Deep Double Descent

Does the double descent phenomenon exist in deep neural networks for regression problem?
Existerar fenomenet "double descent" i djupa neurala nätverk för regressionsproblem?

Zaitsev, Anton

Handledare: Chun-Biu Li
Examinator: Josefin Ahlkrona
Inlämningsdatum: 2024-05-20

Abstract

This thesis project examines if the double descent phenomenon is exhibited by deep neural networks for a regression problem. The phenomenon was first observed by Nakkiran and others [1] for a classification problem. Three experiments are conducted by training different sized feedforward neural networks for long periods (thousands of iterations). Additionally, we investigate if model complexity increases over training time by estimating a complexity measure called Effective Model Complexity. The complexity measure is estimated by averaging over multiple initializations of models and over multiple training data sets.

The results show that double descent is not exhibited by feedforward neural networks for a regression problem. Furthermore, the results show that model complexity increases over training time.

Sammanfattning

Denna avhandling undersöker om fenomenet "double descent" uppvisas av djupa neurala nätverk för ett regressionsproblem. Fenomenet observerades först av Nakkiran med flera [1] för ett klassificeringsproblem. Tre experiment utförs genom att träna frammåtmatande neurala nätverk av olika storlekar under långa perioder (tusentals iterationer). Dessutom undersöker vi om modell-komplexitet ökar med träningstiden genom att uppskatta ett komplexitetsmått kallat "Effective Model Complexity". Komplexitetsmåttet uppskattas genom att ta medelvärdet över flera initialiseringar av modeller och över flera träningsdatamängder.

Resultaten visar att "double descent" inte uppvisas av frammåtmatande neurala nätverk för ett regressionsproblem. Vidare visar resultaten att modell-komplexitet ökar med träningstiden.

Contents

1 Introduction

In machine learning and statistics, we generally expect models to follow a "standard" bias-variance trade-off curve. This means that as we increase the complexity of a model, we expect to see it perform better on unseen data. At some point, it reaches an optimal fit, after which it starts to perform increasingly worse on unseen data. However, for models like deep neural networks, it has been observed that after some threshold, the error starts to decrease again, hence the name "double descent." This has been observed in [1].

Overall, there is an idea of what happens during this threshold, but since neural networks are essentially "black boxes," it remains elusive why this double descent occurs. Furthermore, the paper [1] only demonstrates this phenomenon for classification problems. It shows that the phenomenon can occur when varying the size (width and depth) of a model as well as the training time. This leads us to what this project will cover. We are interested in investigating whether this double descent phenomenon also occurs for regression problems. Therefore, we aim to perform similar analyses as in the paper but on a regression problem using a feedforward network, incorporating generalizations such as skip connections inspired by residual blocks [2].

As stated before, the bias-variance trade-off happens over varying model complexity. Nakkiran and others [1] show that double descent happens over training time. Moreover, Nakkiran and others [1] are not alone in showing some sort of bias-variance trade-off over training time. There is a general suggestion that training time increases model complexity. This is not something that has been rigorously shown, and thus it is something we would like to investigate in this project. For this, we will define a complexity measure called Effective Model Complexity, loosely defined as the largest sample size n for which the expected training error is approximately zero, and investigate if this measure increases over training time.

We expect that the outcome of this project can inform us about the existence of the double descent phenomenon in regression models, as well as the possible causes of such behavior. The motivation for this is the importance of understanding the models we are training. Since the aspiration is to later use these models in the real world, we need to be able to understand them to trust them. This is especially important for applications in medicine and fields where people's lives could be greatly affected.

2 Preliminaries and Methods

To understand the problem of double descent we first have to understand all of the components possibly exhibiting the behaviour. Before we go further into neural networks we need look at the base of the problem that we are trying to solve. As well as how neural networks go about to solve the problem.

2.1 Solving the regression problem

2.1.1 Regression

The goal of machine learning is to approximate some unknown point-target rule. This rule we describe as some true function f^*. Since our problem is a regression problem the function will be a mapping from some variables $\boldsymbol{x} = (x_1, ..., x_d)$, where d is the number of features, to some continuous target variables $\boldsymbol{y}$ defined in some intervals $\boldsymbol{I}$. We will restrict ourselves to working with one target variable and thus y will be restricted to the interval (a, b).

As a first assumption we say that we have some true function f^* describing the relationship between $\boldsymbol{x}$ and y. We will also assume that we have some irreducible error ϵ that can not be modeled. This irreducible error ϵ can for example be noise when gathering data or possibly some rounding error. We now have that y can be written as

$$y = f^*(\boldsymbol{x}) + \epsilon. \tag{2.1}$$

Our goal has now been reduced down to approximating $f^*(\boldsymbol{x})$ with a parametric family $\hat{f}(\boldsymbol{x}; \boldsymbol{\theta})$ where $\boldsymbol{\theta}$ are the parameters (A parametric family is essentially some model that depends on parameters $\boldsymbol{\theta}$).

Now, we already have a way of approximating functions without neural networks. This can be done with different kinds of regression, polynomial regression, exponential regression et cetera. But to do this we need to have some sort of understanding of the true function we are trying to model beforehand. If we, for example, have one dimensional points x and targets y and know that the true underlying function is an exponential one, we can use $\hat{f}(x; \boldsymbol{\theta}) = \alpha e^{\beta x}$ as our parametric family and find parameters $\boldsymbol{\theta} = (\alpha, \beta)$ that best fit the data. The reason for using neural networks is that we want to model the true underlying function without knowing anything about it beforehand.

2.1.2 Optimizing the model

Now that we have a regression problem, we want to find some solution to it. What we will do is define a cost function which will tell us how good our model fits some data. Then we can optimize the model using the cost function.

In the case of linear regression the cost function is mean squared error. In this case the cost function is convex and there exists methods that guarantee the finding of a global minimum. However, for neural networks most of the problems are non-convex and finding a global minimum is not guaranteed. This means that the methods we will use to optimize, can most likely end up in a local minimum of the cost function.

Now, a cost function $J(\boldsymbol{\theta})$ can be seen as a function that says how good some parameters are at modeling a data set. Sometimes the cost function $J(\boldsymbol{\theta})$ will be called the loss function. When we refer to it as a loss function we are talking about it in some arbitrary calculation. So in an arbitrary calculation we will refer to the value of the loss function as the training loss, test loss or sometimes error loss. But as a general function we call it a cost function.

Our optimization method will look something like, taking data points, feeding them into the model, which gives us some prediction. Then we can compare the target variables the model has given us to the true target variables from the data set. We use the cost function for this comparison. The information we get from the comparison can then be used to update the model parameters in some optimal way. This process is then repeated many times until some stopping criterion is met (this will be covered later).

The method that will be used for updating the model parameters in an optimal way will be what is called gradient descent.

2.1.3 Gradient descent

Optimization will in our case be minimizing some function F that depends on parameters $\boldsymbol{\theta}$. This will be done by varying $\boldsymbol{\theta}$. The parameters that give us the optimal value will be denoted by $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} F(\boldsymbol{\theta})$.

We begin with an example. In the two dimensional case take the function $F(\boldsymbol{\theta}) = \theta_1^2 + \theta_2^2$. We call this a positively-definite function since it has one minimum which is the global minimum and lies at the point $\boldsymbol{\theta}^* = (0,0)$. Taking any other point $\boldsymbol{\theta}$ which is not $\boldsymbol{\theta}^*$ we want to somehow update $\boldsymbol{\theta}$ and reach the minimum $\boldsymbol{\theta}^*$. For this specific (convex) case it is just a matter of finding the roots of the derivative for the minimum. But, as we remember, the cost function for neural networks is rarely such that this can be done. This means that an iterative method has to be used.

The gradient $\nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta})$ of a function at a point $\boldsymbol{\theta}$ tells us the direction of most increase, meaning that $-\nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta})$ will give us the direction of most decrease. For some ϵ, called the learning rate, and $\boldsymbol{\theta} \neq \boldsymbol{\theta}^*$ we have that

$$F(\boldsymbol{\theta} - \epsilon \nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta})) < F(\boldsymbol{\theta}).$$

Consequently, for any starting point $\boldsymbol{\theta}$ we can calculate the gradient $\nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta})$ and a new point

$$\boldsymbol{\theta}' = \boldsymbol{\theta} - \epsilon \nabla_{\boldsymbol{\theta}} F(\boldsymbol{\theta}) \tag{2.2}$$

which should in theory give us a point lower in value than for $\boldsymbol{\theta}$. Doing this iteratively we should get that our final $\boldsymbol{\theta}'$ will be an approximation of $\boldsymbol{\theta}^*$.

2.1.4 Cost function and maximum likelihood

Previously we have mentioned a cost function as a means of evaluating how a model, with some specific parameters, performs on some data. Now we will go further into detail about the cost function and how it is derived. We will give two ways of interpretation.

Earlier it was said that the goal is to estimate some sort of true function f^*. This can also be though of as finding a distribution for the model that best fits the data's generating distribution. Thus, there exists two distributions, the real data generating distribution $p_{data}(\boldsymbol{x})$, which can be estimated with $\hat{p}_{data}(\boldsymbol{x})$, and the model distribution $p_{model}(\boldsymbol{x}; \boldsymbol{\theta})$. Somehow we want to get the model distribution $p_{model}(\boldsymbol{x}; \boldsymbol{\theta})$ as close to our estimate of the real one $\hat{p}_{data}(\boldsymbol{x})$. Using something called information theory we can quantify the (dis)similarity of two distributions and then use this to minimize the distance between them.

We refer the reader to chapter 3 in [3] for more details on information theory. However, the important thing to take away from information theory is that it gives us the ability to quantify a distance between two distributions.

Using information theory we can define the information of an event, namely self information.

Definition 2.1.1 (Self information)**.** Let $X \sim P$ be a random variable. We define self information of an event $X = x$ as

$$I(x) = -\log P(x). \tag{2.3}$$

Now, self information only gives us information about a single event. However, we would like to define something that quantifies information for an entire probability distribution, namely the Shannon entropy.

Definition 2.1.2 (Shannon entropy)**.** Let $X \sim P$ be a random variable. We define the Shannon entropy as

$$H(X) = \mathbb{E}_{X\sim P}[I(x)] = -\mathbb{E}_{X\sim P}[\log P(x)]. \tag{2.4}$$

Furthermore, we can define a measure for quantifying the difference in information between two distributions.

Definition 2.1.3 (Kullback-Leibler divergence)**.** Let $X \sim P$ and $X \sim Q$ be two distributions over a random variable. We define the Kullback-Leibler (KL) divergence as

$$D_{KL}(P\|Q) = \mathbb{E}_{X\sim P}\left[\log \frac{P(x)}{Q(x)}\right] = \mathbb{E}_{X\sim P}[\log P(x) - \log Q(x)]. \tag{2.5}$$

Then, using the KL divergence we can define cross entropy (The reason for this definition will be clear later).

Definition 2.1.4 (Cross-entropy)**.** Let $X \sim P$ and $X \sim Q$ be two distributions over a random variable. We define the cross-entropy divergence as

$$H(P, Q) = H(P) + D_{KL}(P\|Q) = -\mathbb{E}_{X\sim P}[\log Q(x)]. \tag{2.6}$$

With all of these definitions we recall that our goal was to make p_{model} as close to $\hat{p}_{data}$. Now we have a way of do so. Using the KL divergence we can measure the similarity (distance) in information between these two distributions and then minimize it somehow. Thus, using the KL divergence with $P = \hat{p}_{data}$ and $Q = p_{model}$ we get that

$$D_{KL}(\hat{p}_{data}\|p_{model}) = \mathbb{E}_{X\sim\hat{p}_{data}}[\log \hat{p}_{data}(\boldsymbol{x}) - \log p_{model}(\boldsymbol{x}; \boldsymbol{\theta})]. \tag{2.7}$$

So, we have reduced the problem down to minimizing the KL divergence. Furthermore, if we look at equation 2.7 we see that minimizing cross entropy with respect to $\boldsymbol{\theta}$ leads to minimization of the KL divergence, since in equation 2.6 $H(P)$ is just a constant independent of $\boldsymbol{\theta}$. Therefore we can further reduce our problem to minimizing the cross entropy between the estimated data distribution and the model distribution

$$H(\hat{p}_{data}, p_{model}) = -\mathbb{E}_{X\sim\hat{p}_{data}}[\log p_{model}(\boldsymbol{x}; \boldsymbol{\theta})]. \tag{2.8}$$

At this point it might not be entirely clear why we use cross entropy or what it means to minimize it. Another way of getting to the same answer is to use maximum likelihood. We begin by looking at the definition for maximum likelihood.

Definition 2.1.5 (Maximum likelihood)**.** Let $\mathbb{X} = \{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(n)}\}$ be a random sample of size n from our data distribution $p_{data}(X)$ and let $p_{model}(X; \theta)$ be a parametric family of probability distributions over the same space. The maximum likelihood estimator for $\boldsymbol{\theta}$ is

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} p_{model}(\mathbb{X}; \boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{n} p_{model}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}). \tag{2.9}$$

Using this definition and using the log likelihood instead of the regular likelihood we can simplify the problem. This can be done since taking the logarithm of a function does not change the optimization problem. Conveniently we get that

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \ \log\left(\prod_{i=1}^{n} p_{model}(\boldsymbol{x}^{(i)};\boldsymbol{\theta})\right) = \arg\max_{\boldsymbol{\theta}} \ \sum_{i=1}^{n} \log p_{model}(\boldsymbol{x}^{(i)};\boldsymbol{\theta}). \quad (2.10)$$

Dividing by n we finally get that

$$\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{X\sim\hat{p}_{data}}\left[\log p_{model}(\boldsymbol{x};\boldsymbol{\theta})\right]. \quad (2.11)$$

Instead of maximizing the term in equation 2.11 we can take the negation of it and then change the problem to a minimization one. So we want to minimize

$$-\mathbb{E}_{X\sim\hat{p}_{data}}\left[\log p_{model}(\boldsymbol{x};\boldsymbol{\theta})\right] \quad (2.12)$$

which we see from equation 2.8 corresponds to minimizing the cross-entropy between the data and model distributions.

Now since, our data consists of points and targets, the model and data distributions are conditional as in $\hat{p}_{data}(y\,|\,\boldsymbol{x})$ and $p_{model}(y\,|\,\boldsymbol{x};\boldsymbol{\theta})$. According to [4] (page 133) the maximum likelihood estimator can be generalized to this conditional case.

Consequently we have that our neural network will be trained using maximum likelihood. Which we saw comes down to minimizing the negative log-likelihood. Or as we also saw, minimizing the cross-entropy between the training data and our model distribution. Finally we say that the cost function is given by

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{X,Y\sim\hat{p}_{data}}\left[\log p_{model}(y\,|\,\boldsymbol{x};\boldsymbol{\theta})\right]. \quad (2.13)$$

For our case of regression we will assume that our model is normally distributed, so $p_{model}(y\,|\,\boldsymbol{x};\boldsymbol{\theta}) \sim N(y;\hat{f}(\boldsymbol{x};\boldsymbol{\theta}),\sigma^2)$. This gives us that

$$\begin{aligned} J(\boldsymbol{\theta}) &= -\mathbb{E}_{X,Y\sim\hat{p}_{data}}\left[\log \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y-\hat{f}(\boldsymbol{x};\boldsymbol{\theta})}{\sigma}\right)^2}\right] \\ &= -\mathbb{E}_{X,Y\sim\hat{p}_{data}}\left[\log\left(\frac{1}{\sigma\sqrt{2\pi}}\right) - \frac{1}{2}\left(\frac{(y-\hat{f}(\boldsymbol{x};\boldsymbol{\theta}))^2}{\sigma^2}\right)\right] \\ &= \frac{1}{2}\,\mathbb{E}_{X,Y\sim\hat{p}_{data}}\left[(y-\hat{f}(\boldsymbol{x};\boldsymbol{\theta}))^2\right] + constants. \end{aligned}$$

Here the constants are from the variance of the distribution and these we will assume are fixed. Toghether with the factor $\frac{1}{2}$ the constants can be dismissed since they do not change the minimization problem. Thus we get that the cost function reduces down to the mean squared error

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{f}(\boldsymbol{x}^{(i)};\boldsymbol{\theta}))^2. \tag{2.14}$$

2.2 Feedforward neural networks

To this point we have only been looking at the shell of our machine learning algorithm and now we will look at what will be under the shell.

Earlier we talked about a parametric family of functions $\hat{f}(\boldsymbol{x};\boldsymbol{\theta})$ that will be used to estimate the true function f^*. We will see that neural networks are precisely this parametric family of functions.

2.2.1 What is a neural network?

The major motivation for using neural networks is that a large enough network with a non-linear activation function can model any function. Or, at least, model the data (that represents the function) to a high precision.

We will here only introduce the basic feedforward neural networks where information flows forward from an input, through layers and in the end an output is given. The body of our neural network will be something called a multi-layer perceptron (MLP). It will consist of so called hidden layers, together with an output layer. The hidden layers are simply put, a general linear transformation and translation. This linear transformation together with a translation is sometimes called an affine transformation. lastly, between hidden layers, information is activated by an activation function.

Let $\boldsymbol{x}$ be a data point in the data set which will be the input to our MLP. Let y be the the target variable or output and $\boldsymbol{h}^{(i)}$ the vector at hidden layer i. Starting from the input layer, the vector $\boldsymbol{x}$ is fed into the first hidden layer where it undergoes a linear transformation and translation $\boldsymbol{h}^{(1)} = W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}$ by a matrix W and vector $\boldsymbol{b}$. The vector in the first hidden layer $\boldsymbol{h}^{(1)}$ is then activated by an activation function g and we get the output of the hidden layer as $\boldsymbol{z}^{(1)} = g(\boldsymbol{h}^{(1)})$. This is then fed into the next layer where the process is repeated. This process is iterated for some $m > 0$ number of hidden layers until the output layer where we get the output. The output is again an affine transformation to one variable $y = W^{(m+1)}\boldsymbol{z}^{(m)} + \boldsymbol{b}^{(m+1)}$, however this layer lacks an activation by g.

Regarding the activation function g, there is no unique choice, rather it is a group of non-linear functions that we can choose from. This will be further investigated later.

With neural networks comes quite some jargon. Sometimes we may refer to matrices $W^{(i)}$ as weights and vectors $\boldsymbol{b}^{(i)}$ as biases. The procedure or process of calculating an output is called forward propagation. In addition, sometimes a hidden layer can be seen as a collection of d_i number of units. A unit is sometimes also called a neuron. We say that d_i gives us the dimension of the hidden layer i. The calculations that would happen in each unit $j \in 1, ..., d_i$ of a hidden layer i would be

$$\begin{aligned} j = 1 \ &: \ h_1^{(i)} = W_1^{(i)} \boldsymbol{z}^{(i-1)} + b_1^{(i)} \\ j = 2 \ &: \ h_2^{(i)} = W_2^{(i)} \boldsymbol{z}^{(i-1)} + b_2^{(i)} \\ &\vdots \\ j = d_i \ &: \ h_{d_i}^{(i)} = W_{d_i}^{(i)} \boldsymbol{z}^{(i-1)} + b_{d_i}^{(i)} \end{aligned}$$

where $W_j^{(i)}$ is the j'th row of matrix $W^{(i)}$ and $b_j^{(i)}$ the j'th element of vector $\boldsymbol{b}^{(i)}$. Lastly, we say that a unit is "active" if it produces a non-zero (and non-negative) value after the activation function. Thus, a unit with value zero after the application of ReLU is "non-active". The process of turning a non-active unit active is called activating the unit.

The thought behind this type of network is that affine transformations are useful for translations, rotations, reflections et cetera. However, the linear transformations lack the ability to model non-linearites. Since we want to be able to model non-linearites, an activation function is applied after the affine transformation.

The output layer will be different depending on if the problem is a regression or classification problem. As we saw above, in the case of regression the output layer will be an affine transformation from the previous layer to one target variable y (you can have multiple targets but as one might remember we restricted ourselves to one target variable).

This is the basic feedforward neural network. An illustration of a neural networks is given in figure 2.1.

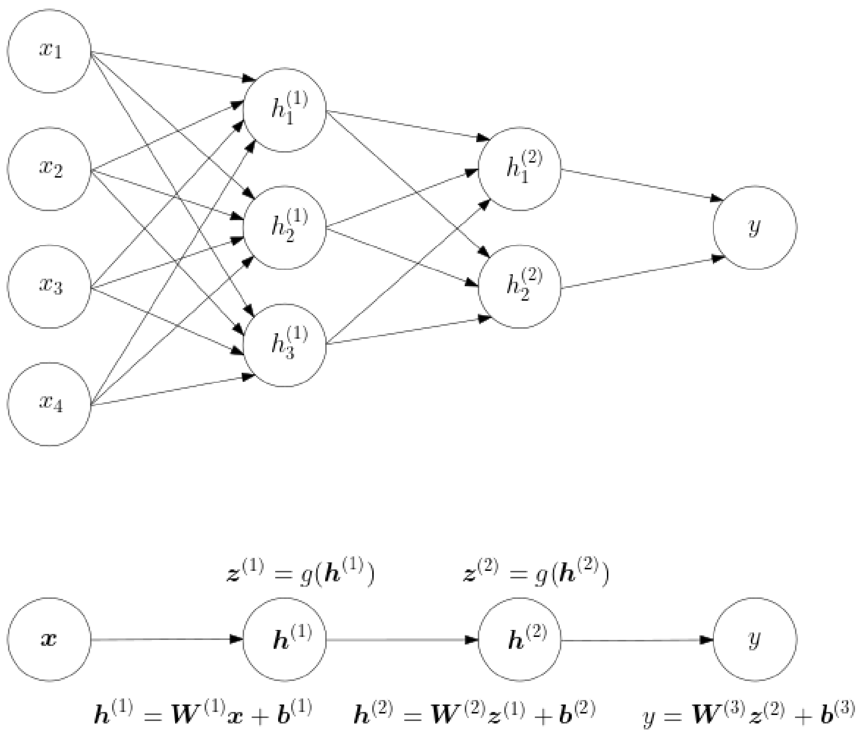
2.2.2 Activation function

As earlier said, activation functions enable the modeling of non-linearities. There are several common activation functions that one can use. In our case we will use Rectified Linear Unit (ReLU) as an activation function. It is defined for some input h as

$$g(h) = \max\{0, h\}.$$

As we see, ReLU sets all negative values to zero and keeps positive values as they are. In the case of vectors it is applied element-wise.

Looking at the ReLU function we see that it has a very simple derivative, either one or zero (if one sets the derivative equal to one at $x = 0$). In turn this gives the network a gradient that is quite simple to calculate.



x_1
x_2
x_3
x_4
$h_1^{(1)}$
$h_2^{(1)}$
$h_3^{(1)}$
$h_1^{(2)}$
$h_2^{(2)}$
y
$\boldsymbol{z}^{(1)} = g(\boldsymbol{h}^{(1)})$
$\boldsymbol{z}^{(2)} = g(\boldsymbol{h}^{(2)})$
$\boldsymbol{x}$
$\boldsymbol{h}^{(1)}$
$\boldsymbol{h}^{(2)}$
y
$\boldsymbol{h}^{(1)} = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}$
$\boldsymbol{h}^{(2)} = \boldsymbol{W}^{(2)}\boldsymbol{z}^{(1)} + \boldsymbol{b}^{(2)}$
$y = \boldsymbol{W}^{(3)}\boldsymbol{z}^{(2)} + \boldsymbol{b}^{(3)}$

Figure 2.1: This is the general body of an MLP. Both figures represent the same MLP. In the upper figure each circle represents a neuron whereas in the lower one a circle represents a whole hidden layer. The x-values are the inputs, the h-values the hidden units and y is the target value. In the figure below we see how each next layer is calculated from a previous one. The function g is the activation function and preceding layers are calculated using a linear transformation by $W^{(i)}$ and translation by $\boldsymbol{b}^{(i)}$.

Compared to a sigmoid activation function, defined as

$$\sigma(h) = \frac{e^h}{1 + e^h},$$

ReLU also has the nice property of not saturating as much. For the sigmoidal function we see that for values $|h| > 4$ the function is roughly zero or one. This means that on most of the domain the sigmoidal function has essentially zero-valued gradients. Now, gradients with good values (large enough) means a more stable and efficient training procedure since the training procedure can continue. Looking at the update of gradient descent, gradients equal or close to zero means a small update of the parameters and thus a procedure that has halted.

As with the sigmoid function, ReLU has the same "zero gradient" problem for values h less than zero. However this is not considered a saturation case since half the domain of ReLU still gives a unit derivative which overall contributes to a stable gradient. If we can keep most data points with postive values then saturation should not be a problem. However, if a hidden layer produces an output consisting of all negative values then the gradient will be zero. This means that gradient descent stops learning. This can be solved by altering ReLU such that no zero gradient exist, for example using a Parametric Rectified Linear Unit (PReLU), defined as

$$g(h) = \begin{cases} h & \text{if } h \geq 0 \\ \alpha h & \text{if } h < 0 \end{cases}$$

where $\alpha \in \mathbb{R}_{>0}$. Another solution for this problem will be seen in section 2.3.1 (Batch normalization).

2.2.3 Architecture

Looking at the introduction of neural networks in section 2.2.1 one can see the many ways of structuring a neural network. We refer to the structure of a network as architecture. We will focus on two specific parameters of architecture, the amount of hidden layers a network has and the dimension of each hidden layer.

The universal approximation theorem tells us that any function can be modeled using a neural network with only one hidden layer. But dependent on the complexity (non-linearity and so on) of the function, the number of units in this one hidden layer can be ten or ten million. This is where the motivation for more hidden layers comes in. It has been shown by [5] that a deeper network with piece-wise linear activation units may need considerably fewer units (in total) compared to a network with only one hidden layer.

The main reason for this considerable difference is due to folding of the input space. Folding can best be described by a small example. Assume that we have a network with one input, one output, two hidden layers and we use ReLU as an activation function. In the first layer we have some d number of units. Thus the function produced by this layer will be one with d linear regions. Further, the second layer gives rise to some function and what essentially happens in forward propagation is that this function is superimposed with each of the linear regions. This can be seen as we are folding the input space in the first layer, then applying some function on top of that and lastly unfolding. This is illustrated in figure 2.2. If we have multiple (more than two) hidden layers in a row then we can see it as a chain of folds. Essentially what happens is that the function is built up in a way where firstly, the network produces the "simplest" version of the function. Then for each fold it builds more details on top of that simple function. See it as we are constructing a house, where we first lay the groundwork, then build up the body of the house with walls and such and lastly add decorations, et cetera.

Additionally, looking at pictures of feedforward networks one will notice that networks have a specific shape (there are some exceptions). First the dimension is increased as we go along hidden layers and then decreased. This is tied to what we described as folding. As we explained with this successively build up of a function for each layer that is added we would like a finer precision. This is accomplished by making the hidden layers larger (wider). Then we usually have one or a few target variable(s) and thus the dimension of the hidden layers needs to eventually shrink and match the dimension of the output unit.

2.3 Extending the standard MLP

There are some simple improvements that one can make to a MLP that can improve the performance. Meaning that we find a better minimum faster and that the training procedure is more stable.

2.3.1 Batch normalization

As we saw it is desired to have multiple hidden layers. But due to the multiplicative structure of the network a problem with data points $\boldsymbol{h}$ being shifted and rescaled can occur.

Later we will see that forward propagation will be carried out for the so called minibatches of vectors. So in each hidden layer i (step of forward propagation) we will have a small batch of n' number of vectors $\mathbb{B}_i = \{\boldsymbol{h}^{(i,1)}, \boldsymbol{h}^{(i,2)}, ..., \boldsymbol{h}^{(i,n')}\}$ with $n' << n$ where n is the total number of samples in the training set. This shifting and rescaling problem can then be solved by normalizing each of these batches. This is done by batch normalization which we now define.

Definition 2.3.1 (Batch normalization)**.** Let $\mathcal{BN}$ be the function applied in a batch normalization layer. For an element $\boldsymbol{h}$ in a minibatch $\mathbb{B}$ batch normalization is defined as

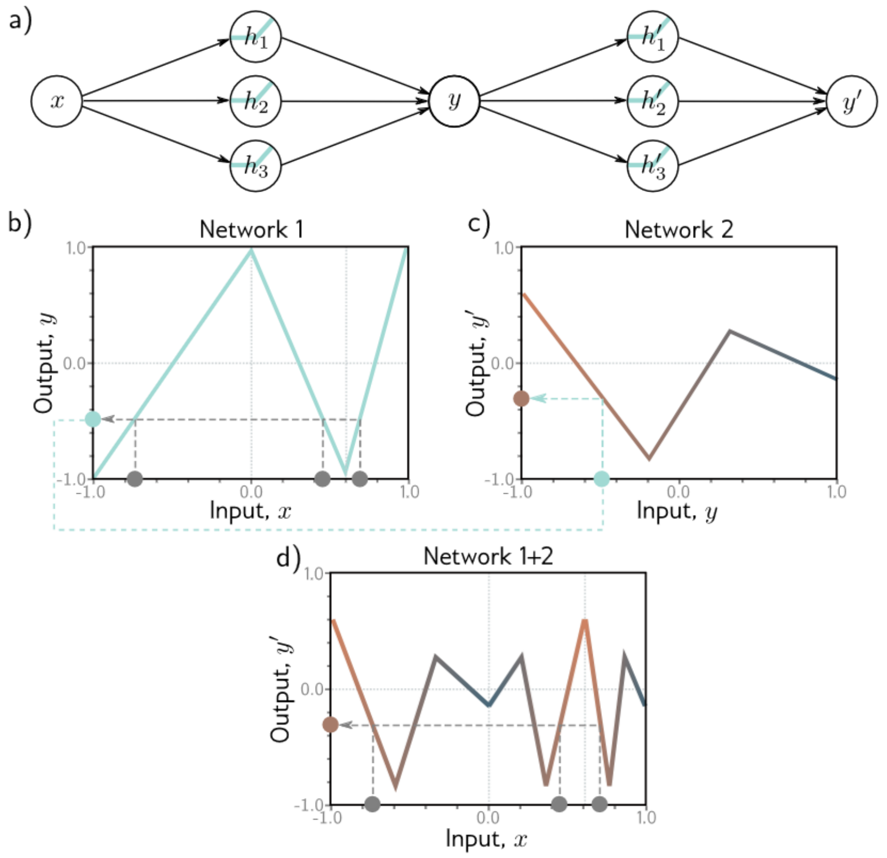


Figure 2.2: In a) we see the structure of this particular neural network. We can also look at it like two networks, one taking points x and mapping them to targets y and one taking points y and mapping them to targets y'. In b) we see that three points (gray dots) are mapped to the same value (blue dot). This value is then in c) mapped to another value in y'. In d) we see the whole mapping x to y'. Notice how the figure of d) is taking the line in c) and applying it on each linear segment in figure b). Essentially, the mapping in c) has been superimposed with the mapping in b) folded two times. Figure taken from [6] (page 42).

$$\mathcal{BN}(\boldsymbol{h}) = \boldsymbol{\gamma} \cdot \frac{\boldsymbol{h} - \overline{\boldsymbol{\mu}}_{\mathbb{B}}}{\overline{\boldsymbol{\sigma}}_{\mathbb{B}}} + \boldsymbol{\beta}. \tag{2.15}$$

where division by $\overline{\boldsymbol{\sigma}}_{\mathbb{B}}$ and multiplication by $\boldsymbol{\gamma}$ is done pairwise.

Here $\overline{\boldsymbol{\mu}}_{\mathbb{B}}$ is the sample mean of the minibatch $\mathbb{B}$ given by

$$\overline{\boldsymbol{\mu}}_{\mathbb{B}} = \frac{1}{|\mathbb{B}|} \sum_{\boldsymbol{h} \in \mathbb{B}} \boldsymbol{h}$$

and $\overline{\boldsymbol{\sigma}}_{\mathbb{B}}$ the sample variance of the same minibatch given by

$$\overline{\boldsymbol{\sigma}}_{\mathbb{B}} = \frac{1}{|\mathbb{B}|} \sum_{\boldsymbol{h} \in \mathbb{B}} (\overline{\boldsymbol{\mu}}_{\mathbb{B}} - \boldsymbol{h})^2 + \epsilon.$$

The small constant $\epsilon > 0$ is added to ensure that divition by zero is not attempted.

As we see in equation 2.15 each batch is normalized to zero mean and unit variance. However, the values in $\boldsymbol{g}$ are then rescaled to have mean $\boldsymbol{\gamma}$ and variance $\boldsymbol{\beta}$. These parameters are learnable and have the same shape as $\boldsymbol{h}$. The parameters $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ are needed for the network to find a mean and variance that works well across all minibatches.

As explained in [7] (section 8.5.2.1) there have been applications of batch normalization where it is inserted before the activation function and applications where it is inserted after the activation function. In our case we will implement batch normalization before the activation function. The motivation for this is closely tied to the problem we encountered in section 2.2.2 when talking about ReLU not saturating. As we saw, it is the affine transformations that impose the value shifting and rescaling which batch normalization then inverts or at least shifts data points closer to the origin. Now, if enough values are shifted back into the positive domain then the SGD algorithm could in the next iteration produce a gradient that activates non-active units. Meaning that batch normalization further decreases the chance of a possible ReLU saturation.

2.3.2 Skip connections

In a MLP, for each layer that is added, the previous space of functions is not a proper subset of the new one. Say that for one hidden layer we get a space of functions $\mathcal{F}_1$. Adding another hidden layer would result in another space of functions $\mathcal{F}_2$ and so on. Since data has to pass more layers the network will lose the ability to model some of the simpler functions. Mathematically speaking we have that $\mathcal{F}_1 \not\subset \mathcal{F}_2 \not\subset \mathcal{F}_3 \cdots$. We can think of it as, the set of possible functions is moving in some direction for every hidden layer that is added. This movement occurs in some unknown direction in function space. Since the space of functions is (very) large there is a highly small probability that this set will move toward the true function.

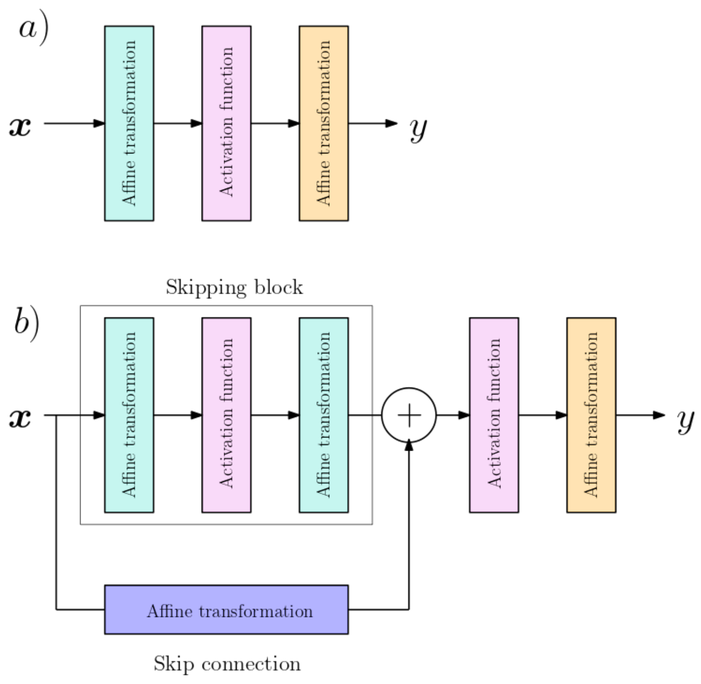


Figure 2.3: In a) we have a network with a single hidden layer (turquoise and violet) and an output layer (orange). In b) we have taken the network in a) and added another hidden layer. Additionally we have attached a skip connection (blue) from the input $\boldsymbol{x}$ of the "skipping block" into the output of the block. In the skip connection an affine transformation is applied and then summed with the output of the "skipping block". Now, if we were to turn of the "skipping block" by enforcing a zero mapping, the network in b) would be identical to the one in a). However, the network in b) also has the possibility of turning off (zero mapping) the skip connection and thus acting as a network with two hidden layers.

Take the same example of having one layer and adding another. However, this time, as in figure 2.3 b), we impose a mechanism for skipping a layer. Then, the model is able to skip a layer and only use one of the layers. So the network will then be able to model the space of functions $\mathcal{F}_1$. However, it would also be able to ignore the skipping mechanism and use both layers to model the space of functions $\mathcal{F}_2$. With the inclusion of the skipping mechanism, whenever we add a hidden layer we are increasing the space of functions that the network is able to model in all dimensions. Mathematically speaking we have that $\mathcal{F}_1 \subset \mathcal{F}_2 \subset \mathcal{F}_3 \cdots$. Theoretically, as illustrated in figure 2.4, for every hidden layer that is added the space of possible functions should be moving closer to the true function.

It is important to remember that the dimension of the skip connections must match the dimension of the hidden layer where the skip connection is added. This is the reason an affine transformation is applied in the skip connection, as seen in blue in figure 2.3. Additionally, it is not required that the skip connection only skips one hidden layer.

The original application of a mechanism for skipping hidden layers was done for a convolutional neural network called ResNet [2]. In such a network the skip connections are implemented such that the network has to learn a "residual mapping". However, as we see in figure 2.3, this is not needed for our case since the network can simply learn a zero mapping.

2.4 Optimizing the neural network

Now we present how we are going to optimize the neural network.

2.4.1 Stochastic gradient decent (SGD)

As said earlier, gradient descent will be used for optimization. We recall that the goal was to optimize

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{f}(\boldsymbol{x}^{(i)};\boldsymbol{\theta}))^2.$$

For gradient descent it is needed to calculate the gradient which will be

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\nabla_{\boldsymbol{\theta}}(y^{(i)} - \hat{f}(\boldsymbol{x}^{(i)};\boldsymbol{\theta}))^2.$$

for the cost function $J(\boldsymbol{\theta})$.

In this case n gradients need to be computed. Now, if n is large and the model has many parameters this calculation will take a lot of time. To reduce this time we can split our data set into so called minibatches $\{\mathbb{B}_1, \mathbb{B}_2, ..., \mathbb{B}_m\}$ each consisting of n'

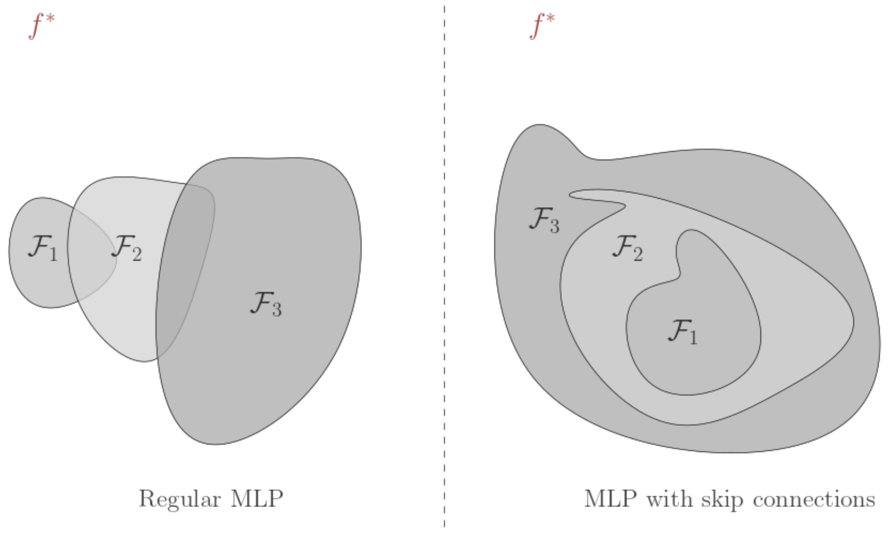


Figure 2.4: The left picture shows what adding layers to a neural network most likely will have in effect. We see that the space of functions that a network can model is moving in some direction in function space, but in this case not in the direction of the true function f^*. In the right picture we see the same procedure but with a skip connection implementation. Instead of moving the function space in one direction the function space increases in multiple directions. What essentially happens is that the network is more likely to move in a direction closer to the true function. Picture inspired by figures from [7] (section 8.6.1).

samples. Thus, if we have n samples and a size n' there will be a total of $m = \lceil \frac{n}{n'} \rceil$ minibatches. Consequently this is no longer an accurate estimation of the gradient, rather a rough estimate. The estimate will be given by

$$\boldsymbol{g}_{t'} = \frac{1}{n'} \nabla_{\boldsymbol{\theta}} \sum_{i \in \mathbb{B}_{t'}} (y^{(i)} - \hat{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}))^2.$$

where $t' \in \{1, ..., m\}$. Using this estimated gradient the parameters are updated as

$$\boldsymbol{\theta}_{t'+1} = \boldsymbol{\theta} - \epsilon_{t'} \boldsymbol{g}_{t'}.$$

In this way, instead of having the computational cost $\mathcal{O}(n)$ for each update of the parameters it has been reduced down to $\mathcal{O}(m)$, for each update. Thus, instead of computing the exact gradient and making one update of the parameters the gradient is estimated and in the end m updates of the parameters are made. This method is what is called minibatch stochastic gradient descent.

Another maybe surprising effect of SGD is that the stochasticity itself actually helps the algorithm to find a better local minimum. Since the problem is in general non-convex there may exist multiple local minima. Furthermore if one has a situation as in figure 2.5, regular gradient descent would get stuck in minimum "M2". While SGD would most likely jump around in the minimum and could by chance escape this shallow minimum. However one could say that there is a possibility for SGD to estimate the gradient in a way that would make it jump up multiple times and into a higher local minimum. This is a possible situation but is highly unlikely since we expect the negative gradient to be pointing "downwards".

2.4.2 Back propagation

A problem with gradient descent for neural networks is that the network is essentially many nested functions which in turn gives complicated gradients. For example, taking a three layer MLP the final model might look something like $\hat{f}(x) = s(h(k(x)))$. Finding the gradient for this can be hard. The solution for this is what we call back propagation.

Essentially, back propagation is an application of the chain rule. The whole algorithm for this is quite technical and it is not needed to understand double descent. Hence, we refer to other literature for explanation. For example, section 6.5 in [4].

2.4.3 ADAM

We said that SGD will be used for optimization. However, a variant of SGD will instead be used, namely ADAM.

ADAM is a short name for "adaptive moments". It implements moments and momentum which in turn have some important effects on the learning procedure. Here we will

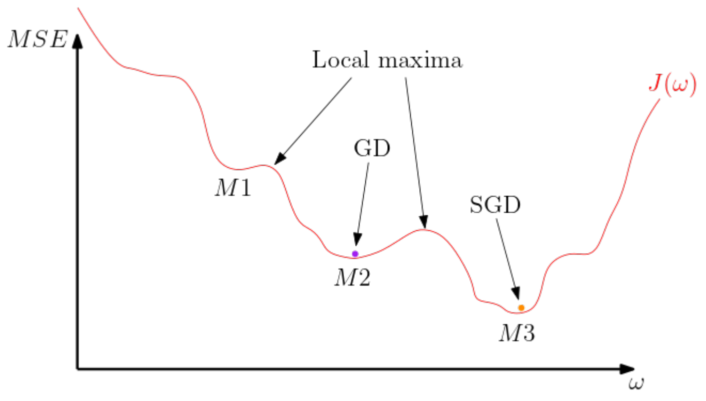


Figure 2.5: The red line represents the actual topology of the cost function, MSE stands for mean squared error and ω is a parameter in some theoretical model. This figure is a scenario where stochastic gradient descent (SGD) would perform better than regular gradient descent (GD). Say that both algorithms start near $\omega = 0$. Then GD could probably jump over the first local maximum (left one) but it would have a problem with the second one (right one). Whereas, SGD when randomly jumping around could overcome that second maximum and find a lower minimum.

briefly explain the intuition of momentum and how it solves some problems. We refer to section 8.5 in [4] for detailed explanation of momentum and [8] for an explanation of moments.

The word momentum is very much tied to the physical phenomenon having to do with velocity. In regular gradient descent the algorithm does not take into account the gradients of previous parameter updates. But with the implementation of momentum this is exactly what is done. The effect of this is an algorithm that learns faster and one that can slow down once a good minimum is reached. It does not overshoot the minimum and also helps the algorithm to move in directions where the gradient is small. Take for example a ball rolling down a hill with valleys, the steeper the hill is at some point the more velocity the ball is going to gain. Depending on the environment, the ball might have enough momentum to skip some small valleys but once it reaches a big enough valley it will find the lowest point in this valley.

Additionally momentum solves an issue that SGD has with long and narrow valleys with steep walls. This is depicted and explained in figure 2.6.

2.4.4 Initialization

The initialization of the weights and biases in the neural network can have large effects on finding a good minimum. There are two important issues that might arise from a

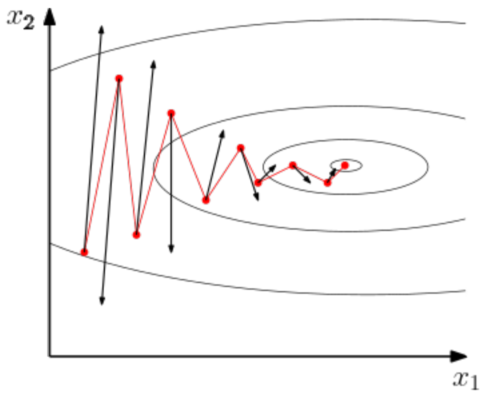


Figure 2.6: The figure represents a contour plot of a convex function with two variables. The red lines indicate the path that a SGD algorithm with momentum takes. The arrows depict the parameter update that a normal SGD algorithm would do in that specific point. We see that the Hessian matrix has small values in the x_1-direction and large in the x_2-direction. This means that a SGD algorithm without moments would have to take a lot of steps rocking back and forth in the x_2-direction and slowly moving in the x_1-direction. But an algorithm with momentum would gain velocity in the x_1-direction and thus faster reach the minimum.

badly initialized network.

The first problem is one with symmetry. As an example let us look at two arbitrary units in a arbitrary hidden layer. If we initialize both units identically, then a deterministic training procedure will update these units the same way. An easy way of solving this is by initializing the parameters randomly. However, one might remember that we will be using a stochastic (non-deterministic) algorithm for optimization. Even if a stochastic algorithm is used, as explained in [4] (page 301) a random initialization is still preferred. Since then, each unit would have to compute a different function which may help the algorithm from getting stuck in an unfavorable area of the search space.

The second problem arises if, specifically the weights are initialized to large numbers. Since the foundation of the network is built on nested matrix multiplication, too large weight parameters may lead to exploding values in hidden layers. This is unwanted since computers are limited to working with a certain size of numbers.

Another issue occurs when the values of weight parameters have a large variance. This could lead to some units dominating over others. For example say that we have two units A and B in some layer. Unit A produces a large number and unit B produces a value close to zero. This is due to the fact that the unit in B has small weight values and the unit in A large weight values. When the values of A and B are added into preceding layers the value of unit A will dominate and essentially treat value B as zero. This means that initializing with a large variance on weight values might lead to some units being non-active before even starting any training.

In our case the second problem will not really be an issue since we use batch normalization after every affine transformation. Since in a batch normalization layer the variance is normalized to unit variance.

For our neural network we will use a random initialization from a Gaussian with mean zero and variance 0.1 for the sake of convenience. According to [4] (page 302) the choice of distribution does not matter that much.

2.5 Final training procedure

Now that we have all the pieces we can finally assemble them together and define a training procedure. A training procedure $\mathcal{T}$ will be a neural network as described above and an optimization algorithm. This training procedure is then trained on a training set S.

The first step of the procedure is to split the data into minibatches $\mathbb{B}_1, \mathbb{B}_2, ..., \mathbb{B}_m$. Then iteratively for each minibatch we forward propagate, to calculate the loss, then backward propagate to calculate the gradient and lastly update the model parameters according to the ADAM algorithm. An iteration over all minibatches is called an epoch.

Training will then proceeded until some stopping criterion is met. In this study, we employ early stopping to monitor the training and validation/test errors. In early stopping a small validation/test set is created and the loss over this set is calculated after an epoch. Then we say that the early stopping criterion is met and stop training whenever this validation error does not decrease for some number of epochs, say ten or twenty. But in our double descent experiment we want to train beyond what such a criterion would allow us to and thus we will set the stopping criterion to a specific large number, for example, 4000 epochs. This number is somehow based on our experience on how many epochs is needed for the network to reach the lowest test error. Then it is just a matter of deciding on a number much larger than this.

Finally, after stopping the training procedure, we get a classifier $\mathcal{T}(S)$.

2.6 Model evaluation and generalization

2.6.1 Bias variance trade-off

When a machine learning model is optimized it is done so on one data set, the training data. However, it is important to consider how the model is going to perform on

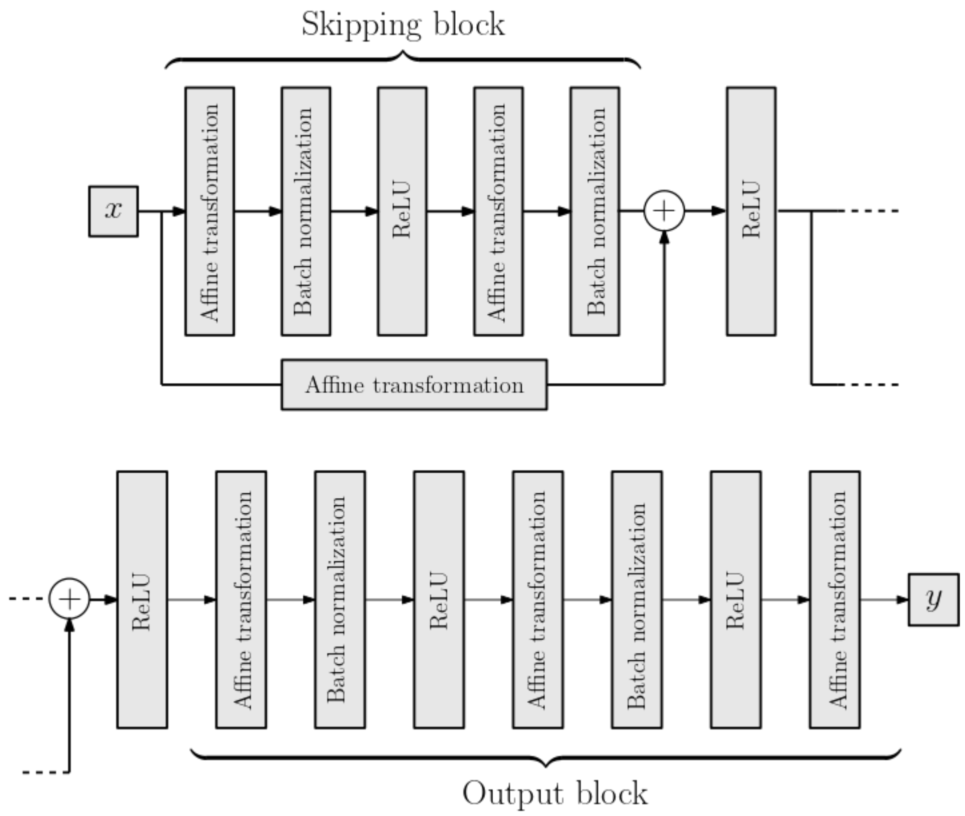


Figure 2.7: The figure shows the general structure of the network that will be used in our experiments. Each hidden layer is represented by an affine transformation a batch normalization step and a activation step ("ReLU" in figure). We have also visualized the skip connection which is an affine transformation that is then added in the preceding layer before the activation. We call a block with one skip connection a skipping block and the final block that does not implement a skip connection the output block. Notice that the output block contains a output layer which is an affine transformation to one variable y.

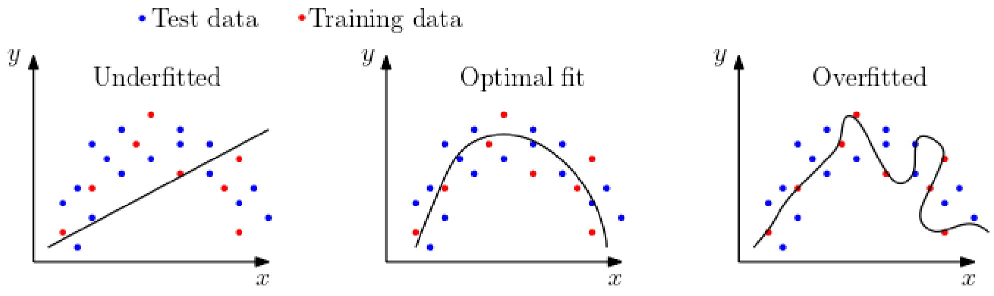


Figure 2.8: The leftmost figure shows a severely underfitted model. The central figure shows an optimal fit and the rightmost a severely overfitted model.

new unseen data. We call this validation or test data. With the cross entropy as loss function we thus get a test error or test loss. This test error can be split into two parts, bias (squared) and variance. Bias is how far away we expect the predictions to be to the data and variance is how varying the predictions are expected to be. Say, for example, that our true function is a polynomial of degree five. Then using polynomial regression as a model we see that a linear model (polynomial of degree one) would be biased but have low variance. If the model is a polynomial of degree five then the fit will be optimal. A polynomial of degree higher than five will have low bias since it can fit the training data perfectly. However, it will have higher variance than the model of degree five. This means that there is some model that results in an optimal fit and then there are models that could either underfit or overfit the data. See figure 2.8 for an example of an optimal fit and two extreme over- and underfitted cases, respectively.

As we saw with the example of polynomial regression, altering the "compleixty" of a model, changes the bias and variance. The relationship, as one can observe in figure 2.9 is that for simpler models (low complexity) the bias is high yet variance is low and as model complexity increases bias decreases and variance increases. This is termed as bias variance trade-off. Now, since test error is a combination of bias and variance we have that model complexity alters the test error and results in a curve as in figure 2.9.

2.7 Model complexity for neural networks

As we saw complexity is the key player in bias variance trade-off. Complexity is also needed for comparing different machine learning models with each other. Say we have two different models A and B and know that A and B have the same complexity. Then we can see how they perform on a specific task and compare them with each other. It is inappropriate to compare models with different complexities since it can not be decided whether it is the complexity or the model itself that causes a difference between the models.

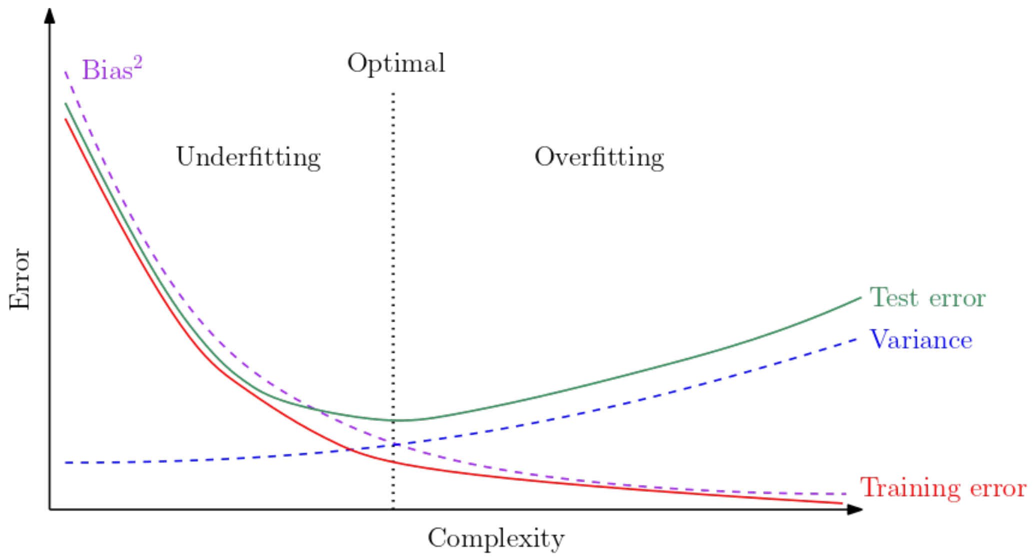


Figure 2.9: This is a training curve that we expect to see. As we increase the complexity we first go through a stage of underfitting where the model is still learning the data and then a stage of overfitting where the model fits too much to the training data.

For neural networks there is however a problem, since there is no obvious measure of model complexity. In early stopping (chapter 7.8 figure 7.3) [4] the training time (i.e number of epochs) serves the role of model complexity and in [1] (section 5) the model size (width) is treated as a measure for complexity. Now, the problem is that we want a more general measure of complexity. This is where we can define the Effective Model Complexity (EMC).

2.7.1 Effective model complexity (EMC)

Definition 2.7.1. The *Effective Model Complexity* of a training procedure $\mathcal{T}$ with respect to distribution $\mathcal{D}$ and a parameter $\epsilon > 0$, is defined as

$$EMC_{\mathcal{D},\epsilon}(\mathcal{T}) \coloneqq \max\{n \mid \mathbb{E}_{S\sim D^n}[MSE_S(\mathcal{T})] \leq \epsilon\} \tag{2.16}$$

where $MSE_S(\mathcal{T})$ is the mean squared error of training procedure $\mathcal{T}$ on n number of samples $S = \{(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)\}$. Definition taken from [1].

A problem with this definition is that ϵ can vary depending on what kind of model one uses. Thus we would sometimes want a simpler definition for EMC. From now on we will define it as the maximal number of samples such that the model has expected training loss (over the data distribution) roughly equal to zero.

At first glance it may seem that giving a model more data should decrease the training error forever. However, this is not true if the model is simpler than the underlying function it is trying to fit. This is where the intuition of EMC comes from, namely that a model has some sort of complexity and can only fit a certain number of training samples near perfectly. As we can see in figure 2.10, after increasing the sample size enough, the model can no longer fit all the training data perfectly, since it is not complex enough, it is too simple.

The reason EMC is defined as an expected value over different data sets is because data is randomly sampled and thus MSE is a random variable. This means that we have to report it as a summary statistic of the random variable. For example if the data set happens to consist of sample points in one simple part of the function this will not give a representation of the complexity over the whole function, rather a representation of that one simple part of the function.

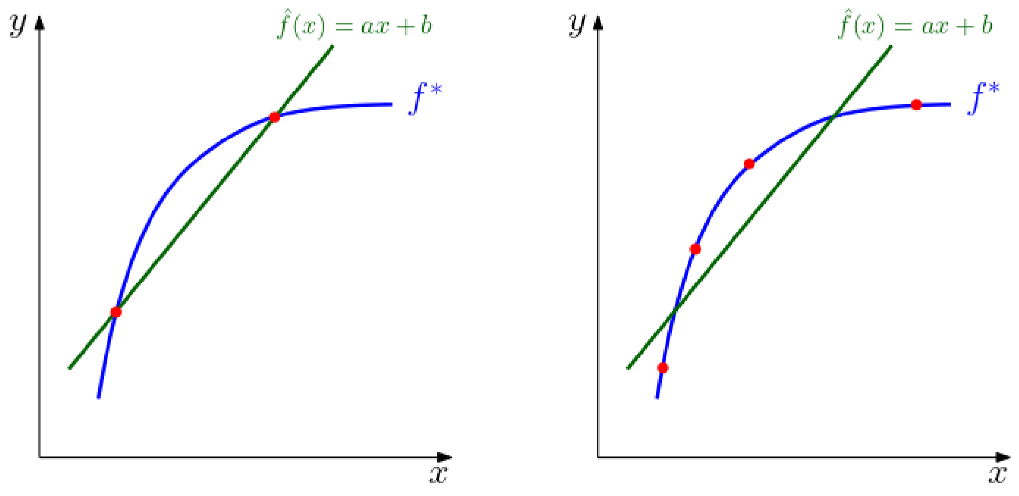


Figure 2.10: The left graph shows a model $\hat{f}$ that has been fit using the two sampled data points marked in red. The data points have been sampled from the true function f^* (assume no irreducible error ϵ). Here we have a perfect fit and the training error will always be zero for any two sampled points from f^*. The right graph shows the same model but one that has been fit on more data points. By increasing the number of data points we have increased the training error.

As a further way of understanding EMC we can look at how EMC reduces down to the order of the polynomial for polynomial regression. Take the case where we have a data set $S = \{(x_1, y_1), ..., (x_n, y_n)\}$ and a model $\hat{f}(x; \boldsymbol{\beta}) = \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \cdots + \beta_p x^p$. Here p is the order of the polynomial and n the number of data points. Fitting this model is done by solving the system of equations

$$\boldsymbol{y} = \boldsymbol{X\beta} \iff$$

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 & x_1^2 & \cdots & x_1^p \\ x_2 & x_2^2 & \cdots & x_2^p \\ \vdots & \vdots & \ddots & \vdots \\ x_n & x_n^2 & \cdots & x_n^p \end{pmatrix} \times \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{pmatrix}.$$

Now, there are three potential cases regarding solutions for different sizes of n and p. We can have that $n < p$, $n = p$ or $n > p$. For the first two we will either have an underdeterimed system of equations or a system where we have a unique solution. For the underdetermined system of equations we will either have infinitely many solutions or none. For both the underdetermined and uinque solution case the solution will fit the data such that the training loss is zero. However, when $n > p$ we have an overdetermined system of equations which does not have any solution. Rather we will have to estimate a solution and so the training error will no longer be zero.

Thus the maximal number of samples for when the polynomial model has training error zero, or the EMC of it, will be the order of the polynomial.

2.7.2 Estimating EMC.

The method for estimating EMC will firstly be averaging over multiple training sets. Furthermore we have to take into account the fact that a training procedure contains randomness. Thus averaging over multiple training procedures with different initialization but on the same data set is needed. If we average over δ number of training sets and initialize ι number of different networks for each training set, the total number of classifiers will be $\delta \cdot \iota$.

The estimation of expected training error will be as following. Assume that we have some training sets $S^{(1)}, S^{(2)}, ..., S^{(\delta)}$ and training procedures $\mathcal{T}^{(1)}, \mathcal{T}^{(2)}, ..., \mathcal{T}^{(\iota)}$. The estimated expected training loss of this ensemble at training step (epoch) t will be

$$\hat{\mu} = \frac{1}{\delta \cdot \iota} \sum_{j=1}^{\delta} \sum_{k=1}^{\iota} MSE_{S^{(j)}}(\mathcal{T}_t^{(k)})$$

where MSE is the mean squared error on the given data set of the particular training procedure.

Now, from the more relaxed definition of EMC we also have to define what it means in our case that the expected training loss is roughly equal to zero. Assume as a null hypothesis that the expected value of a model that has zero training error is $\mu_{error} = \sigma_{noise}$. The reason for choosing σ_{noise} is that it is somehow a measure of the training error for an optimal model. See figure 2.11 for a further explanation. We can

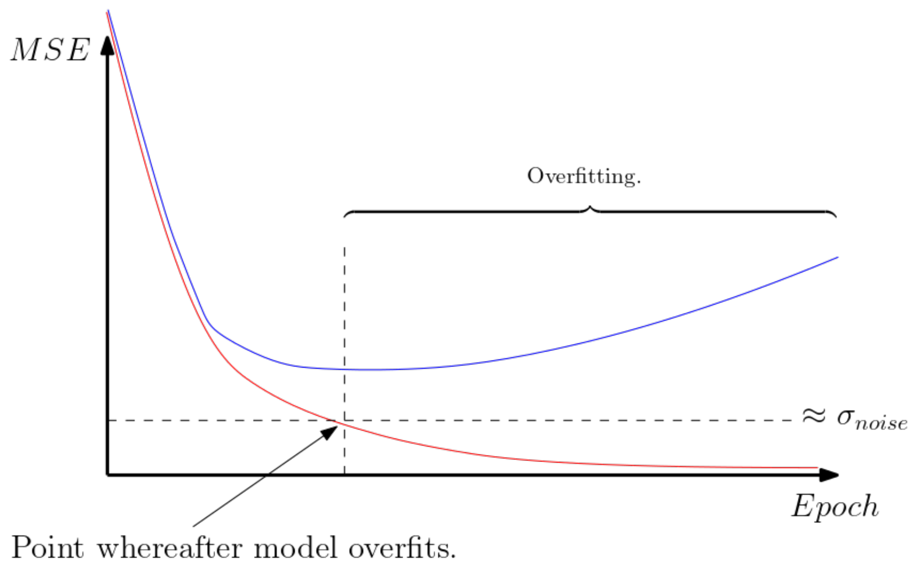


Figure 2.11: From the bias variance trade-off we know that a model starts to overfit at the marked point in the figure. Assuming that at this point we have a fit that is close to the true function f^* (like the middle plot in figure 2.8) we know that the MSE of the training error will roughly be equal to σ_{noise}. Thus, σ_{noise} is an approximation for training error for when a model is overfitted.

then test our test statistic $\hat{\mu}$ against μ_{error}. We will use a confidence interval with a 99% confidence level for this. In the case where the null hypothesis is not rejected we say that the model has training error roughly equal to zero.

2.8 Double descent

It has been shown [1] that the bias variance trade-off from section 2.6.1 does not show the whole picture of how large neural networks generalize on new unseen data. Specifically, for a classification problem, there is evidence that test error decreases again after some threshold, as shown in figure 2.12. The paper [1] shows that double descent happens epoch-wise and model-wise as shown in figure 2.12. Epoch-wise refers to a relation between test error and training time for a fixed model. Model-wise refers to a relation between test error and model size (depth and width) for a fixed epoch.

To test if double descent happens for regression we are going to train different sized networks beyond what early stopping would allow. We will focus on the epoch-wise double descent.

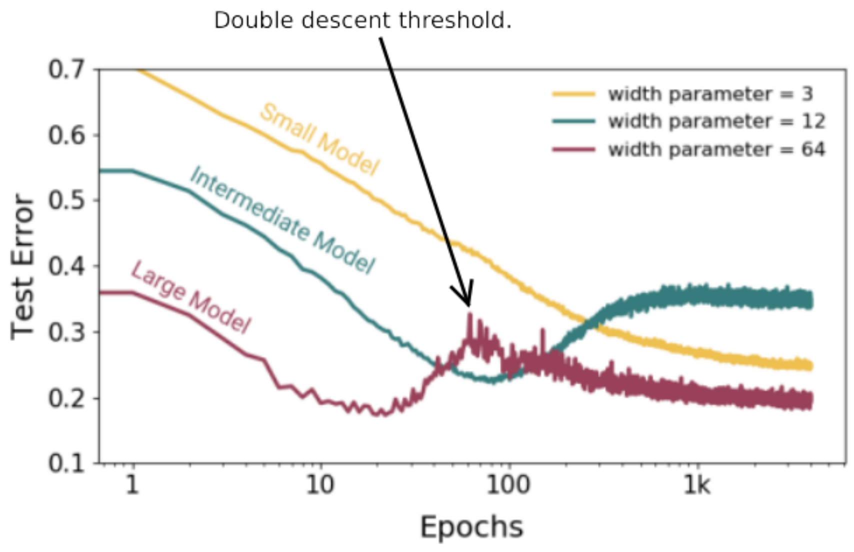


Figure 2.12: The figure shows the test error curve over training time for three different models. All models are ResNet models with width parameters 3, 12 and 64. All models are trained for four thousand epochs. We see that the large model exhibits the double descent phenomenon. The test error follows the original bias variance trade-off curve until epoch 100 where after we see another descent. Picture taken from [1].

For this experiment we would like the network we are going to train to overfit. When we say overfit we really mean that the training error should be close to zero. This is because in [1] they observed that the second descent in double descent occurs once training error is close to zero. For overfitting to occur we need a sufficiently large network that can fit the given training set.

To find a network that is able to do this we can increase the size of the network. This can be done until there are indications that a larger network would not perform better. A simple way of analyzing this is by looking at the distribution of the weight parameters and monitor how they evolve throughout training, taking snapshots around every 500 epochs. Now, if many weight parameters are zero or close to zero this means that many units are non-active. If this problem persists for a long training period this may mean that the network is too large and many units are inactivated. Thus, a larger network would just add redundant units.

Another way of analyzing a training procedure is by looking at how the first and second moments of the optimization algorithm evolve over time. We expect moments to be larger in early stages of training and then decrease to smaller values. This would suggest that the algorithm has done some searching and found a good minimum.

It is important to note that in the paper [1] they use 0-1 error for the test loss.

Definition 2.8.1 (0-1 loss function)**.** Let $\mathbb{X}\times\mathbb{Y} = \{(\boldsymbol{x}^{(1)}, y^{(1)}), (\boldsymbol{x}^{(2)}, y^{(2)}), ..., (\boldsymbol{x}^{(n)}, y^{(n)})\}$ be a data set where n is the number of samples and $\mathbb{U} = \{\hat{f}(\boldsymbol{x}^{(1)}), \hat{f}(\boldsymbol{x}^{(2)}), ..., \hat{f}(\boldsymbol{x}^{(n)})\}$ their corresponding predictors for some model M. Elements in $\mathbb{Y}$ and $\mathbb{U}$ can be of some m number of classes $c_1, c_2, ..., c_m$. We define the 0-1 loss function as

$$\mathcal{L}_{01}(\mathbb{Y}, \mathbb{U}) = \frac{1}{n}\sum_{i=1}^{n} \varphi(y^{(i)}, \hat{f}(\boldsymbol{x}^{(i)}))$$

$$\text{where } \varphi(y^{(i)}, \hat{f}(\boldsymbol{x}^{(i)})) = \begin{cases} 1, & \text{if } y^{(i)} = \hat{f}(\boldsymbol{x}^{(i)}) \\ 0, & \text{otherwise} \end{cases}.$$

For the case of regression problems there is no real equivalent to the 0-1 error and so we will have to use cross entropy (mean squared error in regression case) for the test loss.

2.9 Data

Simulating our own data set gives us more flexibility. We can choose how simple or complex we want our data to be. After this we can find a model that will be complex enough to fit the data. Choosing all of these factors makes it possible for us to tune the complexity of the data (and thus the model) to match the computational power that we have available. Additionally using simulated data means that we do not have to worry about running out of uncorrelated data sets for the estimation of EMC.

When talking about the complexity of data we mean how nonlinear it is. It is also important to remember that the more non-linear a function is the more data is needed to fully represent the function. For example, for a sinusoidal function we need more data points than say a polynomial of order two to represent the function in the interval $[0, 5]$.

2.9.1 Choosing a function

If we take a trigonometric function (or a combination of them) then there are essentially three factors that determine its degree of nonlinearity.

1. The range of the function, so the difference between the minimum and maximum value.
2. The amount of oscillations, how frequent the bumps are.
3. How different the heights of bumps are.

So we see that trigonometric functions are good candidates for our purpose since they give multiple aspects of non-linearity.

Additionally we would like to parameterize this function such that we can tune the non-linearity. The parametrization should be such that only one variable is needed to change the non-linearity. Take the function

$$f(c, k, \vec{x}) = k \cdot \left[\sin((c-k) \cdot \vec{x}) + \cos((c-k) \cdot \vec{x})\right].$$

Setting c to some number we can increase $k > c$ and get more frequent bumps. This also results in increasing the range or the amplitude of the function. Increasing c will as a whole increase the rate at which the range increased when increasing k.

Now we can use a model and then simulate a data set for it in a way that allows us to investigate some desired property of the training procedure, for example overfitting.

2.9.2 Creating a data set

Let $\mathbb{X}^n = \{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, ..., \boldsymbol{x}^{(n)}\}$ be the set of n number of data points where $dim(\boldsymbol{x}) = 5$ and each component $x_i \in \boldsymbol{x}$, $i \in \{1, .., 5\}$ is sampled from a uniform distribution $\mathcal{U}_{[-a,a]}$ where $a \in \mathbb{R} \backslash \{0\}$. Let $\mathbb{Y}^n = \{y^{(1)} = f(\boldsymbol{x}^{(1)}) + \epsilon, y^{(2)} = f(\boldsymbol{x}^{(2)}) + \epsilon, ..., y^{(n)} = f(\boldsymbol{x}^{(n)}) + \epsilon\}$ be the respective target variables given by the function

$$\begin{aligned} &f(\boldsymbol{x} = \{x_1, x_2, x_3, x_4, x_5\}) \\ &= k \cdot \{\sin[(c-k) \cdot x_1 x_3] + \cos[(c-k) \cdot x_1 x_2 x_4] + \sin[(c-k) \cdot x_2 x_3 x_5]\}. \end{aligned}$$

where $\epsilon \sim N(0, \sigma_{noise}^2)$ is a random variable representing the added Gaussian noise. The variance σ_{noise}^2 is calculated such that the signal-to-noise ratio

$$\frac{SD(f(\boldsymbol{x}))}{\sigma_{noise}} = r_{stn}$$

where $r_{stn} = 10$. This is done using a simulation where we sample roughly 25 million points $\boldsymbol{x}$ uniformly on the range $[-a, a]$ and calculate the standard deviation on the respective targets $f(\boldsymbol{x})$.

For the estimation of EMC a simplification of the data set will be made. Instead of using all five variables the data is simplified to three variables by setting the last two to the value of one, meaning that $\boldsymbol{x} = \{x_1, x_2, x_3, 1, 1\}$. This enables the networks to faster learn the function which is important since we are limited by computational power.

3 Results

For all tests using neural networks we used the PyTorch framework running on Python 3.10. The training procedure was similar to the one in this (The Training Loop) tutorial.

3.1 Double descent experiments

To test if the epoch-wise double descent occurs in a regression problem we used five different sized neural networks. The structures of each network are as in table 3.2. Each model is evaluated on a test set with samples such that the ratio of training data to test data is 70/30.

Three experiments were conducted. Each experiment was done on the same models but on different true underlying functions. The first experiment was picked out by trial and error such that one could observe an epoch-wise overfitting. This is since our goal was for the model to overfit epoch-wise. Then it was a matter of training the each network for a large number of epochs. The other two experiments were conducted to see if increasing the complexity of the true underlying function would affect the outcome. Furthermore, the third experiment was conducted on a smaller sample size than the first two. The motivation was that networks on such a function would overfit faster on a smaller sample size and potentially affect the outcome somehow.

The first experiment was conducted with parameters as in the second column in table 3.1. As we see in figure 3.1 there is a standard bias variance trade-off epoch-wise curve for all networks except the "small" one.

For the second experiment, with parameters as in column three in table 3.1, we see in figure 3.2 that no double descent is exhibited. Over epochs we do not see the typical bias variance trade-off curve. However, if we look at the larger models ("large" and "huge") we see that test error, for epochs 2000 and greater, is higher than for the "medium" model. This could indicate a model-wise bias variance trade-off, for larger epochs.

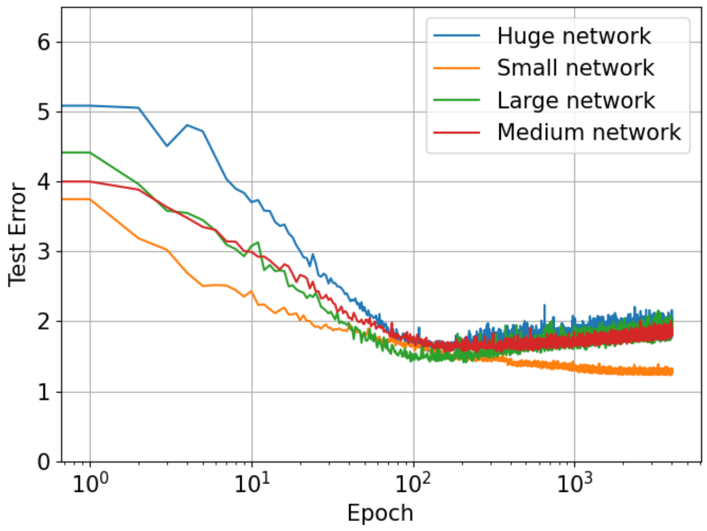
For the third and final experiment we increased the complexity of the function further and reduced the sample size to 5000. See column four in table 3.1 for details. The results in 3.3 suggest a model-wise regular bias variance trade-off and no double descent epoch-wise.

Experiment	1	2	3
Sample size	10k	10k	5k
Noise standard error	0.250	0.348	0.397
Minibatch size	50	50	50
LR ADAM	0.001	0.001	0.001
Betas ADAM	0.9 and 0.99	0.9 and 0.99	0.9 and 0.99
c	2	2.7	3.5
k	2.5	3.2	3.9
a	3	2.6	2.5

Table 3.1: The setup for each experiment as described in section 3.1. Each network was trained for 4000 epochs. The learning rate is denoted as "LR".

Network	"Small"	"Medium"	"Large"	"Huge"	"Extreme"
Number of skipping blocks	2	4	6	8	14
Number of parameters	317	1555	5773	12527	83337

Table 3.2: Table that shows the number of skipping blocks and total number of parameters in each type of neural network.



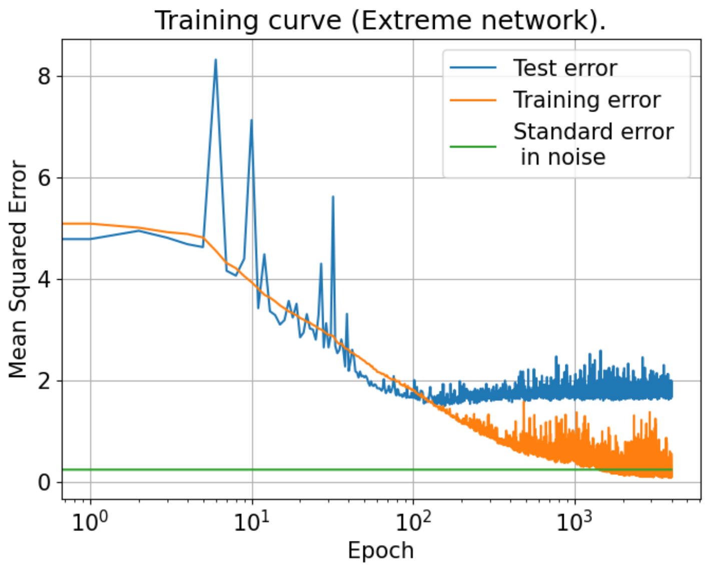
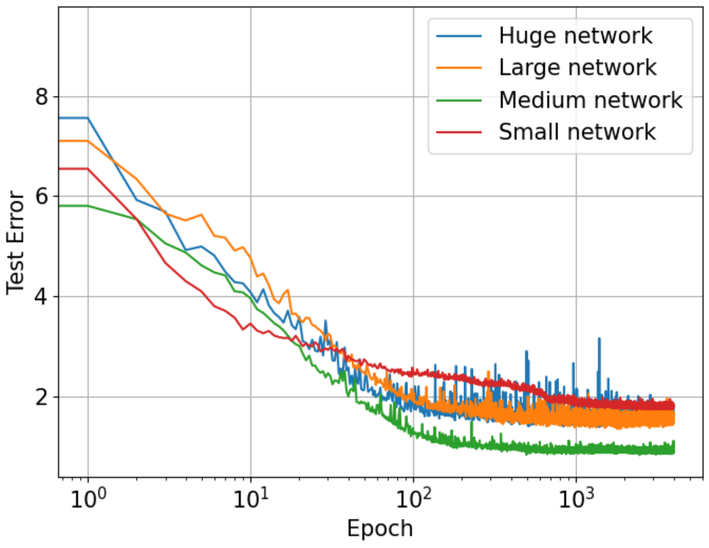


Figure 3.1: The upper plot shows four test error curves for four different sized networks as in table 3.2 The lower plot shows the training curve for an "extreme" network. Experiment one.



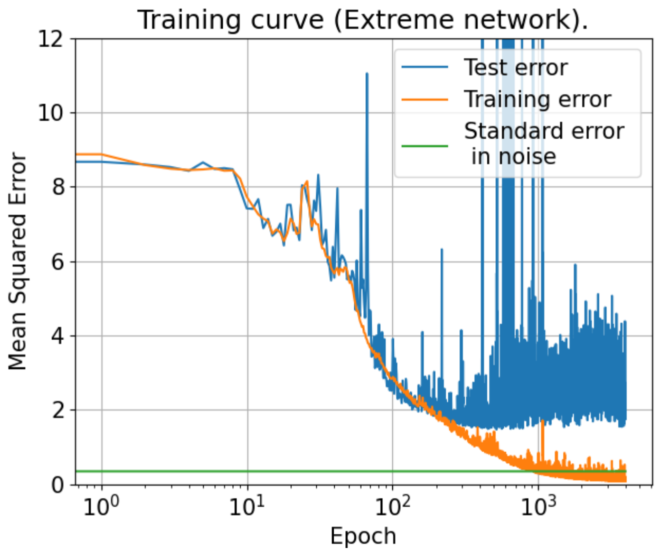
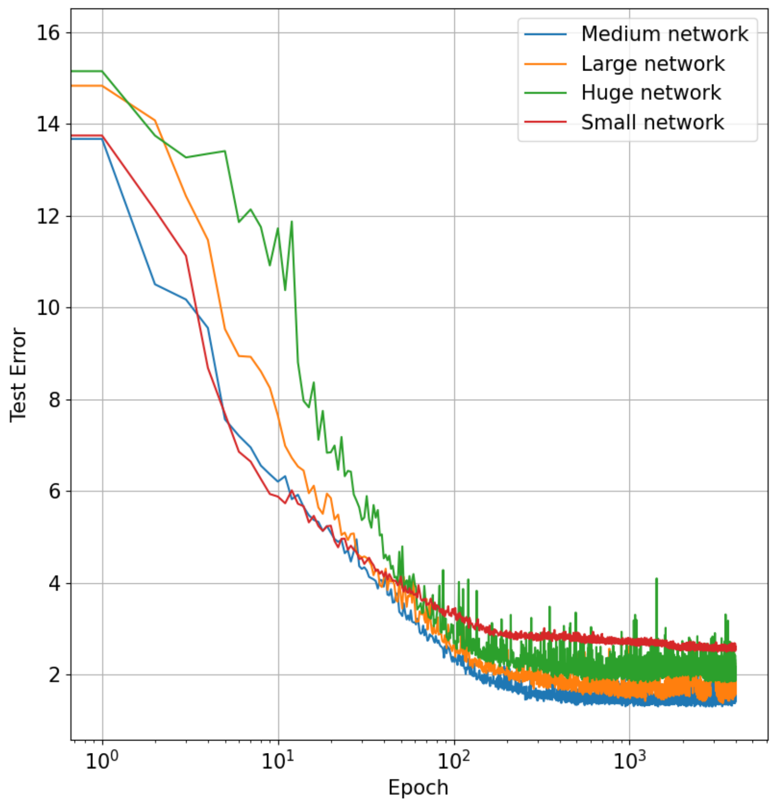


Figure 3.2: The figure shows four test error curves for four different sized networks as in table 3.2. The lower plot shows the training curve for an "extreme" network. The lower plot has been limited to a maximal MSE of 12 due to some extremely large values in the test error at certain epochs. Experiment two.



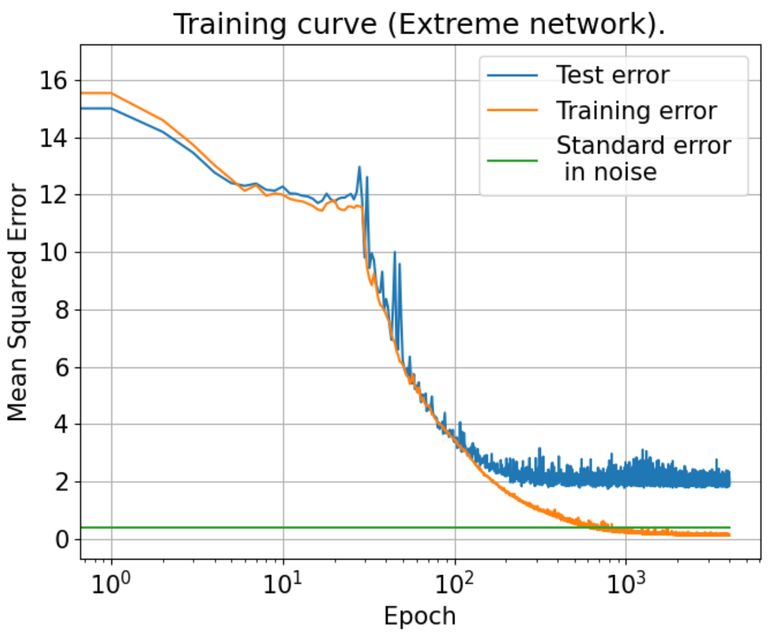


Figure 3.3: The upper plot shows four test error curves for four different sized networks as in table 3.2. The lower plot shows the training curve for an "extreme" network. Experiment three.

Looking at all three results we see that the more complex a function is the larger the optimal (one that minimizes test error) network has to be. Now, this is something that is expected asymptotically, meaning as we increase the number of epochs to significantly large numbers. Since for smaller number of epochs smaller models will not have had time to fit to their maximal potential. For example looking at the upper plot in figure 3.1 we see that for epoch 100 that the "large" model performs best (lowest error), where as for epoch 1000 and higher the "small" model performs best. Additionally we see that all "extreme" models overfit since the training error is roughly zero for all models around 4000 epochs. However the test errors all indicate that once the models have overfitted test error does not change in any significant way. This would imply that once a model has overfitted it has found some minimum and training for longer will just mean no large updates in the optimization algorithm. This is also seen in the moments in figure 3.4. We see that the moments for larger epochs tends to zero.

As mentioned in section 2.8 in the original study of double descent [1] they used cross entropy to train the model yet 0-1 error to evaluate it. As we saw in definition 2.8.1, the 0-1 error is the ratio of correct classifications on a test set. Whereas cross entropy gives more information than the fact that the prediction was the right class or not. Cross entropy gives a probability distribution over all the classes. Meaning, if we are predicting two classes "cat" and "dog" cross entropy could give us more information about the classification. Something like $Pr(dog) = 0.8$ and $Pr(cat) = 0.2$ while the 0-1 error would just say if the classification is correct or not.

Now all three experiments above suggest that double descent does not occur for a regression problem when evaluating the model using cross entropy as loss. However, further developing the discussion above, this could either mean that double descent does not occur for regression problems or that double descent does not occur when evaluating the model using cross entropy.

Theoretically double descent does not make sense for regression when using cross entropy for evaluation. It would be quite strange that a models training error would constantly be decreasing and that at some point of overfitting it would stop overfitting and start to generalize better. The only scenario that seems plausible is something like we see in figure 3.5. However, the chances of such a scenario are highly unlikely.

3.2 EMC estimation

For the estimation of EMC we used a setup as in table 3.3. For each sample size we trained five networks, each on ten different data sets S. Thus, averaging over 50 different training procedures for the estimation.

The results in figure 3.6 suggest that EMC increases over epochs. Furthermore, again

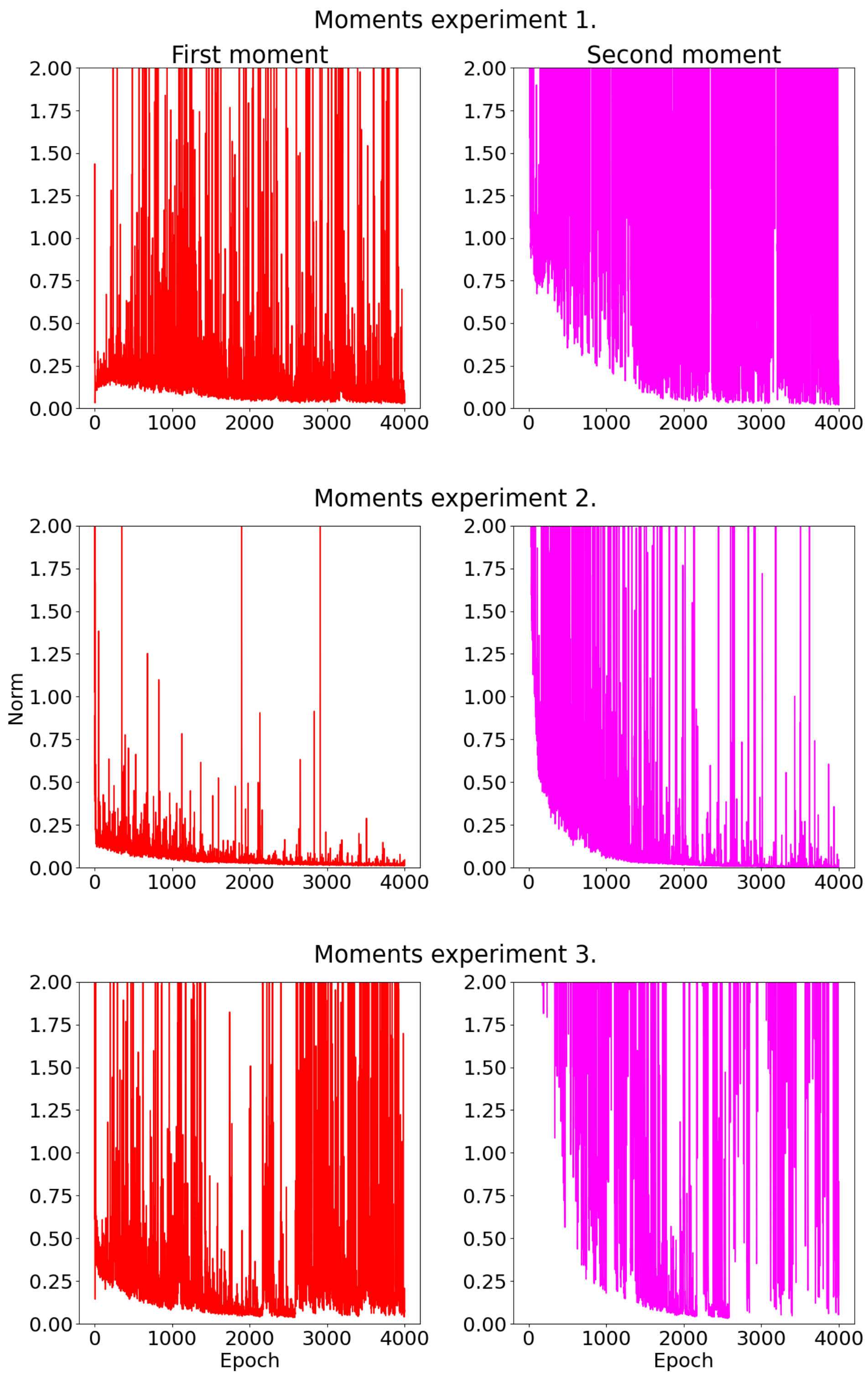


Figure 3.4: The figure shows the norm of the first and second moments over epochs for the training of "extreme" networks in experiments one to three. The "norm" is the mean Frobenius norm of tensors associated with the first and second moments.

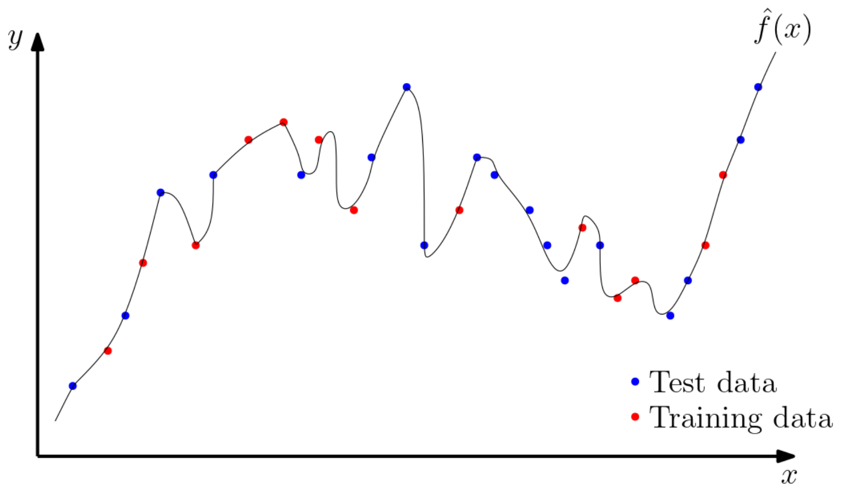


Figure 3.5: This is scenario where the training error is low since the model is overfitted. But the test error is also low since the overfited model happens to also fit the test data quite well. The model has to make this sort of fit without knowing the test data.

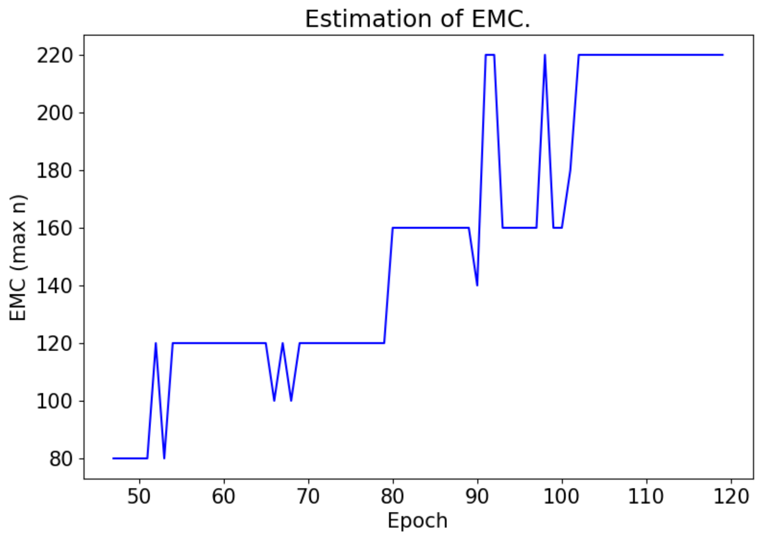


Figure 3.6: The figure shows the plot for EMC n (y-axis) over epochs (x-axis). The number n in EMC is as in definition 2.7.1.

Network	**LR**	**Betas ADAM**	**Epochs**	**a**	**c**	**k**	**Minibatch size**
"Small"	0.1	0.9 and 0.999	120	1.5	2	2.8	20

Table 3.3: Setup for training procedures for the estimation of EMC. The learning rate is denoted as "LR".

looking at figure 3.6 one can see that the bottom axis starts at around 50 epochs. This is due to the fact that it takes some time for a neural network to reach zero training error.

EMC is a solid definition, theoretically, while practically, some problems occur. The obvious one is that it is hard to estimate, especially for more complex problems that require larger models and larger data sets. Furthermore, as we said, neural networks can not reach close to zero training error for early epochs (For more difficult problems this number could be quite large). Thus we can not estimate EMC for these regions making the definition limited.

Another downside is something that happens when the model is sufficiently large. When estimating EMC the training is done on different sample sizes and then it is decided when training error is no longer close to zero. This is where a problem occurs. Since for smaller sample sizes a larger learning rate is needed to find a proper minimum than when training on a larger sample size. See figures 3.7 and 3.8 for more clarity. A solution to this would be decreasing the learning rate as we increase sample size. But the problem is that changing the learning rate changes the training procedure. This can not be done since the estimation of the complexity should be performed with a fixed training procedure as described in section 2.5.

3.3 Evaluation

The results for the experiments on double descent are mostly limited in the sense that we are restricted by computational power. In the original double descent paper the phenomenon was exhibited by a neural network with close to thirty million parameters. Comparing this to the number of samples used (60k) the difference is huge compared to our experiments. It may be so that the networks used in our experiments are just not large enough for us to see a double descent.

As we saw in definition 2.7.1 for EMC, there exists a parameter ϵ that needs to be decided. As we saw in section 2.7.2, using some intuitive argument we can find a candidate σ_{noise} for ϵ. However, it is not appropriate to set $\epsilon = \sigma_{noise}$ and instead we used a statistical test (using σ_{noise}) to evaluate when expected training error for a model is roughly equal to zero. This way of deciding when a model has zero training error is more rigorous than arbitrarily deciding on some value for ϵ.

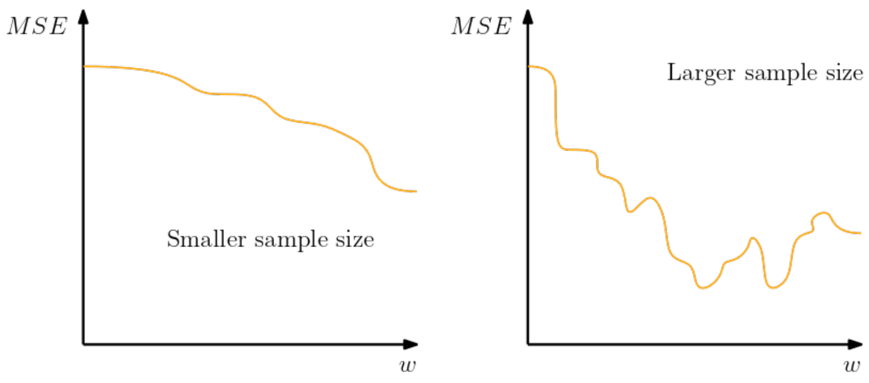


Figure 3.7: This is a simplified figure where the parameter w somehow represents the parameters of a neural network as a whole. If we look at the left figure, for a smaller sample size a small change in some parameters does not affect the training loss that much since we have so many weights. As we see in figure 3.8 we could probably alter multiple units and this would barely affect the training loss. So for smaller sample sizes we will have a flatter surface. Now, looking at the right figure, for a larger sample size, changing a couple of weights will affect the error loss more, since the loss is relying on more samples modeled by less wights in comparison to the case with a smaller sample size.

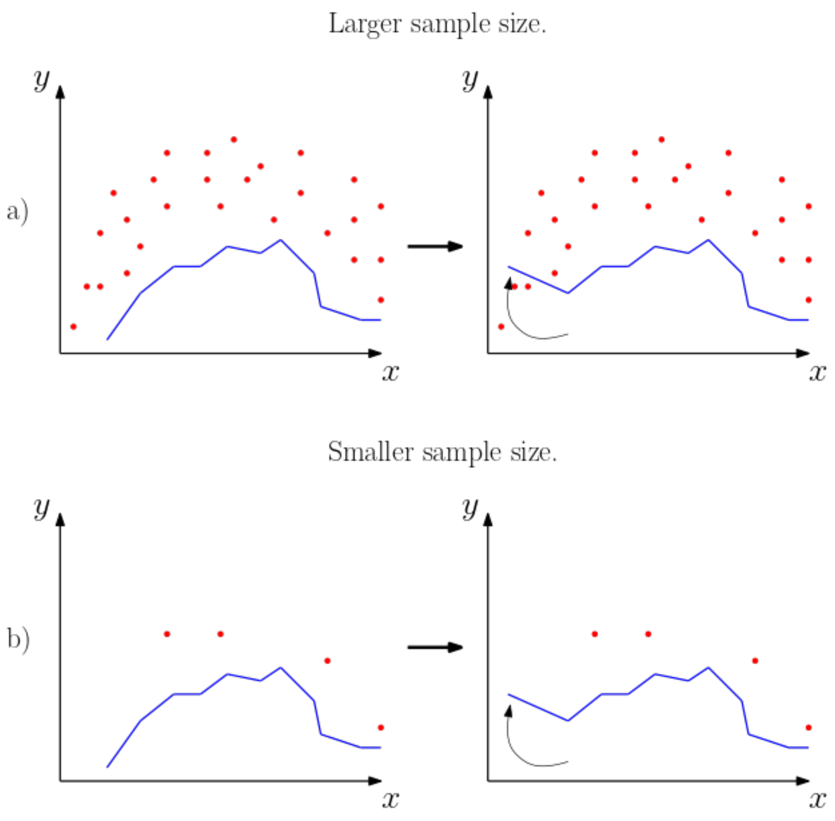


Figure 3.8: The red dots in the figure represent data points in some theoretical training set. In the figure we see that for a larger sample size a), changing some parameters of our model (blue line) alters the loss (MSE) more than changing the same parameters but on a smaller sample size b). Thus, for a smaller sample size there are more parameters that can be alerted such that the training loss is barley affected.

4 Conclusion

The goal of this thesis was to investigate if the double descent phenomenon presented by Nakkiran and others in [1] for classification problems occurs for a regression problem. Additionally, we desired to investigate how training time for feedforward neural networks alters the model complexity by defining and estimating a complexity measure called Effective Model Complexity (EMC). The main methods used for this were the following.

1. Training a neural network with skip connections and batch normalization using the ADAM algorithm.
2. Estimating EMC for a neural network using averaging over an ensemble of training procedures. Repeating this process for different sample sizes. Then evaluating that each ensemble has reached a sufficiently low training error using a statistical test.

The results from estimating EMC suggests that it does increase over training time. Thus giving us some evidence that model complexity for neural networks increases over training time.

For double descent the results showed that the phenomenon is not exhibited by feedforward neural networks for a regression problem.

Finally, it remains elusive why double descent happens for classification problems and further studies need to be performed. This thesis also reminds us about the concerns raised in the introduction. The difficulty of trusting a machine learning model that one might not fully understand the works of. The future thus holds many studies about understanding machine learning and artificial intelligence.

4.1 Future directions

As we saw with EMC it is theoretically a good definition but is hard to estimate. The techniques used in this study may be too simple and thus further studies need to be performed to find alternative approaches to estimate EMC. The key aspect is determining when a training procedure has reached zero training error.

EMC is not the only way to quantify and measure complexity for neural networks. It would thus be an intriguing aspect if and how different measures align with our observations that training time tends to increase model complexity.

Lastly, the field of deep learning for regression is quite new and unexplored. It differs from the case of classification problems and thus everything we know about classification problems may not be applicable to regression problems. Meaning, just like with the double descent phenomenon, research on other phenomena exhibited by neural networks on classification problems need to be performed. For example

- The stability of a model against noise.
- Transfer learning.
- Attacks on machine learning algorithms and on defenses against such attacks.

And many more!

Bibliography

[1] Preetum Nakkiran et al. *Deep Double Descent: Where Bigger Models and More Data Hurt.* 2019. arXiv: `1912.02292 [cs.LG]`.

[2] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* June 2016.

[3] Terry Bossomaier et al. "Transfer Entropy". In: *An Introduction to Transfer Entropy: Information Flow in Complex Systems.* Cham: Springer International Publishing, 2016, pp. 65–95. ISBN: 978-3-319-43222-9. DOI: `10.1007/978-3-319-43222-9_4`. URL: `https://doi.org/10.1007/978-3-319-43222-9_4`.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* `http://www.deeplearningbook.org`. MIT Press, 2016.

[5] Guido Montúfar et al. *On the Number of Linear Regions of Deep Neural Networks.* 2014. arXiv: `1402.1869 [stat.ML]`.

[6] Simon J.D. Prince. *Understanding Deep Learning.* The MIT Press, 2023. URL: `http://udlbook.com`.

[7] Aston Zhang et al. *Dive into Deep Learning.* `https://D2L.ai`. Cambridge University Press, 2023.

[8] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2017. arXiv: `1412.6980 [cs.LG]`.

[9] Anton Zaitsev. *DA6007.* Version 1. May 2024. URL: `https://github.com/anza4662/DA6007`.