Universidad Politécnica de Madrid

ETSII

Estimating Physical Stellar Parameters from High Resolution Spectra with a Deep Learning Regression Approach

Supervisor

Joaquin Ordieres Meré

Master's Thesis by

Jakob Salomonsson

Madrid, September 2018

Estimating Physical Stellar Parameters from High Resolution Spectra with a Deep Learning Regression Approach

MASTER'S THESIS

Author:

Jakob Salomonsson

jakob.lsalomonsson@alumnos.upm.es

Supervisor:

Joaquin Ordieres Meré

j.ordieres@upm.es

Department of Industrial Engineering, Business Administration and Statistics
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES (ETSII)

UNIVERSIDAD POLITÉCNICA DE MADRID (UPM)

c/ José Gutiérrez Abascal, 2. 28006 – Madrid
Tel: +34 9106 76700
Website: http://www.etsii.upm.es

Madrid, September 2018

Abstract

For life as we know it to exist, water is an absolute necessity. Astronomers are therefore searching the cosmos for similar planets with similar conditions as our own. Determining effective temperature (T_{eff}), surface gravity ($\log g$) and metallicity (M/H) of a star is important in understanding the orbiting planets. Although explorations around the most common *M-type* stars have been successful, little further effort has been made in searching around these stars due to their faintness in the optical where most observations have been made.

This Master's Thesis makes use of data obtained by the *CARMENES* project, which was initiated specifically for M-type star exoplanet searches, and a theoretical *BT-Settl* dataset. The datasets underwent pre-processing in several meticulous steps, involving normalising both within each data file and over both datasets, calculating the impact of nearby flux values from nearby wavelengths on a current wavelength, calculating rolling means and to cancel out the importance of the distance between the Earth and the observed star.

As the objective of this project was to evaluate the capabilities of *Deep Learning* algorithms, more specifically *Convolutional Neural Networks* (*CNN*s), with a *regression* approach, on predicting physical stellar parameters from spectrograms, the pre-processed data was subsequently transformed to be represented by spectra. Due to the size of the original input, the hardware wasn't able to process it, and as a result, had to be down-sized. Five distinct models with five different down-sized input data were developed and evaluated.

Based upon the obtained results, it can be concluded that the approach is valid. It is possible to estimate physical stellar parameters with the use of CNNs on spectra with a regression approach.

T_{eff} was very accurately estimated on all down-sized input data, with mean squared errors of 0.000079 and 0.000003 for near infrared (NIR) and visible (VIS) wavelengths respectively. This is equivalent to errors in the range of $12 - 27K$ and $2 - 5K$, respectively, for M-type stars spanning the effective

temperature range of $1300 - 3000K$. $\mathrm{Log}\, g$ was estimated within uncertainties of 0.0002dex and 0.14dex while M/H within 0.058dex and 0.30dex for NIR and VIS respectively for the best performing model. For T_{eff}, down-sized input data is still reliable for estimations, yielding very good results even on the most down-sized data, while less so for $\log g$ and M/H. Especially $\log g$ experienced substantial performance decline already after modest down-sizing.

KEYWORDS: Deep Learning, Convolutional Neural Networks, regression, physical stellar parameters, CARMENES.

Abstracto

Para que la vida como la conocemos pueda existir, el agua es una necesidad absoluta. Por eso, los astrónomos están buscando en el cosmos planetas con condiciones parecidas al nuestro. Determinar la temperatura efectiva (T_{eff}) la gravedad superficial ($\log g$) y la metalicidad (M/H) de una estrella es importante para entender las planetas que la orbitan. Aunque búsquedas acerca de las estrellas más comunes de tipo M han sido exitosas, poco esfuerzo se ha realizado respecto a estas estrellas debido a su palidez en los instrumentos ópticos donde la mayor parte de las observaciones han sido hechas.

Este Trabajo de Fin de Máster hace uso de datos obtenidos por el proyecto *CARMENES*, el cual fue iniciado específicamente para buscar exoplanetas en estrellas de tipo M, así como un conjunto de datos teórico BT-Settl. El conjunto de datos fue pre-procesado en una serie de pasos minuciosos, entre ellos la normalización tanto dentro de cada archivo como entre el conjunto de datos, calculando el impacto de los valores de flujo cercanos de longitudes de onda cercanas en una longitud de onda actual, calculando los medios de rodadura y para eliminar la importancia de la distancia entre la Tierra y la estrella observada.

Como el objetivo del proyecto era evaluar las capacidades de los algoritmos de *Aprendizaje Profundo* (en inglés, *Deep Learning*), específicamente las *Redes Neuronales Convolucionales* (en inglés, *Convolutional Neural Networks*, o solo CNNs) con un enfoque de *regresión* para estimar parámetros estelares físicos mediante espectrogramas, los datos pre-procesados fueron subsecuentemente transformados para ser representados por espectrogramas. Debido al tamaño de la entrada, el hardware no pudo procesarlo, y como resultado, hubo que reducirlo. Cinco modelos distintos de datos de entrada con cinco tamaños diferentes de reducción fueron desarrollados y evaluados.

Basado en los resultados obtenidos, se puede concluir que este enfoque es válido. Es posible estimar parámetros estelares físicos con el uso de CNNs por sus espectrogramas, utilizando un método de regresión.

El T_{eff} fue estimado con una precisión muy alta por todas las entradas de datos reducidas, con errores cuadráticos medio de 0.000079 y 0.000003 (equivalente a $12-27K$ y $2-5K$) para el espectro del infrarrojo cercano (NIR) y el longitudes de onda visibles (VIS) respectivamente para el rango $1300-3000K$. El $\log g$ fue estimado con incertidumbres de 0.0002dex y 0.14dex mientras que M/H entre 0.058dex y 0.30dex para NIR y VIS respectivamente.

Para T_{eff}, la entrada de datos reducida es aún confiable para estimaciones, rindiendo muy buenos resultados incluso en los datos más reducidos mientras que un poco menos para $\log g$ y M/H. Especialmente, $\log g$ experimentó un declive en su desempeño después de una modesta reducción.

PALABRAS CLAVE: Aprendizaje Profundo, Redes Neuronales Convolucionales, regresión, parámetros estelares físicos, CARMENES.

Acknowledgements

I have many to thank for the completion of this Master's Thesis. First of all, and maybe the biggest reason for making it possible, is my tutor Joaquin Ordieres Meré. Without his professionalism, competence, insights and help I would have found myself lost many times. This project would have never started, even less finished without him. Thank you very much for always being available, if not physical, then by email, at any time of the day, including holidays.

A big thanks to my colleagues and friends at the office as well, José, John, Isaac, Ebru, Hossein and Sun for the awesome discussions we've had during lunches and breaks. Both related to the field and other completely different topics. Thank you, it's been great fun!

Last but absolutely not least, I would like to thank my beloved family, Susanne, Ulf, Maja, Artur and Lydia for always being there, both during hard and beautiful moments. Thank you, you're the best and I love you!

This research has made use of the Spanish Virtual Observatory (http://svo.cab.inta-csic.es) supported from the Spanish MINECO/FEDER through grant AyA2014-55216.

Table of Contents

Chapter 1 Introduction and Objective

1.1 Introduction

The first evidence of a planet outside our own solar system, i.e. exoplanet, was found as early as 1917 [1]. However, the findings were so ahead of their time that they were not recognised as such for a century to come. Thus, the first scientific confirmed detection came in 1992 when A. Wolszczan and D. A. Frail used a radio telescope to detect two planet-sized bodies orbiting a pulsar [2].

One can argue that the discovery increased the interest in the field both among scientist as well as the public. This was later indirectly confirmed with a huge investment in the Kepler Mission, where a telescope was sent into orbit in 2009 to discover Earth-size planets orbiting distant stars [3]. The exoplanet discovery rate has increased steadily over time since the first findings, leading to a total of 3774 confirmed exoplanets as of August 8th 2018 [4].

Humanity has taken great leaps forwards since the time when Galileo Galilei was put into life time house arrest for strongly arguing in favour for the Copernican model. However, the holy grail in astronomy, and arguably in any other field, would be the discovery of life on another planet. For life as we know it, water is an absolute necessity. Astronomers are therefore searching the cosmos for similar planets with similar conditions as our own. Although scientists' understanding of what makes up a habitable zone continues to evolve, it can generally be depicted as a band not too close nor too far away from the host star, where liquid water can be found. See Figure 1.1 below. As stars and planets come in many types and sizes, the interplay between those factors

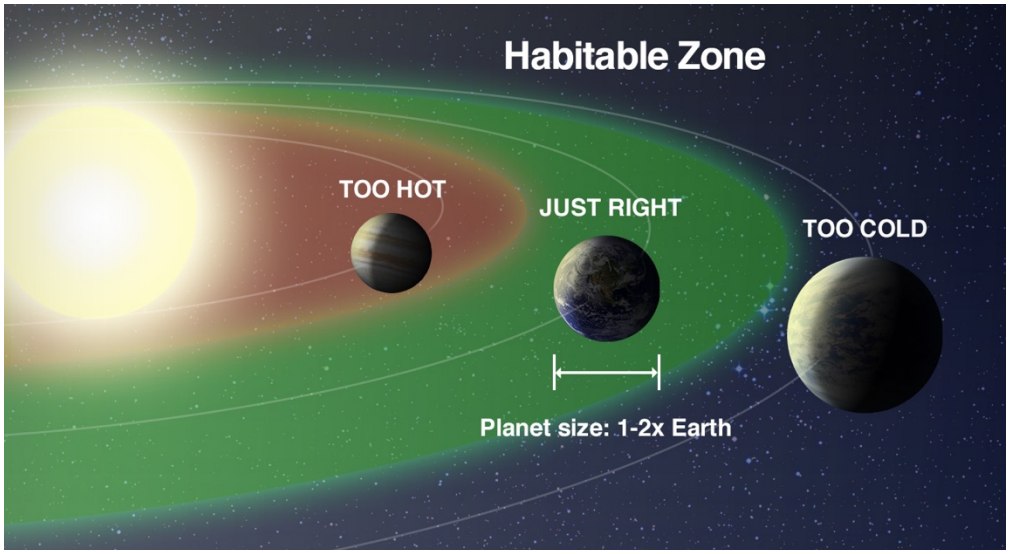

Figure 1.1: Habitable Zone [5]. If the planet is too close to its host star, the temperature will be too high to hold life as we know it. Conversely, if it's too far away, it'll be too cold.

defines the characteristics of the habitable zone. If we stick to the idea of "looking for something we already know", then small, Earth-sized rocky planets would be our best bet to find life [5].

Not only is the size and composition of the stars and planets critical to hold life, but so is time. Giant bright stars burn out within a few million years while smaller dwarfs, as our own Sun, can produce a steady shine for billions of years. It might be enough with a few hundred million years for microscopical single-celled life to form. Reasoning from what we know about life on Earth though, it's most likely a far too short time frame for advanced life forms, such as our own, to evolve.

Based on conservative measurements of small portions of the sky and extrapolating, there are around 100 billion stars in our Milky Way galaxy [6]. Recent statistical calculations estimate that there is, at least, one planet orbiting each star in the galaxy [4]. With the same method, it can be estimated that there are around 10 billion galaxies in the observed universe, yielding some 1,000,000,000,000,000,000,000,000, or 10^{23}, planets to search.

Most of the exoplanet searches have been conducted on G-type Sun stars. See Figure 1.2 for Star Spectral Classification. Despite *M-type* stars being the

most common star type in the galaxy and that searches around these stars have proven successful, they haven't been as extensive as around K, G and F stars. The main reason for this is their faintness in the optical, where most of the radial velocity (RV) searches are being performed [7] [8]. Since no surveys on high-resolution near-infrared spectrograph dedicated to planet search existed, the *CARMENES* study was initiated.

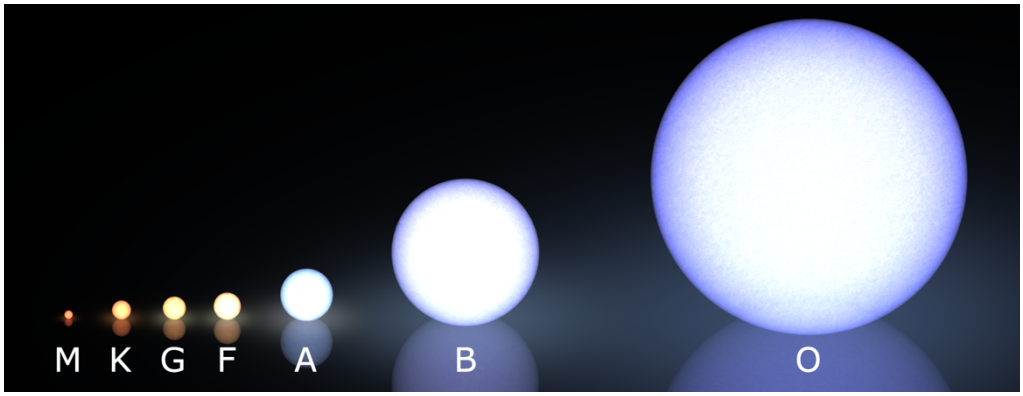

Figure 1.2: Morgan-Keenan Modern Star Type Spectral Classification displaying the colour and size distribution of stars.

CARMENES (Calar Alto high-Resolution search for M dwarfs with Exoearths with Near-infrared and optical Échelle Spectrographs) is a Spanish-German consortium project between 11 different organisations. Among these are the Complutense University of Madrid, the Max-Planch-Institute in Heidelberg, Germany, and the Calar Alto Astronomical Observatory in Almería, Spain [9]. The project is targeting some 300 M dwarfs with the main scientific goal of detecting Earth-sized planets ($2\ M_{Earth} < M_{New\ planet} < 5\ M_{Earth}$) in the habitable zone of the host star. Cool dwarfs are targeted where the habitable zone lies closer to the star. The research conducted will thus provide a comprehensive overview of planetary systems orbiting M-type stars in the northern celestial hemisphere, where follow-up studies can further investigate their habitability [10].

To better understand a star and its possible exoplanets, it is of importance to determine several physical parameters such as effective temperature, surface

gravity and metallicity. By using high resolution data obtained from the CARMENES study, and a synthetical BT-Settl dataset generated by a model, spectrograms can be created. As the BT-Settl data comes with physical parameters unique for the modelled stars, *Deep Learning* algorithms can be applied to analyse the spectrograms for patterns, ideally making it possible to predict the stars' parameters. Earlier research has attempted to determine these parameters using other Machine Learning algorithms [11]. However, the use of Deep Learning algorithms might also contribute to such goal, and thus, this project is an exploratory analysis regarding the feasibility of *Convolutional Neural Networks* (*CNNs*) in the field.

The Deep Learning techniques employed in this Master's Thesis are applicable in a wide set of sectors and systems, covering Business Intelligence (BI), Management Systems, Decision Making Systems etc. etc., where these algorithms increasingly have been implemented for their capacity to process and analyse vast amounts of data. By finding patterns previous methods weren't capable of and making predictive and normative suggestions, they have become the foundation in the BI sector [12]. Through minor or no adjustments, the very same methods can be applied on a wide set of fields. As a manager or decision maker in the 21th century, it will be ever increasingly important to have an understanding of their potentiality.

1.2 Objective

The objective of this Master's Thesis is to evaluate the capabilities of Deep Learning with a *Regression* approach on estimating physical stellar parameters from spectra.

Chapter 2 Related Works

CNNs have been very successful in image classification tasks with many studies carried out in the field, such as [13] [14] [15] [16] and [17]. Other studies displayed how Deep Forest is an easier and potent alternative to CNNs with, in some cases, both superior results and faster implementation process [18] [19]. However, questions can be raised in the latter about the complexity of the CNN model's architecture used for comparison. The works reviewed in this chapter demonstrates some studies where different algorithms have been used in regression tasks to determine one or several output parameters.

2.1 A. González-Marcos et al. (2016)

A. González-Marcos et al. analysed and compared several data compression techniques in this paper with the conclusion that independent component analysis (ICA) performed best on all the different signal-to-noise ratio (SNR) arrangements tested [20].

The experiments were performed on high-resolution (m/s accuracy) spectra of stars in the temperature range of $4000 - 8000$ K. By training a simple support vector machine (SVM) model with basic configurations, the objective was to determine effective temperature (T_{eff}), surface gravity ($\log g$), metallicity ($[M/H]$) and/or the alpha-to-iron ratio ($[\alpha/Fe]$). Furthermore, the team approached the problems as a regression task rather than a classification task, which is often the case. The performance was later measured on the evaluation set using the root-mean-square error (RMSE).

The synthetic dataset used in the study was compressed with three linear data compression techniques; principal component analysis (PCA), independent

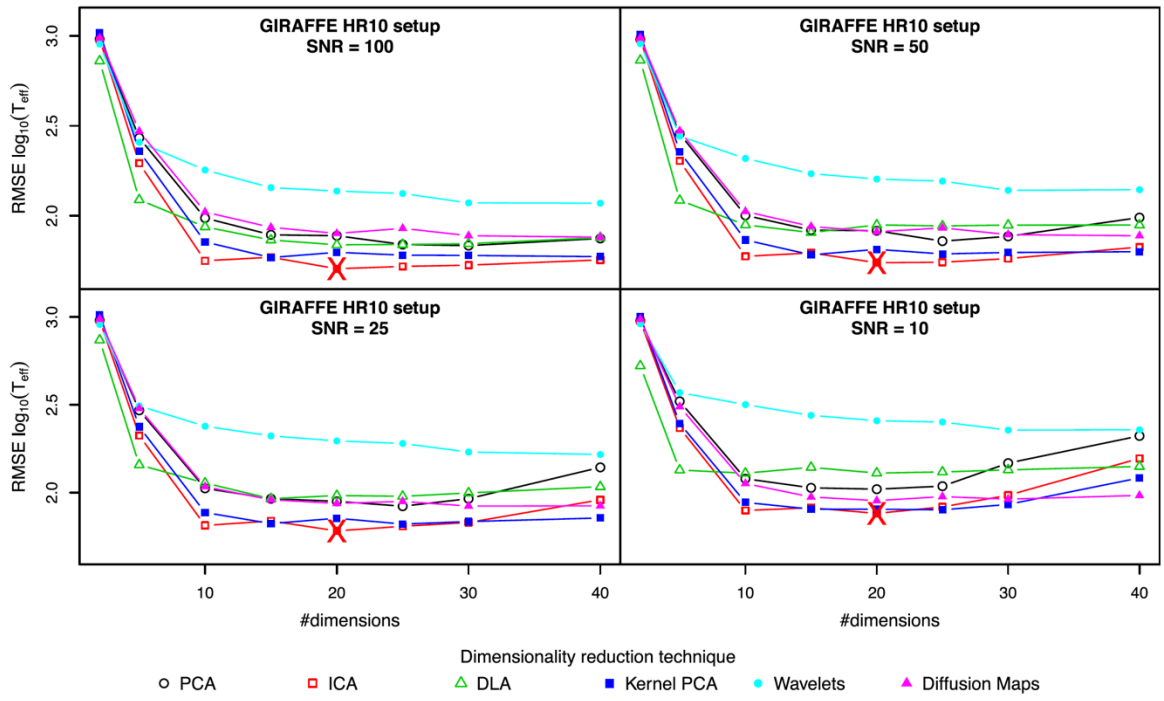

Figure 2.1: Temperature estimation error as a function of number of dimensions used for data compression [20]. SNR 100, 50, 25 and 10.

component analysis (ICA) and discriminative locality alignment (DLA), as well as with three nonlinear techniques that can be generalised on unseen data; kernel PCA, wavelets and diffusion maps (DMs).

The results for T_{eff} are displayed in Figure 2.1, where the RMSE is plotted against the dimensionality compression for different SNR regimes. The lowest error was obtained with a 20 dimensionality compression along all the noise regimes for ICA, closely followed by Kernel PCA.

However, a question arise from examining this study, which is their use of a fairly simple SVM model to predict the stellar parameters. A more complex deep learning model, such as a CNN, might find patterns in the dataset the SVM can't find, even when the dataset is heavily compressed. If that's the case, the conclusions drawn in this study might be more relevant for different Machine Learning algorithms, and not so much for Deep Learning.

2.2 NVIDIA Corporation (2016)

13 different authors for the NVIDIA Corporation empirically demonstrated in 2016 that CNNs can be learned the entire task of following a lane and road, without extra systems for road and lane marking detection, path planning and control [21]. This end-to-end approach proved remarkable potent and could for example be used in self-driving cars. The authors argue that this solution, if

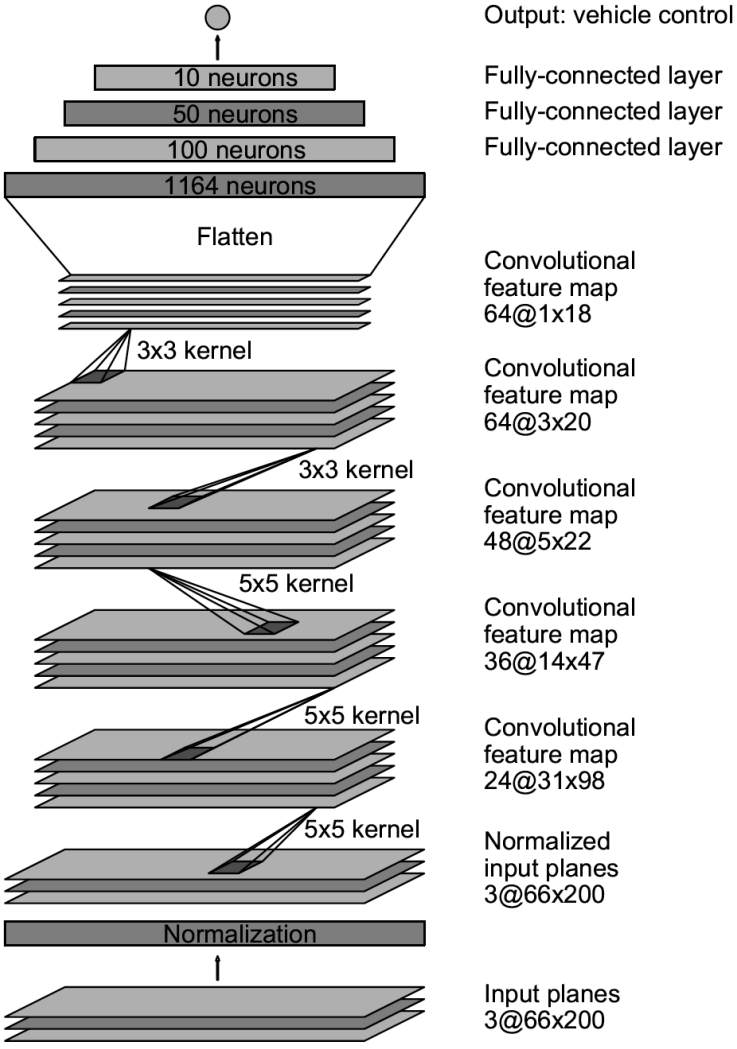

Figure 2.2: CNN architecture used in the NVIDIA paper. The network holds some 27 million connections and 250 thousand parameters [21].

further developed, might lead to both better overall performance and smaller systems than current.

The network learned to drive on local roads and on highways, with or without lane markings, using the human steering angle as only training signal. It also operates successfully in areas with less visual guidance such as parking lots or during snowy and rainy weather. The model was trained on a surprisingly small amount of data, obtained from less than one hundred hours of driving, from a front mounted car camera.

The model network consist in total of 9 layers; a normalisation layer, 5 convolutional layers and 3 connected layers, where the initial layer was hard-coded and, thus, not adjusted during the learning process. Figure 2.2 displays the CNN architecture.

Interestingly, the team approached the problem in this paper as a regression task, with the steering angle as an output. Thus, the authors displayed that processing complex images with CNNs can result in reliable output in regression tasks.

2.3 L. M. Sarro et al. [2017]

L. M. Sarro et al. displayed in this paper that physical stellar parameters of M-type stars can be estimated from synthetic BT-Settl data with regression models [11]. The evaluated models were eight in total:

- Random Forest Regression Models
- k-Nearest Neighbours
- Generalised Boosted Regression Models
- Multi-layer Perceptron Neural Networks
- Bagging with Multiadaptive Spline Regression Models
- Support Vector Regression with Gaussian Kernel
- Kernel Partial Least Squares Regression

- Rule Regression models

The input data was pre-processed in several steps, although not as to be represented as spectrograms, before fed into a GA algorithm who selected the relevant parts in the spectra. Each one of the eight models were subsequently evaluated starting from the selected parts.

Despite the prediction errors for the best performing Random Forest Regression model were fairly high ($\approx 0.25\ dex$, or $MSE \approx 0.61$) for metallicities, the team was able to detect five sub dwarfs for further investigation.

2.4 Shun Miao et al. (2016)

Shun Miao et al. demonstrated in this paper that it is possible to estimate transformation parameters for medical X-ray images with a CNN regression approach, referred to as Pose Estimation via Hierarchical Learning (PEHL) [22].

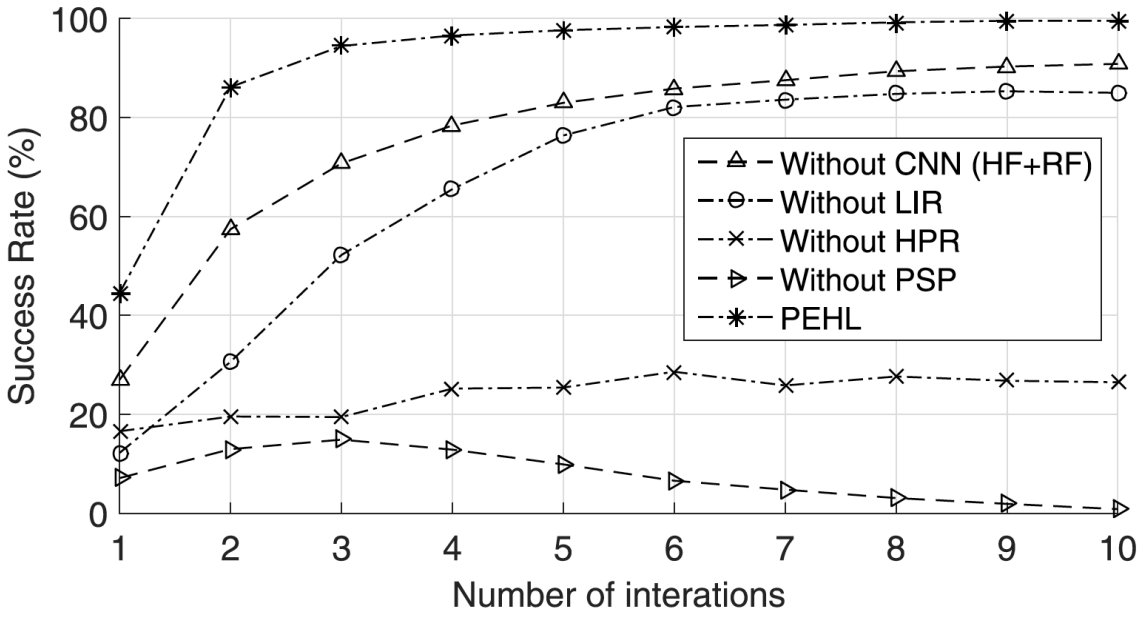

Figure 2.3: Success rates with and without PEHL [22].

To facilitate accurate diagnosis and/or to provide advanced image guidance, it is desirable to bring the pre-operative 3-D data and intra-operative 2-D data into the same coordinate system. Such mapping is highly complex and training a regressor to undo the mapping is a difficult task. As the transformation should operate in real-time, the process needed to be fast, and thus, a smaller CNN architecture was chosen.

As displayed in Figure 2.3, the model performed very well (see the star-dotted PEHL lines) and achieved nearly perfect (99.6%) parameter estimations in few iterations. To confirm the PEHL model's importance, the authors disabled and replaced it with an implementation of a companion algorithm using HAAR features (HF) with Regression Forest (RF). The results confirm the CNN's importance in image processing as the success rate decreased to around 90% without it. LIR, HPR and PSP were all different configurations of the CNN regression model.

Chapter 3 Theories and Frameworks

3.1 Convolutional Neural Networks

3.1.1 Background

Convolutional Neural Networks (*CNN*) are a specific type of *Artificial Neural Networks* (ANN). They have since some vital breakthroughs in research, especially with the publication of Yann LeCun et al.'s paper in 1998 [23], delivered state of the art performance in a magnitude of tasks. They are successfully being applied in natural language processing [24], image and video recognition [17] [21], recommender systems [25], among other areas. AlphaGo, who beat the world's best Go player Lee Sedol in 2016 [26], a game which was previously considered impossible for artificial machines to master, was powered by CNNs.

According to the authors in the 1998 paper, CNNs are biologically-inspired models. Earlier research where mammals' visualisation of the world was explained [27], inspired the engineers to develop similar artificial pattern recognition systems.

3.1.2 Biological and Artificial Neurons

The basic computational unit in both the brain and CNNs is a neuron [28]. There are approximately 86 billion neurons in the human brain and, depending on the size and complexity of the architecture, thousands, millions or even billions, in a CNN model.

Figure 3.1 displays a drawing of both a biological (left) and artificial (right) neuron. Each neuron receives input signals from several dendrites (e.g.

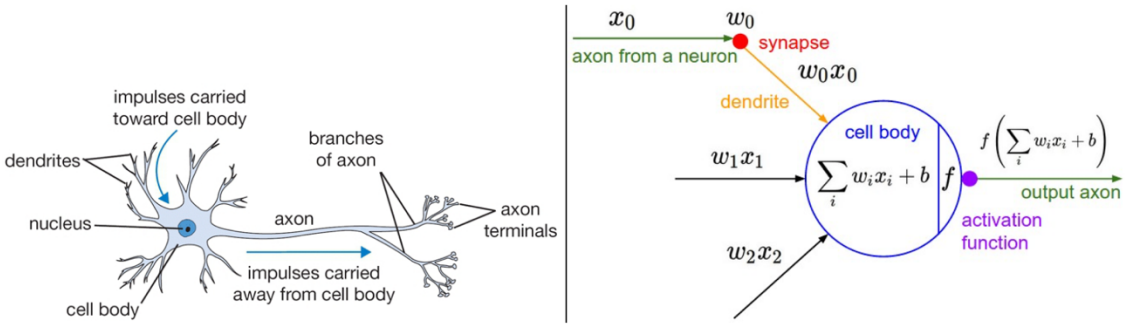

Figure 3.1: A drawing of a biological neuron (left) and its artificial equivalent (right) [28].

x_0), multiplies the input with its weights (w_0), adds a bias (b) and lastly, applies the activation function before it outputs the resulting signal through its axon. The axon then branches further out connecting with dendrites of other neurons.

The idea is that the synaptic strengths (or weights *w*) are learnable, and they control the strength, direction (positive or negative weight) and influence one neuron has on another. If the final sum is above some threshold, the neuron fire, sending information along its axon. The firing rate is modelled by activation functions where the two chosen for this project are further explained in chapter 3.2.

3.1.3 Input/Output Volumes and Kernel Operations

CNNs are frequently applied to image data of different kinds and an image is merely a 2D matrix of pixel values, normally in the interval [0, 255]. However, coloured RGB images presents a third dimension making the input 3-dimensional. Hence, for a given RGB (Red, Green, Blue) image of size [200 x 200] pixels, each image will be represented by three matrices, one for each colour. The input volume would thus be [200 x 200 x 3] (width, height, colour channels). See Figure 3.2 for an illustrative explanation.

CNN tries to learn an image's features in order to be able to recognise what it constitutes of. More specifically, if used in Face Detection, a nose or a

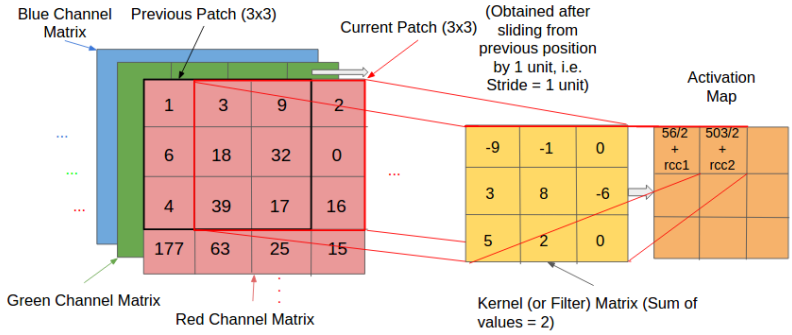

Figure 3.2: RGB Input Image Volume with Kernel Matrix and Activation Map [53]. The Input Volume has in this case the size [4 x 4 x 3].

mouth would be treated as a feature and subsequently distinguished and learned by the different layers.

The features are detected by the convolution kernel, or filter, by processing the entire input volume in smaller sections, shown as a red square in the figure, and mapped onto the Activation Map (or Feature Map) [29]. This process is also referred to as convolution of an image input and can mathematically be described as (for *tanh* non-linearities):

$$h_{ij}^{k} = \tanh\left((W^{k} * x)_{ij} + b_{k}\right) \quad (1)$$

Where h_{ij}^{k} is the activation map, whose filters are determined by the weights W^{k}, the input signal x and the bias b_{k}.

The Activation Map contains the most relevant features, found by the kernel, to explain the characteristics of the image input. This is done for all colour channels. Depending on the *kernel size* and *stride* chosen, the process will run faster or slower, detecting features with lower or higher accuracy.

3.1.4 Architecture Overview

CNNs constitute of several layers, each made up of a set of neurons. Every neuron in a layer is connected to a small section of the layer before it, instead of being fully-connected as for Regular Neural Nets. This has the advantage that far less connections, and thus weights, are needed, leading to faster training and

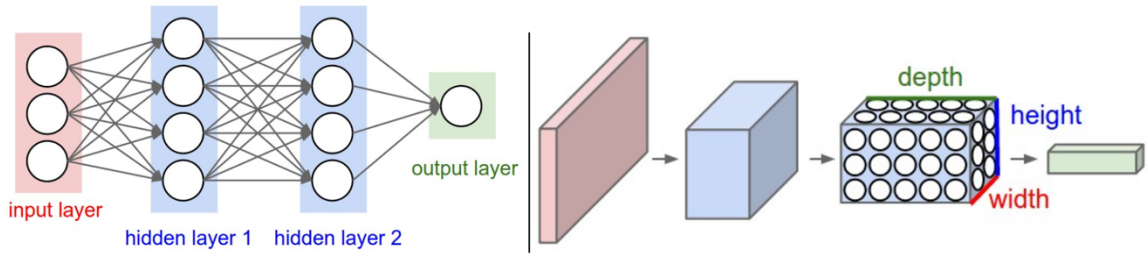

Figure 3.3: Simple architecture overview of a regular 3-layer Neural Network with two hidden layers [29]. Note the 1-dimensional vector in the output layer (in green).

less overfitting, i.e. better performance. Moreover, each layer in a CNN transforms the 3D input volume to a 3D output volume through an activation function [29].

The final output layer will result in a vector as the architecture will reduce the full image into one dimension. Figure 3.3 demonstrates a simple architecture with a 3D input image, two hidden layers and an 1-dimensional output layer.

Building CNN architectures is accomplished through stacking one or several layers, where *TensorFlow* and Theano are common libraries. Three main layers are used: *Convolutional Layers (Conv)*, *Pooling Layers (POOL)* and *Fully-Connected Layers* (FC), where the Conv Layers are the main building blocks, performing the majority of the heavy computations. Frequently, POOL Layers are used in pair with the Conv Layers to downsample along the spatial dimensions, from [32, 32, 3] to [16, 16, 3], for example. The FC layers will lastly compute the class scores for a classification task, or as in the case of the regression problem addressed in this thesis, predictions for each one of the desired parameters. Class scores or predictions are thus a CNN architecture's output.

3.2 Activation Functions

3.2.1 ReLU Activation Function

Most recent Deep Learning networks are now using *rectified linear units* (*ReLU*) for the hidden layers. Recent research has shown that ReLUs result in much faster training for larger and more complex networks [13]. It computes the function:

$$f(x) = \max(0, x) \tag{2}$$

which is zero when $x < 0$ and linear with an inclination of 1 when $x > 0$. See Figure 3.4 left for an illustrative explanation.

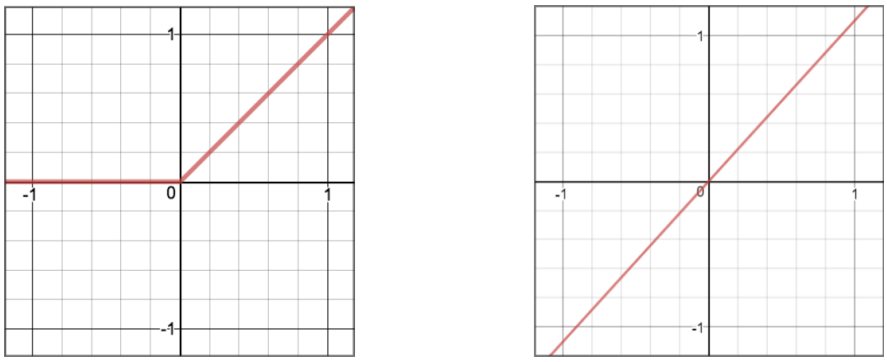

Figure 3.4: ReLU (left) and linear (right) activation functions.

The disadvantage of ReLUs is how fragile they can be when trained with larger learning rates, resulting in big disactivated chunks of the total network. However, by setting the learning rate properly, this is a less frequent problem [28].

3.2.1 Linear Activation Function

The linear activation function gives a range of activations and can thus be used in the last layer for a regression problem [30]. The computed function is very simple (see Figure 3.4 right for a visual representation):

$$f(x, m) = \mathrm{x} * \mathrm{m} \tag{3}$$

However, it shouldn't be used in a hidden layer if there are more than one, since several hidden layers (each one with a linear activation function) will always be replaceable with one single layer. E.g. ten linear functions will always produce a linear output.

3.3 Loss Function Mean Squared Error

A highly useful quantity to measure during training is the loss. It is evaluated on the individual batches (a group of training examples) during training and is helpful for adjusting the different hyperparameter settings to achieve a more efficient learning [31].

A common loss function for regression problems is the *Mean Squared Error* (*MSE*). It measures the average of the difference between the predicted quantity and the true answer, and is explained mathematically by:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2 \tag{4}$$

Where N is the number of predictions, $\hat{Y}$ the predicted and Y the true value.

Efficient learning is desirable as it speeds up the entire learning process as well as leads to better results. If an inadequate learning rate is chosen the network might never converge (i.e. find a solution), see the yellow line in Figure 3.5, or might find a suboptimal solution (green line). The blue line results in almost linear improvements with its low learning rate, and will thus take longer to train. A learning rate equivalent to the red line is desirable as the training is faster. The loss is initially decreasing in a higher pace but over time, when closer to convergence, decreases.

The right part of Figure 3.5 displays a typical and acceptable loss function over time as the model is trained (however, it might indicate a slightly low learning rate based on its decay, and a too low batch size as the loss is slightly too noisy).

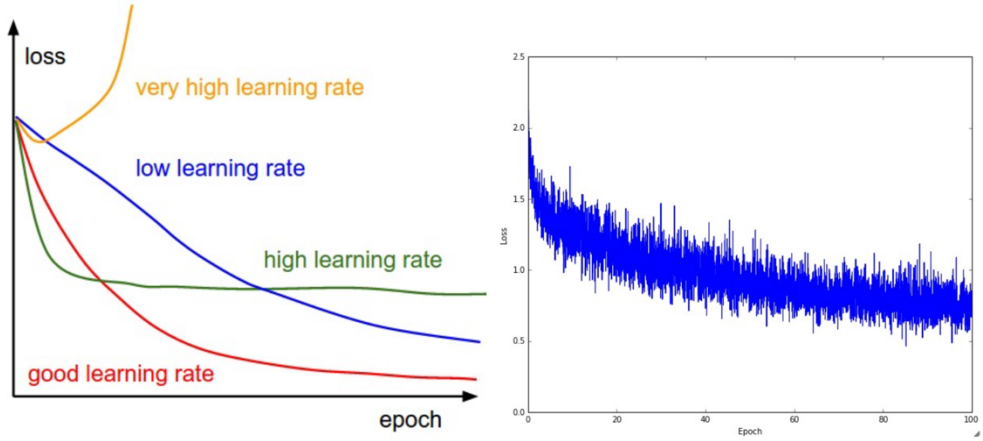

Figure 3.5: Depicting different learning rates and their behaviour (left), and an example of an acceptable loss function over time during training (right) [31].

3.4 Train, Validation and Test Sets

In order to train a model data is needed, and the dataset used for this is referred to as the training set. The model will see and learn from this dataset. The validation set on the other hand, is devoted to validate the model's performance during training. The model will not be trained on this dataset. Subsequently, the trained models' performance will be tested on the, by the model, unseen test set. It's necessary that the model hasn't been trained on the test set as it would be equivalent to cheating (imagine a human doing a test where she has seen all the answers in advance). The main objective of the validation and test sets is to test the model's ability to predict on previously unseen data in order to mitigate overfitting. An overfitted model performs very well, or perfectly, on the training set but fails in predicting on unseen data.

Depending on the size and characteristics of the total dataset, the split between train, validate and test sets can differ. However, a good practise is to choose 80% for training and 10% for each one of the validate and test sets [32] [33]. This approach is chosen for this project.

3.5 Brief Introduction to Keras and TensorFlow

3.5.1 Keras

Keras is a high-level open source neural networks API written in Python, with the possibility to run on top of TensorFlow, Theano or CNTK computational backends [34]. The main idea with Keras is to provide a smooth and fast implementation for starting with deep learning, enabling fast experimentation independently of the backend used. Start-ups as well as huge corporations use Keras for its friendly but still powerful user interface. It is backed by key companies such as Google, AWS and Microsoft and had, as of November 2017, more than 200,000 individual users.

A model is created by combining several standalone modules such as neural layers, activation functions, cost functions, optimisers, etc. It is easy to add and create new tailor-made modules, allowing total expressiveness, making Keras convenient for advanced research as well as fast implementations in the industry.

An example of creating a neural network model, before training and evaluating it, is shown in Code Snip 3.1 below.

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

# Create the model architecture
model = Sequential()
```

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

# Compile the model
model.compile(loss='categorical_crossentropy',
                        optimizer='sgd',
                        metrics=['accuracy'])

# Fit the model to the training data
model.fit(x_train, y_train, epochs=5, batch_size=32)

# Train the model
model.train_on_batch(x_batch, y_batch)

# Evaluate the performance of the trained model
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
```

Code Snip 3.1: Displaying a Keras implementation of building a neural network model, train it and finally evaluate its performance.

3.5.2 TensorFlow

TensorFlow is an open source software library, for implementing numerical computational machine learning algorithms as well as for executing these [35]. It was originally developed by the Google Brain team within Google's AI organisation and can be used on a variety of units (CPUs, GPUs, TPUs) on many different platforms (desktops, servers, mobiles, etc.).

The API is offered in several languages (Python, C++, Java, Go and Swift), but the Python API is the most complete, stable and easiest to use. TensorFlow is used for conducting research as well as for deploying full scaled machine learning solutions in the industry within a wide range of fields such as natural language processing, speech recognition, geographic information extraction, information retrieval, computer vision, robotics, drug discovery etc.

Figure 3.6 displays a typical *TensorBoard* for a CNN architecture, visualising the architecture graph. With thousands of nodes, such graphs can often be highly complex, but TensorBoard collapse low-degree nodes into blocks

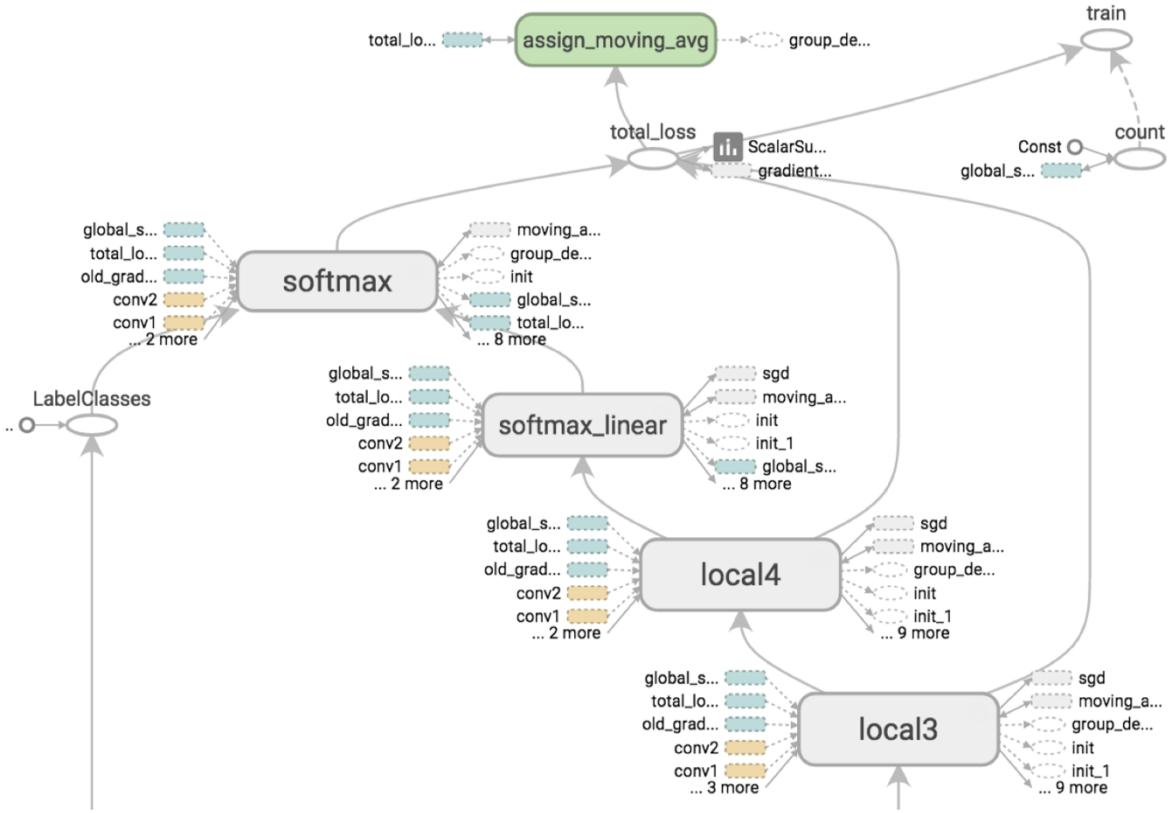

Figure 3.6: TensorBoard graph visualisation of a CNN model in TensorFlow [54].

and separate high-degree nodes, resulting in a more comprehensive visualisation. Being a graphical tool for TensorFlow, it can also be used in Keras.

Chapter 4 Data Pre-processing

4.1 Carmenes and BT-Settl Data Collection

Two different datasets were collected: the low resolution *BT-Settl* data set obtained from the *Spanish Virtual Observatory* (*SVO*), an affiliate of the *European Space Agency* (*ESA*), and the higher resolution Carmenes dataset, obtained from the CARMENES project (see chapter 1.1).

To collect the Carmenes dataset, M-type stars had been observed with the 3.5 m telescope at the Calar Alto Observatory, and registered by the CARMENES RV instrument. Earlier simulations concluded that a wide range of both visible and near-infrared wavelengths provide the best results for RV observations of M-dwarfs [36]. This approach will not only yield the highest possible precision, but also prevent false-positive RV signals caused by stellar activity.

The instrument included two separate échelle spectrographs, one for *visible wavelengths* (*VIS*), see Figure 4.1, and one for *near infrared wavelengths* (*NIR*), covering the total range 520 to 1710 nm, with a spectral resolution of at least $R = 80{,}400$ [37]. To achieve the projects' goal of $1\,m/s$ radial velocity precision, the spectrographs were installed inside climatic vacuum chambers (Figure 4.2), located in the observatory's dome. This enabled the necessary environmental conditions where the temperature had to be held constant within a range of ± 0.01 ℃ during the entire observation period. The NIR instrument operated at around $140\,K$ while the VIS at room temperature. Chambers and spectrographs for VIS and NIR were developed to be as similar as possible.

The NIR spectrograph was equipped with a $4112\ x\ 4096$ pixel CCD in the $520 - 1060\ nm$ range, while the VIS had installed a $2048\ x\ 2048$ pixel HgCdTe detector in the $960 - 1710\ nm$ range. They were both separately connected

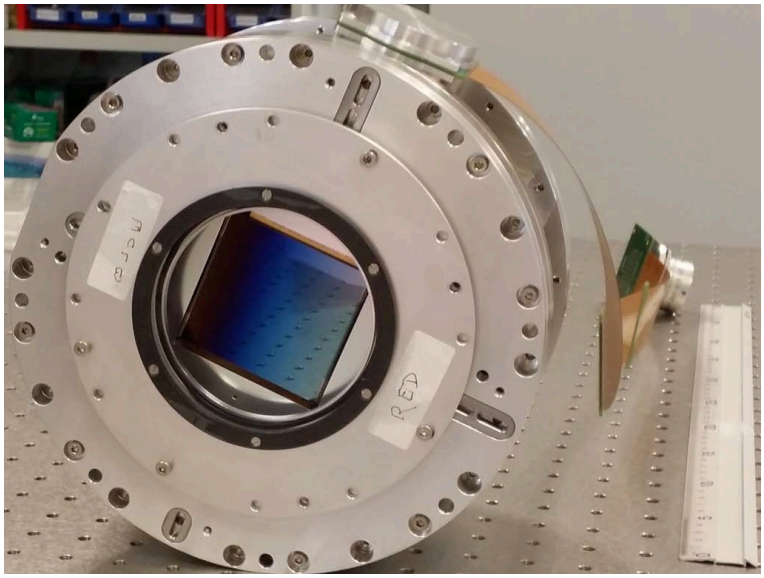

Figure 4.1: VIS light detector installed in its protective cryostat [37]. Length around 300 mm.

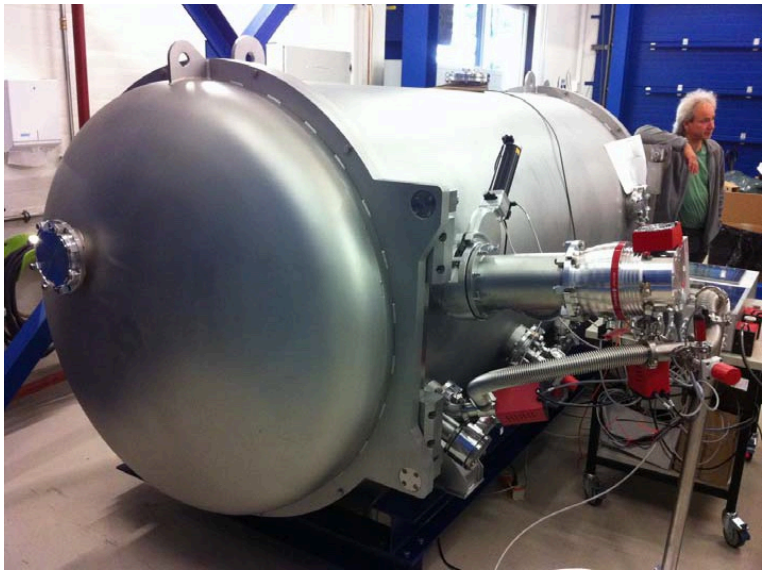

Figure 4.2: The vacuum chamber for the VIS spectrograph, with an inner diameter and length of 1400 and 3500 mm respectively [37].

	VIS channel	NIR channel
Wavelength coverage, Δλ	520-1060 nm, ([V]RIZ)	960-1710 nm, (YJH)
Detector	1 x 4kx4k e2v CCD231-84	2 x 2kx2k Hawaii-2RG, (2.5 μm cutoff)
Wavelength calibration	Th-Ne lamps & Fabry-Pérot etalon	U-Ne lamps & Fabry-Pérot etalon
Working temperature, T_{work}	285.000 ± 0.005 K	140.000 ± 0.005 K
Spectral resolution, R	94.600	80.400
Mean sampling	2.8 pixels	
Mean inter-fiber spacing	7.0 pixels	
Cross disperser	Grism, LF5 glass	Grism, infrasil
Reflective optic coating	Silver	Gold
No. of orders	61	28
Échelle grating	2 x Richardson Gratings R4 (31.6 mm^{-1})	

Table 4.1: Specification summary for the VIS and NIR equipment used to obtain the Carmenes dataset [38].

directly to the 3.5 m telescope with optical fibres, providing stable input illumination for the spectrographs. See Table 4.1 for the full equipment specification summary for the Carmenes VIR and NIR spectra. Although the observations and subsequent data collection was out of scope for this project, a total of around 100 MB of Carmenes data was downloaded.

BT-Settl is a theoretical dataset, downloaded from SVO's website, where the total database contains around 556 GB of data, covering 126,000 spectra [39]. It is generated by PHOENIX's NextGen model with validity across the entire parameter range. Despite that NextGen isn't the latest version of the model, it provides the best spectra coverage for M-type stars, which is why it was chosen for this project. A uniform grid spanning over ($1200 \leq T_{eff} \leq 4900\ K$) in steps of $100\ K$, $\log g = -0.5$ to 6.0 in steps of 0.25 dex and metallicity from -2.5 to $+0.5$ in steps of 0.5 is provided by the model. The gravitational settling of sedimentation, supersaturation, nucleation and mixing is taken into account by the model [40]. 62 GB was totally downloaded, covering a wide range of temperatures, surface gravities and metallicities.

4.2 Data Comprehension

4.2.1 Carmenes Data

The high-resolution Carmenes datasets are stored in separate VIS and NIR files, named according to the date and time of the observations and the spectra. The wavelength ranges are divided into orders, where the VIS files constitutes of 61 orders (in the range $520 - 1060\,nm$), each one with around 4096 measurements, yielding some 250,000 measurements in total per file. The equivalent numbers for NIR are 28 orders (range $960 - 1710\,nm$) and 4080 measurements, resulting in around 114,000 total measurements.

Table 4.2 shows the ten first rows of the first order for a raw NIR input file. The equivalent table for VIS can be found in Appendix A. Wavelengths, Flux 1 (from a stellar) and Flux 2 (from a black body) are stored in the files. As a result of insufficient, or no photons reaching the spectrograph in some edging wavelengths, these flux positions have been assigned a NaN (Not A Number) value. The reason for this is that there's not enough information for the estimations to be made, and as a result, the beginning and end of many orders for fluxes contains NaN. These will later be removed in the pre-processing steps.

	Wavelength (Angstrom)	Flux 1	Flux 2
0	9603.606112	NaN	NaN
1	9603.659492	NaN	NaN
2	9603.712867	NaN	NaN
3	9603.766237	NaN	NaN
4	9603.819602	NaN	NaN
5	9603.872962	NaN	NaN
6	9603.926317	0.082979	0.002337
7	9603.979667	0.107247	0.002717
8	9604.033011	0.105707	0.002713
9	9604.086350	0.113610	0.002797

Table 4.2: The first 10 rows in the first order from a raw Carmenes NIR data input file. Each datafile constitutes of 28 orders , resulting in around 114,000 measurements. The equivalent numbers for VIS are 61 orders and around 250,000 measurements.

Three VIS (top) and their three NIR equivalents (bottom) raw input data files are plotted in Figure 4.3. Inspecting the plot, we can confirm that the energy level, Flux, for the visual wavelengths is both higher, and with larger variations, than the NIR's by a factor of around $1e7$. Only the Flux 1 data has been plotted in relation to the wavelength, and from here on, *Flux* (or *flux*) will exclusively refer to Flux 1 as we are only interested in flux from stellars.

Due to the physical design of the sensors, there's space between the orders, something which is more evident in the NIR plot and for the larger wavelengths. A common organisation of the Carmenes dataset is therefore needed, where the first step is described in chapter 4.3 and then further prepared in chapter 4.4 along with the BT-Settl dataset.

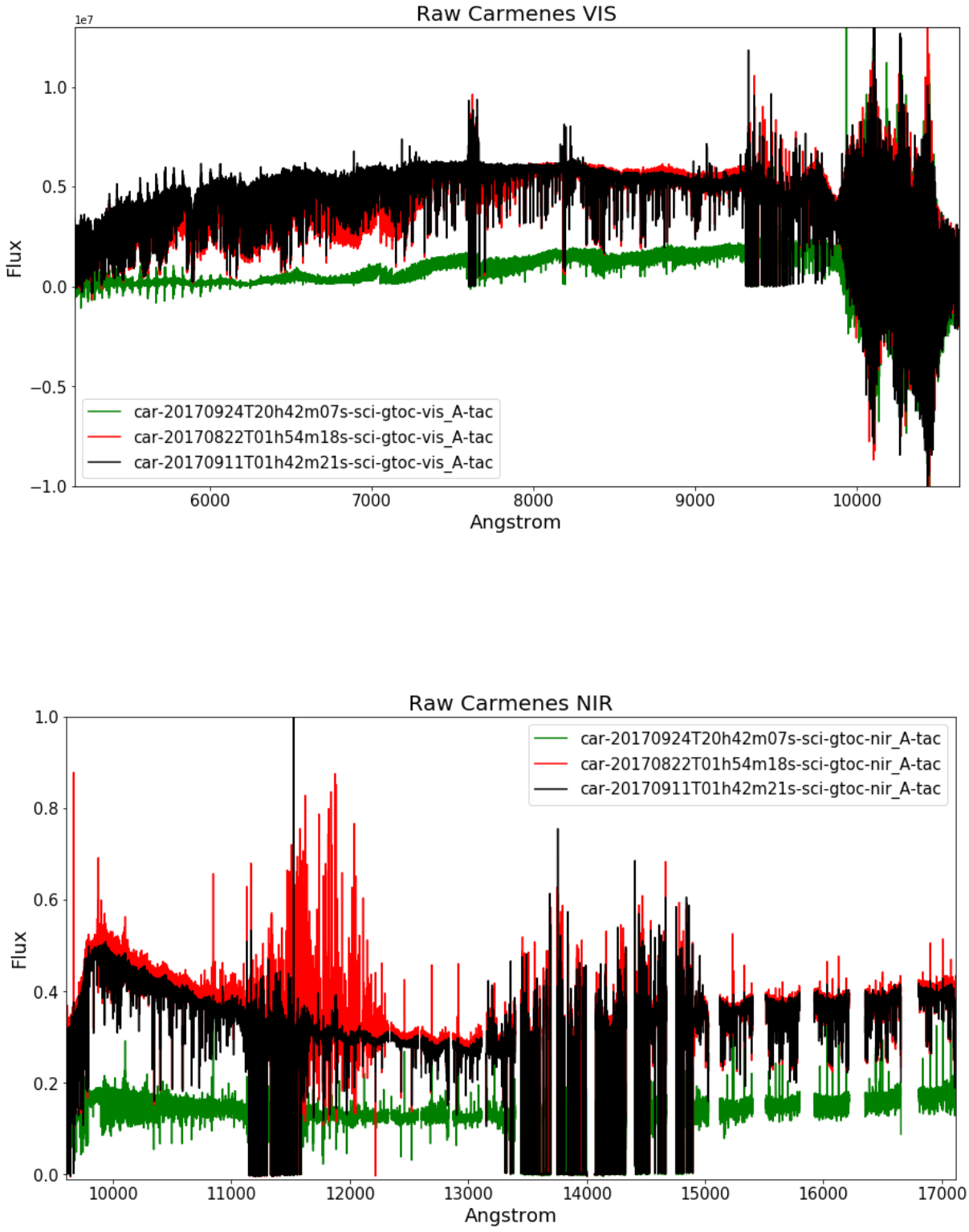

Figure 4.3: Three different raw datasets plotted in both the VIS (above) and NIR (below) spectra. The names constitute of the date and time when the observations were carried out, as well as whether it's VIS or NIR wavelengths.

4.2.2 BT-Settl Data

The BT-Settl data set constitutes of in total 4046 files, each one with between 350,000 and 1,200,000 measurements, covering the full spectra (both VIS and NIR). The relevant parameter ranges for M-type stars are chosen as temperature ($1200 \leq T_{eff} \leq 4900\,K$) in steps of 100 K, the surface gravity in the range $-0.5 \leq \log g \leq +5.5$ with $0.5\,cm/s^2$ steps, and the metallicity span ($-0.5 \leq M/H \leq +4.0$) in 0.5 intervals. Values for alpha enhancement are also included with the range $0.0 \leq Fe/H \leq 0.4$ in 0.2 intervals. However, these will later be discarded for their small influence as well as for being partly included in the metallicity. Neglecting them will thus have very small impact.

Three BT-Settl files are plotted and displayed in Figure 4.4. One with parameters $T_{eff} = 1200\,K$, $\log g = -3.0$ and $M/H = 0$ (black), another with $T_{eff} = 1200\,K$, $\log g = -3.5$ and $M/H = 0$ (red) and a last one with $T_{eff} = 1200\,K$, $\log g = -4.0$ and $M/H = 0$ (green). From observing the figure, it is clear that the absolute majority of the flux is concentrated on the central left part, lower

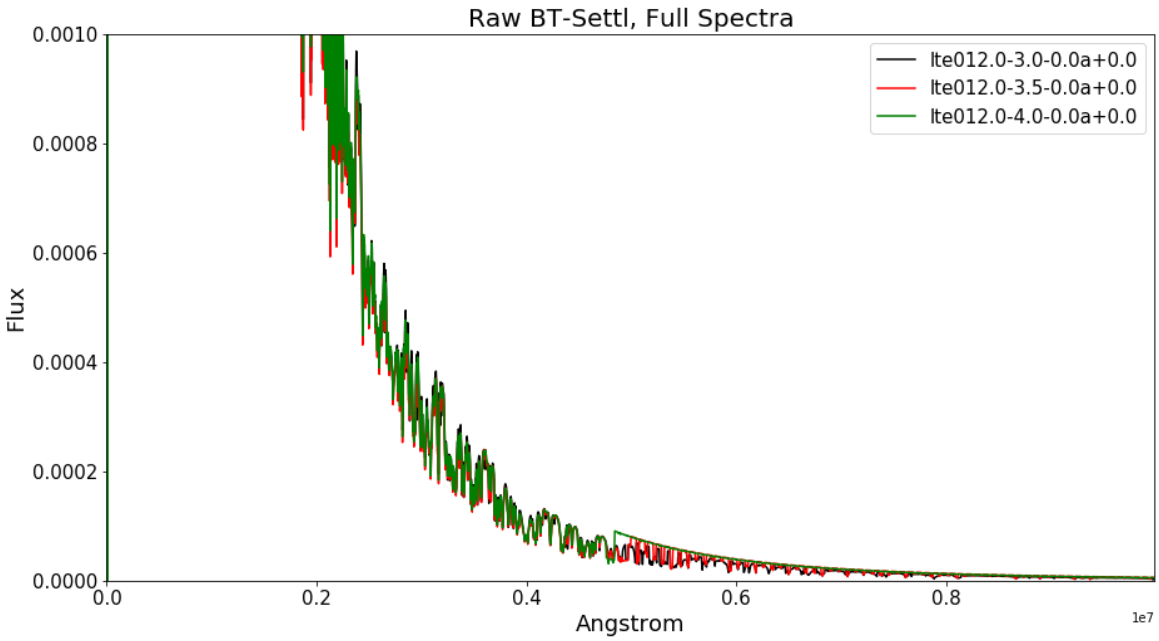

Figure 4.4: The full spectra for the raw BT-Settl data plotted for three files. It is evident that the majority of the Flux is concentrated on the lower left side of the spectra. We saw the same pattern comparing the Carmenes data in chapter 4.2.1.

wavelength, side of the spectra. The same pattern was observed in the Carmenes data files, where the VIS's flux was larger than the NIR's with a magnitude of $1e7$.

Table 4.3 displays ten rows in a raw BT-Settl input data file with parameter values $T_{eff} = 1200\,K$, $\log g = -3.0$ and $M/H = 0$, for wavelengths and stellar Flux. It can be noted that the flux values are extremely low for the lowest ($<\approx 0.2e^7$) wavelengths. Due to resolution issues in Figure 4.4, it might seem like the plot is contradicting the data displayed in Table 4.3. However, closer investigations of the raw BT-Settl files confirms that this is not the case. See Appendix B for access to these files (they are too big to fit into this report).

lte012.0-3.0-0.0a+0.0	Angstrom	Flux
190	0.000	8.040810e-98
191	0.000	8.040810e-98
192	0.000	8.040810e-98
193	0.000	8.040810e-98
194	0.001	8.040810e-98
195	0.001	8.040810e-98
196	0.001	8.040810e-98
197	0.001	8.040810e-98
198	0.001	8.040810e-98
199	0.001	8.040810e-98

Table 4.3: Raw BT-Settl input data as displayed in Figure 4.4 in black, with parameter values $T_{eff} = 1200\,K$, $\log g = 3.0$ and $M/H = 0$. Very small flux values are detected for the lower wavelengths.

4.3 Pre-processing the Carmenes Data Set

The first step in pre-processing the Carmenes data set is to cut the data to align all corresponding orders with each other. That is, all the first orders in all the VIS files have to be adjusted so they start and end at the same wavelengths, and all the second orders have to be aligned to start and end at the same wavelengths and so on. As the NaN values are removed when loading the data, the orders don't cover the same wavelength span anymore. This is solved by

choosing the highest common wavelength among all the lowest wavelengths for the first order in all VIS files and choose this as the new common lowest wavelength for the first order. All lower wavelengths in that order will be discarded. Conversely, the lowest wavelength among the highest wavelengths for the first order in all VIS files will be chosen as the new common highest wavelength for the first order. All higher wavelengths for that order will be discarded. The same procedure will be repeated for all the 61 orders and for all the VIS files. The NIR files will be processed in the same way.

There exist some smaller differences in wavelengths (normally $\leq 0.05\%$) between equivalent wavelengths among the Carmenes data files. In order to get an exact alignment across all the data, the average wavelength for each individual measurement, or line, among all the files will be calculated replacing the current wavelengths. VIS and NIR files will be treated separately equally as before.

All the Carmenes VIS data files now share exactly the same wavelength intervals, but with different fluxes, unique for every file. The equivalent is true for the NIR data. Figure 4.5 depicts the same three datasets as displayed in Figure 4.3 for, now normalised, VIS (top) and NIR (bottom) files.

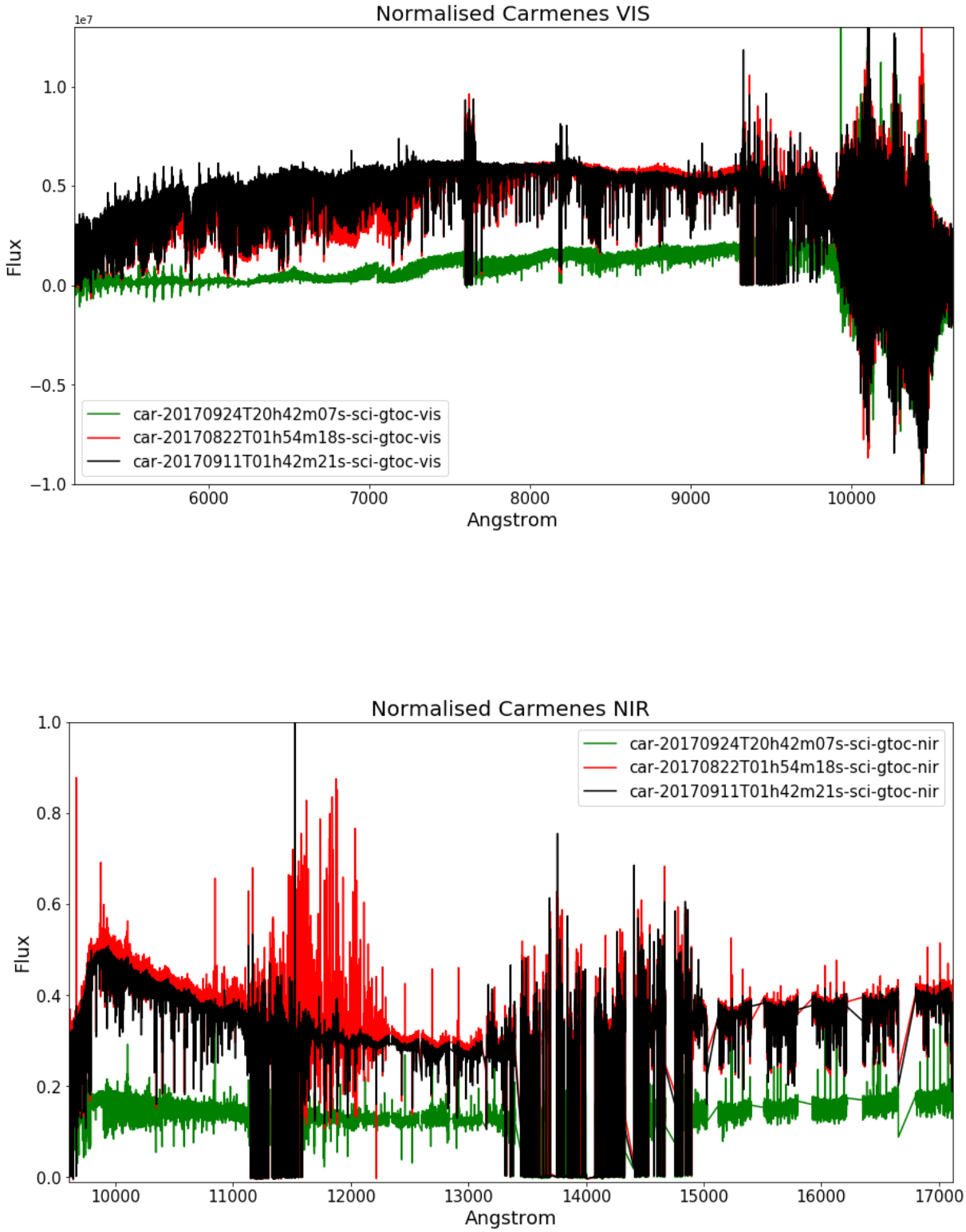

Figure 4.5: Normalised Carmenes for VIS (top) and NIR (bottom). The same three datasets as in Figure 4.3 are displayed.

4.4 Preparing for Power Matrices

To be able to analyse the data with a regression approach using CNNs, the data needs to be transformed into images, or matrices (see chapter 3.1.3). The matrices are representing the power spectra for flux per order, and must have the same wavelengths in both the Carmenes and BT-Settl datasets to be comparable.

As the synthetical BT-Settl dataset already is clean, there is no need for any substantial pre-processing other than removing the initial first rows containing meta data in each file. This is easily done when loading the data.

The main idea in this step is to try to see the BT-Settl data from the Carmenes data's perspective. We do this by using the Carmenes data's wavelengths together with the processed BT-Settl's flux values.

This is a complicated task, both in terms of coding and of optimising the process to run as fast as possible. The initial solution processed one BT-Settl file in 16 hours. With 4046 files to process, this would have taken far too long to be feasible. After several solution iterations during several weeks of work, 1,020 BT-Settl data files were eventually managed to be processed in little less than 48 hours. Due to time reasons in the project, only roughly a quarter of the total dataset was processed.

Code Snip 4.1 displays the function (*create_btsettl_data_FINAL*) performing the heavy calculations in the convolutions. Starting by loading the Carmenes data and selecting the Carmenes wavelength (CW) on the n^{th} position. The function "*find_nearest*" (see Code Snip 4.2), will subsequently find the nearest wavelength, among all the wavelengths in the loaded BT-Settl data that corresponds to CW. Choosing 50 flux values above and 50 below (nv = 50) gives a window of 100 for the calculation of these fluxes' impact on that wavelength. The choice of the window size is justified by that fluxes further away have little or no impact on that particular wavelength. Pixel resolution R is high for the Carmenes matrix and set to 82,000. The last operation performed

by the function is a standardisation of energy per order by equalling the area to 1. This will make the flux energy indifferent to the distance between the Earth and the observed star, making the data comparable.

```
def create_btsettl_data_FINAL(carmenes_data, file_names, output):
    """ Convolution of the BT-Settl and Carmenes data sets.
       'file_names' is a list of BT-Settl data file names, output is a string,
        either "NIR" or "VIS". """
    if output.upper() == "NIR":
        light = "_NIR"
    elif output.upper() == "VIS":
        light = "_VIS"
    else:
        print("Some kind of in(out)put error has occured")

nv, R = 50, 82000
for name in file_names:
    order = 0
    data = get_btsettl_data_byName(name)
    name = name + light
    for m in range(len(carmenes_data[0])):
        small_list = []
        for n in range(len(carmenes_data[0][0])):
            """Perform the necessary calculations"""

            # Get the n'th wavelength value from the carmenes dataset,
            # store it in the variable 'look_here'
            look_here = float(copy.deepcopy(carmenes_data[0][m].values[n,0]))

            # Select the BT-Settl wavelengths
            ar = data['Angstrom'].values

            # Get the wavelength among the BT-Settl wavelengths
            # that is closest to the 'look_here variable'
            use_this = find_nearest(ar,look_here)

            # Get the corresponding index
            itemindex = np.where(ar==use_this)[0]

            # Convert to integers and floats
            ind = int(itemindex[int(0)])
            use_this = float(use_this)

            # Convolution intervals
            xsg1 = ind - nv
            xsg2 = ind + nv

            # Apply these intervals to the BT-Settl data
```

```
        xsgA = data["Angstrom"].iloc[xsg1:(xsg2+1)]
        xsgF = data["Flux"].iloc[(xsg1-1):xsg2]

        # Calculate the impact from the chosen BT-Settl flux values
        xt = copy.deepcopy(use_this)
        sgm = float(use_this/(R*2*np.sqrt(2.*np.log(2.))))
        flt = np.exp(-(xsgA - xt)**2/(2*sgm**2))/(np.sqrt(2*np.pi)*sgm)

        # Calculate the sum of this impact, reverse xsgF before multiplying
        xsgA2 = data["Angstrom"].iloc[(xsg1-1):(xsg2+1)]
        the_sum = np.sum(np.diff(xsgA2)*flt*xsgF[::-1])

        # Add the newly calculated flux to the Carmenes data to the adequate
        # position
        temp = copy.deepcopy(carmenes_data[0][m].iloc[n,:])
        temp["Flux"] = the_sum
        small_list.append(temp)

    # Add some space
    temp_list = []
    temp_list.append(' ')
    temp_list.append(' ')
    temp_list.append("#order: {0}".format(order))
    pd.DataFrame(temp_list).to_csv(create_btsettl_path(name),
                                   header=False, index=False, mode="a")
    order += 1

    # Create the dataframe
    df = pd.DataFrame(small_list)

    # Convert the df to numerical values
    df = df.apply(pd.to_numeric, errors='ignore')

    # Calculate the Rolling mean for the Flux and equal the "area below" to 1.
    temp_df = (df["Flux"].rolling(2).mean()[1:]*(np.diff(df["Angstrom"])))
    df["Flux"] = df["Flux"]/temp_df.sum()

    # Write to file
    df.to_csv(create_btsettl_path(name), header=False, index=False, mode="a")
```

Code Snip 4.1: Function for the BT-Settl and Carmenes convolution calculations.

The substantial speed improvements were achieved mainly through two implementations: more efficient and smarter functions, and by using multiprocessing.

The biggest impact on the speed was the implementation of the "find_nearest" function, displayed in Code Snip 4.2. The function cut the

processing time per file around 30 times, from 16 hours to around 30 min, by using a bisection method rather than iterating over every single value. This method is normally a lot faster for larger datasets, with the only requirement that the data should be monotonically increasing, which is the case for the wavelengths (this is checked before initiating).

The idea behind this method is to divide the data into two equal halves, subsequently checking whether the desired value is higher or lower than the mid threshold dividing the two halves. If higher, the half above the threshold will be chosen, and if lower, the other half will be selected. The same procedure will be repeated until the desired value's closest counterpart is found in the data. Hence, this approach divides that amount of data to be searched over by half for every iteration.

```
def find_nearest(array,value):
    '''Given an 'array' , and given a 'value' , returns a value j such that 'value' is
      between array[j] and array[j+1]. 'array' must be monotonic increasing. j=-1 or
      j=len(array) is returned to indicate that 'value' is out of range below and above
      respectively.'''

    n = len(array)
    if (value < array[0]):
        return -1
    elif (value > array[n-1]):
        return n
    jl = 0                                          # Initialise lower
    ju = n-1                                        # and upper limits.
    while (ju-jl > 1):                              # If we are not yet done,
        jm=(ju+jl) >> 1                             # compute a midpoint with a bitshift
        if (value >= array[jm]):
            jl=jm                                   # and replace either the lower limit
        else:
            ju=jm                                   # or the upper limit, as appropriate.
        # Repeat until the test condition is satisfied.
    up_dif = np.abs(value-array[jl-1])
    down_dif = np.abs(value-array[jl+1])
    if (value == array[0]):                         # edge cases at bottom
        return 0
    elif (value == array[n-1]):                     # and top
        return n-1
    elif up_dif > down_dif:
        return array[jl+1]
    elif up_dif < down_dif:
```

```
        return array[j1-1]
    else:
        return array[j1]
```

Code Snip 4.2: The "find_nearest" function, which when implemented, cut the processing time around 30 times, from 16 hours per file to around 30 min. Source [41], plus added functionality for this Master's Thesis.

By using a computer with multiple processors, it was possible to implement multiprocessing, and by this cutting the time even more (see Appendix B, Code Snip B2 for the code). By dedicating 16 cores, one file per core, the improvements were around 16 times. It shall be clarified that this method doesn't intend to process one file as fast as possible. Instead, by processing 16 files in parallel, it will be possible to process 16 files during the same time as one file took before. Thus, it can be argued that the processing time per file is cut by 16.

Consequently, implementing the "find_nearest" function, making use of multiprocessing and some other minor improvements, cut the total processing time per file around 750 times. From 16 hours to around 75 seconds on average.

The resulting data, now adequate for creating power matrices, is displayed in Figure 4.6 for VIS (top) and NIR (bottom). Both plots display slightly more homogeneous patterns compared to their raw equivalents in Figure 4.3.

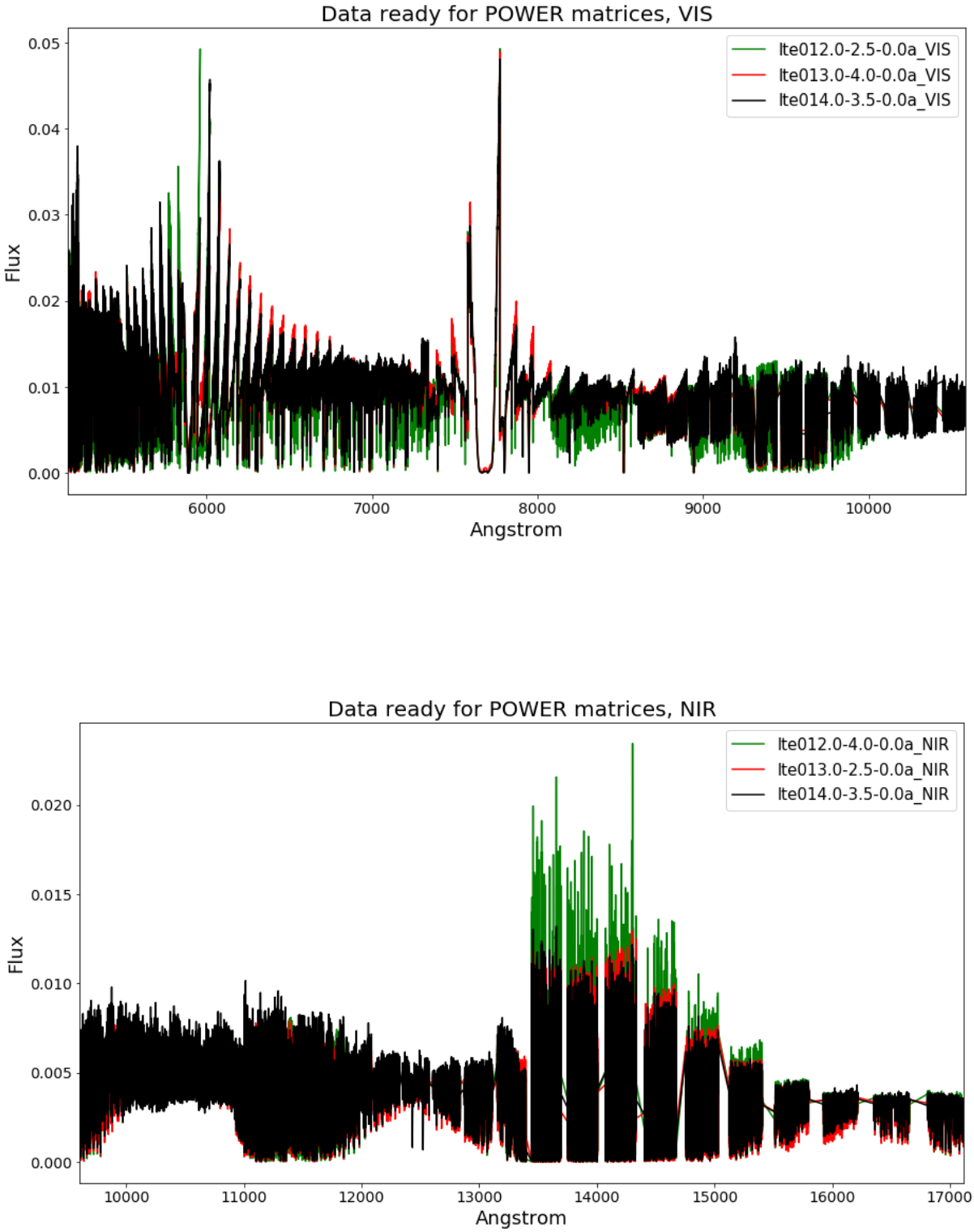

Figure 4.6: Pre-processed VIS (top) and NIR (bottom) data now ready for power matrices.

4.5 Power Matrix Creation

The chosen approach for power matrix creation is the one proposed by Torrence and Compo (1998) [42], implemented in a wavelet spectral analysis module (PyCWT) for Python [43]. The function carrying out the transformation can be found in Appendix B, Code Snip B.3.

According to research reviewed in chapter 2.1, ICA is the best transformation technique for power matrices. However, as the data constitutes of several orders for each wavelength interval, ICA would apply one transformation function for each order. To keep the current desired relation between the different orders, *Morlet Wavelet* function is a better fit. It's a fixed function and would thus apply the same transformation on all orders. Morlet Wavelet with ($\omega_0 = 6$) is chosen.

The power matrices are rescaled to the interval [0, 255], before the datatype is converted to *float16,* or *"short"*. This is done to save disk space and loading time during training later on. With the initial choice of *float64* datatype, the resulting power matrices occupied around 400 MB (for NIR) and 600 MB (for VIS) each. The datatype choice reduced the disk space more than tenfold, but with the cost of a slightly loss in resolution. However, the loss was considered negligible and thus acceptable.

2044 power matrices were created, out of potentially 8096 from the data collected, where the total size of the created matrices was around 93 GB. The spectrogram for the power matrix with parameters $T_{eff} = 1200\,K,\ \log g = -3.5$ and $M/H = 0$, NIR order 0, is displayed in Figure 4.7. Order 0 refers to the first order as Python is a 0-indexed programming language. Although trivial in the context, the first position in a list is always the 0^{th} position.

Table 4.4 provides a comprehensive overview over the created power matrices.

	VIS	NIR
Matrices created	1020	1024
Size	7747 x 2871	3724 x 4073
Number of Orders	61	28
Total size on disk (GB)	55	38

Table 4.4: Summary over the created power matrices.

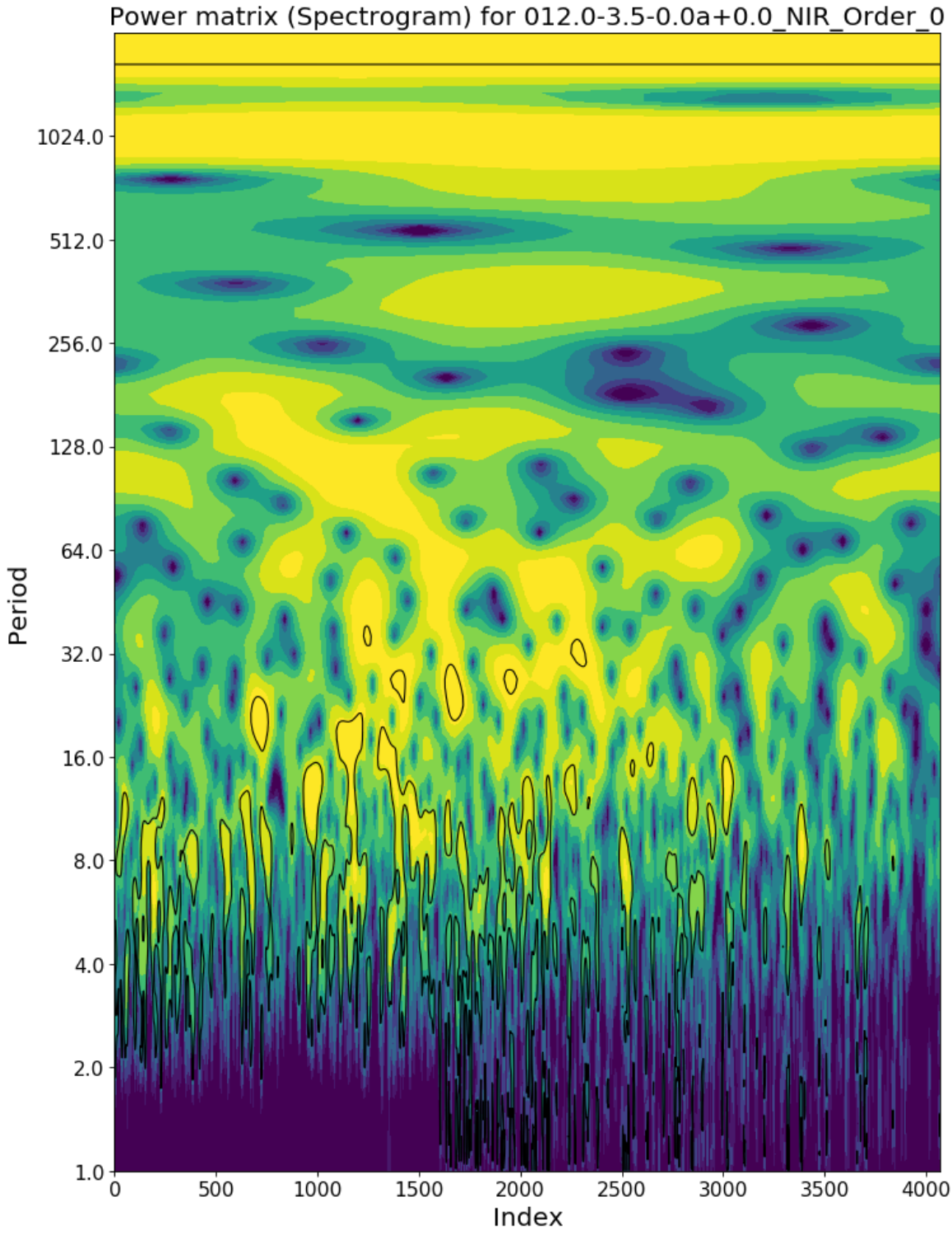

Figure 4.7: Spectrogram as a result of plotting the power matrix with parameters $T_{eff} = 1200\ K$, $log\ g = -3.5$ *and* $M/H = 0$ *for NIR, Order 0. This matrix will subsequently be processed with the CNN.*

Chapter 5 CNN Analysis

5.1 CNN Architecture

5.1.1 Main CNN Architecture

Designing a CNN architecture is a complicated task, many choices have to be made, regarding number of layers of each type, how to stack them in relation to each other, hyperparameters for each type of layer such as activation functions, kernel size, stride etc. etc. The combination of different parameters is so large that a conventional approach is to find a great solution on a similar problem, use that architecture and keep on optimising it for the problem at hand, while testing the performance on the test data. Although the hardware used for the project was fairly powerful (2x8 cores 3,7 MHz AMD CPU, 16GB RAM, 4GB GeForce GTX GPU), careful considerations had to be taken into account to keep the architecture as nimble as possible in relation to the input dimension.

Following this approach, the architecture displayed in Figure 2.2, designed by the NVIDIA team, was chosen as starting point. However, applying this model directly generated 7,4 billion trainable parameters for the architecture, and would impossibly fit within the capacity of the hardware. Substantial changes had to be made.

As the NVIDIA team's input size was small ($66\ x\ 200$) in comparison to the power matrices' size of $3724\ x\ 4073$ for NIR and $7747\ x\ 2871$ for VIS, more Conv layers were added to keep the total amount of parameters at a minimum. By increasing the amount of Conv layers and lowering the amount of fully connected Dense layers, the amount of network parameters was kept to a minimum without compromising performance [29]. Initiating the first Conv layers with a lower dimensionality in the output space, i.e. filters ($filters =$

16), and successively double this to $filters = 1024$ in the last layers, while doing the opposite with the kernel size, from $(7\ x\ 7)$ to $(3\ x\ 3)$, was in accordance with the approach. ReLU activation function was used for all the hidden layers as it's much faster when training more complex models, see Chapter 3.2.1.

A common practice is to insert Pooling layers in intervals in-between successive Conv layers in a CNN structure. Their main objective is to reduce the spatial size and thus, reduce the amount of parameters, leading to more efficient model training. They also reduce overfitting and by this generate a model with higher performance on unseen data. Choosing Max Pooling layers over Average or Global pooling layers is motivated by results and conclusions drawn in earlier studies [44] [45].

A Flatten layer is added to flatten the input to two dimensions, before three fully connected Dense layers are added to step wise lower the parameter size. The last Dense layer's output dimension is a $1\ x\ 3$ vector, referring to the three physical parameters this project aims to estimate. As our approach has been to regress the parameter values, rather than classify them, a linear activation function must be chosen in this last Dense layer. The two previous Dense layers are still activated through a ReLU activation function.

Despite minimising the parameter size extensively, the computer's GPU memory wasn't able to handle the big input sizes. Errors were constantly thrown when the model tried to fit the initial or one of the subsequent layers. As a result, the power matrices are down-sized twice for NIR (to $1862\ x\ 2036$) and three times for the slightly larger VIS matrices (to $2582\ x\ 957$). See chapter 5.2 for further explanations of how this was solved. Several other combinations were tried and evaluated as well, see chapter 5.1.2.

All in all, the architecture constitutes of 8 Conv layers, in conjunction with 8 Max Pooling layers, followed by a Flatten layer and three fully connected Dense layers. The total amount of trainable parameters when NIR matrices are fitted is 19,6 million, while 18,9 million for the slightly more resized VIS matrices. Table 5.1 displays the final main CNN architecture fitted on NIR

Layer (type)	Output Shape	Param #
conv2d_65 (Conv2D)	(None, 1856, 2030, 16)	2368
max_pooling2d_65 (MaxPooling	(None, 928, 1015, 16)	0
conv2d_66 (Conv2D)	(None, 922, 1009, 32)	25120
max_pooling2d_66 (MaxPooling	(None, 461, 504, 32)	0
conv2d_67 (Conv2D)	(None, 455, 498, 64)	100416
max_pooling2d_67 (MaxPooling	(None, 227, 249, 64)	0
conv2d_68 (Conv2D)	(None, 223, 245, 128)	204928
max_pooling2d_68 (MaxPooling	(None, 111, 122, 128)	0
conv2d_69 (Conv2D)	(None, 107, 118, 256)	819456
max_pooling2d_69 (MaxPooling	(None, 53, 59, 256)	0
conv2d_70 (Conv2D)	(None, 49, 55, 512)	3277312
max_pooling2d_70 (MaxPooling	(None, 24, 27, 512)	0
conv2d_71 (Conv2D)	(None, 22, 25, 1024)	4719616
max_pooling2d_71 (MaxPooling	(None, 11, 12, 1024)	0
conv2d_72 (Conv2D)	(None, 9, 10, 1024)	9438208
max_pooling2d_72 (MaxPooling	(None, 4, 5, 1024)	0
flatten_10 (Flatten)	(None, 20480)	0
dense_29 (Dense)	(None, 50)	1024050
dense_30 (Dense)	(None, 20)	1020
dense_31 (Dense)	(None, 3)	63

Total params: 19,612,557
Trainable params: 19,612,557
Non-trainable params: 0

Table 5.1: The main CNN architecture fitted on a NIR matrix, constituting of eight Conv, in conjunction with, eight Max Pooling layers, one Flatten layer and three fully connected Dense layers. In total 19,6 million trainable parameters

matrices while Figure 5.1 depicts it in a more illustrative way. The implemented *Keras* code is found in Appendix B, Code Snip B.7.

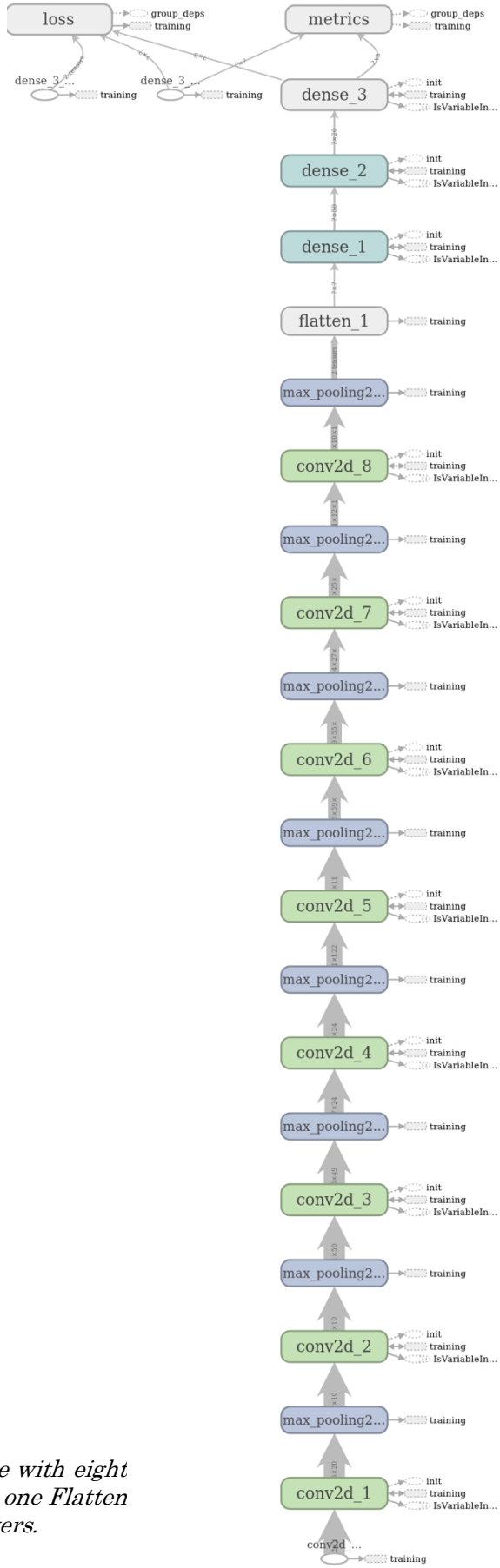

Figure 5.1: Main CNN Architecture with eight Conv and eight MaxPooling layers, one Flatten layer and three Fully connected layers.

5.1.2 Alternative CNN Architectures

As the main CNN architecture was fairly complex with many parameters to train, other more simple alternatives were implemented and tested. In the context of studying earlier work, by L.M. Sarro et al. [2017] among others, see chapter 2.3, it was suspected that effective temperature was stronger correlated with the spectra than, in particular, metallicity. Thus, a more downscaled input size could still yield satisfying estimations for T_{eff}, while at the same time decreasing training time and hardware requirements.

As a consequence, the input size was downscaled, not only twice as for the main NIR model and three times for the VIS model, but 5, 10, 20 and 30 times as well. The same methodology as described in chapter 5.1.1 was applied in terms of filter and kernel sizes. However, as the size of the data fed into the model was substantially smaller, the last Conv layers threw "*Negative dimension size*" errors. The reason for this was the negative spatial size obtained after several down sampling operations by the hidden Pooling layers. This was solved by removing the last Conv and Max Pooling layers as the down-sizing was increased.

The resulting architectures have 7,5 million (down-sizing 5), 4,5 million (down-sizing 10), 1,0 million (down-sizing 20) and 340,000 (down-sizing 30) parameters. Tables are to be found in Appendix C for the resulting architectures while Figures 5.2 – 5.4 below depicts them in a more illustrative way.

It shall be noted that the architectures for 10 and for 20 down-sizing are the same, even though the number of parameters is 4.5 times as high for the first one.

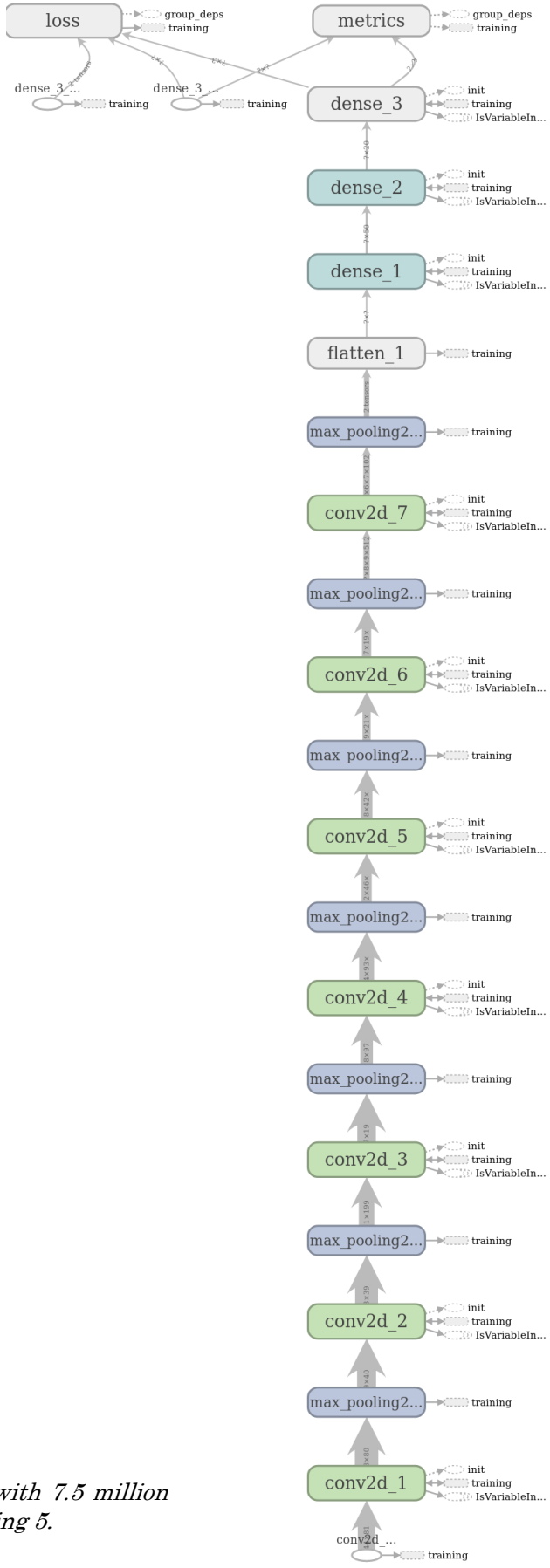

Figure 5.2: CNN Architecture with 7.5 million hyper parameters and down-sizing 5.

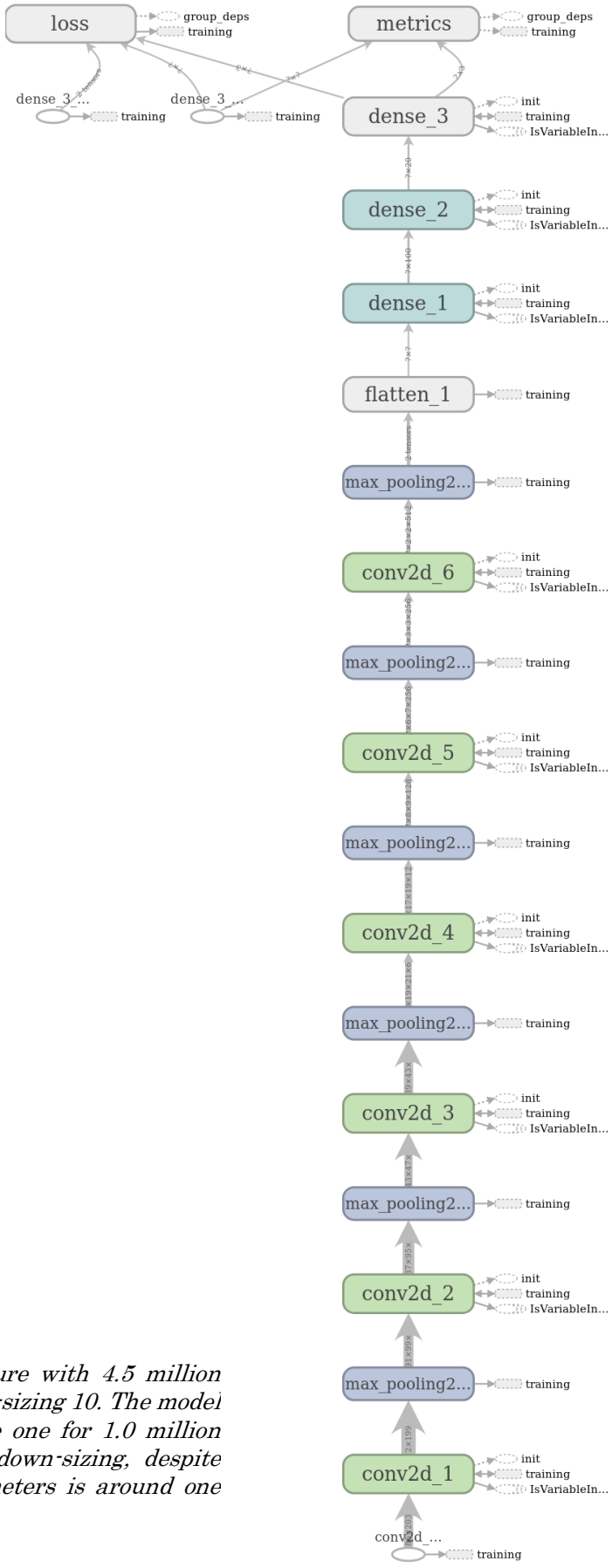

Figure 5.3: CNN Architecture with 4.5 million hyper parameters and down-sizing 10. The model architecture is equal to the one for 1.0 million hyper parameters and 20 down-sizing, despite that the number of parameters is around one fifth.

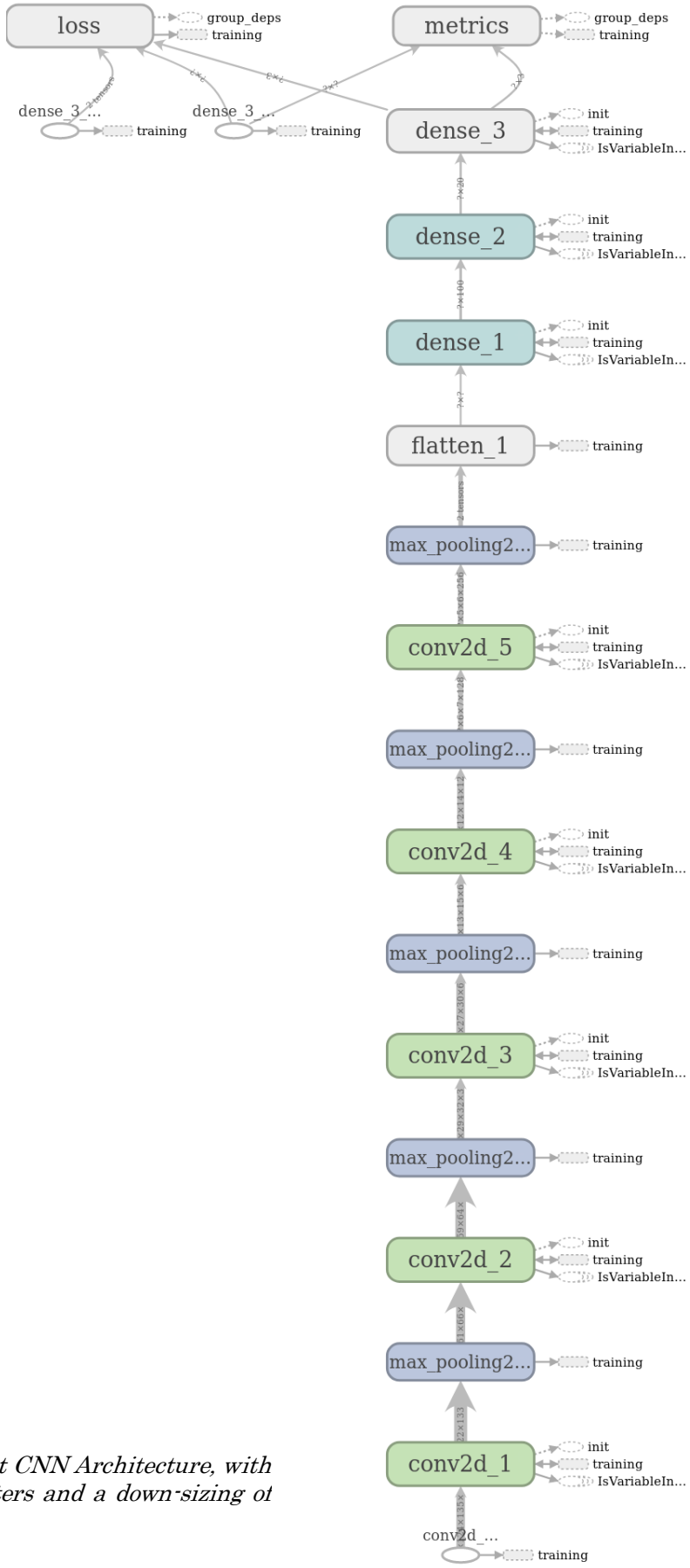

Figure 5.4: The smallest CNN Architecture, with 340.000 hyper parameters and a down-sizing of the input data by 30.

5.2 Feeding the Models with Data

Although the model will be trained on only 80% of the data, it still won't fit into the computer's RAM or GPU memory at once. As a result, it has to be fed concurrently during training. There are mainly two solutions for this in Keras; one is to create a Generator and the other is to create a Sequence. They both generate batches of data to feed the model during training. However, *Keras* recommends Sequence if using multiprocessing, as it will guarantee that the model will only train once on each sample per epoch, i.e. number of iterations over the training set, which is not the case for Generators [46]. This is to minimise the risk of overfitting, and was thus chosen. Nevertheless, a Generator function was also developed for comparable reasons on single core processing. Code Snip 5.1 displays the implemented Sequence class, while the Generator function can be found in Appendix B.

```
class My_Sequence(Sequence):
    """ Generates batches of training data and ground truth.
        Inputs are the image paths and batch size.
    """
    def __init__(self, image_paths, batch_size):
        self.image_paths, self.batch_size = image_paths, batch_size

    def __len__(self):
        return int(np.ceil(len(self.image_paths) / float(self.batch_size)))

    def __getitem__(self, idx):
        batch = self.image_paths[idx * self.batch_size:(idx + 1) * self.batch_size]
        matrices, parameters = [], []
        for file_path in batch:
            mat, param, name = get_Matrix_and_Parameters(file_path)

            # Transform the matrix from 2D to 3D as a (mat.shape[0], mat.shape[1])
            # RBG image. Rescale its values to [0,1]. Set "preserve_range=True" to not
            # rescale the matrix, and by this saving memory and load time.
            mat = skimage.transform.resize(mat, (mat.shape[0]//down_size,
                                                 mat.shape[1]//down_size, 3),
                                           mode='constant', preserve_range=True)
            param = MMscale_param(param, name)          # Rescale the parameters
            mat = normalise(mat)                        # Rescale the matrix to [0, 1]
```

```
            matrices.append(mat)
            parameters.append(param)
        MAT, PAM = np.array(matrices), np.array(parameters)
        PAM = np.reshape(PAM, (PAM.shape[0], PAM.shape[1]))

        gc.collect()                                    # Garbage Collector

        return MAT, PAM
```

Code Snip 5.1: Displaying the Sequence class used to feed the model with data, batch-by-batch, taking advantage of multiprocessing.

The development of the Sequence class and the Generator function was inspired by the examples provided in *Keras* documentation [46] [47]. The input constitutes of the power matrices' file paths and the batch size. During each iteration, a batch-sized chunk of the file paths is chosen and the corresponding power matrices, with its parameters and name, are loaded with the help of the function *get_Matrix_and_Parameters() (see Appendix B for the code snip).* Each matrix is subsequently transformed from a 2D matrix to a 3-dimensional RBG image and down-sized the desired amount (2, 5, 10, 20 or 30 times), before both the parameters and the matrix are normalised to the interval [0, 1]. This is done through the use of the *scikit-image* library for Python. It is a common and powerful library for image processing where interpolation is used for down-sizing [48].

Normalising often leads to faster solution convergence, and thus, training [49]. Some data collection is subsequently done to organise the output into desired dimensions before it returns the entire batch. This procedure continues for $\frac{length\ of\ the\ dataset}{batch\ size}$ number of times and is repeated *epoch* times.

5.3 Train, Validate and Test Sets

As pointed out in chapter 3.4, the total dataset will be split 80% for the train set, 10% for the validation set and the last 10% for the test set. To decrease the risk of overfitting for some specific features, while at the same time being able to reproduce the results, the data, i.e. file paths, are randomly shuffled in a

controlled way before split. The resulting sets for NIR are 819 matrices for training, 102 for validation and 103 for testing, while the equivalent numbers are 816, 102 and 102 for VIS.

5.4 Normalising

Both the stellar parameters and the power matrices are scaled, or normalised, to the interval [0, 1] for faster convergence and thus, lower training time. The solution for the matrices is quite straight forward as there isn't any need for reversing the scaling afterwards. The calculation is followed by the equation:

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{5}$$

Where x_{new} is the new, normalised value, x is the original value and x_{min} and x_{max} are the min and max values in the power matrix respectively. This is also referred to as a min-max normalisation since the scaling is done in reference to the min and max values in the data.

The stellar parameters have to be addressed slightly different, owing to the fact that the exact relation between the min and max values have to be stored to enable the scaling to be reversed after training. This is done through the library *scikit-learn* and by making use of the class *MinMaxScaler* [50]. By creating a MinMaxScaler with the exact name as corresponds to the stellar parameters (the power matrix's name), and fitting it onto the data, the scaling variables will be stored in the scaler, which later can be called by the exact name to reverse the process. Two functions, *MMscale_param()* and *Un_scale_data(),* are created for this purpose, while dictionaries, *many_MinMaxScalers_NIR_param()* (and an equivalent for VIS), are created for storing the MinMaxScalers. See Appendix B, Code Snip B.4 for the corresponding implementation.

5.5 Extracting the Physical Stellar Parameters

In accordance with the objective of this Master's Thesis to determine the physical stellar parameters, the three parameters have to be extracted from the power matrices' names and stored in a vector. This is implemented within the function *get_Matrix_and_Parameters()* (see Appendix B, Code Snip B.6 for the code snip).

Although the names presents with some difficulties as they have some irregularity and with different lengths, they still have some commonalities which helps in the extraction. The approach here is to find something common to split the string by. The alpha parameter is firstly removed by splitting by an "a" or an "_". Effective temperature, surface gravity and metallicity are all split by a "+" or by a "−", starting from behind. A vector of size [1 x 3] is subsequently returned by the function.

5.6 Training the CNN Models

The same architecture but two different models will be chosen. One trained on the NIR data and one on the VIS data. Before starting the training procedure, the models have to be compiled. *Adam* (short for Adaptive Moment Estimation) is chosen as the optimiser since it has shown to work well across a wide range of network architectures and is easy to implement, computationally efficient and performs well on larger datasets [51].

Several learning rates are tested ($1e^{-3}, 1e^{-5}$, $1e^{-6}, 2e^{-6}, 2.5e^{-6}$ and $7.5e^{-6}$), but a standard $lr = 1e^{-4}$ resulted in least prediction errors and was ultimately chosen. MSE was selected as loss function.

Although Keras saves several logs to be extracted after training, it doesn't store anything on the disk for later recuperation and inspection of runs. Therefore, three *callbacks* are created. The first one, *CSVLogger* stores the error over epochs. *ModelCheckpoint* saves the best performing model on the

validation set so it can easily be loaded afterwards. This is also very useful for continuing training an already trained model if the model hasn't converged after the pre-set number of epochs chosen. *TensorBoard*, on the other hand, provides a visualisation of the CNN architecture, as the one displayed in Figure 5.1.

Subsequently, the models are fit onto the data. As the data is fed batch-by-batch, the *fit_generator* method is chosen. For efficiency reasons, Keras runs it in parallel, allowing real-time data augmentation on the CPU while the models are being trained on the GPU.

Four cores, or workers, are used to feed the data to the RAM, as more of them seems to fill up the memory killing the training process, while fewer increases the training time. *Max_queue_size*, i.e. the amount of batches to be prepared in advance during training, is set to 1. Also this for memory reasons as choosing a larger batch size was considered to be more important.

A normal batch size when training Machine Learning algorithms is around 16, 32 or 64. But as they resulted in exhaustion of the GPU memory, sizes of 4 and 8 were elaborated. The same memory issues led to the final choice of size 2. The models were subsequently trained, with batch shuffling set to *True* during 100 epochs for the NIR model and 150 epochs for the VIS model, as the errors had stopped decreasing by this time. Interestingly, the VIS models took longer to train although containing fewer, or an equal amount, of trainable parameters than the NIR equivalent. Sometimes as much as by a factor of 2.

Code Snip 5.2 displays the implemented code for training the CNN.

```
num_epochs = 100

with warnings.catch_warnings():    # Catch and ignore a User warning
    warnings.simplefilter("ignore")
    fxn()

    # Log the training
    csv_logger = CSVLogger(log_path, separator=',', append=False)

    # Save the best model
    checkpointer = ModelCheckpoint(filepath=model_path, verbose=1,
                                   save_best_only=True)
    # Save the CNN Architecture
```

```
    architecture = TensorBoard(log_dir='./Graph/NIR', histogram_freq=1,
                               write_graph=True, write_images=True, write_grads=True)

    history =
NIR_model.fit_generator(generator=my_training_batch_generator_NIR,
                        steps_per_epoch=(len(validation_paths_NIR) // batch_size),
                        epochs=num_epochs,
                        verbose=1,
                        callbacks=[checkpointer, csv_logger, architecture],
                        validation_data=my_validation_batch_generator_NIR,
                        validation_steps=(len(validation_paths_NIR) // batch_size),
                        use_multiprocessing=True,
                        max_queue_size=1,
                        workers=4,
                        shuffle=True)
```

Code Snip 5.2: The implemented code for the CNN training process.

Chapter 6 Results and Discussion

6.1 Model Validation

Before any predictions can be made with the trained models, their performance have to be validated. This is done by plotting the MSE, per epoch for both the training and validation sets. Figure 6.1 below illustrates this for both the VIS (top) and NIR (bottom) models. Note that the errors were substantially higher during the first epoch. These data points were therefore treated as outliers, and thus, removed for plotting.

The first impression of studying the curves is their somewhat noisy appearance. A reasonable explanation for this is the small batch size resulting in slightly oscillating error rates between individual epochs. See Figure 3.5 for comparable plots. With the size of the input data, the resulting size of the architecture and the subsequent amount of trainable parameters, in combination with the hardware restrictions, it can be considered as tolerable.

The error curves are overall following an acceptable descent, resulting in convergence after around 80 epochs for the NIR model and after roughly 120 epochs for the VIS model. Note that the VIS model takes longer to converge, and is thus, trained for more epochs. Total training time is approximately 18h for the NIR model and 37h for VIS. The NIR model returns a training MSE of 0.000137 and a validation error of 0.000208, while the VIS model achieves errors of 0.000140 on the training set and 0.000226 on the validation set.

Training and validation errors for both models are very similar in size, giving us confidence that the models can be trusted. Also, the errors are larger on the validation sets than on the training sets, which normally is the case. Although the errors are smaller on the training set, we shouldn't necessarily

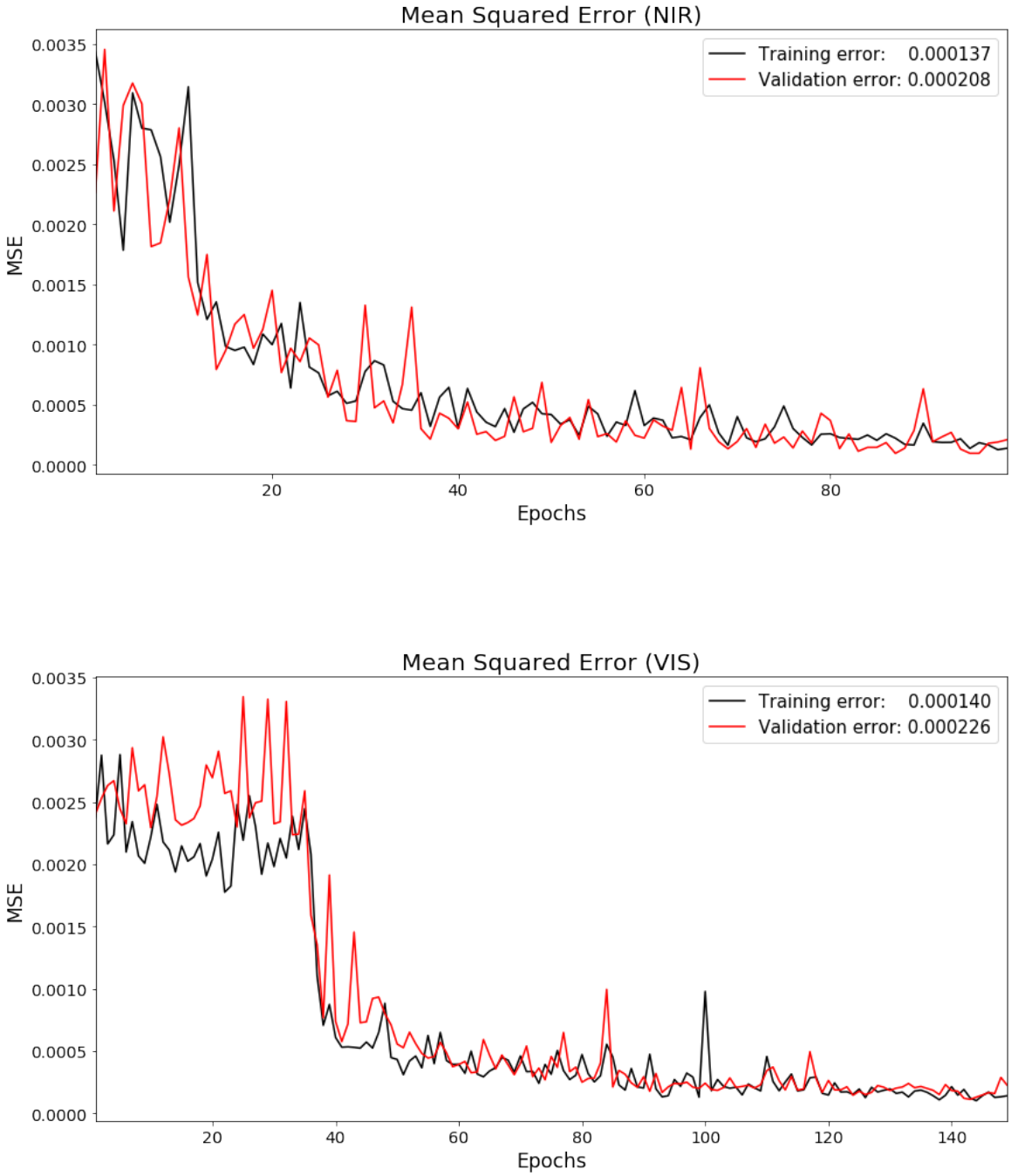

Figure 6.1: Training and validation errors for NIR (top) and VIS (bottom) for the main CNN architecture. Note that the errors are substantially higher in the first epoch. By treating them as outliers, they are removed for the plot.

draw any extensive conclusions based on this as the error curves are oscillating quite heavily. Nevertheless, this is in general a valid approach and it strengthens the confidence in the models' predictions.

Remember that the models haven't been trained on the validation set. Producing an error close to the errors obtained during training is therefore a good indicator of their validity. In other words, the models aren't overfitted on the training data and can generalise well on previous unseen data. The models should therefore be able to produce satisfying estimations on the test set a well.

6.2 Estimating Temperature, Surface Gravity and Metallicity

6.2.1 NIR

Having reassured the models' trustworthiness, they can now be used for estimations on previous unseen data. Figure 6.2 displays the predicted, in black, and true, in red, values for effective temperature (top), surface gravity (centre) and metallicity (bottom) for ten different spectra, or test runs (A through J), for the NIR model. The letters represents the same power matrices with corresponding spectra across all three sub plots.

The overall verdict is that the NIR model performs very well on the test data. Especially for T_{eff} the model estimates the true values with an impressively low MSE of 0.000079 over the test set. The equivalent errors are 0.000005 and 0.003318 for $\log g$ and M/H respectively. The relation between the errors are in accordance with the suspicions stated in chapter 5.1.2, regarding T_{eff} being the parameter with strongest correlation to the spectra. Errors are calculated within two standard deviations of the mean, meaning the model predicts with these error rates on 95% of the data. However, as for the plots concerning, no such division has been made. They display ten arbitrary spectra among all the test spectra. The total MSE over the three parameters in total on all the test data is 0.00010. It is by a factor of 2 lower than the validation error (0.000208) and roughly in the same size as the training MSE (0.000137), which

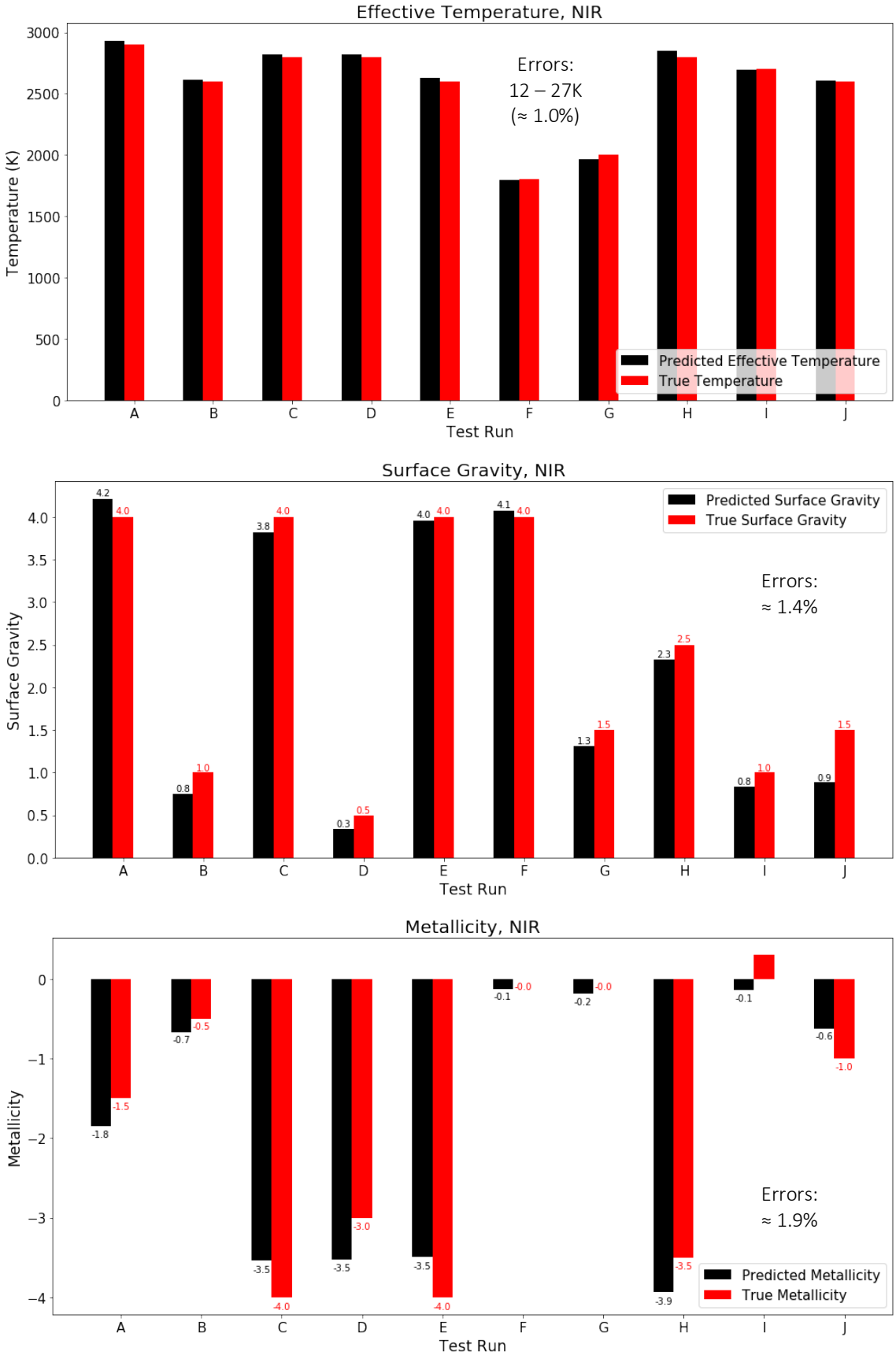

Figure 6.2: Predicted (black) and true (red) values for effective temperature, surface gravity and metallicity for the main NIR model on 10 different spectra.

can be considered as plausible. The reason for the testing error being smaller might be due to the too small size of the test set. Contemplate the less biased median squared errors confirms the conclusions drawn on the MSE observations. These errors are also displayed on each sub-plot. Comparing the 1.9% median error (not squared) for metallicity with the slightly different Machine Learning approach applied by L. M. Sarro et al. [2017], see chapter 2.3, they are some 35 times better.

Matching the calculated errors against the visualisations display a similar pattern. The disparity between red and black piles for each spectra for T_{eff} are very small, while they're slightly bigger for $\log g$ and metallicity. This seems in general to be valid across all the plotted spectra.

As some metallicities' true values are zero (e.g. F and G), the overall error percentage will be higher when the predicted values are non-zero (i.e. the difference in percentage for F and G are 100%). This might explain the larger error for M/H in relation to $\log g$ (0.003318 vs. 0.000005) although the visual disparities aren't as obvious when comparing the two plots.

See table 6.1 for a summary over mean and median squared errors for the NIR model's performance over the test set.

Performance, NIR model	
Total MSE	0.000103
MSE Effective Temperature	0.000079
MSE Surface Gravity	0.000005
MSE Metallicity	0.003318
Median Sq. Error Eff. Temp.	0.000080
Median Sq. Error Surf. Gravity	0.000196
Median Sq. Error Metallicity	0.000359

Table 6.1: Summary over the NIR model's performance.

6.2.2 VIS

Predictions for ten different (A through J) visual wavelength spectra by the VIS model are plotted in Figure 6.3, T_{eff} (top), $\log g$ (central) and M/H (bottom). Just as for NIR, the VIS model estimates T_{eff} with a very high precision of 0.000003 MSE, while slightly worse for $\log g$ (0.018612) and M/H (0.091275). They are calculated in the same way as for the NIR model. This seems plausible, also when studying the plots, as the disparity between piles are smaller for T_{eff} than for $\log g$ and M/H.

Generally speaking, the same conclusions drawn on the NIR predictions can be made on VIS. Identical as for NIR, the VIS model estimates surface gravity with a higher precision than metallicity. These observations are strengthened by the median squared errors (0.000002 vs. 0.004638). Overall MSE for the VIS model on the entire test set is 0.000136, slightly above NIR's 0.00010, but still in the same magnitude. For easier comparison, the median errors are displayed on each sub-plot, equally as for NIR. The VIS models seems to estimate T_{eff} (0.30% vs. 1.0%) and $\log g$ (0.14% vs. 1.4%) with a slightly higher precision than the NIR model.

A summary over the VIS model's performance over the training set is displayed in Table 6.2.

Performance, VIS model	
Total MSE	0.000136
MSE Effective Temperature	0.000003
MSE Surface Gravity	0.018612
MSE Metallicity	0.091275
Median Sq. Error Eff. Temp.	0.000008
Median Sq. Error Surf. Gravity	0.000002
Median Sq. Error Metallicity	0.004638

Table 6.2: Summary over the VIS model's performance.

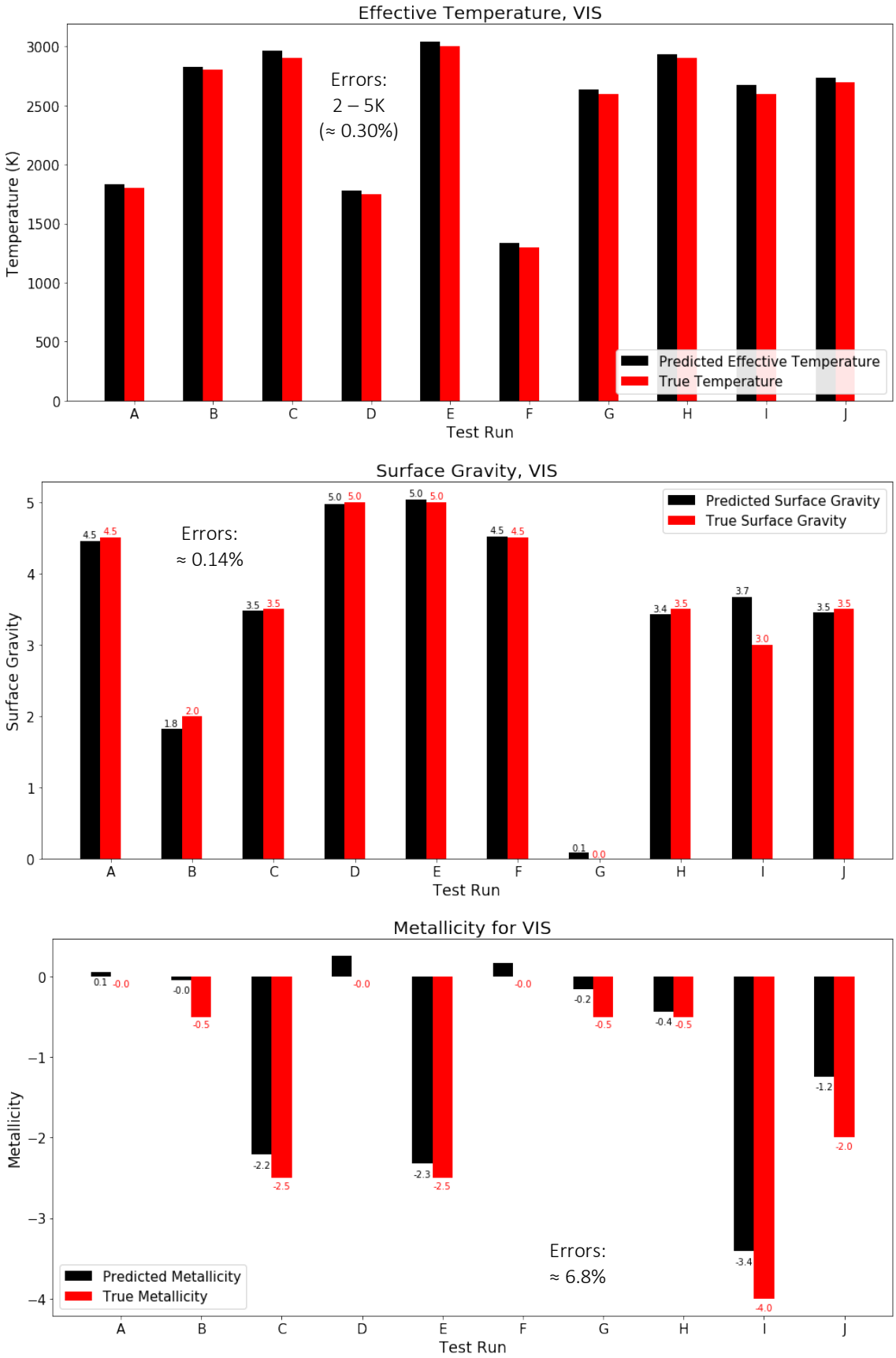

Figure 6.3: Predicted (black) and true (red) values for effective temperature, surface gravity and metallicity for the main VIS model on 10 different spectra.

6.3 Alternative CNN Architectures' Predictions

6.3.1 NIR down-sized 5, 10, 20 and 30 times

Figures 6.4, 6.5 and 6.6 respectively, display the effective temperature, surface gravity and metallicity for the four downsized NIR input models; 5, 10, 20 and 30. The MSE on the entire test set over all three physical parameters for the four models are 0.00039, 0.00081, 0.00320 and 0.00102, respectively.

In terms of effective temperature (Figure 6.4), all four models perform very well, even the most down-sized, 30 times, estimates with the very low MSE of 0.000056. Compare this with the full model's MSE on T_{eff} of 0.000079 and the more down-sized model seemingly performs better. However, the small size of the test data and the fact that the errors have been calculated within two standard deviations of the mean might affect the results in favour for the more

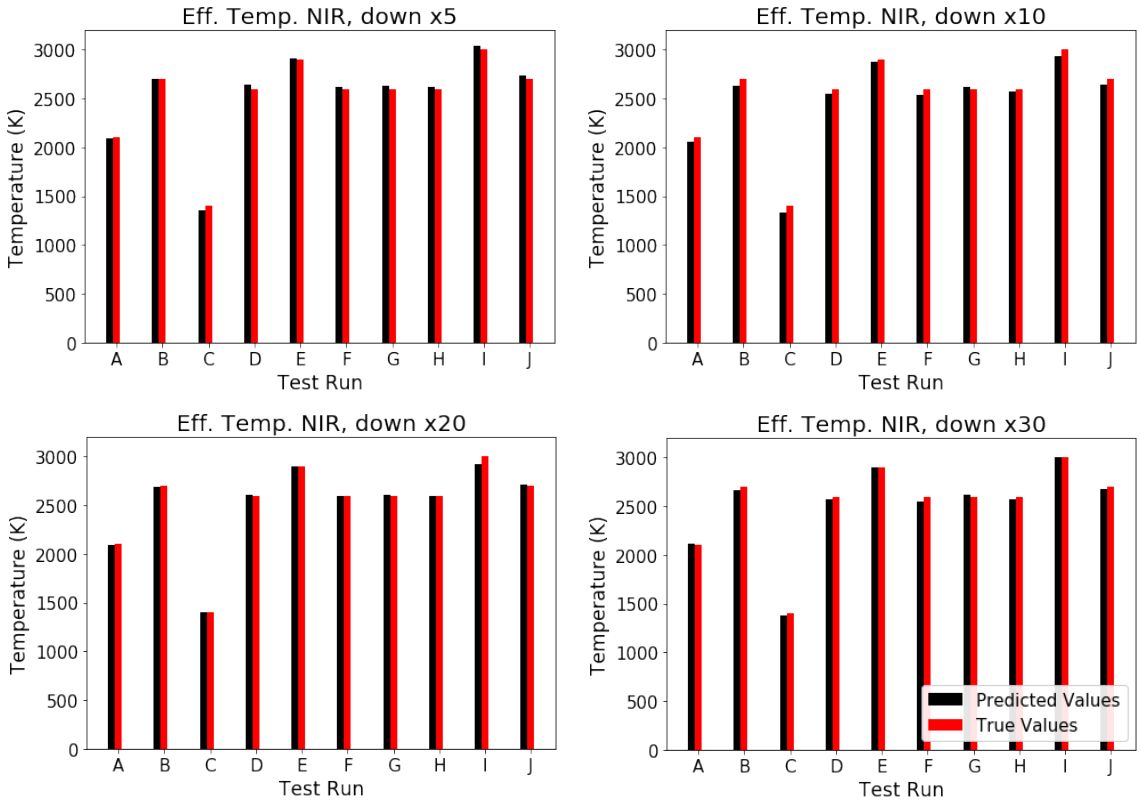

Figure 6.4: Effective temperature estimated by the 5 (top left), 10 (top right), 20 (bottom left) and 30 (bottom right) down-sized input data for the NIR models.

down-sized input data. Comparing the medians, which are very similar in size, indicate that this might be the case. Nevertheless, a logical conclusion to make is still, as far as for T_{eff} on NIR concerning, there are no big disadvantages in down-scaling the input size. Training will go faster and local drive storage will take up less space, without compromising on precision.

With regards to $\log g$, there are larger differences in the error size when comparing the models. The least down-scaled model (x5), yields an MSE of 0.0015, while the most down-scaled (x30) has an error of 0.0080. It is thus a factor 5 smaller for the more down-scaled model. When comparing it with the main model's performance of 0.000005, the differences are substantial. The main model outperforms x5 by almost 300 times and x30 by around 1,600 times. The differences for the x10 and x20 models in relation to the main model are 520 and around 7,500 times respectively. A comparison over the median values follows the same line. This can be interpreted as down-sizing the input data, with the goal of estimating $\log g$, will affect the results substantially, already after doing so by a smaller fraction. The features, related to estimate $\log g$ in the spectra, are thus very sensitive to down-scaling and will easily be lost. It would therefore be interesting to train a model on a full scaled input size model and learn the result from that. An even more potent GPU would be needed though.

In terms of metallicity, the MSE for the x5 model is 0.00277 while it's 0.0501 for the x30 model. The differences in performance are some 18 times between the x5 and the x30 model. This is confirmed when comparing the medians as well. A non-trivial amount of information is thus lost when down-sizing, although not as much as for $\log g$. If top performance for M/H is desired however, the input data should be kept in original size.

Table 6.4 displays a summary over the models' performance while Table 6.5 displays the training time.

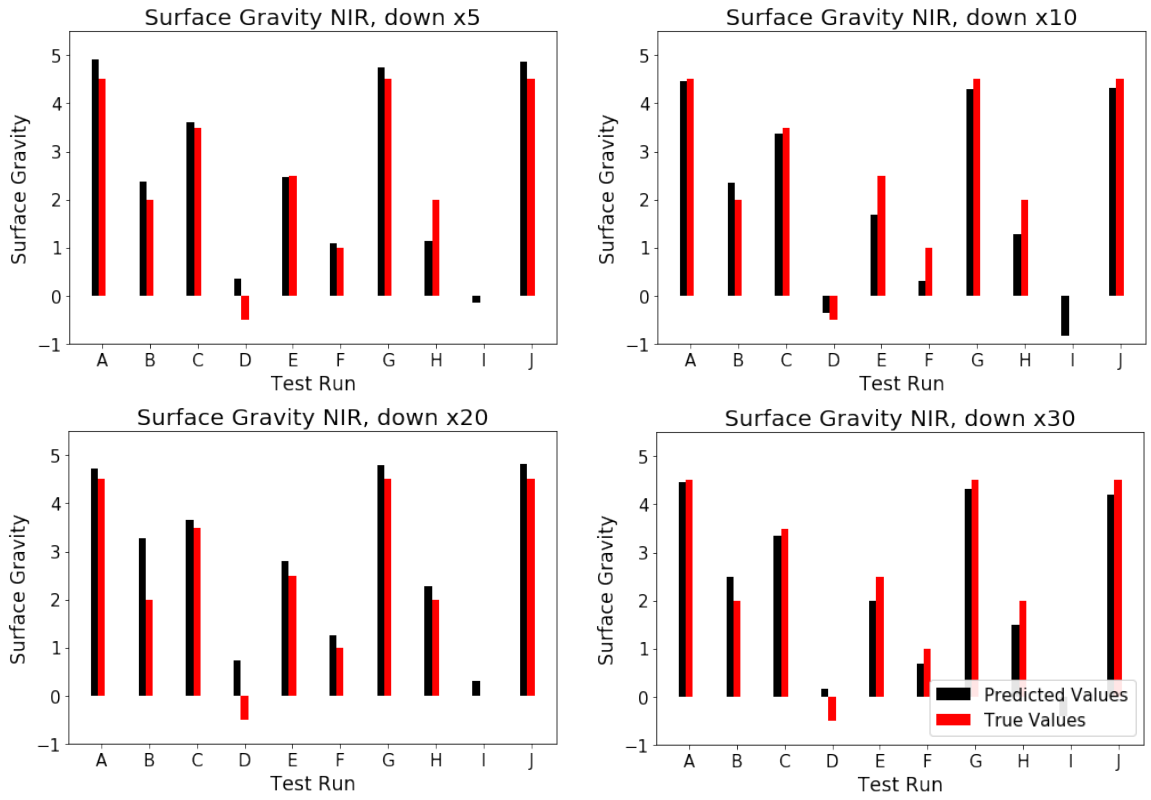

Figure 6.5: Surface gravity estimated by the 5 (top left), 10 (top right), 20 (bottom left) and 30 (bottom right) down-sized input data for the NIR models.

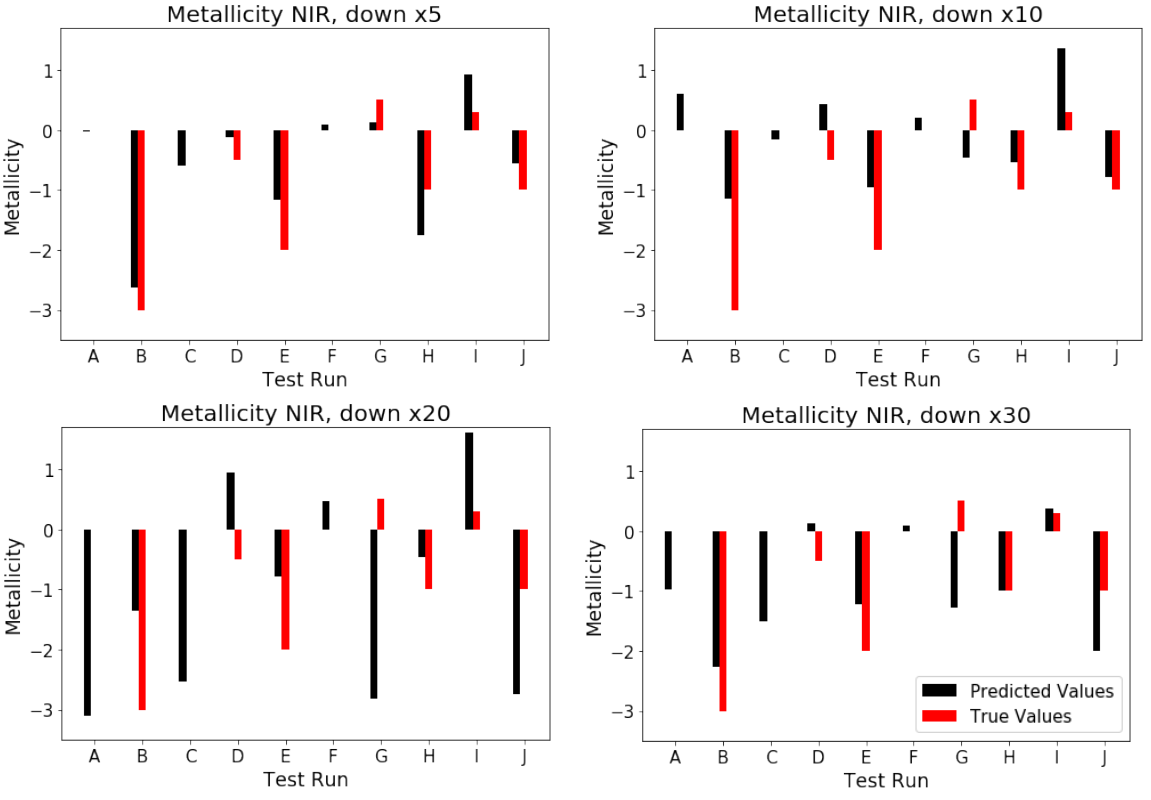

Figure 6.6: Metallicity estimated by the 5 (top left), 10 (top right), 20 (bottom left) and 30 (bottom right) down-sized input data for the NIR models.

6.3.2 VIS down-sized 5, 10, 20 and 30 times

Figures 6.7, 6.8 and 6.9 displays the four down-sized inputs, 5, 10, 20 and 30 times, on VIS for effective temperature, surface gravity and metallicity respectively. The MSE per model, see Table 6.3, over all three physical stellar parameters on the test set are 0.00066, 0.00028, 0.00076 and 0.00032, from least down-sized to most, respectively.

Equally as for NIR, T_{eff} is estimated with a very high precision by the VIS models, yielding the low MSE of 0.000058 by the most downscaled model. By investigating Figure 6.7, this low error rate can be visually observed as the discrepancy between the predicted and true values are overall very small for all four models. Although not as low as for the main model, we can still conclude, equally as for the NIR models, that the features the CNN learn in order to estimate the corresponding temperature for a spectrum is very well preserved,

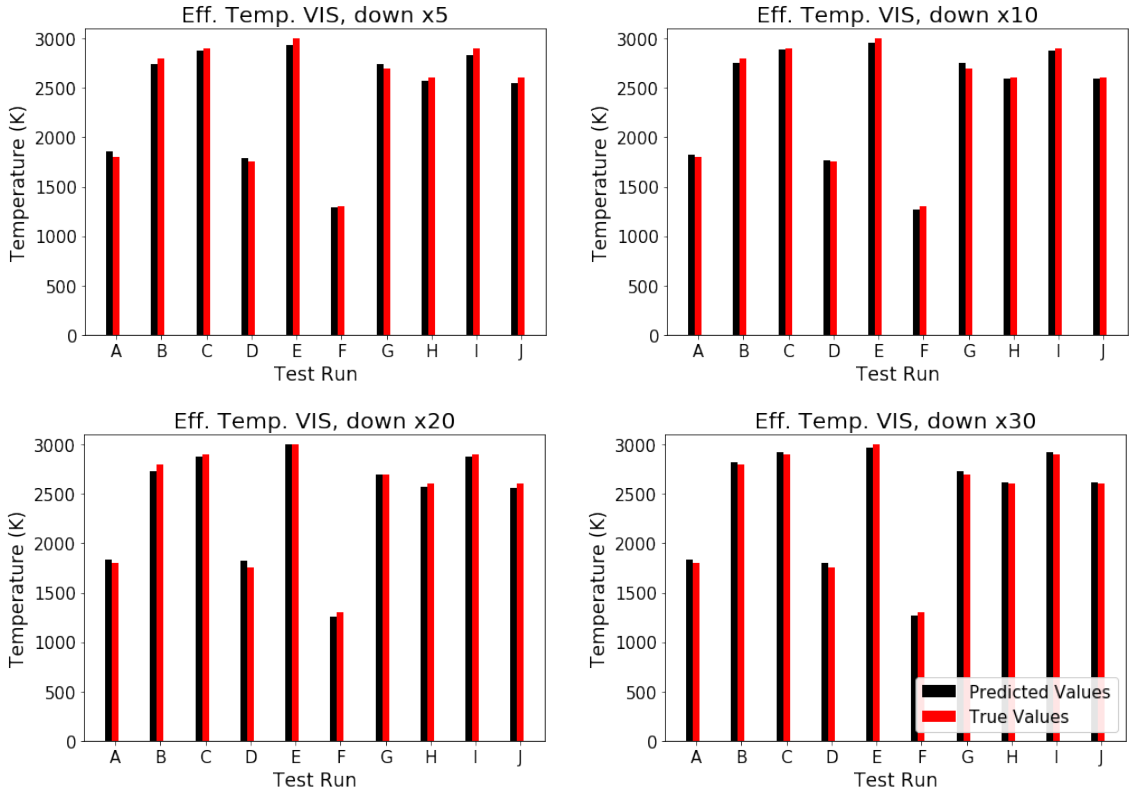

Figure 6.7: Effective temperature estimated by the 5 (top left), 10 (top right), 20 (bottom left) and 30 (bottom right) down-sized input data for VIS.

even though substantial down-sizing of the input data has been applied. The low median values, displayed in Table 6.3, across all models confirms this.

In terms of $\log g$, it seems that the x10 down-sized model continuously overestimates the true values, while the rest appears to underestimate them, see Figure 6.8. The model with the lowest error is, surprisingly, the most downscaled x30. With an MSE of 0.0013 it actually outperforms the main model's error of 0.019 ten times. This can probably partly be explained by the small test set which fails in estimating the models' performance accurately. Nevertheless, comparing the squared medians, 0.0031 (for x30) and 0.000002 (for main), yields the opposite insights. Median values are less biased for outliers and should thus be a better predictor of the models' performance. Taking this into consideration, the main model outperform all others by a large margin (1500 times in relation to x30).

Exploring the results on M/H however, see Figure 6.9 and Table 6.3, displays a uniform performance advantage for the more down-sized models. The best performing model's squared median, x5, of 0.000001 and MSE of 0.0330 are both better than the main model's equivalents (0.0046 and 0.091 respectively). The best explanation to this is thus the small test set who fails in estimating the models' performance accurately and that the error has been calculated on two standard deviations of the error. Both of which can benefit the more down-sized model.

The training time for the different NIR and VIS models are displayed in Table 6.4. The VIS models takes in general around twice as long time to train.

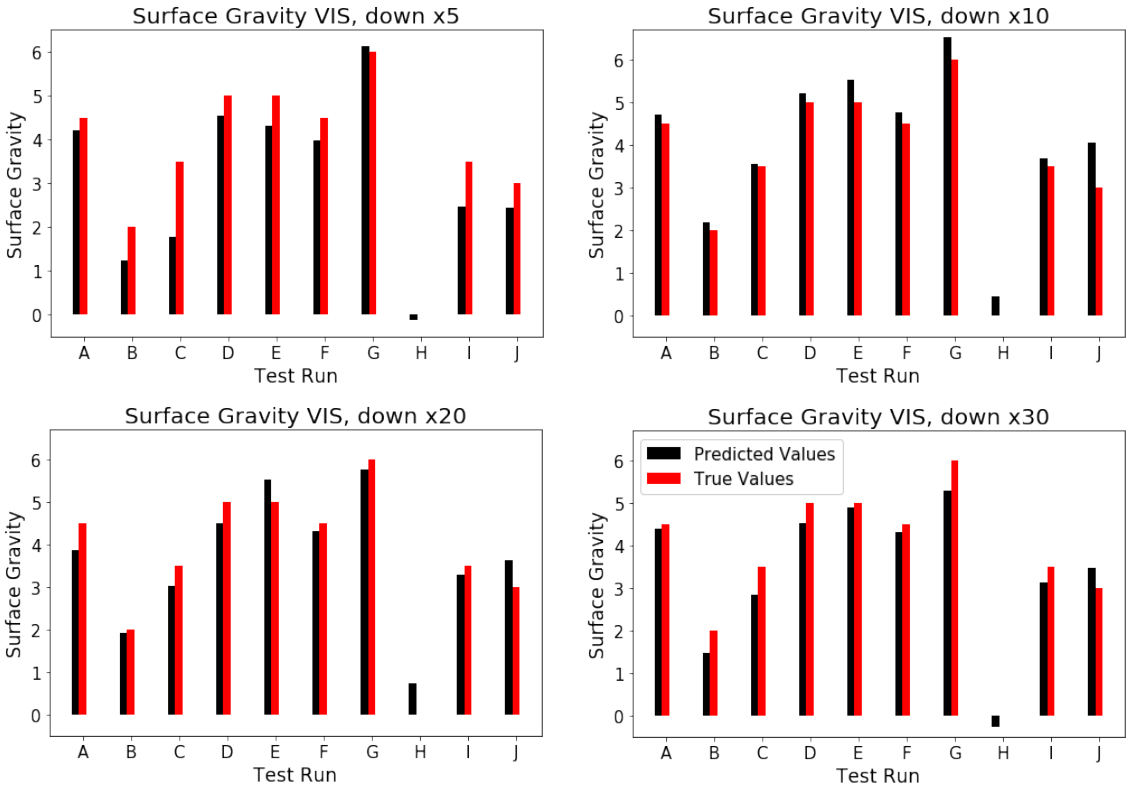

Figure 6.8: Surface gravity estimated by the 5 (top left), 10 (top right), 20 (bottom left) and 30 (bottom right) down-sized input data for VIS. Note that some of the true values are equal to zero.

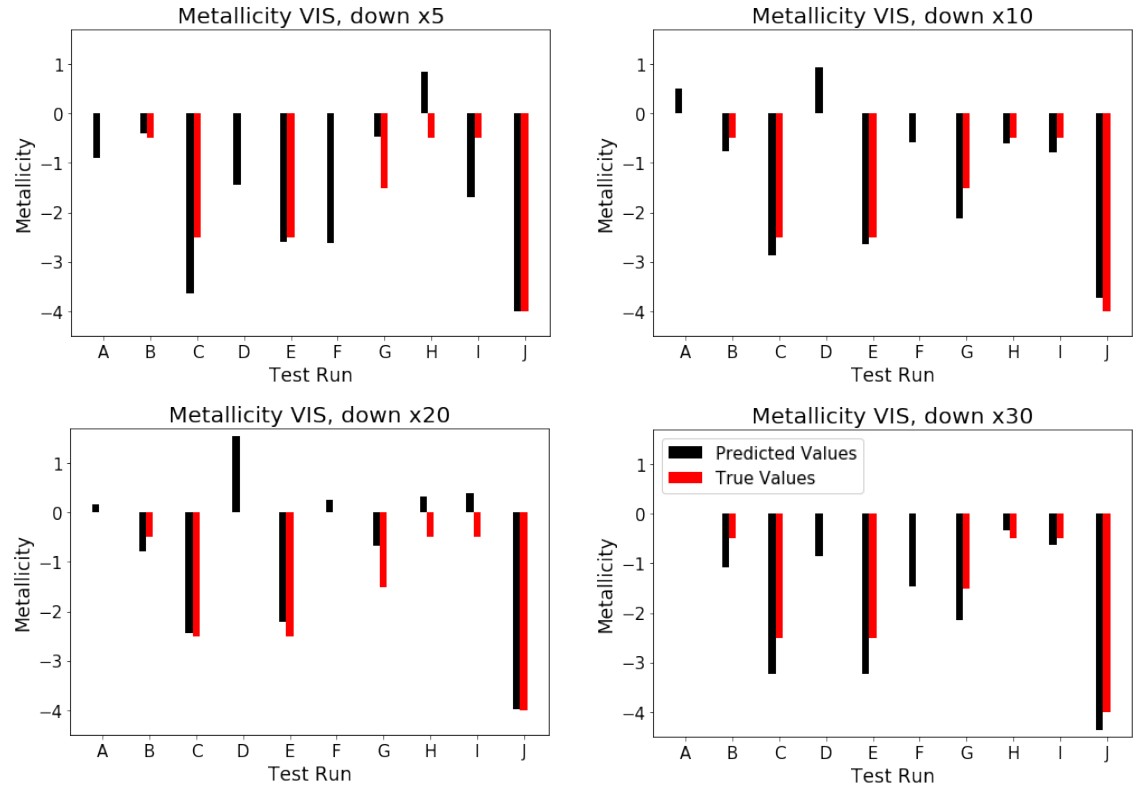

Figure 6.9: Metallicity estimated by the 5 (top left), 10 (top right), 20 (bottom left) and 30 (bottom right) down-sized input data for VIS.

Model summary	NIR (MSE)	VIS (MSE)	Median squared NIR	Median squared VIS
x5 Effective Temperature	0.000154	0.000237	0.000129	0.000289
x5 Surface Gravity	0.001514	0.184926	0.002462	0.063007
x5 Metallicity	0.002774	0.032663	0.012741	0.000001
x5 Total	0.000389	0.000661	----------	----------
x10 Effective Temperature	0.000123	0.000054	0.000099	0.000064
x10 Surface Gravity	0.002595	0.001861	0.002822	0.004042
x10 Metallicity	0.051235	0.000140	0.213698	0.000043
x10 Total	0.000813	0.000275	----------	----------
x20 Effective Temperature	0.000002	0.000066	0.0000004	0.000039
x20 Surface Gravity	0.037693	0.013175	0.007739	0.000149
x20 Metallicity	0.001388	0.154711	0.101900	0.001923
x20 Total	0.003196	0.000761	----------	----------
x30 Effective Temperature	0.000056	0.000058	0.000073	0.000064
x30 Surface Gravity	0.007950	0.001268	0.009446	0.003123
x30 Metallicity	0.050085	0.001189	0.031612	0.000002
x30 Total	0.001020	0.000319	----------	----------
Main model Eff. Temp	0.000079	0.000003	0.000080	0.000008
Main model Surface Gravity	0.000005	0.018612	0.000196	0.000002
Main model Metallicity	0.003318	0.091275	0.000359	0.004638
Main model Total	0.000103	0.000136	----------	----------

Table 6.3: MSE summary over the down-sized and the main models' performance on the test set.

Model	Training time
x5 NIR	3h 20min
x5 VIS	8h 30min
x10 NIR	3h 40 min
x10 VIS	9h
x20 NIR	3h
x20 VIS	4h 20min
x30 NIR	2h 20min
x30 VIS	4h
main NIR	17h 30min
main VIS	36h 30min

Table 6.4: Summary over training time for down-sized 5, 10 20, 30 and the main models. The VIS models takes roughly twice as long time to train than the NIR equivalents.

6.4 Estimated parameters for the Carmenes datasets

Estimated physical parameters for the Carmenes data are displayed in Table 6.5 and 6.6 below for the main NIR and VIS models respectively. Note that it is the same nine matrix names in both tables, with the only difference being the wavelengths.

There's quite some disparity between the estimated values for the NIR and VIS models. Ideally, they would differ little, and this seems to be the case for $\log(g)$. As for T_{eff} and M/H however, the estimations are diverging in a larger extent. There might be several reasons for this. The most crucial one is whether the models trained on the theoretical BT-Settl dataset are able to describe the observed Carmenes spectra in a satisfying way. It is expected to be so, but there are no such guarantees.

The individual contribution from each order in the spectra creation might also play a part. The assumption of equal contribution might be flawed, leading to important information losses, and thus, lower model performance on the Carmenes dataset.

A third reason might be that the VIS model seems to estimate T_{eff} with a higher precision than the NIR model, which in turn better estimates $\log(g)$ and M/H (see chapter 6.2). Taking this into account, it's plausible that the estimations differ extensively merely because of the models' capability in estimating the different parameters.

One, or a combination, of the abovementioned reasons are most likely the cause for the differences when estimating T_{eff} and M/H with the NIR and VIS models.

Matrix name	Eff. Temperature (K)	Surface Gravity	Metallicity
car-20170520T20h38m14s-sci-gtoc-nir	2260	1,9	-1,1
car-20170609T20h33m03s-sci-gtoc-nir	2411	1,7	-1,8
car-20170822T01h54m18s-sci-gtoc-nir	2356	1,8	-1,3
car-20170825T00h06m21s-sci-gtoc-nir	2418	1,7	-1,9
car-20170911T01h42m21s-sci-gtoc-nir	2355	1,8	-1,3
car-20170912T02h41m57s-sci-gtoc-nir	2355	1,8	-1,3
car-20170913T21h52m13s-sci-gtoc-nir	2412	1,6	-1,3
car-20170914T03h24m58s-sci-gtoc-nir	2367	1,7	-1,5
car-20170924T20h42m07s-sci-gtoc-nir	2401	1,7	-1,2

Table 6.5: Estimated parameters for the Carmenes data set on NIR.

Matrix name	Eff. Temperature (K)	Surface Gravity	Metallicity
car-20170520T20h38m14s-sci-gtoc-vis	2791	2,0	0,1
car-20170609T20h33m03s-sci-gtoc-vis	2782	1,8	0,3
car-20170822T01h54m18s-sci-gtoc-vis	2845	1,9	-0,3
car-20170825T00h06m21s-sci-gtoc-vis	2796	2,2	0,4
car-20170911T01h42m21s-sci-gtoc-vis	2854	1,9	-0,7
car-20170912T02h41m57s-sci-gtoc-vis	2813	2,2	0,4
car-20170913T21h52m13s-sci-gtoc-vis	2742	2,1	2,6
car-20170914T03h24m58s-sci-gtoc-vis	2783	1,8	1,1
car-20170924T20h42m07s-sci-gtoc-vis	2813	2,1	0,4

Table 6.6: Estimated parameters for the Carmenes data set on VIS.

Chapter 7 Organisational Aspects

7.1 Gantt Chart

A Gantt chart is shown in Figure 7.1 below, displaying all the mayor events for the project. A total of 6 months have to be devoted if the tasks are to be solved consecutively. Several steps can be done in parallel however, such as pre-processing the Carmenes and BT-Settl datasets, developing a data generator and extracting the physical stellar parameters as well as training both the NIR and VIS models on two machines. The report and the presentation can also partly be worked on in parallel during a more extended part of the development process.

Nevertheless, around 1200 manhours are needed, from understanding the problem at hand to render reliable estimations for the physical stellar parameters and finally deliver the report. Another 24 hours can be added for preparing the presentation for stakeholders.

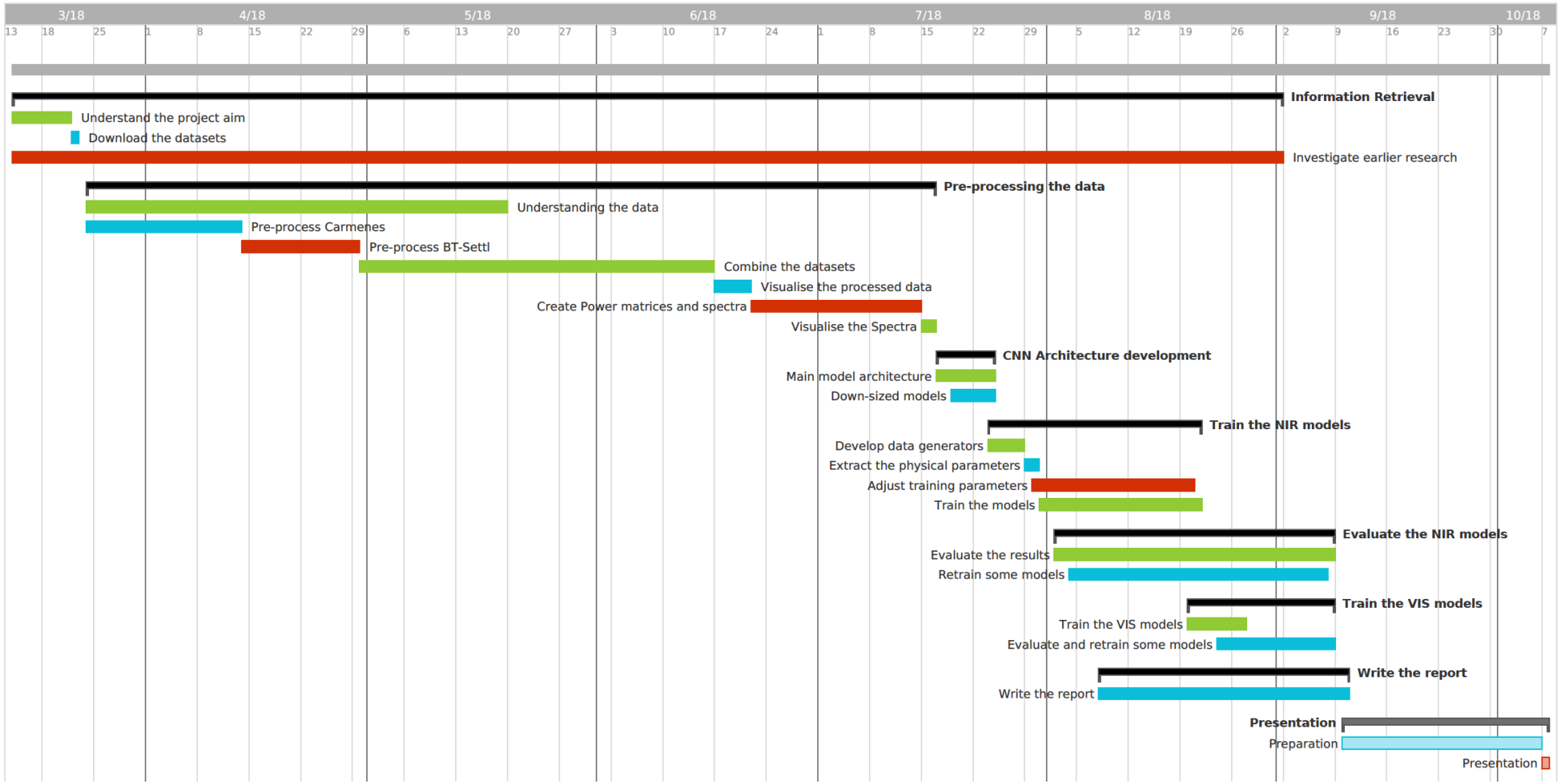

Figure 7.1: Gantt Chart over the project.

7.2 Budget

The project's total budget is summarised in Table 7.1 below. The total product expenses are estimated to 4,682€ while the labour cost is approximated to 15,300€. This is calculated through considering an engineer with one year of experience working on the project full time with an average Spanish salary for an industrial or software engineer with similar experience. With taxes and profit margin included, the total cost of the project is estimated to *27,362,69€*.

Project Title; CNN Project TOTAL COST: **€27 362,69**

PRODUCT EXPENSES				
ITEM NAME	ITEM DESCRIPTION	UNITS	€ / UNIT	TOTAL
Computer	Sufficiently powerful	1	€ 2 000,00	€ 2 000,00
Software	MO license per month	6	€ 7,00	€ 42,00
Softwares	Open source	1	$ -	$ -
Office Space	Necessary facilities	6	€ 405,00	€ 2 430,00
Miscellaneous	Paper, pencils etc.	1	€ 20,00	€ 20,00
Electricity	kwh	800	€ 0,24	€ 189,60
	PRODUCT EXPENSE TOTAL			€ 4 681,60

LABOR EXPENSES				
TASK NAME	TASK DESCRIPTION	HOURS	€ / HOUR	TOTAL
Engineer	1 year of experience	1224	€ 12,50	€ 15 300,00
	LABOUR EXPENSE TOTAL			€ 15 300,00

Operational margins and Taxes				
ITEM NAME		%	Sum	TOTAL
Profit Margin		5%	€ 765,0	€ 20 746,60
Cost Margin		9%	€ 1 867,2	€ 22 613,79
VAT		21%	€ 4 748,9	€ 27 362,69
				€ 27 362,69

Table 7.1: The project's total cost is 27 362,69€.

Chapter 8 Conclusions and Future Work

8.1 Conclusions

Due to the vast amount of planets to search in the universe, several approaches are being tested to focus the hunt on the most promising ones. The objective of this Master's Thesis has been to evaluate the capabilities of Deep Learning with a regression approach to estimate physical parameters from observed stars' spectra. It is the first time this approach is tested for physical parameter estimation. These parameters are important in learning more about the star and its orbiting planets.

By pre-processing the data in several steps, spectra were created and subsequently processed by CNNs. The models were trained on NIR and VIS separately.

The results indicate that it is possible to estimate physical stellar parameters with the use of CNNs on spectra with regression, yielding higher precision than earlier, among these, Machine Learning approaches.

Especially effective temperature was estimated with the low median squared errors of 0.000079 and 0.000003 for NIR and VIS respectively. This is equivalent to errors in the range $12 - 27K$ and $2 - 5K$, respectively, for M-type stars spanning the effective temperature range of $1300 - 3000K$.

Regarding surface gravity, MSEs are 0.000005 and 0.018612 for NIR and VIS, which are equivalent to errors of 0.0002dex and 0.14dex respectively. Metallicity on the other hand is estimated with MSEs of 0.003318 and 0.091275, or errors of 0.058dex and 0.30dex for NIR and VIS. This can be compared with the metallicity errors of 0.25dex from earlier research by L. M. Sarro et al. [2017].

The referred errors in this conclusions section are the ones obtained through the main models as they are considered to be more reliable. Although they seemingly perform worse on some individual occasions (see Table 6.3) it can mainly be explained by the small test set not being able to fully capture the models' performance accurately.

The NIR model seems to estimate metallicity with highest precision while the VIS model does it for effective temperature and surface gravity, when the less biased median squared errors are taken into account.

Although the main models are considered to be more reliable, the down-sized versions still estimate with high precision across most parameters. This has also practical implications as the code and the models can be fitted onto different GPU cards with various capacities.

The trained models were subsequently used to estimate the physical parameters on the observed Carmenes dataset where the estimations for these are displayed in Tables 6.5 and 6.6. Training has been performed on theoretical spectra, and although expected, there are no guarantees that these spectra are sufficiently "similar" to the real Carmenes spectra to output reliable estimations. The differences between the NIR and VIS estimations indicate that further research in various directions is needed.

8.2 Future Work

Pre-processing large amounts of data and then analyse it with deep neural networks isn't a trivial task if one wants to obtain good results. There are many decisions to be taken along the way and one can argue for and against each one of them. However, there are some matters that might be extra interesting for future research or investigations.

The first thing would be to process all the data at hand, creating more power matrices with resulting spectra. During this Master's Thesis, there was

only time for processing 2044 (1024 NIR and 1020 VIS) files, out of total 8096 (4046 NIR and 4046 VIS). As for Deep Learning concerning, larger datasets are almost always considered as better since the model would by this have more data to be trained on, yielding both better parameter estimations and subsequent model evaluations.

Little research has been done on the Adam optimiser since it has fairly recently been introduced. A common approach in Machine Learning though is letting the learning rate decay over the training epochs. It would thus be interesting to try this approach on the problem at hand for Adam.

There is also the alternative of using transfer learning, by removing the top layers of an already trained model, add some new ones on top, and train the new layers on the data at hand. This approach has frequently resulted in state of the art results. However, as these models often are trained on images that are very different from the spectrograms in this project (such as on *ImageNet*, in the case of the high performance *Xception* model [44]), the decision was made not to proceed in this way. Nevertheless, this approach has proven very successful for medical image analysis among others [52], where the applied CNN method is very similar to the one used in this Master's Thesis.

Initially, the assumption was made that every order contributes equally in representing the power matrices and the subsequent spectra, and they were thus stacked equally during the power matrix creation. However, this assumption should be questioned as there are no guarantees that this is the case. Further research should investigate this by, for example, creating one model for each order and assigning a confidence interval for each one of them when testing on the BT-Settl dataset.

As mentioned in the conclusion section, further investigations have to be conducted to learn whether a model trained on the theoretical BT-Settl data is able to describe the observed Carmenes spectra with confidence. Since the differences are quite substantial between the NIR and the VIS models' estimations, especially for T_{eff} and M/H, this would thus be a priority in further research.

Since T_{eff} seemingly is stronger correlated with the spectra, these estimations can be collected and used for further training of the model. The resulting model will thus have more training data, and might as a result, yield better estimations for $\log g$ and M/H.

Bibliography

[1] NASA, Jet Propulsion Laboratory, "Overlooked Treasure: The First Evidence of Exoplanets," California Institute of Technology, 01 11 2017. [Online]. Available: https://www.jpl.nasa.gov/news/news.php?feature=6991. [Accessed 09 08 2018].

[2] A. W. &. D. A. Frail, "A planetary system around the millisecond pulsar PSR1257 + 12," *nature - International journal of science,* pp. 145-147, 09 01 1992.

[3] NASA, "Mission overview Kepler and K2," NASA, [Online]. Available: https://www.nasa.gov/mission_pages/kepler/overview/index.html. [Accessed 09 08 2018].

[4] NASA, "Exoplanet Discoveries, Latest Data from NASA's Exoplanet Archive," NASA, 08 08 2018. [Online]. Available: https://exoplanets.nasa.gov. [Accessed 09 08 2018].

[5] NASA, "Warm welcome: finding habitable planets," NASA, [Online]. Available: https://exoplanets.nasa.gov/the-search-for-life/habitable-zones/. [Accessed 09 08 2018].

[6] NASA, "How Many Stars in the Universe?," NASA, 14 06 2012. [Online]. Available: https://helios.gsfc.nasa.gov/qa_star.html#howmany. [Accessed 09 08 2018].

[7] J. T. Wright, "Radial Velocities as an Exoplanet Discovery Method," Cornell University Library, 2017.

[8] A. a. t. C. C. Quirrenbach, "CARMENES: Calar Alto high-Resolution search for M dwarfs with Exo-earths with a Near-infrared Echelle Spectrograph," in *CARMENES*, 2009.

[9] CARMENES Consortium, "CARMENES Consortium," [Online]. Available: https://carmenes.caha.es/ext/consortium/index.html#top. [Accessed 16 08 2018].

[10] Carmenes Consortium, "Carmenes, Science," Carmenes Consortium, [Online]. Available: https://carmenes.caha.es/ext/science/index.html. [Accessed 16 08 2018].

[11] L. M. Sarro, J. Ordieres-Meré, A. Bello-García, A. González-Marcos and E. Solano, "Estimates of the atmospheric parameters of M-type stars: a machine learning perspective.," Madrid, 2017.

[12] IESE Business School, "10 Ways Artificial Intelligence Is Transforming Management - Conference gathers top business leaders and academics to look at data-fueled future," Barcelona, 2018.

[13] A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," University of Toronto, Toronto, 2012.

[14] K. Zhou and B. Kainz, "Efficient Image Evidence Analysis of CNN Classification Results," Imperial College London, London, 2018.

[15] D. Hsu, "Using Convolutional Neural Networks to Classify Dog Breeds," Stanford University, Stanford, California, 2016.

[16] F. Wang, M. Jiang, C. Qian, S. Yang and C. Li, "Residual Attention Network for Image Classification," Hong Kong, Beijing, 2017.

[17] Szegedy, Christian; Liu, Wei; Jia, Yangqing; Sermanet, Pierre; Reed, Scott; Anguelov, Dragomir; Erhan, Dumitru; Vanhoucke, Vincent; Rabinovich, Andrew; Google Inc.; University of North Carolina; University of Michigan, "Going deeper with convolutions," Google Inc., 2014.

[18] Z.-H. Zhou and J. Feng, "Deep Forest," Nanjing University, Nanjing, 2018.

[19] Z.-H. Zhou and J. Feng, "Deep Forest: Towards an Alternative to Deep Neural Networks," Nanjing University, Nanjing, 2017.

[20] A. Gonzáles-Marcos, L. M. Sarro, J. Ordieres-Meré and A. Bello-García, "Evaluation of data compression techniques for the inference of stellar atmospheric parameters from high-resolution spectra," *Oxford University Press,* pp. 4556-4571, 23 11 2016.

[21] NVIDIA Corporation, "End to End Learning for Self-Driving Cars," NVIDIA Corporation, Holmdel, NJ, USA, 2016.

[22] S. Miao, Z. Jane Wang and R. Liao, "A CNN Regression Approach for Real-Time 2D/3D Registration," IEEE, New Jersey, 2016.

[23] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, IEEE, 1998.

[24] C. Alexis, H. Schwenk, Y. LeCun and L. Barrault, "Very Deep Convolutional Networks for Text Classification," 2017.

[25] A. van den Oord, S. Dieleman, and B. Schrauwen, "Deep content-based music recommendation," 2013.

[26] J. Russell, "techcrunch.com," 15 03 2016. [Online]. Available: https://techcrunch.com/2016/03/15/google-ai-beats-go-world-champion-again-to-complete-historic-4-1-series-victory/?_ga=2.116547205.209713902.1534604929-10151628.1526584328. [Accessed 18 08 2018].

[27] D. H. Hubel and T. N. Wiesel, “Receptive fields and functional architecture of monkey striate cortex,” Boston, 1968.

[28] A. Karpathy, “Neural Networks Part 1: Setting up the Architecture,” Stanford, [Online]. Available: http://cs231n.github.io/neural-networks-1/. [Accessed 18 08 2018].

[29] Andrej Karpathy for Stanford, “Convolutional Neural Networks (CNNs / ConvNets),” Stanford, [Online]. Available: http://cs231n.github.io/convolutional-networks/. [Accessed 19 08 2018].

[30] Open-source, “Activation Functions,” 2017. [Online]. Available: https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html. [Accessed 19 08 2018].

[31] Andrej Karpathy for Standford University, “CS231n: Convolutional Neural Networks for Visual Recognition,” Stanford, 2018. [Online]. Available: http://cs231n.github.io/neural-networks-3/. [Accessed 19 08 2018].

[32] Coursera, “Structuring Machine Learning Projects,” 2018.

[33] I. Guyon, “A scaling law for the validation-set training-set size ratio,” Berkeley, Califronia.

[34] F. Chollet, “Keras Documentation,” MIT, 27 03 2015. [Online]. Available: https://keras.io. [Accessed 20 08 2018].

[35] Google Brain team, “About TensorFlow,” Google Inc., [Online]. Available: https://www.tensorflow.org. [Accessed 20 08 2018].

[36] A. Reiners, J. Bean, K. Huber, S. Dreizler, A. Seifahrt and S. Czesla, “Detecting planets around very low mass stars with the radial velocity method.,” ApJ 710, 2010.

[37] CARMENES Consortium, “CARMENES Instrument Overview,” CARMENES Consortium, 2014.

[38] CARMENES Consortium, “CARMENES Instrument,” [Online]. Available: http://carmenes.caha.es/ext/instrument/index.html. [Accessed 22 08 2018].

[39] Spanish Virtual Observatory, “Theoretical spectra web server,” Spanish Virtual Observatory, [Online]. Available: http://svo2.cab.inta-csic.es/theory//newov2/. [Accessed 21 08 2018].

[40] Lyon University, “FORMAT OF THE SPECTRA OUTPUT FILES for the PHOENIX Simulator,” [Online]. Available: https://phoenix.ens-lyon.fr/Grids/FORMAT. [Accessed 21 08 2018].

[41] J. Albert, “Stack overflow - Find nearest value in numpy array,” 25 01 2017. [Online]. Available: https://stackoverflow.com/questions/2566412/find-nearest-value-in-numpy-array. [Accessed 24 08 2018].

[42] C. Torrence and G. P. Compo, “A Practical Guide to Wavelet Analysis,” *American Meterological Society (AMS),* vol. 79, no. 1, pp. 61-78, 01 1998.

[43] S. Krieger, N. Freij, A. Brazhe, C. Torrence and G. P. Compo, “Tutorial: Time-series spectral analysis using wavelets,” 2017. [Online]. Available: https://pycwt.readthedocs.io/en/latest/tutorial.html#f1. [Accessed 25 08 2018].

[44] F. Chollet, “Xception: Deep Learning with Depthwise Separable Convolutions,” Google, Inc., 2017.

[45] Y. Zhang and B. C. Wallace, “A Sensitivity Analysis of (and Practitioners’ Guide to) Convolutional Neural Networks for Sentence Classification,” Austin, 2016.

[46] François Chollet with Community, “Keras Documentation Utils Sequence,” 2015. [Online]. Available: https://keras.io/utils/#sequence. [Accessed 27 08 2018].

[47] François Chollet with Community, “Keras Documentation - Sequential model methods,” 2015. [Online]. Available: https://keras.io/models/sequential/. [Accessed 28 08 2018].

[48] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu and a. t. s.-i. c. , “scikit-image: image processing in Python,” PeerJ 2:e453 , New York, 2014.

[49] A. Ng, “Normalizing inputs,” Coursera, 2018. [Online]. Available: https://www.coursera.org/lecture/deep-neural-network/normalizing-inputs-lXv6U. [Accessed 27 08 2018].

[50] P. Fabian, V. Gaël, G. Alexandre, M. Vincent, T. Bertrand, G. Olivier, B. Mathieu, P. Peter, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau and M. Brucher, “Scikit-learn: Machine Learning in Python,” 2011.

[51] D. P. Kingma and J. Lei Ba, “ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION,” in *ICLR*, San Diego, 2015.

[52] N. Tajbakhsh, J. Y. Shin, S. R. Gurudu, R. Todd Hurst, C. B. Kendall, M. B. Gotway, and J. Liang, “Convolutional Neural Networks for Medical Image Analysis: Full Training or Fine Tuning?,” arXiv:1706.00712v1, 2017.

[53] A. Saxena, “Convolutional Neural Networks (CNNs): An Illustrated Explanation,” XRDS Crossroads - The ACM Magazine for Students, 29 06 2016. [Online]. Available: https://xrds.acm.org/blog/2016/06/convolutional-neural-networks-cnns-illustrated-explanation/. [Accessed 18 08 2018].

[54] Google Brain team, “TensorFlow - Large-Scale Machine Learning on Heterogeneous Distributed Systems,” Google Inc., 2015.

Appendix

Appendix A

Table A.1 displays the first ten rows in a raw Carmenes VIS input file. It constitutes in total of around 250,000 measurements dividend into 61 orders.

	Wavelength (Angstrom)	Flux 1	Flux 2
0	5136.508188	NaN	NaN
1	5136.535813	NaN	NaN
2	5136.563434	NaN	NaN
3	5136.591053	NaN	NaN
4	5136.618670	NaN	NaN
5	5136.646283	NaN	NaN
6	5136.673894	NaN	NaN
7	5136.701502	NaN	NaN
8	5136.729108	NaN	NaN
9	5136.756711	NaN	NaN

Table A.1: Top ten first lines of the first order in a raw Carmenes VIS input file. There are no values for neither of the fluxes.

Appendix B – Major python code snips and functions

This part of the Appendix displays some major functions and code parts which were developed for the sake of this project. If the reader desires the full code or the input files, he or she is most welcome to contact the author or the tutor.

```
"""Visual wavelength (VIS) data"""
for i in range(len(all_loaded_data_VIS[0])):
    all_min_VIS_values, all_max_VIS_values = [], []
    for j in range(len(all_loaded_data_VIS)):
        all_min_VIS_values.append(all_loaded_data_VIS[j][i].index[0])
        all_max_VIS_values.append(all_loaded_data_VIS[j][i].index[-1])

    # choose the min and max limits for each order
    VIS_min, VIS_max = max(all_min_VIS_values), min(all_max_VIS_values)

    # Align the values for all the VIS data (according to VIS_min and VIS_max)
    # so they match each other.
    for h in range(len(all_loaded_data_VIS)):
        all_loaded_data_VIS[h][i] = all_loaded_data_VIS[h][i].loc[VIS_min:VIS_max]
```

Code Snip B.1: Code snip for aligning the min and max values so all the equivalent orders are within the same interval.

```
from multiprocessing import Process

# Time the process
start1 = time.time()

if __name__ == '__main__':

    try:
        mp.set_start_method('fork')
    except RuntimeError:
        pass

    number_of_cores = 16
    files_per_core = 68

    # Setup a list of processes that we want to run
    file_nbr = 0
    for _ in range(files_per_core):
        processes = []
        for _ in range(number_of_cores):
            prr = Process(target=create_btsettl_data_FINAL,
                          args=(data_VIS_avg_LOCAL,
                                file_names_settl[file_nbr:(file_nbr+1)], "VIS"))
            processes.append(prr)
            file_nbr += 1

        # Run processes
        for p in processes:
            p.start()

        # Exit the completed processes
        for p in processes:
            p.join()

# Check how long time it took.
end1 = time.time()
tt1 = end1 - start1
print("Total run time: {0:0.0f} min and {1:3.0f} s ".format((tt1-tt1%60)/60, tt1%60))
```

Code Snip B.2: Implementing multiprocessing with 16 cores.

```
def get_POWER_matrix(data, rescale=True):
    """ Returns the power matrix, the scale indices and the Fourier frequencies.
        Set 'rescale' to False if rescaling is not desired. """
    t0, dt, N = 0, 0.5, data.size
    t = np.arange(0, N) * dt*2 + t0
    p = np.polyfit(t - t0, data, 1)
    dat_notrend = data - np.polyval(p, t - t0)
    std = dat_notrend.std()              # Standard deviation
    var = std ** 2                       # Variance
    dat_norm = dat_notrend / std         # Normalised dataset
    mother = wavelet.Morlet(6)           # Morlet transformation function

    # The following routines perform the wavelet transformation using the
    # parameters defined above.
    wave, scales, freqs, coi, fft, fftfreqs = wavelet.cwt(dat_norm, dt, J=-1)
    power = (np.abs(wave)) ** 2
    if rescale == True:
        power *= 255.0/power.max()    # rescale 0-255
        power = power.astype('h')     # dtype = short
        scales *= 255.0/scales.max()
        scales = scales.astype('h')
        freqs *= 255.0/freqs.max()
        freqs = freqs.astype('h')
    return power, scales, freqs


def get_FULL_POWER_matrix(data):
    """ Returns the power matrix, the scale indices and the
        Fourier frequencies of "data". "data" is the pre-processed
        BT-Settl data. All orders included.
    """
    power_matrix, full_scales, full_freqs = np.array([1]), np.array([1]), np.array([1])
    for i in range(len(data)):
        power, scales, freqs = get_POWER_matrix(data[i].loc[:, "Flux"])
        if len(power_matrix) > 10:                    # if no power matrix is created
            power_matrix = np.append(power_matrix, power, axis=0)
            full_scales = np.append(full_scales, scales, axis=0)
            full_freqs = np.append(full_freqs, freqs, axis=0)
        else:
            power_matrix = np.array(power)
            full_scales = np.array(scales)
            full_freqs = np.array(freqs)
    return power_matrix, full_scales, full_freqs


def plot_data(data, rescale=True):
    """ Plot the data for a single order. data is a one column array. """
    t0, dt, N = 0, 0.5, data.size
    t = np.arange(0, N) * dt*2 + t0
    p = np.polyfit(t - t0, data, 1)
```

```
dat_notrend = data - np.polyval(p, t - t0)
std = dat_notrend.std()             # Standard deviation
var = std ** 2                      # Variance
dat_norm = dat_notrend / std    # Normalised dataset
mother = wavelet.Morlet(6)      # Morlet transformation function
s0 = 2 * dt                 # Starting scale, in this case 2 * 0.25 years = 6 months
dj = 1 / 48                 # Twelve sub-octaves per octaves
J = -1
alpha, _, _ = wavelet.ar1(data)  # Lag-1 autocorrelation for red noise

# The following routines perform the wavelet transformation using the
# parameters defined above.
wave, scales, freqs, coi, fft, fftfreqs = wavelet.cwt(dat_norm, dt, J=-1)
power = (np.abs(wave)) ** 2
if rescale == True:
    power *= 255.0/power.max()    # rescale 0-255
    power = power.astype('h')       # dtype = short
period = 1 / freqs
signif, fft_theor = wavelet.significance(1.0, dt, scales, 0,
                                alpha, significance_level=0.95, wavelet=mother)
sig95 = np.ones([1, N]) * signif[:, None]
sig95 = power / sig95

# Finally, we plot our results.
plt.close('all')
plt.ioff()
fig = plt.figure(figsize=(12.5, 40), dpi=80)

# Second sub-plot, the normalised wavelet power spectrum
# and significance level contour lines and cone of influece hatched area.
# Note that period scale is logarithmic.
bx = plt.axes([0.1, 0.37, 0.65, 0.28])
levels = [0.0625, 0.125, 0.25, 0.5, 1, 2, 4, 8, 16]

bx.contourf(t, np.log2(period), np.log2(power), np.log2(levels),
            extend='both', cmap=plt.cm.viridis)
extent = [t.min(), t.max(), 0, max(period)]
bx.contour(t, np.log2(period), sig95, [-99, 1],
                colors='k', linewidths=1, extent=extent)
bx.set_ylabel('Period', fontsize=18)
bx.set_xlabel('Index', fontsize=18)
plt.autoscale(enable=True, axis="x", tight=True)
Yticks = 2 ** np.arange(np.ceil(np.log2(period.min())),
                                np.ceil(np.log2(period.max())))
bx.set_yticks(np.log2(Yticks))
bx.set_yticklabels(Yticks, fontsize=12)
plt.xticks(fontsize=12)
plt.show()
```

Code Snip B.3: Functions for creating power matrices get_POWER_matrix and get_FULL_POWER_matrix, as well as for plotting the corresponding Spectrogram (plot_data).

```
def Data_Generator(image_paths, batch_size=16):
    """ Generates batches of training data and ground truth.
        Inputs are the image paths and batch size.
    """
    L = len(image_paths)

    while True:
        batch_start = 0
        batch_end = batch_size
        while batch_start < L:
            matrices, parameters = [], []

            # Select the batch
            batch = image_paths[batch_start:batch_end]
            limit = min(batch_end, L)

            for path in batch:
                # Get the matrix, parameters and the name
                mat, param, name = get_Matrix_and_Parameters(path)

                # Transform the matrix from 2D to 3D as a
                # (mat.shape[0], mat.shape[1]) RBG image.
                # Rescale its values to [0,1]. Set "preserve_range=True" to not
                # rescale the matrix, and by this saving memory and load time.
                mat = skimage.transform.resize(mat, (mat.shape[0]//down_size,
                                                     mat.shape[1]//down_size, 3),
                                               mode='constant', preserve_range=True)

                # Scale and normalise the parameters and the matrix
                param = MMscale_param(param, name)
                mat = normalise(mat)

                matrices.append(mat)
                parameters.append(param)

            # Convert the lists into arrays and reshape the
            # parameter array to (batch size, number of parameters)
            MAT, PAM = np.array(matrices), np.array(parameters)
            PAM = np.reshape(PAM, (PAM.shape[0], PAM.shape[1]))

            # Output the matrix and parameters array. Delete the data
            # from RAM after it's been used to save memory
            yield MAT, PAM

            batch_start += batch_size
            batch_end += batch_size
            gc.collect()
```

Code Snip B.4: Data Generator to feed the model with data during training batch by batch. Evaluated on single core processing.

```
def MMscale_param(param, name):
    """ A function for scaling the parameters in param.
        name is the matrix name from where the parameters origin.
    """
    if "NIR" in name:
        scalers = many_MinMaxScalers_NIR_param
    elif "VIS" in name:
        scalers = many_MinMaxScalers_VIS_param
    else:
        print('Have you entered a valid name?')

    scaled_param = scalers[name].fit_transform(param.reshape(-1, 1))
    return scaled_param


def Un_scale_data(scaled_param, name):
    """ A function for unscaling the parameters
        in the variable scaled_param.
    """
    if "NIR" in name:
        scalers = many_MinMaxScalers_NIR_param
    elif "VIS" in name:
        scalers = many_MinMaxScalers_VIS_param
    else:
        print('Have you entered a valid name?')

    un_scaled_param = scalers[name].inverse_transform(scaled_param.reshape(-1,1))
    return un_scaled_param


NIR_paths = get_MATRIX_paths("NIR")

many_MinMaxScalers_NIR_param = {}
for i_s in range(len(NIR_paths)):
    name = os.path.basename(NIR_paths[i_s])[:-4]
    many_MinMaxScalers_NIR_param["{0}".format(name)]=MinMaxScaler(
        feature_range=(0,1), copy=True)
```

Code Snip B.5: Displays the creation of MinMaxScalers for the NIR data as well as the two functions used to call the scalers for scaling and reversing the same.

Two functions for extracting the parameters (T_{eff}, $\log g$ and M/H) from a power matrix's name are displayed in Code Snip B.6. *get_Matrix_and_Parameters()* returns the parameters, the matrix and corresponding matrix name.

```
def get_Matrix_and_Parameters(path):
    """ Returns the matrix stored in 'path', with the corresponding parameters
        (Temperature, Surface Gravity and Metallicity)
    """
    matrix = pd.read_csv(path)     # load the power matrix
    parameters = []
    filename = os.path.basename(path)[:-4]
    name = filename

    # Process the file name and extract the individual parameters.
    # Store them in a list.
    #####
    # Remove the alpha parameter and do some editing to the file name
    for l in str(filename):
        if 'a' in l:     # if 'a' is in the file name
            d = 'a'
            break
        else:
            d = '_'
    filename = substring_before(filename[3:], d)[0]

    # Extract the Temperature
    for l in str(filename[:6]):
        if '-' in l:
            d1 = '-'
            break
        elif '+' in l:
            d1 = '+'
            break
        else:
            pass
    p, rest = substring_before(filename, d1)
    parameters.append(float(p))

    # Extract the Surface gravity, Log(G)
    for l in str(rest[2][:5]):
        if '-' in l:
            de = '-'
            break
        elif '+' in l:
            de = '+'
            break
        else:
            pass
```

```
    p1, rest1 = substring_before(rest[2], de)
    parameters.append(float(d1 + p1))

    # Extract the Metallicity
    for l in str(rest[2]):
        if '-' in l:
            de1 = '-'
            break
        elif '+' in l:
            de1 = '+'
            break
        else:
            pass
    p2, rest2 = substring_before(rest[2], de1)
    parameters.append(float(rest2[1] + rest2[2]))
    #####
    return matrix.values, np.array(parameters), name


def substring_before(s, delim):
    """ Divide the string 's' at 'delim' and
        return both what's before and after.
    """
    before = s.partition(delim)[0]
    all = s.partition(delim)
    return before, all
```

Code Snip B.6: Functions for extracting the parameters from a power matrix.

```
def CNN_model_FULL(input_shape):
    """ Create the model architecture. """
    model = Sequential()

    model.add(Conv2D(16, (7, 7), input_shape=input_shape, activation='relu'))
    model.add(pooling.MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(32, (7, 7), activation='relu'))
    model.add(pooling.MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(64, (7, 7), activation='relu'))
    model.add(pooling.MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(128, (5, 5), activation='relu'))
    model.add(pooling.MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(256, (5, 5), activation='relu'))
    model.add(pooling.MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(512, (5, 5), activation='relu'))
    model.add(pooling.MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(1024, (3, 3), activation='relu'))
    model.add(pooling.MaxPooling2D(pool_size=(2, 2)))

    model.add(Conv2D(1024, (3, 3), activation='relu'))
    model.add(pooling.MaxPooling2D(pool_size=(2, 2)))

    model.add(Flatten())
    model.add(Dense(50, activation='relu'))
    model.add(Dense(20, activation='relu'))
    model.add(Dense(3, activation="linear"))

    model.summary()

    return model
```

Code Snip B.7: CNN Architecture for the final model.

Appendix C – Alternative CNN Architectures

5 times down-sized input size with the resulting CNN architecture.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 738, 808, 16)	2368
max_pooling2d_1 (MaxPooling2	(None, 369, 404, 16)	0
conv2d_2 (Conv2D)	(None, 363, 398, 32)	25120
max_pooling2d_2 (MaxPooling2	(None, 181, 199, 32)	0
conv2d_3 (Conv2D)	(None, 177, 195, 64)	51264
max_pooling2d_3 (MaxPooling2	(None, 88, 97, 64)	0
conv2d_4 (Conv2D)	(None, 84, 93, 128)	204928
max_pooling2d_4 (MaxPooling2	(None, 42, 46, 128)	0
conv2d_5 (Conv2D)	(None, 38, 42, 256)	819456
max_pooling2d_5 (MaxPooling2	(None, 19, 21, 256)	0
conv2d_6 (Conv2D)	(None, 17, 19, 512)	1180160
max_pooling2d_6 (MaxPooling2	(None, 8, 9, 512)	0
conv2d_7 (Conv2D)	(None, 6, 7, 1024)	4719616
max_pooling2d_7 (MaxPooling2	(None, 3, 3, 1024)	0
flatten_1 (Flatten)	(None, 9216)	0
dense_1 (Dense)	(None, 50)	460850
dense_2 (Dense)	(None, 20)	1020
dense_3 (Dense)	(None, 3)	63

Total params: 7,464,845
Trainable params: 7,464,845
Non-trainable params: 0

Table C.1: CNN architecture for 5 times down-sized input size.

10 times down-sized input size with the resulting CNN architecture.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 366, 401, 16)	2368
max_pooling2d_1 (MaxPooling2	(None, 183, 200, 16)	0
conv2d_2 (Conv2D)	(None, 177, 194, 32)	25120
max_pooling2d_2 (MaxPooling2	(None, 88, 97, 32)	0
conv2d_3 (Conv2D)	(None, 82, 91, 64)	100416
max_pooling2d_3 (MaxPooling2	(None, 41, 45, 64)	0
conv2d_4 (Conv2D)	(None, 37, 41, 128)	204928
max_pooling2d_4 (MaxPooling2	(None, 18, 20, 128)	0
conv2d_5 (Conv2D)	(None, 14, 16, 256)	819456
max_pooling2d_5 (MaxPooling2	(None, 7, 8, 256)	0
conv2d_6 (Conv2D)	(None, 3, 4, 512)	3277312
max_pooling2d_6 (MaxPooling2	(None, 1, 2, 512)	0
flatten_1 (Flatten)	(None, 1024)	0
dense_1 (Dense)	(None, 100)	102500
dense_2 (Dense)	(None, 20)	2020
dense_3 (Dense)	(None, 3)	63

Total params: 4,534,183
Trainable params: 4,534,183
Non-trainable params: 0

Table C.2: CNN architecture for 10 times down-sized input size.

20 times down-sized input size with the resulting CNN architecture.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 182, 199, 16)	1216
max_pooling2d_1 (MaxPooling2	(None, 91, 99, 16)	0
conv2d_2 (Conv2D)	(None, 87, 95, 32)	12832
max_pooling2d_2 (MaxPooling2	(None, 43, 47, 32)	0
conv2d_3 (Conv2D)	(None, 39, 43, 64)	51264
max_pooling2d_3 (MaxPooling2	(None, 19, 21, 64)	0
conv2d_4 (Conv2D)	(None, 17, 19, 128)	73856
max_pooling2d_4 (MaxPooling2	(None, 8, 9, 128)	0
conv2d_5 (Conv2D)	(None, 6, 7, 256)	295168
max_pooling2d_5 (MaxPooling2	(None, 3, 3, 256)	0
conv2d_6 (Conv2D)	(None, 2, 2, 512)	524800
max_pooling2d_6 (MaxPooling2	(None, 1, 1, 512)	0
flatten_1 (Flatten)	(None, 512)	0
dense_1 (Dense)	(None, 100)	51300
dense_2 (Dense)	(None, 20)	2020
dense_3 (Dense)	(None, 3)	63

Total params: 1,012,519
Trainable params: 1,012,519
Non-trainable params: 0

Table C.3: CNN architecture for 20 times down-sized input size.

30 times down-sized input size with the resulting CNN architecture.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 122, 133, 16)	448
max_pooling2d_1 (MaxPooling2	(None, 61, 66, 16)	0
conv2d_2 (Conv2D)	(None, 59, 64, 32)	4640
max_pooling2d_2 (MaxPooling2	(None, 29, 32, 32)	0
conv2d_3 (Conv2D)	(None, 27, 30, 64)	18496
max_pooling2d_3 (MaxPooling2	(None, 13, 15, 64)	0
conv2d_4 (Conv2D)	(None, 12, 14, 128)	32896
max_pooling2d_4 (MaxPooling2	(None, 6, 7, 128)	0
conv2d_5 (Conv2D)	(None, 5, 6, 256)	131328
max_pooling2d_5 (MaxPooling2	(None, 2, 3, 256)	0
flatten_1 (Flatten)	(None, 1536)	0
dense_1 (Dense)	(None, 100)	153700
dense_2 (Dense)	(None, 20)	2020
dense_3 (Dense)	(None, 3)	63

Total params: 343,591
Trainable params: 343,591
Non-trainable params: 0

Table C.4: CNN architecture for 30 times down-sized input size.