Classifying galaxy images with a Convolutional Neural Network in PyTorch

Daniel Horta Darrington

February 2026

1 Introduction

Around 1926, another white man (in this case, Edwin Hubble) used one of the most powerful telescopes of the time to show that many of the previously classified "nebulae" were instead galaxies of their own, ending the "Great Debate" of the 1920s. This discovery[1] opened a window into our understanding of the make-up of our Universe, expanding it from one galaxy (the Milky Way) to billions. Mr Hubble, in his seminal work in the 1920s-1930s showed that not only the Universe is comprised by more than one galaxy, but that these come in many shapes, sizes, and colours (amongst other properties). To classify the different characteristics he observed, he devised the famous "tunning fork" sequence, which distinguished galaxies based on their more prominent features, and enabled one to distinguish more round shape galaxies from those flatter discy ones (see Fig 1 below).

Ever since this work, astronomers have been obsessed with classifying galaxies, for a good reason. Galaxy classification can be done in many different ways, but the typical avenue involves dividing them based on their appearance (i.e., morphology). The reason for this is because galaxy morphology, at its most basic level, tells us about the physics of assembly of a galaxy. Since the light from galaxies (mostly) comes from the stars, morphology is a frozen snapshot of where those stars are, and where they are forming today in a galaxy. You

[1]While Edwin Hubble is known to be the first scientist to prove that there are "other" galaxies in the Universe apart from our own, this realisation was conceived by Immanuel Kant in the 1750s, and was debated heavily during the 1920s, largely pushed by Heber Curtis. However, I wonder if it was postulated by another human thousands of years before these guys. The Large and Small Clouds (typically named the Magellanic clouds) are a pair of satellite galaxies (i.e., other galaxies) visible to the naked eye in the southern hemisphere; these galaxies were known by many ancient civilisations (e.g., south-american civilisations like the Incas, or the aboriginal and Torres Strait Islander peoples of Australia). While there is no record stating that these civilisations knew these were other galaxies, I wonder what a person in those times would have thought seeing these beautiful systems in the sky, and if they also imagined them being other galaxies? Regardless, Mr Hubble was certainly the first to document it scientifically, and to later devise a classification method... pretty scientific I say! Thus, he does deserve the credit he is given. After all, we did name a telescope after him, so he must have done something important!

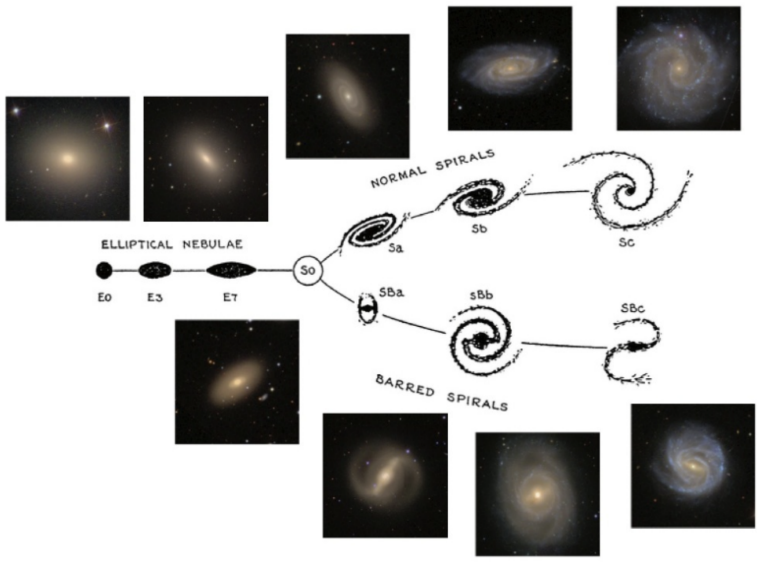


Figure 1: An illustration based on a diagram from Hubble (1936) showing the original Tuning Fork classification scheme, with additional example galaxy images from the Sloan Digital Sky Survey (SDSS). Image first published in Masters (2015).

can also consider galaxy morphology as an (albeit imperfect) proxy for measurements of the orbital motions of the stars - visible discs in a galaxy reveal places where stars are coherently rotating, dynamical features like spirals and bars give even more information on those orbits, while smoother spheroidal blobs reveal the presence of more random motions.

Thus, classifying galaxies is pretty important. In the good ol' days, this was primarily done by trained astronomers, who painstakingly looked at galaxy images and, one-by-one, classified galaxies into catergories like "dwarf lenticular galaxy" or "unbarred tight spiral galaxy". However, with the advent of sophisticated computing, machine-learning, and now artificial intelligence, astronomers have become lazy and instead have built smart models that do the classification for them. This is where this report comes in.

Listed in this short report are the results of a week-long exploration I undertook to learn how to write a convolutional neural network (CNN) model that can classify galaxies based on galaxy images. This report is by no means supposed to be a scientific publishable result. Instead, it is a training exercise, or could even be treated as a guide, for someone wanting to learn how to build up a CNN to classify images (although if someone in the field thinks that this is interesting enough to be publishable, hit me up!). As I am an astronomer, and I like looking at pretty galaxy images, I decided to do this on galaxy classification. However, for those that prefer starring at cats (I am more of a dog person myself) or other things, you could envisage re-purposing this model to train on images you have collected instead. Most people train on numbers, but numbers that aren't in my bank account are boring to me. Thus, for this report we will stick to galaxies.

2 Data

The data used in this document to train, test, and validate the model developed contains solely galaxy images obtained with the DECals survey (Dark Energy Camera Legacy Survey, see here for more information). Thanks to the very kind people in Toronto (muchas gracias Henry Leung and Jo Bovy!), there is a very easy to use and already nicely compiled data-set of galaxy images (called Galaxy10 DECaLS, you can find these here). Galaxy10 DECaLS has combined a bunch of images of galaxies from the Galaxy Zoo project, SDSS images, and DECaLS), where $\sim$ 18k images were selected to be classified into 10 broad classes using volunteer votes with more rigorous filtering.

In detail, this catalogue contains 17,736 images in three observing bands (g, r, and z) of galaxies classified into 10 categories: Disturbed, Merging, Round Smooth, In-between Round Smooth, Cigar Shaped Smooth, Barred Spiral, Unbarred Tight Spiral, Unbarred Loose Spiral, Edge-on without Bulge, and Edge-on with Bulge. A summary of what these galaxies look like is shown in Fig 2. Each image has 256×256 pixels.

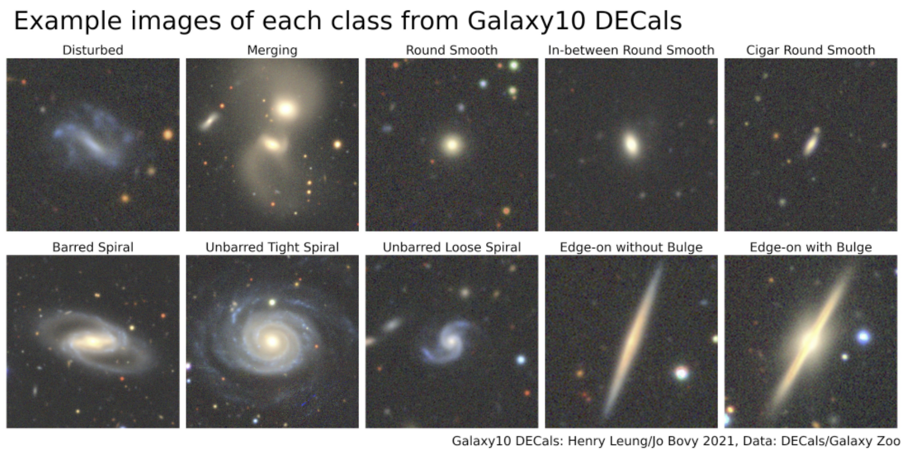


Figure 2: Example of images of each class from the Galaxy10 DECals/Galaxy Zoo survey. Image taken from astroNN/Henry Leung/Jo Bovy 2021.

2.1 Data processing

In order to put the galaxy image data into a format that the CNN model can exploit, I perform a set of tweaks. Firstly, I crop the images from 256×256 pixels to 128×128 pixels, selecting the central pixels. Since all the galaxy images are centered on the galaxy of interest, I didn't need to do any shifting of the image (thank god, cause this is a pain!), and by cropping the image I could focus each image on the main pixels I want the model to learn on. I also tried cropping to a size of 64×64, but found that the model didn't learn as well the properties in the images.

After cropping the images, I used the `TensorDataset` function in `PyTorch` to create a dataset object that had both the images and label information. I then divided the parent sample into a training set (70%), a test set (15%), and a validation set (15%), by randomly splitting the data; this allowed me to have an equal balance of each class in all of the different training, testing, and validation datasets (see Fig 3).

Lastly, before running the model on the data, I decided to flip and rotate some of the images in the training dataset. I did this because, since galaxies come upside down, twisted, and in mirror-versions of eachother, I wanted the model to learn the features of a type of galaxy, rather than learning, for example, that spiral galaxies come with spiral arms only in the left to right direction in the image (i.e., learning the pixels). Thus, by implementing a 50% chance of a galaxy being flipped horizontally or vertically, and including random rotations between [–180,180] for some of the images, I ensured that the model learned galaxy features. This is applicable to galaxy data, but may not be applicable to other images (like cats, unless upside-down cats exist?). The code I used to do this is all documented here.

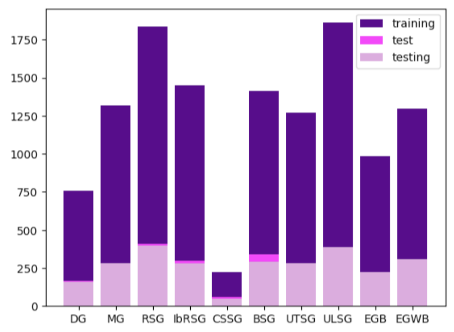


Figure 3: Galaxy classes for each of the training, testing, and validation samples used in this experiment.

3 Convolutional Neural Networks

Before going on to classify galaxies, I thought it would be good to describe what model I use, and why.

3.1 What are they?

A Convolutional Neural Network (CNN) is a type of (feedforward[2]) neural network that learns features (or labels) via filter optimisation. This filtering is performed using a "kernel" (or convolution matrix, hence the name), that in image processing is a matrix that masks the input information of the image in order to blur, sharpen, or detect edges (amongst other tasks). In practice, this procedure is accomplished by doing a convolution between the kernel and an image; in simpler terms, convolution here just means that a kernel or filter "slides" across the image to create a new (typically compressed) version of that image that highlights specific features that will then be outputted.

Mathematically, convolution is the operation of applying a mathematical operation on two functions, say f and g to yield a third function, $f * g$. In CNNs, convolution is a specialised kind of linear operation, which can be viewed as multiplication by a matrix. Convolutional layers in a CNN can be computed by

$$x_j^l = f(\Sigma_i \in M_j x_i^{l-1} \times k_{ij}^l + b_j^l), \tag{1}$$

where l is the number of the layer, f is the activation function (typically a Rectified Linear Unit or ReLU), k is the convolutional kernel, M_j represents the receptive field, b is the bias, and x is the input/output.

In summary, CNNs work via filter optimisation, which is is a process in which you reduce the amount of information in an image using a kernel (via convolu-

[2]Feedforward models, as the name implies, are models in which the information flows in only one direction (forward).

tion) in order to determine the necessary information (edges, sharp transitions, etc) that will allow the model to pick up the important aspects to learn on to then be able to classify.

3.2 Why are they useful/important?

CNNs are the de-facto standard method in deep-learning based approaches to computer vision and image processing (although transformers nowadays are starting to take over in some cases). They are used widely for tasks such as image classification and object recognition. They are so highly used because of their ability to regularise by using shared weights over fewer connections between the neural-net layers. Conversely, traditional neural networks (with fully connected networks) are not scalable for large images, as they treat every pixel as a separate input, resulting in massive networks with too many parameters. For example, for each neuron in a fully-connected layer (i.e., a bog-standard neural-net), 10,000 weights would be required for processing an image sized 100×100 pixels. However, by applying convolution kernels, only 25 weights for each convolutional layer are required to process 5×5-sized tiled versions of the image. Via this process, higher-layer features are extracted from wider context of information stored in the image, compared to lower-layer features.

In simpler terms, because of the convolutional aspect of CNNs, these models can "pick out" the important information in images and reduce the amount of information, allowing the model to learn the important features to focus on to then be able to classify new images.

In summary, CNNs are powerful because:

- They have far fewer parameters than regular networks because they share weights across the image.
- They are excellent at detecting patterns, even when slightly shifted or distorted.
- They are good at recognizing patterns, even if they are distorted or shifted.
- They can automatically learn what is important, without us having to tell them.

However, a disadvantage is that CNNs require a lot of data and computing power to train effectively.

3.3 How do they work?

CNNs work similar to the human eye and brain. In the first step (these earlier layers), the CNN inspects small segments of the image (e.g., 3×3 pixel squares) to scan for important features (e.g., colours or edges). This allows the model to learn the general patterns. The model then discards the less important information (e.g., background) to work more efficiently and use less memory.

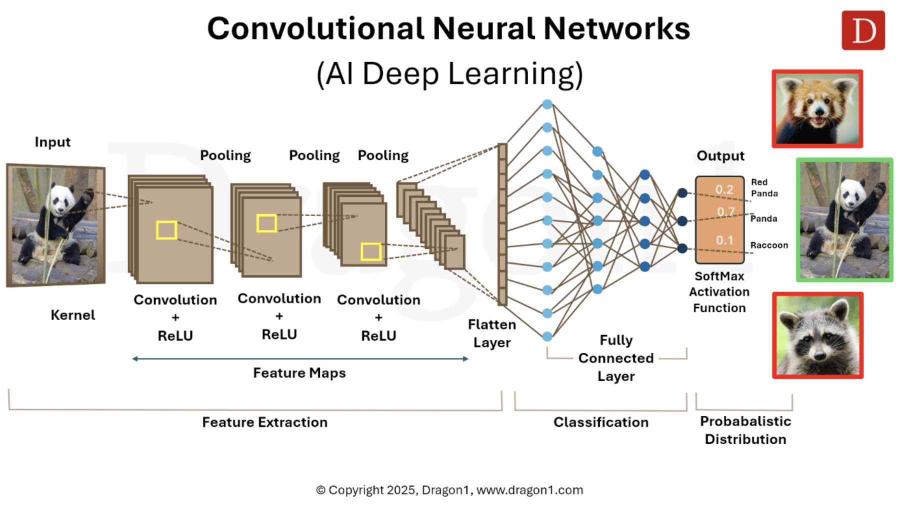


Figure 4: Diagram of a Convolutional Neural Network for animal classification, taken from Dragon1.

This process is the same as looking at a family photograph and focusing on the face of your (what was then) young grandmother, for example, whilst ignoring the fact that she was probably sipping on a big glass of nana juice. Lastly, the model then combines all the information from all stages and states to make a decision (e.g., I think this image contains a cat with $X\%$ certainty).

Structurally, CNNs are constructed using three main types of layers: a convolutional layer, a pooling layer, and a fully connected layer (see Fig 4). Typically, a CNN will have a mix of convolutional and pooling layers, with activation functions in between, and will finish with a final fully-connected layer. With each additional layer, the CNN increases in complexity at the cost of computational expense, allowing it to identify greater portions of the image. In practice, early layers focus on simpler features (e.g., colours or edges), while later layers start to recognise larger elements or shapes of the image until identifying the object.

In more detail, the common structure of a CNN includes:

- **Input data (image):** Typically a $N \times N$ size image, centered on the object of interest (but doesn't have to be!).
- **Convolutional layer:** A small filter slides over an image and recognizes patterns, such as edges or corners. This produces a feature map: a map of where specific patterns were found.
- **Activation function:** for example, a ReLU (Rectified Linear Unit) or a sigmoid function. Activation functions are used to add non-linearity to the system, enabling the network to learn more complex relationships and patterns.

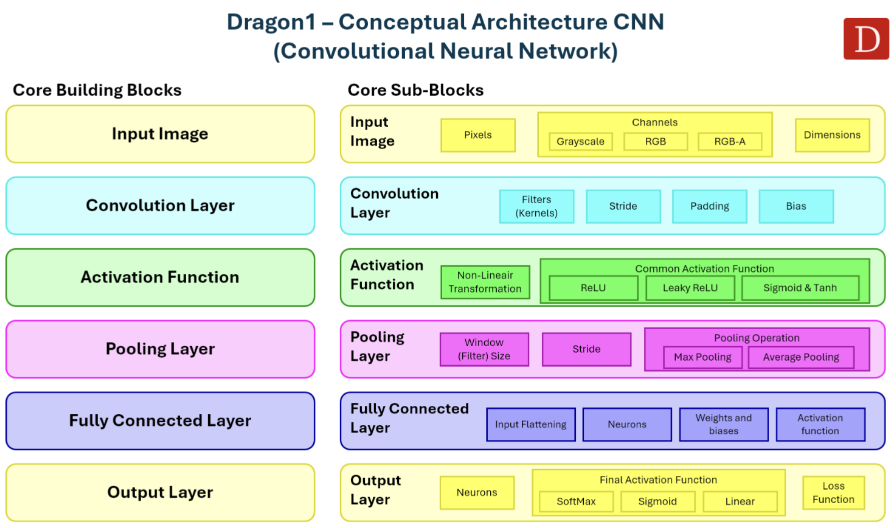


Figure 5: CNN architecture, with a summary of typical properties of each layer. Figure taken from Dragon1.

- **Pooling layer:** Simplifies the data by reducing its size (e.g., using max pooling — picking the largest value in a region). For example, for a small 2×2 pixel window [4, 7, 1, 5], only the 7 is retained in max pooling. The pooling layer helps reduce data size, increase speed, and perhaps most importantly helps prevent overfitting.
- **Fully connected (or flattening) layer:** Flattens 2D feature maps into a 1D vector for feeding into the final layers. Combines all learned information to make a prediction.
- **Output layer:** Produces the final result, often using a softmax function to calculate probabilities.

A summary of each layer is shown in Fig 5

4 A CNN for galaxy classification

For this experiment, I built a CNN using five convolutional layer blocks, followed by an adaptive average pooling block, a dropout layer, and a final fully connected linear layer. In more detail, each convolutional layer block was instantiated with a kernel size equal to 3 and a padding equal to 1, to ensure that the image sizes remained untouched (the padding), and that the kernel is large enough to capture spatial relationships but small enough to keep the model efficient and fast. This was achieved using the `Conv2d` function in `PyTorch.nn`. I could have adopted instead a kernel size of 2 or 4 to test how this affects the model, but didn't want to.

At each convolutional layer I added a batch normalisation using the `BatchNorm2d` function in `PyTorch`; this function essentially grabs all the images in the batch (in my case, 64), calculates the mean and variance for all pixels in the batch, and then normalises the data to have a mean of 0 and variance of 1, allowing the model to learn two special parameters (γ and β) to slightly nudge the data back to where it needs to be for the ReLU activation function to work best. After each batch normalisation, the output then goes through a Rectified Linear Unit (ReLU) activation function and a max pooling of 2×2. After the five convolution blocks, the output is passed through an adaptive average pooling and is flattened, before applying a 40% dropout (i.e., 40% chance to drop a neuron during training). Finally, the model is then passed through a fully connected linear layer before being outputted.

At the input stage, the images have a shape of 128×128 pixels. At each convolutional layer, the images size is halved (because of the max pooling), going from $128 \times 128 \rightarrow 64 \times 64 \rightarrow 32 \times 32 \rightarrow 16 \times 16 \rightarrow 8 \times 8 \rightarrow 4 \times 4$; at each of these layers, the number of neurons in the CNN goes from $3 \rightarrow 32 \rightarrow 64 \rightarrow 128 \rightarrow 256 \rightarrow 512$. The final fully connected layer then compresses all the information from the 512 neurons into 10 classifications, yielding a prediction for each image and class.

I choose to run this model using a cross entropy loss criterion with weights for each class. The weights are calculated as

$$w = \frac{N_{\text{galaxies}}}{N_{\text{classes}} \times N_{\text{galaxies in each class}}}. \tag{2}$$

I choose to use a cross entropy loss criterion (implemented using `CrossEntropyLoss` in `PyTorch`) because this criterion has been shown to be successful (i.e., the standard choice) for classification tasks. In practice, the cross entropy loss criterion calculates the negative log of the probability assigned to the true class, such that

$$Loss = -\sum_{c=1}^{N_{\text{classes}}} y_c \log(\hat{y}_c) \tag{3}$$

where y_c is the true label and $\hat{y}_c$ is the model prediction. This criterion punishes overconfidence; since it uses a logarithmic scale, the further away the prediction is from the actual truth, the more the loss increases exponentially.

Lastly, I use the `Adam` optimiser, initially with a learning rate equal to 0.001, and later during fine-tuning with a learning rate equal to 0.0005. I choose to use the Adaptive Moment Estimation (`Adam`) optimiser since this has been proven to be highly efficient. In detail, `Adam` works (unlike other methods) by using a separate learning rate for each parameter in a model (in this case weights in the network), allowing it to reach global minimum much easier and faster. `Adam` also remembers previous gradients (and thus has momentum): if the optimizer finds a small, flat plateau in the loss landscape, the momentum helps carry the model across that flat spot so it can find the next downward slope toward global minimum. Structurally, the `Adam` optimiser is a marriage of two techniques: momentum from stochastic gradient descent, and scaling learning rate from Root Mean Square Propagation (RMSProp). In practice, the parameters/weights, θ, in the CNN get updated like this:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t, \tag{4}$$

where $\hat{m}_t$ is the estimate of the mean (momentum), $\hat{v}_t$ is the estimate of the variance (scaling), and η is the learning rate.

4.1 Results

I run the model initially for 50 epochs, allowing for early stopping with a patience tolerance of 5. During each epoch, the model trains on batches of 64 images, and then predicts a classification for test data. At each epoch, I calculate both the training and test accuracies, where if the model does not improve during 5 consecutive epochs, I invoke early stopping. My results show that the model starts with a training accuracy of $\sim 34\%$ and a test accuracy of $\sim 23\%$ at the first epoch, and then reaches a training accuracy of $\sim 76\%$ and a test accuracy of $\sim 75\%$ by epoch 19, where it terminates due to the early stopping. Overall, the model quickly increases in accuracy (it took around 34 minutes) whilst maintaining a similar value between the training and test predictions, suggesting no overfitting and good generalisation.

After initial training, I set out to fine-tune the model by running the training-testing algorithm for another 50 epochs with the same set-up, instead this time using a lower learning rate (of 0.0005). I find that the model slowly increases in accuracy until reaching $\approx 80\%$ overall for both training and test data before early stopping; to achieve this level of accuracy, the model ran for 22 epochs and ≈ 40 minutes.

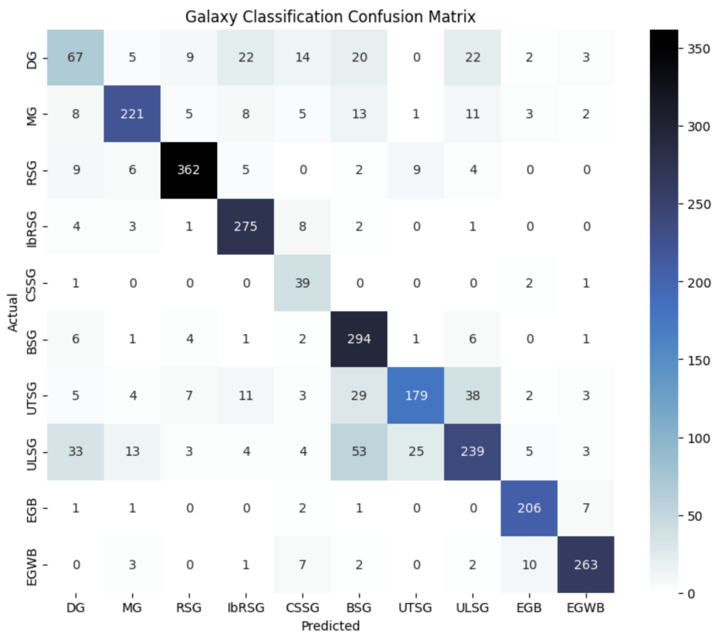
In order to test how well my model does on unseen data, I validate its performance by predicting galaxy classifications using the validation set. To visualise this, I calculate the confusion matrix (i.e., the number of times the predicted and real classifications match per class), which I show in Fig 6, and calculate the accuracy of my model for each class using the following formula:

$$\text{accuracy} = \frac{N_{\text{correct}}}{N_{\text{total}}}. \tag{5}$$

The resulting accuracies are summarised in Table 1

5 Takeaways

Overall, I find that my model does a decent job in classifying galaxies. However, it does do better for specific types of galaxies than for others. For example, my model classifies Round Smooth galaxies, In-between Round Smooth galaxies, Cigar Shaped Smooth galaxies, Barred Spiral galaxies, Edge-on without Bulge



	DG	MG	RSG	lbRSG	CSSG	BSG	UTSG	ULSG	EGB	EGWB
DG	67	5	9	22	14	20	0	22	2	3
MG	8	221	5	8	5	13	1	11	3	2
RSG	9	6	362	5	0	2	9	4	0	0
lbRSG	4	3	1	275	8	2	0	1	0	0
CSSG	1	0	0	0	39	0	0	0	2	1
BSG	6	1	4	1	2	294	1	6	0	1
UTSG	5	4	7	11	3	29	179	38	2	3
ULSG	33	13	3	4	4	53	25	239	5	3
EGB	1	1	0	0	2	1	0	0	206	7
EGWB	0	3	0	1	7	2	0	2	10	263

Figure 6: Confusion matrix of the results.

galaxies, and Edge-on with Bulge galaxies extremely well, with an accuracy above 90%. Conversely, my model is unsuccessful classifying Disturbed galaxies (with the worst performance of $\approx 41\%$ accuracy), and struggles to distinguish small scale features between Unbarred Tight and Unbarred Loose Spiral galaxies.

In order to increase the overall accuracy of this model, I would focus on including tweaks to the structure and data, without changing the overall structure of my model, that will enable it to learn small scale features. Since my model is struggling to capture the small scale features of galaxies, to improve this model I would try fine-tuning with smaller batch sizes and a lower learning rate, allowing the model to slowly learn small scale features. I could also potentially artificially adjust the weights in the model to be biased towards learning those classes that my model is not as successful at classifying (e.g., disturbed or unbarred galaxies). I could also remove the max pooling for the first two layers of the model, which would allow the model to retain more pixels and thus learn more information.

In the future, it would also be interesting to test how well pre-trained models such as ResNet or AlexNet work on galaxy image classification. I am sure those sophisticated pre-trained models would work much better than my simple model.

Class	Accuracy (%)
Disturbed	40.8
Merging	79.8
Round Smooth	91.2
In-between Round Smooth	93.5
Cigar Shaped Smooth	90.7
Barred Spiral	93.0
Unbarred Tight Spiral	63.7
Unbarred Loose Spiral	62.6
Edge-on without Bulge	94.5
Edge-on with Bulge	91.3

Table 1: Table summarising the accuracy of the model.