Bachelor's Thesis Computing Science

Radboud University Nijmegen

Stellar Classification of Main Sequence Stars using Vision Transformers

---

Author:
Adam Farrag
s1073320

First supervisor/assessor:
prof. dr. ir. Arjen P. de Vries

Second assessor:
dr. Yuliya Shapovalova

September 22, 2024

Abstract

Astronomical surveys planned in the near future are expected to generate petabytes of data during their operation windows. The increase in data volume has led to increased demand for automated data processing solutions. This thesis explores the use of multi-band images obtained from the Sloan Digital Sky Survey for automated stellar classification using vision transformers. We have used 4 types of images and we have discovered that the best-performing image type varies depending on the star class and the size of the class representation within the training set. On O- and B-class stars, the model achieves decent performance when these classes are not severely under-represented in the training set but generally suffers from poor classification performance due to the lack of data available for these classes. With F-class stars, good performance is only achieved when they are over-represented in the training set. With G-class stars, performance is at best decent when the number of stars in the training sets are equally distributed over all classes but is very poor across all image types when the classes are not equally distributed. With K- and M-class stars, the model performs at worst reasonably well when using GRI images, with exceptional performance when using RGB images. This thesis concludes that while there is significant potential for automated stellar classification using vision transformers, further investigation into various image synthesis options to enhance star visibility is necessary in order to improve classification performance.

Contents

Chapter 1

Introduction

Astronomy is increasingly dealing with a vast variety and ever-growing volumes of data, with observations being made along the entire electromagnetic spectrum, from neutrino astronomy, gravitational waves, and simulations in theoretical astronomy. Data volumes of entire surveys from a decade ago can now be collected in a single night. One such instance is the Sloan Digital Sky Survey (SDSS), generating over 200GB of data per night. Future surveys are planned to collect even larger amounts of data, such as the Legacy Survey of Space and Time, which is expected to generate 15TB of data per night, resulting in over 200 PB of uncompressed data in its 10-year operation window. [8]

One use case for all this data is stellar classification. Stellar classification is required for a various number of reasons. The first and foremost reason is to get a better understanding of stars. By for example examining the class based on spectral lines or luminosity, other important information can be inferred about the star, such as its mass, radius, or the habitable zone of the star. This information can then be used for further analysis, such as in stellar evolution or detecting habitable planets that may contain extraterrestrial life.

As the amount of data increases, so does the demand for real-time analysis, preferably as efficiently and as accurately as possible. [10] Real-time analysis is wanted for a various number of reasons. In transient astronomical events such as supernovae, the need to mine continuous streams of data in near real-time is needed as these events are only short-lived. Analysis of such transient events is then not only required for detection but also for timely and well-chosen follow-up observations. [9] Other motives include on-the-fly sensor calibrations and the reduction of raw data to scientifically useful catalogs and images with minimum human intervention to reduce storage requirements.[7]

Automated methods are required to process the large and complex data that traditional data analysis methods can no longer keep up with. Machine learning can provide many solutions, allowing astronomers to more easily extract the data they need to perform their scientific analyses. [5] Specifically deep learning has better performance on large-scale datasets and recognition accuracy. [19] The vision transformer (ViT) has shown to have promising results in various image classification tasks. In the context of stellar classification, ViTs offer the potential to analyze multi-band imaging data and being able to capture spectral features that distinguish stars from one another. By learning these features, ViTs can automate the classification process, reducing the need for manual analysis and allowing astronomers to focus on higher-level analyses.

In this thesis, we assess the performance of vision transformers on various types of imaging data for the stellar classification of main-sequence stars. We aim to provide a framework that can be adapted and extended for various astronomical classification tasks. In chapter 2 we outline the fundamentals required to understand and reproduce this research. Chapter 3 discusses already existing work on the applications of machine learning in cosmological classification. In chapter 4 we provide the methodology. Chapter 5 contains the results measured from the created models. Chapter 6 will discuss the results as well as any shortcomings with corresponding ideas on how to overcome them. In chapter 7 we conclude this thesis and summarize key takeaways. Finally, the appendix contains all the code used for this research, as well as some examples of images from the datasets.

Chapter 2

Preliminaries

2.1 Sloan Digital Sky Survey (SDSS)

The SDSS is a dataset containing various types of data on millions of stellar objects. This thesis makes use of the latest data release, DR18. We make use of imaging data provided in RGB, as well as in ultraviolet (u), green (g), red (r), near-infrared (i), and infrared (z) wavelengths. The RGB images are synthesized using the middle bands, i, u, and z, and mapping them to the RGB channels.[1] The exact order of the mapping is not specified. SDSS contains imaging data for about one-third of the night sky.

2.2 (Vision) Transformer

The standard transformer was introduced for natural language processing in 2017 by Vaswani et al. [18] The transformer made a breakthrough in natural language processing (NLP) due to the use of self-attention mechanisms. These mechanisms allow it to draw global dependencies between input and output sequences, which in the context of NLP, means understanding the relationships between words (tokens) in a sequence. A transformer applies multi-headed self-attention, where self-attention is applied in parallel, allowing it to attend to different parts of the sequence simultaneously. This allows the model to learn the relationships between all words in a sequence. Transformers have vastly superior performance while being more parallelizable and requiring less time to train than traditional recurrent or convolutional neural networks. Since its introduction, the standard transformer has become the de facto standard for natural language processing. [6]

The Vision Transformer (ViT) was first introduced in 2020 by Dosovitskiy et al. [6] It was a proposed variation of the original Transformer specifically meant for computer vision. In a ViT, images are split into patches,

[1] https://skyserver.sdss.org/dr16/en/tools/getimg/getimghome.aspx

which are then provided as a sequence of inputs to the ViT. These image patches are treated the same way as words in the standard Transformer's natural language processing applications. Like the standard Transformer, the ViT attains excellent results compared to state-of-the-art convolutional neural networks while requiring substantially fewer computational resources to train. The ViT performs best when it is pre-trained on a large dataset and then fine-tuned on a smaller dataset for specific applications.

2.3 Main Sequence Stars

Main Sequence Stars (MSS), also known as dwarf stars, are stars that fuse hydrogen to helium in their cores. They vary in luminosity, color, and size from a tenth to 200 times the mass of the Sun and have lifespans ranging from millions to billions of years. [14] MSS are split into 7 classes, depending on the effective temperature (T_{eff}) in Kelvin (K), color, and spectral characteristics. In astronomy, spectral characteristics refer to the specific wavelengths of light that are absorbed by the materials in a star's atmosphere. The 7 classes of MSS are:

- O: O-class stars are bluish-white stars with a T_{eff} of over 30000 K. Their spectra show metals, significant amounts of ionized helium, and weak amounts of hydrogen lines. [16][17]
- B: B-class stars are also bluish-white stars but have a T_{eff} of 10000-30000 K. Their spectra show ionized metals, large amounts of neutral helium, and developing signs of hydrogen Balmer lines (absorption lines of hydrogen). [16][17]
- A: A-class stars are white stars with a T_{eff} of 7500-10000 K. Their spectra are dominated by very strong hydrogen Balmer lines and ionized singly ionized metals. [16][17]
- F: F-class stars are white stars with a T_{eff} of 6000-7500 K. Their spectra feature hydrogen Balmer lines and singly ionized metals, though these are less prominent than in A-class stars. [16][17]
- G: G-class stars are yellow stars with a T_{eff} of 5000-6000 K. Their spectra show many metal lines, with increasing amounts of neutral metals. There are also large amounts of positively charged calcium and some weaker hydrogen lines visible. [16][17]
- K: K-class stars are reddish stars with a T_{eff} of 3500-5000 K. The elements visible in their spectra show molecular bands (spectra produced by molecules) together with a dominating presence of neutral metals. [16][17]

- M: M-class stars are red stars with a T_{eff} of 2000-3500 K. Their spectra show large amounts of neutral metals, but even larger amounts of molecular bands, in particular titanium oxide. [16][17]

Each spectral class is further subdivided into the subclasses 0 to 9, with 0 being the hottest and 9 being the coolest type in the class. These star classes are also easily remembered with the mnemonic "Oh be a fine girl/guy, kiss me", where the first letter of each word represents each star class in descending order of temperature. [16][17] The stars are then also ordered from least occurring to most occurring.

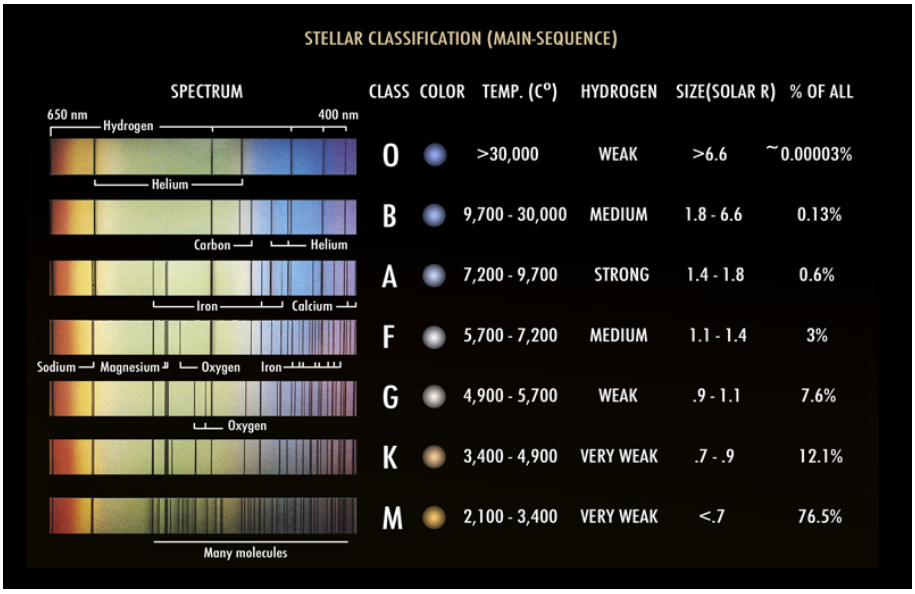


Figure 2.1: Spectral classification chart [2]

2.4 Celestial coordinates

The sky appears as a sphere over the earth, which we call the celestial sphere. Using this sphere we can define coordinates to map the universe. These coordinates consist of a right ascension (ra) and a declination (dec) coordinate, which correspond to longitude and latitude respectively. [4]

[2]By Pablo Carlos Budassi, https://commons.wikimedia.org/w/index.php?curid=92588077

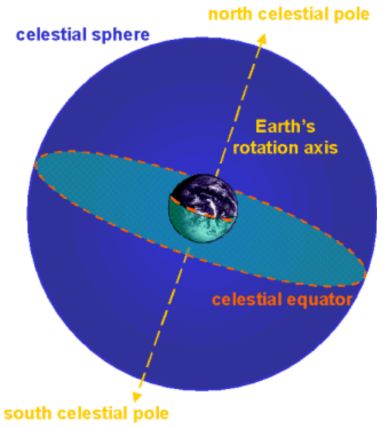


Figure 2.2: The celestial sphere surrounding Earth [3]

2.5 FITS Images

FITS images are a data type mainly used in astronomy, that supports the storage of an unlimited number of multi-dimensional arrays within the same file. This allows the storage of an image as well as tables containing data about the image. In this case, we require the header data to use in the World Coordinate System (WCS)[4]. [15] The WCS allows the transformation of real-world coordinates to pixel values and vice versa, which will allow for locating stars in full-frame images.

2.6 Image stretching

In astrophotography, images are often captured using cameras with sensors that support up to 16-bit color depth. However, traditional computer displays only support 8-bit color depth. To display these higher-depth images, we have to scale the data into an 8-bit range. This is known as stretching.

[3] https://astronomy.swin.edu.au/cosmos/C/Celestial+Sphere
[4] https://docs.astropy.org/en/stable/wcs/

Chapter 3

Related Work

There has been a handful of research performed recently on cosmological classification tasks using vision transformers.

3.1 ViT Stellar Classificaiton

The research that is most related to this thesis is also on stellar classification using SDSS images by Yang et al. [20] In their study they use individual bands provided by SDSS to synthesize images also using the method from Lupton et al. [13] These images are then used to evaluate the performance of the classification of main-sequence stars using a pre-trained ViT. However, the data collection method described in their work does not correlate with the data collection methods according to SDSS documentation. They also do not specify which band images are mapped to which color in the synthesized RGB images. In their findings, they observe an accuracy of 0.839 using images synthesized from the g, r and i bands, and an accuracy of 0.863 when combining the g, r, i images with images synthesized from the u, r, and z bands. The method for combining gri images with urz images is also not defined.

3.2 CNN vs ViT

Similar work has been done on photometric identification of compact galaxies, stars, and quasars using multiple neural networks. [3] In this study they present a new deep learning-based classifier MargNet, which consists of a combination of convolutional neural networks for image modeling and artificial neural networks for modeling photometric parameters. This model is tested on 2 classification problems, compact galaxy-star and compact galaxy-star-quasar classification, in both cases incorporating the use of faint

imaging data with faint objects. In both cases, MargNet performs better than both the CNN and ANN. This was further built upon in a study by Bhavanam et al. [2], by incorporating attention mechanisms and ViT-based models into MargNet. The resulting model with attention mechanisms marginally outperforms MargNet and the proposed ViT-based MargNet models. However, the ViT-based MargNet models are the most lightweight and easiest to train, while providing nearly identical performance to that of the attention-enhanced MargNet model. [2]

3.3 Pre-trained ViT vs ViT trained from scratch

In a study by Kumar et al., a comparison is made between fine-tuning a ViT pre-trained on the ImageNet dataset and training a ViT from scratch for galaxy morphology classification. They show that training a ViT from scratch on the entire dataset results in an accuracy of 69.49%, while this accuracy is beaten by the pre-trained ViT when training on only 50% of the dataset. Training the pre-trained ViT on the entire dataset resulted in an accuracy of 75.53%. [11]

Chapter 4

Research

4.1 Collecting and preparing star catalogue data

4.1.1 Collecting star catalogue

We will use the Sloan Digital Sky Survey (SDSS) dataset, which contains data on more than 1 million stars.[1] The catalog data can be collected from the SDSS CasJobs SkyServer[1], which allows selecting information from the catalog via SQL queries. See Appendix A.1 for the SQL query used to only select stars from the catalog of Data Release 18 (DR18). An account is required to access the data, but this can be created for free. After querying, a CSV file of this data can be downloaded from the MyDB tab.

4.1.2 Filtering duplicates and unusable entries

The obtained data contains duplicates and some entries which do not contain values for the rerun, run, camcol, or field columns. These entries must be removed, as the duplicate entries have the same imaging data, and imaging data cannot be collected for entries without rerun, run, camcol, or field values. See Appendix A.2 for the code to remove duplicates from the resulting CSV file and Appendix A.3 for the code to remove entries without rerun, run, camcol, or field values from the CSV file.

4.1.3 Assigning main class classifications

The dataset provides the subclass for each star, but as we only care about the main class, we add a new column 'mainclass', for the main class of the star, and fill entries by taking the first character of the star's subclass. After filtering we are left with 3601, 11734, 84869, 102686, 167869, 184638, and 382311 entries for the O, B, G, A, K, M and F classes respectively.

[1] https://skyserver.sdss.org/CasJobs

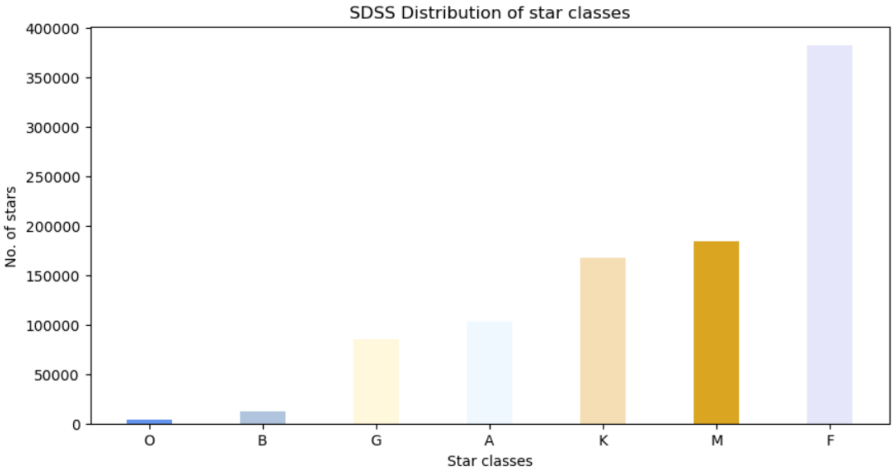


Figure 4.1: Distribution of stars in filtered dataset

4.1.4 Obtaining unique camera frame entries and the star count per frame

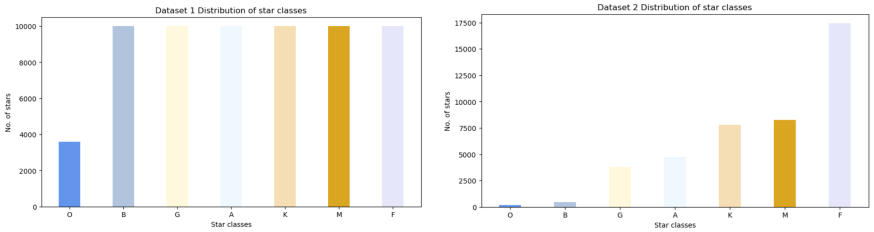
The unique camera frame entries are required to obtain images taken in the different bands of light. We also want to know how many star entries there are per FITS file, as this reduces the amount of data that has to be downloaded. See Appendix A.4 for the code to obtain unique camera frame entries with the respective number of stars per frame.

4.1.5 Collecting unique camera frame entries

We chose to download FITS files containing the most number of stars, in order to reduce the number of FITS files that must be downloaded. We will collect 3 datasets in order to evaluate the performance of each image type. The first dataset is created with 10000 samples per class, except for the O class of which we sample all 3601 entries. The code for this is provided in Appendix A.5.1. The second dataset is randomly sampled to have less entries, at ∼42000. The third dataset is also randomly sampled, this time with ∼75000 entries. The code for this can be found in Appendix A.5.2, where the `target_count` variable should be adjusted to the size of the dataset. All proceeding steps are identical for all 3 sets.

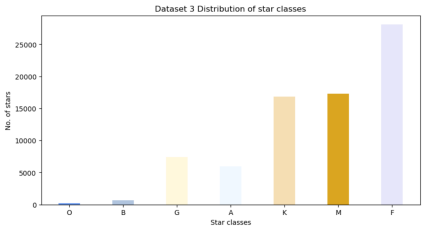
4.1.6 Converting camera frame entries back to star data entries

The data that we have does not contain any information about the stars, so we must convert it back to individual star data. This is done by only



(a) Dataset 1

(b) Dataset 2



(c) Dataset 3

Figure 4.2: Distribution of star classes

selecting the stars which have the same rerun, run, camcol, and field values as the values in the camera frame entries. See Appendix A.6 for the code on how to do so.

4.1.7 Splitting data into entries for training, validation, and testing sets

Now that we have all our data, we can split the data into training, validation, and testing sets. We will split each class into 70% training, 10% validation, and 20% testing. See Appendix A.7 for the code on how to split the data.

4.1.8 Getting unique camera frames for FITS images

The final part of filtering the data is to convert each of the sets of stars back to unique camera frame values in order to download the FITS images. This is done by selecting the unique combinations of run, rerun, camcol, and field values. See Appendix A.8 for the code on how to do so. The CSV read and save names can be modified to also perform these actions on the validation and testing sets.

4.2 Collecting star imaging data

4.2.1 RGB Images

The SDSS provides an image cutout tool[2], which will provide RGB images for objects given their stellar coordinates. We can make use of this, as we have ra and dec coordinates for each of the stars in our datasets. The URL to the RGB images is built using the ra and dec coordinates of the star, as well as a scale value that dictates how zoomed in the image is, and a width and height parameter that define the pixel dimensions of the image. In our case, we will use a scale of 0.025, as this provides enough zoom to block out most other objects in the image, but is not zoomed in so far that the star is no longer recognizable as a star. We will use an image dimension of 224x224 pixels, as this is the size image that is accepted by our pre-trained ViT. The images can then be downloaded using the get from the requests python package. See Appendix B.1 for the code for the aforementioned. This code is adapted for the validation and testing sets, by simply changing the names of the input CSV file, the save directory, and the log file.

4.2.2 FITS Images

FITS images for each band of light can be obtained from SDSS Science Archive Server (SAS)[3], compressed in the bz2 format. We need to extract the rerun, run, camcol, and field values of each star in order to construct the URL for the required FITS images. We extract these values from the unique camera entries for each of the sets. After extracting these values, the URL is created and the FITS bz2 files are retrieved using wget. See Appendix B.2 for the code. The `generate_urls_and_dirs` function ensures that the values are in the correct format required for the URL.

4.2.3 Decompressing FITS bz2 files

In order to get the raw FITS files, we must decompress the bz2 files that were downloaded. Each image band is saved to the respective band directories. The code on how to do so can be found in Appendix B.3. Here again, the name of the source and destination folders can be modified to the set that is being worked on.

4.2.4 Cropping stars out of full frame FITS images

We have to crop the individual stars out of the full-frame FITS images for each band. This can be done using the World Coordinate System, which maps real-world coordinates to pixel values in the image. Unlike the RGB

[2]https://skyserver.sdss.org/dr18/VisualTools/list
[3]https://dr18.sdss.org/sas/

images, scaling is not available when cropping, so we must choose a crop size small enough to minimize the visibility of other objects. We opt to crop stars at a resolution of 64x64 pixels.

4.2.5 Synthesizing RGB image from individual light bands

In addition to the RGB images provided by SDSS, we will also be making use of RGB images synthesized from individual light bands. To synthesize RGB images from individual light bands, we have to resize all cropped images to 64x64 pixels. This is due to some images being smaller than 64x64, as they lie on the border of the frame image. We can then create RGB images from the individual band images using the Lupton RGB method by Lupton et al.[13] This can be done by providing a specific image for the R, G, and B bands of the final image. Furthermore, the method uses α (stretch) and Q values. The α value determines the non-linearity of the stretch applied and can be set in [0-1]. The Q value determines at which point the algorithm compresses bright pixel values to prevent the image from being over-saturated. We make use of $\alpha = 0.02$ and $Q = 8$. Furthermore, we make use of 3 types of synthesized images. The first image has the G, R, and I images mapped to the R, G, and B channels respectively. The second image has the U, R, and Z images mapped to the R, G, and B channels respectively. See Appendix B.5 for the code. The images must then be sorted into folders for each class. This can be done using the code in Appendix B.6. The final image is created by combining the GRI and URZ images into 1 image by sticking them side by side. The code for this process can be found in Appendix B.7. Some example images can be found in Appendix D.

4.2.6 Upscaling images to 224x224

We must now upscale images to 224x224, as this is the image size required for the ViT. This can be done using the image resize function when using the PIL package. We opt to use Lanczos resampling. The code can be found in Appendix B.8. The input and output directory names can be modified to upscale the different sets of data.

4.3 ViT

At no point in the following sections should you call `torch.to_device()`, as this will result in CUDA errors which are extremely uninformative. Should you encounter these errors, it is advised to run on CPU until you receive a more clear error. The ViT makes the `torch.to_device()` call internally. The CUDA compilation tools version is `release 12.2, V12.2.140`.

4.3.1 Preparing dataset for ViT

The ViT expects the data to be in a custom DatasetDict object which contains the training, validation and testing data, which can be loaded using `load_dataset` while specifying that the data type within the folders are images. In order to avoid having a lengthy loading process every time we want to use the data, we save the dataset dictionary to disk. The code can be found in Appendix C.1 and C.2. Again, modify the `data_dir` names and the dictionary save name to load the different types of imaging data being worked on.

4.3.2 Importing ViT

We will be making use of a ViT from the Hugging Face that is pre-trained on the ImageNet-21k dataset[4]. Importing the ViT can be done using the transformers package. See Appendix C.3.

4.3.3 Transforming images to ViT inputs

The images must be transformed into pixel values to be proper input for the ViT. We use the `transform` function for this. This function will transform each batch of images into a batch of tensors so that it can be fed to the ViT. This function must also be modified depending on the imaging data that is being used for the ViT. In the case of the SDSS RGB images, they can simply be selected and passed to the processor to be transformed into inputs. In the case of the synthesized RGB images, we must convert the image to RGB, as these images contain a fourth "A" channel for the opacity. We can then create the transformed dataset using `dataset_dict.with_transformer(transform)`. The code is available in Appendix C.4, where the `rgb_images` variable should be changed to the commented line when using synthesized images.

4.3.4 Collate batches

The `collate_fn` function creates batches for the ViT by transforming the data dictionary into batch dictionaries. See Appendix C.5.

4.3.5 Evaluation Metrics

To evaluate the performance of the models, we make use of the metrics accuracy, weighted precision, weighted recall, and weighted F1-score. These metrics can be defined using the `compute_metrics` function, see Appendix C.6. The 4 metrics are defined as follows:

[4]https://huggingface.co/google/vit-base-patch16-224

$$\text{Accuracy} = \frac{\text{True Positives}}{\text{Total Samples}}$$

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives+False Positives}}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives+False Negative}}$$

$$\text{F1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision+Recall}}$$

When evaluating each of the classes individually, we make use of precision, recall, and F1-score.

4.3.6 Classification Head

To ensure that the model's final layer has the same number of outputs as there are star classes, we will pass the number of labels to the model. We also create and id2label and label2id dictionary, which allows for the labels to be converted to indices for the model and vice versa for human readability. See Appendix C.7.

4.3.7 Initializing training arguments

The `TrainingArguments` object defines the arguments for the trainer. We will use a maximum number of training epochs of 50, a batch size of 32 and a learning rate of $1 \cdot 10^{-5}$. The optimization function that is used by default is the AdamW optimizer, which is the Adam optimizer with weight decay implemented in order to avoid the convergence problems in Adam. [12] The `load_best_model_at_end` variable is set to true in order to have the best performing model at the end. All other variables can be modified to personal preference.

4.3.8 Early stopping

To prevent over-fitting the model and needlessly wasting resources, we define an early stopping variable, which is an `EarlyStoppingCallback` object. In this object we define the `early_stopping_patience` as 5, which tells the model to stop training when the model has not improved for `early_stopping_patience` evaluation calls. The `early_stopping_threshold` defines the threshold that the `metric_for_best_model` must improve by in order to not trigger an early stop, which we set to 0.001.

4.3.9 Trainer

The trainer is defined using the `model`, `training_args`, `collate_fn` function, `compute_metrics` function, `prepared_ds` dataset, `processor` and `early_stopping_callback` that we have used or defined previously. See Appendix C.10 for the details.

4.3.10 Training the model

The trainer can then be called using `trainer.train()` to start training. Once this is completed the model can be saved using `trainer.save_model` and the metrics can be logged and saved. Finally, the state of the trainer is saved. For the exact code see Appendix C.11.

4.3.11 Evaluating the model

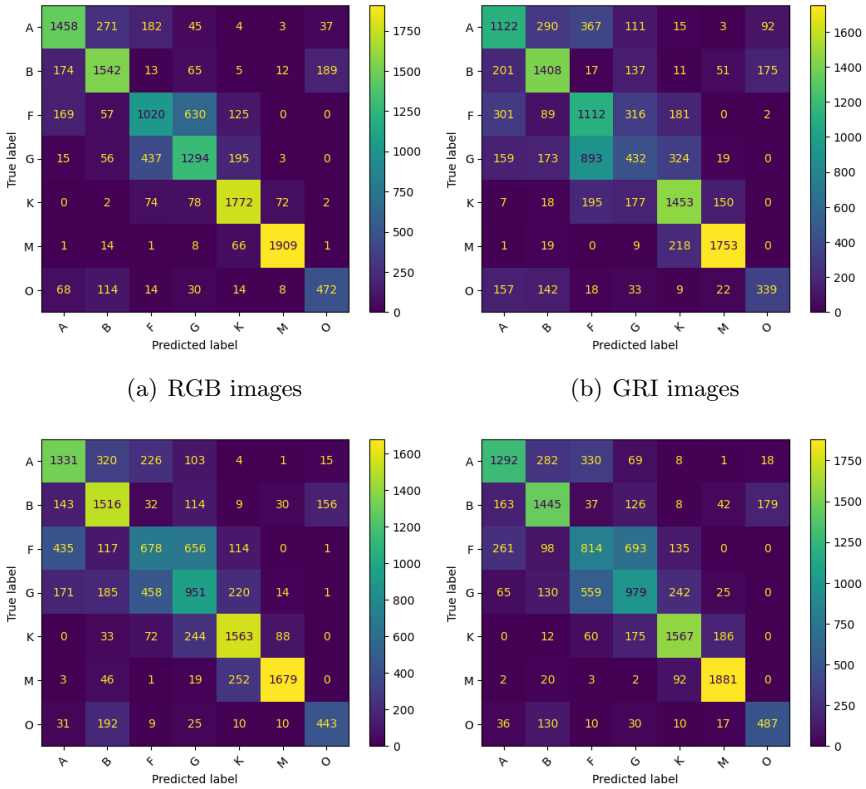
The final step is to evaluate the model using the testing set. This can be done by calling `trainer.predict(prepared_ds["test"])`. The metrics can then be logged and saved. For the exact code see Appendix C.12.

Chapter 5

Results

Dataset	Accuracy	F1	Precision	Recall
Dataset 1 RGB	0.7442	0.742	0.7417	0.7442
Dataset 1 GRI	0.5989	0.5898	0.59	0.5989
Dataset 1 URZ	0.6415	0.6374	0.6397	0.6415
Dataset 1 GRI & URZ	0.6654	0.6625	0.661	0.6654
Dataset 2 RGB	0.7719	0.7475	0.7421	0.7719
Dataset 2 GRI	0.6799	0.6387	0.6117	0.6799
Dataset 2 URZ	0.684	0.622	0.6024	0.684
Dataset 2 GRI & URZ	0.7047	0.6815	0.6708	0.7047
Dataset 3 RGB	0.7841	0.7479	0.7594	0.7841
Dataset 3 GRI	0.7181	0.6936	0.6903	0.7181
Dataset 3 URZ	0.7348	0.7018	0.6975	0.7348
Dataset 3 GRI& URZ	0.7613	0.7268	0.7327	0.7613

Table 5.1: Model accuracy, f1, precision and recall scores per dataset and image type.



(a) RGB images

(b) GRI images

(c) URZ images

(d) GRI & URZ images

Figure 5.1: Confusion matrices for dataset 1

Class	Precision	Recall	F1
O	0.67	0.66	0.66
B	0.75	0.77	0.76
A	0.77	0.729	0.75
F	0.59	0.51	0.55
G	0.60	0.65	0.62
K	0.81	0.89	0.85
M	0.95	0.95	0.95

Table 5.2: Accuracy, F1, Precision, Recall scores dataset 1 RGB

Class	Precision	Recall	F1
O	0.56	0.47	0.51
B	0.66	0.70	0.68
A	0.58	0.56	0.57
F	0.43	0.56	0.48
G	0.36	0.22	0.27
K	0.66	0.73	0.69
M	0.88	0.88	0.88

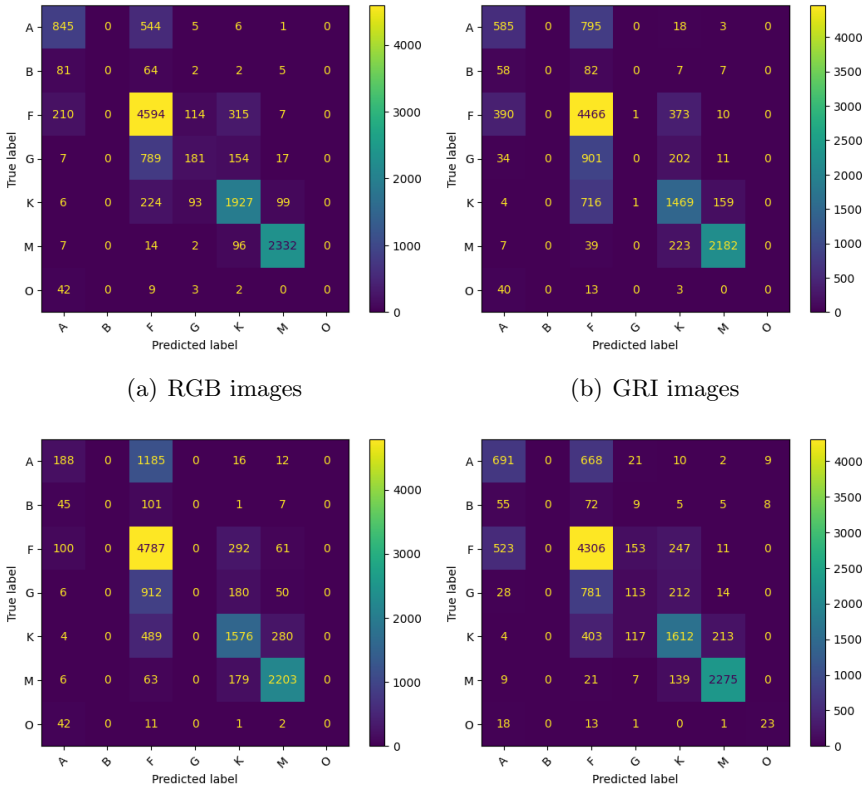
Table 5.3: Accuracy, F1, Precision, Recall scores dataset 1 GRI

Class	Precision	Recall	F1
O	0.72	0.62	0.66
B	0.63	0.76	0.69
A	0.63	0.67	0.65
F	0.46	0.34	0.39
G	0.45	0.48	0.46
K	0.72	0.78	0.75
M	0.92	0.84	0.88

Table 5.4: Accuracy, F1, Precision, Recall scores dataset 1 URZ

Class	Precision	Recall	F1
O	0.71	0.68	0.69
B	0.68	0.72	0.70
A	0.71	0.65	0.68
F	0.45	0.41	0.43
G	0.47	0.49	0.48
K	0.76	0.78	0.77
M	0.87	0.94	0.91

Table 5.5: Accuracy, F1, Precision, Recall scores dataset 1 GRI & URZ



(a) RGB images

(b) GRI images

(c) URZ images

(d) GRI & URZ images

Figure 5.2: Confusion matrices for dataset 2

Class	Precision	Recall	F1
O	Undefined	0	Undefined
B	Undefined	0	Undefined
A	0.71	0.60	0.65
F	0.74	0.88	0.80
G	0.45	0.16	0.23
K	0.77	0.82	0.79
M	0.95	0.95	0.95

Table 5.6: Accuracy, F1, Precision, Recall scores dataset 2 RGB

Class	Precision	Recall	F1
O	Undefined	0	Undefined
B	Undefined	0	Undefined
A	0.52	0.42	0.46
F	0.64	0.85	0.73
G	0	0	0
K	0.64	0.63	0.63
M	0.92	0.89	0.90

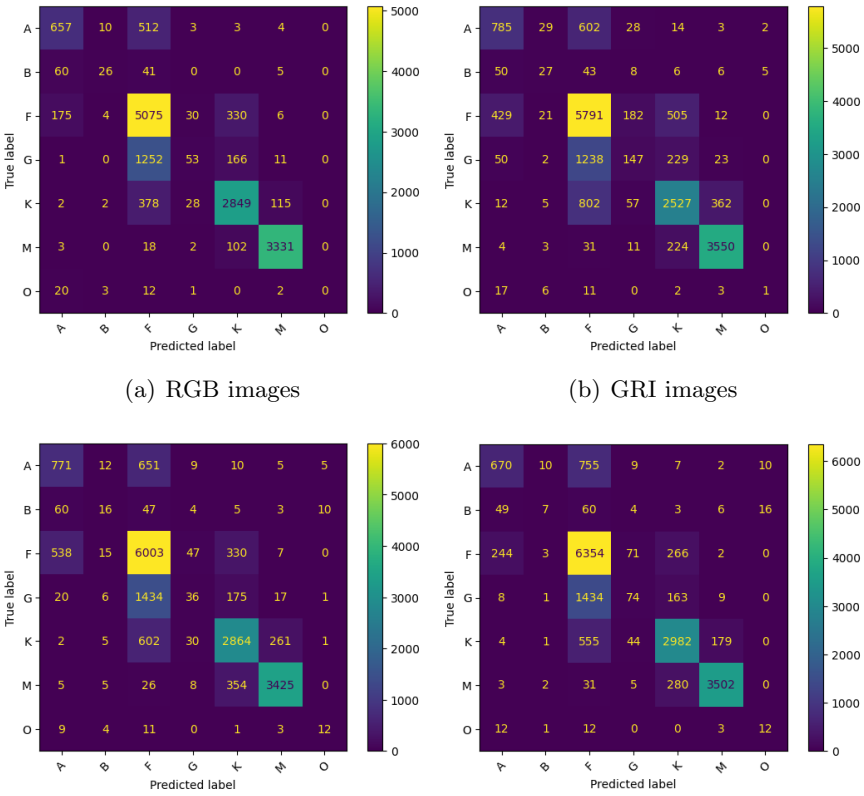
Table 5.7: Accuracy, F1, Precision, Recall scores dataset 2 GRI

Class	Precision	Recall	F1
O	Undefined	0	Undefined
B	Undefined	0	Undefined
A	0.48	0.13	0.21
F	0.63	0.91	0.75
G	Undefined	0	Undefined
K	0.70	0.67	0.69
M	0.84	0.90	0.87

Table 5.8: Accuracy, F1, Precision, Recall scores dataset 2 URZ

Class	Precision	Recall	F1
O	0.58	0.41	0.48
B	Undefined	0	Undefined
A	0.51	0.48	0.49
F	0.69	0.82	0.75
G	0.27	0.10	0.14
K	0.72	0.69	0.70
M	0.90	0.93	0.92

Table 5.9: Accuracy, F1, Precision, Recall scores dataset 2 GRI & URZ



(a) RGB images

(b) GRI images

(c) URZ images

(d) GRI & URZ images

Figure 5.3: Confusion matrices for dataset 3

Class	Precision	Recall	F1
O	Undefined	0	Undefined
B	0.58	0.20	0.29
A	0.72	0.55	0.62
F	0.70	0.90	0.79
G	0.45	0.04	0.07
K	0.83	0.84	0.83
M	0.96	0.96	0.96

Table 5.10: Accuracy, F1, Precision, Recall scores dataset 3 RGB

Class	Precision	Recall	F1
O	0.13	0.03	0.04
B	0.29	0.19	0.23
A	0.58	0.54	0.56
F	0.68	0.83	0.75
G	0.34	0.09	0.14
K	0.72	0.67	0.69
M	0.90	0.93	0.91

Table 5.11: Accuracy, F1, Precision, Recall scores dataset 3 GRI

Class	Precision	Recall	F1
O	0.41	0.23	0.30
B	0.25	0.11	0.15
A	0.55	0.53	0.54
F	0.68	0.86	0.76
G	0.27	0.02	0.04
K	0.77	0.76	0.76
M	0.92	0.90	0.91

Table 5.12: Accuracy, F1, Precision, Recall scores dataset 3 URZ

Class	Precision	Recall	F1
O	0.32	0.3	0.31
B	0.28	0.05	0.08
A	0.68	0.46	0.55
F	0.69	0.92	0.79
G	0.36	0.04	0.08
K	0.81	0.79	0.80
M	0.95	0.92	0.93

Table 5.13: Accuracy, F1, Precision, Recall scores dataset 3 GRI & URZ

Chapter 6

Discussion

6.1 Overall model performance

From Table 5.1 we observe that in all cases, models training on the RGB images provided by SDSS have a higher performance. In the worst case, the RGB model is ∼2% more accurate than the GRI and URZ model while in the best case being ∼8 better. The standalone GRI and URZ are consistently outperformed by the combined GRI and URZ models. The combined GRI and URZ models are ∼2% more accurate than the URZ models and ∼2-7% more accurate than the GRI models. It should be noted that the F1-scores for the models using dataset 2 are ill-defined, as each model doesn't predict at least 2 of the 7 classes. These findings do not reflect the findings of related work. This can be due to several factors, of which the most likely explanation is discrepancies in the image (pre)processing pipeline.

6.2 Model performance per class

6.2.1 O-class stars

O-class observations benefit from the use of combined GRI and URZ as correct predictions are made more often when using GRI and URZ bands together. In dataset 1 the F1 score of the combined GRI and URZ model is 0.69, being slightly better than the RGB and URZ models with an F1 score of 0.66 and triumphing over the GRI model's F1 score of 0.51. In dataset 2, no predictions are made for the O-class in the RGB, GRI, and URZ models. In the combined GRI and URZ model, an underwhelming F1 score of 0.48 is achieved. In dataset 3 we observe an F1 score for the combined GRI and URZ model of 0.31, which is barely any better than the F1 score of 0.30 for the URZ model. The lack of classification in dataset 2 and the low F1 scores are likely a result of the lack of data for this class. When misclassified, they are mostly classified as B-class stars, which is not unreasonable considering

similar spectral characteristics.

6.2.2 B-class stars

B-class stars benefit most from the RGB images. They decently predicted by all models trained on dataset 1, with an F1 score of 0.76 for the RGB model and scores of 0.68, 0.69, and 0.70 for GRI, URZ, and combined GRI and URZ models respectively. In dataset 2, B-class stars are not predicted at all, mostly being predicted as A- and F-class stars. This does not significantly improve in dataset 3, achieving an f1 score of 0.29, 0.23, 0.15, and 0.08 for the RGB, GRI, URZ, and combined GRI and URZ models respectively. The misclassification in dataset 1 as A-class stars can be explained by their spectral similarities. In dataset 2 and 3, the increase in misclassifications as F-class stars is likely due to the F-class stars being over-represented.

6.2.3 A-class stars

A-class stars are predicted best by RGB images. The F1 scores vary greatly between models and datasets, with the best F1 score of 0.75 being achieved using RGB images of dataset 1, and the worst score of 0.21 being achieved using URZ images from dataset 2. When incorrectly classified in dataset 1, they are classified as B- and F-class stars due to similar spectral characteristics. In dataset 2 and 3, they are mainly classified at F-class stars due to the F-class stars being over-represented.

6.2.4 F-class stars

F-class stars are best predicted using RGB images. Achieving an F1 score of 0.55 in dataset 1 when using RGB images, compared to 0.48, 0.39, and 0.43 when using GRI, URZ, and combined GRI and URZ images respectively. In dataset 2 and 3 the difference in F1 score is smaller at only 0.07 between the best-performing RGB models and the worst-performing synthesized image models. In dataset 1 F-class stars are mostly misclassified as A- and G-class stars as they share many spectral characteristics. In dataset 2 and 3 they are also misclassified as K-class stars, which requires further analysis, as while they do share some spectral characteristics, they also have certain characteristics which are very dissimilar. It is also notable that the F-class stars only get good predictions when they are over-represented, as is the case with dataset 2 and 3.

6.2.5 G-class stars

G-class stars have low performance and are often misclassified as F- and K-class stars and in the case of URZ images on dataset 2, G-class stars are not predicted at all. The best F1 score on G-class stars is 0.62 in the dataset

1 RGB model, with all but 1 other model having F1 scores of ≤ 0.27. This is likely due to how we map bands to the RGB channels. G-class stars emit yellow light, but by making use of G, R, I, U, and Z bands, we barely provide any information about the yellow part of the visible light spectrum. The F-class stars being slightly whiter have more information available in the U band, and the K-class stars being slightly redder have more information available in the I, R, and Z bands. As a result, it becomes difficult for the models to distinguish G-class stars from the slightly whiter F-class and slightly redder K-class stars.

6.2.6 K-class stars

K-class stars have good performance, but benefit the most from RGB images, achieving an F1 score of 0.85 in dataset 1, 0.79 in dataset 2, and 0.83 in dataset 3. The combined GRI and URZ models closely follow with F1 scores worse by 0.03-0.09. The URZ models have F1 scores of 0.01-0.04 worse than the combined GRI and URZ model, followed by the GRI models which have F1 scores worse than those of the URZ models by 0.06-0.07. The good classification performance using RGB images can be explained by the fact that K-class stars are reddish stars and by the I, U, and Z bands being used to create the RGB images. These bands result in more information from the red side of the electromagnetic spectrum, thus making it easier to classify redder stars.

6.2.7 M-class stars

M-class stars have the best and most consistent performance across all datasets and imaging types. They benefit the most from RGB images, achieving an F1 score of 0.95 in dataset 1 and 2 and 0.96 in dataset 3. In all other datasets they achieve a high F1 score of 0.87-0.93. The small number of M-class stars that are misclassified are classified as K-class stars, which is reasonable considering their similar spectral characteristics. The good classification performance can again be explained by the bands in the RGB, GRI, and URZ images all providing more information on the red side of the electromagnetic spectrum.

6.3 Skewed data

As can be seen in Figure 4.1, the data is extremely skewed, with the B and O class stars being significantly under-represented. Unfortunately, this cannot be solved by using more SDSS data but requires the use of observations from numerous astronomical surveys. This raises the problem of surveys not making observations in the same way as other surveys, which then requires more pre-processing. Alternatively, augmented versions of the under-represented

stars can be added to the dataset, or the under-represented classes can be removed entirely, resulting in a model that can predict the remaining classes with more certainty. However, depending on the specific use case, such a solution may not be desirable.

6.4 Data collection

As is visible in the confusion matrices for the datasets, not all image sets have the same number of images as the other sets. This has to do with how we obtain the data. As we make use of requests and wget to download the data, it is prone to failed or corrupted downloads. When paired together with errors from the SDSS ImageCutout tool occasionally not returning an image, or the SDSS archive sometimes not containing a FITS file, we can get quite varying amounts of imaging data per sampled dataset. In the best-case scenario, this is at most a handful of images. However, as can be seen in the confusion matrix for Dataset 3, the sets can differ by thousands of images. In this case, the training, validation, and testing sets for RGB and synthesized images are not of equal size. This problem makes it more difficult to directly compare the models. The ideal solution would be to access the data on-site or by analyzing the data as it is being collected from the surveys. As this is not possible in many cases, an alternative could be to remove entries that are not present in both RGB and synthesized sets, but as the dataset is already skewed this could result in having even fewer samples from the under-represented classes.

6.5 Images

6.5.1 Cropping and upscaling

The star crops from the FITS images were cropped at 64x64. At this cropping size, it is not rare to see other stellar objects surrounding the stars. Therefore, a better choice would be to reduce the cropping size to 32x32. However, this introduces a problem with the upscaler. The decision was made to use Lanczos, which makes use of interpolation for upscaling or downscaling images. It has good upscaling performance when the upscaling factor is not too large, and it is more efficient than any AI upscaling method. When upscaling images from 32x32 to the required 224x224, the upscaling factor is too large, and upscaling with Lanczos becomes undesirable due to the amount of artifacts that are introduced. Thus, when upscaling 32x32 images, it would be better to use an AI upscaler, such as Topaz Labs Photo

AI[1] or Upscayl[2]. Naturally, this comes at the cost of larger computational requirements.

6.5.2 Stars on frame edges

The next issue is stars that lie on the border of the image frames, resulting in a crop of a smaller size than desired. We currently solve this by resizing the image to the correct size. However, this can be improved upon by combining 2 adjacent frames into a single frame and then performing the crop. The fields overlap by 128 pixels along the scanning frame, so this should be feasible.[3] This ensures that the entire star is properly visible in the final image. The WCS header from both images will then have to be combined to get the correct coordinates for the stars.

6.5.3 Synthesized images

The next point has to do with the Lupton RGB synthesizing method. In some cases, the synthesized image has a very recognizable star in the middle, with little noise surrounding it. However, in other cases, the image contains a lot of noise, resulting in the star being more difficult to see and in some cases indistinguishable from the surrounding noise. This noise will be taken along when upscaling, resulting in unusable images. Therefore, various values for Q and α should be tested to get images that result in visible stars in the center, with minimal amounts of noise surrounding them.

Furthermore, the mapping of image bands to channels in the RGB images should also be tested more. With the 5 bands, a total of 60 possible mappings can be made to RGB images. In this thesis, we have only tested 3 types of images that can be created, so there is more research needed to discover the best possible mappings.

[1] https://www.topazlabs.com/topaz-photo-ai
[2] https://upscayl.org/
[3] https://skyserver.sdss.org/dr16/en/tools/getimg/getimghome.aspx

Chapter 7

Conclusions

In this thesis, we have explored the use of individual band images for automated stellar classification using vision transformers. The results demonstrate the potential for creating a classifier with strong overall performance. However, the model's performance on certain star classes, particularly the O-, B- and G-type stars, leaves significant room for improvement.

We have provided a few ideas on how to overcome certain challenges in order to improve the classification performance. Most importantly, a more diverse and balanced dataset is necessary, specifically with an increase in O- and B-class star observations. Furthermore, additional work is needed to investigate the performance of the different RGB images that can be generated from individual band images, as well as tuning of the parameters used in the Lupton RGB function to allow for images with clear stars and minimal noise.

Bibliography

[1] Almeida et al. The Eighteenth Data Release of the Sloan Digital Sky Surveys: Targeting and First Spectra from SDSS-V. *apjs*, 267(2):44, August 2023.

[2] Srinadh Reddy Bhavanam, Sumohana S. Channappayya, Srijith P. K, and Shantanu Desai. Enhanced astronomical source classification with integration of attention mechanisms and vision transformers. *Astrophysics and Space Science*, 369(8):92, August 2024.

[3] Siddharth Chaini, Atharva Bagul, Anish Deshpande, Rishi Gondkar, Kaushal Sharma, M Vivek, and Ajit Kembhavi. Photometric identification of compact galaxies, stars, and quasars using multiple neural networks. *Monthly Notices of the Royal Astronomical Society*, 518(2):3123–3136, January 2023.

[4] Arnab Rai Choudhuri. *Astrophysics for Physicists*. Cambridge University Press, Cambridge, 2010.

[5] Seán Cody, Sebastian Scher, Iain McDonald, Albert Zijlstra, Emma Alexander, and Nick Cox. Machine learning based stellar classification with highly sparse photometry data. *Open Research Europe*, 4:29, 02 2024.

[6] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. June 2021.

[7] Jurić et al. The LSST Data Management System, December 2015. arXiv:1512.07914 [astro-ph].

[8] Kavli Institute for Particle Astrophysics and Cosmology. Vera Rubin Observatory's Legacy Survey of Space and Time | Kavli Institute for Particle Astrophysics and Cosmology (KIPAC).

[9] Matthew J. Graham, S. G. Djorgovski, Ashish Mahabal, Ciro Donalek, Andrew Drake, and Giuseppe Longo. Data challenges of time domain astronomy. *Distributed and Parallel Databases*, 30(5):371–384, October 2012.

[10] Jan Kremer, Kristoffer Stensbo-Smidt, Fabian Gieseke, Kim Steenstrup Pedersen, and Christian Igel. Big universe, big data: Machine learning and image analysis for astronomy. *IEEE Intelligent Systems*, 32(2):16–22, 2017.

[11] Rahul Kumar, Md Kamruzzaman Sarker, and Sheikh Rabiul Islam. Vision transformers for galaxy morphology classification: Fine-tuning pre-trained networks vs. training from scratch. In Donatello Conte, Ana Fred, Oleg Gusikhin, and Carlo Sansone, editors, *Deep Learning Theory and Applications*, pages 115–125, Cham, 2023. Springer Nature Switzerland.

[12] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.

[13] Robert Lupton, Michael R. Blanton, George Fekete, David W. Hogg, Wil O'Mullane, Alex Szalay, and Nicholas Wherry. Preparing Red-Green-Blue Images from CCD Data. *Publications of the Astronomical Society of the Pacific*, 116(816):133, February 2004. Publisher: IOP Publishing.

[14] NASA Science. Types of stars. `https://science.nasa.gov/universe/stars/types/`, 2023. Accessed: 07-09-2024.

[15] JD Ponz, RW Thompson, and JR Munoz. The fits image extension. *Astronomy and Astrophysics Suppl., Vol. 105, p. 53-55 (1994)*, 105:53–55, 1994.

[16] Roger J. Tayler. *The Stars: Their Structure and Evolution*. Cambridge University Press, June 1994.

[17] A.G.G.M. Tielens. Stars, spectroscopy of. In John C. Lindon, editor, *Encyclopedia of Spectroscopy and Spectrometry*, pages 2199–2204. Elsevier, Oxford, 1999.

[18] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. 6 2017.

[19] Pin Wang, En Fan, and Peng Wang. Comparative analysis of image classification algorithms based on traditional machine learning and deep learning. *Pattern Recognition Letters*, 141:61–67, 2021.

[20] Yi Yang and Xin Li. Stellar Classification with Vision Transformer and SDSS Photometric Images. *Universe*, 10(5):214, May 2024. Number: 5 Publisher: Multidisciplinary Digital Publishing Institute.

Appendix A

Catalogue collection and filtering

This appendix contains the SQL query and code required to collect and filter data that is needed to obtain the star imaging data.

A.1 CasJobs SQL query

```
SELECT specobjid, class, subclass, type, ra, dec, rerun, run, camcol, field
INTO mydb.MSS_with_duplicates
FROM SpecPhotoAll
WHERE class = 'STAR' AND NOT subclass = 'CV'
```

A.2 Removing duplicates

```
import pandas as pd

# Load the CSV file
df = pd.read_csv('MSS_with_duplicates_afarrag.csv')

# Drop duplicates and keep 1st occurrence
df_unique = df.drop_duplicates(subset=['ra', 'dec'], keep='first')

# Save resulting dataframe to CSV
df_unique.to_csv('MSS_with_duplicates_removed_afarrag.csv', index=False)

print("CSV file with unique entries created successfully.")
```

A.3 Removing entries with NULL values in rerun, run, camcol or field columns

```
import pandas as pd

data = pd.read_csv('MSS_with_duplicates_removed_afarrag.csv',
↪  delimiter=',' )

# Read ra and dec coordinates as floats
data['ra'] = data['ra'].astype(float)
data['dec'] = data['dec'].astype(float)

main_classes = ['O', 'B', 'A', 'F', 'G', 'K', 'M']
filtered_data =
↪  data[data['subclass'].str.startswith(tuple(main_classes))]
↪  .reset_index(drop=True) # Filter to only get main sequence
↪  stars
filtered_data['mainclass'] = filtered_data['subclass'].str[0]
↪  # Add main class column
filtered_data = filtered_data.dropna(subset = ['rerun', 'run',
↪  'camcol', 'field']) # Drop all entries that have null
↪  values for rerun, run, camcol and/or field, as these are
↪  required to download the fits files.
filtered_data.to_csv('MSS_with_duplicates_and_NULL_removed_afarrag.csv',
↪  index=False) # Save DF to CSV
```

A.4 Obtaining unique image entries and star count

```
import pandas as pd

data =
↪  pd.read_csv('MSS_with_duplicates_and_NULL_removed_afarrag.csv')
count_per_entry = data.groupby(['rerun', 'run', 'camcol',
↪  'field', 'mainclass']).size().reset_index(name='count')
count_per_entry_sorted =
↪  count_per_entry.sort_values(by='count', ascending=False)
count_per_entry_sorted.to_csv
↪  ('MSS_unique_cam_entry_counts_with_duplicates_and_NULL_removed_afarrag.csv',
↪  index=False)
```

A.5 Selecting image entries

A.5.1 Selecting with restriction of at least 10000 samples per class, and 3601 samples for class O

```
import pandas as pd

# Read the CSV file
df =
↪  pd.read_csv('MSS_unique_cam_entry_counts_with_duplicates_and_NULL_removed
↪  _afarrag.csv')

# Group by mainclass
grouped = df.groupby('mainclass')

# Function to select entries until sum reaches the threshold,
↪  prioritizing higher counts
# Higher counts per file means less individual fits files that
↪  have to be downloaded
def select_entries(group, threshold):
    current_sum = 0
    sorted_group = group.sort_values('count',
    ↪  ascending=False).reset_index(drop=True)
    selected_rows = []

    for _, row in sorted_group.iterrows():
        if current_sum + row['count'] <= threshold:
            selected_rows.append(row)
            current_sum += row['count']
        elif current_sum < threshold:
            # If adding the full count would exceed the
            ↪  threshold,
            # we create a new row with the remaining count
            ↪  needed
            # This results in sometimes taking a few entries
            ↪  over 10000
            remaining = threshold - current_sum
            new_row = row.copy()
            new_row['count'] = remaining
            selected_rows.append(new_row)
            current_sum = threshold

        if current_sum == threshold:
            break
```

```
    return pd.DataFrame(selected_rows)

# Process each group
result_list = []
for name, group in grouped:
    if name == 'O':
        threshold = 3601
    else:
        threshold = 10000

    result_list.append(select_entries(group, threshold))

# Combine all results to a single df
result = pd.concat(result_list, ignore_index=True)

# Save the result
result.to_csv('MSS_final_filtered_camera_data4.csv',
↪  index=False)
```

A.5.2 Selecting randomly

```
import pandas as pd

count_per_entry_sorted =
↪  pd.read_csv('unique_fits_entries_with_counts_sorted.csv')
count_per_entry_sorted =
↪  count_per_entry_sorted.sample(frac=1).reset_index(drop=True)

target_count = 42200
sampled_entries = []
current_count = 0

for _, row in count_per_entry_sorted.iterrows():
    if current_count >= target_count:
        break
    entry_count = row['count']
    sampled_entries.append(row)
    current_count += entry_count

# Convert sampled entries to DataFrame
sampled_entries_df = pd.DataFrame(sampled_entries)
sampled_entries_df = sampled_entries_df[['rerun', 'run',
↪  'camcol', 'field']].astype(int)
```

```
sampled_entries_df.to_csv('sampled_frame_entries_with_63700_stars.csv',
↪  index=False)
```

A.6 Converting individual camera frame entries to their star entries

```
import pandas as pd

# Read the result CSV (the one with the entries we want to
↪  match)
df_result = pd.read_csv('MSS_final_filtered_camera_data4.csv')

# Read the larger CSV file that we want to filter
df_large =
↪  pd.read_csv('MSS_with_duplicates_and_NULL_removed_afarrag.csv')

# Create a set of unique combinations from the result CSV
unique_combinations = set(df_result[['rerun', 'run', 'camcol',
↪  'field', 'mainclass']].itertuples(index=False, name=None))

# Filter the larger DataFrame based on these combinations
filtered_df = df_large[df_large.set_index(['rerun', 'run',
↪  'camcol', 'field',
↪  'mainclass']).index.isin(unique_combinations)]

# Save the filtered result to a new CSV file
filtered_df.to_csv('MSS_final_filtered_camera_data5.csv',
↪  index=False)
```

A.7 Creating training, validation and testing sets

```
import pandas as pd
import numpy as np

# Load the data
df = pd.read_csv('MSS_final_filtered_camera_data5.csv')

# Function to split the data for each mainclass
def split_data(df, train_ratio=0.7, validation_ratio=0.1,
↪  test_ratio=0.2):
    train_list = []
    validation_list = []
```

```
    test_list = []

    for mainclass in df['mainclass'].unique():
        class_df = df[df['mainclass'] == mainclass]
        class_df = class_df.sample(frac=1,
        ↪  random_state=42).reset_index(drop=True)  # Shuffle
        ↪  the data

        train_end = int(train_ratio * len(class_df))
        validation_end = train_end + int(validation_ratio *
        ↪  len(class_df))

        train_list.append(class_df[:train_end])

        ↪  validation_list.append(class_df[train_end:validation_end])
        test_list.append(class_df[validation_end:])

    train_df = pd.concat(train_list).reset_index(drop=True)
    validation_df =
    ↪  pd.concat(validation_list).reset_index(drop=True)
    test_df = pd.concat(test_list).reset_index(drop=True)

    return train_df, validation_df, test_df

# Split the data
train_df, validation_df, test_df = split_data(df)

# Save to CSV files
train_df.to_csv('MSS_training_entries.csv', index=False)
validation_df.to_csv('MSS_validation_entries.csv',
↪  index=False)
test_df.to_csv('MSS_testing_entries.csv', index=False)
```

A.8 Getting unique camera frame entries for FITS downloading

```
import pandas as pd

# Load the CSV file into a DataFrame
df = pd.read_csv('MSS_training_entries.csv')

# Select the relevant columns and drop duplicates to get
↪  unique combinations
```

```
unique_combinations = df[['run', 'rerun', 'camcol',
↪  'field']].drop_duplicates()
unique_combinations.to_csv('MSS_training_unique_cam_entries.csv',
↪  index=False)
```

Appendix B

Imaging Data Collection

This appendix contains the code required to retrieve and generate the imaging data.

B.1 RGB image downloads

```
import pandas as pd
import requests
import os
from concurrent.futures import ThreadPoolExecutor,
↪  as_completed
import time

# Read CSV file into DataFrame
data = pd.read_csv('MSS_training_entries.csv', delimiter=',')

# Read ra and dec celestial coordinates as floats
data['ra'] = data['ra'].astype(float)
data['dec'] = data['dec'].astype(float)

os.makedirs('Training_RGB', exist_ok=True)  # Create the
↪  directory if it doesn't exist

log_file = 'Training_RGB_downloading.txt' # Log file for
↪  downloading, avoids out of memory errors in browser

# Function to download images and classify
def download_image(ra, dec, mainclass, idx):
    url =
    ↪  f"http://skyserver.sdss.org/dr18/SkyServerWS/ImgCutout/getjpeg?
    ↪  ra={ra}&dec={dec}&scale=0.025&width=224&height=224"
```

```
        class_dir = os.path.join('Training_RGB', mainclass)  #
        ↪  Create a subdirectory for each star class
        os.makedirs(class_dir, exist_ok=True)  # Create the
        ↪  subdirectory if it doesn't exist
        image_path = os.path.join(class_dir,
        ↪  f'{mainclass}_class_image_ra_{ra}_dec_{dec}_rgb_image.png')

        with open(log_file, 'a') as log:
            if not os.path.exists(image_path): # Check if the
            ↪  image already exists
                response = requests.get(url)
                with open(image_path, 'wb') as f:
                    f.write(response.content)
                log.write(f"Downloaded image {idx} to
                ↪  {image_path}\n")
            else:
                log.write(f"Image {idx} already exists at
                ↪  {image_path}, skipping download.\n")

        # Print time after every 10000 downloads
        if idx > 0 and idx % 10000 == 0:
            print(f"{time.strftime('%Y-%m-%d %H:%M:%S')} -
            ↪  Downloaded {idx} images")

    # Function to handle parallel downloading
    def parallel_download(data):
        with ThreadPoolExecutor(max_workers=16) as executor:
            futures = []
            for idx, row in data.iterrows():
                future = executor.submit(download_image,
                ↪  row['ra'], row['dec'], row['mainclass'], idx)
                futures.append(future)

            for future in as_completed(futures):
                try:
                    future.result()
                except Exception as e:
                    with open(log_file, 'a') as log:
                        log.write(f"An error occurred: {e}\n")

    print(f"Started: {time.strftime('%Y-%m-%d %H:%M:%S')}")
    parallel_download(data) # For Training
```

B.2 FITS downloading

```
import os
import pandas as pd
import concurrent.futures
from tqdm import tqdm # Loading bar

def download_file(url, save_dir):
    os.makedirs(save_dir, exist_ok=True)
    command = f"wget -nc -P {save_dir} {url}" # -nc to skip
    ↪  already downloaded files
    os.system(command)

def generate_urls_and_dirs(df):
    for i in range(len(df)):
        rerun = int(df['rerun'][i])
        run1 = int(df['run'][i])
        run2 = str(int(run1)).zfill(6)
        camcol = int(df['camcol'][i])
        field = str(int(df['field'][i])).zfill(4)

        for band in ['u', 'g', 'r', 'i', 'z']:
            image_url =
            ↪  f"https://dr18.sdss.org/sas/dr18/prior-surveys/
            ↪  sdss4-dr17-eboss/photoObj/frames/{rerun}/{run1}/
            ↪  {camcol}/frame-{band}-{run2}-{camcol}-{field}.fits.bz2"
            image_save_dir =
            ↪  f"training_fits_images_{band}_band_bz2/"
            yield (image_url, image_save_dir)

sampled_unique_fits_entries =
↪  pd.read_csv('MSS_training_unique_cam_entries.csv') # Read
↪  unique fits image values into DataFrame
total_downloads = len(sampled_unique_fits_entries) * 5  # 5
↪  image bands

# Use ThreadPoolExecutor to perform parallel downloads
with concurrent.futures.ThreadPoolExecutor(max_workers=16) as
↪  executor:
    futures = []
    for url, save_dir in
    ↪  generate_urls_and_dirs(sampled_unique_fits_entries):
        futures.append(executor.submit(download_file, url,
        ↪  save_dir))
```

```
    for _ in tqdm(concurrent.futures.as_completed(futures),
    ↪  total=total_downloads, desc="Downloading files"): #
    ↪  progress bar
        pass
```

B.3 Decompressing bz2

```
import os
import bz2
import gzip
import shutil
from concurrent.futures import ThreadPoolExecutor
from tqdm import tqdm

def unpack_file(filepath, dest_dir):
    filename = os.path.basename(filepath)
    if filepath.endswith('.bz2'):
        newpath = os.path.join(dest_dir, filename[:-4])  #
        ↪  Remove .bz2 extension
        try:
            with bz2.open(filepath, 'rb') as source,
            ↪  open(newpath, 'wb') as dest:
                shutil.copyfileobj(source, dest)
            os.remove(filepath)  # Remove the original
            ↪  compressed file
        except EOFError: # Catch the EOFError
            print(f"Corrupted file detected: {filepath}.") #
            ↪  Print corrupted filename
            os.remove(filepath) # Remove the corrupted file
    elif filepath.endswith('.gz'): # For catalogue files
        newpath = os.path.join(dest_dir, filename[:-3])  #
        ↪  Remove .gz extension
        try:
            with gzip.open(filepath, 'rb') as source,
            ↪  open(newpath, 'wb') as dest:
                shutil.copyfileobj(source, dest)
            os.remove(filepath)  # Remove the original
            ↪  compressed file
        except EOFError: # Catch the EOFError
            print(f"Corrupted file detected: {filepath}.")
            ↪  Print corrupted filename
            os.remove(filepath) # Remove the corrupted file
```

```
    else:
        print(f"Unsupported file format: {filepath}") #
        ↪ Windows file system sometimes puts hidden files in
        ↪ the directory

# Get all files from a directory that have to be unpacked
def unpack_directory(directory, dest_dir):
    for root, _, files in os.walk(directory):
        for file in files:
            if file.endswith(('.bz2', '.gz')):
                yield os.path.join(root, file), dest_dir

# List of directories to process
directories = [
    ("validation_fits_images_u_band_bz2/",
    ↪ "validation_unpacked_fits_images_u_band/"),
    ("validation_fits_images_g_band_bz2/",
    ↪ "validation_unpacked_fits_images_g_band/"),
    ("validation_fits_images_r_band_bz2/",
    ↪ "validation_unpacked_fits_images_r_band/"),
    ("validation_fits_images_i_band_bz2/",
    ↪ "validation_unpacked_fits_images_i_band/"),
    ("validation_fits_images_z_band_bz2/",
    ↪ "validation_unpacked_fits_images_z_band/")
]

files_to_unpack = []
for src_dir, dest_dir in directories:
    os.makedirs(dest_dir, exist_ok=True)  # Create destination
    ↪ directory if it doesn't exist
    files_to_unpack.extend(unpack_directory(src_dir,
    ↪ dest_dir))

# Use ThreadPoolExecutor for parallel processing
with ThreadPoolExecutor(max_workers=16) as executor:
    list(tqdm(executor.map(lambda x: unpack_file(*x),
    ↪ files_to_unpack), total=len(files_to_unpack),
    ↪ desc="Unpacking files")) # *x unpacks the tuple into 2
    ↪ arguments
print("All files have been unpacked.")
```

B.4 Cropping stars out of full frame FITS images

```
import pandas as pd
from astropy.io import fits
from astropy.wcs import WCS
from astropy.nddata import Cutout2D
from astropy.coordinates import SkyCoord
import astropy.units as u
import os
import glob
from astropy.time import Time
from datetime import datetime
import warnings
from astropy.utils.exceptions import AstropyWarning

# There are deprecated columns in the FITS file headers.
↪  They're fixed automatically but print warnings, so I
↪  disabled the printing
warnings.filterwarnings('ignore', category=AstropyWarning,
↪  append=True)

os.makedirs("training_cropped_images 64x64", exist_ok=True)

df = pd.read_csv('MSS_training_entries.csv')
df['run'] = df['run'].astype(int).astype(str).apply(lambda x:
↪  x.zfill(6)) # Conversion required for filename
print(df)
df['field'] = df['field'].astype(int).astype(str).apply(lambda
↪  x: x.zfill(4)) # Conversion required for filename
df['camcol'] = df['camcol'].astype(int)
bands = ['u', 'g', 'r', 'i', 'z']

# Iterate through each star in the CSV
for _, star in df.iterrows():
    for band in bands:
        # Construct the filename pattern
        fits_file =
        ↪  f"training_unpacked_fits_images_{band}_band/
        ↪  frame-{band}-{star['run']}-{star['camcol']}-{star['field']}.fits"

        if os.path.exists(fits_file):
            try:
                with fits.open(fits_file) as hdul:
                    image_data = hdul[0].data
```

```
                    header = hdul[0].header
                    wcs = WCS(header) # Get WCS information
                    ↪ from the FITS header
                    ↪ https://docs.astropy.org/en/stable/wcs/
                    star_coord =
                    ↪ SkyCoord(ra=star['ra']*u.degree,
                    ↪ dec=star['dec']*u.degree) # SkyCoord
                    ↪ object for the star
                    cutout = Cutout2D(image_data, star_coord,
                    ↪ (64, 64), wcs=wcs) # Make the crop

                    # Create a new FITS Header Data Unit with
                    ↪ the cropped data
                    new_hdu = fits.PrimaryHDU(cutout.data)

                    ↪ new_hdu.header.update(cutout.wcs.to_header())
                    new_filename = f"training_cropped_images
                    ↪ 64x64/{star['mainclass']}_class_cutout
                    ↪ _specobjid_{star['specobjid']}_
                    ↪ ra_{star['ra']}_dec_{star['dec']}_band_{band}.fits"
                    new_hdu.writeto(new_filename,
                    ↪ overwrite=True) # Save the new FITS
                    ↪ file
                except Exception as e:
                    print(f"Error processing {fits_file}: {e}")
                    continue
            else:
                print(fits_file)
                print(f"No matching file found for star
                ↪ {star['specobjid']}, ra {star['ra']}, dec
                ↪ {star['dec']}, run {star['run']}, field
                ↪ {star['field']}, camcol {star['camcol']} in
                ↪ band {band}")
```

B.5 Synthesizing RGB images from individual band images

```
from tqdm import tqdm
import os
import numpy as np
from astropy.visualization import make_lupton_rgb
from astropy.io import fits
import matplotlib.pyplot as plt
```

```
from skimage.transform import resize
import gc
from concurrent.futures import ProcessPoolExecutor,
↪  as_completed

def normalize_image(image):
    scale_min, scale_max = np.percentile(image, [1, 99])
    scaled = np.clip((image - scale_min) / (scale_max -
    ↪  scale_min), 0, 1)
    return scaled

def combine_to_rgb(g_or_u_image, r_image, i_or_z_image, Q,
↪  alpha):
    # We resize to avoid images on edges of the full frames
    ↪  being cropped into smaller than 64x64 images
    g_or_u = normalize_image(resize(g_or_u_image, (64, 64),
    ↪  preserve_range=True, anti_aliasing=True))
    r = normalize_image(resize(r_image, (64, 64),
    ↪  preserve_range=True, anti_aliasing=True))
    i_or_z = normalize_image(resize(i_or_z_image, (64, 64),
    ↪  preserve_range=True, anti_aliasing=True))
    return make_lupton_rgb(g_or_u, r, i_or_z, Q=Q,
    ↪  stretch=alpha) # Expects float, see
    ↪  https://docs.astropy.org/en/stable/visualization/
    ↪  rgb.html#astropy-visualization-rgb



def process_fits_file(filename):
    with fits.open(filename, memmap=True) as hdul: # allows
    ↪  the array data of each HDU to be accessed with mmap,
    ↪  rather than being read into memory all at once
    ↪  https://docs.astropy.org/en/stable/io/fits/
        return hdul[0].data.copy()

def process_image(prefix, input_directory, output_directory,
↪  band_set, Q, alpha):
    bands = {}
    for band in band_set:
        filename = os.path.join(input_directory,
        ↪  f"{prefix}_band_{band}.fits")
        if os.path.exists(filename):
            bands[band] = process_fits_file(filename)
```

```
    if all(band in bands for band in band_set):
        band1 = bands[band_set[0]]
        band2 = bands[band_set[1]]
        band3 = bands[band_set[2]]
        rgb_image = combine_to_rgb(band1, band2, band3, Q,
        ↪  alpha)
        plt.imsave(os.path.join(output_directory,
        ↪  f'{prefix}_{band_set}_RGB.png'), rgb_image,
        ↪  format='png', dpi=300)
        return True
    return False

# Process the images in batches to avoid crashing due to
↪  insufficient memory
def process_batch(batch, Q, alpha):
    results = []
    for item in batch:
        prefix, input_directory, gri_output, urz_output, Q,
        ↪  alpha = item
        gri_result = process_image(prefix, input_directory,
        ↪  gri_output, 'gri', Q, alpha)
        urz_result = process_image(prefix, input_directory,
        ↪  urz_output, 'urz', Q, alpha)
        results.append((gri_result, urz_result))
    return results

def process_directory(input_directory, gri_output_directory,
↪  urz_output_directory, Q, alpha):
    os.makedirs(gri_output_directory, exist_ok=True)
    os.makedirs(urz_output_directory, exist_ok=True)

    prefixes = set()
    for filename in os.listdir(input_directory):
        if filename.endswith('.fits'):
            prefix = filename.rsplit('_band_', 1)[0] #
            ↪  maxsplit parameter to 1, will return a list
            ↪  with 2 elements
            ↪  https://www.w3schools.com/python/ref_string_rsplit.asp
            prefixes.add(prefix)

    batch_size = 50
    batches = [list(prefixes)[i:i+batch_size] for i in
    ↪  range(0, len(prefixes), batch_size)]
```

```
    with ProcessPoolExecutor(max_workers=16) as executor:
        futures = []
        for batch in batches:
            batch_items = [(prefix, input_directory,
            ↪  gri_output_directory, urz_output_directory, Q,
            ↪  alpha) for prefix in batch]
            futures.append(executor.submit(process_batch,
            ↪  batch_items, Q, alpha))

        with tqdm(total=len(prefixes) * 2, desc="Processing
        ↪  images", unit="step") as pbar:
            for future in as_completed(futures):
                results = future.result()
                for gri_result, urz_result in results:
                    if gri_result:
                        pbar.update(1)
                    if urz_result:
                        pbar.update(1)

process_directory('training_cropped_images 64x64',
↪  'training_gri_q_8_a_0.02 64x64', 'training_urz_q_8_a_0.02
↪  64x64', 8, 0.02)
process_directory('validation_cropped_images 64x64',
↪  'validation_gri_q_8_a_0.02 64x64',
↪  'validation_urz_q_8_a_0.02 64x64', 8, 0.02)
process_directory('testing_cropped_images 64x64',
↪  'testing_gri_q_8_a_0.02 64x64', 'testing_urz_q_8_a_0.02
↪  64x64', 8, 0.02)
```

B.6 Sorting images into mainclass folders

```
import os
import shutil

def sort_files_by_class(directory):
    for filename in os.listdir(directory):
        file_path = os.path.join(directory, filename)

        if not os.path.isfile(file_path):
            continue

        star_class = filename[0].upper()
        subdirectory_path = os.path.join(directory,
        ↪  star_class)
```

```
        os.makedirs(subdirectory_path, exist_ok=True)
        shutil.move(file_path, os.path.join(subdirectory_path,
        ↪  filename))


sort_files_by_class("training_gri_q_8_a_0.02 64x64")
sort_files_by_class("training_urz_q_8_a_0.02 64x64")
```

B.7 Side by side GRI-URZ images

```
import os
from PIL import Image

# Define directories
gri_dir = 'training_gri_q_8_a_0.02 64x64'
urz_dir = 'training_urz_q_8_a_0.02 64x64'
output_dir = 'training_side_by_side_q_8_a_0.02'
os.makedirs(output_dir, exist_ok=True)

def get_image_pairs(gri_dir, urz_dir, class_label):
    gri_class_dir = os.path.join(gri_dir, class_label)
    urz_class_dir = os.path.join(urz_dir, class_label)

    gri_images = [f for f in os.listdir(gri_class_dir) if
    ↪  f.endswith('.png')]
    urz_images = [f for f in os.listdir(urz_class_dir) if
    ↪  f.endswith('.png')]
    image_pairs = list(zip(gri_images, urz_images))  # Create
    ↪  image pairs

    return image_pairs, gri_class_dir, urz_class_dir

for class_label in ['O', 'B', 'A', 'F', 'G', 'K', 'M']:
    output_class_dir = os.path.join(output_dir, class_label)
    os.makedirs(output_class_dir, exist_ok=True)
    image_pairs, gri_class_dir, urz_class_dir =
    ↪  get_image_pairs(gri_dir, urz_dir, class_label)

    for gri_image_name, urz_image_name in image_pairs:
        gri_image_path = os.path.join(gri_class_dir,
        ↪  gri_image_name)
```

```
        urz_image_path = os.path.join(urz_class_dir,
        ↪  urz_image_name)
        gri_image = Image.open(gri_image_path)
        urz_image = Image.open(urz_image_path)

        # Combine the images side by side
        combined_img = Image.new('RGB', (128, 64))
        combined_img.paste(gri_image, (0, 0))
        combined_img.paste(urz_image, (64, 0))

        base_name =
        ↪  os.path.basename(gri_image_path).replace('_gri_RGB.png',
        ↪  '_side_by_side.png')
        output_path = os.path.join(output_class_dir,
        ↪  base_name)
        combined_img.save(output_path)

        print(f'Saved {output_path}')
```

B.8 Upscaling images

```
from PIL import Image
import os
import time


# https://pillow.readthedocs.io/en/stable/
↪  handbook/concepts.html#filters-comparison-table
def upscale_and_save_image(image_path, output_path):
    try:
        if os.path.exists(output_path): # Skip if image exists
            return

        with Image.open(image_path) as image:
            upscaled_image = image.resize((224, 224),
            ↪  Image.LANCZOS) # Resize image to 224x224
            upscaled_image.save(output_path)

    except Exception as e:
        print(f"Error processing {image_path}: {e}")

input_dir = 'training_gri_q_8_a_0.02 64x64'
```

```
output_dir = 'training_gri_q_8_a_0.02 224x224 from 64x64'

counter = 0
for root, dirs, files in os.walk(input_dir):
    for file in files:
        input_path = os.path.join(root, file)
        os.makedirs(output_dir, exist_ok=True)
        output_path = os.path.join(output_dir, file) # output
        ↪  path

        try:
            upscale_and_save_image(input_path, output_path)
            counter += 1
            if counter % 10000 == 0:
                print(f"{time.strftime('%Y-%m-%d %H:%M:%S')} -
                ↪  Upscaled {counter} images")

        except Exception as e:
            print(f"Error processing {input_path}: {e}")

print(f"All images have been upscaled and saved to
↪  {output_dir}")
```

Appendix C

ViT

This appendix contains the code to set up the ViT.

C.1 Preparing and saving DatasetDict object for ViT

```
from datasets import load_dataset, DatasetDict

# Load the training, validation and testing sets
train_dataset = load_dataset("imagefolder",
↪  data_dir="Training_RGB")
validation_dataset = load_dataset("imagefolder",
↪  data_dir="Validation_RGB")
test_dataset = load_dataset("imagefolder",
↪  data_dir="Testing_RGB")

# Create a DatasetDict directly with the train, validation and
↪  testing datasets.
dataset_dict = DatasetDict({
    'train': train_dataset['train'],
    'validation': validation_dataset['train'],
    'test': test_dataset['train']
})

dataset_dict.save_to_disk('dataset_dictionary_RGB')
```

C.2 Loading DatasetDict object

```
dataset_dict = load_from_disk("dataset_dictionary_RGB")
```

C.3 Importing ViT

```
from transformers import ViTImageProcessor
model_name_or_path = 'google/vit-base-patch16-224-in21k'
processor =
↪  ViTImageProcessor.from_pretrained(model_name_or_path)
```

C.4 Transforming images to ViT inputs

```
def transform(example_batch):
    # Take a list of PIL images and turn them to pixel values
    rgb_images = [img for img in example_batch['image']] # Use
    ↪  this for native RGB images
    # rgb_images = [img.convert('RGB') for img in
    ↪  example_batch['image']] # Use this for Synthesized RGB
    ↪  images
    inputs = processor(rgb_images, return_tensors='pt')

    # Don't forget to include the labels!
    inputs['labels'] = example_batch['label']
    return inputs

prepared_ds = dataset_dict.with_transform(transform)
```

C.5 Data Collator

```
import torch

def collate_fn(batch):
    return {
        'pixel_values': torch.stack([x['pixel_values'] for x
        ↪  in batch]),
        'labels': torch.tensor([x['labels'] for x in batch])
    }
```

C.6 Evaluation Metrics

```
import numpy as np
from datasets import load_metric

accuracy_metric = load_metric("accuracy")
precision_metric = load_metric("precision")
recall_metric = load_metric("recall")
```

```
f1_metric = load_metric("f1")


def compute_metrics(p):
    preds = np.argmax(p.predictions, axis=1) # Get the
    ↪  predicted class
    labels = p.label_ids # Labels

    # Calculate accuracy, precision, recall, f1 (with
    ↪  average='weighted' for multi-class and to account for
    ↪  label imbalance)
    accuracy = accuracy_metric.compute(predictions=preds,
    ↪  references=labels)
    precision = precision_metric.compute(predictions=preds,
    ↪  references=labels, average='weighted')
    recall = recall_metric.compute(predictions=preds,
    ↪  references=labels, average='weighted')
    f1 = f1_metric.compute(predictions=preds,
    ↪  references=labels, average='weighted')

    # Return dictionary of metrics
    return {
        "accuracy": accuracy["accuracy"],
        "precision": precision["precision"],
        "recall": recall["recall"],
        "f1": f1["f1"]
    }
```

C.7 Classification Head

```
from transformers import ViTForImageClassification

labels = dataset_dict['train'].features['label'].names

model = ViTForImageClassification.from_pretrained(
    model_name_or_path,
    num_labels=len(labels),
    id2label={str(i): c for i, c in enumerate(labels)},
    label2id={c: str(i) for i, c in enumerate(labels)}
)
```

C.8 Training arguments

```
from transformers import TrainingArguments

training_args = TrainingArguments(
  output_dir="./vit-stellar-classification-RGB",
  per_device_train_batch_size=32,
  eval_strategy="steps",
  num_train_epochs=50,
  fp16=True,
  save_steps=1000,
  eval_steps=1000,
  logging_steps=1000,
  learning_rate=1e-5,
  save_total_limit=10,
  metric_for_best_model="accuracy",
  remove_unused_columns=False,
  push_to_hub=False,
  report_to='tensorboard',
  load_best_model_at_end=True,
)
```

C.9 Early stopping

```
from transformers import EarlyStoppingCallback

early_stopping_callback = EarlyStoppingCallback(
    early_stopping_patience=5,
    early_stopping_threshold=0.001
)
```

C.10 Trainer

```
from transformers import Trainer

trainer = Trainer(
    model=model,
    args=training_args,
    data_collator=collate_fn,
    compute_metrics=compute_metrics,
    train_dataset=prepared_ds["train"],
    eval_dataset=prepared_ds["validation"],
    tokenizer=processor,
```

```
    callbacks=[early_stopping_callback]
)
```

C.11 Start training

```
train_results = trainer.train()
trainer.save_model()
trainer.log_metrics("train", train_results.metrics)
trainer.save_metrics("train", train_results.metrics)
trainer.save_state()
```

C.12 Evaluate model using test set

```
test_results = trainer.predict(prepared_ds["test"])
trainer.log_metrics("test", test_results.metrics)
trainer.save_metrics("test", test_results.metrics)
```

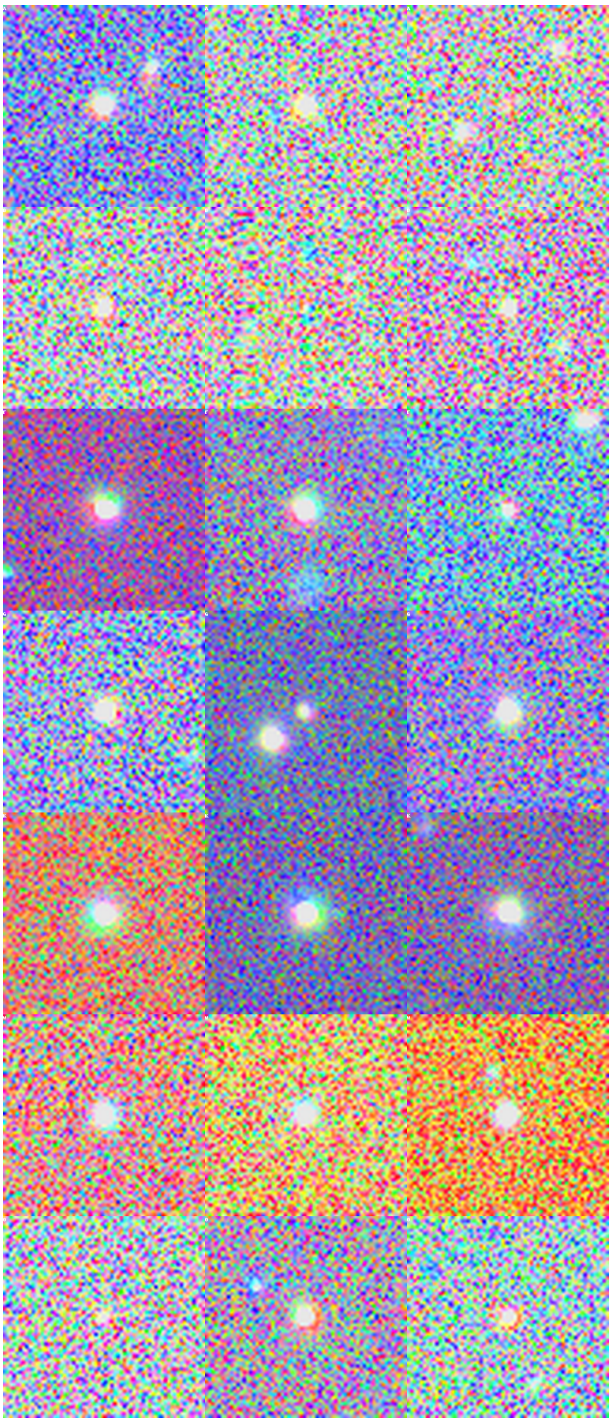
Appendix D

Images

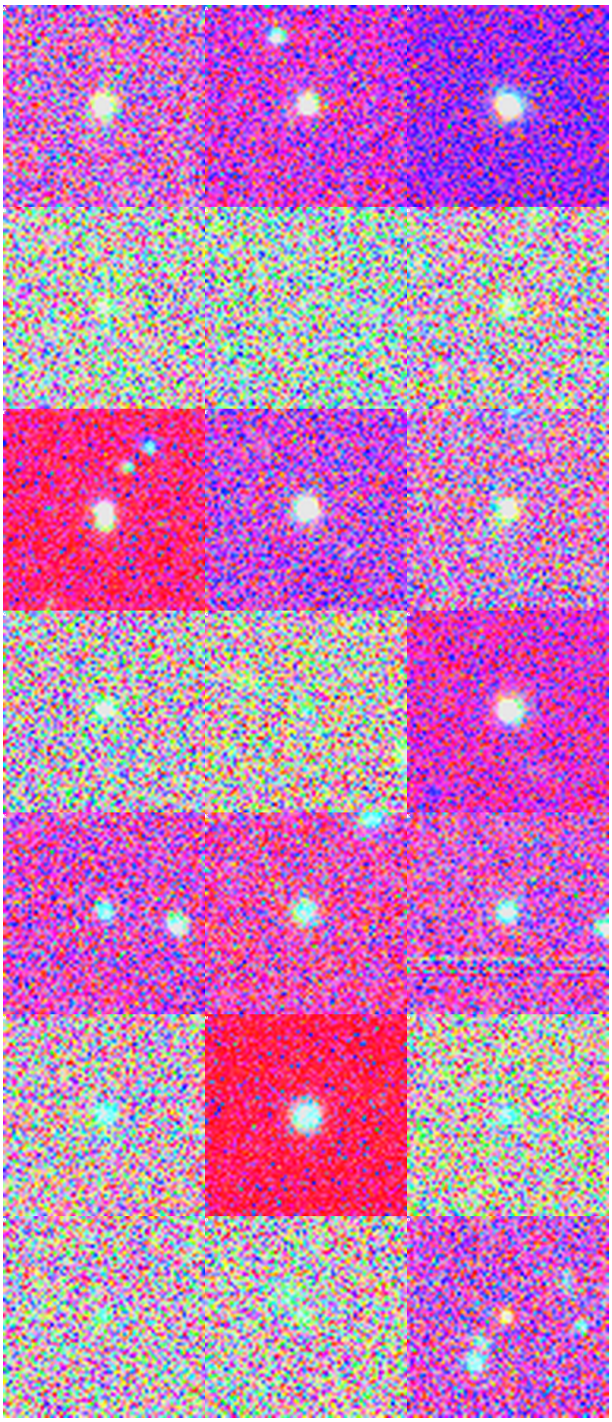
This appendix contains example images that are used. In each section there are 3 images per row. Each row is a star class. The stars appear in the class order A, B, F, G, K, M, O

D.1 RGB images

D.2 GRI images



D.3 URZ images



D.4 GRI-URZ side-by-side images