HOCHSCHULE
MITTWEIDA
University of Applied Sciences

MASTER THESIS

Mr.
Saransh Rastogi, M.Sc.

Solving Common Classification and Regression Problems using a Single Convolutional Neural Network

Mittweida, May 2024

Faculty of **Engineering Sciences**

MASTER THESIS

Solving Common Classification and Regression Problems using a Single Convolutional Neural Network

Author:
Saransh Rastogi

Course of Study:
Elektrotechnik - Automation

Seminar Group:
EA20wV-M

First Examiner:
Prof. Dr.-Ing. Daniel Kriesten

Second Examiner:
Prof. Dr.-Ing Jan Thomanek

Submission:
Mittweida, 01.05.2024

Defense/Evaluation:
Mittweida, 2024

Bibliographic Description

Rastogi, Saransh:
Solving Common Classification and Regression Problems using a Single Convolutional Neural Network. – 2024. – 65 S.
Mittweida, Hochschule Mittweida – University of Applied Sciences, Faculty of Engineering Sciences, Master Thesis, 2024.

Referat

This master's thesis aims to explore the potential of utilizing convolutional neural networks (CNNs) within a learning framework to address both classification and regression problems simultaneously. The primary objective is to investigate the feasibility of developing a unified neural network architecture capable of handling diverse problem types. This research will not only assess the model's predictive performance in terms of accuracy and efficiency but also delve into the underlying mechanisms contributing to its effectiveness.

Contents

List of Figures

List of Tables

1 Analysis of Task Statement

This master's thesis aims to explore the potential of utilizing deep learning methodologies, including convolutional neural networks (CNNs) as well as feed forward networks (FNNs) within a learning framework to address both classification and regression problems simultaneously. The primary objective is to investigate the feasibility of developing a unified neural network architecture capable of handling diverse problem types. This research will not only assess the model's predictive performance in terms of accuracy and efficiency but also delve into the underlying mechanisms contributing to its effectiveness.

Research Objectives

- CNN Architecture Design: Develop a novel architecture that integrates components for both classification and regression tasks within a single convolutional neural network. Explore various strategies to effectively share and combine information between the two tasks.
- Model Training and Optimization: Implement training procedures for the multi-task CNN, incorporating appropriate loss functions and optimization strategies for simultaneous classification and regression.
- Performance Evaluation Metrics: Define comprehensive evaluation metrics that capture accuracy, precision, comparability of networks, as well as metrics like mean squared error (MSE) and mean absolute error (MAE) for regression tasks. Compare the proposed multi-task approach against single-task models.
- Efficiency Analysis: Conduct thorough experimentation to analyze the efficiency gains achieved through the multi-task architecture. Compare the computational resources required for the multi-task network versus separate networks for classification and regression tasks.

Expected Contributions

This research is expected to yield insights into the viability of employing a single convolutional neural network to address diverse classification and regression challenges. The findings will provide a deeper understanding of the interplay between tasks within a multi-task learning framework and offer guidance on designing efficient and accurate multi-task neural architectures.

Keywords

Convolutional Neural Networks, Multi-Task Learning, Classification, Regression, Deep Learning, Efficiency, Performance Evaluation

2 Motivation

The motivation for this thesis is built upon the work done by Mr. Christian Schreiber (Dipl.-Ing. (FH)) during the course of his thesis [1]. During his research, Mr. Schreiber focused on the application of two Convolutional Neural Networks (CNNs) to solve two distinctly different problems. The goal of the thesis was to implement the neural networks on a NVIDIA Jetson Nano Jetracer.

The first CNN was used to implement a 'Follow-Me Function'. A camera, attached to the Jetracer, was used to capture images. The images were used as the input data for lane detection and tracking. This was simplified to a regression problem, with the goal being to estimate final coordinates, indicating the direction where the Jetracer would drive. The coordinates would inturn be converted to a specific steering angle.

The second task was to implement an image detection algorithm, in order to make the Jetracer detect and react to a 'Stop sign'. Once again, the attached camera on the Jetracer was used as the data basis. The problem was formulated as a typical classification problem, with three different classes being defined (Stop sign visible, stop sign close, stop sign far away). A second CNN was trained for this task.

In his research, Mr. Schreiber focused on the implementation of the CNNs on the Jetracers. The application of autonomous driving robots with lane tracking and object detection on the Jetson Nano Microchips was the focus of the thesis. To solve both the problems, individual CNNs were implemented. To this end, pre-trained neural networks, such as AlexNet, were employed. For both of the networks, the last layer was modified to tailor the networks for the underlying problem.

For the final implementation, the trained networks were then uploaded onto the Jetson Nano. These networks then use the 'live' data collected from the cameras to allow the Jetracers to drive autonomously.

Due to computational constraints of the Jetracers, running two trained CNNs of considerable size proves, although successful, also taxing for the micro-controller. A certain amount of computational time is required for, first, the regression problem to be solved and then the classification problem. This causes a certain latency in the calculations. A disadvantage of this latency is that the Jetracer does not perform well at higher velocities, since the decisions (of the driving directions) are not made fast enough.

The principal motivation of this thesis has been derived from the results of aforementioned research. This thesis attempts to answer the question, whether there are frameworks present which allow such different problems to be solved with a single neural network. This would require a certain modification of the architecture of the neural network, since two different outputs would be needed. A large advantage of such an approach would be the utilization of computational resources in a more efficient manner, since only one network would need to be operated.

On the other hand, whether such an approach would be more efficient than a sequential execution of two separate networks, with regard to metrics commonly used in the deep learning field, remains an open question. The fundamental task of this thesis is to conduct experiments to obtain an insight to exactly this question.

To this end, the main research conducted during the course of this thesis and the developed methodology is tested on existing datasets. The MNIST dataset, a standard in the deep learning community, was chosen due to its popularity in the field and the vast amount of comparative studies already available. On the other hand, a lesser known 'Abalone' Dataset was selected not only to show a further implementation of the methodology, but also due to its unique problem statement allowing a demonstration of regression and classification with the same input (see Chapter 5 for further details).

Due to time constrains and the extensive nature of the experiments conducted, the tests could not be extended to real world data. The reader is referred to the Chapter 7 for potential ways to further build upon the work of this thesis.

3 Fundamentals

This chapter provides a brief introduction and overview of the principles employed during this thesis. After summarizing the different kind of problems that can be tackled with neural networks, a short glimpse of the building blocks of neural networks is also presented.

3.1 Types of problems

The basic problem to be solved determines which kind of algorithm, deep learning network, etc. will be implemented. Machine learning algorithms, of which deep learning is a subset, are based on 'learning' from past experiences and predicting results based on, as of yet unknown, inputs.
A concise definition of **'learning'** is provided by Mitchell (1997): 'A computer program is said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P*, if its performance at tasks in *T*, as measured by *P*, improves with experience *E*.' [2]
Whilst all of these entities are important to this thesis, and will be touched upon during the course of the thesis, in this chapter the focus will be laid on tasks, *T*.
With regard to machine learning, tasks are usually described with regard to how **quantified** and **labelled** input data should be processed. Typically, the input data will be defined as a vector

$$x \in \mathbb{R}^n$$

where each entry

$$x_i$$

describes some *feature* of the data. For example, the pixels of an image would its features, and an image would be provided as an input vector. [3]
This thesis focuses on so called classification and regression problems. These will be discussed briefly in the following chapter.

3.1.1 Regression

Regression tasks deal with predicting numerical values for some input data. The learning algorithm produces a function

$$f : \mathbb{R}^n \to \mathbb{R}$$

Whilst the type of task is similar to classification (see below), the format of the output is quite different. An example would be to predict the price of houses in the future, based on data and patterns available over a certain time period.

3.1.2 Classification

Classification tasks deal with categorizing input data into predefined subsets, or classes. The learning algorithm produces a function

$$f : \mathbb{R}^n \to \{1, ..., k\}$$

with the input

$$x$$

being mapped into a class

$$k$$

by a mathematical function. This is quite often done via a probability density function. A simple example of this would be the classification of an image of an animal, the input, as either a cat or a dog, the classes where the input shall be classified into.

Figure 3.1 portrays a comparison of classification and regression: Basically, classification attempts to divide the data into subsets (in this case two subsets), whereas regression attempts to obtain a value (in this case, the mean value of the data set). Potential use-case of classification would be to sort data, whereas regression could be used to extrapolate data to obtain answers to hypothetical questions which are not covered by the data.

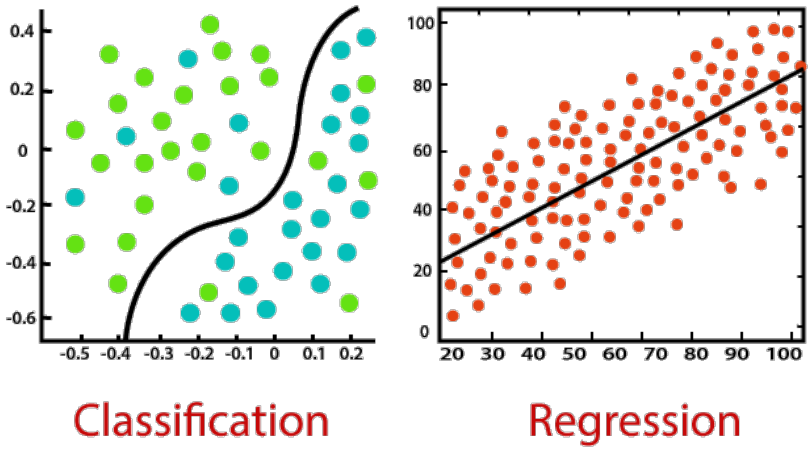

Figure 3.1: Classification vs. Regression in Machine Learning [4]

3.2 Fundamentals of AI and Neural Networks

Whilst this thesis focuses on a very specific sub-field of Artificial Intelligence (AI): Convolutional Neural Networks (CNNs), we also use Feedforward Neural Networks (FNNs) to a great extent during the experimentation section to test certain methodologies. In this chapter we touch on the basics of AI, and how FNNs along with CNNs can be contextualized within the large world of deep neural networks.

Overview

Artificial Neural Networks (ANNs) are computational methods which are inspired by biological neural systems, such as the human brain. History of ANNs is described in detail in literature, such as (Deep Learning by Goodfellow, Bengio and Courville [3] / An Introduction to Convolutional Neural Networks by O'Shea and Nash [5] / Deep Learning Lecture Notes by Prof. Dr.Ing. Andreas Geiger [6]) and will not be repeated here.

In order to continue the discussion, the basic building blocks of neural networks shall be briefly described.

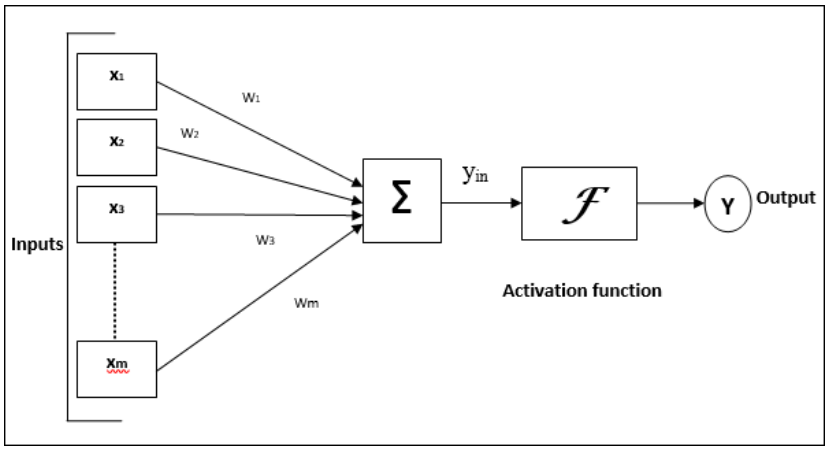

Figure 3.2: A simple model of a neuron within an ANN [7]

The image in Figure 3.2 shows a simplified model of a neuron within an ANN with the following blocks:

- Input: multiple inputs build the input vector → i.e. net input is $\sum_i^m x_i w_i$
- Weights: Each input element is weighted accordingly depending on its importance. With multiple neurons, each neuron is also weighted as per its significance
- The output is computed using an activation function. The activation function reacts to the data provided and provides an additional filter → $Y = F(y_{in})$

This model of a neuron can be used to build a basic structure of an ANN, also known as a Feedforward Neural Network (FNN), as shown in Figure 3.3.

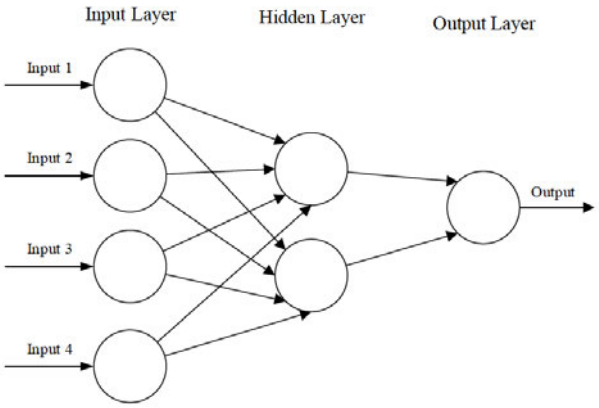

Figure 3.3: Simple schematic of a three layered FFN [5]

The input dataset (usually multidimensional) is loaded into the input layer, which feeds one or more hidden layers. The hidden layers will then make decisions from the previous layer and adjust the weights with regard to how a stochastic change within itself harms or improves the final output. This adjustment of weights is known as the process of learning. The term deep learning is coined from stacking multiple hidden layers one after the other.

Within image processing, which builds the basis of this thesis, one can differentiate between **supervised learning**, where the input dataset is pre-labelled. Here, the input dataset is comprised of two parts: one vector builds the value dataset, and a second vector comprises of the corresponding labels. In order to train a network, the dataset is split into a training set and a testing set. The training set, due to the labelled nature of the data, provides a *current state* and a *desired state*. This information can be used to adjust the parameters of the network through reducing the defined cost function.

Unsupervised learning, on the other hand differs in that the training dataset does not have any labels. Most image processing problems however are dealt with supervised learning algorithms - we will therefore not delve into unsupervised learning deeper. An overview can however be found in literature [5].

Convolutional neural networks (CNN) are a specialized kind of neural network and fall within the framework of supervised learning. The name implies that a mathematical operation called **convolution** is employed at least in one layer. CNNs are particularly advantageous for grid-like topologies, since the convolution operation can be thought of sliding and implementing a filter (kernel) over the data. Therefore CNNs are specialized in inferring information from data that has some spatial structure and correlation. CNNs are hence also extensively implemented in image processing and computer vision applications. The next chapter provides a brief introduction into the basic architecture and building blocks of CNNs.

3.3 Fundamentals of Convolutional Neural Networks

The **convolution** operation is denoted in the following manner:

$$s(t) = (x * w)(t)$$

The first argument, x, is referred to as the **input** and the second argument, w as the **kernel** or the **filter**. The output is often referred to as the **feature map**. The spatial or temporal discretization of the input data would be parameter t. This would mean, the discrete convolution may be defined as $s(t) = (x * w)(t) = \sum_{a=-\infty}^{\inf} x(a)w(t-a)$

Neural networks use kernels to transform the input in a more useful form. For example, in order to classify an image as a dog or a cat, we can perhaps come up with a kernel to extract features specific to a dog, and analyse the differences to a cat. However, this would fail if the kernel was to function on different images of dogs. A computer can however be trained to come up with a more generalized kernel, or better multiple kernels, which would work on different images of dogs. This is a CNN. Therefore, in the context of CNNs, **kernels are also known as filters**. Examples of kernels are shown in Figure 3.4.

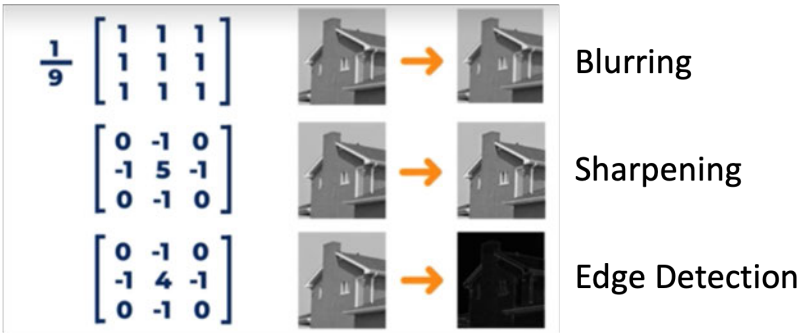

Figure 3.4: Example of Kernels as implemented in CNNs [8]

As mentioned earlier, a CNN differs from a deep NN in that it contains at least one convolutional layer $\rightarrow$ i.e. at least one layer, where the input will convolve with a kernel. This leads us to the largest advantage of CNNs over traditional deep NNs: **the underlying structure of the input data (e.g. image) is kept until the very last layer**. In a deep NN an image would be first transformed to a vector, so the NN would not actually see the image but instead a row of numbers. This would mean a slight change in the image would 'confuse' the network.

Figure 3.5: Comparison of NN and CNN on an image input [8]

Figure 3.5 shows the advantage a CNN possesses over a deep NN when it comes to number of parameters. A CNN 'only' requires 25 weights for the kernel matrix, and 1 for the bias for the given input image. This means, multiple convolutional layers can be stacked together, which would still constitute an immense computational win.

With the background of the convolution operation, along with the utilisation of kernels, we can now introduce a basic CNN architecture, as shown in Figure 3.6.

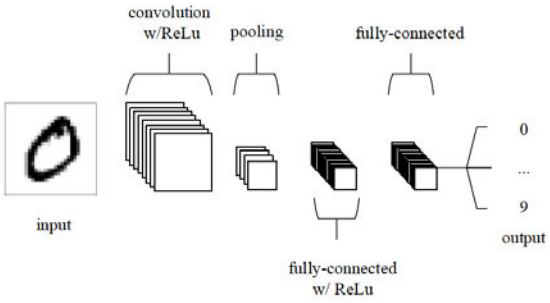

Figure 3.6: A basic CNN Architecture [5]

Feature Maps

Kernels slide over an image to extract a certain feature (dependent on the kernel). Multiple (100s) of kernels are/can be used in a convolutional layer. This means, we would split the input image into its many features. The output of the convolution of the kernel with the image, i.e., the extraction of said feature, results in an output matrix, which is called a feature map. An example is portrayed in Figure 3.7.

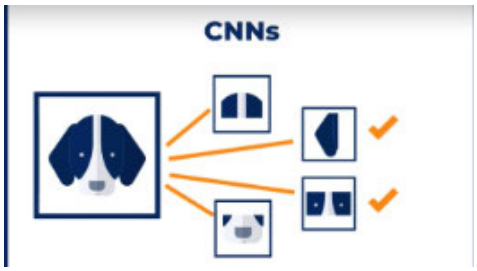

Figure 3.7: An example of feature maps on an input image of a dog [8]

If the input dataset, or images, are black and white, then 2-dimensional kernels are sufficient. On the other hand, coloured images, which are three dimensional in their nature, require the kernels to be three dimensional as well. Since the image has a depth of 3 (r, g, b) the tensor will also have a depth of three.

The output feature map would however still be 2D. Since a convolutional layer would comprise of hundreds of kernels, there would be hundreds of 2D feature maps.

Pooling

The major goal of a pooling layer is to reduce the dimensionality of feature maps. For this reason, **it is also called down-sampling**. The factor to which the down-sampling will be done is called stride or down-sampling factor. An example of how such down-sampling works is shown in Figure 3.8.

This is largely used to avoid overfitting, i.e. to prevent the CNN from learning the training examples too well, so that they cannot predict new 'test' data well. Pooling reduces the noise in a feature map, which therefore makes it more difficult for the NN to learn the noise. This prevents overfitting.

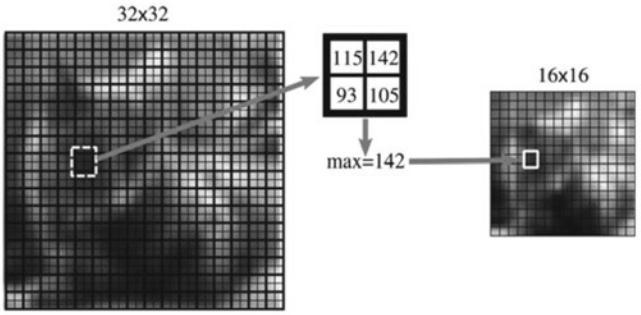

Figure 3.8: Implementation of a pooling layer [9]

A pooling layer can be constructed using any kernel or function that produces a single number, such as Max(). The kernel itself can be of of varying size (KxM). The kernel slides over the image, however, the kernel does not need to slide over each pixel. It can also skip pixels, which is known as the **stride**.

Whilst terms such as the convolutional layer, pooling layer, pooling stride, etc. may seem arbitrary here, they will be implemented within the CNNs used during the experiments in this thesis.

4 State of the art

In this chapter we shall focus on the various deep learning frameworks available and in particular the ones used in this thesis. Since Tensorflow has been used throughout the thesis, the Tensorflow framework and its APIs shall be discussed in detail in this section.

It should also be noted, the sub chapters about the **sequential** and **functional** methodologies made available by Tensorflow build the basis of experiments conducted during the course of this thesis.

4.1 Tensorflow

TensorFlow, TF, is an open-source machine learning framework developed by the Google Brain team. Since its release in 2015, TensorFlow has become one of the most widely used and influential tools in the field of artificial intelligence and deep learning. This chapter provides an overview of TensorFlow, its key features, and its significance in the world of machine learning and artificial intelligence.

Tensorflow is named after tensors - an n-dimensional vector or matrix. These tensors are used in most calculations when programming with Tensorflow. The tensor values are all of the same data type and have a (pre-) known shape, determined by the dimension of the matrix/array. A visualization of tensors is shown in Figure 4.1.

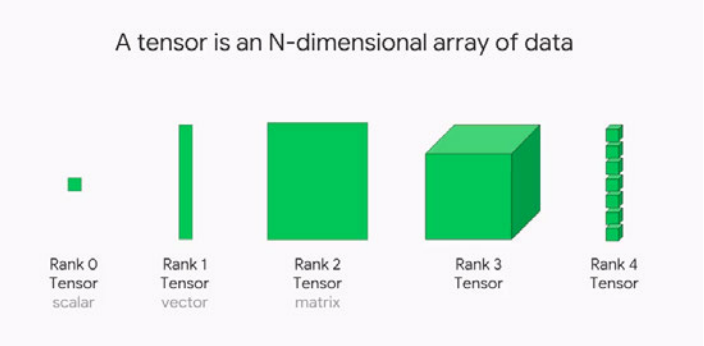

Figure 4.1: Explanation / Visualization of a Tensor [10]

Tensorflow performs all calculations in the form of a graph-based architecture. The graph is made up of sequential computations, where each operation is called a node. Each node is interconnected.

The framework allows the user to design such dataflow graphs through which the flow of the data can be specified by receiving inputs as multi-dimensional arrays (tensors). It enables the user to create a type of flowchart of operations. An example of this can be seen in the next chapter, where an implementation of a CNN has been performed.

Specific to this thesis, we will focus on one of the more popular tools provided by TF, tf.keras. Tensorflow APIs are arranged hierarchically, as shown in Figure 4.2. This allows users to quickly build and train deep learning models without having to write the code from scratch.

Whilst creating a neural network with TF, the architecture is divided into three parts:

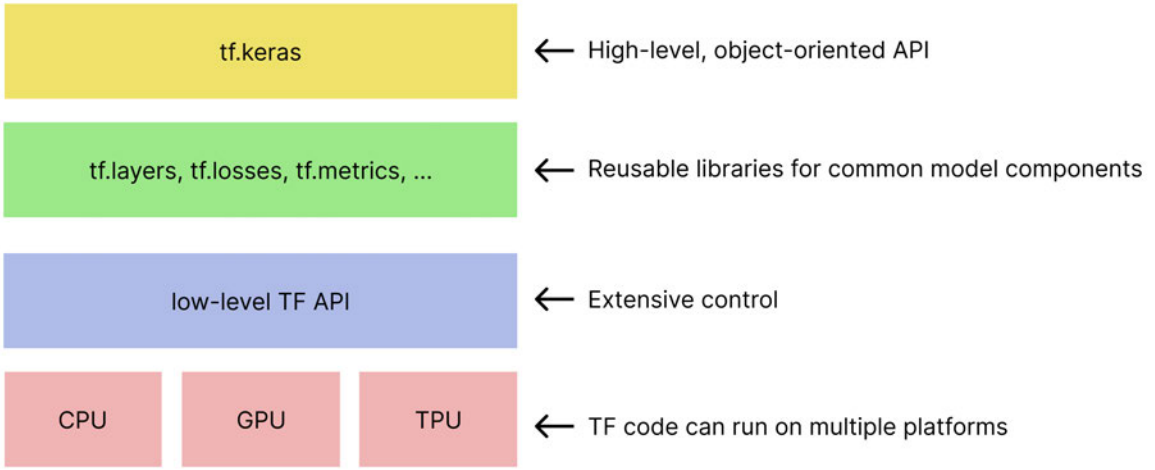

Figure 4.2: Tensor Flow Hierarchy [11]

- Preparing the data
- Creating the model
- Training the model

We will next look at the Keras utility.

4.1.1 Keras

Keras is a high level, deep learning API developed by Google for implementing neural networks. It is written in Python and provides users with multiple back-end libraries, making the programming of neural networks quite straight forward.

Whilst python front-end allows for a high level of abstraction, several back-ends are supported. Tensorflow, as mentioned in the chapter above is naturally one of them. In addition to that, the deeplearning framework from Microsoft, CNTK, as well as that from Meta/Facebook, Theano, are also supported.

Relative to the other back-ends, Tensorflow has integrated Keras as its official high level API. This, combined with the fact that all of the computations can be performed using tensors in computation graphs, gives the user a great amount of flexibility and control over the end application. New ideas can also be implemented and experimented with in a short amount of time.

Building a Model in Keras

- Define the network architecture: Keras offers two central architectures, **sequential** and **functional**. Respective of the problem to be solved, the most apt architecture is chosen and the various layers are implemented within the given framework.
- The model is compiled: A loss function computes the losses of the model, an optimizer helps to calibrate the model to reduce the losses and metrics help portray the quality of the compiled model

- Fitting the network: The model is then fitted to the training data
- The model can then be evaluated using the testing dataset, and then used for predictions if the quality is deemed sufficient. Otherwise the training step is repeated, possibly with an adjustment of certain parameters.

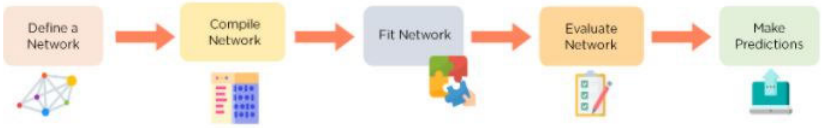

Figure 4.3: Basic steps involved in building a model with Keras [12]

4.1.1.1 Sequential Model

Within the sequential model framework, a neural network is built using **a simple, linear stack of layers**. This means, that the network can also be extended/modified one layer at a time. The layers are arranged in a straight sequence from input to output. Hence, the data flows through the layers in a single direction, which makes it quite suitable for feedforward networks.

Due to this linear topology, sequential models are ideal for networks where each layer has one input tensor and one output tensor. The output tensor of the preceding layer will be the input tensor of the following layer.

Here is a simple, systematic example of a network programmed using the sequential model:

```
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(units=64, activation='relu', input_dim
=100))
model.add(Dense(units=10, activation='softmax'))

```

A visual representation of this network would be as follows:

```
Input
  |
Hidden
  |
Output

```

4.1.1.2 Functional Model

The Functional model in Keras offers more flexibility in the architecture. This framework can be used to build more complex neural networks, including multi-input and multi-output models. Layers can be shared between a number of inputs, and the designs can be truly modified and customized to the problem at hand.

In the background, the framework creates directed acyclic graphs. This means that one can have multiple inputs, multiple outputs, and the layers can be connected in a non-sequential manner.

Due to this level of customization, the code complexity of functional models is higher than that of sequential models.

A simple example utilizing the Functional API is shown below:

```
from keras.layers import Input, Dense
from keras.models import Model

# Define the input layers
input_layer = Input(shape=(100,))

# Create model layers and connect them
x = Dense(units=64, activation='relu')(input_layer)
output_layer = Dense(units=10, activation='softmax')(x)

# Create the functional model
model = Model(inputs=input_layer, outputs=output_layer)

```

The network created above can be visualized as follows:

```
[Input 1]   [Input 2]
    |           |
   [Shared Layer]
    |           |
[Output 1]   [Output 2]

```

In summary, the choice between Sequential and Functional models in Keras depends on the complexity of the neural network architecture. A Sequential model is often sufficient for a simple feed forward network and is easier to work with. However, if more complex models with multiple inputs, outputs, or shared layers are required, the Functional model is the better choice due to its flexibility and customization capabilities.

We make use of the Functional model in the subsequent chapters to show the difference between both the models. In addition to this, the functional model is the key through which we solve our original problem, as stated in the beginning of the thesis.

5 Datasets

This chapter will give an introduction to the datasets used for the experiments described in Chapter 6. Both of the datasets have been pulled from the public domain and are available online. Each dataset has been chosen with the aim to solve a certain problem, and to test the functional approach against the sequential one.

5.1 MNIST

The MNIST database of handwritten digits can be found on its homepage: `http://yann.lecun.com/exdb/mnist/` [13, LeCun et al.]. **MNIST** stands for **M**odified **N**ational **I**nstitute of **S**tandards and **T**echnology and is a subset of the NIST database.

The MNIST database consists of a collection of handwritten digits, which are often used for various tasks, in particular for image processing and classification tasks.

Here is a brief description of the dataset:

- **Contents:**
  - Greyscale images of handwritten digits
  - 60000 images for training purposes and 10000 images for testing purposes
  - The images are 784 pixels large, spanning 28x28 pixels (2-Dimensional, since they are greyscale), and most importantly,
  - the images are **labelled**. This allows us to use the dataset for supervised learning methods, as described in the thesis as of yet.
- **History:**
  - Created by Yann LeCun, Corinna Cortes and Christopher J.C. Burges
  - Often used as a benchmark dataset to research, develop and compare various machine learning algorithms
- More information can be found on the home page of the dataset as well as on several other sources on the internet.

An example of a few images from the dataset has been shown in the figure below.

Figure 5.1: MNIST Dataset with labels

Whilst this dataset is relevant for classification problems, it is not suitable for regression problems. The regression method is based on estimating, and/or extrapolating a certain value, dependant on existing parameters. The nature of the MNIST dataset is, however, to provide a problem where images can be sorted into their respective labels - by definition a classification problem.

5.2 Abalone

In contrast to the MNIST dataset, this dataset is text-based. The motivation for this dataset comes from the task of predicting the age of abalones from physical measurements. The age of an abalone is determined by cutting the shell through the cone, staining it, and counting the number of rings through a microscope – a boring and time-consuming task. Other measurements, which are easier to obtain, are used to predict the age. Further information, such as weather patterns and location (hence food availability) may be required to solve the problem.

This dataset provides possibilities to implement both classification as well as regression models to solve the problem of predicting the age of the abalones.

This was the main criteria to select this dataset for this thesis. On the basis of this, an experiment was devised where both models were implemented using the sequential as well as the functional models in keras. These experiments have been described in detail in the next chapter.

Contents

The abalone dataset was fist introduced in the UC Irvine Machine Learning Repository [14, W. Nash et al.]. Whilst the dataset is typically used for regression tasks to predict the age of the abalones, by counting the number of rings in their shells, we shall also use it to implement classification models.

Purpose of this task is to assist ecological and fishery management institutes in determining the age of abalones based on their physical attributes. A big advantage offered is that the age of the abalones can be determined without having to open their shells.

The dataset consists of several attributes, such as:

- measurements such as length, diameter, height
- various weights such as whole weight, shucked weight, viscera weight, shell weight
- number of rings $\rightarrow$ this is also the target variable. The number of rings is a proxy for the age of the abalone.

The dataset consists of approximately 4000 examples which can be appropriately used and split for training and testing purposes.

In the example below, we use a simple Python routine to portray a few examples from the dataset:

```
from ucimlrepo import fetch_ucirepo

# fetch dataset
abalone = fetch_ucirepo(id=1)

# data (as pandas dataframes)
X = abalone.data.features
y = abalone.data.targets

# metadata
print(abalone.metadata)

```

```
13              # variable information
14              print(abalone.variables)
15
```

Listing 5.1: Abalone, Dataset Example

```
(4177, 9)
   0      1      2      3      4      5      6      7   8
0  M  0.455  0.365  0.095  0.5140  0.2245  0.1010  0.150  15
1  M  0.350  0.265  0.090  0.2255  0.0995  0.0485  0.070   7
2  F  0.530  0.420  0.135  0.6770  0.2565  0.1415  0.210   9
3  M  0.440  0.365  0.125  0.5160  0.2155  0.1140  0.155  10
4  I  0.330  0.255  0.080  0.2050  0.0895  0.0395  0.055   7
```

Figure 5.2: First five entities of Abalone Dataset

As portrayed above, the dataset is built in the following way:

- 4177 examples $\rightarrow$ these can be split into a training and testing dataset
- first coloumn is a string (sex of abalone), others are numerical. First coloumn can be eleiminated from the dataset to keep the model simple
- 7 coloumns which showcase certain features. Last coloumn gives the age, which is to be predicted

The source of the dataset, the Machine Learning Repository of the University of California, Irvine, provides a succinct overview of the parameters of the dataset and their corresponding attributes. These are reiterated here to provide a basic understanding of the data:

Variable Name	Role	Type	Description	Units
Sex	Feature	Categorical	M, F, and I (Infant)	
Length	Feature	Continuous	Longest shell measurement	mm
Diameter	Feature	Continuous	Perpendicular to length	mm
Height	Feature	Continuous	with meat in shell	mm
Whole weight	Feature	Continuous	whole abalone	grams
Shucked weight	Feature	Continuous	weight of meat	grams
Viscera weight	Feature	Continuous	gut weight (after bleeding)	grams
Shell weight	Feature	Continuous	after being dried	grams
Rings	**Target**	**Integer**	**+1.5 gives age in years**	

Table 5.1: Attributes of Abalone Dataset [14]

6 Experiments

This chapter will showcase the experiments conducted during the course of this thesis to investigate the problem described during the motivation of this thesis: whether it is possible to solve two fundamentally different problems with a single neural network configuration. To attempt a solution, we make use of the Functional API provided by Keras.

We will start off with the MNIST dataset, as it also allows us to give a portrayal of basics of CNNs which have been discussed above in theory. We begin by implementing this theory into practice, before diving into the comparison of the different CNN architectures.

6.1 MNIST: Sequential vs. Functional

We first tackle the problem of designing a CNN from scratch. To do this, we shall make use of the sequential model, as it is the most intuitive one.

Once a CNN architecture has been implemented using the sequential approach, an attempt will be made to implement the same to demonstrate the functional approach as well. As mentioned in **section 5.1**, the MNIST Dataset is not suitable to be analysed using the regression method.

However, the basic motivation in this thesis to try out the Keras Functional model is to solve two problems simultaneously. As described in section 5.1, the basic problem provided by the MNIST dataset is to classify each image to its corresponding label and thereby identify the number. For the functional implementation, another fictive problem has been introduced: We shall use the functional approach to divide the identified numbers into two bins: all numbers less than or equal to 5 in one bin (arbitrarily defined as 'written by the left hand') and all numbers larger than 5 in the other bin (arbitrarily defined as 'written by the right hand').

To summarize, the sequential implementation will focus on building a CNN from scratch, thereby using the sequential approach. The next step in this chapter will portray the functional implementation, which shall attempt to solve both of the problems declared above.

Designing CNN from scratch

The MNIST dataset consists of 70000 labelled images of handwritten digits from 0 to 9. Each image is of size 28x28 pixels. The problem to be solved is as follows: **classify an MNIST image to its corresponding number**.

Since the input into the CNN will be a dataset comprising of images, we shall begin with a convolutional layer. A dense layer at this point would unpack the image into an one-dimensional vector, thereby losing the advantage of the convolutional aspect. The ReLu activation function would be a suitable choice for the convolutional layer.

The Kernel/Filter size will need to be odd. An evenly sized kernel would not work, since we shall need a center pixel in order to treat the edges of the images. Experience shows that the kernel should not be larger that 25% of the image, so at most 7x7 pixels. Anything smaller would work too. We shall begin with 50 kernels. This is an arbitrary number of kernels and experimentation can be conducted here to find an optimal number.

Dimension analysis

We can use dimension analysis to design the network.

Lets assume that the input layer is 28x28x1 large (Since the images are black and white. If the images were coloured, then the input layer would be 28x28x3 - i.e. a depth of three, each representing r, g and b). If the first convolutional layer uses a kernel 5x5 large with 50 kernels, the output of the layer would be 24x24x50 (the edge pixels would be omitted). A pooling layer of size 2x2 with a stride of two after that would result in an output of 12x12x50. We can implement another convolutional layer after that, but the kernel can't be larger than 25% of the 'image' (i.e. the output of the previous layer), which is 12 pixels large now. Therefore, the kernel for this layer can be maximum 3x3 large. Lets says we use 50 kernels again. This would give us an output of 10x10x50, again with the edge pixels being omitted. A pooling layer following this with a size of 2x2 and a stride of 2 would result in an output of 5x5x50.

Most of the work has now been performed. To obtain the desired output however, we need the final layer to have 10 different outputs or classes, each corresponding to the digits from 0 to 9. In order to implement this however, we first need to **flatten out the 5x5x50 output into a one dimensional vector**. This can be done with a **flattening layer**, which will make the output 1250x1 large. Finally a dense layer with a Softmax activation function can be used as the final output layer with 10 neurons. The Softmax activation function is used to transform the output into a valid probability density function.

The architecture of the network would be as follows:

```
                [Input Layer] 28x28x1 --> Image 28x28 pixels, 1 dimension (greyscale images. Coloured images would have a dimension of 3)
                        |
                [Conv, 5x5, 50, ReLu] --> Kernel size 5x5, 50 Kernels, with ReLu activation function
                        |               --> Size: 24x24x50
                [Maxpool, 2x2, Stride 2]
                        |               --> Size: 12x12x50
                [Conv, 3x3, 50, ReLu] --> Kernel size 3x3, 50 Kernels with ReLu activation function
                        |               --> Size: 10x10x50
                [Maxpool, 2x2, Stride 2]
                        |               --> Size: 5x5x50
                [Flattening Layer]
                        |               --> Size: 1250x1
                [Dense, 10, Softmax]  --> Size: 10x1, Vector representing the probability of each possible class (in this case each digit)

```

6.1.1 Sequential Implementation

The first experiment was conducted using the previously described MNIST dataset. This section provides a brief description into the implementation of a simple Convolutional Neural Network using the **Keras Sequential API**.

For a detailed look into the entire code, presented as a Jupyter Notebook, the reader is redirected to the Appendix A.1. The focus of this section shall be to showcase the particularities experienced whilst experimenting and implementing the neural network. We shall not focus on trivialities such as which libraries to import, etc. An overview of how the algorithm comes together, however, shall be provided.

The key aim of this experiment is to provide a comparison between both of the keras methods: sequential and functional. In order to provide a fair comparison between both of the methods, care has to be taken that implementations of both methods only differ in the methods themselves. Further parameters and other attributes, which can influence the neural networks, must be kept unchanged between both implementations. Along this chapter, the author has attempted to highlight the attributes, which can affect the performance and quality of the results and have therefore been given extra thought.

The first step is to import all the relevant libraries. The main libraries to be imported include Tensorflow, Keras and some standard mathematical and plotting libraries such as numpy and matplotlib. See Listing A.1.

Next, the dataset can be imported directly from Keras, see Listing A.2. This step highlights an essential advantage of the choice of this dataset. The MNIST dataset is a standard dataset used in the machine learning community. It is therefore integrated into many existing frameworks, including Keras. The intended aim to select the MNIST dataset was to work on a known dataset and with that build a credible reference point.

A short snippet of code below shows an example output of the dataset, divided into the two problem statements described above in Section 6.1.

```
...
...
    # Create a dictionary out of the output labels, with two keys --> in
    order to solve two distinct problems
    y = {"category_output": y_train,
        "leftright_output": y_leftright }
```

Listing 6.1: MNIST Sequential, Loading Dataset from Keras

```
uint8 [5 0 4 1 9 2 1 3 1 4 3 5 3 6 1 7 2 8 6 9]
uint8 [0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1]
```

Creating the Models

The following steps are described in this section:

- Define the architecture of the model
- Compile the model
- Train the model using model.fit method

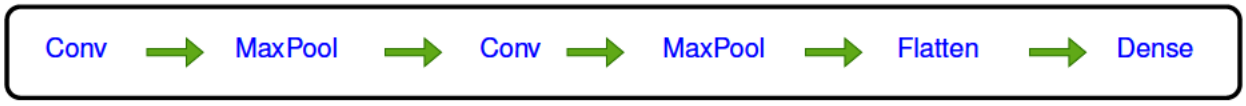

Figure 6.1: MNIST Sequential: Architecture as implemented

As described in the introduction to this experiment in Chapter 6.1, the aim of this experiment is to solve two problems using the same dataset. The problems can be summarised as follows:

1. **Identification**: Identify the digit from the image
2. **Classification**: Classify each identified digit in one of the following bins: Left handed (if less than or equal to 5) or right handed (if greater than 5)

Figure 6.1 portrays the basic architecture used for both of the models. It is essential to point out that the basis of this experiment is built upon this architecture. We shall keep this unchanged during the course of this experiment, so that the subsequent comparison with the functional model can take place without other influences.

Since both of the models are the same apart from the last dense layer, which is the key differentiating parameter, the code is shown here exemplarily for the first model:

```
1    modelCategory = tf.keras.Sequential([
2        tf.keras.layers.Conv2D(50, 5, activation='relu', input_shape
     =(28,28,1)),
3        tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
4        tf.keras.layers.Conv2D(50, 3, activation='relu'),
5        tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
6        tf.keras.layers.Flatten(),
7        tf.keras.layers.Dense(10)
8        ])
```

Listing 6.2: MNIST Sequential, Define Model 'Category'

Similarly, the second problem statement is solved by creating a modelLR using the keras sequential implementation, see Listing A.4 in the Appendix.

The dense layer of the second model is defined as follows:

```
1    ...
2    ...
3    tf.keras.layers.Dense(1)
```

Listing 6.3: MNIST Sequential, Define Model 'LR'

A short comment can be provided here to the choice of the dense layer. A deeper analysis will be shown subsequently in the Dimensional Analysis section. The discrepancy between both of the dense layer originates within the respective problem statements:

- the first problem is to identify a digit between 0 and 9 for each image, i.e. each image is classified into one of 10 bins. This is represented by the size of the last layer.
- the second problem is to classify each image as to whether it is written by the left or right hand. This can be interpreted as a **boolean problem**, where one bin is enough to give sufficient information. The interpretation of the results is a post-processing matter.

The output of both of the models shows the details of the hardware used for testing:

- CPU: Apple M2 Pro
- System RAM: 16.00 GB
- Max. Cache Size: 5.33 GB

This will build the basis of the hardware used during the experiments. As with the basic architecture of the models, the hardware used for the experiments obviously presents an important characteristic as to the performance of the training of the models. The hardware is kept unchanged during the course of this entire thesis.

Using the Keras method model.summary, we can portray each model to check whether they have been defined as envisioned:

```
1  modelCategory.summary()
2  modelLR.summary()
```

Listing 6.4: MNIST Sequential, Model 'Category' Summary

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 24, 24, 50)        1300
max_pooling2d (MaxPooling2D) (None, 12, 12, 50)        0
conv2d_1 (Conv2D)            (None, 10, 10, 50)        22550
max_pooling2d_1 (MaxPooling2D) (None, 5, 5, 50)        0
flatten (Flatten)            (None, 1250)              0
dense (Dense)                (None, 10)                12510
=================================================================
Total params: 36360 (142.03 KB)
Trainable params: 36360 (142.03 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________


Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 24, 24, 50)        1300
max_pooling2d_2 (MaxPooling2D) (None, 12, 12, 50)      0
```

```
conv2d_3 (Conv2D)              (None, 10, 10, 50)      22550
max_pooling2d_3 (MaxPooling2D) (None, 5, 5, 50)        0
flatten_1 (Flatten)            (None, 1250)            0
dense_1 (Dense)                (None, 1)               1251
=================================================================
Total params: 25101 (98.05 KB)
Trainable params: 25101 (98.05 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

A few observations can be made through the outputs shown above:

- *'None'* in the attribute **'Output Shape'** refers to the fact that the batch size is not fixed and has not been predefined. It is therefore variable and will be automatically defined in the model.fit method
- Trainable parameters: these are the weights of the model. In this case, the values of all of the kernels and last dense layer
- Non-trainable parameters: we haven't defined any in both of the models. However, an example of this would be if we decided to multiply the input by a factor (to manipulate or modify it). This factor would not change over the course of the training process, and is therefore defined as a non-trainable parameter.

One could, understandably, question the change in the order of magnitude of the second convolutional layer as opposed to the first. In both models, the number of parameters in the second convolutional layer is more than 10 times that of the first convolutional layer, despite the fact that it succeeds a max pooling layer.

This question can again be answered using dimensional analysis. The first convolution deals with an input of 28x28x1, with fifty 5x5 sized kernels. Hence, the input into the second convolutional layer has a depth of 50, much larger than the depth of the first convolution (depth = 1). This causes the large increase in parameters.

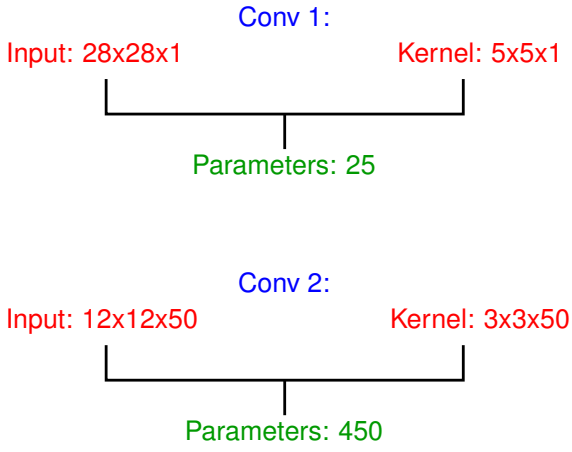

Figure 6.2: MNIST: Dimensional Analysis: Parameters

Dimensional Analysis

Input data/image is **28x28x1**, i.e. the length is 28 pixels, width is 28 pixels and depth is 1, as shown in Figure 6.3

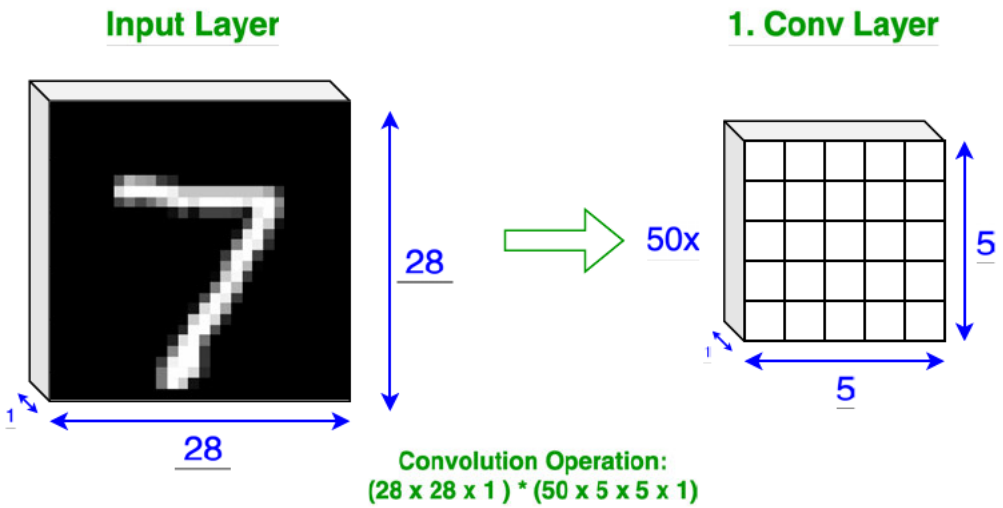

Figure 6.3: MNIST Functional: Dimensional Analysis, First Convolution

The Figure 6.3 also shows the convolution operation as implemented with a 5x5 kernel. The depth of the kernel is the same as the depth of the input data.

The kernel is 5x5 (i.e. odd number), in order to be able to handle the edges, as described previously. This causes the resulting feature map to be of size 24x24, since two pixels are removed from all four sides (since the kernel is 5x5, two pixels would be 'over the edge'). In addition, as there are 50 kernels, the end result is 50 feature maps. Therefore, the resulting dimension of the first convolutional layer is **24x24x50**.

The number of parameters of the convolutional layer can therefore be calculated as follows:

$$Parameters = \underbrace{[Number\ of\ Kernels * Kernel\ Size(length, width, depth)]}_{\text{Weights}} + \underbrace{Number\ of\ Biases}_{\text{Biases}} \tag{6.1}$$

Each kernel is associated with a number of weights (depending on the size of the kernel) and one bias. Since we have 50 kernels, there will be 50 biases. Therefore,

$$Parameters = (50 * 5 * 5 * 1) + 50 = 1300$$

Next, a max pooling layer, with a stride of 2 has been implemented. The pooling layer reduces, in this case halves, the spatial dimension of the feature maps. The number of feature maps does not reduce. Hence, the resulting output dimension is **12x12x50** and **no additional parameters are added through this layer** (the reader may refer to the model summary shown by Listing 6.4).

Similarly to the first convolutional layer, the second convolutional layer can now be discussed. This has been implemented again with 50 kernels, this time however of size 3x3. This means, due to edge handling, only one pixel would go 'over the edge'. The resulting feature map would be of size 10x10, since only one pixel would be removed from each side. Again, the depth of the feature map is taken over from the previous layer, resulting in an output dimension of **10x10x50**.

The number of parameters created in this layer can be computed as follows:

$$Parameters = (50 * 3 * 3 * 50) + 50 = 22550$$

The following pooling layer 'max_pooling2d_1' again halves the spatial dimensions, not the number of feature maps. This causes an output dimension of **5x5x50**, although again **does not create further parameters**.

The flattening layer transforms the data from a spatial dimension to a vectorial dimension, so that it can be further processed by a fully connected dense layer. This is simply a transformation, **so no further parameters are added**. It can be thought of as flattening an image into a row of pixels. Therefore, the resulting output dimension is simply 5x5x50 = **(1, 1250)**. For the sake of simplicity, Keras omits the '1' in the model summary output.

As can be seen in the model summary of both the models, until now, the dimension analysis of both the models is identical. The number of parameters needed to be trained until the final dense layer do not vary between the models.

This leads us to the output layers of each model. The output layers are fully connected dense layers. Here, each input has a separable weight to each output unit. For a inputs and b outputs, the resulting number of weights is $a * b$. Additionally, each output node has one bias. Therefore, the total number of parameters becomes:

$$Parameters = \underbrace{(a+1) * \overbrace{b}^{\text{Biases}}}_{\text{Weights}} \tag{6.2}$$

The output layer of the first model consists of 10 nodes, since 10 outputs (0 to 9) are required. This causes the output dimensions to be simply **(1x10)**. The number of parameters now are:

$$Parameters = (1250 + 1) * 10 = 12510$$

The output layer of the second model, consists of only 1 node, since only a 'true' or 'false' output is required. This causes the output dimensions to be simply **(1x1)**. The number of parameters now are:

$$Parameters = (1250 + 1) * 1 = 1251$$

Since the idea here is to run the models subsequently, i.e. each model shall be trained individually and run one after the other. Therefore, if we look at the problem in its entirety, the total number of dimensions that will be trained can be computed as follows:

$$\begin{aligned} Total\ Parameters &= \underbrace{(1300 + 22550 + 12510)}_{\text{Model Category}} + \overbrace{(1300 + 22550 + 1251)}^{\text{Model LR}} \\ &= 36360 + 25101 \\ &= 61461 \end{aligned}$$

Simultaneously, the size in memory of the models can also be used as a metric to evaluate the size of the model. Eventually, this will prove to be an important metric when it comes to evaluating model performance:

$$\begin{aligned} Total\ Memory\ Used &= \underbrace{(142\ kB)}_{\text{Model Category}} + \overbrace{(98\ kB)}^{\text{Model LR}} \\ &= 240\ kB \end{aligned}$$

Loss Function and Model Compilation

- For classification of digits from 0 to 9, a sparse categorical cross entropy loss function works well
- For classification of digits whether left handed or right handed, the binary cross entropy function would be suitable

The compilation step combines each model with its respective loss function and the chosen optimizer. The choice of optimizer would naturally have an influence on the training routine. Also here, in order to keep complete control over the models and to ensure a fair comparison in the subsequent chapters, the optimizer has been kept as a 'constant' throughout the course of the experiment.

The optimizer builds the actual 'engine' or 'computation' of the CNN. This describes how the trainable parameters will be varied to achieve the required goal. As the model is trained, the parameters (weights and biases, see calculations above) are tuned to minimize the loss in order to improve the validation accuracy and with that of course also the prediction accuracy. How these parameters are tuned, is dependent on the chosen method of optimization. A loss function is simply a metric telling the optimizer, how well the model would perform in the current iteration.

Basically, during each iteration, the chosen optimizer will tune the parameters in a defined manner. These parameters shall be used to compute an output from the model. The output from the model is then compared with the training data, and the 'difference' between the two, to put it simply, is quantified in the loss function. The aim is to minimize the loss function, hence maximizing the training accuracy. When this step is repeated, the direction the change in the loss function moves in, describes whether the optimizer is tuning the parameters in the correct direction.

A popular approach to solve such optimization problems is stochastic gradient descent. This is a very common approach in deep learning and has been described various literature and numerous textbooks, e.g. [3, Goodfellow], [5, O'Shea and R.Nash] to name a few. The reader is therefore directed towards the literature for further information on gradient descent.

The ADAM optimizer, provided in Keras, is an implementation of the stochastic gradient descent algorithm. This has been chosen and implemented in this experiment as well, see e.g. Listing A.8 in the Appendix.

Training the Models

The fundamental problem within training the model to the given dataset lies within answering the question: when should we end the training, i.e. when is the perceived accuracy of the model sufficient?

A distinction has been made between **validation accuracy** of the model and the **prediction accuracy** of the model. This shall be described in further detail at this point:

- Validation accuracy: this is the accuracy which shall be evaluated during the training process: i.e., when we train the model with labelled, known data
- Prediction accuracy: this would be the accuracy the model would achieve when we test it against unknown, albeit it labelled data. This would give us the metric as to how well we can solve the fundamental problem at hand, e.g. identification of the digits, with data unknown to us.

During the training process, the focus shall be on the validation accuracy. In the implementation below, we have chosen the training duration to be of 15 EPOCHS. This number is chosen after conducting multiple tests, as it has provided the best validation accuracy as of yet. The method to obtain this number (15 EPOCHS) is however prone to trial and error.

Another method would be **early stopping**. The Keras method 'early stopping' allows one to track a certain metric (e.g. the validation loss or accuracy) and stops the training after a certain plateau has reached over a defined number of training runs. One large advantage of early stopping is to reduce over-fitting of the model over the known training dataset as much as possible.

Whilst the method **early stopping** has not explicitly been implemented here, it was experimented with during the course of this thesis. Through these tests, it was determined that a minimum of 12 Epochs were necessary to reach an appropriate minimum of the loss function. An additional two Epochs after the first twelve ones were needed to evaluate that a plateau in the loss function was reached (i.e. one can image a turning point of an arbitrary function). It is through these experiments that 15 EPOCHS was decided upon.

One disadvantage of directly implementing the 'early stopping' method in these experiments is, dependent on the course of the loss function, the number of Epochs needed for training the models would be variable. Since this is also a parameter which heavily influences the performance and quality of the models, it was decided that the final implementation of the networks would be done using a predefined number of Epochs. This gives the author complete control over the model and ensures future comparability with other methods.

The following shows the training results of each of the models. Only the last few iterations of each training step have been shown here - for the entire Listing and corresponding output, the reader is referred to the Listings A.9 and A.10.

```
modelCategory.fit(x_train, y=y_train, epochs=15,
      batch_size=64, verbose=2)

modelLR.fit(x_train, y_leftright, epochs=15,
        batch_size=64, verbose=2)
```

Listing 6.5: MNIST Sequential, Training Both Models

```
---------------------- Model Category ----------------------
...
...
Epoch 14/15
938/938 - 8s - loss: 0.0063 - accuracy: 0.9979 - 8s/epoch - 9ms/step
Epoch 15/15
938/938 - 8s - loss: 0.0049 - accuracy: 0.9984 - 8s/epoch - 9ms/step


---------------------- Model Left/Right ----------------------
...
...
Epoch 14/15
938/938 - 9s - loss: 0.2274 - accuracy: 0.9504 - 9s/epoch - 9ms/step
Epoch 15/15
938/938 - 9s - loss: 0.1189 - accuracy: 0.9688 - 9s/epoch - 9ms/step
```

The output shows that the model to categorize the input image to a corresponding number achieves a training accuracy of 99.8%. However, the model to group the images into left-handed or right-handed only achieves an accuracy of 96.9%.

Testing the Models

Next, we can make use of the Keras method model.predict to test the models with the test (i.e. unknown) data.

```
# list with predictions
predictionCategory = modelCategory.predict(x_test)
len(predictionCategory)
```

```
4
5     predictionLR = modelLR.predict(x_test)
6     len(predictionLR)
7
8
9     # lets observe how the model works for the first 100 MNIST images
10    numMNIST = 100
11
12    labels_cat = np.argmax(predictionCategory[0:numMNIST], axis=1)
13    labels_lr = np.array([1 if p >= 0.5 else 0 for p in predictionLR[0:
      numMNIST]])
14
15
16    print('Test Labels: ', y_test[0:numMNIST])
17    print('Predicted Labels, Digit: ', labels_cat)
18    print('Predicted Labels, L/R: ', labels_lr)
```

Listing 6.6: MNIST Sequential, Testing both Models

```
313/313 [==============================] - 1s 2ms/step
313/313 [==============================] - 1s 2ms/step
Test Labels:
[7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6 6 5 4 0 7 4 0 1 3 1 3 4 7 2 7
 1 2 1 1 7 4 2 3 5 1 2 4 4 6 3 5 5 6 0 4 1 9 5 7 8 9 3 7 4 6 4 3 0 7 0 2 9
 1 7 3 2 9 7 7 6 2 7 8 4 7 3 6 1 3 6 9 3 1 4 1 7 6 9]
Predicted Labels, Digit:
[7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 2 4 9 6 6 5 4 0 7 4 0 1 3 1 3 4 7 2 7
 1 2 1 1 7 4 2 3 5 1 2 4 4 6 3 5 5 6 0 4 1 9 5 7 8 9 3 7 4 6 4 3 0 7 0 2 9
 1 7 3 2 9 7 7 6 2 7 8 4 7 3 6 1 3 6 9 3 1 4 1 7 6 9]
Predicted Labels, L/R:
[1 0 0 0 0 0 0 1 0 1 0 1 1 0 0 0 1 1 0 0 1 1 1 0 0 0 1 0 0 0 0 0 0 0 1 0 1
 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 1 1 1 0 0 0 1 0 0 0 1 0 0 1
 0 1 0 0 1 1 1 1 0 1 1 0 1 0 1 0 0 1 1 0 0 0 0 1 1 1]
```

Performing a quick check to see how well the identification of the digits works, by computing the difference between the **test labels** and the **predicted labels** (ideally, all of them should be 0):

```
1     print('Test Labels - Predicted Labels: ',y_test[0:numMNIST]-labels_cat)
```

```
Test Labels - Predicted Labels:  [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0]
```

This shows, when the model is tested against the first 100 images in the MNIST dataset, it only makes one error. The same is true for the first 1000 images.

6.1.2 Functional Implementation

This section is a follow-up of the sequential implementation of the Convolutional Neural Network to make predictions using the MNIST dataset.

The same dataset shall be implemented here, using the same problem statements, but using the functional architecture as provided by the MNIST dataset. This architecture has been briefly described in Chapter 4.1.1.2 and will be implemented here.

The code has been implemented in Jupyter Notebooks, and has been provided in Appendix A.2. Here, once again, the particularities of this experiment shall be portrayed in an attempt to show the advantages and disadvantages of this approach.

In order to ensure comparability with the previous experiment, most of the structure and code is identical, especially the trivialities such as libraries required or downloading the dataset. The code shall not be repeated here. We therefore jump straight to the model creation step.

Creating the Model

The steps here are identical to the previous chapter (see Section 6.1.1. We shall begin by creating the model, compiling it and then training it using model.fit method.

The model will be built in a similar fashion, with the only difference being that we attempt to solve both the problems using one model. Schematically, the architecture is shown in Figure 6.4.

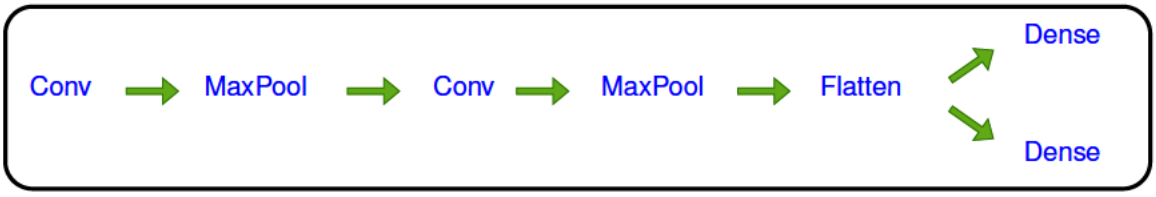

Figure 6.4: MNIST Functional: Architecture as implemented

The following comments can be made at this point:

- Figure 6.4 shows that the fundamental architecture has been kept unchanged. The implementation below will show that the details (e.g kernel size, number of kernels, etc.) have also been kept identical to the sequential implementation
- The final output comprises of two fully connected, dense layers in this case. Here, both of the models implemented in the previous chapter are combined as shown in the schematic. The parameters of the dense layers, however, have been kept unchanged.

We now continue with the actual implementation of the model, as shown in Listing 6.7

```
1    img_rows = 28
2    img_cols = 28
3
4    inputLayer = Input(shape=(img_rows, img_cols, 1))
5
6    layer1 = Conv2D(50, kernel_size=(5,5), activation='relu')(inputLayer)
```

```
7
8     layer2 = MaxPooling2D(pool_size=(2,2))(layer1)
9
10    layer3 = Conv2D(50, kernel_size=(3,3), activation='relu')(layer2)
11
12    layer4 = MaxPooling2D(pool_size=(2,2))(layer3)
13
14    layer5 = Flatten()(layer4)
15
16    layer6 = Dense(10, activation='softmax', name='category_output')(layer5)
17    # This is the first output layer, with 10 outputs
18
19    layer7 = Dense(1, activation='sigmoid', name='leftright_output')(layer5)
20    # This is the second output layer, with 1 output
```

Listing 6.7: MNIST Functional, Define Functional Model

The output of the Listing 6.7 confirms the details of the hardware used for testing. As mentioned earlier, this is one of the factors that are being kept constant. The details of the hardware shall not be reiterated here, as the reader is referred to the previous chapter.

A few observations can be made here, in comparison to the sequential implementation. First of all, the largest difference between the two implementations is that with the functional architecture, each layer has to be linked to a previous layer. Without this, the architecture would break down. This builds the fundamental difference between both the architectures, with the functional architecture providing further degrees of freedom due to this. In other words, a separate input can be defined for each layer, as can be seen at the end of the definition of each layer in parenthesis - it is not assumed that the previously defined layer is the input for the current layer, i.e. **the concept of a 'previous' layer does not exist**: $layer4 = MaxPooling2D(pool_size = (2,2))$**(layer3)** .

The other, and the most relevant, difference, is built upon the construct described above: **Two output** layers are defined, layers 6 and 7 in the case above. Each layer uses layer 5, the **Flattening** layer, as the input.

With the architecture of the model defined, the actual model can now be created. Once again, the method model.summary shall be utilized to display and cross-check the model.

```
1     model = keras.Model(inputs=[inputLayer], outputs=[layer6, layer7], name="
      mnist_model")
2
3     model.summary()
```

Listing 6.8: MNIST Functional, Model Summary

It should be noted, the definition of model clearly states one input layer and two output layers. Again, it should be stressed that this is precisely the advantage presented by this architecture, allowing the user complete freedom as to create a model architecture as required.

```
Model: "mnistFunc_model"
_________________________________________________________________
```

```
 Layer (type)        Output Shape          Param #   Connected to
==================================================================================
 input_1             [(None, 28, 28, 1)]   0         []
 (InputLayer)

 conv2d              (None, 24, 24, 50)    1300      ['input_1[0][0]']
 (Conv2D)

 max_pooling2d       (None, 12, 12, 50)    0         ['conv2d[0][0]']
 (MaxPooling2D)

 conv2d_1            (None, 10, 10, 50)    22550     ['max_pooling2d[0][0]']
 (Conv2D)

 max_pooling2d_1     (None, 5, 5, 50)      0         ['conv2d_1[0][0]']
 (MaxPooling2D)

 flatten             (None, 1250)          0         ['max_pooling2d_1[0][0]']
 (Flatten)

 category_output     (None, 10)            12510     ['flatten[0][0]']
 (Dense)

 leftright_output    (None, 1)             1251      ['flatten[0][0]']
 (Dense)
==================================================================================
Total params: 37611 (146.92 KB)
Trainable params: 37611 (146.92 KB)
Non-trainable params: 0 (0.00 Byte)
----------------------------------------------------------------------------------
```

Once again, this opportunity will be utilized to make a few observations.

First of all, the point made above with respect to defining an input layer for each layer is shown in the Listing above. In comparison to the model.summary of the sequential model, as seen for example in the Listing A.5, the summary of the functional model consists of one additional attribute: **Connected to**. This shows, that the first layer, i.e. the input layer, called arbitrarily 'input_1' is consists of a data structure of size 28x28x1, is however not connected to any other input. This represents the input data from the black and white MNIST dataset (reminder: each image is 28x28x1 pixels large).

It should also be noted, that the number of parameters in this case is approx. 37600 - an unmistakable reduction in the number of parameters as opposed to the sequential implementation.

Additionally, an advantage of analysing the model.summary is the transparency provided in the implementation of the model. Using this summary, it is very simple to understand why and how many parameters are needed to solve the problem at hand. Taking the example above, one can go through the model step by step and obtain the number of parameters using dimensional analysis.

Dimensional Analysis

Since both the implementations up until the implementation of the output layers are the same, the calculations for the number of parameters shall not be repeated.

The only difference, and large advantage, of the functional model is again shown when the total number of parameters in the entire network are calculated.

The computation comes about in the following manner:

$$\begin{aligned} Total\ Parameters &= \underbrace{1300 + 22550}_{\text{Identical to Sequential}} + \overbrace{12510}^{\text{Output Layer, Category}} + \underbrace{1251}_{\text{Output Layer, LR}} \\ &= 23850 + 12510 + 1251 \\ &= 37611 \end{aligned}$$

In a similar fashion, the size of the model can also be highlighted: **147kB**.

This shows, that the functional implementation consists of $61461 - 37611 = 23850$ less parameters and approximately 90kB of less space in memory.

This large difference between both the implementations can be simply explained. In the sequential implementation, two independent networks were needed, one for each problem statement. This means, both networks comprise of the same initial layers, until the final output layer comes into play. At the end, to solve both of the problems, both the networks have to be trained and run sequentially, hence increasing their resource requirement. The functional implementation, on the other hand, does the same job in a more efficient manner - at least with respect to the number of parameters. Whether it is an efficient approach with respect to performance/efficiency will be briefly discussed in the Chapter 7.

Defining Loss Function

The choice of the loss functions is kept identical to that during the sequential implementation. This was done not only to provide meaningful comparison, but also since the choice matches the problem at hand. Therefore, here too the category problem shall be judged by a **sparse categorical cross-entropy loss function** and the left/right problem shall be judged by a **binary cross-entropy loss function**, as shown in the Listing 6.9.

```
1
2     loss1 = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

```
    # the parameter 'from_logits=True' tells tensorflow to incorporate the
    SOFTMAX activation function in the loss function
    loss2 = keras.losses.BinaryCrossentropy(from_logits=False)
    # binary cross entropy loss function for classification 0 / 1

    losses = {
        "category_output": loss1,
        "leftright_output": loss2,
    }
```

Listing 6.9: MNIST Functional, Define Loss Functions and Loss vector

The difference here to the sequential implementation can be seen in the second part of the code. Since both the problems are being solved using one network, the compilation of the network will require a loss function for both of the dense layers together. This is realized by grouping both of the loss functions in a python list, and feeding this grouped loss into the compilation step (see Listing 6.10.

```
    optim = keras.optimizers.legacy.Adam(learning_rate=0.001)

    model.compile(optimizer=optim, loss=losses, metrics = ['accuracy'])
    # metrics of model should be stored in a list called accuracy
```

Listing 6.10: MNIST Functional, Compilation of Model

Training the Model

As mentioned in the sequential implementation, a predefined number of Epochs (15) was used to train the model. The main motivation of doing this this way, instead of via **early stopping**, was to keep complete control over the process and to provide comparable results.

```
model.fit(x_train, y=y, epochs=15,
          batch_size=64, verbose=2)
```

Listing 6.11: MNIST Functional, Training the Model

```
...
...

Epoch 14/15
938/938 - 9s - loss: 0.0259 - category_output_loss: 0.0059 -
leftright_output_loss: 0.0200 - category_output_accuracy: 0.9981 -
leftright_output_accuracy: 0.9930 - 9s/epoch - 10ms/step
Epoch 15/15
938/938 - 9s - loss: 0.0236 - category_output_loss: 0.0057 -
leftright_output_loss: 0.0179 - category_output_accuracy: 0.9981 -
leftright_output_accuracy: 0.9936 - 9s/epoch - 10ms/step
```

In comparison to the sequential implementation, as expected, two losses are provided. Both of the losses, one each for each problem statement, correspond to the **validation accuracy**, as explained in the sub section A.1. The validation accuracy achieved for the identification of digits (category output) is 99.8%, whilst that for the binning of the digits into one of two groups (leftright output) is 99.4%.

Testing the Model

```
    # list with predictions
    prediction_new = model.predict(x_test)
    len(prediction_new)

    # identification of the digit is the first element of the prediction list
    prediction_category = prediction_new[0]
    # classification left/right is the second element of the prediction list
    prediction_lr = prediction_new[1]

    # lets observe how the model works for the first 100 MNIST images
    numMNIST = 100

    pr_cat = prediction_category[0:numMNIST]
    prediction_lr = prediction_lr[0:numMNIST]

    labels_cat = np.argmax(pr_cat, axis=1)
    labels_lr = np.array([1 if p >= 0.5 else 0 for p in prediction_lr])


    print('Test Labels: ', y_test[0:numMNIST])
    print('Predicted Labels, Digit: ', labels_cat)
    print('Predicted Labels, L/R: ', labels_lr)

```

Listing 6.12: MNIST Sequential, Testing both Models

```
313/313 [==============================] - 1s 2ms/step
Test Labels:  [7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6 6 5 4 0 7 4
0 1 3 1 3 4 7 2 7 1 2 1 1 7 4 2 3 5 1 2 4 4 6 3 5 5 6 0 4 1 9 5 7 8 9
3 7 4 6 4 3 0 7 0 2 9 1 7 3 2 9 7 7 6 2 7 8 4 7 3 6 1 3 6 9 3 1 4 1 7
6 9]
Predicted Labels, Digit:  [7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6
6 5 4 0 7 4 0 1 3 1 3 4 7 2 7 1 2 1 1 7 4 2 3 5 1 2 4 4 6 3 5 5 6 0 4 1
9 5 7 8 9 3 7 4 6 4 3 0 7 0 2 9 1 7 3 2 9 7 7 6 2 7 8 4 7 3 6 1 3 6 9 3
1 4 1 7 6 9]
Predicted Labels, L/R:  [1 0 0 0 0 0 0 1 0 1 0 1 1 0 0 0 1 1 0 0 1 1 1
0 0 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0
1 0 1 1 1 0 1 0 1 0 0 0 1 0 0 1 0 1 0 0 1 1 1 1 0 1 1 0 1 0 1 0 0 1 1
0 0 0 0 1 1 1]
```

Performing a quick check to see how well the identification of the digits works, by computing the difference between the **test labels** and the **predicted labels** (ideally, all of them should be 0):

```
print('Test Labels - Predicted Labels: ',y_test[0:numMNIST]-labels_cat)
```

```
Test Labels - Predicted Labels:  [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

6.2 Abalone: Sequential vs. Functional

The first half of this chapter is dedicated to portraying the Functional API by Keras implemented via a practical example. The choice of using the MNIST Dataset is not coincidental, rather it is quite intentional.

The MNIST images are one of the most common problems in the deep learning community. These images are used commonly to solve classification problems, and to try out different architectures. Therefore, a large number of studies have already been performed with the MNIST dataset. The datasets represents the current state of the art when it comes to labelled data, especially image based data, and hence gives us a good reference to compare our experiments to.

Whilst the above is principally true, the reader is reminded of the true motivation of this thesis: to use the results described by Mr. Schreiber in his Thesis [1, C. Schreiber: KI-gestützte „follow-me“-funktion am Beispiel des JetRacer] and to provide methods which could theoretically be used to optimize his implementations.

The implementations were two-fold: first implementation was of a Follow-Me function, where the Jetson Nano (the hardware used in the thesis) followed a predefined path. This was described as a regression problem, where the deviation between the actual path and the target path was minimized. The second problem was for the robot to detect a stop sign and react accordingly. Three different bins were defined to classify each stop sign as (far away, close by, very close by). Further details may be found in the thesis.

The first experiment described here was used to showcase that two different problems can be solved either in series, i.e. one after another, as also implemented by Mr. Schreiber, or they may also be implemented within one network, as shown in the functional implementation. However, one large simplification was made in this experiment: **both problems that were solved were classification problems**. The MNIST Dataset was nevertheless chosen for the first experiment because of the advantages explained above. In addition, the dataset provided an opportunity to work with image based data, which at least is in line with Mr. Schreiber's work.

After having observed the positive results from the MNIST dataset, and a viable proof of concept regarding the Keras functional architecture, the author was keen to implement the same concepts on another dataset. **The idea here is to prove to the reader that the concepts explained in this thesis are not only independent of the input data and its attributes (image based, text based, data size, etc.), but also independent of the type of problems that are solved**.

To this extent, a second experiment was devised during the course of this thesis using the Abalone Dataset, which has briefly been described in Chapter 5.2. The fundamental motivation of this experiment is three fold:

- To test the hitherto described concepts on a text based dataset
- To test the concepts on both regression and classification problems, hence making sure that the ideas may be transferred to the problems that Mr. Schreiber described in the outlook of his thesis.
- To provide and prove to the reader that the concepts may be implemented on wide variety of problem statements and on very different kinds of input datasets.

Being a text based dataset, one compromise however had to be made: the implementations that will follow in this chapter have not been done using CNNs, but rather using relatively simple feed forward networks, as described in the Fundamentals, e.g. in Figure 3.3.

An excerpt of the dataset is shown with the Listing 6.13. The entire code, in the format of a Jupyter Notebook, along with all of the detailed outputs, may be found in the Appendix.

```
    # load dataset
    # fetch dataset
    abalone = fetch_ucirepo(id=1)

    # data (as pandas dataframes)
    # split dataset into features and targets
    X = abalone.data.features
    y = abalone.data.targets

    # metadata
    print(abalone.metadata)

    # variable information
    print(abalone.variables)
```

Listing 6.13: Abalone, Dataset Example

The dataset is downloaded from the UC Irvine ML Repository (ID = 1). It is then split into its features (X value) and target (y value). The metadata of the the dataset gives an explanation to the background of the data, as covered in Chapter 5.2. The variables show the attributes of the data, as explained in Table 5.1. These will not be displayed here again, the reader however is directed towards the Appendix for the formal outputs.

For this task, the dataset needs to be pre-processed before it may be used to solve the problem at hand. First of all, the information about the gender/sex of the Abalones will not be used. We are attempting to solve the problem with regression and classification and will only be using numerical data in our attempts. More over, a partition of the data shall be done into a training dataset and a testing dataset. This provides the author with some untested data to evaluate the accuracy of the models. 33% of the data is classified as the test data, the remaining data will be used to train the model. The function 'train_test_split()' is a built-in function provided by the library *Scikit Learn* [15, Pedregosa et al.].

```
    # split data into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
     random_state=1)
    print('Size of X_train is ', len(X_train))
    print('Size of X_test is ', len(X_test))
    print('Size of y_train is ', len(y_train))
    print('Size of y_test is ', len(y_test))
```

Listing 6.14: Split data into training and testing datasets

```
Size of X_train is  2798
Size of X_test is  1379
Size of y_train is  2798
```

```
Size of y_test is  1379
```

It should be noted, the goal of this experiment is **not** to create the best model to solve the problem of determining the age of abalones. This problem is just a means to showcase a different implementation of the sequential and functional architectures of Keras. The final goal, as with the last experiment, is to convince the reader that:

1. Two fundamentally different kinds of problems (regression and classification) can be grouped together into one network
2. The Keras functional architecture is not dependent on the type of neural network - it may be used with CNNs, as shown in the MNIST example, as well as deep neural networks, as will be shown here
3. The methods shown here are not dependent on the type of input data - be it images as with MNIST, or textual, as here.

6.2.1 Regression Task

In this chapter, the problem of determining the age of an abalone shall be considered as a regression problem, i.e. the input parameters are used to determine a numerical output. Since the data is textual by nature, an implementation of a CNN here is not necessary.

To demonstrate a possible solution, a simple multi-layer perceptron model shall be implemented. The reader is referred to the basic architecture of feed forward networks described in the fundamentals in the beginning of the thesis, see Figures 3.2 and 3.3.

As with the previous experiment, only the essential ideas will be presented here. The entire code as well as the corresponding output blocks may be found in the Appendix.

Defining the model

After the dataset has been downloaded and prepared for the problem, i.e. it has been split into the training and testing sets, the architecture of the model shall be defined. In order to keep the solution as simple as possible, and also to ensure that a comparison between this and subsequent implementations is possible, the architecture presented in this section was decided upon.

```
1    model = Sequential()
2    model.add(Dense(20, input_dim=number_features, activation='relu',
     kernel_initializer='he_normal'))
3    model.add(Dense(10, activation='relu', kernel_initializer='he_normal'))
4    model.add(Dense(1, activation='linear'))
```

Listing 6.15: Sequential Regression: Defining the model

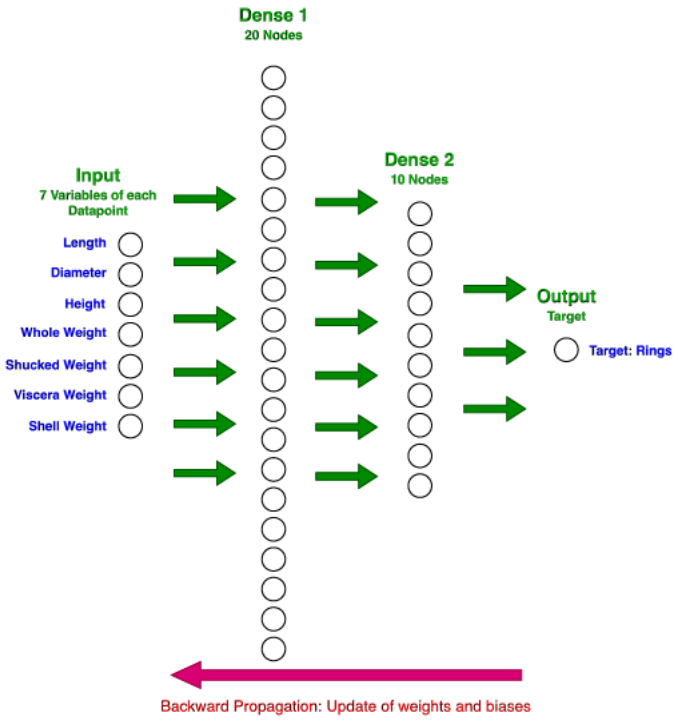

Figure 6.5: Abalone Sequential: Architecture schematic, Regression

Before we move on to a dimensional analysis of this model, a few comments shall be made at this point:

- As shown in the Figure 6.5 above, the model comprises of three layers, with two dense layers
- The size of each dense layer has been chosen arbitrarily
- The final output layer only needs to have one node, since only one final value (number of rings) needs to be computed and returned
- Each data point of the input dataset consists of seven attributes, each of which are then evaluated in the model
- As the name suggests (**feed forward network**), the actual computation of the number of rings, which ultimately is converted to the age of the abalone, happens from left to right, i.e. in the forward propagation of the model
- This value is then compared with the actual number of rings from the validation data, the loss is computed and the actual algorithm to reduce the loss sets in by adjusting the parameters (weights and biases) until the loss is minimized in an iterative manner

The number of dimensions of the model may be portrayed via its summary:

```
1 model.summary()
```

Listing 6.16: Sequential Regression: Model Summary

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 20)                160

 dense_1 (Dense)             (None, 10)                210

 dense_2 (Dense)             (None, 1)                 11

=================================================================
Total params: 381 (1.49 KB)
Trainable params: 381 (1.49 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 6.6 shows a visualization of the dimensions of the neural network. The number of dimensions can be computed quite simply. According to Equation 6.2, the computation of the number of parameters has been shown in Figure 6.6.

Therefore, the total number of parameters are as follows, taking up **1.49 kB** of space in memory:

$$Total\ Parameters = \underbrace{160}_{\text{Dense Layer}} + \overbrace{210}^{\text{Dense Layer}} + \underbrace{11}_{\text{Output Layer}}$$
$$= 381$$

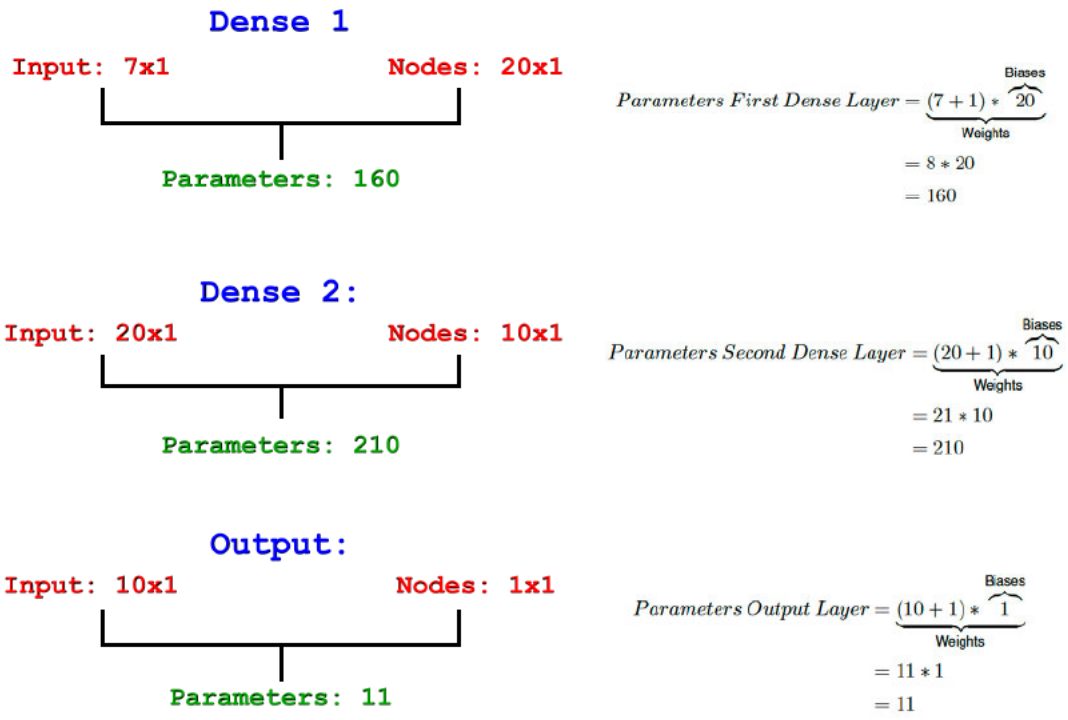

Figure 6.6: Abalone Sequential: Regression Model Dimensions

Loss Function and Model Compilation

Having defined the model and understood the number of dimensions created, we now proceed with defining the loss function and the optimizer.

The optimizer selected here is, as in the previous MNIST example, the ADAM optimizer - as explained before, this is the implementation of the stochastic gradient descent algorithm in Keras.

Since the choice of the optimizer plays a crucial role here, due to the absence of any other elements in a simple feed forward network (in comparison to a CNN with its intricacies such as pooling, convolving, etc.), we shall describe the ADAM optimizer in more detail.

Adam Optimizer

This optimizer implements the ADAM algorithm, which stands for 'Adaptive Moment Estimation'. It is an implementation of the stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments.

The algorithm maintains two moving averages of gradients: The mean (first moment) and the variance (second moment). These moving averages are used to adaptively adjust the learning rate of each parameter during the training phase.

The paper [16, Adam: A Method for Stochastic Optimization] by Kingma et al. gives a detailed overview of the update mechanism of this algorithm. In short, the update rule for Adam can be summarized as follows:

- Initialize parameters such as the model parameters (weights and biases), the learning rate, decay rate of first and second moment)

- initialize the first and second moments
- For each iteration, compute gradient based on current batch of data (epoch) and update both the moments based on this.
- For each iteration, use the computed gradient and current values of moments to update the model parameters

Loss Function

As opposed to a classification task, this experiment is for the first time in this thesis demonstrating a regression task. Until now, we have made use of different probabilistic loss functions for all of the MNIST classification tasks. In deep learning, probabilistic loss functions are used when the output of the model is a probability distribution. The idea is to measure the difference between the predicted probability distributions and the true distribution of the target variable. Some examples of probabilistic loss functions, as we have already implemented are sparse cross entropy loss, binary cross entropy loss, and more [3, Goodfellow et al.].

In contrast, as explained in the fundamentals section of this thesis, the main idea of regression tasks is to predict a single value. For this, a different set of loss functions are utilized: so-called regression losses. The idea here is to measure the difference between the predicted values and the actual target values. This difference is then minimized by the optimizer implemented, thereby attempting to reach a minimum of the function. One commonly implemented example of such a loss function is the **Mean Squared Error**, or **MSE**, which has also been implemented in this example.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

- MSE builds the average of the squared differences between the predicted value$\hat{y}_i$ and the actual target value y_i
- By squaring the differences, larger errors are penalized more heavily

In the MNIST example, we used the training accuracy as well as the prediction accuracy as the metrics for evaluation purposes. For regression tasks, the mean squared error and/or the mean absolute error offer equivalent opportunities to evaluate the performance of the model. The mean squared error has been described above. The **mean absolute error**, **MAE** is defined as follows:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

This builds the average of the absolute difference between the predicted values $\hat{y}_i$ and the actual values y_i. In comparison to MSE, MAE treats all errors, large or small, equally. Therefore, the impact of outliers on the loss function is minimized. This allows us to use the MAE as a metric to evaluate, how well the model has performed in the prediction stage. Of course, one idea would also be to test the fitting of the model using the MAE as the loss function instead of the MSE, hence punishing outlier datapoints less than the current implementation with the MSE loss function would do.

In this current state, we have implemented the model with the MSE error as our loss function to be optimized using the ADAM optimizer, and the MAE error as our metric to evaluate the model performance. The implementation of the loss function along with the model compilation is as follows:

```
model.compile(loss='mse', optimizer='adam', metrics = ["
mean_absolute_error"])
```

Listing 6.17: Sequential Regression: Loss Function and Model Compilation

The next step is to train the model on the training dataset. To keep the training process the same over the upcoming variations of this experiment, following training attributes have been decided upon:

- **Epoch size**: 150, with no early stopping
- **Batch size**: 32, this is also the default value for gradient descent as recommended by Keras
- **Verbose**: 2, this simply means that each iteration shall be outputted during the training phase on an individual, single, line.

```
model.fit(X_train, y_train, epochs=150, batch_size=32, verbose=2)
```

Listing 6.18: Sequential Regression: Training Model

```
Epoch 1/150
88/88 - 1s - loss: 57.2522 - mean_absolute_error: 6.9538
- 674ms/epoch - 8ms/step
Epoch 2/150
88/88 - 0s - loss: 20.8046 - mean_absolute_error: 3.6703
- 278ms/epoch - 3ms/step
...
...
Epoch 149/150
88/88 - 0s - loss: 4.9994 - mean_absolute_error: 1.6172
- 282ms/epoch - 3ms/step
Epoch 150/150
88/88 - 0s - loss: 4.9852 - mean_absolute_error: 1.6246
- 275ms/epoch - 3ms/step
```

As can be seen in the output snippet (for full output please see Appendix), both of the losses have gradually decreased with increasing iterations.

The trained model can now be tested using the predict function on the test dataset, and be evaluated as per suitable metrics.

```
yhat = model.predict(X_test)
ma_error = mean_absolute_error(y_test, yhat)
ms_error = mean_squared_error(y_test, yhat)
print('MAE: %.3f' % ma_error)
print('MSE: %.3f' % ms_error)
```

Listing 6.19: Sequential Regression: Predicting values using testing data

```
44/44 [==============================] - 0s 2ms/step
MAE: 1.630
MSE: 5.060
```

This shows, that the result is accurate to an error of approx. 1.6 (rings).

6.2.2 Classification Task

Having solved the problem using regression, we now attempt to do the same using classification. A lot of the methodology will be kept the same, largely to ensure that both the models can be replicated in the next chapter in the functional approach.

The entire code of the notebook has once again been provided in the Appendix. This chapter shall focus on the idiosyncrasies of attempting to solve this problem within a classification framework.

The base of the model shall once again be a deep Feed Forward Network, as in the previous section, with some adaptations. The reader is reminded, the goal of this study is not to create the best neural network to solve this issue - rather to provide further proves of the capabilities of the sequential vs. functional architectures.

Up to this point, the Abalone dataset was described from the point of view of a regression problem - the previous study attempts to predict the number of rings as a continuous variable, based on the relationships existing between the various attributes of the dataset. To solve this via classification however, the target parameter 'Number of Rings' needs to be viewed as a set of different classes rather than a natural target that can be predicted. The prediction shall be probabilistic, as is usual in classification problems, and not target-based.

To this end, we re-frame the study performed above as a classification problem **with defining each unique ring number as a class**. One way to achieve this is to assign an integer to each unique class, starting at 0 and ending at *number of classes - 1*. This can be done with the help of a so called **Label Encoder**, as provided by the Scikit Learn Library. For further information on the method, the reader is referred to the Citation [15, sklearn.preprocessing.LabelEncoder].

Having imported the data as described in the Listing 6.13, we use this *Label Encoder* to define and therefore assign the classes:

```
from sklearn.preprocessing import LabelEncoder

# encode strings to integer
y = LabelEncoder().fit_transform(y)
print('y = ', y)
# total number of classes
n_class = len(unique(y))
print('n_class = ', n_class)
```

Listing 6.20: Classify target data using label encoder

```
y =  [14  6  8 ...  8  9 11]
n_class =  28
```

The code above shows that there are 28 distinct classes that each data-point can be classified into.

Define the Model

Once the data has been pre-processed and split into the training and testing datasets, as described previously for example in Listing 6.14, we continue with defining the model.

The fundamental architecture of the model has been kept identical, as shown in Figure 6.5, apart from the output layer - in this case of course, the final output layer has to be modified to consider the number of classes. This modified model has been visualized below in Figure 6.7.

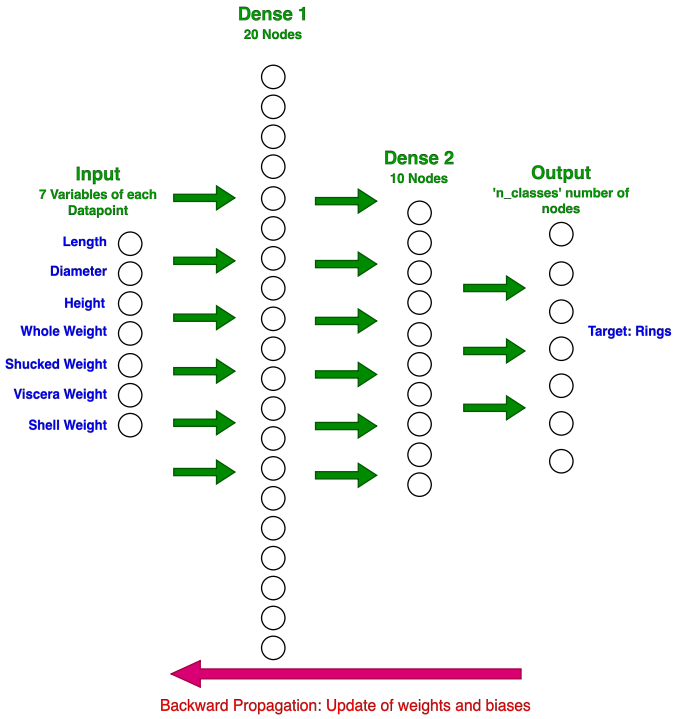

Figure 6.7: Abalone Sequential: Architecture schematic, Classification

Listing 6.21 shows the implementation of the shown architecture:

```
1     model = Sequential()
2     model.add(Dense(20, input_dim=number_features, activation='relu',
      kernel_initializer='he_normal'))
3     model.add(Dense(10, activation='relu', kernel_initializer='he_normal'))
4     # note, the last dense layer now has an output = number of classes and
      uses a softmax activation
5     model.add(Dense(n_class, activation='softmax'))
```

Listing 6.21: Sequential Classification: Defining the model

A few notes can be made here:

- The first two dense layers of the model, after the input layer, have been deliberately kept identical
- The number of nodes of the last layer, i.e. the output layer, is equal to 'n_class', as identified above by the Label Encoder (See Listing 6.14)
- As is common for multi-class optimization problems, we use a Softmax activation function, as also implemented in the multi-class MNIST problem (identification of digits) earlier in this thesis

As shown in the previous example, the model.summary() shall be used to showcase the implementation of the model architecture and to compute the number of parameters.

```
model.summary()
```

Listing 6.22: Sequential Classification: Model Summary

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 20)                160

 dense_1 (Dense)             (None, 10)                210

 dense_2 (Dense)             (None, 28)                308

=================================================================
Total params: 678 (2.65 KB)
Trainable params: 678 (2.65 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Figure 6.8 shows a visualisation of the dimensions of the neural network. The number of dimensions, as shown earlier, can be computed quite simply - we once again make use of equation 6.2 for the computations.

Therefore, the total number of parameters are as follows, taking up **2.65 kB** of space in memory:

$$\begin{aligned} Total\ Parameters &= \underbrace{160}_{\text{Dense Layer}} + \overbrace{210}^{\text{Dense Layer}} + \underbrace{308}_{\text{Output Layer}} \\ &= 678 \end{aligned}$$

An obvious observation can be made here. Solving the same problem statement as a classification rather than a regression causes an increase in the number of parameters by a factor of almost two. One could raise the question, whether a classification is more appropriate than regression for this dataset, i.e. whether the performance of the model is better with regard to accuracy. We check this by now compiling and training the model.

Compiling and Training the Model

The compilation step is slightly different than with the regression problem, since we cannot use the regression losses here. For classification, we make use of the cross entropy losses, as also done during the MNIST examples. The idea here is to minimize the spare categorical cross-entropy loss, which is well suited for multi-class classification tasks with class labels encoded as integers. Apart from the loss function, the implementation is equivalent (ADAM optimizer).

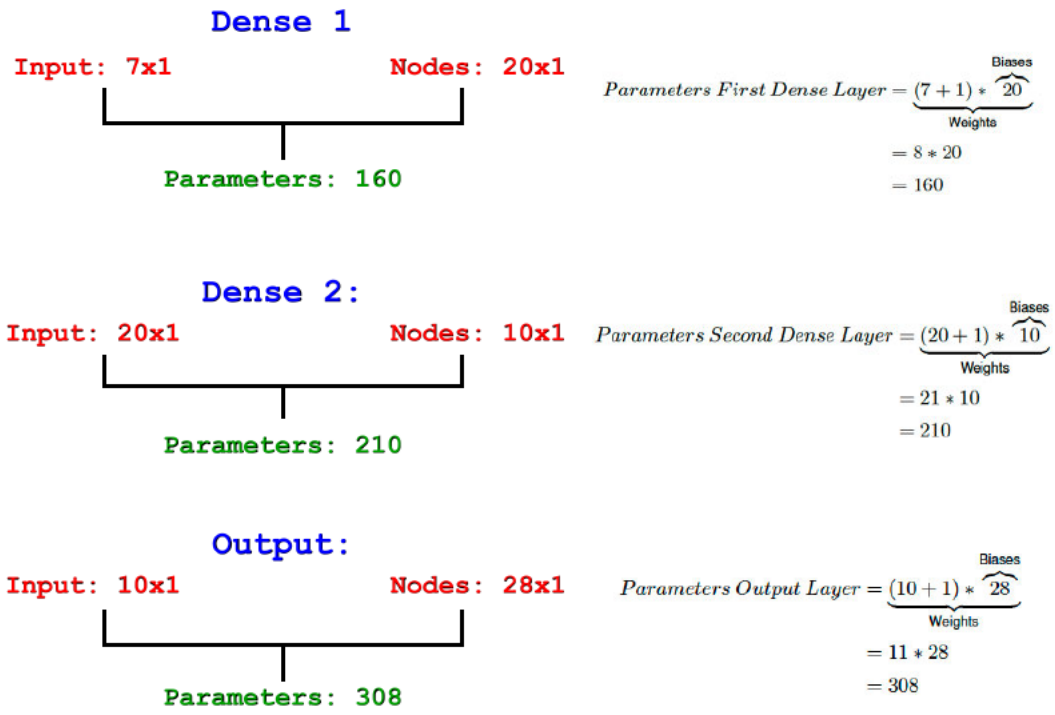

Figure 6.8: Abalone Sequential: Classification Model Dimensions

```
model.compile(loss='sparse_categorical_crossentropy', optimizer='adam')
```

Listing 6.23: Sequential Classification: Loss Function and Model Compilation

The next step is to train the model using the previously defined training dataset. The parameters defined above in the regression task have been kept identical, so that we also get the opportunity to make a fair comparison of the performance of the regression vs. classification tasks - whilst this is not the main idea behind this thesis, the opportunity to analyse this aspect is offered here.

```
model.fit(X_train, y_train, epochs=150, batch_size=32, verbose=2)
```

Listing 6.24: Sequential Classification: Training Model

```
Epoch 1/150
118/118 - 1s - loss: 3.1270 - 879ms/epoch - 7ms/step
Epoch 2/150
118/118 - 0s - loss: 2.6433 - 437ms/epoch - 4ms/step
...
...
Epoch 149/150
118/118 - 0s - loss: 1.9466 - 438ms/epoch - 4ms/step
Epoch 150/150
118/118 - 0s - loss: 1.9445 - 427ms/epoch - 4ms/step
```

As visible in the output snippet (for full output please see Appendix), the loss has been reduced over the iterations.

We now move on to the evaluation step. The trained model can now be applied onto the Testing Dataset to predict the number of rings. To determine the accuracy of the model, we must introduce a new metric, the so-called **accuracy score**.

```
# evaluate on test set
yhat = model.predict(X_test)
yhat = argmax(yhat, axis=-1).astype('int')
acc = accuracy_score(y_test, yhat,normalize=False)
print('Number of test datapoints: ', len(y_test))
print('Accuracy: %.3f' % acc)
```

Listing 6.25: Sequential Classification: Predicting values using testing data

```
44/44 [==============================] - 0s 2ms/step
Number of test datapoints:  1379
Accuracy: 368.000
```

The accuracy score is a method from the Scikit Learn library (see [15]) and describes the matching of the two parameters to one another. I.e. in the code above, we check how many of the predicted yhat values match the labels of the testing dataset (y_test). The attribute 'normalize=False' gives an absolute evaluation, whereas a 'normalize=True' would give a relative one. This shows that out of the 1379 data points of the testing dataset, only 368 have been classified correctly, giving an accuracy of approximately 27%.

Whilst the accuracy is not great, one must keep the simplicity of the network implemented in mind. The accuracy achieved here is not comparable to the regression network, where such an accuracy cannot be calculated. There, we had a accuracy of approximately ± 1.5 Rings. Both of the models definitely don't represent the state of the art, when it comes to solve this problem. However, the reader is reminded that the idea behind this experiment is not to solve this problem in the most optimal manner, rather to use the uniqueness provided by this dataset to be presented as a regression and a classification task and to implement both in the functional architecture - hence proving that it is possible to solve two very different kind of problems within one network.

6.2.3 Sequential Implementation

Before moving to the functional approach however, the author would like to point out a few aspects of the sequential approach. Up to this point, we have solved the abalone problem using two different methods individually and not actually sequentially. Therefore, we now proceed to solve the problem sequentially, i.e one after the other.

The code shall not be repeated, since the idea is really to train and run the models one after the other in a sequence. However, this task shall provide the reader with a reference to compare the functional approach with, since both the methods shall be implemented within one network then.

A sequential implementation of both regression and classification means running the network as shown in Figure 6.9.

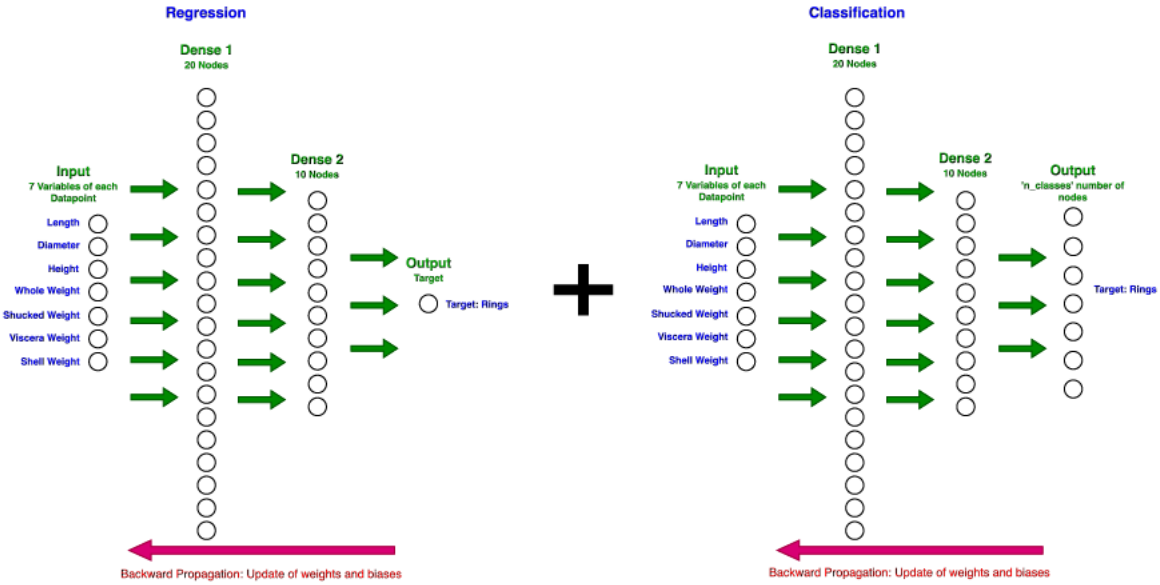

Figure 6.9: Abalone Sequential: Regression and Classification

The total number of parameters then becomes as follows:

$$\begin{aligned} Total\ Parameters &= \underbrace{(160 + 210 + 11)}_{\text{Regression Task}} + \overbrace{(160 + 210 + 308)}^{\text{Classification Task}} \\ &= 381 + 678 \\ &= 1059 \end{aligned}$$

And the total amount of space taken in the memory from the parameters is as follows:

$$\begin{aligned} Total\ Space\ in\ Memory &= \underbrace{(1.49)}_{\text{Regression Task}} + \overbrace{(2.65)}^{\text{Classification Task}} \\ &= 4.14\ kB \end{aligned}$$

This visualization and the calculation of number of parameters makes one disadvantage of the sequential implementation very clear: the doubling of the initial layers, which are actually identical. These layers are unnecessarily modelled, compiled, **trained** and run separately, one after the other. In this example, the disadvantage does not represent a large loss of time in terms of resources used, simply due to the small nature of the dataset. The reader can however imagine, how this problem would scale with larger datasets. A small peek into this behaviour was displayed with the MNIST example.

This visualization provides us the perfect reference to continue this journey and implement both of the tasks described above within the Keras functional framework.

6.2.4 Functional Implementation

Having solved the problem using regression and classification individually, as well as sequentially, we now focus on optimizing the problem statement and implementing it via the functional architecture.

The code has been implemented in Jupyter Notebooks, and has been provided in the Appendix. Here, once again, the particularities of this experiment shall be portrayed in an attempt to show the advantages and disadvantages of this approach.

In order to ensure comparability with the previous experiment, most of the structure and code is identical, especially the trivialities such as libraries required or downloading the dataset. The code shall not be repeated here.

Since we are combining classification and regression into one network, two sets of output datasets are needed. To be precise, the *y component* of the dataset, i.e. the target values first of all are duplicated. The duplicated version is named y_class, to differentiate it from the original y-component. This y_class component of the target dataset is then defined as the target data for the classification problem, using the label encoder as explained in the Section 6.2.2. The original y component of the dataset is then used as the target for the regression task.

As in the individual implementations, the data is then split into the respective training and testing datasets, although this time around there are three components instead of just two (X, y, y_class).

```
    # split data into train and test sets
    X_train, X_test, y_train, y_test, y_train_class, y_test_class =
    train_test_split(X, y, y_class, test_size=0.33, random_state=1)
    print('Size of X_train is ', len(X_train))
    print('Size of X_test is ', len(X_test))
    print('Size of y_train is ', len(y_train))
    print('Size of y_test is ', len(y_test))
    print('Size of y_train_class is ', len(y_train_class))
    print('Size of y_test_class is ', len(y_test_class))
```

Listing 6.26: Split 3 data componenets into training and testing datasets

```
Size of X_train is  2798
Size of X_test is  1379
Size of y_train is  2798
Size of y_test is  1379
Size of y_train_class is  2798
Size of y_test_class is  1379
```

As is shown in the output, the entire dataset of 4177 data points has been divided into a training and testing dataset, with 33% of the data points being reserved for testing purposes. In addition, the duplication of the target values has been shown here as well.

We now proceed with defining the model.

Defining the Model

The model definition has been kept as similar to the individual components as possible, so that a fair comparison can be made. The only difference that is presented with this approach is with the architecture. The attributes of each layer, however, have been kept identical. Basic architecture is as follows:

- Begin with the input data
- Use two dense layers to perform the computations (feed forward and backward propagation)
- Use the second hidden, dense layer as the input to two output layers, one for each task statement

This process has been visualized in Figure 6.10

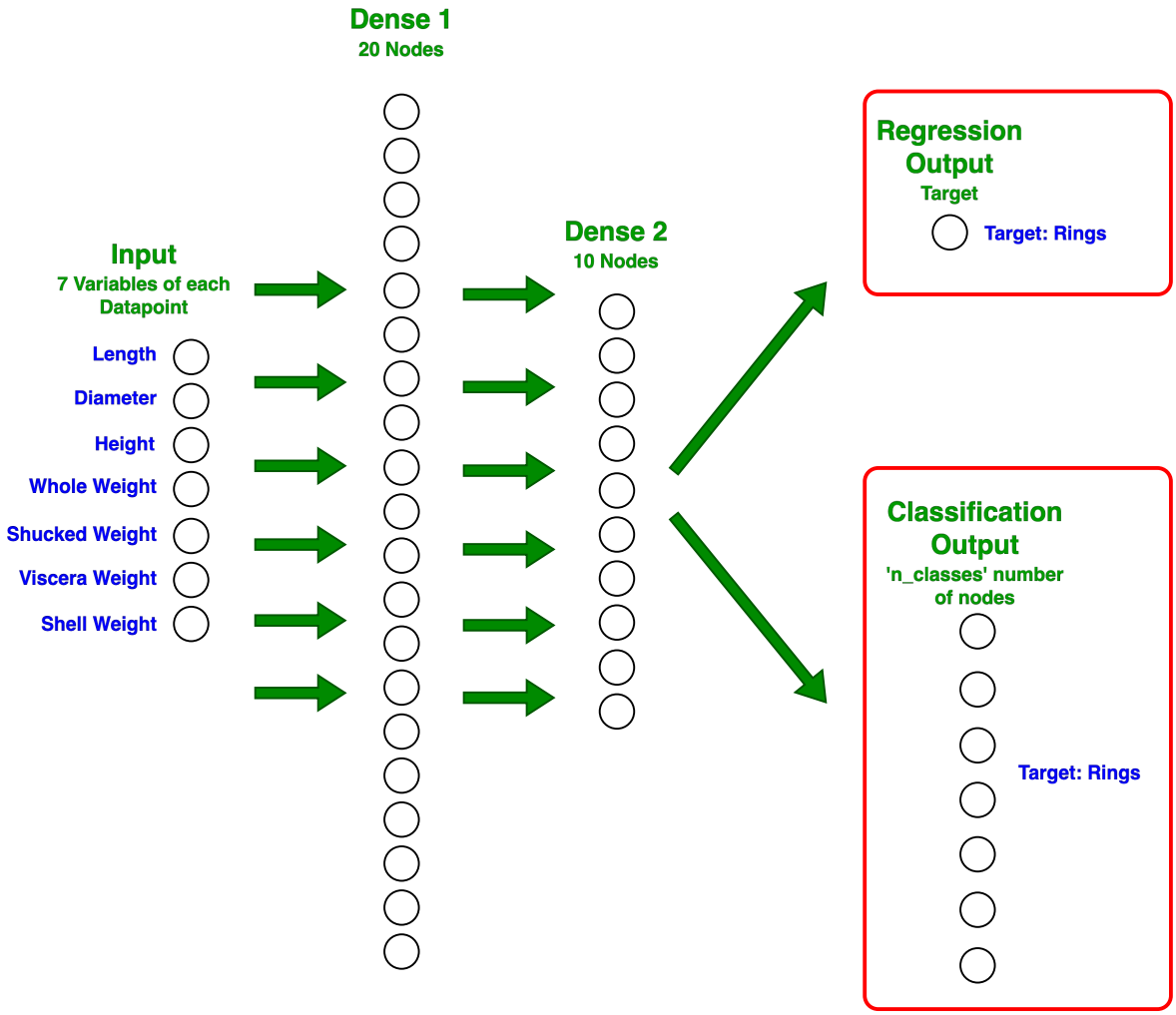

Figure 6.10: Abalone Functional Architecture

The Listing 6.27 shows the actual implementation.

```
1     visible = Input(shape=(number_features,))
2     hidden1 = Dense(20, activation='relu', kernel_initializer='he_normal')(
      visible)
3     hidden2 = Dense(10, activation='relu', kernel_initializer='he_normal')(
      hidden1)
4
```

```
5     # regression output: layer with a single node and a linear activation
      function
6     out_reg = Dense(1, activation='linear')(hidden2)
7
8     # classification output: layer with number of nodes = number of classes
      and a softmax activation function
9     out_class = Dense(n_class, activation='softmax')(hidden2)
```

Listing 6.27: Functional Implementation: Defining the Model

An the model summary can now be viewed as well:

```
1 model.summary()
```

Listing 6.28: Functional Implementation: Model Summary

```
Model: "model"
    _________________________________________________________________
     Layer (type)           Output Shape    Param #   Connected to
    =================================================================
     input_1 (InputLayer)   [(None, 7)]     0         []

     dense (Dense)          (None, 20)      160       ['input_1[0][0]']

     dense_1 (Dense)        (None, 10)      210       ['dense[0][0]']

     dense_2 (Dense)        (None, 1)       11        ['dense_1[0][0]']

     dense_3 (Dense)        (None, 28)      308       ['dense_1[0][0]']

    =================================================================
    Total params: 689 (2.69 KB)
    Trainable params: 689 (2.69 KB)
    Non-trainable params: 0 (0.00 Byte)
    _________________________________________________________________
```

The reader may be familiar with the attributes shown in the output from the MNIST example as well. In any case, here is a short description of the implementation:

- In Listing 6.27, one can see that each layer is given an extra attribute at the end. This is the previous layer from the perspective of the current layer, which the current layer is connected to
- As opposed to the sequential implementation, the output shows the extra attribute 'Connected to'. This gives us the exact information as to which layer is connected to which, once again showcasing the flexibility offered by this functional approach.

The total number of parameters may be determined via dimensional analysis, and is simply a composition of both of the individual implementations. The largest advantage of this entire exercise is shown via this analysis: the functional approach allows us to decide which layers remain identical between two or more different kinds of problems. These layers are then not repeated or duplicated,

instead they are 'shared' between each problem. For sake of simplicity, the same template from the previous implementations for the computation of the number of parameters has been used here, as shown in Figure 6.11.

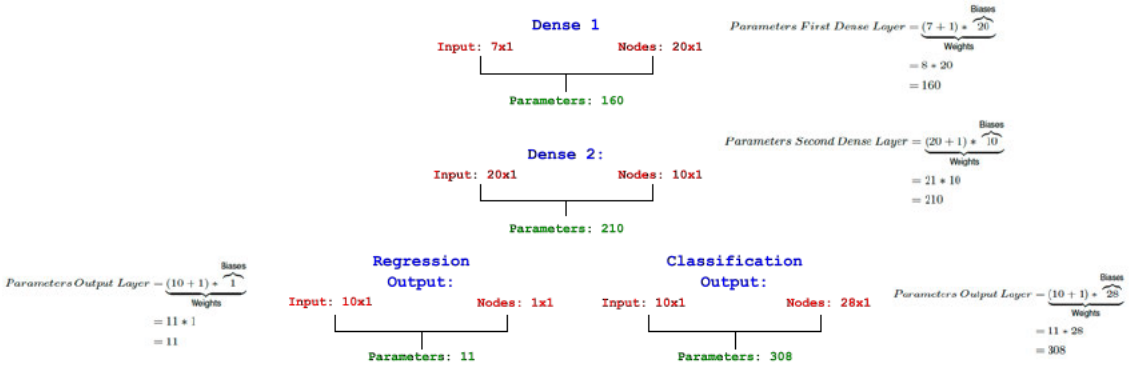

Figure 6.11: Abalone Functional Architecture Dimensions

It should be noted, as opposed to the sequential implementation, the parameters only take 2.69kB of space in memory - 1.45kB or 35% less space.

Compiling and Training the Model

Compiling the models is similar to the previous sections, only difference is that care has to be taken to define both of the loss functions: **MSE** for the regression output and **sparse categorical cross-entropy** for the classification output. At the same time, we also define the metric used at least for the regression layer, the **MAE**.

The training of the model shall also be done in a similar fashion. As before, for the sake of comparisons, all of the attributes have been kept the same. Only particularity here is to define two targets, which will be used to optimize towards.

```
1    model.compile(loss=['mse','sparse_categorical_crossentropy'], optimizer='
     adam', metrics = ["mean_absolute_error"])
2
3    model.fit(X_train, [y_train,y_train_class], epochs=150, batch_size=32,
     verbose=2)
```

Listing 6.29: Functional Implementation: Compiling and Training Model

```
Epoch 1/150
88/88 - 2s - loss: 93.3056 - dense_2_loss: 90.0292 - dense_3_loss: 3.2764
- dense_2_mean_absolute_error: 8.9144
- dense_3_mean_absolute_error: 8.8925 - 2s/epoch - 25ms/step
Epoch 2/150
88/88 - 1s - loss: 32.8813 - dense_2_loss: 30.0461 - dense_3_loss: 2.8351
- dense_2_mean_absolute_error: 4.6822
- dense_3_mean_absolute_error: 8.8925 - 505ms/epoch - 6ms/step

...
```

```
...
...

Epoch 149/150
88/88 - 0s - loss: 7.0324 - dense_2_loss: 5.0317 - dense_3_loss: 2.0007
- dense_2_mean_absolute_error: 1.6334
- dense_3_mean_absolute_error: 8.8925 - 486ms/epoch - 6ms/step
Epoch 150/150
88/88 - 0s - loss: 7.0065 - dense_2_loss: 5.0068 - dense_3_loss: 1.9997
- dense_2_mean_absolute_error: 1.6223
- dense_3_mean_absolute_error: 8.8925 - 486ms/epoch - 6ms/step
```

One interesting aspect can be noted here. The metric **MAE**, the Mean Absolute Error, is as previously explained a typical metric for regression problems. Here however, it is implemented on both of the output layers ('dense_2' and 'dense_3'). The output shown above confirms this - for the 'dense_2', the regression output, the mean absolute error reduces as expected to approximately 1.6, a similar value as was achieved with the sequential implementation. For the 'dense_3' layer however the mean absolute error remains constant. It is not an appropriate metric for the classification problem and is therefore not influenced here. We once again evaluate the classification problem using the **accuracy score**. First, the trained model is used to predict values using the testing data, and the accuracy is then evaluated.

```
1  yhat1, yhat2 = model.predict(X_test)
2
3  # calculate error for regression model: mean absolute square
4  error = mean_absolute_error(y_test, yhat1)
5  print('MAE: %.3f' % error)
6
7
8  # evaluate accuracy for classification model: argmax
9  yhat2 = argmax(yhat2, axis=-1).astype('int')
10 acc = accuracy_score(y_test_class, yhat2, normalize=False)
11 print('Number of test datapoints: ', len(y_test_class))
12 print('Accuracy: %.3f' % acc)
```

Listing 6.30: Functional Implementation: Predicting values using testing data

```
44/44 [==============================] - 0s 2ms/step
MAE: 1.622
Number of test datapoints:  1379
Accuracy: 396.000
```

This result shows that the final results of the functional model are more or less comparable to the sequential implementation. The regression has outputted a result with an accuracy of approximately ± 1.6 Rings and the classification has an accuracy of approx. 28.7% - very similar values to the initial, sequential implementation.

This experiment therefore clearly proves at least one thing: the functional implementation works at least as well as the sequential implementation, when it comes to model accuracies, even though the number of parameters to be optimized have been drastically reduced by almost 35%.

The author would like to reiterate the goal of this experiment: to prove that two inherently different kind of problems can be solved via a single implementation of a network. The goal was **not** to create the best neural network to solve the problem of determining the age of abalones. This has been done in various different studies, which can be viewed by the reader. The author would therefore urge the reader in the direction of these studies, if the reader is interested in implementations optimizing the accuracies.

7 Discussion and Outlook

The previous chapter consisted of two experiments which were conducted during the course of this thesis. Each experiment had its own goals and provided proofs of different aspects of the methodologies tested in this thesis. The author also tried to include discussions necessary to understand the train of thought behind the experiments together with the experiment descriptions, e.g. the dimension analysis that was done, the choice of the evaluation metrics, etc.

The task statement of the thesis can be used as a blue print to review the success of the choice and methodology of the experiments. As stated in Chapter 1, the **primary objective of this thesis is to investigate the feasibility of developing a unified neural network architecture capable of handling diverse problem types.**

The author has attempted to tackle this objective via the offerings of the Tensorflow Keras framework. As was described in the Chapter 4, 'State of the Art', the two methodologies offered by the Keras Framework **sequential** and **functional** offer a possible solution to this problem.

The first experiment showcases an implementation of a convolutional neural network, and attempts to solve two problem statements posed by the MNIST Dataset. The main ideas behind this experiment were to prove that the functional approach works as well as the sequential approach with image-based datasets, especially with CNN architectures. One downside, for the lack of a better word, is that both problems solved within the scope of this experiment are different classification problems. First problem was to identify the digits from the images, where as the other problem, devised by the author, was to classify the images into two bins, differentiated by a simple rule (larger or smaller than '5').

This *downside* led the author to devise the second experiment: Here, the 'Abalone' dataset was used to determine the age of abalones. The experiment offers proof that the functional architecture works well with text-based datasets as well. Due to the nature of the dataset, the architecture of the neural network was not convolutional, rather feed forward. The upside, however, of this experiment, is that this dataset can be 'tricked' to be used as a regression problem as well as a classification problem.

Therefore, the combination of both experiments have allowed us to provide a decent attempt at solving the original problem at hand: with the functional architecture, a neural network can be designed, with which two or more problems are solved within a singular, unified network. Depending on the dataset in question, the fundamental architecture of the neural network would be different (e.g. image based dataset favours CNNs).

The research objectives of this thesis, as referenced in Section 1, have also been attempted via the two experiments described above. With the MNIST dataset, the author has introduced a CNN implementation to solve two classification tasks. The abalone dataset on the other hand has allowed the author to combine regression with classification. In addition, for both experiments, detailed descriptions of how the training of the model(s) is performed has been provided, along with the optimization technique. Various metrics have been used, as deemed appropriate for the dataset and problem statement in question. Finally the question of efficiency has also been tackled via the

dimension analysis for each task - thereby showing how the number of parameters, and therefore the amount of space in memory, reduces with the functional approach in comparison to the sequential approach.

The author would however take this opportunity to discuss the last two points in further detail: comparing the functional to the sequential approach with regard to the model accuracies as well as the training efficiencies.

First of all, the author would like to introduce the metric 'accuracy score', as implemented in the abalone classification problem, also to the MNIST classification problems - a very rudimentary prediction accuracy was presented in Chapter 6.1.1, where simply a output comparison of the test labels as well as the prediction labels was displayed. The implementation of the 'accuracy score' will provides us with a quantifiable measure to compare the different models with one another.

In the ***sequential*** implementation, following accuracies were achieved for both of the classification tasks: identification of digits represented by the model Category and the binning of the digits in Left-handed or Right-handed by the model L/R. The reader may compare the Listings provided below (Listings 7.1 or 7.2) with the Listings from Chapter 6.1.1, e.g. Listing 6.6

```
1    # Use Accuracy Score to determine prediction accuracy CATEGORY
2    acc_cat = accuracy_score(y_test[0:numMNIST] , labels_cat , normalize =
     False)
3    print('Number of test datapoints :' , len(y_test[0:numMNIST]))
4    print('Prediction Accuracy: %.3f' % acc_cat)
```

Listing 7.1: Prediction Accuracy of MNIST Category Model

```
Number of test datapoints : 100
Prediction Accuracy: 99.000
```

```
1    # Use Accuracy Score to determine prediction accuracy L/R
2    test_labels_lr = np.array([1 if i >= 5 else 0 for i in y_test[0:numMNIST
     ]])
3    print('y_test = ', y_test[0:numMNIST])
4    print('Test = ', test_labels_lr)
5    print('Prediction = ', labels_lr)
6
7    acc_lr = accuracy_score(test_labels_lr , labels_lr , normalize = False)
8    print('Number of test datapoints :' , len(test_labels_lr))
9    print('Prediction Accuracy: %.3f' % acc_lr)
```

Listing 7.2: Prediction Accuracy of MNIST L/R Model

```
y_test =  [7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6
6 5 4 0 7 4 0 1 3 1 3 4 7 2 7 1 2 1 1 7 4 2 3 5 1 2 4
4 6 3 5 5 6 0 4 1 9 5 7 8 9 3 7 4 6 4 3 0 7 0 2 9 1 7
3 2 9 7 7 6 2 7 8 4 7 3 6 1 3 6 9 3 1 4 1 7 6 9]

Test =  [1 0 0 0 0 0 0 1 1 1 0 1 1 0 0 1 1 1 0 0 1 1
```

```
1 1 0 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 1 0 0
0 0 1 0 1 1 1 0 0 0 1 1 1 1 1 0 1 0 1 0 0 0 1 0 0 1
0 1 0 0 1 1 1 1 0 1 1 0 1 0 1 0 0 1 1 0 0 0 0 1 1 1]

Prediction =  [1 0 0 0 0 0 1 0 0 1 0 1 1 0 0 0 1 1
1 0 1 1 1 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 1 1 1 0 1 0 1 0 0 0 1
0 0 1 0 1 0 0 1 1 1 1 0 1 1 0 1 0 1 0 0 1 1 0 0 1 0 1 0 1]

Number of test datapoints : 100
PredictionAccuracy: 88.000
```

Similarly, the model accuracies may be computed via the accuracy score for the ***functional*** implementation as well. The code shown in the Listings above (Listings 7.1 and 7.2) would be identical for the functional implementation as well, therefore it shall not be repeated here. The outputs, however, differ slightly, due to the difference in the architectures.

Having now described the prediction accuracies across all of the models up until now in a unified manner, the author would like to use the opportunity to show a comparison.

Dataset		Neural Network				Prediction Accuracy	
Type	Name	Type	Methodology	Total Number of Parameters	Memory Occupied by Parameters	Problem 1	Problem 2
Image Based	MNIST	CNN	**Sequential**: Classification of digits THEN Binning into Left/Right	61461	240 kB	99%	88%
Image Based	MNIST	CNN	**Functional**: Classification of digits AND Binning into Left/Right	37611	147 kB	99%	93%
Text Based	Abalone	FFN	**Sequential**: Regression of No. Of Rings THEN Classification of No. Of Rings	1059	4.14 kB	MAE = 1.63 Rings	27%
Text Based	Abalone	FFN	**Functional**: Regression of No. Of Rings THEN Classification of No. Of Rings	689	2.69 kB	MAE = 1.62 Rings	29%

Figure 7.1: Overview of predicted accuracies

Based on the Figure 7.1, a few observations can be made:

- The sizes of both datasets are vastly different from each other. In addition, the implemented neural networks are of different depths, causing an order of magnitude of difference with regard to the number of parameters between each dataset. This however has no influence on the comparison we want to draw in this thesis, as long as comparison between both of the methodologies can be made
- Both experiments each solve two problems. 'Problem 1' and 'Problem 2' in the last two coloumns refer to these problems
- However, one similarity may be extracted here: the functional approach is always more economical than the sequential approach with respect to the number of parameters or the amount of memory used: in both cases, a reduction of more than 35% of parameters can be observed

- The prediction accuracies are similar between both of the approaches - at first glance, the functional approach seems to be slightly better when it comes to 'Problem 2'. However, the author would like to remind the reader of the stochastic nature of the problems, which could theoretically cause the optimization problems to get 'stuck', for a lack of a better word, in a local minima. This in turn could cause better results to occur than expected. The author expects, over a statistically significant number of training runs, the accuracies would most probably be quite similar, regardless of the approach (sequential or functional) chosen.
- The statement made above can be deduced quite confidently, since the base architecture of the deep, hidden layers in each neural network, along with the hyperparameters which are associated with those layers, have been kept identical between both of the approaches. The only difference between both of the approaches, which is also the reason for the increase in efficiency pointed out above, is that the functional approach utilizes/shares base hidden layers for both of the problems, whereas base hidden layers are repeated in the sequential approach.

With the analysis above, it is safe to say that the functional approach offered by the Keras Framework offers a viable, technically feasible and economical approach to the question, whether two different kind of problems can be solved with a unified neural network.

Until this point, we have focused on the final accuracy of the model. A further possibility to prove the feasibility of the functional approach is to consider and investigate the training step of the exercise in further detail.

The author has attempted to showcase this in the most pragmatic manner as possible. One idea would be, to measure the time taken for the training step for each implementation. In Chapter 6, the details of the hardware used for the exercises was also specified. This hardware was not modified during the course of this thesis, and can be used as a baseline for a comparison.

To measure the time taken for the training step, the time.time library publicly available in Python shall be utilized. Here is a simple idea for an implementation of this library to measure execution time of the training step:

```
import time

start = time.time()
.....................
.....................
### TRAINIG STEP OF NEURAL NETWORK ###
.....................
.....................
end = time.time()

print('Time taken = ', end - start)
```

Listing 7.3: Measuring execution Time of Training Step

The pseudocode shown above was implemented for each problem. The implementation for each experiment is identical to the pseudocode shown in the Listing 7.3 and shall not be repeated here.

Dataset		Neural Network				Execution Time (s)
Type	Name	Type	Methodology	Total Number of Parameters	Memory Occupied by Parameters	
Image Based	MNIST	CNN	**Sequential**: Classification of digits THEN Binning into Left/Right	61461	240 kB	272
Image Based	MNIST	CNN	**Functional**: Classification of digits AND Binning into Left/Right	37611	147 kB	139
Text Based	Abalone	FFN	**Sequential**: Regression of No. Of Rings THEN Classification of No. Of Rings	1059	4.14 kB	99
Text Based	Abalone	FFN	**Functional**: Regression of No. Of Rings THEN Classification of No. Of Rings	689	2.69 kB	67

Figure 7.2: Comparison of Training Times

The Figure 7.2 shows a comparison of the training time taken for each experiment. This comparison further proves that the functional approach is more efficient than the sequential approach. Due to the stochastic nature of the problems, it is difficult to quantify this efficiency. The author would take caution in claiming a gain of efficiency by at least 30% - the times reported here are simply those computed during single runs and do not represent a statistically significant average value. Nevertheless, a tendency can definitely be drawn from the data provided.

This brings us to the conclusion of the thesis. Having presented the current state of the art in Chapter 4 with regard to the possibilities of the technologies available, the core of this thesis has been an attempt to solve the underlying problem presented during the motivational section in the beginning of this thesis.

The motivation of this work stemmed from Mr. Schreiber's research on implementing two neural networks on a NVIDIA Jetson Nano Jetracer. Due to the limited capacity of the hardware available, his work posed the question whether such problems can be solved more efficiently.

The research performed here has attempted to answer this question, albeit with generic datasets. The problem was divided into two parts and tackled separately: the MNIST dataset allowed the author to prove that a CNN can be successfully used within a functional architecture to solve two sets of problems. The Abalone dataset, on the other hand, provided the author the opportunity to combine a regression problem with a classification problem. An added advantage was the added diversification with regard to the type of the dataset - it is now clear that the functional approach may be implemented regardless of the type of the data or the type of the neural network required - the basic idea may be used independently of such limitations.

In addition, the experiments clearly show that functional approach is at least as good as the standard sequential approach of running two networks one after the other. The experiments have been thoroughly designed via dimensional analysis, which provides a detailed insight into the resources required and hence used by the implemented neural networks. The results are then analysed using appropriate metrics, which provide information as to the success rate, i.e. prediction accuracies, of each approach.

One of the largest problems faced during this thesis was to find generic datasets which could be tweaked to be solved as regression and/or classification. In addition, the author also wanted to make sure that the methodologies discussed were independent of the type of neural network implemented.

One should however mention that the scope of this thesis was not to create the best solutions for each set of problems. Whilst the prediction accuracies obtained with the MNIST dataset are quite high, enough studies may be found online which would provide further improvements. At the same time, the abalone dataset was one of the only ones found by the author where the problem statement could be tweaked to be solved as a regression problem as well as a classification problem. Many further resources are available where the basic problem of determining the age of the abalones has been implemented with a much higher accuracy. This was not the focus here, and simplicity of the feed forward network implemented in this thesis is a proof of that. Be that as it may, the dataset proved valuable for the comparison it was intended for.

This chapter has provided a summary of the research performed and cross referenced the results and analysis to the objectives described in the initial stages. One can confidentially claim that the objectives of this thesis have been met. Nonetheless, as always, there are questions left open ended which could not be attempted during the course of this thesis. The author would like to present these as possible paths for future works.

The main one would be to attempt the findings of this thesis on a real-world dataset - perhaps the dataset created by Mr. Schreiber during his research would provided a good starting point. This thesis has already proven that the functional approach provides a good solution to solve different problems within a unified network - it may be an interesting exercise to test this statement on a dataset which comprises of a higher number of dimensions. Although the process would be similar to the MNIST experiment performed here, care would have to be taken in the pre-processing of the data and defining the problems to be solved clearly.

Another improvement which can be done, from the point of view of the prediction accuracies, would be to repeat the experiments conducted with the abalone dataset. The FFN implemented during this thesis was quite rudimentary, and provided a very low accuracy at the end. This can be solved by tuning the hyperparameters of the deep layers (e.g. number of nodes, the learning rate of the optimizer, etc.). Although the technique described here with the functional approach would remain the same, the author is confident that some tweaking of the parameters could help improve the accuracy obtained by the neural network.

Appendix A: MNIST Dataset Experiments

A.1 Sequential Implementation Of CNN using MNIST Dataset

KERAS Sequential API

Implementing a simple Convolutional Neural Network to predict MNIST Dataset using the Keras Sequential API

Import all relevant libraries

```
# Import Tensorflow Libraries
import tensorflow as tf
import tensorflow_datasets as tfds
import keras

# Import further necessary libraries
import numpy as np
import matplotlib.pyplot as plt

# Import specific Keras libraries
from keras.datasets import mnist
from keras.models import Model
from keras.layers import Dense, Input
from keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten
from keras import backend
```

Listing A.1: MNIST Sequential, Imported Libraries

Load the MNIST Dataset

```
# load MNIST Dataset directly from Keras/Tensorflow
mnist = keras.datasets.mnist

# Split the data into a training set and testing set
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# normalize the dataset
x_train, x_test = x_train / 255.0, x_test / 255.0

# 0=left, 1=right
# rule: if number <=5 --> category: left, if number >5 --> category right

y_leftright = np.zeros(y_train.shape, dtype=np.uint8)
for idx, y in enumerate(y_train):
    if y > 5:
        y_leftright[idx] = 1

# Display the first twenty digits from the training dataset, to make sure
 the dataset is sensible
print(y_train.dtype, y_train[0:20])
```

```
20      print(y_leftright.dtype, y_leftright[0:20])
21
22      # Create a dictionary out of the output labels, with two keys --> in
        order to solve two distinct problems
23      y = {"category_output": y_train,
24          "leftright_output": y_leftright }
```

Listing A.2: MNIST Sequential, Loading Dataset from Keras

```
uint8 [5 0 4 1 9 2 1 3 1 4 3 5 3 6 1 7 2 8 6 9]
uint8 [0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1]
```

Creating the Models

The following steps are described in this section:

- Define the architecture of the model
- Compile the model
- Train the model using model.fit method

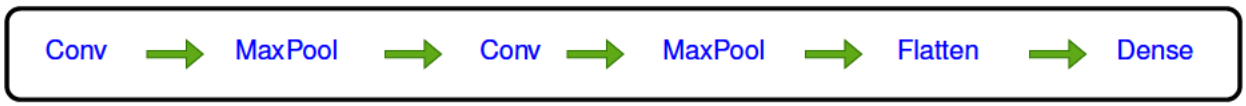

Figure A.1: MNIST Sequential: Architecture as implemented

As described in the introduction to this experiment in Chapter 6.1, the aim of this experiment is to solve two problems using the same dataset. The problems can be summarised as follows:

1. **Identification**: Identify the digit from the image
2. **Classification**: Classify each identifed digit in one of the following bins: Left handend (if less than 5) or right handed (if greater than or equal to 5)

We begin by first defining the model for the identification of the numbers, called modelCategory:

```
1       modelCategory = tf.keras.Sequential([
2           tf.keras.layers.Conv2D(50, 5, activation='relu', input_shape
        =(28,28,1)),
3
4           # Conv is the convolutional layer, 2D because we have a greascale
        image, i.e. it is 2D
5           # hyperparamters: number of kernels, size of kernels, activation
        function,
6           # input shape (images are of size 28x28 and greyscale therefore 1)
7           tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
8
9           # input size after the max pooling layer will be half of 28x28 and
        there will be 50 images, since we have 50 kernels in the first
        convolutional layer. However, the input_size parameter is only needed in
        the first convolutional layer of the model
10          tf.keras.layers.Conv2D(50, 3, activation='relu'),
11          tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
12          tf.keras.layers.Flatten(),
13          tf.keras.layers.Dense(10)
```

```
        ])

        # 10 outputs, further paramters are not needed
        # Choice of activation function: Softmax --> will be defined further
    along in the code
```

Listing A.3: MNIST Sequential, Define Model 'Category'

```
2023-07-19 23:54:45.110544: I metal_plugin/src/device/metal_device.cc:1154]
Metal device set to: Apple M2 Pro
2023-07-19 23:54:45.110568: I metal_plugin/src/device/metal_device.cc:296]
systemMemory: 16.00 GB
2023-07-19 23:54:45.110574: I metal_plugin/src/device/metal_device.cc:313]
maxCacheSize: 5.33 GB
2023-07-19 23:54:45.110605: I
tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:303]
Could not identify NUMA node of platform GPU ID 0, defaulting to 0. Your kernel
may not have been built with NUMA support.
2023-07-19 23:54:45.110622: I
tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:269]
Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 0
MB memory) -> physical PluggableDevice (device: 0, name: METAL, pci bus id:
<undefined>)
```

The next step is to define the next model, to classify the identified images into the two predefined bins, called modelLR:

```
    modelLR = tf.keras.Sequential([
    tf.keras.layers.Conv2D(50, 5, activation='relu', input_shape=(28,28,1)),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
    tf.keras.layers.Conv2D(50, 3, activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(1)
    ])
    # 1 outputs, further paramters are not needed
    # Choice of activation function: Softmax --> will be defined further
    along in the code
```

Listing A.4: MNIST Sequential, Define Model 'LR'

The output of this block is omitted at this point, since it is identical to the block above.

We can now check whether each model has been defined as envisioned.
Here is a portrayal of the **model** modelCategory:

```
    modelCategory.summary()
```

Listing A.5: MNIST Sequential, Model 'Category' Summary

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 24, 24, 50)        1300
max_pooling2d (MaxPooling2D) (None, 12, 12, 50)        0
conv2d_1 (Conv2D)            (None, 10, 10, 50)        22550
max_pooling2d_1 (MaxPooling2D) (None, 5, 5, 50)        0
flatten (Flatten)            (None, 1250)              0
dense (Dense)                (None, 10)                12510
=================================================================
Total params: 36360 (142.03 KB)
Trainable params: 36360 (142.03 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

And here is a portrayal of the **model** modelLR:

```
1 modelLR.summary()
```

Listing A.6: MNIST Sequential, Model 'LR' Summary

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            (None, 24, 24, 50)        1300
max_pooling2d_2 (MaxPooling2D) (None, 12, 12, 50)      0
conv2d_3 (Conv2D)            (None, 10, 10, 50)        22550
max_pooling2d_3 (MaxPooling2D) (None, 5, 5, 50)        0
flatten_1 (Flatten)          (None, 1250)              0
dense_1 (Dense)              (None, 1)                 1251
=================================================================
Total params: 25101 (98.05 KB)
Trainable params: 25101 (98.05 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Loss Function

- For classification of digits from 0 to 9, a sparse categorical cross entropy loss function works well
- For classification of digits whether left handed or right handed, the binary cross entropy function would be suitable

```
lossCategory = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=
True)
# the parameter 'from_logits=True' tells tensorflow that the
probabilities are not normalized. In other words, the Softmax Activation
function has not been applied to produce a probability distribution. This
helps with numerical stability.
lossLR = keras.losses.BinaryCrossentropy(from_logits=False)
# binary cross entropy loss function for classification 0 / 1
```

Listing A.7: MNIST Sequential, Define Loss Functions

Now we need to compile the model. This will combine the model with the loss function and the optimizer, thereby preparing it for the training:

```
optim = keras.optimizers.legacy.Adam(learning_rate=0.001)
modelCategory.compile(optimizer=optim, loss=lossCategory, metrics = ['
accuracy'])
# metrics of model should be stored in a list called accuracy
modelLR.compile(optimizer=optim, loss=lossLR, metrics = ['accuracy'])
```

Listing A.8: MNIST Sequential, Compile both Models

Training the Models

First, we train the **model** modelCategory:

```
modelCategory.fit(x_train, y=y_train, epochs=15,
      batch_size=64, verbose=2)
```

Listing A.9: MNIST Sequential, Training Model 'Category'

```
Epoch 1/15
2024-02-25 21:48:40.532281: I tensorflow/core/grappler/optimizers/
custom_graph_optimizer_registry.cc:114]
Plugin optimizer for device_type GPU is enabled.
938/938 - 9s - loss: 0.1889 - accuracy: 0.9434 - 9s/epoch - 9ms/step
Epoch 2/15
938/938 - 8s - loss: 0.0556 - accuracy: 0.9827 - 8s/epoch - 9ms/step
Epoch 3/15
938/938 - 8s - loss: 0.0407 - accuracy: 0.9873 - 8s/epoch - 9ms/step
Epoch 4/15
938/938 - 8s - loss: 0.0320 - accuracy: 0.9900 - 8s/epoch - 9ms/step
Epoch 5/15
938/938 - 8s - loss: 0.0260 - accuracy: 0.9914 - 8s/epoch - 9ms/step
Epoch 6/15
938/938 - 8s - loss: 0.0220 - accuracy: 0.9929 - 8s/epoch - 9ms/step
Epoch 7/15
938/938 - 8s - loss: 0.0177 - accuracy: 0.9945 - 8s/epoch - 9ms/step
Epoch 8/15
938/938 - 8s - loss: 0.0145 - accuracy: 0.9953 - 8s/epoch - 9ms/step
```

```
Epoch 9/15
938/938 - 8s - loss: 0.0125 - accuracy: 0.9960 - 8s/epoch - 9ms/step
Epoch 10/15
938/938 - 8s - loss: 0.0106 - accuracy: 0.9965 - 8s/epoch - 9ms/step
Epoch 11/15
938/938 - 8s - loss: 0.0099 - accuracy: 0.9969 - 8s/epoch - 9ms/step
Epoch 12/15
938/938 - 8s - loss: 0.0075 - accuracy: 0.9974 - 8s/epoch - 9ms/step
Epoch 13/15
938/938 - 8s - loss: 0.0082 - accuracy: 0.9970 - 8s/epoch - 9ms/step
Epoch 14/15
938/938 - 8s - loss: 0.0063 - accuracy: 0.9979 - 8s/epoch - 9ms/step
Epoch 15/15
938/938 - 8s - loss: 0.0049 - accuracy: 0.9984 - 8s/epoch - 9ms/step
```

We then proceed with training the **model** modelLR:

```
1  modelLR.fit(x_train, y_leftright, epochs=15,
2          batch_size=64, verbose=2)
```

Listing A.10: MNIST Sequential, Training Model 'LR'

```
Epoch 1/15
2024-02-25 21:51:48.139571: I tensorflow/core/grappler/optimizers/
custom_graph_optimizer_registry.cc:114]
Plugin optimizer for device_type GPU is enabled.
938/938 - 9s - loss: 0.4787 - accuracy: 0.8732 - 9s/epoch - 10ms/step
Epoch 2/15
938/938 - 9s - loss: 0.2342 - accuracy: 0.9344 - 9s/epoch - 9ms/step
Epoch 3/15
938/938 - 9s - loss: 0.1814 - accuracy: 0.9499 - 9s/epoch - 9ms/step
Epoch 4/15
938/938 - 9s - loss: 0.1586 - accuracy: 0.9585 - 9s/epoch - 9ms/step
Epoch 5/15
938/938 - 9s - loss: 0.1647 - accuracy: 0.9555 - 9s/epoch - 9ms/step
Epoch 6/15
938/938 - 9s - loss: 0.1941 - accuracy: 0.9521 - 9s/epoch - 9ms/step
Epoch 7/15
938/938 - 9s - loss: 0.2533 - accuracy: 0.9489 - 9s/epoch - 9ms/step
Epoch 8/15
938/938 - 9s - loss: 0.3100 - accuracy: 0.9411 - 9s/epoch - 9ms/step
Epoch 9/15
938/938 - 9s - loss: 0.1390 - accuracy: 0.9634 - 9s/epoch - 9ms/step
Epoch 10/15
938/938 - 9s - loss: 0.1355 - accuracy: 0.9646 - 9s/epoch - 9ms/step
Epoch 11/15
938/938 - 9s - loss: 0.1959 - accuracy: 0.9567 - 9s/epoch - 9ms/step
```

```
Epoch 12/15
938/938 - 9s - loss: 0.1271 - accuracy: 0.9696 - 9s/epoch - 9ms/step
Epoch 13/15
938/938 - 9s - loss: 0.1255 - accuracy: 0.9670 - 9s/epoch - 9ms/step
Epoch 14/15
938/938 - 9s - loss: 0.2274 - accuracy: 0.9504 - 9s/epoch - 9ms/step
Epoch 15/15
938/938 - 9s - loss: 0.1189 - accuracy: 0.9688 - 9s/epoch - 9ms/step
Out[20]:
<keras.src.callbacks.History at 0x2fa7781f0>
```

Testing the Models

```
# list with predictions
predictionCategory = modelCategory.predict(x_test)
len(predictionCategory)

predictionLR = modelLR.predict(x_test)
len(predictionLR)


# lets observe how the model works for the first 100 MNIST images
numMNIST = 100

labels_cat = np.argmax(predictionCategory[0:numMNIST], axis=1)
labels_lr = np.array([1 if p >= 0.5 else 0 for p in predictionLR[0:
numMNIST]])


print('Test Labels: ', y_test[0:numMNIST])
print('Predicted Labels, Digit: ', labels_cat)
print('Predicted Labels, L/R: ', labels_lr)
```

Listing A.11: MNIST Sequential, Testing both Models

```
313/313 [==============================] - 1s 2ms/step
313/313 [==============================] - 1s 2ms/step
Test Labels:
[7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6 6 5 4 0 7 4 0 1 3 1 3 4 7 2 7
 1 2 1 1 7 4 2 3 5 1 2 4 4 6 3 5 5 6 0 4 1 9 5 7 8 9 3 7 4 6 4 3 0 7 0 2 9
 1 7 3 2 9 7 7 6 2 7 8 4 7 3 6 1 3 6 9 3 1 4 1 7 6 9]
Predicted Labels, Digit:
[7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 2 4 9 6 6 5 4 0 7 4 0 1 3 1 3 4 7 2 7
 1 2 1 1 7 4 2 3 5 1 2 4 4 6 3 5 5 6 0 4 1 9 5 7 8 9 3 7 4 6 4 3 0 7 0 2 9
 1 7 3 2 9 7 7 6 2 7 8 4 7 3 6 1 3 6 9 3 1 4 1 7 6 9]
Predicted Labels, L/R:
[1 0 0 0 0 0 0 1 0 1 0 1 1 0 0 0 1 1 0 0 1 1 1 0 0 0 1 0 0 0 0 0 0 0 1 0 1
 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 1 1 1 0 0 0 1 0 0 0 1 0 0 1
 0 1 0 0 1 1 1 1 0 1 1 0 1 0 1 0 0 1 1 0 0 0 0 1 1 1]
```

Performing a quick check to see how well the identification of the digits works, by computing the difference between the **test labels** and the **predicted labels** (ideally, all of them should be 0):

```
print('Test Labels - Predicted Labels: ',y_test[0:numMNIST]-labels_cat)
```

```
Test Labels - Predicted Labels:  [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0]
```

This shows, when the model is tested against the first 100 images in the MNIST dataset, it only makes one error. The same is true for the first 1000 images.

A.2 Functional Implementation Of CNN using MNIST Dataset

KERAS Functional API

This section is a follow-up of the sequential implementation of the Convolutional Neural Network to make predictions using the MNIST dataset.
The same dataset shall be implemented here, using the same problem statements, but using the functional architecture as provided by the MNIST dataset.

[Please Note: the first few blocks of code, e.g. importing the libraries or the MNIST Dataset shall be a repetition from Section A.1]. These are however necessary to run the Jupyter Notebook.

Import all relevant libraries

```
# Import Tensorflow Libraries
import tensorflow as tf
import tensorflow_datasets as tfds
import keras

# Import further necessary libraries
import numpy as np

import matplotlib.pyplot as plt

# Import specific Keras libraries
from keras.datasets import mnist
from keras.models import Model
from keras.layers import Dense, Input
from keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten
from keras import backend
```

Listing A.12: MNIST Functional, Imported Libraries

Load the MNIST Dataset

```
# load MNIST Dataset directly from Keras/Tensorflow
mnist = keras.datasets.mnist

# Split the data into a training set and testing set
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# normalize the dataset
x_train, x_test = x_train / 255.0, x_test / 255.0

# 0=left, 1=right
# rule: if number <=5 --> category: left, if number >5 --> category right

y_leftright = np.zeros(y_train.shape, dtype=np.uint8)
for idx, y in enumerate(y_train):
    if y > 5:
        y_leftright[idx] = 1

```

```
    # Display the first twenty digits from the training dataset, to make sure
     the dataset is sensible
    print(y_train.dtype, y_train[0:20])
    print(y_leftright.dtype, y_leftright[0:20])

    # Create a dictionary out of the output labels, with two keys --> in
    order to solve two distinct problems
    y = {"category_output": y_train,
         "leftright_output": y_leftright }
```

Listing A.13: MNIST Functional, Loading Dataset from Keras

```
uint8 [5 0 4 1 9 2 1 3 1 4 3 5 3 6 1 7 2 8 6 9]
uint8 [0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 1 1 1]
```

Creating the Model

The steps here are identical to the previous notebook (see Section A.1. We shall begin by creating the model, compiling it and then training it using model.fit method.

The model will be built in a similar fashion, with the only difference being that we attempt to solve both the problems using one model. Schematically, the architecture is shown in Figure A.2.

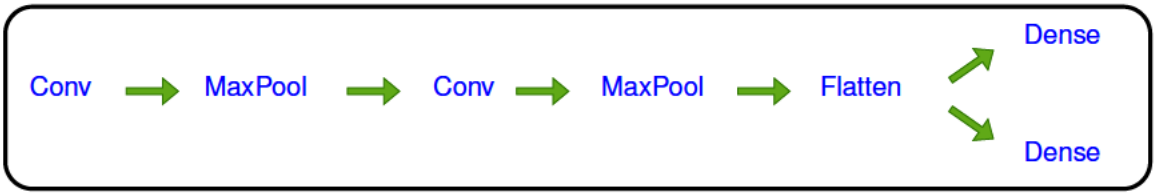

Figure A.2: MNIST Functional: Architecture as implemented

```
    img_rows = 28
    img_cols = 28

    inputLayer = Input(shape=(img_rows, img_cols, 1))

    layer1 = Conv2D(50, kernel_size=(5,5), activation='relu')(inputLayer)
    # Conv is the convolutional layer, 2D because we have a greascale image,
    i.e. it is 2D
    # hyperparamters: number of kernels, size of kernels, activation function
    ,
    # input shape (images are of size 28x28 and greyscale therefore 1)

    layer2 = MaxPooling2D(pool_size=(2,2))(layer1)
    # input size after the max pooling layer will be half of 28x28 and there
    will be 50 images,
    # since we have 50 kernels in the first convolutional layer
    # however, the input_size parameter is only needed in the first
    convolutional layer of the model

    layer3 = Conv2D(50, kernel_size=(3,3), activation='relu')(layer2)

    layer4 = MaxPooling2D(pool_size=(2,2))(layer3)
```

```
19
20      layer5 = Flatten()(layer4)
21
22      layer6 = Dense(10, activation='softmax', name='category_output')(layer5)
23      # This is the first output layer, with 10 outputs
24
25      layer7 = Dense(1, activation='sigmoid', name='leftright_output')(layer5)
26      # This is the second output layer, with 1 output
```

Listing A.14: MNIST Functional, Define Functional Model

```
2024-03-03 17:23:18.379261: I metal_plugin/src/device/metal_device.cc:1154]
Metal device set to: Apple M2 Pro
2024-03-03 17:23:18.379303: I metal_plugin/src/device/metal_device.cc:296]
systemMemory: 16.00 GB
2024-03-03 17:23:18.379314: I metal_plugin/src/device/metal_device.cc:313]
maxCacheSize: 5.33 GB
2024-03-03 17:23:18.379408:
I tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:303]
Could not identify NUMA node of platform GPU ID 0, defaulting to 0.
Your kernel may not have been built with NUMA support.
2024-03-03 17:23:18.379472:
I tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:269]
Created TensorFlow device
(/job:localhost/replica:0/task:0/device:GPU:0 with 0 MB memory) ->
physical PluggableDevice (device: 0, name: METAL, pci bus id: <undefined>)
```

With the architecture of the model defined, the actual model can now be created. Once again, the method model.summary shall be utilized to display and cross-check the model.

```
1       model = keras.Model(inputs=[inputLayer], outputs=[layer6, layer7], name="
        mnist_model")
2
3       # to check if the model has been correctly initialized, the model summary
         can be displayed
4       model.summary()
```

Listing A.15: MNIST Functional, Model Summary

```
Model: "mnistFunc_model"
__________________________________________________________________________
 Layer (type)        Output Shape          Param #   Connected to
==========================================================================
 input_1             [(None, 28, 28, 1)]   0         []
 (InputLayer)

 conv2d              (None, 24, 24, 50)    1300      ['input_1[0][0]']
 (Conv2D)
```

```
 max_pooling2d      (None, 12, 12, 50)    0         ['conv2d[0][0]']
 (MaxPooling2D)

 conv2d_1           (None, 10, 10, 50)    22550     ['max_pooling2d[0][0]']
 (Conv2D)

 max_pooling2d_1    (None, 5, 5, 50)      0         ['conv2d_1[0][0]']
 (MaxPooling2D)

 flatten            (None, 1250)          0         ['max_pooling2d_1[0][0]']
 (Flatten)

 category_output    (None, 10)            12510     ['flatten[0][0]']
 (Dense)

 leftright_output   (None, 1)             1251      ['flatten[0][0]']
 (Dense)
==================================================================================
Total params: 37611 (146.92 KB)
Trainable params: 37611 (146.92 KB)
Non-trainable params: 0 (0.00 Byte)
----------------------------------------------------------------------------------
```

Defining Loss Function

```
1
2     loss1 = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
3     # the parameter 'from_logits=True' tells tensorflow to incorporate the
      SOFTMAX activation function in the loss function
4     loss2 = keras.losses.BinaryCrossentropy(from_logits=False)
5     # binary cross entropy loss function for classification 0 / 1
6
7     losses = {
8         "category_output": loss1,
9         "leftright_output": loss2,
10    }
```

Listing A.16: MNIST Functional, Define Loss Functions and Loss vector

The difference here to the sequential implementation can be seen in the second part of the code. Since both the problems are being solved using one network, the compilation of the network will require a loss function for both of the dense layers together. This is realized by grouping both of the loss functions in a python list, and feeding this grouped loss into the compilation step (see Listing A.17.

```
1     optim = keras.optimizers.legacy.Adam(learning_rate=0.001)
2
3     model.compile(optimizer=optim, loss=losses, metrics = ['accuracy'])
```

```
    # metrics of model should be stored in a list called accuracy
```

Listing A.17: MNIST Functional, Compilation of Model

Training the Model

```
model.fit(x_train, y=y, epochs=15,
          batch_size=64, verbose=2)
```

Listing A.18: MNIST Functional, Training the Model

```
Epoch 1/15
2024-03-10 22:27:13.845977: I tensorflow/core/grappler/optimizers/
custom_graph_optimizer_registry.cc:114]
Plugin optimizer for device_type GPU is enabled.

938/938 - 10s - loss: 0.3648 - category_output_loss: 0.1859 -
leftright_output_loss: 0.1789 - category_output_accuracy: 0.9467 -
leftright_output_accuracy: 0.9308 - 10s/epoch - 11ms/step
Epoch 2/15
938/938 - 9s - loss: 0.1350 - category_output_loss: 0.0548 -
leftright_output_loss: 0.0802 - category_output_accuracy: 0.9834 -
leftright_output_accuracy: 0.9718 - 9s/epoch - 10ms/step
Epoch 3/15
938/938 - 9s - loss: 0.1030 - category_output_loss: 0.0399 -
leftright_output_loss: 0.0631 - category_output_accuracy: 0.9876 -
leftright_output_accuracy: 0.9774 - 9s/epoch - 10ms/step
Epoch 4/15
938/938 - 9s - loss: 0.0862 - category_output_loss: 0.0333 -
leftright_output_loss: 0.0529 - category_output_accuracy: 0.9897 -
leftright_output_accuracy: 0.9811 - 9s/epoch - 10ms/step
Epoch 5/15
938/938 - 9s - loss: 0.0705 - category_output_loss: 0.0250 -
leftright_output_loss: 0.0455 - category_output_accuracy: 0.9921 -
leftright_output_accuracy: 0.9838 - 9s/epoch - 10ms/step
Epoch 6/15
938/938 - 9s - loss: 0.0633 - category_output_loss: 0.0217 -
leftright_output_loss: 0.0416 - category_output_accuracy: 0.9931 -
leftright_output_accuracy: 0.9851 - 9s/epoch - 10ms/step
Epoch 7/15
938/938 - 9s - loss: 0.0542 - category_output_loss: 0.0181 -
leftright_output_loss: 0.0361 - category_output_accuracy: 0.9941 -
leftright_output_accuracy: 0.9877 - 9s/epoch - 10ms/step
Epoch 8/15
938/938 - 9s - loss: 0.0474 - category_output_loss: 0.0146 -
leftright_output_loss: 0.0328 - category_output_accuracy: 0.9952 -
leftright_output_accuracy: 0.9880 - 9s/epoch - 10ms/step
```

```
Epoch 9/15
938/938 - 9s - loss: 0.0433 - category_output_loss: 0.0133 -
leftright_output_loss: 0.0300 - category_output_accuracy: 0.9959 -
leftright_output_accuracy: 0.9894 - 9s/epoch - 10ms/step
Epoch 10/15
938/938 - 9s - loss: 0.0384 - category_output_loss: 0.0114 -
leftright_output_loss: 0.0270 - category_output_accuracy: 0.9963 -
leftright_output_accuracy: 0.9902 - 9s/epoch - 10ms/step
Epoch 11/15
938/938 - 9s - loss: 0.0352 - category_output_loss: 0.0097 -
leftright_output_loss: 0.0255 - category_output_accuracy: 0.9972 -
leftright_output_accuracy: 0.9910 - 9s/epoch - 10ms/step
Epoch 12/15
938/938 - 9s - loss: 0.0308 - category_output_loss: 0.0080 -
leftright_output_loss: 0.0228 - category_output_accuracy: 0.9975 -
leftright_output_accuracy: 0.9922 - 9s/epoch - 10ms/step
Epoch 13/15
938/938 - 9s - loss: 0.0297 - category_output_loss: 0.0081 -
leftright_output_loss: 0.0216 - category_output_accuracy: 0.9970 -
leftright_output_accuracy: 0.9922 - 9s/epoch - 10ms/step
Epoch 14/15
938/938 - 9s - loss: 0.0259 - category_output_loss: 0.0059 -
leftright_output_loss: 0.0200 - category_output_accuracy: 0.9981 -
leftright_output_accuracy: 0.9930 - 9s/epoch - 10ms/step
Epoch 15/15
938/938 - 9s - loss: 0.0236 - category_output_loss: 0.0057 -
leftright_output_loss: 0.0179 - category_output_accuracy: 0.9981 -
leftright_output_accuracy: 0.9936 - 9s/epoch - 10ms/step
Out[27]:
<keras.src.callbacks.History at 0x28f3a5b70>
```

In comparison to the sequential implementation, as expected, two losses are provided. Both of the losses, one each for each problem statement, correspond to the **validation accuracy**, as explained in the sub section A.1. The validation accuracy achieved for the identification of digits (category output) is 99.8%, whilst that for the binning of the digits into one of two groups (leftright output) is 99.4%.

Testing the Model

```
    # list with predictions
    prediction_new = model.predict(x_test)
    len(prediction_new)

    # identification of the digit is the first element of the prediction list
    prediction_category = prediction_new[0]
    # classification left/right is the second element of the prediction list
    prediction_lr = prediction_new[1]

```

```
    # lets observe how the model works for the first 100 MNIST images
    numMNIST = 100

    pr_cat = prediction_category[0:numMNIST]
    prediction_lr = prediction_lr[0:numMNIST]

    labels_cat = np.argmax(pr_cat, axis=1)
    labels_lr = np.array([1 if p >= 0.5 else 0 for p in prediction_lr])


    print('Test Labels: ', y_test[0:numMNIST])
    print('Predicted Labels, Digit: ', labels_cat)
    print('Predicted Labels, L/R: ', labels_lr)

```

Listing A.19: MNIST Sequential, Testing both Models

```
313/313 [==============================] - 1s 2ms/step
Test Labels:  [7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6 6 5 4 0 7 4
0 1 3 1 3 4 7 2 7 1 2 1 1 7 4 2 3 5 1 2 4 4 6 3 5 5 6 0 4 1 9 5 7 8 9
3 7 4 6 4 3 0 7 0 2 9 1 7 3 2 9 7 7 6 2 7 8 4 7 3 6 1 3 6 9 3 1 4 1 7
6 9]
Predicted Labels, Digit:  [7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6
6 5 4 0 7 4 0 1 3 1 3 4 7 2 7 1 2 1 1 7 4 2 3 5 1 2 4 4 6 3 5 5 6 0 4 1
9 5 7 8 9 3 7 4 6 4 3 0 7 0 2 9 1 7 3 2 9 7 7 6 2 7 8 4 7 3 6 1 3 6 9 3
1 4 1 7 6 9]
Predicted Labels, L/R:  [1 0 0 0 0 0 0 1 0 1 0 1 1 0 0 0 1 1 0 0 1 1 1
0 0 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0
1 0 1 1 1 0 1 0 1 0 0 0 1 0 0 1 0 1 0 0 1 1 1 1 0 1 1 0 1 0 1 0 0 1 1
0 0 0 0 1 1 1]
```

Performing a quick check to see how well the identification of the digits works, by computing the difference between the **test labels** and the **predicted labels** (ideally, all of them should be 0):

```
    print('Test Labels - Predicted Labels: ',y_test[0:numMNIST]-labels_cat)
```

```
Test Labels - Predicted Labels:  [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Appendix B: Abalone Dataset Experiments

B.1 Sequential Implementation of Abalone Dataset

B.1.1 Sequential Regression

We begin this chapter first by importing all of the relevant libraries.

```
    # load and summarize the abalone dataset

    # For data preprocessing
    from pandas import read_csv

    # For Calculations
    import numpy as np

    # Tensorflow, for neural network
    from tensorflow.keras.models import Sequential
    from tensorflow.keras.layers import Dense

    # for post processing and splitting data
    from sklearn.metrics import mean_absolute_error
    from sklearn.model_selection import train_test_split
    from sklearn.metrics import mean_squared_error

    # Source of the Dataset
    from ucimlrepo import fetch_ucirepo
```

Listing B.1: Abalone, Regression Relevant Libraries

Next, the dataset shall be downloaded from the source. The source is the Machine Learning Repository of the University of California, Irvine. Once downloaded, it is then preprocessed into pandas dataframes for further use.

```
    # load dataset
    # fetch dataset
    abalone = fetch_ucirepo(id=1)

    # data (as pandas dataframes)
    # split dataset into features and targets
    X = abalone.data.features
    y = abalone.data.targets

    # metadata
    print(abalone.metadata)

    # variable information
    print(abalone.variables)
```

Listing B.2: Abalone, Dataset Example

{'uci_id': 1, 'name': 'Abalone', 'repository_url':
'https://archive.ics.uci.edu/dataset/1/abalone', 'data_url':
'https://archive.ics.uci.edu/static/public/1/data.csv',
'abstract': 'Predict the age of abalone from physical measurements',
'area': 'Biology', 'tasks': ['Classification', 'Regression'],
'characteristics': ['Tabular'], 'num_instances': 4177,
'num_features': 8, 'feature_types': ['Categorical',
'Integer', 'Real'], 'demographics': [],
'target_col': ['Rings'], 'index_col': None,
'has_missing_values': 'no', 'missing_values_symbol': None,
'year_of_dataset_creation': 1994,
'last_updated': 'Mon Aug 28 2023', 'dataset_doi':
'10.24432/C55C7W',
'creators': ['Warwick Nash', 'Tracy Sellers',
'Simon Talbot', 'Andrew Cawthorn', 'Wes Ford'],
'intro_paper': None, 'additional_info':
{'summary': 'Predicting the age of abalone from
physical measurements. The age of abalone is
determined by cutting the shell through the cone,
staining it, and counting the number of rings
through a microscope -- a boring and time-consuming
task. Other measurements, which are easier to obtain,
are used to predict the age. Further information,
such as weather patterns and location
(hence food availability) may be required to solve
the problem.\r\n\r\nFrom the original data examples
with missing values were removed (the majority
having the predicted value missing), and the ranges
of the continuous values have been scaled for use
with an ANN (by dividing by 200).', 'purpose': None,
'funded_by': None, 'instances_represent': None,
'recommended_data_splits': None, 'sensitive_data': None,
'preprocessing_description': None, 'variable_info':
'Given is the attribute name, attribute type, the
measurement unit and a brief description. The
number of rings is the value to predict: either as
a continuous value or as a classification problem.
\r\n\r\nName / Data Type / Measurement Unit /
Description\r\n-----------------------------\r\nSex
/ nominal / -- / M, F, and I (infant)\r\nLength
/ continuous / mm / Longest shell measurement\r\n
Diameter\t/ continuous / mm / perpendicular to length
\r\nHeight / continuous / mm / with meat in shell\r\n
Whole weight / continuous / grams / whole abalone\r\n
Shucked weight / continuous\t / grams / weight of meat
\r\nViscera weight / continuous / grams / gut weight
(after bleeding)\r\nShell weight / continuous / grams

```
    / after being dried\r\nRings / integer / -- /
    +1.5 gives the age in years\r\n\r\n
    The readme file contains attribute statistics.',
    'citation': None}}
             name     role         type demographic  \
0             Sex  Feature  Categorical        None
1          Length  Feature   Continuous        None
2        Diameter  Feature   Continuous        None
3          Height  Feature   Continuous        None
4    Whole_weight  Feature   Continuous        None
5  Shucked_weight  Feature   Continuous        None
6  Viscera_weight  Feature   Continuous        None
7    Shell_weight  Feature   Continuous        None
8           Rings   Target      Integer        None

                  description  units missing_values
0        M, F, and I (infant)   None             no
1   Longest shell measurement     mm             no
2     perpendicular to length     mm             no
3          with meat in shell     mm             no
4               whole abalone  grams             no
5              weight of meat  grams             no
6  gut weight (after bleeding)  grams             no
7           after being dried  grams             no
8  +1.5 gives the age in years   None             no
```

Next, we remove the first column, since the 'Sex' information shall not be used for the task.

```
1    X = X.iloc[:, 1:]
2    X, y = X.astype('float'), y.astype('float')
3    n_features = X.shape[1]
4
5    print('X = ', X)
6    print('y = ', y)
7    print('n_features = ', n_features)
```

Listing B.3: Abalone, Removeing first coloumn

```
    X =
    Length  Diameter  Height  Whole_weight  Shucked_weight  Viscera_weight
0    0.455     0.365   0.095        0.5140          0.2245          0.1010
1    0.350     0.265   0.090        0.2255          0.0995          0.0485
2    0.530     0.420   0.135        0.6770          0.2565          0.1415
3    0.440     0.365   0.125        0.5160          0.2155          0.1140
4    0.330     0.255   0.080        0.2050          0.0895          0.0395
...    ...       ...     ...           ...             ...             ...
4172 0.565     0.450   0.165        0.8870          0.3700          0.2390
4173 0.590     0.440   0.135        0.9660          0.4390          0.2145
```

```
4174   0.600     0.475   0.205        1.1760          0.5255          0.2875
4175   0.625     0.485   0.150        1.0945          0.5310          0.2610
4176   0.710     0.555   0.195        1.9485          0.9455          0.3765

      Shell_weight
0           0.1500
1           0.0700
2           0.2100
3           0.1550
4           0.0550
...            ...
4172        0.2490
4173        0.2605
4174        0.3080
4175        0.2960
4176        0.4950

[4177 rows x 7 columns]

y =        Rings
0       15.0
1        7.0
2        9.0
3       10.0
4        7.0
...      ...
4172    11.0
4173    10.0
4174     9.0
4175    10.0
4176    12.0

[4177 rows x 1 columns]

n_features =  7
```

Next, the dataset is split into a training part and a testing part. The ratio chosen here is 66% for training purposes, and 33% for testing purposes.

```
    # split data into train and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
     random_state=1)
    print('Size of X_train is ', len(X_train))
    print('Size of X_test is ', len(X_test))
    print('Size of y_train is ', len(y_train))
    print('Size of y_test is ', len(y_test))
```

Listing B.4: Split data into training and testing datasets

```
Size of X_train is  2798
Size of X_test is  1379
Size of y_train is  2798
Size of y_test is  1379
```

Having pre-processed the data, the model is then defined. The point of this experiment is to do this implementation using the Sequential API of Keras. The model is a simple FFN, with 20 nodes in the first hidden layer and 10 nodes in the second hidden layer.

```
1    model = Sequential()
2    model.add(Dense(20, input_dim=n_features, activation='relu',
     kernel_initializer='he_normal'))
3    model.add(Dense(10, activation='relu', kernel_initializer='he_normal'))
4    model.add(Dense(1, activation='linear'))
```

Listing B.5: Sequential Regression: Defining the model

And the model is visualized via the model summary:

```
1 model.summary()
```

Listing B.6: Sequential Regression: Model Summary

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 20)                160

 dense_1 (Dense)             (None, 10)                210

 dense_2 (Dense)             (None, 1)                 11

=================================================================
Total params: 381 (1.49 KB)
Trainable params: 381 (1.49 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Next, the model is compiled using an appropriate loss function and an optimizer. The Mean Squared Error has been implemented here as the loss function and the ADAM algorithm as the optimizer.

```
1    model.compile(loss='mse', optimizer='adam', metrics = ["
     mean_absolute_error"])
```

Listing B.7: Sequential Regression: Loss Function and Model Compilation

The Mean Absolute Error has been used as the metric used to evaluate the results.

Once the model has been compiled, it is trained using the training part of the dataset, as shown below. Please keep in mind, due to the stochastic nature of the network, the output would differ if the code is re-run. The results however should always be similar, although would not always be the same (the result may get 'stuck' in a local minima).

```
1    model.fit(X_train, y_train, epochs=150, batch_size=32, verbose=2)
```

Listing B.8: Sequential Regression: Training Model

```
poch 1/150
88/88 - 0s - loss: 5.0039 - mean_absolute_error: 1.6172 - 342ms/epoch - 4ms/step
Epoch 2/150
88/88 - 0s - loss: 4.9924 - mean_absolute_error: 1.6266 - 316ms/epoch - 4ms/step
Epoch 3/150
88/88 - 0s - loss: 4.9796 - mean_absolute_error: 1.6156 - 303ms/epoch - 3ms/step
Epoch 4/150
88/88 - 0s - loss: 4.9913 - mean_absolute_error: 1.6147 - 289ms/epoch - 3ms/step
Epoch 5/150
88/88 - 0s - loss: 4.9970 - mean_absolute_error: 1.6255 - 299ms/epoch - 3ms/step
Epoch 6/150
88/88 - 0s - loss: 4.9825 - mean_absolute_error: 1.6165 - 286ms/epoch - 3ms/step
Epoch 7/150
88/88 - 0s - loss: 4.9983 - mean_absolute_error: 1.6226 - 293ms/epoch - 3ms/step
Epoch 8/150
88/88 - 0s - loss: 4.9864 - mean_absolute_error: 1.6231 - 286ms/epoch - 3ms/step
Epoch 9/150
88/88 - 0s - loss: 4.9823 - mean_absolute_error: 1.6129 - 291ms/epoch - 3ms/step
Epoch 10/150
88/88 - 0s - loss: 5.0281 - mean_absolute_error: 1.6205 - 288ms/epoch - 3ms/step
Epoch 11/150
88/88 - 0s - loss: 5.0215 - mean_absolute_error: 1.6313 - 285ms/epoch - 3ms/step
Epoch 12/150
88/88 - 0s - loss: 5.0001 - mean_absolute_error: 1.6247 - 287ms/epoch - 3ms/step
Epoch 13/150
88/88 - 0s - loss: 4.9995 - mean_absolute_error: 1.6311 - 286ms/epoch - 3ms/step
Epoch 14/150
88/88 - 0s - loss: 4.9859 - mean_absolute_error: 1.6126 - 289ms/epoch - 3ms/step
Epoch 15/150
88/88 - 0s - loss: 4.9930 - mean_absolute_error: 1.6260 - 290ms/epoch - 3ms/step
Epoch 16/150
88/88 - 0s - loss: 5.0009 - mean_absolute_error: 1.6175 - 286ms/epoch - 3ms/step
Epoch 17/150
88/88 - 0s - loss: 4.9851 - mean_absolute_error: 1.6171 - 283ms/epoch - 3ms/step
Epoch 18/150
88/88 - 0s - loss: 4.9675 - mean_absolute_error: 1.6204 - 286ms/epoch - 3ms/step
Epoch 19/150
88/88 - 0s - loss: 5.0634 - mean_absolute_error: 1.6283 - 273ms/epoch - 3ms/step
```

```
Epoch 20/150
88/88 - 0s - loss: 4.9670 - mean_absolute_error: 1.6109 - 272ms/epoch - 3ms/step
Epoch 21/150
88/88 - 0s - loss: 4.9899 - mean_absolute_error: 1.6217 - 277ms/epoch - 3ms/step
Epoch 22/150
88/88 - 0s - loss: 4.9983 - mean_absolute_error: 1.6251 - 278ms/epoch - 3ms/step
Epoch 23/150
88/88 - 0s - loss: 4.9806 - mean_absolute_error: 1.6147 - 276ms/epoch - 3ms/step
Epoch 24/150
88/88 - 0s - loss: 5.0557 - mean_absolute_error: 1.6365 - 280ms/epoch - 3ms/step
Epoch 25/150
88/88 - 0s - loss: 5.0621 - mean_absolute_error: 1.6209 - 281ms/epoch - 3ms/step
Epoch 26/150
88/88 - 0s - loss: 4.9864 - mean_absolute_error: 1.6220 - 280ms/epoch - 3ms/step
Epoch 27/150
88/88 - 0s - loss: 4.9808 - mean_absolute_error: 1.6169 - 289ms/epoch - 3ms/step
Epoch 28/150
88/88 - 0s - loss: 5.0648 - mean_absolute_error: 1.6292 - 280ms/epoch - 3ms/step
Epoch 29/150
88/88 - 0s - loss: 5.0154 - mean_absolute_error: 1.6238 - 285ms/epoch - 3ms/step
Epoch 30/150
88/88 - 0s - loss: 4.9789 - mean_absolute_error: 1.6198 - 272ms/epoch - 3ms/step
Epoch 31/150
88/88 - 0s - loss: 4.9931 - mean_absolute_error: 1.6115 - 278ms/epoch - 3ms/step
Epoch 32/150
88/88 - 0s - loss: 4.9984 - mean_absolute_error: 1.6242 - 279ms/epoch - 3ms/step
Epoch 33/150
88/88 - 0s - loss: 4.9887 - mean_absolute_error: 1.6158 - 284ms/epoch - 3ms/step
Epoch 34/150
88/88 - 0s - loss: 4.9860 - mean_absolute_error: 1.6139 - 281ms/epoch - 3ms/step
Epoch 35/150
88/88 - 0s - loss: 5.0660 - mean_absolute_error: 1.6391 - 284ms/epoch - 3ms/step
Epoch 36/150
88/88 - 0s - loss: 5.0157 - mean_absolute_error: 1.6181 - 280ms/epoch - 3ms/step
Epoch 37/150
88/88 - 0s - loss: 5.0655 - mean_absolute_error: 1.6367 - 286ms/epoch - 3ms/step
Epoch 38/150
88/88 - 0s - loss: 5.0523 - mean_absolute_error: 1.6382 - 279ms/epoch - 3ms/step
Epoch 39/150
88/88 - 0s - loss: 4.9726 - mean_absolute_error: 1.6112 - 277ms/epoch - 3ms/step
Epoch 40/150
88/88 - 0s - loss: 4.9930 - mean_absolute_error: 1.6257 - 292ms/epoch - 3ms/step
Epoch 41/150
88/88 - 0s - loss: 5.0104 - mean_absolute_error: 1.6223 - 279ms/epoch - 3ms/step
Epoch 42/150
88/88 - 0s - loss: 4.9900 - mean_absolute_error: 1.6182 - 278ms/epoch - 3ms/step
Epoch 43/150
```

```
88/88 - 0s - loss: 5.0252 - mean_absolute_error: 1.6257 - 278ms/epoch - 3ms/step
Epoch 44/150
88/88 - 0s - loss: 4.9896 - mean_absolute_error: 1.6222 - 279ms/epoch - 3ms/step
Epoch 45/150
88/88 - 0s - loss: 5.0339 - mean_absolute_error: 1.6191 - 275ms/epoch - 3ms/step
Epoch 46/150
88/88 - 0s - loss: 5.0093 - mean_absolute_error: 1.6216 - 278ms/epoch - 3ms/step
Epoch 47/150
88/88 - 0s - loss: 5.0007 - mean_absolute_error: 1.6193 - 276ms/epoch - 3ms/step
Epoch 48/150
88/88 - 0s - loss: 4.9777 - mean_absolute_error: 1.6146 - 274ms/epoch - 3ms/step
Epoch 49/150
88/88 - 0s - loss: 5.0165 - mean_absolute_error: 1.6207 - 273ms/epoch - 3ms/step
Epoch 50/150
88/88 - 0s - loss: 5.0018 - mean_absolute_error: 1.6227 - 281ms/epoch - 3ms/step
Epoch 51/150
88/88 - 0s - loss: 5.0045 - mean_absolute_error: 1.6235 - 276ms/epoch - 3ms/step
Epoch 52/150
88/88 - 0s - loss: 5.0177 - mean_absolute_error: 1.6226 - 280ms/epoch - 3ms/step
Epoch 53/150
88/88 - 0s - loss: 5.0289 - mean_absolute_error: 1.6205 - 291ms/epoch - 3ms/step
Epoch 54/150
88/88 - 0s - loss: 4.9610 - mean_absolute_error: 1.6121 - 298ms/epoch - 3ms/step
Epoch 55/150
88/88 - 0s - loss: 4.9598 - mean_absolute_error: 1.6161 - 277ms/epoch - 3ms/step
Epoch 56/150
88/88 - 0s - loss: 4.9910 - mean_absolute_error: 1.6199 - 278ms/epoch - 3ms/step
Epoch 57/150
88/88 - 0s - loss: 5.0144 - mean_absolute_error: 1.6240 - 283ms/epoch - 3ms/step
Epoch 58/150
88/88 - 0s - loss: 4.9739 - mean_absolute_error: 1.6196 - 280ms/epoch - 3ms/step
Epoch 59/150
88/88 - 0s - loss: 4.9496 - mean_absolute_error: 1.6117 - 278ms/epoch - 3ms/step
Epoch 60/150
88/88 - 0s - loss: 4.9904 - mean_absolute_error: 1.6164 - 286ms/epoch - 3ms/step
Epoch 61/150
88/88 - 0s - loss: 4.9906 - mean_absolute_error: 1.6283 - 284ms/epoch - 3ms/step
Epoch 62/150
88/88 - 0s - loss: 4.9575 - mean_absolute_error: 1.6111 - 283ms/epoch - 3ms/step
Epoch 63/150
88/88 - 0s - loss: 4.9843 - mean_absolute_error: 1.6263 - 285ms/epoch - 3ms/step
Epoch 64/150
88/88 - 0s - loss: 4.9929 - mean_absolute_error: 1.6312 - 276ms/epoch - 3ms/step
Epoch 65/150
88/88 - 0s - loss: 5.0006 - mean_absolute_error: 1.6142 - 278ms/epoch - 3ms/step
Epoch 66/150
88/88 - 0s - loss: 5.0179 - mean_absolute_error: 1.6308 - 284ms/epoch - 3ms/step
```

```
Epoch 67/150
88/88 - 0s - loss: 5.0005 - mean_absolute_error: 1.6209 - 282ms/epoch - 3ms/step
Epoch 68/150
88/88 - 0s - loss: 5.0257 - mean_absolute_error: 1.6267 - 279ms/epoch - 3ms/step
Epoch 69/150
88/88 - 0s - loss: 4.9770 - mean_absolute_error: 1.6143 - 284ms/epoch - 3ms/step
Epoch 70/150
88/88 - 0s - loss: 5.0725 - mean_absolute_error: 1.6355 - 283ms/epoch - 3ms/step
Epoch 71/150
88/88 - 0s - loss: 4.9513 - mean_absolute_error: 1.6144 - 275ms/epoch - 3ms/step
Epoch 72/150
88/88 - 0s - loss: 4.9822 - mean_absolute_error: 1.6222 - 279ms/epoch - 3ms/step
Epoch 73/150
88/88 - 0s - loss: 4.9605 - mean_absolute_error: 1.6194 - 283ms/epoch - 3ms/step
Epoch 74/150
88/88 - 0s - loss: 4.9708 - mean_absolute_error: 1.6234 - 283ms/epoch - 3ms/step
Epoch 75/150
88/88 - 0s - loss: 4.9842 - mean_absolute_error: 1.6195 - 277ms/epoch - 3ms/step
Epoch 76/150
88/88 - 0s - loss: 4.9916 - mean_absolute_error: 1.6215 - 281ms/epoch - 3ms/step
Epoch 77/150
88/88 - 0s - loss: 4.9689 - mean_absolute_error: 1.6190 - 276ms/epoch - 3ms/step
Epoch 78/150
88/88 - 0s - loss: 4.9839 - mean_absolute_error: 1.6158 - 279ms/epoch - 3ms/step
Epoch 79/150
88/88 - 0s - loss: 4.9844 - mean_absolute_error: 1.6245 - 279ms/epoch - 3ms/step
Epoch 80/150
88/88 - 0s - loss: 4.9624 - mean_absolute_error: 1.6130 - 280ms/epoch - 3ms/step
Epoch 81/150
88/88 - 0s - loss: 4.9867 - mean_absolute_error: 1.6168 - 278ms/epoch - 3ms/step
Epoch 82/150
88/88 - 0s - loss: 4.9670 - mean_absolute_error: 1.6208 - 277ms/epoch - 3ms/step
Epoch 83/150
88/88 - 0s - loss: 5.0117 - mean_absolute_error: 1.6197 - 277ms/epoch - 3ms/step
Epoch 84/150
88/88 - 0s - loss: 4.9713 - mean_absolute_error: 1.6162 - 280ms/epoch - 3ms/step
Epoch 85/150
88/88 - 0s - loss: 4.9748 - mean_absolute_error: 1.6188 - 281ms/epoch - 3ms/step
Epoch 86/150
88/88 - 0s - loss: 5.0195 - mean_absolute_error: 1.6297 - 278ms/epoch - 3ms/step
Epoch 87/150
88/88 - 0s - loss: 4.9833 - mean_absolute_error: 1.6166 - 276ms/epoch - 3ms/step
Epoch 88/150
88/88 - 0s - loss: 4.9573 - mean_absolute_error: 1.6146 - 281ms/epoch - 3ms/step
Epoch 89/150
88/88 - 0s - loss: 4.9966 - mean_absolute_error: 1.6261 - 271ms/epoch - 3ms/step
Epoch 90/150
```

```
88/88 - 0s - loss: 4.9791 - mean_absolute_error: 1.6205 - 267ms/epoch - 3ms/step
Epoch 91/150
88/88 - 0s - loss: 4.9725 - mean_absolute_error: 1.6142 - 265ms/epoch - 3ms/step
Epoch 92/150
88/88 - 0s - loss: 4.9791 - mean_absolute_error: 1.6202 - 265ms/epoch - 3ms/step
Epoch 93/150
88/88 - 0s - loss: 4.9640 - mean_absolute_error: 1.6186 - 265ms/epoch - 3ms/step
Epoch 94/150
88/88 - 0s - loss: 5.0538 - mean_absolute_error: 1.6323 - 286ms/epoch - 3ms/step
Epoch 95/150
88/88 - 0s - loss: 4.9686 - mean_absolute_error: 1.6198 - 290ms/epoch - 3ms/step
Epoch 96/150
88/88 - 0s - loss: 5.0177 - mean_absolute_error: 1.6267 - 283ms/epoch - 3ms/step
Epoch 97/150
88/88 - 0s - loss: 4.9954 - mean_absolute_error: 1.6222 - 275ms/epoch - 3ms/step
Epoch 98/150
88/88 - 0s - loss: 5.0145 - mean_absolute_error: 1.6238 - 277ms/epoch - 3ms/step
Epoch 99/150
88/88 - 0s - loss: 4.9608 - mean_absolute_error: 1.6181 - 327ms/epoch - 4ms/step
Epoch 100/150
88/88 - 0s - loss: 4.9882 - mean_absolute_error: 1.6109 - 309ms/epoch - 4ms/step
Epoch 101/150
88/88 - 0s - loss: 4.9940 - mean_absolute_error: 1.6290 - 302ms/epoch - 3ms/step
Epoch 102/150
88/88 - 0s - loss: 4.9816 - mean_absolute_error: 1.6151 - 282ms/epoch - 3ms/step
Epoch 103/150
88/88 - 0s - loss: 4.9784 - mean_absolute_error: 1.6239 - 287ms/epoch - 3ms/step
Epoch 104/150
88/88 - 0s - loss: 4.9785 - mean_absolute_error: 1.6175 - 288ms/epoch - 3ms/step
Epoch 105/150
88/88 - 0s - loss: 4.9836 - mean_absolute_error: 1.6183 - 276ms/epoch - 3ms/step
Epoch 106/150
88/88 - 0s - loss: 4.9599 - mean_absolute_error: 1.6152 - 277ms/epoch - 3ms/step
Epoch 107/150
88/88 - 0s - loss: 4.9947 - mean_absolute_error: 1.6202 - 277ms/epoch - 3ms/step
Epoch 108/150
88/88 - 0s - loss: 4.9731 - mean_absolute_error: 1.6122 - 277ms/epoch - 3ms/step
Epoch 109/150
88/88 - 0s - loss: 4.9782 - mean_absolute_error: 1.6201 - 280ms/epoch - 3ms/step
Epoch 110/150
88/88 - 0s - loss: 4.9882 - mean_absolute_error: 1.6200 - 281ms/epoch - 3ms/step
Epoch 111/150
88/88 - 0s - loss: 4.9710 - mean_absolute_error: 1.6173 - 286ms/epoch - 3ms/step
Epoch 112/150
88/88 - 0s - loss: 4.9902 - mean_absolute_error: 1.6169 - 279ms/epoch - 3ms/step
Epoch 113/150
88/88 - 0s - loss: 4.9859 - mean_absolute_error: 1.6225 - 276ms/epoch - 3ms/step
```

```
Epoch 114/150
88/88 - 0s - loss: 5.0228 - mean_absolute_error: 1.6220 - 274ms/epoch - 3ms/step
Epoch 115/150
88/88 - 0s - loss: 4.9591 - mean_absolute_error: 1.6171 - 274ms/epoch - 3ms/step
Epoch 116/150
88/88 - 0s - loss: 4.9847 - mean_absolute_error: 1.6220 - 274ms/epoch - 3ms/step
Epoch 117/150
88/88 - 0s - loss: 5.0143 - mean_absolute_error: 1.6219 - 281ms/epoch - 3ms/step
Epoch 118/150
88/88 - 0s - loss: 4.9704 - mean_absolute_error: 1.6153 - 280ms/epoch - 3ms/step
Epoch 119/150
88/88 - 0s - loss: 4.9941 - mean_absolute_error: 1.6215 - 280ms/epoch - 3ms/step
Epoch 120/150
88/88 - 0s - loss: 4.9739 - mean_absolute_error: 1.6191 - 280ms/epoch - 3ms/step
Epoch 121/150
88/88 - 0s - loss: 5.0445 - mean_absolute_error: 1.6312 - 279ms/epoch - 3ms/step
Epoch 122/150
88/88 - 0s - loss: 4.9808 - mean_absolute_error: 1.6199 - 278ms/epoch - 3ms/step
Epoch 123/150
88/88 - 0s - loss: 4.9628 - mean_absolute_error: 1.6133 - 277ms/epoch - 3ms/step
Epoch 124/150
88/88 - 0s - loss: 4.9737 - mean_absolute_error: 1.6178 - 276ms/epoch - 3ms/step
Epoch 125/150
88/88 - 0s - loss: 4.9867 - mean_absolute_error: 1.6172 - 278ms/epoch - 3ms/step
Epoch 126/150
88/88 - 0s - loss: 4.9306 - mean_absolute_error: 1.6129 - 277ms/epoch - 3ms/step
Epoch 127/150
88/88 - 0s - loss: 4.9728 - mean_absolute_error: 1.6175 - 296ms/epoch - 3ms/step
Epoch 128/150
88/88 - 0s - loss: 4.9794 - mean_absolute_error: 1.6221 - 295ms/epoch - 3ms/step
Epoch 129/150
88/88 - 0s - loss: 4.9832 - mean_absolute_error: 1.6152 - 325ms/epoch - 4ms/step
Epoch 130/150
88/88 - 0s - loss: 4.9779 - mean_absolute_error: 1.6148 - 289ms/epoch - 3ms/step
Epoch 131/150
88/88 - 0s - loss: 5.0353 - mean_absolute_error: 1.6293 - 281ms/epoch - 3ms/step
Epoch 132/150
88/88 - 0s - loss: 4.9632 - mean_absolute_error: 1.6098 - 283ms/epoch - 3ms/step
Epoch 133/150
88/88 - 0s - loss: 5.0038 - mean_absolute_error: 1.6264 - 281ms/epoch - 3ms/step
Epoch 134/150
88/88 - 0s - loss: 4.9985 - mean_absolute_error: 1.6243 - 278ms/epoch - 3ms/step
Epoch 135/150
88/88 - 0s - loss: 5.0050 - mean_absolute_error: 1.6271 - 280ms/epoch - 3ms/step
Epoch 136/150
88/88 - 0s - loss: 5.0631 - mean_absolute_error: 1.6367 - 276ms/epoch - 3ms/step
Epoch 137/150
```

```
88/88 - 0s - loss: 4.9687 - mean_absolute_error: 1.6133 - 280ms/epoch - 3ms/step
Epoch 138/150
88/88 - 0s - loss: 4.9969 - mean_absolute_error: 1.6141 - 279ms/epoch - 3ms/step
Epoch 139/150
88/88 - 0s - loss: 5.0437 - mean_absolute_error: 1.6338 - 278ms/epoch - 3ms/step
Epoch 140/150
88/88 - 0s - loss: 4.9680 - mean_absolute_error: 1.6143 - 281ms/epoch - 3ms/step
Epoch 141/150
88/88 - 0s - loss: 4.9740 - mean_absolute_error: 1.6143 - 288ms/epoch - 3ms/step
Epoch 142/150
88/88 - 0s - loss: 4.9729 - mean_absolute_error: 1.6193 - 280ms/epoch - 3ms/step
Epoch 143/150
88/88 - 0s - loss: 4.9978 - mean_absolute_error: 1.6228 - 280ms/epoch - 3ms/step
Epoch 144/150
88/88 - 0s - loss: 4.9570 - mean_absolute_error: 1.6108 - 281ms/epoch - 3ms/step
Epoch 145/150
88/88 - 0s - loss: 5.0534 - mean_absolute_error: 1.6360 - 276ms/epoch - 3ms/step
Epoch 146/150
88/88 - 0s - loss: 5.0302 - mean_absolute_error: 1.6199 - 278ms/epoch - 3ms/step
Epoch 147/150
88/88 - 0s - loss: 4.9746 - mean_absolute_error: 1.6178 - 278ms/epoch - 3ms/step
Epoch 148/150
88/88 - 0s - loss: 4.9919 - mean_absolute_error: 1.6247 - 282ms/epoch - 3ms/step
Epoch 149/150
88/88 - 0s - loss: 4.9776 - mean_absolute_error: 1.6100 - 277ms/epoch - 3ms/step
Epoch 150/150
88/88 - 0s - loss: 5.0081 - mean_absolute_error: 1.6245 - 281ms/epoch - 3ms/step
```

Once the model has been trained, it may be used to make predictions on the testing dataset.

```
yhat = model.predict(X_test)
ma_error = mean_absolute_error(y_test, yhat)
ms_error = mean_squared_error(y_test, yhat)
print('MAE: %.3f' % ma_error)
print('MSE: %.3f' % ms_error)
```

Listing B.9: Sequential Regression: Predicting values using testing data

```
44/44 [==============================] - 0s 2ms/step
MAE: 1.630
MSE: 5.060
```

The mean absolute error, MAE, shows that the result is accurate to an error of approx. 1.6 (rings).

B.1.2 Sequential Classification

We begin this chapter first by importing all of the relevant libraries. The libraries required here slightly differ from those needed for the regression task, therefore this code is included here as well:

```
# load and summarize the abalone dataset

# For data preprocessing
from pandas import read_csv

# For Calculations
import numpy as np
from numpy import unique
from numpy import argmax

# Tensorflow, for neural network
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# for post processing and splitting data
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# Source of the Dataset
from ucimlrepo import fetch_ucirepo
```

Listing B.10: Abalone, Classification Relevant Libraries

Importing the data (Listing B.2) and pre-processing (Listing B.3) it are done in an identical manner as with the regression task, therefore the code shall not be repeated here. The reader is referred to the Listings mentioned.

Splitting the data works slightly differently here. For the classification problem each unique ring number is defined as a class. One way to achieve this is to assign an integer to each unique class, starting at 0 and ending at *number of classes - 1*. This can be done with the help of a so called **Label Encoder**, as provided by the Scikit Learn Library. For further information on the method, the reader is referred to the Citation [15, sklearn.preprocessing.LabelEncoder].

```
from sklearn.preprocessing import LabelEncoder

# encode strings to integer
y = LabelEncoder().fit_transform(y)
print('y = ', y)
# total number of classes
n_class = len(unique(y))
print('n_class = ', n_class)
```

Listing B.11: Classify target data using label encoder

```
y =  [14  6  8 ...  8  9 11]
n_class =  28
```

The code above shows that there are 28 distinct classes that each data-point can be classified into.

Next, the model can now be defined for the classification task. The model is intentionally kept similar to the regression task, apart from the final output layer. The output layer shall now consist of 'n_features' number of nodes, i.e. one node for each class.

```
    model = Sequential()
    model.add(Dense(20, input_dim=n_features, activation='relu',
    kernel_initializer='he_normal'))
    model.add(Dense(10, activation='relu', kernel_initializer='he_normal'))
    # note, the last dense layer now has an output = number of classes and
    uses a softmax activation
    model.add(Dense(n_class, activation='softmax'))
```

Listing B.12: Sequential Classification: Defining the model

And the model summary can be used to visualize the model:

```
model.summary()
```

Listing B.13: Sequential Classification: Model Summary

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 20)                160

 dense_1 (Dense)             (None, 10)                210

 dense_2 (Dense)             (None, 28)                308

=================================================================
Total params: 678 (2.65 KB)
Trainable params: 678 (2.65 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

The model definition is followed by the model compilation. Care has to be taken here. A classification task is inherently different than a regression task - therefore the loss function needs to be adapted to a more suitable one. Also, the metric MAE cannot be used here since it is not appropriate for classification problems. Instead, as will be shown later on in the code, the so-called accuracy score shall be used for that purpose.

```
    model.compile(loss='sparse_categorical_crossentropy', optimizer='adam')
```

Listing B.14: Sequential Classification: Loss Function and Model Compilation

After compiling the model, it is fitted to the training data set:

```
model.fit(X_train, y_train, epochs=150, batch_size=32, verbose=2)
```

Listing B.15: Sequential Classification: Training Model

```
Epoch 1/150
88/88 - 1s - loss: 3.0768 - 700ms/epoch - 8ms/step
Epoch 2/150
88/88 - 0s - loss: 2.7059 - 334ms/epoch - 4ms/step
Epoch 3/150
88/88 - 0s - loss: 2.5772 - 328ms/epoch - 4ms/step
Epoch 4/150
88/88 - 0s - loss: 2.5148 - 327ms/epoch - 4ms/step
Epoch 5/150
88/88 - 0s - loss: 2.4654 - 329ms/epoch - 4ms/step
Epoch 6/150
88/88 - 0s - loss: 2.4114 - 327ms/epoch - 4ms/step
Epoch 7/150
88/88 - 0s - loss: 2.3515 - 324ms/epoch - 4ms/step
Epoch 8/150
88/88 - 0s - loss: 2.2957 - 323ms/epoch - 4ms/step
Epoch 9/150
88/88 - 0s - loss: 2.2508 - 323ms/epoch - 4ms/step
Epoch 10/150
88/88 - 0s - loss: 2.2212 - 324ms/epoch - 4ms/step
Epoch 11/150
88/88 - 0s - loss: 2.1953 - 319ms/epoch - 4ms/step
Epoch 12/150
88/88 - 0s - loss: 2.1740 - 329ms/epoch - 4ms/step
Epoch 13/150
88/88 - 0s - loss: 2.1575 - 321ms/epoch - 4ms/step
Epoch 14/150
88/88 - 0s - loss: 2.1416 - 321ms/epoch - 4ms/step
Epoch 15/150
88/88 - 0s - loss: 2.1279 - 320ms/epoch - 4ms/step
Epoch 16/150
88/88 - 0s - loss: 2.1148 - 323ms/epoch - 4ms/step
Epoch 17/150
88/88 - 0s - loss: 2.1036 - 320ms/epoch - 4ms/step
Epoch 18/150
88/88 - 0s - loss: 2.0931 - 318ms/epoch - 4ms/step
Epoch 19/150
88/88 - 0s - loss: 2.0816 - 324ms/epoch - 4ms/step
Epoch 20/150
88/88 - 0s - loss: 2.0713 - 317ms/epoch - 4ms/step
Epoch 21/150
88/88 - 0s - loss: 2.0643 - 321ms/epoch - 4ms/step
```

```
Epoch 22/150
88/88 - 0s - loss: 2.0543 - 319ms/epoch - 4ms/step
Epoch 23/150
88/88 - 0s - loss: 2.0498 - 318ms/epoch - 4ms/step
Epoch 24/150
88/88 - 0s - loss: 2.0413 - 322ms/epoch - 4ms/step
Epoch 25/150
88/88 - 0s - loss: 2.0381 - 323ms/epoch - 4ms/step
Epoch 26/150
88/88 - 0s - loss: 2.0298 - 321ms/epoch - 4ms/step
Epoch 27/150
88/88 - 0s - loss: 2.0242 - 322ms/epoch - 4ms/step
Epoch 28/150
88/88 - 0s - loss: 2.0200 - 321ms/epoch - 4ms/step
Epoch 29/150
88/88 - 0s - loss: 2.0144 - 319ms/epoch - 4ms/step
Epoch 30/150
88/88 - 0s - loss: 2.0093 - 320ms/epoch - 4ms/step
Epoch 31/150
88/88 - 0s - loss: 2.0065 - 325ms/epoch - 4ms/step
Epoch 32/150
88/88 - 0s - loss: 2.0017 - 323ms/epoch - 4ms/step
Epoch 33/150
88/88 - 0s - loss: 1.9963 - 324ms/epoch - 4ms/step
Epoch 34/150
88/88 - 0s - loss: 1.9945 - 323ms/epoch - 4ms/step
Epoch 35/150
88/88 - 0s - loss: 1.9932 - 323ms/epoch - 4ms/step
Epoch 36/150
88/88 - 0s - loss: 1.9885 - 319ms/epoch - 4ms/step
Epoch 37/150
88/88 - 0s - loss: 1.9854 - 317ms/epoch - 4ms/step
Epoch 38/150
88/88 - 0s - loss: 1.9838 - 338ms/epoch - 4ms/step
Epoch 39/150
88/88 - 0s - loss: 1.9833 - 324ms/epoch - 4ms/step
Epoch 40/150
88/88 - 0s - loss: 1.9820 - 322ms/epoch - 4ms/step
Epoch 41/150
88/88 - 0s - loss: 1.9803 - 320ms/epoch - 4ms/step
Epoch 42/150
88/88 - 0s - loss: 1.9757 - 321ms/epoch - 4ms/step
Epoch 43/150
88/88 - 0s - loss: 1.9746 - 320ms/epoch - 4ms/step
Epoch 44/150
88/88 - 0s - loss: 1.9713 - 323ms/epoch - 4ms/step
Epoch 45/150
```

```
88/88 - 0s - loss: 1.9717 - 323ms/epoch - 4ms/step
Epoch 46/150
88/88 - 0s - loss: 1.9716 - 325ms/epoch - 4ms/step
Epoch 47/150
88/88 - 0s - loss: 1.9719 - 324ms/epoch - 4ms/step
Epoch 48/150
88/88 - 0s - loss: 1.9714 - 315ms/epoch - 4ms/step
Epoch 49/150
88/88 - 0s - loss: 1.9653 - 325ms/epoch - 4ms/step
Epoch 50/150
88/88 - 0s - loss: 1.9659 - 323ms/epoch - 4ms/step
Epoch 51/150
88/88 - 0s - loss: 1.9649 - 323ms/epoch - 4ms/step
Epoch 52/150
88/88 - 0s - loss: 1.9642 - 321ms/epoch - 4ms/step
Epoch 53/150
88/88 - 0s - loss: 1.9652 - 319ms/epoch - 4ms/step
Epoch 54/150
88/88 - 0s - loss: 1.9646 - 319ms/epoch - 4ms/step
Epoch 55/150
88/88 - 0s - loss: 1.9610 - 320ms/epoch - 4ms/step
Epoch 56/150
88/88 - 0s - loss: 1.9606 - 321ms/epoch - 4ms/step
Epoch 57/150
88/88 - 0s - loss: 1.9604 - 323ms/epoch - 4ms/step
Epoch 58/150
88/88 - 0s - loss: 1.9596 - 321ms/epoch - 4ms/step
Epoch 59/150
88/88 - 0s - loss: 1.9592 - 323ms/epoch - 4ms/step
Epoch 60/150
88/88 - 0s - loss: 1.9597 - 324ms/epoch - 4ms/step
Epoch 61/150
88/88 - 0s - loss: 1.9586 - 326ms/epoch - 4ms/step
Epoch 62/150
88/88 - 0s - loss: 1.9574 - 320ms/epoch - 4ms/step
Epoch 63/150
88/88 - 0s - loss: 1.9587 - 324ms/epoch - 4ms/step
Epoch 64/150
88/88 - 0s - loss: 1.9587 - 321ms/epoch - 4ms/step
Epoch 65/150
88/88 - 0s - loss: 1.9603 - 322ms/epoch - 4ms/step
Epoch 66/150
88/88 - 0s - loss: 1.9561 - 321ms/epoch - 4ms/step
Epoch 67/150
88/88 - 0s - loss: 1.9558 - 323ms/epoch - 4ms/step
Epoch 68/150
88/88 - 0s - loss: 1.9560 - 321ms/epoch - 4ms/step
```

```
Epoch 69/150
88/88 - 0s - loss: 1.9554 - 323ms/epoch - 4ms/step
Epoch 70/150
88/88 - 0s - loss: 1.9560 - 323ms/epoch - 4ms/step
Epoch 71/150
88/88 - 0s - loss: 1.9540 - 320ms/epoch - 4ms/step
Epoch 72/150
88/88 - 0s - loss: 1.9551 - 326ms/epoch - 4ms/step
Epoch 73/150
88/88 - 0s - loss: 1.9536 - 320ms/epoch - 4ms/step
Epoch 74/150
88/88 - 0s - loss: 1.9556 - 320ms/epoch - 4ms/step
Epoch 75/150
88/88 - 0s - loss: 1.9556 - 320ms/epoch - 4ms/step
Epoch 76/150
88/88 - 0s - loss: 1.9535 - 321ms/epoch - 4ms/step
Epoch 77/150
88/88 - 0s - loss: 1.9535 - 326ms/epoch - 4ms/step
Epoch 78/150
88/88 - 0s - loss: 1.9528 - 325ms/epoch - 4ms/step
Epoch 79/150
88/88 - 0s - loss: 1.9536 - 325ms/epoch - 4ms/step
Epoch 80/150
88/88 - 0s - loss: 1.9555 - 325ms/epoch - 4ms/step
Epoch 81/150
88/88 - 0s - loss: 1.9526 - 320ms/epoch - 4ms/step
Epoch 82/150
88/88 - 0s - loss: 1.9533 - 324ms/epoch - 4ms/step
Epoch 83/150
88/88 - 0s - loss: 1.9533 - 325ms/epoch - 4ms/step
Epoch 84/150
88/88 - 0s - loss: 1.9511 - 322ms/epoch - 4ms/step
Epoch 85/150
88/88 - 0s - loss: 1.9555 - 322ms/epoch - 4ms/step
Epoch 86/150
88/88 - 0s - loss: 1.9519 - 321ms/epoch - 4ms/step
Epoch 87/150
88/88 - 0s - loss: 1.9543 - 323ms/epoch - 4ms/step
Epoch 88/150
88/88 - 0s - loss: 1.9518 - 321ms/epoch - 4ms/step
Epoch 89/150
88/88 - 0s - loss: 1.9504 - 321ms/epoch - 4ms/step
Epoch 90/150
88/88 - 0s - loss: 1.9504 - 319ms/epoch - 4ms/step
Epoch 91/150
88/88 - 0s - loss: 1.9544 - 320ms/epoch - 4ms/step
Epoch 92/150
```

```
88/88 - 0s - loss: 1.9496 - 324ms/epoch - 4ms/step
Epoch 93/150
88/88 - 0s - loss: 1.9509 - 324ms/epoch - 4ms/step
Epoch 94/150
88/88 - 0s - loss: 1.9511 - 317ms/epoch - 4ms/step
Epoch 95/150
88/88 - 0s - loss: 1.9482 - 307ms/epoch - 3ms/step
Epoch 96/150
88/88 - 0s - loss: 1.9497 - 306ms/epoch - 3ms/step
Epoch 97/150
88/88 - 0s - loss: 1.9503 - 307ms/epoch - 3ms/step
Epoch 98/150
88/88 - 0s - loss: 1.9497 - 306ms/epoch - 3ms/step
Epoch 99/150
88/88 - 0s - loss: 1.9507 - 300ms/epoch - 3ms/step
Epoch 100/150
88/88 - 0s - loss: 1.9493 - 321ms/epoch - 4ms/step
Epoch 101/150
88/88 - 0s - loss: 1.9496 - 318ms/epoch - 4ms/step
Epoch 102/150
88/88 - 0s - loss: 1.9484 - 324ms/epoch - 4ms/step
Epoch 103/150
88/88 - 0s - loss: 1.9483 - 322ms/epoch - 4ms/step
Epoch 104/150
88/88 - 0s - loss: 1.9488 - 321ms/epoch - 4ms/step
Epoch 105/150
88/88 - 0s - loss: 1.9491 - 318ms/epoch - 4ms/step
Epoch 106/150
88/88 - 0s - loss: 1.9482 - 319ms/epoch - 4ms/step
Epoch 107/150
88/88 - 0s - loss: 1.9460 - 321ms/epoch - 4ms/step
Epoch 108/150
88/88 - 0s - loss: 1.9508 - 320ms/epoch - 4ms/step
Epoch 109/150
88/88 - 0s - loss: 1.9505 - 319ms/epoch - 4ms/step
Epoch 110/150
88/88 - 0s - loss: 1.9493 - 317ms/epoch - 4ms/step
Epoch 111/150
88/88 - 0s - loss: 1.9478 - 329ms/epoch - 4ms/step
Epoch 112/150
88/88 - 0s - loss: 1.9502 - 335ms/epoch - 4ms/step
Epoch 113/150
88/88 - 0s - loss: 1.9498 - 338ms/epoch - 4ms/step
Epoch 114/150
88/88 - 0s - loss: 1.9489 - 327ms/epoch - 4ms/step
Epoch 115/150
88/88 - 0s - loss: 1.9510 - 322ms/epoch - 4ms/step
```

```
Epoch 116/150
88/88 - 0s - loss: 1.9495 - 320ms/epoch - 4ms/step
Epoch 117/150
88/88 - 0s - loss: 1.9490 - 319ms/epoch - 4ms/step
Epoch 118/150
88/88 - 0s - loss: 1.9489 - 326ms/epoch - 4ms/step
Epoch 119/150
88/88 - 0s - loss: 1.9485 - 320ms/epoch - 4ms/step
Epoch 120/150
88/88 - 0s - loss: 1.9478 - 320ms/epoch - 4ms/step
Epoch 121/150
88/88 - 0s - loss: 1.9479 - 321ms/epoch - 4ms/step
Epoch 122/150
88/88 - 0s - loss: 1.9481 - 326ms/epoch - 4ms/step
Epoch 123/150
88/88 - 0s - loss: 1.9480 - 321ms/epoch - 4ms/step
Epoch 124/150
88/88 - 0s - loss: 1.9511 - 317ms/epoch - 4ms/step
Epoch 125/150
88/88 - 0s - loss: 1.9503 - 320ms/epoch - 4ms/step
Epoch 126/150
88/88 - 0s - loss: 1.9486 - 323ms/epoch - 4ms/step
Epoch 127/150
88/88 - 0s - loss: 1.9497 - 327ms/epoch - 4ms/step
Epoch 128/150
88/88 - 0s - loss: 1.9478 - 326ms/epoch - 4ms/step
Epoch 129/150
88/88 - 0s - loss: 1.9499 - 319ms/epoch - 4ms/step
Epoch 130/150
88/88 - 0s - loss: 1.9457 - 323ms/epoch - 4ms/step
Epoch 131/150
88/88 - 0s - loss: 1.9466 - 320ms/epoch - 4ms/step
Epoch 132/150
88/88 - 0s - loss: 1.9450 - 318ms/epoch - 4ms/step
Epoch 133/150
88/88 - 0s - loss: 1.9480 - 323ms/epoch - 4ms/step
Epoch 134/150
88/88 - 0s - loss: 1.9487 - 318ms/epoch - 4ms/step
Epoch 135/150
88/88 - 0s - loss: 1.9484 - 320ms/epoch - 4ms/step
Epoch 136/150
88/88 - 0s - loss: 1.9477 - 319ms/epoch - 4ms/step
Epoch 137/150
88/88 - 0s - loss: 1.9452 - 321ms/epoch - 4ms/step
Epoch 138/150
88/88 - 0s - loss: 1.9497 - 322ms/epoch - 4ms/step
Epoch 139/150
```

```
88/88 - 0s - loss: 1.9481 - 323ms/epoch - 4ms/step
Epoch 140/150
88/88 - 0s - loss: 1.9456 - 321ms/epoch - 4ms/step
Epoch 141/150
88/88 - 0s - loss: 1.9478 - 306ms/epoch - 3ms/step
Epoch 142/150
88/88 - 0s - loss: 1.9478 - 319ms/epoch - 4ms/step
Epoch 143/150
88/88 - 0s - loss: 1.9447 - 321ms/epoch - 4ms/step
Epoch 144/150
88/88 - 0s - loss: 1.9460 - 320ms/epoch - 4ms/step
Epoch 145/150
88/88 - 0s - loss: 1.9466 - 320ms/epoch - 4ms/step
Epoch 146/150
88/88 - 0s - loss: 1.9464 - 320ms/epoch - 4ms/step
Epoch 147/150
88/88 - 0s - loss: 1.9458 - 321ms/epoch - 4ms/step
Epoch 148/150
88/88 - 0s - loss: 1.9453 - 322ms/epoch - 4ms/step
Epoch 149/150
88/88 - 0s - loss: 1.9434 - 317ms/epoch - 4ms/step
Epoch 150/150
88/88 - 0s - loss: 1.9438 - 322ms/epoch - 4ms/step
```

We now move on to the evaluation step. The trained model can now be applied onto the Testing Dataset to predict the number of rings.

```
    # evaluate on test set
    yhat = model.predict(X_test)
    yhat = argmax(yhat, axis=-1).astype('int')
    acc = accuracy_score(y_test, yhat,normalize=False)
    print('Number of test datapoints: ', len(y_test))
    print('Accuracy: %.3f' % acc)
```

Listing B.16: Sequential Classification: Predicting values using testing data

```
44/44 [==============================] - 0s 2ms/step
Number of test datapoints:  1379
Accuracy: 368.000
```

This shows that out of the 1379 data points of the testing dataset, only 368 have been classified correctly, giving an accuracy of approximately 27%.

B.1.3 Functional Implementation

The goal of this Jupyter Notebook is to combine both of the individual approaches described up to this point into one single network.

We begin this chapter first by importing all of the relevant libraries. The libraries required here are a composition of all of the libraries used for both the regression as well as the classification tasks:

```
    # load and summarize the abalone dataset

    # For data preprocessing
    from pandas import read_csv

    # For Calculations
    import numpy as np
    from numpy import unique
    from numpy import argmax

    # Tensorflow, for neural network
    from tensorflow.keras.models import Model
    from tensorflow.keras.layers import Input
    from tensorflow.keras.layers import Dense

    # for post processing and splitting data
    from sklearn.metrics import mean_absolute_error
    from sklearn.metrics import mean_squared_error
    from sklearn.metrics import accuracy_score
    from sklearn.model_selection import train_test_split
    from sklearn.preprocessing import LabelEncoder

    # Source of the Dataset
    from ucimlrepo import fetch_ucirepo
```

Listing B.17: Abalone, Classification Relevant Libraries

Importing the data (Listing B.2) and pre-processing (Listing B.3) it are done in an identical manner as with the regression task, therefore the code shall not be repeated here. The reader is referred to the Listings mentioned.

Once again, splitting the data works slightly differently here. Here, the regression task is combined with the classification task - so, one target needs to be defined for each task, resulting in a duplicate of the 'y' variable for the training and testing dataset (e.g. y_train and y_train_class) :

```
    # split data into train and test sets
    X_train, X_test, y_train, y_test, y_train_class, y_test_class =
    train_test_split(X, y, y_class, test_size=0.33, random_state=1)
    print('Size of X_train is ', len(X_train))
    print('Size of X_test is ', len(X_test))
    print('Size of y_train is ', len(y_train))
    print('Size of y_test is ', len(y_test))
    print('Size of y_train_class is ', len(y_train_class))
    print('Size of y_test_class is ', len(y_test_class))
```

Listing B.18: Split 3 data componenets into training and testing datasets

```
Size of X_train is  2798
Size of X_test is  1379
Size of y_train is  2798
Size of y_test is  1379
Size of y_train_class is  2798
Size of y_test_class is  1379
```

Once data is processed and the datasets prepared, the model may be defined:

```
visible = Input(shape=(n_features,))
hidden1 = Dense(20, activation='relu', kernel_initializer='he_normal')(visible)
hidden2 = Dense(10, activation='relu', kernel_initializer='he_normal')(hidden1)

# regression output: layer with a single node and a linear activation function
out_reg = Dense(1, activation='linear')(hidden2)

# classification output: layer with number of nodes = number of classes and a softmax activation function
out_class = Dense(n_class, activation='softmax')(hidden2)
```

Listing B.19: Functional Implementation: Defining the Model

An the model summary can now be viewed as well:

```
model.summary()
```

Listing B.20: Functional Implementation: Model Summary

```
Model: "model"
__________________________________________________________________________________
 Layer (type)           Output Shape    Param #   Connected to
==================================================================================
 input_1 (InputLayer)   [(None, 7)]     0         []

 dense (Dense)          (None, 20)      160       ['input_1[0][0]']

 dense_1 (Dense)        (None, 10)      210       ['dense[0][0]']

 dense_2 (Dense)        (None, 1)       11        ['dense_1[0][0]']

 dense_3 (Dense)        (None, 28)      308       ['dense_1[0][0]']

==================================================================================
Total params: 689 (2.69 KB)
Trainable params: 689 (2.69 KB)
Non-trainable params: 0 (0.00 Byte)
__________________________________________________________________________________
```

Explanations to the architecture above have been provided in the thesis, for example in Chapter 6.2.4.

The next step is to compile the model and then to train it using the training dataset:

```
1    model.compile(loss=['mse','sparse_categorical_crossentropy'], optimizer='
     adam', metrics = ["mean_absolute_error"])
2
3    model.fit(X_train, [y_train,y_train_class], epochs=150, batch_size=32,
     verbose=2)
```

Listing B.21: Functional Implementation: Compiling and Training Model

```
Epoch 1/150
88/88 - 1s - loss: 62.2426 - dense_6_loss: 58.7064 -
dense_7_loss: 3.5362 - dense_6_mean_absolute_error: 7.0018
- dense_7_mean_absolute_error: 8.8925 - 959ms/epoch - 11ms/step
Epoch 2/150
88/88 - 1s - loss: 18.2585 - dense_6_loss: 15.5085
- dense_7_loss: 2.7500 - dense_6_mean_absolute_error: 3.0814
- dense_7_mean_absolute_error: 8.8925 - 501ms/epoch - 6ms/step
Epoch 3/150
88/88 - 0s - loss: 13.2946 - dense_6_loss: 10.7651 -
dense_7_loss: 2.5295 - dense_6_mean_absolute_error: 2.4799
- dense_7_mean_absolute_error: 8.8925 - 490ms/epoch - 6ms/step
Epoch 4/150
88/88 - 1s - loss: 12.5165 - dense_6_loss: 10.0305 -
dense_7_loss: 2.4860 - dense_6_mean_absolute_error: 2.3655
- dense_7_mean_absolute_error: 8.8925 - 506ms/epoch - 6ms/step
Epoch 5/150
88/88 - 1s - loss: 11.7552 - dense_6_loss: 9.2962 -
dense_7_loss: 2.4591 - dense_6_mean_absolute_error: 2.2400
- dense_7_mean_absolute_error: 8.8925 - 505ms/epoch - 6ms/step
Epoch 6/150
88/88 - 0s - loss: 11.0331 - dense_6_loss: 8.5985 -
dense_7_loss: 2.4346 - dense_6_mean_absolute_error: 2.1247
- dense_7_mean_absolute_error: 8.8925 - 485ms/epoch - 6ms/step
Epoch 7/150
88/88 - 0s - loss: 10.3740 - dense_6_loss: 7.9611 -
dense_7_loss: 2.4129 - dense_6_mean_absolute_error: 2.0217
- dense_7_mean_absolute_error: 8.8925 - 495ms/epoch - 6ms/step
Epoch 8/150
88/88 - 0s - loss: 9.8612 - dense_6_loss: 7.4724 -
dense_7_loss: 2.3887 - dense_6_mean_absolute_error: 1.9395
- dense_7_mean_absolute_error: 8.8925 - 493ms/epoch - 6ms/step
Epoch 9/150
88/88 - 1s - loss: 9.4501 - dense_6_loss: 7.0813 -
dense_7_loss: 2.3688 - dense_6_mean_absolute_error: 1.8977
```

```
- dense_7_mean_absolute_error: 8.8925 - 504ms/epoch - 6ms/step
Epoch 10/150
88/88 - 0s - loss: 9.1548 - dense_6_loss: 6.8074 -
dense_7_loss: 2.3474 - dense_6_mean_absolute_error: 1.8820
- dense_7_mean_absolute_error: 8.8925 - 493ms/epoch - 6ms/step
Epoch 11/150
88/88 - 0s - loss: 8.9825 - dense_6_loss: 6.6513 -
dense_7_loss: 2.3312 - dense_6_mean_absolute_error: 1.8736
- dense_7_mean_absolute_error: 8.8925 - 494ms/epoch - 6ms/step
Epoch 12/150
88/88 - 0s - loss: 8.8824 - dense_6_loss: 6.5658 -
dense_7_loss: 2.3166 - dense_6_mean_absolute_error: 1.8621
- dense_7_mean_absolute_error: 8.8925 - 495ms/epoch - 6ms/step
Epoch 13/150
88/88 - 1s - loss: 8.7931 - dense_6_loss: 6.4860 -
dense_7_loss: 2.3070 - dense_6_mean_absolute_error: 1.8701
- dense_7_mean_absolute_error: 8.8925 - 503ms/epoch - 6ms/step
Epoch 14/150
88/88 - 1s - loss: 8.7137 - dense_6_loss: 6.4145 -
dense_7_loss: 2.2992 - dense_6_mean_absolute_error: 1.8442
- dense_7_mean_absolute_error: 8.8925 - 509ms/epoch - 6ms/step
Epoch 15/150
88/88 - 0s - loss: 8.6368 - dense_6_loss: 6.3476 -
dense_7_loss: 2.2892 - dense_6_mean_absolute_error: 1.8435
- dense_7_mean_absolute_error: 8.8925 - 494ms/epoch - 6ms/step
Epoch 16/150
88/88 - 1s - loss: 8.5597 - dense_6_loss: 6.2806 -
dense_7_loss: 2.2791 - dense_6_mean_absolute_error: 1.8314
- dense_7_mean_absolute_error: 8.8925 - 503ms/epoch - 6ms/step
Epoch 17/150
88/88 - 1s - loss: 8.4781 - dense_6_loss: 6.2119 -
dense_7_loss: 2.2662 - dense_6_mean_absolute_error: 1.8250
- dense_7_mean_absolute_error: 8.8925 - 509ms/epoch - 6ms/step
Epoch 18/150
88/88 - 0s - loss: 8.4302 - dense_6_loss: 6.1604 -
dense_7_loss: 2.2698 - dense_6_mean_absolute_error: 1.8035
- dense_7_mean_absolute_error: 8.8925 - 492ms/epoch - 6ms/step
Epoch 19/150
88/88 - 0s - loss: 8.3238 - dense_6_loss: 6.0786 -
dense_7_loss: 2.2452 - dense_6_mean_absolute_error: 1.7872
- dense_7_mean_absolute_error: 8.8925 - 482ms/epoch - 5ms/step
Epoch 20/150
88/88 - 1s - loss: 8.2313 - dense_6_loss: 6.0018 -
dense_7_loss: 2.2295 - dense_6_mean_absolute_error: 1.7830
- dense_7_mean_absolute_error: 8.8925 - 506ms/epoch - 6ms/step
Epoch 21/150
88/88 - 0s - loss: 8.1557 - dense_6_loss: 5.9355 -
```

```
dense_7_loss: 2.2203 - dense_6_mean_absolute_error: 1.7790
- dense_7_mean_absolute_error: 8.8925 - 500ms/epoch - 6ms/step
Epoch 22/150
88/88 - 1s - loss: 8.0937 - dense_6_loss: 5.8786 -
dense_7_loss: 2.2152 - dense_6_mean_absolute_error: 1.7628
- dense_7_mean_absolute_error: 8.8925 - 529ms/epoch - 6ms/step
Epoch 23/150
88/88 - 0s - loss: 8.0283 - dense_6_loss: 5.8176 -
dense_7_loss: 2.2107 - dense_6_mean_absolute_error: 1.7554
- dense_7_mean_absolute_error: 8.8925 - 487ms/epoch - 6ms/step
Epoch 24/150
88/88 - 0s - loss: 7.9405 - dense_6_loss: 5.7319 -
dense_7_loss: 2.2086 - dense_6_mean_absolute_error: 1.7455
- dense_7_mean_absolute_error: 8.8925 - 497ms/epoch - 6ms/step
Epoch 25/150
88/88 - 0s - loss: 7.9303 - dense_6_loss: 5.7242 -
dense_7_loss: 2.2062 - dense_6_mean_absolute_error: 1.7377
- dense_7_mean_absolute_error: 8.8925 - 489ms/epoch - 6ms/step
Epoch 26/150
88/88 - 0s - loss: 7.8357 - dense_6_loss: 5.6306 -
dense_7_loss: 2.2051 - dense_6_mean_absolute_error: 1.7204
- dense_7_mean_absolute_error: 8.8925 - 490ms/epoch - 6ms/step
Epoch 27/150
88/88 - 0s - loss: 7.7748 - dense_6_loss: 5.5704 -
dense_7_loss: 2.2043 - dense_6_mean_absolute_error: 1.7225
- dense_7_mean_absolute_error: 8.8925 - 499ms/epoch - 6ms/step
Epoch 28/150
88/88 - 0s - loss: 7.7349 - dense_6_loss: 5.5329 -
dense_7_loss: 2.2020 - dense_6_mean_absolute_error: 1.7106
- dense_7_mean_absolute_error: 8.8925 - 494ms/epoch - 6ms/step
Epoch 29/150
88/88 - 0s - loss: 7.6900 - dense_6_loss: 5.4896 -
dense_7_loss: 2.2004 - dense_6_mean_absolute_error: 1.6974
- dense_7_mean_absolute_error: 8.8925 - 490ms/epoch - 6ms/step
Epoch 30/150
88/88 - 0s - loss: 7.6241 - dense_6_loss: 5.4246 -
dense_7_loss: 2.1995 - dense_6_mean_absolute_error: 1.6951
- dense_7_mean_absolute_error: 8.8925 - 487ms/epoch - 6ms/step
Epoch 31/150
88/88 - 0s - loss: 7.5862 - dense_6_loss: 5.3875 -
dense_7_loss: 2.1986 - dense_6_mean_absolute_error: 1.6838
- dense_7_mean_absolute_error: 8.8925 - 496ms/epoch - 6ms/step
Epoch 32/150
88/88 - 0s - loss: 7.5552 - dense_6_loss: 5.3580 -
dense_7_loss: 2.1972 - dense_6_mean_absolute_error: 1.6655
- dense_7_mean_absolute_error: 8.8925 - 500ms/epoch - 6ms/step
Epoch 33/150
```

```
88/88 - 0s - loss: 7.5320 - dense_6_loss: 5.3361 -
dense_7_loss: 2.1959 - dense_6_mean_absolute_error: 1.6879
- dense_7_mean_absolute_error: 8.8925 - 490ms/epoch - 6ms/step
Epoch 34/150
88/88 - 1s - loss: 7.4764 - dense_6_loss: 5.2834 -
dense_7_loss: 2.1929 - dense_6_mean_absolute_error: 1.6696
- dense_7_mean_absolute_error: 8.8925 - 504ms/epoch - 6ms/step
Epoch 35/150
88/88 - 0s - loss: 7.4615 - dense_6_loss: 5.2700 -
dense_7_loss: 2.1915 - dense_6_mean_absolute_error: 1.6589
- dense_7_mean_absolute_error: 8.8925 - 492ms/epoch - 6ms/step
Epoch 36/150
88/88 - 0s - loss: 7.4059 - dense_6_loss: 5.2157 -
dense_7_loss: 2.1902 - dense_6_mean_absolute_error: 1.6615
- dense_7_mean_absolute_error: 8.8925 - 494ms/epoch - 6ms/step
Epoch 37/150
88/88 - 0s - loss: 7.3808 - dense_6_loss: 5.1944 -
dense_7_loss: 2.1864 - dense_6_mean_absolute_error: 1.6452
- dense_7_mean_absolute_error: 8.8925 - 487ms/epoch - 6ms/step
Epoch 38/150
88/88 - 0s - loss: 7.3999 - dense_6_loss: 5.2132 -
dense_7_loss: 2.1866 - dense_6_mean_absolute_error: 1.6585
- dense_7_mean_absolute_error: 8.8925 - 494ms/epoch - 6ms/step
Epoch 39/150
88/88 - 0s - loss: 7.3227 - dense_6_loss: 5.1395 -
dense_7_loss: 2.1831 - dense_6_mean_absolute_error: 1.6462
- dense_7_mean_absolute_error: 8.8925 - 493ms/epoch - 6ms/step
Epoch 40/150
88/88 - 0s - loss: 7.3134 - dense_6_loss: 5.1335 -
dense_7_loss: 2.1799 - dense_6_mean_absolute_error: 1.6421
- dense_7_mean_absolute_error: 8.8925 - 490ms/epoch - 6ms/step
Epoch 41/150
88/88 - 0s - loss: 7.3000 - dense_6_loss: 5.1218 -
dense_7_loss: 2.1782 - dense_6_mean_absolute_error: 1.6425
- dense_7_mean_absolute_error: 8.8925 - 494ms/epoch - 6ms/step
Epoch 42/150
88/88 - 0s - loss: 7.2781 - dense_6_loss: 5.1041 -
dense_7_loss: 2.1741 - dense_6_mean_absolute_error: 1.6354
- dense_7_mean_absolute_error: 8.8925 - 488ms/epoch - 6ms/step
Epoch 43/150
88/88 - 0s - loss: 7.2631 - dense_6_loss: 5.0897 -
dense_7_loss: 2.1734 - dense_6_mean_absolute_error: 1.6381
- dense_7_mean_absolute_error: 8.8925 - 489ms/epoch - 6ms/step
Epoch 44/150
88/88 - 0s - loss: 7.2594 - dense_6_loss: 5.0902 -
dense_7_loss: 2.1692 - dense_6_mean_absolute_error: 1.6301
- dense_7_mean_absolute_error: 8.8925 - 493ms/epoch - 6ms/step
```

```
Epoch 45/150
88/88 - 0s - loss: 7.2324 - dense_6_loss: 5.0667 -
dense_7_loss: 2.1658 - dense_6_mean_absolute_error: 1.6355
- dense_7_mean_absolute_error: 8.8925 - 492ms/epoch - 6ms/step
Epoch 46/150
88/88 - 0s - loss: 7.2548 - dense_6_loss: 5.0883 -
dense_7_loss: 2.1664 - dense_6_mean_absolute_error: 1.6256
- dense_7_mean_absolute_error: 8.8925 - 485ms/epoch - 6ms/step
Epoch 47/150
88/88 - 0s - loss: 7.2268 - dense_6_loss: 5.0598 -
dense_7_loss: 2.1670 - dense_6_mean_absolute_error: 1.6295
- dense_7_mean_absolute_error: 8.8925 - 491ms/epoch - 6ms/step
Epoch 48/150
88/88 - 0s - loss: 7.2078 - dense_6_loss: 5.0441 -
dense_7_loss: 2.1637 - dense_6_mean_absolute_error: 1.6238
- dense_7_mean_absolute_error: 8.8925 - 489ms/epoch - 6ms/step
Epoch 49/150
88/88 - 0s - loss: 7.2052 - dense_6_loss: 5.0443 -
dense_7_loss: 2.1609 - dense_6_mean_absolute_error: 1.6226
- dense_7_mean_absolute_error: 8.8925 - 491ms/epoch - 6ms/step
Epoch 50/150
88/88 - 0s - loss: 7.1748 - dense_6_loss: 5.0193 -
dense_7_loss: 2.1555 - dense_6_mean_absolute_error: 1.6301
- dense_7_mean_absolute_error: 8.8925 - 493ms/epoch - 6ms/step
Epoch 51/150
88/88 - 0s - loss: 7.2000 - dense_6_loss: 5.0475 -
dense_7_loss: 2.1525 - dense_6_mean_absolute_error: 1.6217
- dense_7_mean_absolute_error: 8.8925 - 489ms/epoch - 6ms/step
Epoch 52/150
88/88 - 0s - loss: 7.1813 - dense_6_loss: 5.0324 -
dense_7_loss: 2.1489 - dense_6_mean_absolute_error: 1.6270
- dense_7_mean_absolute_error: 8.8925 - 484ms/epoch - 6ms/step
Epoch 53/150
88/88 - 0s - loss: 7.2036 - dense_6_loss: 5.0581 -
dense_7_loss: 2.1455 - dense_6_mean_absolute_error: 1.6284
- dense_7_mean_absolute_error: 8.8925 - 498ms/epoch - 6ms/step
Epoch 54/150
88/88 - 0s - loss: 7.2445 - dense_6_loss: 5.1015 -
dense_7_loss: 2.1429 - dense_6_mean_absolute_error: 1.6278
- dense_7_mean_absolute_error: 8.8925 - 481ms/epoch - 5ms/step
Epoch 55/150
88/88 - 0s - loss: 7.1770 - dense_6_loss: 5.0405 -
dense_7_loss: 2.1365 - dense_6_mean_absolute_error: 1.6315
- dense_7_mean_absolute_error: 8.8925 - 487ms/epoch - 6ms/step
Epoch 56/150
88/88 - 0s - loss: 7.1524 - dense_6_loss: 5.0191 -
dense_7_loss: 2.1333 - dense_6_mean_absolute_error: 1.6188
```

```
- dense_7_mean_absolute_error: 8.8925 - 488ms/epoch - 6ms/step
Epoch 57/150
88/88 - 0s - loss: 7.1399 - dense_6_loss: 5.0098 -
dense_7_loss: 2.1301 - dense_6_mean_absolute_error: 1.6257
- dense_7_mean_absolute_error: 8.8925 - 486ms/epoch - 6ms/step
Epoch 58/150
88/88 - 0s - loss: 7.1457 - dense_6_loss: 5.0207 -
dense_7_loss: 2.1250 - dense_6_mean_absolute_error: 1.6221
- dense_7_mean_absolute_error: 8.8925 - 488ms/epoch - 6ms/step
Epoch 59/150
88/88 - 0s - loss: 7.1477 - dense_6_loss: 5.0253 -
dense_7_loss: 2.1224 - dense_6_mean_absolute_error: 1.6226
- dense_7_mean_absolute_error: 8.8925 - 493ms/epoch - 6ms/step
Epoch 60/150
88/88 - 0s - loss: 7.1324 - dense_6_loss: 5.0143 -
dense_7_loss: 2.1181 - dense_6_mean_absolute_error: 1.6174
- dense_7_mean_absolute_error: 8.8925 - 489ms/epoch - 6ms/step
Epoch 61/150
88/88 - 0s - loss: 7.1262 - dense_6_loss: 5.0117 -
dense_7_loss: 2.1145 - dense_6_mean_absolute_error: 1.6217
- dense_7_mean_absolute_error: 8.8925 - 494ms/epoch - 6ms/step
Epoch 62/150
88/88 - 0s - loss: 7.1305 - dense_6_loss: 5.0196 -
dense_7_loss: 2.1110 - dense_6_mean_absolute_error: 1.6218
- dense_7_mean_absolute_error: 8.8925 - 489ms/epoch - 6ms/step
Epoch 63/150
88/88 - 0s - loss: 7.1171 - dense_6_loss: 5.0080 -
dense_7_loss: 2.1091 - dense_6_mean_absolute_error: 1.6238
- dense_7_mean_absolute_error: 8.8925 - 487ms/epoch - 6ms/step
Epoch 64/150
88/88 - 0s - loss: 7.1201 - dense_6_loss: 5.0164 -
dense_7_loss: 2.1037 - dense_6_mean_absolute_error: 1.6146
- dense_7_mean_absolute_error: 8.8925 - 492ms/epoch - 6ms/step
Epoch 65/150
88/88 - 0s - loss: 7.1467 - dense_6_loss: 5.0437 -
dense_7_loss: 2.1030 - dense_6_mean_absolute_error: 1.6243
- dense_7_mean_absolute_error: 8.8925 - 486ms/epoch - 6ms/step
Epoch 66/150
88/88 - 0s - loss: 7.1008 - dense_6_loss: 5.0017 -
dense_7_loss: 2.0991 - dense_6_mean_absolute_error: 1.6182
- dense_7_mean_absolute_error: 8.8925 - 487ms/epoch - 6ms/step
Epoch 67/150
88/88 - 0s - loss: 7.1052 - dense_6_loss: 5.0104 -
dense_7_loss: 2.0948 - dense_6_mean_absolute_error: 1.6191
- dense_7_mean_absolute_error: 8.8925 - 492ms/epoch - 6ms/step
Epoch 68/150
88/88 - 0s - loss: 7.0890 - dense_6_loss: 4.9962 -
```

```
dense_7_loss: 2.0928 - dense_6_mean_absolute_error: 1.6153
- dense_7_mean_absolute_error: 8.8925 - 484ms/epoch - 5ms/step
Epoch 69/150
88/88 - 0s - loss: 7.0887 - dense_6_loss: 5.0007 - dense_7_loss: 2.0880 - dense_6_mean_
Epoch 70/150
88/88 - 0s - loss: 7.0769 - dense_6_loss: 4.9924 -
dense_7_loss: 2.0845 - dense_6_mean_absolute_error: 1.6228
- dense_7_mean_absolute_error: 8.8925 - 486ms/epoch - 6ms/step
Epoch 71/150
88/88 - 0s - loss: 7.0877 - dense_6_loss: 5.0055 -
dense_7_loss: 2.0822 - dense_6_mean_absolute_error: 1.6215
- dense_7_mean_absolute_error: 8.8925 - 484ms/epoch - 6ms/step
Epoch 72/150
88/88 - 0s - loss: 7.1350 - dense_6_loss: 5.0540 -
dense_7_loss: 2.0810 - dense_6_mean_absolute_error: 1.6315
- dense_7_mean_absolute_error: 8.8925 - 486ms/epoch - 6ms/step
Epoch 73/150
88/88 - 0s - loss: 7.1068 - dense_6_loss: 5.0279 -
dense_7_loss: 2.0790 - dense_6_mean_absolute_error: 1.6312
- dense_7_mean_absolute_error: 8.8925 - 483ms/epoch - 5ms/step
Epoch 74/150
88/88 - 0s - loss: 7.0804 - dense_6_loss: 5.0043 -
dense_7_loss: 2.0761 - dense_6_mean_absolute_error: 1.6218
- dense_7_mean_absolute_error: 8.8925 - 486ms/epoch - 6ms/step
Epoch 75/150
88/88 - 0s - loss: 7.0867 - dense_6_loss: 5.0166 -
dense_7_loss: 2.0701 - dense_6_mean_absolute_error: 1.6235
- dense_7_mean_absolute_error: 8.8925 - 487ms/epoch - 6ms/step
Epoch 76/150
88/88 - 0s - loss: 7.0712 - dense_6_loss: 5.0030 -
dense_7_loss: 2.0682 - dense_6_mean_absolute_error: 1.6193
- dense_7_mean_absolute_error: 8.8925 - 488ms/epoch - 6ms/step
Epoch 77/150
88/88 - 0s - loss: 7.0679 - dense_6_loss: 5.0011 -
dense_7_loss: 2.0668 - dense_6_mean_absolute_error: 1.6143
- dense_7_mean_absolute_error: 8.8925 - 488ms/epoch - 6ms/step
Epoch 78/150
88/88 - 0s - loss: 7.0981 - dense_6_loss: 5.0352 -
dense_7_loss: 2.0629 - dense_6_mean_absolute_error: 1.6295
- dense_7_mean_absolute_error: 8.8925 - 485ms/epoch - 6ms/step
Epoch 79/150
88/88 - 0s - loss: 7.0747 - dense_6_loss: 5.0138 -
dense_7_loss: 2.0609 - dense_6_mean_absolute_error: 1.6173
- dense_7_mean_absolute_error: 8.8925 - 490ms/epoch - 6ms/step
Epoch 80/150
88/88 - 0s - loss: 7.0648 - dense_6_loss: 5.0057 -
dense_7_loss: 2.0591 - dense_6_mean_absolute_error: 1.6238
```

```
- dense_7_mean_absolute_error: 8.8925 - 492ms/epoch - 6ms/step
Epoch 81/150
88/88 - 0s - loss: 7.0911 - dense_6_loss: 5.0336 -
dense_7_loss: 2.0575 - dense_6_mean_absolute_error: 1.6305
- dense_7_mean_absolute_error: 8.8925 - 484ms/epoch - 6ms/step
Epoch 82/150
88/88 - 0s - loss: 7.0851 - dense_6_loss: 5.0297 -
dense_7_loss: 2.0554 - dense_6_mean_absolute_error: 1.6223
- dense_7_mean_absolute_error: 8.8925 - 494ms/epoch - 6ms/step
Epoch 83/150
88/88 - 0s - loss: 7.0587 - dense_6_loss: 5.0071 -
dense_7_loss: 2.0516 - dense_6_mean_absolute_error: 1.6230
- dense_7_mean_absolute_error: 8.8925 - 494ms/epoch - 6ms/step
Epoch 84/150
88/88 - 1s - loss: 7.0419 - dense_6_loss: 4.9910 -
dense_7_loss: 2.0509 - dense_6_mean_absolute_error: 1.6129
- dense_7_mean_absolute_error: 8.8925 - 514ms/epoch - 6ms/step
Epoch 85/150
88/88 - 1s - loss: 7.0356 - dense_6_loss: 4.9880 -
dense_7_loss: 2.0476 - dense_6_mean_absolute_error: 1.6187
- dense_7_mean_absolute_error: 8.8925 - 531ms/epoch - 6ms/step
Epoch 86/150
88/88 - 0s - loss: 7.0233 - dense_6_loss: 4.9772 -
dense_7_loss: 2.0461 - dense_6_mean_absolute_error: 1.6135
- dense_7_mean_absolute_error: 8.8925 - 495ms/epoch - 6ms/step
Epoch 87/150
88/88 - 0s - loss: 7.0477 - dense_6_loss: 5.0022 -
dense_7_loss: 2.0455 - dense_6_mean_absolute_error: 1.6214
- dense_7_mean_absolute_error: 8.8925 - 495ms/epoch - 6ms/step
Epoch 88/150
88/88 - 1s - loss: 7.0505 - dense_6_loss: 5.0066 -
dense_7_loss: 2.0439 - dense_6_mean_absolute_error: 1.6158
- dense_7_mean_absolute_error: 8.8925 - 506ms/epoch - 6ms/step
Epoch 89/150
88/88 - 1s - loss: 7.0402 - dense_6_loss: 4.9984 -
dense_7_loss: 2.0418 - dense_6_mean_absolute_error: 1.6205
- dense_7_mean_absolute_error: 8.8925 - 510ms/epoch - 6ms/step
Epoch 90/150
88/88 - 1s - loss: 7.0161 - dense_6_loss: 4.9771 -
dense_7_loss: 2.0390 - dense_6_mean_absolute_error: 1.6097
- dense_7_mean_absolute_error: 8.8925 - 502ms/epoch - 6ms/step
Epoch 91/150
88/88 - 1s - loss: 7.0672 - dense_6_loss: 5.0303 -
dense_7_loss: 2.0369 - dense_6_mean_absolute_error: 1.6273
- dense_7_mean_absolute_error: 8.8925 - 524ms/epoch - 6ms/step
Epoch 92/150
88/88 - 1s - loss: 7.0144 - dense_6_loss: 4.9785 -
```

```
dense_7_loss: 2.0360 - dense_6_mean_absolute_error: 1.6159
- dense_7_mean_absolute_error: 8.8925 - 508ms/epoch - 6ms/step
Epoch 93/150
88/88 - 1s - loss: 7.0357 - dense_6_loss: 5.0037 -
dense_7_loss: 2.0320 - dense_6_mean_absolute_error: 1.6176
- dense_7_mean_absolute_error: 8.8925 - 508ms/epoch - 6ms/step
Epoch 94/150
88/88 - 0s - loss: 7.0199 - dense_6_loss: 4.9862 -
dense_7_loss: 2.0337 - dense_6_mean_absolute_error: 1.6218
- dense_7_mean_absolute_error: 8.8925 - 493ms/epoch - 6ms/step
Epoch 95/150
88/88 - 0s - loss: 7.0313 - dense_6_loss: 5.0021 -
dense_7_loss: 2.0292 - dense_6_mean_absolute_error: 1.6231
- dense_7_mean_absolute_error: 8.8925 - 491ms/epoch - 6ms/step
Epoch 96/150
88/88 - 0s - loss: 7.0279 - dense_6_loss: 4.9982 -
dense_7_loss: 2.0298 - dense_6_mean_absolute_error: 1.6193
- dense_7_mean_absolute_error: 8.8925 - 481ms/epoch - 5ms/step
Epoch 97/150
88/88 - 0s - loss: 7.0363 - dense_6_loss: 5.0081 -
dense_7_loss: 2.0283 - dense_6_mean_absolute_error: 1.6236
- dense_7_mean_absolute_error: 8.8925 - 482ms/epoch - 5ms/step
Epoch 98/150
88/88 - 0s - loss: 7.0471 - dense_6_loss: 5.0211 -
dense_7_loss: 2.0260 - dense_6_mean_absolute_error: 1.6237
- dense_7_mean_absolute_error: 8.8925 - 486ms/epoch - 6ms/step
Epoch 99/150
88/88 - 0s - loss: 7.0148 - dense_6_loss: 4.9899 -
dense_7_loss: 2.0249 - dense_6_mean_absolute_error: 1.6249
- dense_7_mean_absolute_error: 8.8925 - 482ms/epoch - 5ms/step
Epoch 100/150
88/88 - 0s - loss: 7.0282 - dense_6_loss: 5.0037 -
dense_7_loss: 2.0245 - dense_6_mean_absolute_error: 1.6199
- dense_7_mean_absolute_error: 8.8925 - 486ms/epoch - 6ms/step
Epoch 101/150
88/88 - 0s - loss: 7.0283 - dense_6_loss: 5.0055 -
dense_7_loss: 2.0228 - dense_6_mean_absolute_error: 1.6205
- dense_7_mean_absolute_error: 8.8925 - 489ms/epoch - 6ms/step
Epoch 102/150
88/88 - 0s - loss: 7.0185 - dense_6_loss: 4.9983 -
dense_7_loss: 2.0202 - dense_6_mean_absolute_error: 1.6234
- dense_7_mean_absolute_error: 8.8925 - 495ms/epoch - 6ms/step
Epoch 103/150
88/88 - 1s - loss: 7.0136 - dense_6_loss: 4.9927 -
dense_7_loss: 2.0209 - dense_6_mean_absolute_error: 1.6220
- dense_7_mean_absolute_error: 8.8925 - 504ms/epoch - 6ms/step
Epoch 104/150
```

```
88/88 - 0s - loss: 6.9966 - dense_6_loss: 4.9763 -
dense_7_loss: 2.0203 - dense_6_mean_absolute_error: 1.6208
- dense_7_mean_absolute_error: 8.8925 - 494ms/epoch - 6ms/step
Epoch 105/150
88/88 - 0s - loss: 6.9975 - dense_6_loss: 4.9801 -
dense_7_loss: 2.0174 - dense_6_mean_absolute_error: 1.6160
- dense_7_mean_absolute_error: 8.8925 - 488ms/epoch - 6ms/step
Epoch 106/150
88/88 - 0s - loss: 7.0332 - dense_6_loss: 5.0177 -
dense_7_loss: 2.0156 - dense_6_mean_absolute_error: 1.6234
- dense_7_mean_absolute_error: 8.8925 - 498ms/epoch - 6ms/step
Epoch 107/150
88/88 - 1s - loss: 7.0026 - dense_6_loss: 4.9862 -
dense_7_loss: 2.0164 - dense_6_mean_absolute_error: 1.6224
- dense_7_mean_absolute_error: 8.8925 - 510ms/epoch - 6ms/step
Epoch 108/150
88/88 - 1s - loss: 6.9762 - dense_6_loss: 4.9646 -
dense_7_loss: 2.0116 - dense_6_mean_absolute_error: 1.6138
- dense_7_mean_absolute_error: 8.8925 - 504ms/epoch - 6ms/step
Epoch 109/150
88/88 - 0s - loss: 6.9920 - dense_6_loss: 4.9809 -
dense_7_loss: 2.0111 - dense_6_mean_absolute_error: 1.6161
- dense_7_mean_absolute_error: 8.8925 - 493ms/epoch - 6ms/step
Epoch 110/150
88/88 - 0s - loss: 7.0023 - dense_6_loss: 4.9892 -
dense_7_loss: 2.0131 - dense_6_mean_absolute_error: 1.6213
- dense_7_mean_absolute_error: 8.8925 - 482ms/epoch - 5ms/step
Epoch 111/150
88/88 - 0s - loss: 6.9769 - dense_6_loss: 4.9673 -
dense_7_loss: 2.0097 - dense_6_mean_absolute_error: 1.6166
- dense_7_mean_absolute_error: 8.8925 - 490ms/epoch - 6ms/step
Epoch 112/150
88/88 - 0s - loss: 6.9908 - dense_6_loss: 4.9816 -
dense_7_loss: 2.0092 - dense_6_mean_absolute_error: 1.6114
- dense_7_mean_absolute_error: 8.8925 - 490ms/epoch - 6ms/step
Epoch 113/150
88/88 - 0s - loss: 7.0267 - dense_6_loss: 5.0177 -
dense_7_loss: 2.0090 - dense_6_mean_absolute_error: 1.6273
- dense_7_mean_absolute_error: 8.8925 - 493ms/epoch - 6ms/step
Epoch 114/150
88/88 - 0s - loss: 6.9834 - dense_6_loss: 4.9749 -
dense_7_loss: 2.0085 - dense_6_mean_absolute_error: 1.6239
- dense_7_mean_absolute_error: 8.8925 - 490ms/epoch - 6ms/step
Epoch 115/150
88/88 - 0s - loss: 7.0021 - dense_6_loss: 4.9944 -
dense_7_loss: 2.0077 - dense_6_mean_absolute_error: 1.6299
- dense_7_mean_absolute_error: 8.8925 - 494ms/epoch - 6ms/step
```

```
Epoch 116/150
88/88 - 0s - loss: 6.9843 - dense_6_loss: 4.9772 -
dense_7_loss: 2.0071 - dense_6_mean_absolute_error: 1.6133
- dense_7_mean_absolute_error: 8.8925 - 484ms/epoch - 6ms/step
Epoch 117/150
88/88 - 0s - loss: 7.0015 - dense_6_loss: 4.9967 -
dense_7_loss: 2.0048 - dense_6_mean_absolute_error: 1.6248
- dense_7_mean_absolute_error: 8.8925 - 485ms/epoch - 6ms/step
Epoch 118/150
88/88 - 0s - loss: 6.9906 - dense_6_loss: 4.9857 -
dense_7_loss: 2.0050 - dense_6_mean_absolute_error: 1.6169
- dense_7_mean_absolute_error: 8.8925 - 488ms/epoch - 6ms/step
Epoch 119/150
88/88 - 0s - loss: 6.9845 - dense_6_loss: 4.9813 -
dense_7_loss: 2.0032 - dense_6_mean_absolute_error: 1.6188
- dense_7_mean_absolute_error: 8.8925 - 494ms/epoch - 6ms/step
Epoch 120/150
88/88 - 0s - loss: 6.9669 - dense_6_loss: 4.9641 -
dense_7_loss: 2.0028 - dense_6_mean_absolute_error: 1.6163
- dense_7_mean_absolute_error: 8.8925 - 493ms/epoch - 6ms/step
Epoch 121/150
88/88 - 0s - loss: 6.9586 - dense_6_loss: 4.9570 -
dense_7_loss: 2.0016 - dense_6_mean_absolute_error: 1.6157
- dense_7_mean_absolute_error: 8.8925 - 488ms/epoch - 6ms/step
Epoch 122/150
88/88 - 0s - loss: 6.9880 - dense_6_loss: 4.9875 -
dense_7_loss: 2.0005 - dense_6_mean_absolute_error: 1.6143
- dense_7_mean_absolute_error: 8.8925 - 491ms/epoch - 6ms/step
Epoch 123/150
88/88 - 0s - loss: 6.9915 - dense_6_loss: 4.9917 -
dense_7_loss: 1.9998 - dense_6_mean_absolute_error: 1.6173
- dense_7_mean_absolute_error: 8.8925 - 482ms/epoch - 5ms/step
Epoch 124/150
88/88 - 0s - loss: 6.9687 - dense_6_loss: 4.9690 -
dense_7_loss: 1.9997 - dense_6_mean_absolute_error: 1.6225
- dense_7_mean_absolute_error: 8.8925 - 489ms/epoch - 6ms/step
Epoch 125/150
88/88 - 0s - loss: 6.9471 - dense_6_loss: 4.9480 -
dense_7_loss: 1.9990 - dense_6_mean_absolute_error: 1.6165
- dense_7_mean_absolute_error: 8.8925 - 490ms/epoch - 6ms/step
Epoch 126/150
88/88 - 0s - loss: 6.9498 - dense_6_loss: 4.9505 -
dense_7_loss: 1.9993 - dense_6_mean_absolute_error: 1.6173
- dense_7_mean_absolute_error: 8.8925 - 495ms/epoch - 6ms/step
Epoch 127/150
88/88 - 0s - loss: 6.9780 - dense_6_loss: 4.9813 -
dense_7_loss: 1.9967 - dense_6_mean_absolute_error: 1.6144
```

```
- dense_7_mean_absolute_error: 8.8925 - 497ms/epoch - 6ms/step
Epoch 128/150
88/88 - 0s - loss: 6.9568 - dense_6_loss: 4.9571 -
dense_7_loss: 1.9997 - dense_6_mean_absolute_error: 1.6190
- dense_7_mean_absolute_error: 8.8925 - 497ms/epoch - 6ms/step
Epoch 129/150
88/88 - 0s - loss: 6.9753 - dense_6_loss: 4.9808 -
dense_7_loss: 1.9946 - dense_6_mean_absolute_error: 1.6169
- dense_7_mean_absolute_error: 8.8925 - 497ms/epoch - 6ms/step
Epoch 130/150
88/88 - 0s - loss: 6.9594 - dense_6_loss: 4.9614 -
dense_7_loss: 1.9980 - dense_6_mean_absolute_error: 1.6173
- dense_7_mean_absolute_error: 8.8925 - 494ms/epoch - 6ms/step
Epoch 131/150
88/88 - 0s - loss: 6.9704 - dense_6_loss: 4.9759 -
dense_7_loss: 1.9945 - dense_6_mean_absolute_error: 1.6125
- dense_7_mean_absolute_error: 8.8925 - 485ms/epoch - 6ms/step
Epoch 132/150
88/88 - 0s - loss: 6.9665 - dense_6_loss: 4.9699 -
dense_7_loss: 1.9966 - dense_6_mean_absolute_error: 1.6158
- dense_7_mean_absolute_error: 8.8925 - 481ms/epoch - 5ms/step
Epoch 133/150
88/88 - 0s - loss: 6.9560 - dense_6_loss: 4.9601 -
dense_7_loss: 1.9959 - dense_6_mean_absolute_error: 1.6171
- dense_7_mean_absolute_error: 8.8925 - 497ms/epoch - 6ms/step
Epoch 134/150
88/88 - 1s - loss: 6.9805 - dense_6_loss: 4.9860 -
dense_7_loss: 1.9945 - dense_6_mean_absolute_error: 1.6130
- dense_7_mean_absolute_error: 8.8925 - 503ms/epoch - 6ms/step
Epoch 135/150
88/88 - 0s - loss: 6.9694 - dense_6_loss: 4.9770 -
dense_7_loss: 1.9924 - dense_6_mean_absolute_error: 1.6171
- dense_7_mean_absolute_error: 8.8925 - 491ms/epoch - 6ms/step
Epoch 136/150
88/88 - 0s - loss: 6.9512 - dense_6_loss: 4.9555 -
dense_7_loss: 1.9958 - dense_6_mean_absolute_error: 1.6163
- dense_7_mean_absolute_error: 8.8925 - 488ms/epoch - 6ms/step
Epoch 137/150
88/88 - 0s - loss: 6.9517 - dense_6_loss: 4.9556 -
dense_7_loss: 1.9961 - dense_6_mean_absolute_error: 1.6124
- dense_7_mean_absolute_error: 8.8925 - 496ms/epoch - 6ms/step
Epoch 138/150
88/88 - 0s - loss: 7.0051 - dense_6_loss: 5.0089 -
dense_7_loss: 1.9962 - dense_6_mean_absolute_error: 1.6279
- dense_7_mean_absolute_error: 8.8925 - 483ms/epoch - 5ms/step
Epoch 139/150
88/88 - 0s - loss: 6.9476 - dense_6_loss: 4.9513 -
```

```
dense_7_loss: 1.9963 - dense_6_mean_absolute_error: 1.6150
- dense_7_mean_absolute_error: 8.8925 - 483ms/epoch - 5ms/step
Epoch 140/150
88/88 - 0s - loss: 6.9505 - dense_6_loss: 4.9581 -
dense_7_loss: 1.9924 - dense_6_mean_absolute_error: 1.6158
- dense_7_mean_absolute_error: 8.8925 - 486ms/epoch - 6ms/step
Epoch 141/150
88/88 - 0s - loss: 6.9708 - dense_6_loss: 4.9785 -
dense_7_loss: 1.9923 - dense_6_mean_absolute_error: 1.6194
- dense_7_mean_absolute_error: 8.8925 - 488ms/epoch - 6ms/step
Epoch 142/150
88/88 - 0s - loss: 6.9526 - dense_6_loss: 4.9606 -
dense_7_loss: 1.9920 - dense_6_mean_absolute_error: 1.6141
- dense_7_mean_absolute_error: 8.8925 - 491ms/epoch - 6ms/step
Epoch 143/150
88/88 - 0s - loss: 6.9651 - dense_6_loss: 4.9700 -
dense_7_loss: 1.9951 - dense_6_mean_absolute_error: 1.6225
- dense_7_mean_absolute_error: 8.8925 - 484ms/epoch - 6ms/step
Epoch 144/150
88/88 - 0s - loss: 6.9542 - dense_6_loss: 4.9643 -
dense_7_loss: 1.9899 - dense_6_mean_absolute_error: 1.6144
- dense_7_mean_absolute_error: 8.8925 - 489ms/epoch - 6ms/step
Epoch 145/150
88/88 - 0s - loss: 6.9568 - dense_6_loss: 4.9679 -
dense_7_loss: 1.9888 - dense_6_mean_absolute_error: 1.6188
- dense_7_mean_absolute_error: 8.8925 - 491ms/epoch - 6ms/step
Epoch 146/150
88/88 - 0s - loss: 6.9332 - dense_6_loss: 4.9453 -
dense_7_loss: 1.9878 - dense_6_mean_absolute_error: 1.6163
- dense_7_mean_absolute_error: 8.8925 - 487ms/epoch - 6ms/step
Epoch 147/150
88/88 - 0s - loss: 6.9942 - dense_6_loss: 5.0063 -
dense_7_loss: 1.9879 - dense_6_mean_absolute_error: 1.6242
- dense_7_mean_absolute_error: 8.8925 - 489ms/epoch - 6ms/step
Epoch 148/150
88/88 - 0s - loss: 6.9595 - dense_6_loss: 4.9690 -
dense_7_loss: 1.9904 - dense_6_mean_absolute_error: 1.6185
- dense_7_mean_absolute_error: 8.8925 - 484ms/epoch - 5ms/step
Epoch 149/150
88/88 - 0s - loss: 6.9568 - dense_6_loss: 4.9695 -
dense_7_loss: 1.9873 - dense_6_mean_absolute_error: 1.6155
- dense_7_mean_absolute_error: 8.8925 - 494ms/epoch - 6ms/step
Epoch 150/150
88/88 - 0s - loss: 6.9523 - dense_6_loss: 4.9661 -
dense_7_loss: 1.9863 - dense_6_mean_absolute_error: 1.6203
- dense_7_mean_absolute_error: 8.8925 - 485ms/epoch - 6ms/step
```

Next comes the evaluation step. The trained model can now be applied onto the Testing Dataset to predict the number of rings.

```
    yhat1, yhat2 = model.predict(X_test)

    # calculate error for regression model: mean absolute square
    error = mean_absolute_error(y_test, yhat1)
    print('MAE: %.3f' % error)


    # evaluate accuracy for classification model: argmax
    yhat2 = argmax(yhat2, axis=-1).astype('int')
    acc = accuracy_score(y_test_class, yhat2, normalize=False)
    print('Number of test datapoints: ', len(y_test_class))
    print('Accuracy: %.3f' % acc)
```

Listing B.22: Functional Implementation: Predicting values using testing data

```
44/44 [==============================] - 0s 2ms/step
MAE: 1.622
Number of test datapoints:  1379
Accuracy: 396.000
```

This result shows that the final results of the functional model are more or less comparable to the sequential implementation. The regression has outputted a result with an accuracy of approximately ± 1.6 Rings and the classification has an accuracy of approx. 28.7% - very similar values to the initial, sequential implementation.

This experiment therefore clearly proves at least one thing: the functional implementation works at least as well as the sequential implementation, when it comes to model accuracies, even though the number of parameters to be optimized have been drastically reduced by almost 35%.

Bibliography

[1] C. Schreiber, “KI-gestützte „follow-me“-funktion am beispiel des JetRacer”, Master Thesis, Hochschule Mittweida, Jan. 2023, 59 pp.

[2] T. M. Mitchell, *Machine learning*. McGraw-Hill Science/Engineering/Math, Mar. 1, 1997, 432 pp., ISBN: 0-07-042807-7. [Online]. Available: http://www.cs.cmu.edu/~tom/files/MachineLearningTomMitchell.pdf (visited on 11/01/2023).

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016, 775 pp., ISBN: 978-0-262-03561-3. [Online]. Available: https://www.deeplearningbook.org (visited on 10/10/2023).

[4] “Regression vs classification in machine learning - javatpoint”, www.javatpoint.com. (), [Online]. Available: https://www.javatpoint.com/regression-vs-classification-in-machine-learning (visited on 01/27/2024).

[5] K. O’Shea and R. Nash, “An introduction to convolutional neural networks”, *arXiv preprint arXiv:1511.08458*, Dec. 2, 2015. arXiv: 1511.08458[cs]. [Online]. Available: http://arxiv.org/abs/1511.08458 (visited on 11/15/2023).

[6] A. Geiger, *Deep learning lecture notes, university of tübingen, winter semester 2020/2021*, 2020. [Online]. Available: https://uni-tuebingen.de/fakultaeten/mathematisch-naturwissenschaftliche-fakultaet/fachbereiche/informatik/lehrstuehle/autonomous-vision/lectures/deep-learning/ (visited on 11/15/2023).

[7] TutorialPoint. “Artificial neural network - basic concepts”. (), [Online]. Available: https://www.tutorialspoint.com/artificial_neural_network/artificial_neural_network_basic_concepts.htm (visited on 10/20/2023).

[8] 3. D. S. Team and I. Vankov, *Convolutional neural networks with TensorFlow in python*, Mar. 2021. [Online]. Available: https://www.udemy.com/course/convolutional-neural-networks-with-tensorflow-in-python/ (visited on 06/15/2023).

[9] H. Habibi Aghdam and E. Jahani Heravi, *Guide to Convolutional Neural Networks*. Cham: Springer International Publishing, 2017, ISBN: 978-3-319-57549-0 978-3-319-57550-6. DOI: 10.1007/978-3-319-57550-6. [Online]. Available: http://link.springer.com/10.1007/978-3-319-57550-6 (visited on 11/20/2023).

[10] P. Mahajan. “Lighting the PyTorch tensors”, Analytics Vidhya. (Aug. 4, 2020), [Online]. Available: https://medium.com/analytics-vidhya/lighting-the-pytorch-tensors-298b82722420 (visited on 12/01/2023).

[11] “Introduction to TensorFlow | machine learning”, Google for Developers. (), [Online]. Available: https://developers.google.com/machine-learning/crash-course/first-steps-with-tensorflow/toolkit (visited on 12/05/2023).

[12] Simplilearn. “What is keras and why is it so popular in 2023?”, Simplilearn.com. (), [Online]. Available: https://www.simplilearn.com/tutorials/deep-learning-tutorial/what-is-keras (visited on 12/15/2023).

[13] Y. LeCun, C. Cortes, and C. J. Burges. “The mnist database of handwritten digits”. (), [Online]. Available: http://yann.lecun.com/exdb/mnist/.

[14] W. Nash, T. Sellers, S. Talbot, A. Cawthorn, and W. Ford, *Abalone*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C55C7W, 1995.

[15] F. Pedregosa *et al.*, “Scikit-learn: Machine learning in Python”, *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[16] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG].

Statutory Declaration in Lieu of an Oath

I – Saransh Rastogi – do hereby declare in lieu of an oath that I have composed the presented work independently on my own and without any other resources than the ones given.

All thoughts taken directly or indirectly from external sources are correctly acknowledged.

This work has neither been previously [redacted]