referred to as the learning rate. If we take too large steps, we risk stepping over the global minimum point, resulting in unpredictable behavior. The step length also needs to be large enough so we don't get "stuck" in a local minimum point. Another fact is that a small learning rate will require many iterations before we reach a minimum point, which increases CPU time.

One common problem with the GD method is that it has a tendency to overshoot the exact minimum. It has been shown that by slowly decreasing the learning rate we will obtain convergence behaviour similar to the batch gradient descent, which can help it converge towards a local or global minimum point [**?**]. We define a learning rate function which adjust as a function of a variable t, which is proportional to the number of epochs. The learning rate function is given by:

$$\eta(t) = \frac{t_0}{t + t_1} \tag{3.6}$$

Where t_0 and t_1 are constants. This equation will decrease as we run through our batches, since we are dividing by the t term. In Scikit-Learn's

"SGDRegressor" we have that $t_1 = \frac{1}{0.01*t_0}$, which we use in our own model when we compare with the method provided in Scikit-Learn.

Stochastic Gradient Descent with momentum

In almost all cases we can write our cost function as a sum over n data points:

$$C(\beta) = \Sigma_{i=1}^{n} c_i(\mathbf{x}_i, \beta). \tag{3.7}$$

Which means that the total gradient can be computed as a sum over i -gradients:

$$\nabla_\beta C(\beta) = \Sigma_i^n \nabla_\beta c_i(\mathbf{x}_i, \beta) \tag{3.8}$$

The SGD method allows us to take the gradient on randomly selected subsets of data at every step rather than the full data set. These subsets are commonly referred to as mini-batches. In other words, Stochastic Gradient Descent is just like regular Gradient Descent, except it only looks at one mini-batch for each step. The mini-batches are denoted by B_k where k runs from 1 and up to the number of batches, n/M. An iteration over the number of mini-batches is called an epoch.

Introducing randomness by only taking the gradient on a subset of the data, is beneficial as it lowers the chance of getting stuck in a local minimum point. Moreover, splitting the data points into batches reduces the time spent calculating the derivatives of the cost function, since we sum over k batches and not all n data points. Implementing momentum to the Stochastic Gradient Descent, helps accelerate gradient vectors in the right directions, which leads to faster converging. In many ways this improves the algorithms sensitivity to the learning rate. The momentum serves as a memory of the direction we are moving in parameter space and can be written as follows:

$$\mathbf{v}_t = \gamma \mathbf{v}_{t-1} - \eta_t \nabla F(\beta_t) \tag{3.9}$$

$$\beta_{t+1} = \beta_t - \mathbf{v}_t \tag{3.10}$$

where the parameter γ represents the momentum and must be between 0 and 1. The momentum term γ is usually set to 0.9 or a similar value [**?**]. We used $\gamma = 0.9$. We also set the initial "velocity" to $v = 0$.

The momentum SGD algorithm

[H]
Initialize all the parameters. Create design matrix X Call Franke's Function with x and y. This is our z. Guess on some β values. epoch $\in \{N_{epochs}\}$

Shuffle the data index $\in \{m_{minibatches}\}$ *Calculate gradients Find current learning rate* $v = \gamma * v + \eta *$ *gradients* $\beta = \beta - v$

For the ordinary SGD method we have one less term in the loop over minibatches: $\beta = \beta - \eta * gradients$.

Neural Networks

Artificial Neural Networks are computational systems that can learn to perform tasks by considering examples, generally without being programmed with any task-specific rules [**?**].

The biological neural networks of animal brains, wherein neurons interact by sending signals in the form of mathematical functions between layers, has inspired a simple model for an artificial neuron:

a = f $(\Sigma_{i=1}^{n} w_i x_i + b_i) = f(z)$

where the output a of the neuron is the value of its activation function f, which as input has the sum of signals $x_i, x_{i+1}, ..., x_n$ received by n other neurons, multiplied with the weights $w_i, w_{i+1}, ..., w_n$ and added with biases.

Most artificial neural networks consists of an input layer, an output layer and layers in between, called hidden layers. The layers consists of an arbitrary number of neurons, also referred to as nodes. The connection between two nodes is associated with a weight variable w, that weights the importance of various inputs. A more convenient notation for the activation function is:

$$a_i(x) = f_i(z^{(i)}) = f_i(w^i \cdot x + b^i) \tag{3.11}$$

where $w^{(i)} = (w_1^{(i)}, w_2^{(i)}, ..., w_n^{(i)})$ and $b^{(i)}$ are the neuron-specific weights and biases respectively. The bias is normally needed in case of zero activation weights or inputs [**?**].

Feed Forward Neural Network

As the name name suggests, in Feed Forward Neural Network (FFNN) the information only moves in one direction, forward through layers. This means that in an FFNN, the inputs x_i of the activation function f are the outputs of the neurons in the preceding layer.

The Universal Approximation Theorem tells us that no matter what our data set is, there is a Neural Network that can approximately approach the result and do the job. This result holds for any number of inputs and outputs [**?**].

Thus the basic components of a neural network are stylized neurons consisting of a linear transformation that weights the importance of various inputs, followed by a non-linear activation function. A typical example for such an activation function is the logistic Sigmoid (**??**).

We assume that there are L layers in our network with $l = 1, ..., L$ indexing the layer and that different layers have different activation functions.

Further we denote w_{ij}^l as the weight for the connection from the j-th neuron in layer $l-1$ to the $i-th$ neuron in layer l. The bias of this neuron is written as b_i^l. The mathematical model for a FFNN is then reads:

$$a_i^l = f^l(z_i^l) = f^l\left(\Sigma_{j=1}^{N_{l-1}} w_{ij}^l a_j^{l-1} + b_i^l\right) \tag{3.12}$$

where l denotes the l-th layer and N_l is the number of nodes in layer l. A FFNN that is fully-connected consisting of neurons that have non-linear activation functions, receives a weighted sum of the outputs of all neurons in the previous layer (figure **??**).

We will be studying Franke's function which produces a certain type of terrain data. You can read more about Franke's function in our previous report [**?**].

Back propagation algorithm

The back propagation algorithm is a clever procedure that allows us to change the weights in order to minimize the cost function. At its core, back propagation is simply the ordinary chain rule for partial differentiation, and can be summarized using four equations [**?**].

The fist equation is the definition of the error δ_i^L of the i-th neuron in the L-th layer:

$$\delta_i^L = \frac{\partial C}{\partial(z_i^L)}, \tag{3.13}$$

which can be thought of as the change to the cost function by increasing z_i^L infinitesimally. By definition, the cost function measures the error of the output compared to the target data. So if the error δ_i^L is large, that would suggest the cost function hasn't yet reached its minima. The second equation is the analogously defined error of neuron i in layer l, δ_i^l:

$$\delta_i^l = f'(z_i^l)\frac{\partial C}{\partial(a_i^l)} \tag{3.14}$$

where $f'(z_i^l)$ measures how fast the activation function f is changing at the given activation value. Utilizing that $\delta_i^l = \frac{\partial C}{\partial z_j^l}$ we can express the error in terms of the equations for layer $l+1$. This can be done by using the chain rule, and is the third back propagation equation (for full calculations see [**?**] under "Final back propagation equation"):

$$\delta_i^l = \frac{\partial C}{\partial z_i^l} = \Sigma \delta_j^{l+1} w_{ij}^{l+1} f'(z_i^l) \tag{3.15}$$

Finally the last equation of the four back propagation equations the derivative of the cost function in terms of the weights:

$$\frac{\partial C}{\partial w_{ij}^l} = \delta_i^l a_j^{l-1} \tag{3.16}$$

With these four equations in hand we can now calculate the gradient of the cost function, starting from the output layer, and calculating the error of each layer backwards. We then have a way of adjusting all the weights and biases to better fit the target data. The back propagation algorithm then goes as follows:

1. **Activation at input layer:** calculate the activations a_i^1 of all the neurons in the input layer.

2. **Feed forward:** starting with the first layer, utilize the feed-forward algorithm through 2.12 to compute z^l and a^l for each subsequent layer.

3. **Error at top layer:** calculate the error of the top layer using equation 2.13. This requires to know the expression for the derivative of both the cost function $C(W) = C(a^L)$ and the activation function $f(z)$.

4. **"Backpropagate" the error:** use equation 2.15 to propagate the error backwards and calculate δ_j^l for all layers.

5. **Calculate gradient:** use equation 2.14 and 2.16 to calculate $\frac{\partial C}{\partial z_i^l}$ and $\frac{\partial C}{\partial w_{ij}^l} = \delta_i^l a_j^{l-1}$.

6. **Update weights and biases:**
   $w_{jk}^l = w_{jk}^l - \eta \delta_j^l a_k^{l-1}$
   $b_j^l = b_j^l - \eta \delta_j^l$

Structure of our neural network

The way we decided to structure our neural network is based on how to handle the design matrix containing the input data used for producing Franke's Function terrain data. We decided to take a slightly different approach to setting up the design matrix compared to in project 1 [**?**]. Instead of assuming a polynomial of a given complexity as we did there, we now set up the design matrix as shown below in (2.17). This way the columns represent all the x and y values respectively, and each row represent each its own unique combination of the input points. The design matrix then contains all the inputs used for producing the terrain, so by the universal approximation theorem there should then exist a network which should produce Franke's Function terrain. The design matrix then takes the form:

$$X = x_1 y_1 x_2 y_1 .... x_n y_1 x_1 y_2 x_2 y_2 .... x_n y_2 .... x_1 y_n x_2 y_n .... x_n y_n \tag{3.17}$$

Where n is the number of points in one direction. We then define the number of input rows as $N = n \times n$. We can visualize the feed forward method like this:

Where w_i is the input weights and w_o is the output weights. The numbered sub-indexes up to L denote the hidden layers. N is the input data points and n_h is the number of hidden neurons. The superscripts denote the dimensions of the matrices. Notice how we have changed the order of the input-weight matrix multiplication when calculating the z's. The reason for changing the order has to with making the dimensions add up, because when following the standard feed-forward formulas the dimensions didn't add up.

For our neural network we choose the quadratic cost function with a Ridge regularization parameter (L_2-norm), described as follows:

$$\frac{1}{2}\sum_{i=1}^{n}(a_i - t_i)^2 + \lambda||w||_2^2. \tag{3.18}$$

The reason for choosing the quadratic loss function, as well as the L_2-norm, is their versatility and simple derivatives. When we look at the terrain data from Franke's function it is somewhat unnecessary to us regression models and resampling methods with the Ridge regularisation parameter. The reason was discussed in our previous report [**?**], which was that the lack of heavy outliers.

However, when we are analyzing the breast cancer data set, we might expect more heavy outliers, which could benefit from a regularization parameter.

Activation functions

The use of activation functions is inspired by the action potential of a human brain neuron; depending on the incoming current, the neuron will either fire or not. Activation functions work in a similar way, the input to a single neuron is transformed by an activation function causing a non-linear relation between the input and output. It is in this way the network learns. There are many activation functions, but the ones we will experiment with and discuss in this project are mainly:

- The Sigmoid function: $f(z) = \frac{1}{1+e^{-z}}$
- Hyperbolic tangent: $f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- ReLU: $f(z) = max\{0, x\}$.

Sigmoid function

A big reason for using the Sigmoid is that it's range is between $[0, 1]$. This is naturally good when predicting probabilities. Another advantage when

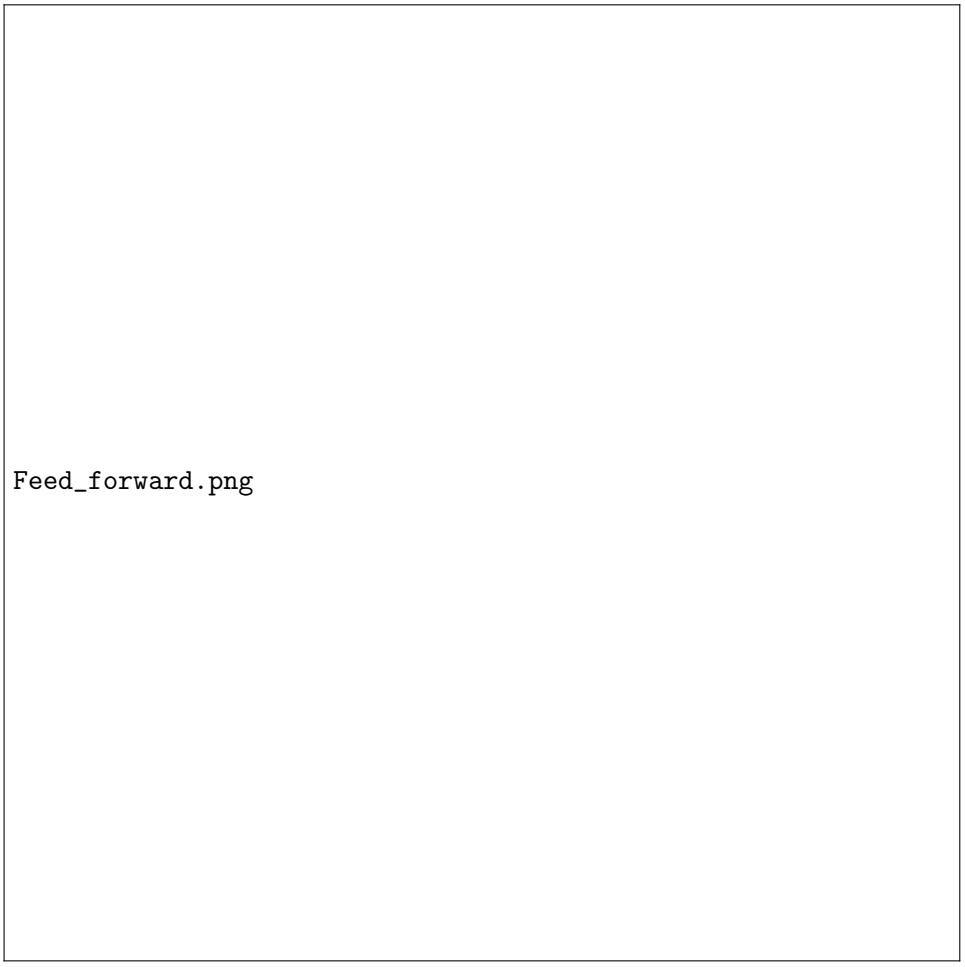

Figure 3.1: The feed forward method.

using the Sigmoid is that it is differentiable everywhere, where its analytical expression re-uses the sigmoid: $f'(x) = f(x)(1 - f(x))$. This allows for effective computing in cases such as the back propogation algorithm which uses the derivative.

Hyperbolic tangent (Tanh)

The hyperbolic tangent is similar to the Sigmoid in its shape and characteristics, but Tanh's range is instead between $[-1, 1]$ and $\text{Tanh}(0) = 0$. Similar to the Sigmoid, Tanh also reuses itself in its expression for its derivative: $f'(x) = 1 - f(x)^2$

ReLU

ReLU is quite simple compared to the two mentioned above, in that it returns its input value if the input is above zero, and returns zero if the input is below zero. Its simplicity is one of the main reason for its wide usage in neural networks. For input values larger than zero it's also effective in avoiding vanishing gradients which the two others mentioned above are susceptible to (This will be discussed further in the next section). Though it must be noted that also the ReLU could be sucseptible to vanishing gradients given that the derivative for negative input values are zero. One minor problem when using ReLU is that it's not differentiable at $x = 0$. This isn't necessarily a big problem as you can just define the derivative there to be zero, which is correct when x approaches zero from the left. But it is something to be wary of when using the ReLU.

Initialization of the weights and biases

The initialization of the weights and biases is seemingly arbitrary, since the network in theory should change them accordingly to fit the target data. But there are naturally some initializations that will minimize the number of epochs necessary to reach convergence, and there are some that in practice won't reach convergence at all. The optimal choices of initialization will also depend on the chosen activation function in the network. Lets use the Sigmoid as an activation function as an example. If you choose very large, or very small weights, the gradient will become very small (figure 2.5), causing the network to learn at an extremely slow rate. This is known as a vanishing gradient.

However, by choosing a different activation function, say the ReLU or $\text{Tanh}(x)$, we might run into different problems. If we again initialize the weights to be very large, we can see that the derivative of the activation grows proportionally large (for certain areas) as the weights increase (figure 2.6), which in turn leads to what's normally called "exploding gradients". This problem will then only become worse as the number of layers are increased,

Figure 3.2: We see the sigmoid with its characteristic S-shape and horisontal asymptotes at 1 and 0. Note that Sigmoid(0) = 0.5.

Figure 3.3: We see the hyperbolic tangent also with its characteristic S-shape, but with its horisontal asymptotes at -1 and 1.

though mainly for ReLU, since the proportionality factor from the weights carry through for each layer.

The neural network can actually also experience exploding gradients when using the Sigmoid function. But if we were to use biases equal to zero as in the example above it wouldn't happen. This is because the Sigmoid is bounded between 0 and 1, with $Sig(0) = \frac{1}{2}$ and $Sig(1) \approx 0.73$. Thus, $Sigmoid(Sigmoid(x))$ is bounded between 0.5 and 0.73. As the number of layers increase, the resulting slope will then only become flatter and flatter. However, if a bias of value $b = \frac{w}{2}$ is added to the input in each layer, the gradient will start exploding, as shown in figure 2.7. Though, it has to be noted that the chances of achieving such an exploding gradient when using the Sigmoid seems rather unlikely, as long as the weights and biases are not initialized in this specific way.

The Xavier and Kaiper/He initialization methods.

From what we have discussed above, it seems that choosing reasonable initial weights and biases in combination with the right activation function is nothing other than a minefield impossible to navigate through. Luckily for us, there are people before us which have found methods to do just this.

The first initialization method we will cover is called the Xavier weight initialization and is an approach normally used for Sigmoid as activation function. The challenge faced is to avoid shrinking the variances of the outputs through each layer. The way this is solved is to initialize the weights of each layer uniformly, and as function of the number of input nodes to that layer. The way we then initialize one layer is to draw from the uniformly distributed weights between:

$\pm\frac{1}{\sqrt{n_l}}$, where n_l specifies the number of inputs to that layer. As Tanh is pretty similar to the Sigmoid, it's only natural that the weight initialization also is similar, which it is. For initializing the wheights when using Tanh we use a variation of the Xavier initialization called the normalized Xavier initialization. Using this the weights are initialized uniformly distributed between:

$\pm\frac{\sqrt{6}}{\sqrt{n_l+n_{l+1}}}$, where n_{l+1} specifies the number of nodes of the outgoing layer.

The second initialization method we will cover is called the Kaiming/He initialization and is an initialization method used for the ReLU activation function. Again the challenge is to avoid shrinking variances through each layer. This method exploits that when using ReLU, the ouput of a layer has a standard deviation of approximately $\sqrt{n_l/2}$, where again n_l is the number of inputs to that layer. The idea is then to scale each layer by $\sqrt{n_l/2}$ so that each layer has an average standard deviation of 1. The weights of a given layer are then initialized from a Gaussian distribution with mean zero and standard deviation of $\sqrt{2/n_i}$.

Figure 3.4: We see the ReLU with its non-differentiable sharp turn at $x = 0$.

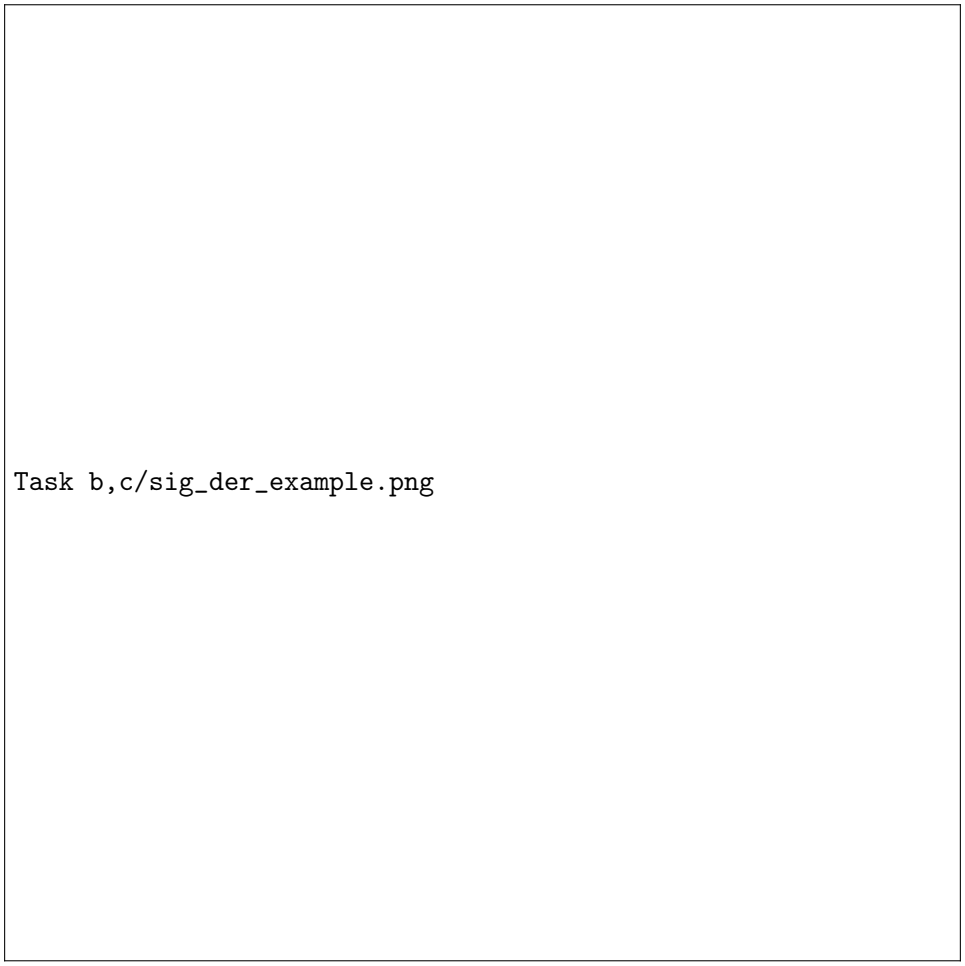

Figure 3.5: The Sigmoid function and its derivative. We see that for large x-values the derivative converges towards zero.

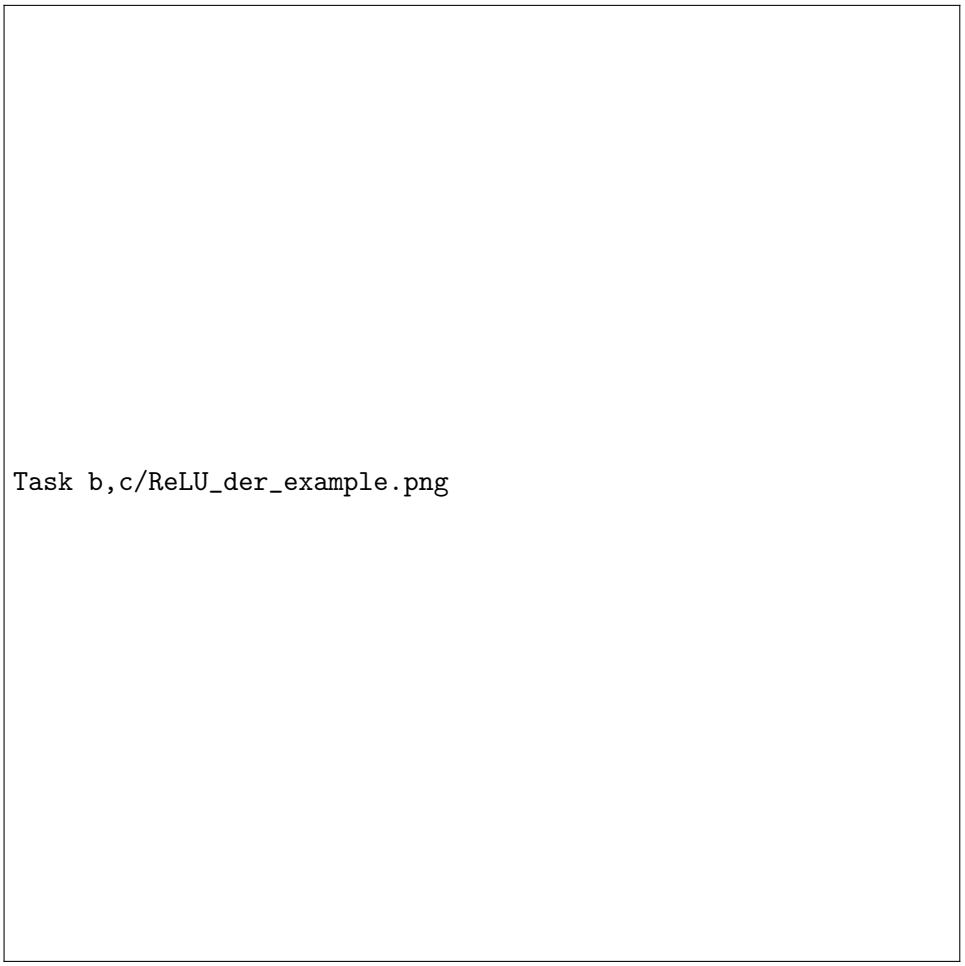

Figure 3.6: In the figure above we can see that the derivatives grow proportionally large as the weights increase. The difference between the two functions can be seen as the derivative of Tanh only has one spike around $x = 0$, while the derivative of the ReLU function is unbounded as $x\infty$

.

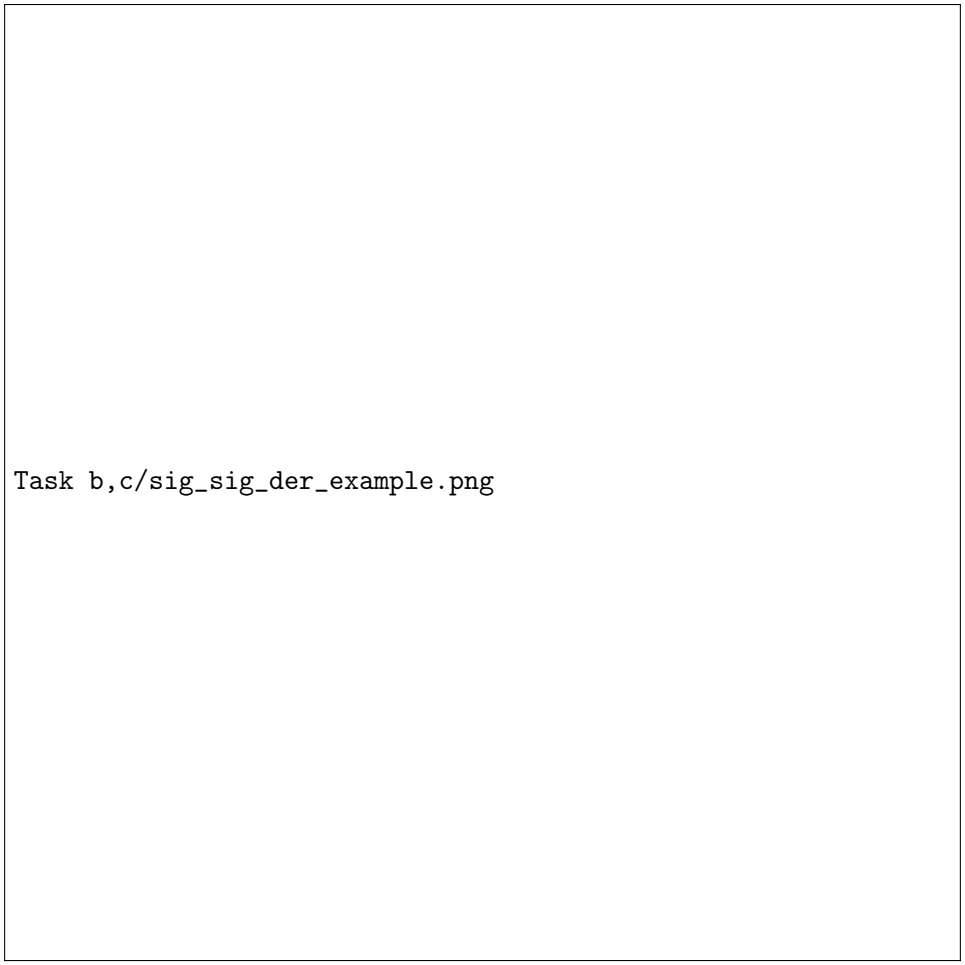

Figure 3.7: In the figure above we can see that when the input of the second layer is specified as $z = Sig(w \cdot x - w/2)$ the derivative starts to grow. This would continue to happen for each layer ultimately causing an exploding gradient. This is in contrast to using $b = 0$ which would cause a vanishing gradient.

Chapter 4

Single Dipol Source Localization

As mentioned in chapter 1, an important topic in EEG signal analysis is the inverse problem of going from measured EEG signals to localized equivalent current dipoles, so-called source localization. In this chapter we will be training and presenting two different(?) neural networks used to localize single dipole sources in the human cortex. Section ... and ... deal with training a simple feed forward neural network and presenting its results.

4.1 The dataset

The cortex matrix of the New York Head Model (NYHM) consists of 74382 points, which refer to the number of possible positions for localization of a dipole source. When training the FFNN we will be using a data set consisting of simulated EEG signals corresponding to dipol sources with randomly selected positions within the cortex matrix. The final data set consists of 10 000 rows, where each row corresponds to one sample, or let us say - one patient. Within the data set we have 231 columns, also referred to as features, representing the dipole measure at every EEG electrode. Thus, we are left with a design matrix with size 10 000 x 231.

In order to train the network faster, one commonly split the data set into mini-batches, which is also done here. When splitting the data such a way, the weights of connection between neurons are updated after each propagation, making the network converge considerable faster. There might be Through trial and error we landed on a batch size equal to 30.

4.2 Feed-Forward Neural Network Approach for localizing single dipole sources

The feedforward neural network (FFNN) was one of the first artificial neural network to be adopted and is yet today an important algorithm used in machine learning. The feed forward neural network is the simplest form of neural network, as information is only processed forward, from the input nodes, through the hidden nodes and to the output nodes.

The FFNN that are trained to solve the inverse problem of ours has an input layer of 231 neurons, corresponding to the M = 231 electrode measures of the potentials. The input layer is followed by three hidden layers with 120, 84 and 16 hidden neurons, respectively. The final output layer holds the predicted x-, y- and z- position of the desired dipole source. For the neurons of the input layers we use the linear activation function ReLu, while for the neurons of the hidden and output layers, we chose the much used hyperbolic tangent activation function.

Cost function

4.2.1 Training, testing and evaluation

In order to make an ANN that generalizes well to new data we split our data into training and testing sets. Randomly selecting 80 percent of the rows in the full dataset, we put this into a separate one and call it our training set. The remaining 20 percent is put into the test set. In practice, the training data set consists of pairs of an input vector with EEG signals and the corresponding output vector, where the answer key is the x-, y- and z coordinate of the dipole source. The neural network is then feed with the training data and produces an estimation of the localization of the dipole. The estimation is found by the network through optimizing the parameters β minimizing the cost function, or said in other words, through finding parameters for the function that produces the smallest outcomes, meaning the smallest errors. The result provided by the network is then compared with the target, for each input vector in the training data. Adjustment of parameters...

When the network is fully trained, we have a final model fit on the training data set. Feeding the network with the test data set, we can assess the performance of the network. The predictions of the fully trained network can now be compared to the holdout data's true values to determine the model's accuracy.

In figure **??** we have provided the bias-variance trade-off for when using Tanh as activation function. We notice that error of the model is approaching 0 and that the variance between the two curves decreases for an increasing number of epochs. In

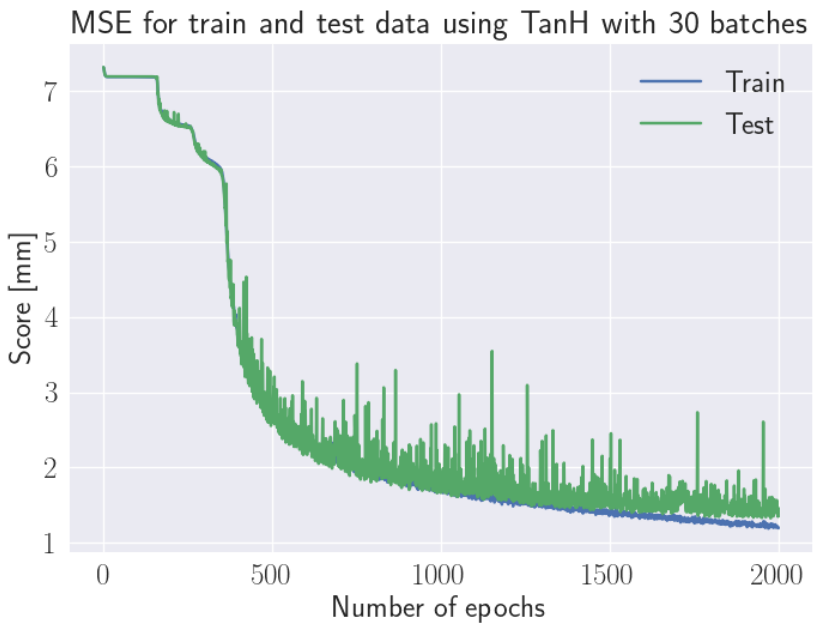

Figure 4.1: The validation accuracy for simple Feed Forward Neural Network with 10 000 samples with tanh activation function.

4.3 Convolution Neural Network Approach for localizing single dipole sources

Some results for the prediction of location for single current dipoles.

4.4 Region of Active Correlated Current Dipoles

Some results for the prediction of the size and location of current dipole populations.

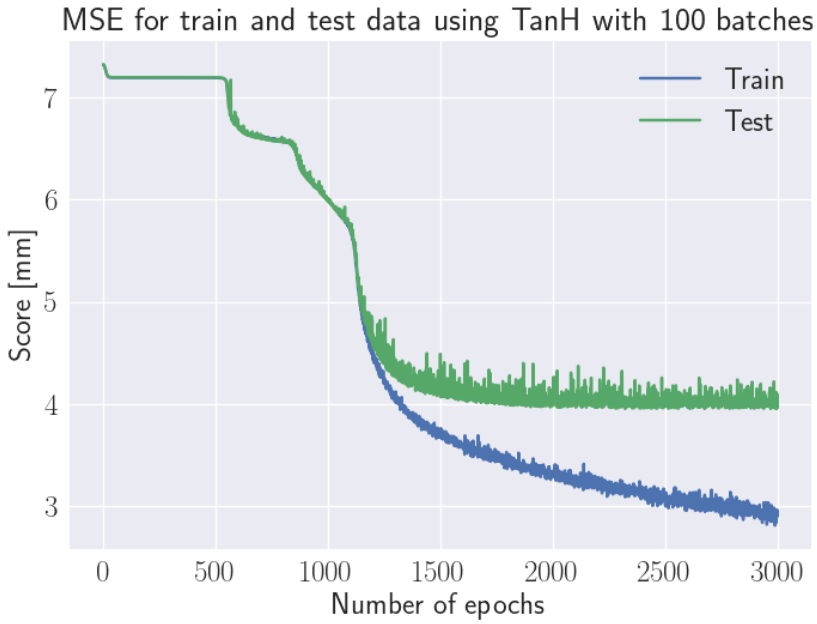

Figure 4.2: The validation accuracy for simple Feed Forward Neural Network with 10 000 samples with tanh activation function and 10% noise added to the data.

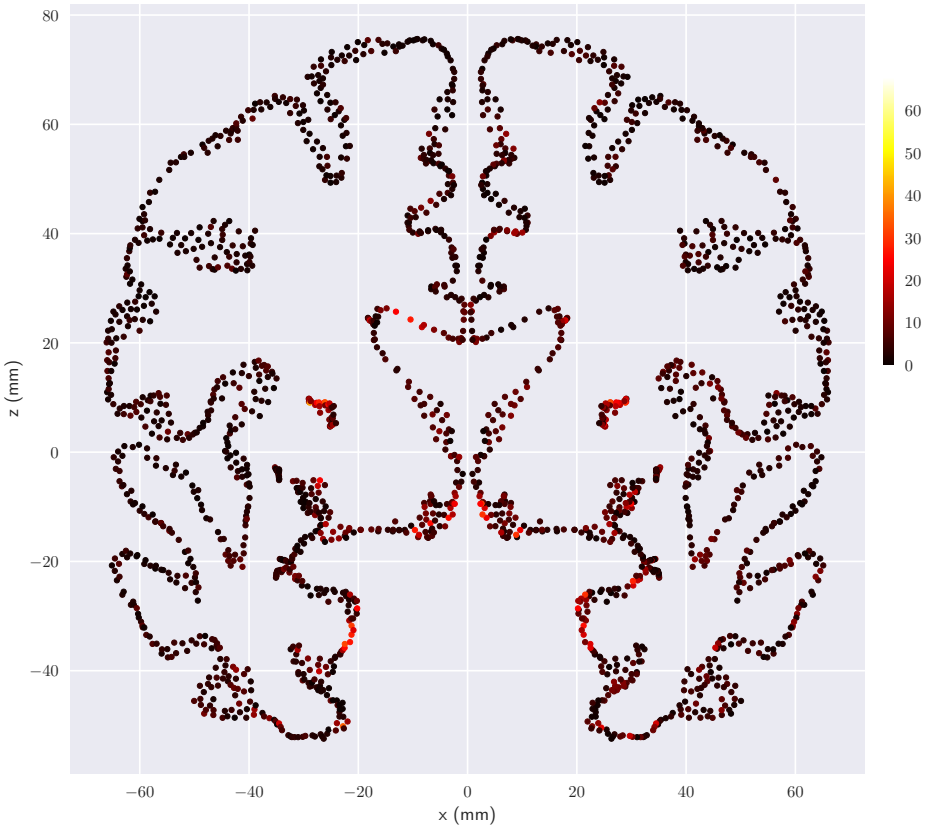

Figure 4.3: The mean squared error of the location accuracy (NN) at different dipole locations in the y cross section.

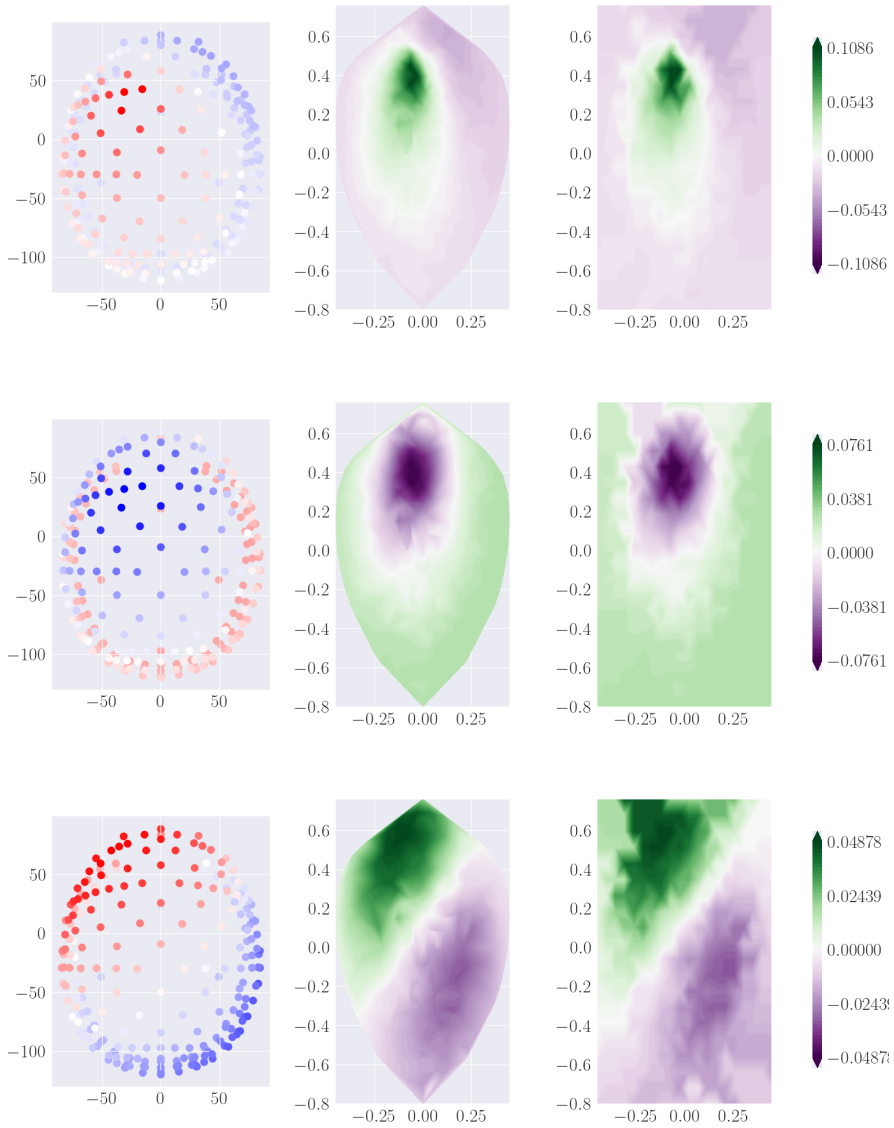

Figure 4.4:
Right: EEG measure for 3 different samples measured in μV.
Middle and Left: Illustration of the interpolation of the EEG data into two-dimensional matrix.

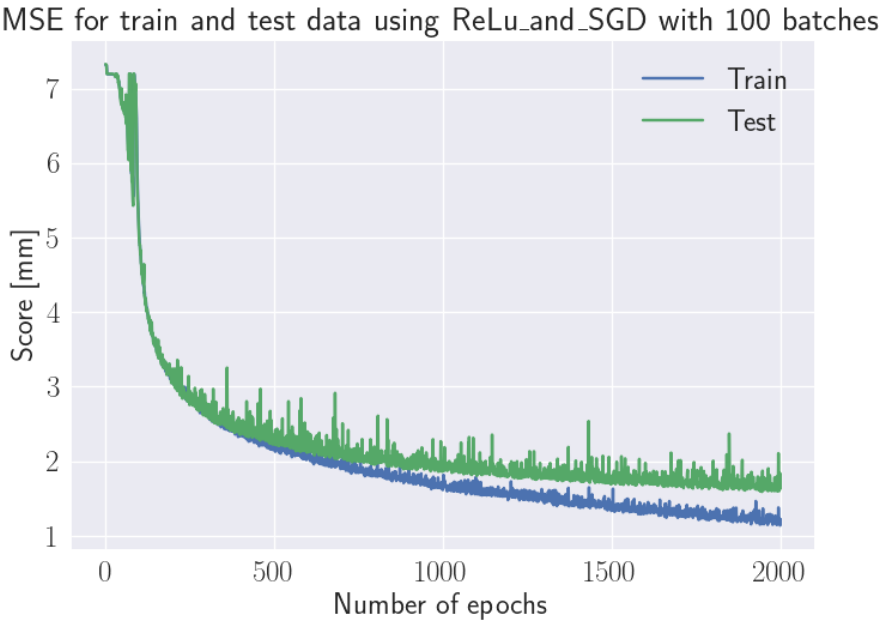

Figure 4.5: The validation accuracy for Convolutional Neural Network with 10 000 samples (20x20 matrix) with ReLU activation function.

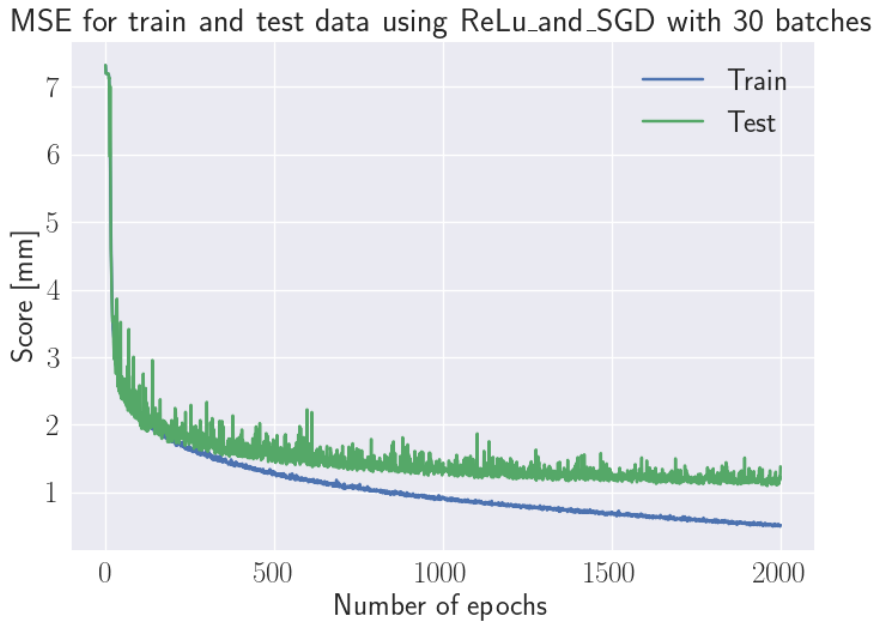

Figure 4.6: The validation accuracy for Convolutional Neural Network with 10 000 samples (20x20 interpolated matrix) with ReLU activation function.

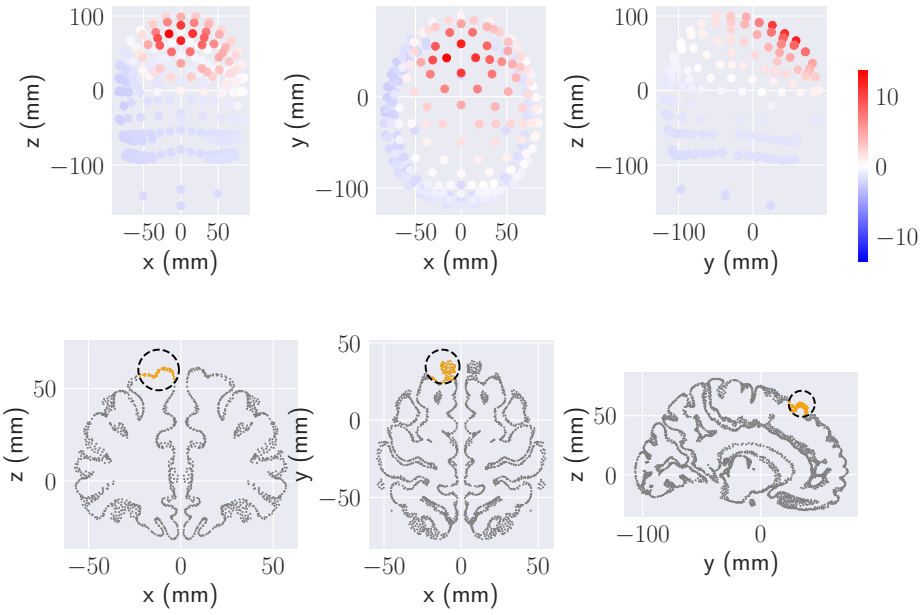

Figure 4.7: The validation accuracy for simple Feed Forward Neural Network with 10 000 samples with tanh activation function.

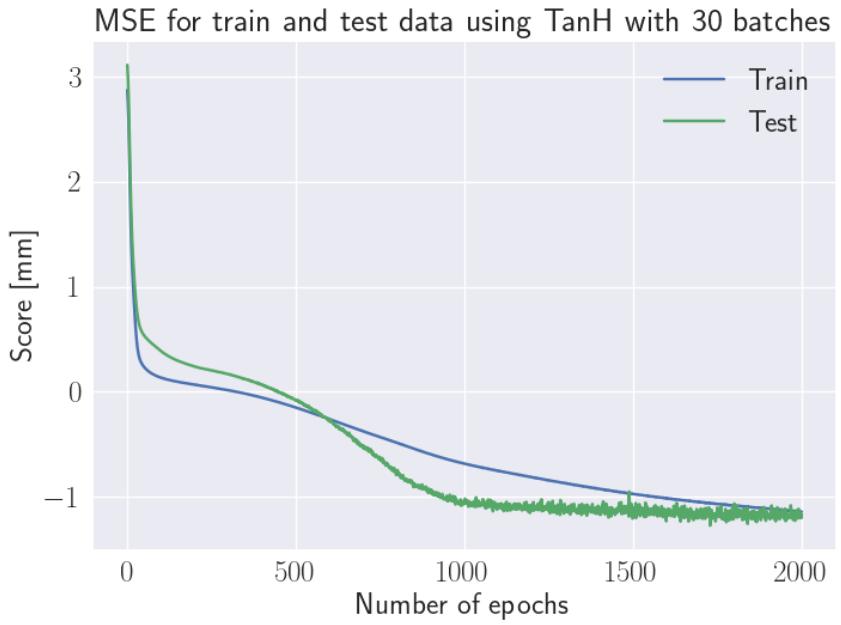

Figure 4.8: The validation accuracy for simple Feed Forward Neural Network with 10 000 samples with tanh activation function.

Chapter 5

Multiple Dipoles Source localization