Chapter 3

Problem formulation

We chose machine learning, in particular the subfield deep learning, as our main approach to solve the problem of time-independent price prediction in terms of the game FIFA 20 and it's game mode FIFA Ultimate Team. Today machine learning is a widely researched field, that constantly gains in relevance and interest.
This leads to the fact, that it is split into many different research areas with varying approaches and targets. We applied four different models to the same problem and compared the results to see which approach provides the best solution. In the subsequent sections we will discover the theoretical base of those different approaches, where we focus the most on the last section about our main concept, the deep learning model.

The subsequent definitions hold for all of the whole chapter:

- x is an input vector of n input feature values representing one data sample.
- X denotes the matrix of m training sample features that are used to train our models. This means $X = \begin{bmatrix} x_1 & ... & x_m \end{bmatrix}$, where m is the number of training data samples.
- y describes the target value of one sample. Hence we are facing a regression problem want to obtain only a one dimensional output value.
- Y designates the vector of m training sample target values, that are used to train our models, this means $Y = \begin{bmatrix} y_1 & ... & y_m \end{bmatrix}$

3.1 Regression

The task of regression can be described by the target of finding a function $r : x \rightarrow y$ where x and y are the at the beginning defined notations.
The main challange is to minimize the resulting error between prediction and actual

target value. This error is called loss and is denoted by $L : y \times y' \to \mathbb{R}_+$
One of the most popular loss functions is the squared loss function, defined by: $L_2(y, y') = \frac{1}{2}|y - y'|^2$.
As L only returns the error for one specific prediction, we want to transfer our obtained results and merge it into a more general formula with more significance, taking into account all training samples.
Therefore we use the following equation, called the mean squared error:

$$R(r) = \frac{1}{m} \sum_{i=1}^{m} L(r(x_i), y_i) \tag{3.1}$$

m denotes the number of single training samples and r and L are the former defined functions.
We have now a regression function r that returns a label value y for an arbitrary input-vector X, a loss function L that measures the magnitude of error between the predicted label and the actual label value and the function $R(r)$ that hands back the mean squared error of our regression function and the training data. [8, p. 237-238]

3.2 Linear Regression

At first we have a look at a well known basic approach when it comes to regression tasks, the linear regression.
The main idea is to find a function $f(x)$ that approximates the related y value to a certain x vector as close as possible for all $x \in X$. We define this function by $f(x) = w \cdot \theta(x) + b$, where w is a transposed vector of adjustable parameters to make the model fit the label values y the best and $\theta : X \to \mathbb{R}^m$, which transforms the input feature matrix into a m-dimensional vector to be combinable with linear shape of the model.
$f(x)$ returns $\hat{y}$ as a prediction value for the related y value of x. As discussed in the former section we can use L_2, the squared loss, to evaluate the error of the prediction of $f(x)$. The idea is to minimize the squared error for all $x \in X$, taking into account all training samples simultaneously, like we did in the former section where we defined the mean squared loss. The squared loss can be applied to our optimization by transforming it into the following formula:

$$\min_{\mathrm{W}} F(W) = \frac{1}{m} \|X^T W - Y\|^2 \tag{3.2}$$

where X are our features denoted as $\begin{bmatrix} \theta(x_1) & \dots & \theta(x_m) \\ 1 & \dots & 1 \end{bmatrix}$ with an additional 1 because of dimensional reasons when multiplying it with the weights vector. W are

the weights written as vector in the following form $\begin{bmatrix} w_1 \\ \dots \\ w_N \\ b \end{bmatrix}$. Because F is a convex, differentiable function it has a global minimum if $\nabla F(W) = 0$. Thus we can reformulate equation 3.2 as following:

$$\frac{2}{m} X(X^T W - Y) = 0 \Leftrightarrow XX^T W = XY \tag{3.3}$$

The obvious unique solution of the right side of equation 3.3 if XX^T is invertible is: $(XX^T)^{-1}XY$. Otherwise there is a family of solutions, defined by the pseudo-inverse of matrix XX^T.
Linear regression can be easily visualized for the simple case of n=1 which corresponds to the case of one input feature for each x.
In figure 3.1 we can see this visualization, where the blue points our data points (x and y) and the red line is the linear approximation that minimizes the mean squared error.

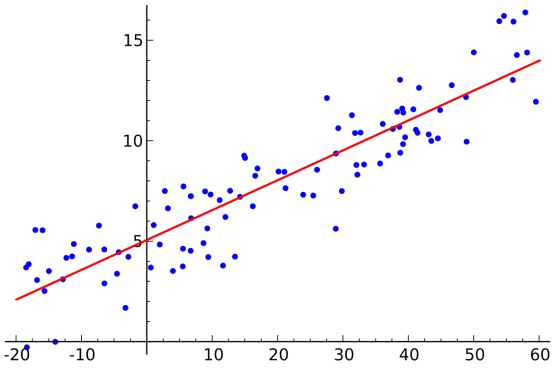

Figure 3.1: linear regression problem []

To close this section about linear regression we want have a short look at the expense of this approach.
As explained it takes a computation of the inverse of a matrix which is a computation task of a complexity of approximately $\Omega(n^{2.4})$ to $\Omega(n^3)$. This is equivalent to a time extension of the computation by a factor of 5.3 to 8. []

3.3 Support Vector Machine

Support vector machines are often utilized to solve classification problems where a data set of training samples with labels is used to determine a hyperplane for the given classification problem like in figure 3.2, that divides the training instances

into two groups by keeping a tube around the hyperplane as large as possible, where the amount of training instances is as low as possible. In figure 3.2 This tube is represented by the dashed lines parallel to the straight line which shows the hyperplane. The gained hyperplane is then applied to the test samples to classify them.

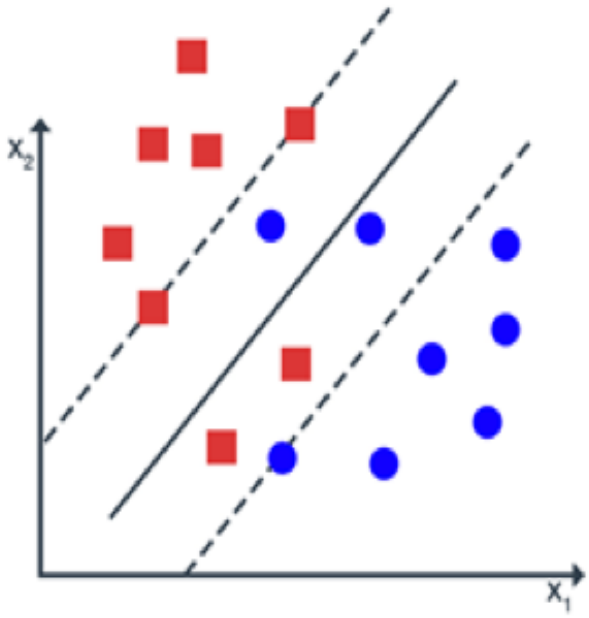

Figure 3.2: SVM hyperplane determination [**?**]

Even though the SVM is more popular for its classification capacity, it is also capable of solving regression problems. To solve a regression problem we have to reverse this method. This means we want to fit a tube of width $\epsilon > 0$ to the data, that includes as many training instances as possible as shown in figure 3.3.

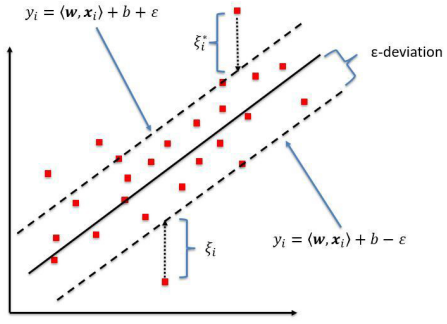

Figure 3.3: Example of the ϵ-tube around f(x)

In other words, we want to find a function that does not deviate more than ϵ for every target value of the training data.
This leads us to the following formulated optimization task:

$$\min_{w,b} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m} |y_i - (w \cdot \theta(x_i) + b|_\epsilon \tag{3.4}$$

where $|\cdot|_\epsilon$ denotes the loss-function:

$$|\hat{y} - y|_\epsilon = \max(0, |\hat{y} - y| - \epsilon) \text{ for } \hat{y}, y \in \mathbb{R} \tag{3.5}$$

Minimizing w and at the same time ensuring that the error is smaller than ϵ, leads to very few solutions up to no solution at all. To cope with this challenge and also allow some errors in given cases, we introduce another concept, called *slack variables* which allows us to reformulate 3.4 to the subsequent, feasible optimization problem:

$$\min_{w,b,\xi,\xi'} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}(\xi_i + \xi_i^{'}) \tag{3.6}$$

$$\text{subject to } (w \cdot \theta(x_i) + b) - y_i \leq \epsilon + \xi_i \tag{3.7}$$

$$y_i - (w \cdot \theta(x_i) + b) \leq \epsilon + \xi_i^{'} \tag{3.8}$$

$$\xi_i \geq 0, \xi_i^{'} \geq 0, \forall i \in [1, m] \tag{3.9}$$

[9]

3.4 Random Forest

The *random forest* algorithm is a another regression method, based on the concept of *decision trees*.
A decision tree is an ordered tree with a root r. Every non-leaf vertex, meaning it does not have any further child-vertexes, represents a question, that can be answered in different ways, that are represented by the originating edges of this vertex. For the edges $(v, x_1), ..., (v, x_n)$ from a nonleaf vertex v, we call $x_1, ..., x_n$ the *decisions* at v. [10, p. 39]
This means every vertex of a decision tree represents a question about the sample and every edge from this vertex is a possibly answer to this question. The next vertex is another question and so on. At the bottom of the tree every leaf represents an output value for the superordinated question the decision tree is meant to answer.
A decision tree can be created from a data set with multiple feature values and one target value. It is built up from the root to the leafs. This means one starts with the determination of which feature should be the first vertex. This question can

be answered by checking how good every single feature performs on independently predicting the right value for the regression problem for all samples of the data set on its own. This idea is continued for every stage/vertex of the tree omitting the already used features.
The transition from one decision tree to random forest happens by applying at first the concept of ensemble learning. This means creating multiple decision trees and aggregating their results. The aggregating for regression is implemented by taking the average of all the predicted values. Secondly it uses bagging (bootstrap aggregation), where only a random subset of all samples of the data set are used to construct every decision tree. The idea behind the concept of bagging is to improve stability and accuracy of the algorithm. This method so far is already a possible approach of solving a regression task on its own.
Random forest is modifying this concept by making the construction of the trees even more random. This is obtained by taking only a subset of the features available for determining a new vertex of a decision tree at every stage. This additional specification makes the algorithm very robust against overfitting. To sum up the random forest algorithm creates multiple decision trees, by firstly using only a subset of the given data set and secondly using only a random subset of all features at every stage for the tree creation. In the end the average of all the resulting values, that are obtained from the trees, is taken. [11]

3.5 Deep Learning

As already mentioned in the beginning of this chapter machine learning is a strongly growing researched field of science with many different subfields.
The question "What is Deep Learning?" brings us to the recognition, that there are many different answers to that question. [12]
What the most of definitions have together is, that deep learning is about learning relations between multiple features/variables by using various layers of neurons to extract and transform those features.
The adjective *various* indicates the intention of not limiting the depth of the neural networks to a specific number, but it is often agreed to talk about a deep neural network if the network consists of three layers or more. There is one input and output layer and one or more hidden layers in between. [13, p. 263]
The neurons of successive layers are related by an arbitrary system of connections as we can see in figure 3.4. The further-reaching question about deep neural networks we want to cope with is about what makes them unique and more advantageous than other machine learning approaches.

3.5.1 General architecture

The idea of artificial neural networks (ANN) comes from biological models like the human neural system. Already the human cortex is structured in multiple layers of biological neurons.
Taking the nature as an example, further research in the field of machine learning resulted in building deep neural networks (DNN) that can cope with very complex related data of enormous size. [13, p. 257]
In general a deep neural network consists of an input layer that takes in all the feature values of the data and has usually as many neurons as the number of features. The input layer is succeeded by an arbitrary amount of hidden layers, where the amount of neurons can't be determined by certain rules, but is part of the research and development process of any individual problem. In the end we apply an output layer, that has exactly one neuron for regression problems.
An example for a specific architecture of a neural network is the fully connected neural network (FCNN), which is defined by the structure of a connection from every neuron to all neurons of the next layer as we can see in figure 3.4.

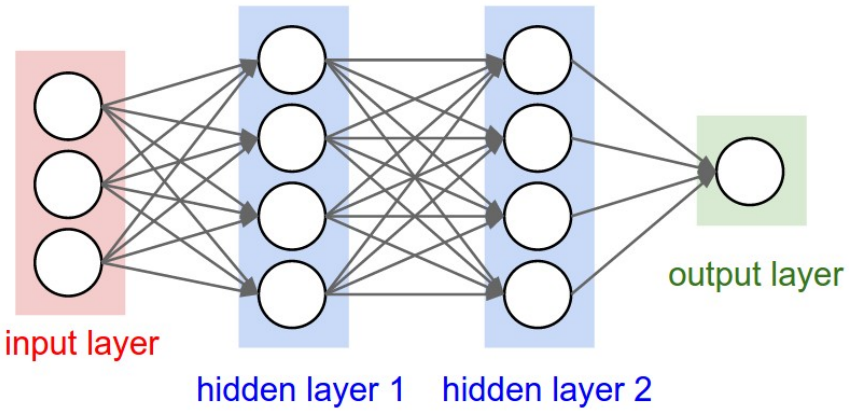

Figure 3.4: Fully connected neural network with 2 hidden layers

3.5.2 Gradient Descent

"Gradient Descent is a very generic optimization algorithm capable of finding optimal solutions to a wide range of problems." [13, p. 113]
The Gradient Descent algorithm is an often used method for optimizing the weights in neural networks. The basic idea is to measure the gradient of the error function, to find out which "direction" minimizes the error the most. The next move is to make a small step, defined by the learning rate, into this descending direction. If the gradient turns out to be zero, the error has reached a local minimum of the error function.
The learning rate is a very important hyperparameter in machine learning and can

be to high and also to low. In case of a learning rate that is to high it is possible to miss out minimas because the algorithm stepped over a minima. Otherwise a to small learning rate can result in a very slow and inefficient training progress.
(MAYBE INCLUDE FIGURE OT BOTH CASES?)

3.5.3 Backpropagation

Now we want to have a look at how the weights in a DNN are being optimized using an algorithm like the gradient descent. Therefor we are having a closer look at the concept of *Backpropagation*.
The breakthrough in developing a method of properly training DNNs or neural networks in general was achieved inter alia with an article published in 1986. The article with the title *Learning Internal Representations by Error Propagation"* introduced the concept of backpropagation. [13, p. 2639]
2 The main idea behind this concept is that for every training instance that is fed into the neural network, the output of the last layer is compared to the known target value. It is traced back how much of the resulted error is caused by each neuron of the last hidden layer.
This procedure is continued by exploring the source of the error of each neuron in the last hidden layer from the preceded layer and so forth until we end up at the input layer.
With this sequence we measure efficiently the error gradient throughout all the weights of the neural network. By now there hasn't been any progress or optimization for the neural network. To effectively learn from the measured errors the backpropagation algorithm does a Gradient Descent step using the obtained information about every single neuron and its weight.

3.5.4 Batch Normalization

To face a very popular problem when it comes to neural networks and especially training them, the *vanishing gradients* problem, we introduce the concept of *Batch Normalization*.
The vanishing gradients problem can be roughly described by the effect, that gradients are often getting smaller when it comes to the lower layers. This leads to the incident of very low change to no change in the weights of the lower layers, which results in a lack of convergence of the training.
As brought up already, we have a solution to this problem.
Batch Normalization was proposed by Serfey Ioffe and Christian Szegedy in one of their papers in 2015 called "Batch Normalization: Accerlerating Deep Network Training by Reducing Internal Covariate Shift" [] and addresses amongst others the mentioned problem.
Roughly speaking Batch Normalization is an additional operation, that is added to a layer, which learns the layer the optimal scale and mean of its input.