Technische Universität Berlin

Architechture of Machine Learning Systems
-
Galaxy and Star Classification Report

Tobias Labarta: 467696

Niccolò Parodi: 473549

Felix Zailskas: 487751

22.07.2023

Contents

1 Data Acquisition and Alignment

Task: Obtain the dataset mentioned above, extract the following input data for your ML pipeline, and materialize these inputs as files. Read the description of the dataset carefully, and see the SDSS website for additional details. The different spectral bands require alignment. You may use gray-scale encoding of each spectral band (combined with the IRG images), and work with small images or patches to simplify their handling. Subsequently, compute meaningful summary statistics and visualization (e.g., comparing aligned and unaligned images).

1.1 Dataset

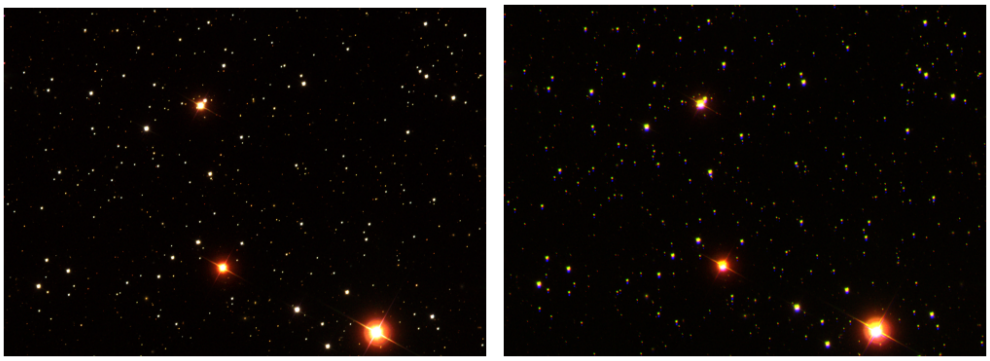
For this task, we used data publicly available at the Sloan Digital Sky Survey (SDSS) (`https://skyserver.sdss.org/dr1/en/proj/advanced/color/sdssfilters.asp`). This data set are night sky images containing stars and galaxies. Each image comprises five channels: i, r, g, u, and z. These are stored in five separate *.fits* files. Additionally, for each image, two further files were available, one containing the coordinates of stars visible in the image and one of visible galaxies. Both these files, called *calibration objects*, were also stored as *.fits* files. So in total, seven separate files were available for each sky survey image.

To download the files, we used a shell script (`./src/data_acquisition/downloader.sh`) with *wget*, specifying the desired files and URL. For file management, we added the directory *data* to our repository, with sub-directories for each image, containing all files related to them.

1.2 Aligning Image Channels.

We had to align the different bands to create a tensor with a channel for each spectral band. Although the images point at the same area in the sky, their positions are slightly different as they were taken one after one, and the earth rotated further. Hence, we need to shift the data in each channel file to align with their reference frame perfectly. To do this, we used the i-frame as a reference. We created a reference world coordinate system (WCS) based on the data stored in its header file. We could then compute the sky coordinate of the reference pixel in each of the other channels using their header files and then compute the pixel coordinate this would correspond to in the reference file. Using these pixel coordinates, we can compute the needed shift for each image and use a simple translation to perform this shift. During this process, we kept track of the maximum shifts, which we then used to cut off invalid regions of the images. A comparison between

the aligned and unaligned channels can be seen in Figure 3.1. We can see that in 1.1b, there is a slight offset around all light sources. Mainly the G channel (here depicted in the blue channel) is offset to the bottom. In 1.1a, we do not see these artifacts. Hence, we assume our alignment to be successful.

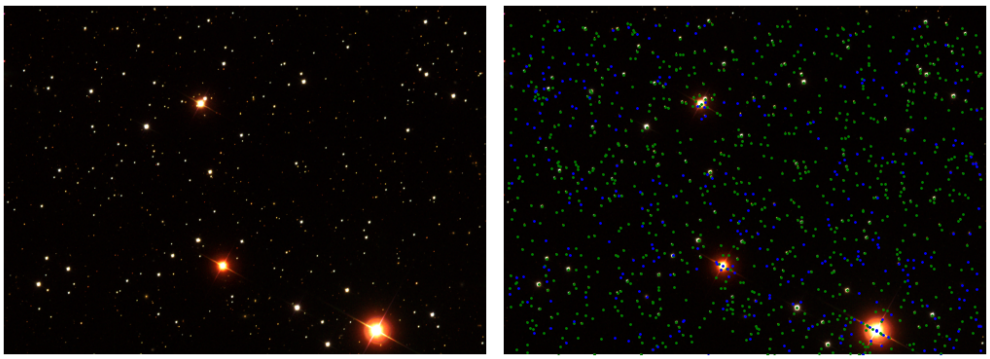


(a) Aligned IRG channels. (b) Unaligned IRG channels.

Figure 1.1: Comparison of aligned and unaligned IRG channels for image 008162-6-0080.

1.3 Computing Star and Galaxy Coordinates.

To get the coordinates of the stars and galaxies of each frame, we again used to i-frame as a reference frame. Using the reference WCS from this frame, we can translate the sky coordinates stored in the calibration objects to pixel coordinates. We limit these coordinates to the ones in the image and shift them by the value of the left and top cutoff resulting from the alignment. This gives us the coordinates of all valid stars and galaxies relative to the coordinates of the aligned image. The computed star and galaxy positions of one frame can be seen in Figure 1.2. After visual inspection, the positions of the stars and galaxies seem correct. We conclude that the extraction of star and galaxy positions was also successful.



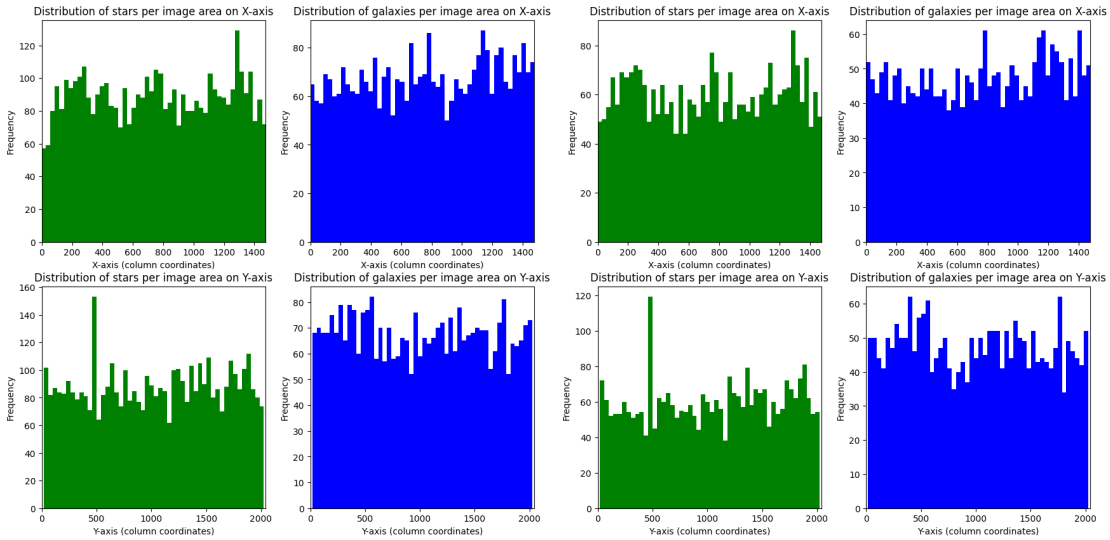
(a) Image without markers.

(b) Image with stars marked green and galaxies marked blue.

Figure 1.2: Calculated positions of stars and galaxies in image 008162-6-0080.

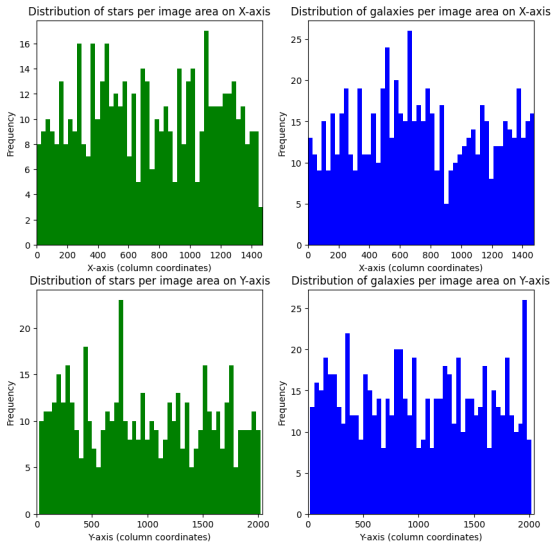
1.4 Statistics and Plots

Our full dataset consists of 12 images, which were used, and how they were split into training, validation, and test set can be seen in Table 1.1. The whole data set contains 4419 stars and 3377 galaxies. The training set contains 2994 (68%) stars and 2371 (70%) galaxies. The validation set contains 518 (12%) stars and 701 (21%) galaxies. The test set contains 907 (20%) stars and 305 (9%) galaxies.
Figure 1.3 shows how stars and galaxies are distributed in regard to pixel coordinates in the images for each dataset. Both stars and galaxies seem to be distributed uniformly across the images, with some spikes at random coordinates. The distribution for the training set behaves similarly. The spikes are more present in the validation and test set. However, there seems to be no particular pattern, so we will uniformly split the images into smaller patches in the following processing steps. The channels of the input images are not normalized. We computed channel means of 0.076, 0.0370, 0.0205, 0.0101, and 0.185 and channel standard deviations of 1.220, 1.001, 0.625, 0.849, and 4.376 for the i, r, g, u, and z channel respectively. These values will be used in further processing to normalize the images before training the network.

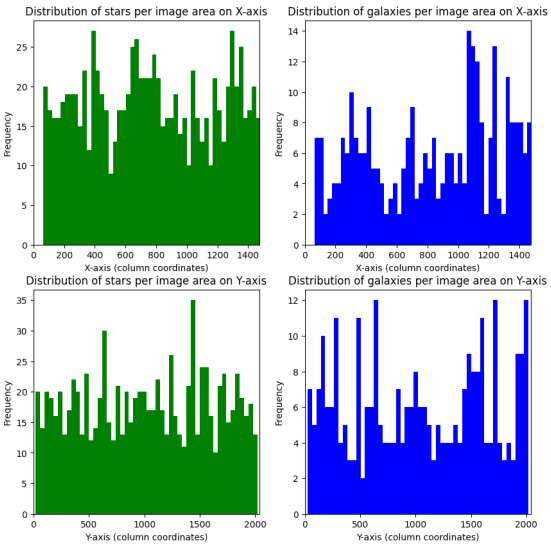


(a) Full dataset.

(b) Training set.



(c) Validation set.



(d) Test set.

Figure 1.3: Distribution of stars and galaxies with respect to the pixel coordinates.

Train	Validation	Test
1043/5/001043-5-0025	3918/3/003918-3-0213	8162/6/008162-6-0080
2188/2/002188-2-0037	7264/3/007264-3-0111	-
5405/6/005405-6-0099	-	-
1992/1/001992-1-0087	-	-
3184/2/003184-2-0169	-	-
4510/4/004510-4-0030	-	-
8115/2/008115-2-0181	-	-
2029/6/002029-6-0022	-	-
1035/6/001035-6-0017	-	-

Table 1.1: Images used for the train, validation, and test data sets. Images can be found by appending the listed link under `https://data.sdss.org/sas/dr17/eboss/photoObj/frames/301/`.

2 Data Preparation

Task: As a prerequisite for model training and testing, split the data into train/validation/test sets of similar data distributions (e.g., label distribution and potentially other properties such as visual similarity or data provenance). For the sake of comparison, everyone is required to use the spectral bands associated with *301/8162/6/frame-irg-008162-6-0080.jpg* as test data. Make sure that the test data is not leaking into the train and validation sets. Therefore, your code should the able to exclude data from specific sky coordinates when creating the train and validation sets. Again, consider working with smaller patches of the input (i.e., reduced H and W) in order to reduce memory requirements and ensure reasonable training times.

2.1 Patch Creation

As suggested in the assignment text, we decided to use smaller patches of the image as input to our ML model. We decided to use 64x64 pixel image patches. The following subsections describe the procedures used to create the patches and exclude certain sky coordinates.

2.1.1 Basic Patch Creation

To create the patch grid, we run over the image in 64 pixel size steps in the x and y directions. We then save the current correctly sized patch as a new image. Furthermore, we determine all star- and galaxy-coordinates that lie within that patch. These are saved along the patch, with their coordinates adjusted relative to the patch they lie in. For the images we used, this resulted in a 23x31 grid of image patches, giving us a total of 713 patches per image. The resulting patch of the image *008162-6-0080* can be seen in Figure 2.1. Note that patches created with a width or height of less than 64 were discarded in this step of the processing pipeline.

2.1.2 Sky Coordinate Exclusion

We solved the issue of possibly overlapping images by allowing the function that creates the patches to take in a *.fits* file which will be used to determine the sky coordinates of the are that should be excluded while creating patches. The first step in this process is to determine the minimal and maximal RA and DEC values contained in the image. As we do not know whether the image is aligned with the astronomical north, we cannot simply use the edges of the image. Instead, we compute the sky coordinates of every edge pixel of the image. We can calculate four corners in sky space from these coordinates that

enclose the coordinates we want to exclude. Using these sky coordinates of the exclusion area and the patch we are processing, we can determine whether they overlap. This is done by translating the sky coordinates into pixel coordinates using the WCS object of either image and checking if the rectangles described by the four corners overlap. If they do, we discard the patch for further processing.

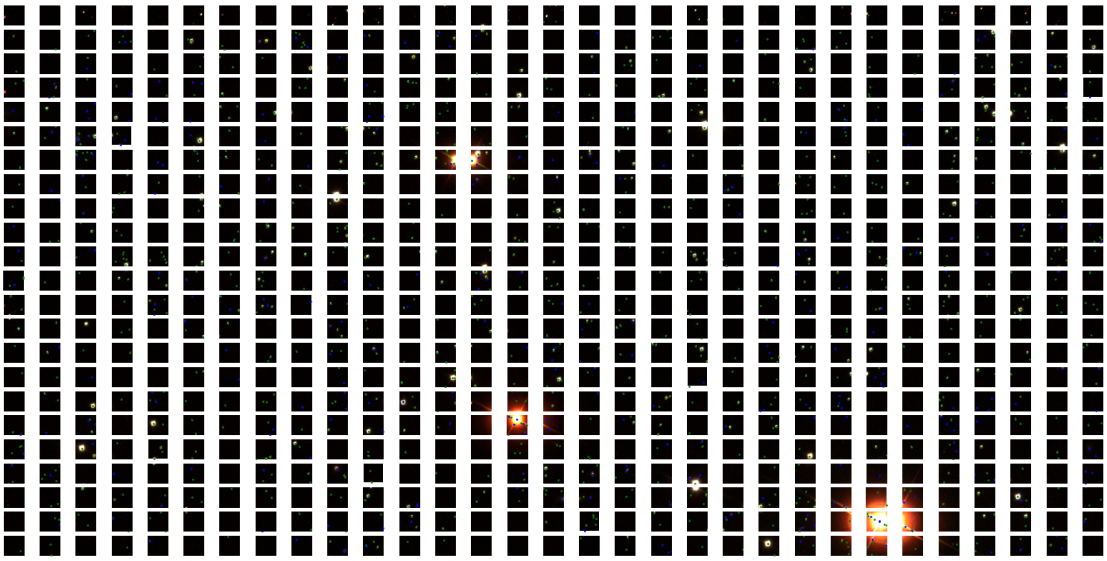


Figure 2.1: 713 patches, of size 64x64, resulting from the segmentation of image 008162-6-0080.

2.2 Data Set Split

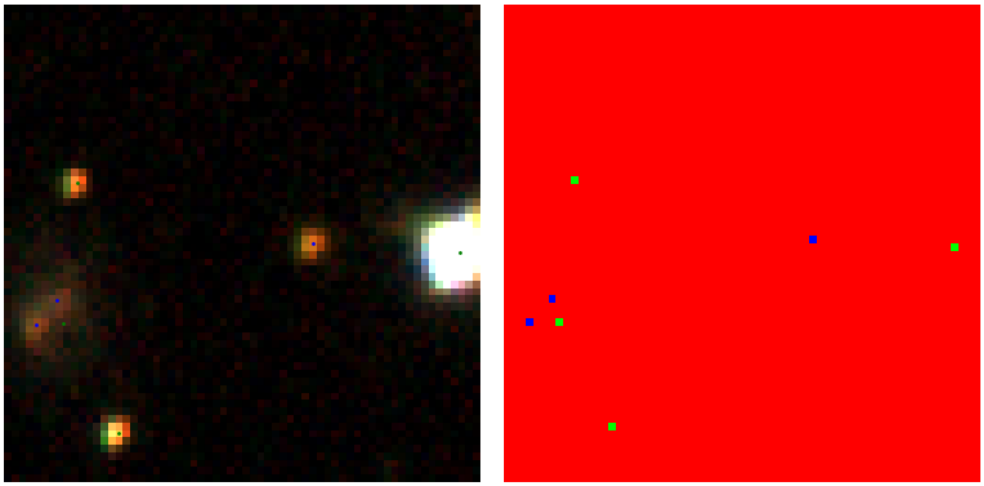
We used separate images to create the train, validation, and test set. Which images were used for which data subset can be seen in Table 1.1. Each image was first processed into patches as described in section 2.1. For both the train and validation set, all sky coordinates contained in the test set files were excluded using the procedure described in subsection 2.1.2. As the target for our neural network, we created a segmentation map for each image patch of the images. These segmentation maps are *numpy* arrays of shape (64x64x3) containing a three-dimensional vector for each pixel in the patch. Each dimension represents whether the pixel shows sky, a star, or a galaxy. Hence, these vectors can be one of three types:

1. $\begin{bmatrix}1\\0\\0\end{bmatrix}$ In this case the corresponding pixel shows sky.

2. $\begin{bmatrix}0\\1\\0\end{bmatrix}$ In this case the corresponding pixel shows a star.

3. $\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ In this case the corresponding pixel shows a galaxy.

Figure 3.5 shows a visualization of the segmentation map of one of the patches in image *008162-6-0080*. The segmentation map matches up with the raw patch that was processed. Hence, we assume that the creation of the segmentation maps was successful. The selected images resulted in 3565 patches for the training set, 1426 patches for the validation set, and 713 patches for the test set.



(a) Patch as IRG image with stars marked green and galaxies marked blue.

(b) Segmentation map with sky marked red, stars marked green, and galaxies marked blue.

Figure 2.2: Single patch from image 008162-6-0080 and the corresponding segmentation map.

2.3 Further Preprocessing

To ensure faster convergence during training, we will normalize the patches to a mean of 0 and a standard deviation of 1, using the channel means and standard deviations presented in section 1.4.

2.4 Data Loading and Augmentation

Data loading and augmentation are crucial steps in preparing our training and validation datasets for the star-galaxy classification task. These steps ensure that our data is appropriately processed and augmented to enhance our model's ability to learn and generalize from the available samples.

The provided code includes functions and a class that facilitate data loading and augmentation. By following a data-centric approach, we ensure that our datasets are constructed in a format suitable for training our star-galaxy classification model.

The `get_training_augmentation()` function defines a set of augmentations to be applied to our training images. These augmentations introduce variations in our data, such as random crops and flips, to create a more diverse training set. This diversity can help our model learn robust features and reduce the risk of overfitting to specific patterns.

The `BuildingsDataset` class plays a pivotal role in data loading and preprocessing. It encapsulates the functionality needed to load and process our image and segmentation mask pairs. The class leverages the capabilities of the PyTorch library, which provides utilities for handling custom datasets.

During initialization, our class reads the file paths of our images from the specified directory. This step establishes a connection between our code and the dataset by providing access to the necessary image data. Our class also accepts parameters such as the RGB values corresponding to different classes and the augmentation and preprocessing operations to be applied.

The `__getitem__()` method retrieves a specific item from our dataset. It loads the corresponding image and segmentation mask using a file path obtained from our dataset's file paths. The loaded data represents the raw input and the ground truth for our image classification task.

If augmentation is specified, our loaded image and mask undergo a series of transformations to augment our dataset. These transformations introduce random variations, such as flips or rotations, that enrich our available training samples. Augmentation helps us address the limited size of our original dataset and mitigate potential biases.

After augmentation, the preprocessing step is applied, if specified. Preprocessing involves operations like normalization or resizing, which ensure that our data is in a suitable format for training our star-galaxy classification model. Proper preprocessing aligns our data and sets the appropriate scale for consistent and effective learning.

3 Modeling and Tuning

Task: Construct an ML pipeline—using the prepared train and validation sets—for classifying specified co- ordinates as stars or galaxies. Choose an appropriate loss function and evaluation metric, and evaluate this metric on both the train and validation data. Example models to use include (but are not limited to) U-Net, SegNet, FastFCN, Gated-SCNN. Subsequently, tune the hyper-parameters of your model (e.g., regularization parameters). You may use different sizes of your model for faster exploration. To further simplify training, coordinates may also be mapped to instances of small patches (e.g., 5x5 pixel). Parallelize the hyper-parameter tuning and model training (e.g., by appropriate configurations). Finally, report the evaluation metric on the train, validation, and test data (again without leaking the test data into model training or hyper-parameter tuning).

Expected Results: Code for model training and running, as well as descriptions of the used model architecture, ML pipeline, and its evaluation. The report must include the statistics or plots of the quality of your model with and without tuning.

3.1 Model Architecture

As model architecture we decided to use the U-Net (`https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/`). It performs with a high accuracy and is commonly used for image segmentation tasks.

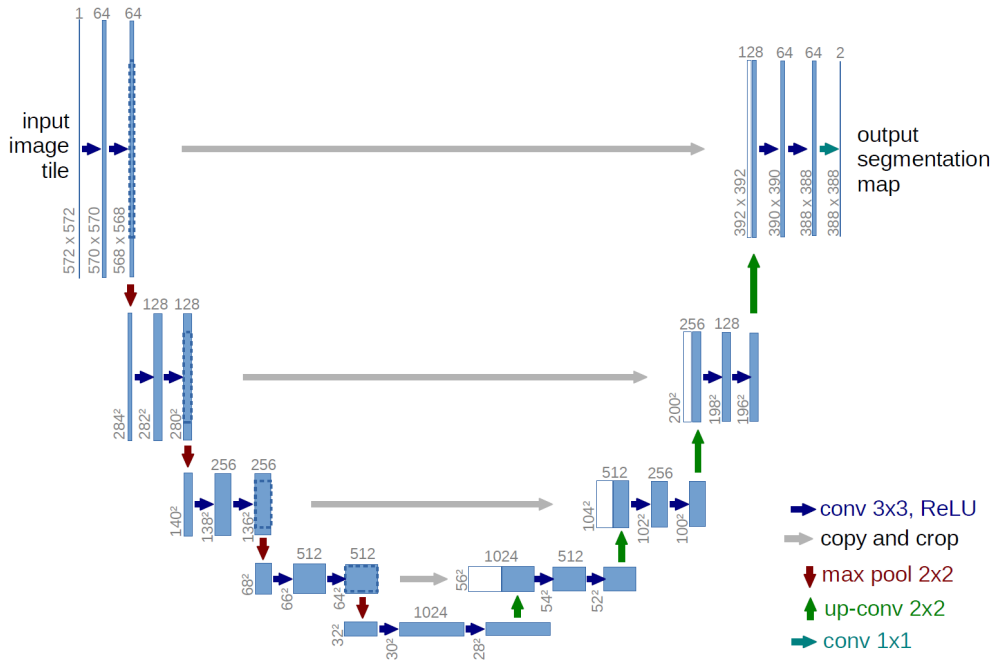


Figure 3.1: Architecture of U-Net Convolutional Neural Network (https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/)

We implemented the U-Net architecture with *torch* and *torchvision* libraries, using the built in *Conv2d* and *MaxPool2d* to build the layers. The following hyper-parameters were initially defined:

- **Kernel Size:** 3x3
- **Padding:** Same
- **Stride:** 1

After running a model summary, we found the total number of trainable parameters to be $31,380,227$ and the model size to be 141.69 MB.

We assumed this model size to be too computationally expensive for training on our personal devices and therefore decided to remove one layer of down- and upsampling.

The resulting reduced model has a total number of $7,784,195$ trainable parameters and a model size of 46.52 MB.

3.2 Model Training and Evaluation

The training pipeline consists of several key steps, including data loading, model initialization, hyperparameter definition, optimization setup, and the actual training loop. Additionally, we incorporate the use of the WandB library for logging and visualization purposes.

To ensure reproducibility, we set a random seed and configure the logging system. We then check whether a GPU is available and assign the appropriate device for model training, whether it be a GPU or CPU. This device information is logged for reference.

To regulate the training process, we define hyperparameters like batch size, epochs, and learning rate. The performance of the model can be improved by adjusting these hyperparameters.

We instantiate the U-Net model, in particular the `UNetWithGradCAM` variation. Due to its capacity to gather contextual data and enable accurate localization, the U-Net architecture is a popular option for picture segmentation tasks. The `UNetWithGradCAM` model adds gradient-based class activation map (GradCAM) visualization to the U-Net architecture. We subsequently send the model to the selected device, enabling effective computing.

The cross-entropy loss, commonly used for multi-class classification tasks, is the loss function that we utilize for training. During model training, we choose the Adam optimization technique to update the model's parameters. Adam combines the advantages of momentum and adjustable learning rates to accomplish effective optimization.

We load and preprocess the train and validation datasets using the `DataLoader` class from the PyTorch library. The `DataLoader` facilitates efficient batch loading and shuffling of the data, which helps in achieving better generalization. We initialize the train and validation loaders with the respective datasets, batch size, and the number of workers responsible for loading the data.

To enable comprehensive logging and visualization of the training process, we initialize the WandB library. We provide the project name and set relevant configuration parameters such as batch size and learning rate. Additionally, we log the model's architecture using the `wandb.watch()` function, allowing for visual inspection and analysis.

The actual training loop begins, iterating over the specified number of epochs. For each epoch, we initialize a running loss variable to keep track of the cumulative loss.

We set the model to training mode using the `model.train()` method, ensuring that any layers involving dropout or batch normalization behave accordingly. We then iterate over the train loader, transferring the inputs and labels to the chosen device (GPU if available) for efficient computation. We zero the gradients using `optimizer.zero_grad()`, and perform a forward pass through the model. We compute the loss based on the model's output and the ground truth labels. Backpropagation and optimization are carried out by calling `loss.backward()` and `optimizer.step()`, respectively, which update the model's parameters.

We update the running loss by summing the losses over the batches and multiplying by the batch size. This helps us keep track of the overall loss during each epoch. Additionally, we log metrics such as the step number and current loss to WandB using the `wandb.log()` function, enabling real-time monitoring and analysis of the training process.

After completing the training loop for one epoch, we set the model to evaluation mode using `model.eval()`. This ensures that any layers involving dropout or batch normalization behave differently during evaluation. We evaluate the model's performance

on the validation set by calling the `evaluate()` function, passing the model, criterion (loss function), device, and validation loader. The `evaluate()` function calculates the validation loss and any other evaluation metrics of interest.

Epoch statistics, including the epoch loss and validation loss, are printed to the console for visibility. This information allows us to monitor the training progress and assess the model's performance over time. The epoch metrics are also logged to WandB, providing a comprehensive record of the training process.

Once all the epochs are completed, we save the trained model to a file named `"model.pth"` using the `torch.save()` function. This allows for future use and deployment of the trained model. Additionally, we save the model to the WandB storage using `wandb.save()`, providing an additional backup of the model.

Finally, we finish the WandB run by calling `wandb.finish()`, ensuring proper termination and cleanup.

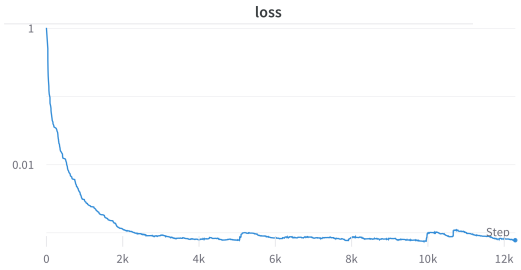
3.3 Metrics

During the training process, we ran 50 epochs using a batch size of 32 and a learning rate of 0.001. The AMD RX580 GPU, coupled with MPS GPU acceleration in PyTorch, significantly improved training efficiency, completing the experiment in approximately 3 hours and 20 minutes.

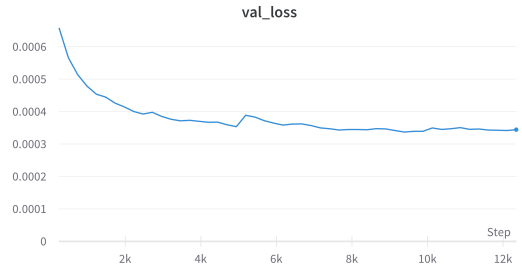
Although we couldn't utilize data augmentation due to limited computational power, the results were still impressive. The model demonstrated strong predictive capabilities for both stars and galaxies, proving the effectiveness of our network design and hyperparameter choices.

We utilized the WandB library, which proved to be instrumental in logging and visualizing various metrics. This encompassed essential aspects such as loss, validation loss, and test loss. The use of WandB's functionalities allowed us to benefit from real-time monitoring, collaboration features, and the capability to share results with ease. Additionally, it facilitated experiment reproducibility, bolstered by hyperparameter optimization support and interactive visualizations. Through the application of these impressive capabilities, we gained valuable insights into our model's training process, enabling us to enhance its performance effectively.

Throughout the training phase, we developed insightful charts that effectively displayed the model's loss, validation loss, and test loss. These graphical representations provided an easy-to-understand visual representation of how the model's performance developed over different epochs. These graphs also provide important insights into the dynamics of the model's training and its capacity for accurate generalisation to new data. In our efforts to comprehend and improve the performance of the model, such visual assistance were quite helpful.



(a) Loss.



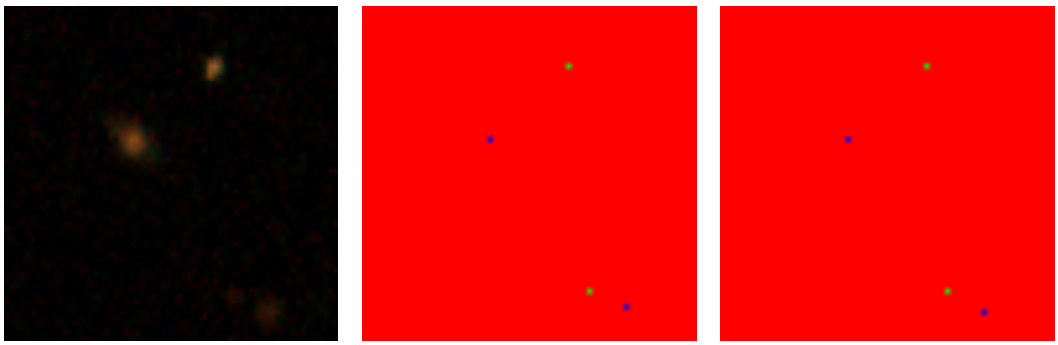
(b) Validation Loss.

(c) Loss.

Figure 3.2: Loss curves for the model.

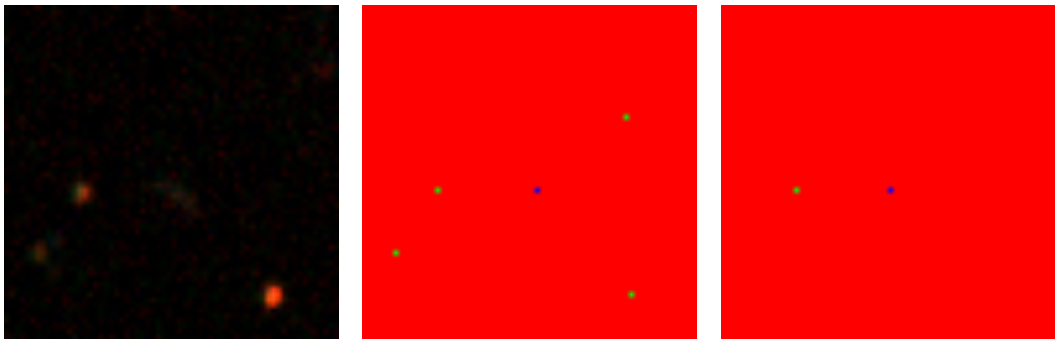
3.4 Results

The model's preliminary performance after training on a consumer GPU with constrained epochs and a constrained network is encouraging. Effectively, it can predict both galaxies and stars. Here are some example results that demonstrate how the model performed. Further optimization and fine-tuning can potentially improve results.



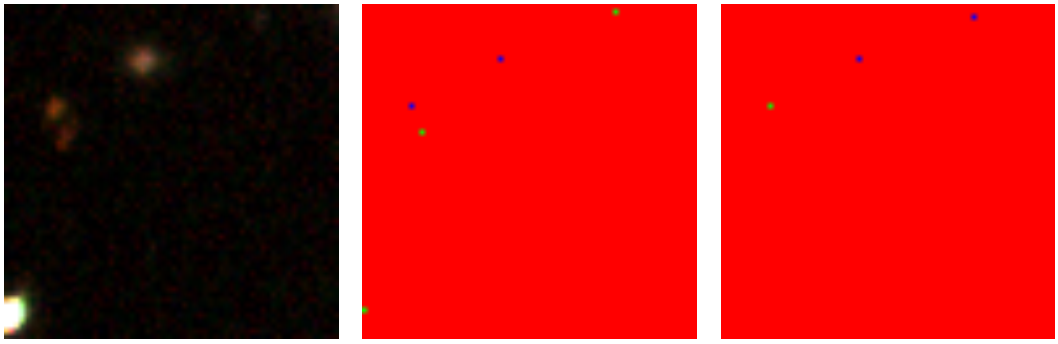
(a) Input Image. (b) Ground Truth Mask. (c) Predicted Mask.

Figure 3.3: Correct prediction of both Stars and Galaxies.



(a) Input Image. (b) Ground Truth Mask. (c) Predicted Mask.

Figure 3.4: Some stars are missing in the prediction.



(a) Input Image. (b) Ground Truth Mask. (c) Predicted Mask.

Figure 3.5: Example of misclassification.

Additionally, to these visual results, we measured the recall, precision, and F1-score for each of the three classes over the test set. The results can be seen in Table 3.1. We can see that the Sky class is predicted almost perfectly with precision, recall, and an F1-score of 0.99 for each. This is likely due to the vast majority of the pixels present being sky pixels. For the other two classes, we got significantly worse results. For the star and galaxy classes, we have a precision, recall, and F1-score of 0.25, 0.18, 0.20, and 0.17, 0.16, 0.16, respectively. This indicates that our model has issues with identifying the presence of the two elements as well as predicting them correctly.

Class	Precision	Recall	F1-score
Sky	0.99	0.99	0.99
Stars	0.25	0.18	0.20
Galaxies	0.17	0.16	0.16

Table 3.1: Evaluation metrics on the test set.

4 Discussion

The results of our experiment showed that the UNet with lower depth was not sufficient to predict the locations of stars and galaxies effectively in the given test set. We assume this to be due to a combination of reasons. Firstly, the model architecture of the UNet was chosen due to hardware constraints, as we did not have the resources to train the regular architecture. Secondly, the dataset contained way more sky in its images than stars and galaxies making them hard to be identified. This combined with our approach to translate the coordinates of them to single pixels left us with a majorly imbalanced class distribution.
The model was better at finding the location of stars compared to galaxies. This could also be due to the imbalance in the dataset or because galaxies are generally harder to identify than stars. The mentioned issue could be addressed when attempting this classification again in the future.

5 Reproducibility

To reproduce the results in this report refer to the README of the repository of this project. (https://github.com/felix-zailskas/star-galaxy-classification)