DEEP LEARNING PROJECT

Master in Data Science and Advanced Analytics

NOVA Information Management School
Universidade Nova de Lisboa

Deep Learning Report

Group 33

Alexandre Gonçalves (20240738)

Bráulio Damba (20240007)

João Henriques (20240499)

Mariana Sousa (20240516)

Victoria Goon (20240550)

Spring Semester 2024-2025

1. Introduction

In recent years, the field of computer vision has seen major developments due to factors such as breakthroughs in optimization algorithms and the rise of GPU computing, which have enabled the training of deep neural networks with thousands of layers. In this project, we applied these methods to the field of biological image classification. Using a dataset of thousands of images of rare species, sourced from the Encyclopedia of Life and curated as part of the BioCLIP project, we set out to build a Convolutional Neural Network (CNN) that can predict the family of a given species based on its image. The remainder of this project is organized in the following manner: The exploratory data analysis can be found in **Section 2**. **Section 3** details the preprocessing steps implemented. **Section 4** details the modelling stage and different steps tried to **reach** the final model selected. Finally, the project's conclusions and future work will be provided in **Section 5** and **6** respectively. To understand how the notebooks complement each report section please check **Appendix 1.1**.

2. Exploratory Data Analysis

Our data is composed of 11,983 images of animals belonging to 5 phyla and 202 families. Before proceeding to the model creation, we need to explore the dataset so that we are familiar with its characteristics, enabling us to identify issues that must be corrected. Since the dataset did not come pre-split, we were able to conduct our EDA on the data as a whole before creating our test, train and validation subsets at a later stage. At a first look, we identified a large imbalance when it came to Phyla, with the phylum *chordata* amounting to 83% of all images represented and including 166 of the 202 families (**Appendix 2.1**). Looking at families, the distribution of image counts per family is heavily right-skewed, with most families having relatively few images. Defining overrepresentation as those families whose image count is above the upper bound (Q3 + 1.5*IQR = 105 images), we found 23 overrepresented families, belonging mostly to *chordata* (**Appendix 2.2**). This class imbalance was addressed during the Preprocessing stage (**Section 3**). Looking for discrepancies, we found that image dimensions varied widely, with no uniform resolution across the dataset. There were also some images that were not in RGB (**Appendix 2.3**) and were therefore converted.

3. Preprocessing

In order to feed our images into a model to train, we first needed to transform our data to be completely uniform. Several issues identified in the EDA were solved when using a Keras method image_dataset_from_directory (Keras Team, n.d.). This function helps with multiple things other than simply loading out data into our code. It resizes the images to have a uniform width and height and also helps ensure all images use the same colour mode. Ours is set to RGB to make sure the pixels have 3 channels each (red, green, and blue) and that the values range from 0 to 255. By ensuring all images are scaled consistently, we reduce the risk of inconsistent value ranges that would negatively affect training.

3.1 Noisy data removal: Mentioned within the EDA, we found several instances where images did not reflect what we considered to be an accurate depiction of an animal. As these images could be considered noise, we came up with two different methods to remove these images: One was with manual inspection, following a strict guideline of criteria to classify whether an image was an outlier and the second was using a method called confident learning (Northcutt et al 2022) (**Annex A**). Due to a lack of computational resources required to implement the second approach, we utilized method one instead. Our confident learning attempt is detailed further in section 6. Future Works.

Manual outlier inspection was considered because the dataset was small, meaning noise images would carry more weight and their removal would be more impactful. Images were removed based on the following criteria: skulls, skeletons, or body parts; papers or diagrams where the animal was not clearly present; unrelated content not showing the animal; and animals not clearly visible due to low resolution, extreme blurriness, or darkness (examples shown in **Appendix 3.1**). It's important to note that these removal criteria were applied uniformly across all classes, meaning that the decision to exclude an image was based solely on its content rather than its class label. This ensured that no information from the target variable (the animal class) was inadvertently used to guide the image removal process. Additionally, this outlier removal process was performed before the train, validation, and test split, meaning the decision to exclude images occurred prior to defining the data used for training and evaluation. As a result, there was no risk of data leakage, since the image selection process was fully independent of the future model training and label information. After applying these criteria, we removed 1,005 images, leaving a total of 10,978 images for further analysis. The final dataset obtained was saved inside the outliers_manual_removed/cleaned_unsplit folder (**Appendix 4.0.1**). The manual removal of outliers served as one of our preprocessing steps used to determine the optimal data preprocessing strategy in Section 4.

3.2 Oversampling/Augmentation: For the large imbalance in data per family, we utilized a method found in Langenkämper et al., 2019. The method involves using transformative oversampling which applies different augmentation methods to pre-existing images to increase the population of the minority classes. Each class that gets increased has a set limit to how many images get generated which follows the formula mentioned in **Appendix 3.2** where s(l) is the number of images in a select class and $s(l_{maj})$ is the size of the majority class. The augmentations used here follow Langenkämper et al 2018. which used a Gaussian Blur, a rotation of the original image by 90 degrees, and a flip along the vertical and horizontal axis. Due to the large discrepancy between the smallest minority class and the majority class, we also chose to define a general cap that limited any family from being oversampled beyond 3× its original size. While this cap was defined uniformly, it mostly affected minority classes, since the resampling tiers provided by the paper approach already imposed stricter limits on the larger families. This was done to reduce the risk of the model overfitting on certain images.

4. Modeling

After data preprocessing, we obtained **various datasets** based on the different preprocessing techniques applied **(Appendix 4.0.1)**. To ensure representative class distributions across train, validation, and test sets in our imbalanced dataset, we applied a **stratified split** based on the family label, preserving the original class proportions. To evaluate the different models obtained, the following metrics were used: Categorical Accuracy (overall prediction correctness), Precision and Recall (class-wise performance), F1 Macro (ideal for imbalanced datasets), F1 Weighted Average, and Train, Validation, and Test Loss (monitor model generalization). Considering the computational requirements of training these models, we purchased **Google Collab Pro** (**Annex B**) to make the process more efficient. It's important to highlight that due to its limitations, the execution order of cells may not appear in our notebooks and even when it does, the order might not reflect the actual sequential execution due to session restarts. The modelling stage is organized with the following structure: Data Preprocessing Strategy implemented, Model Architecture Selection, Hyperparameter Tunning, Overfitting Reduction Approaches, and Final Model Selection and Evaluation.

4.1 Data Preprocessing Strategy: This section explains the steps taken to determine the optimal data preprocessing strategy, including whether to apply augmentation, perform outlier removal, and use oversampling. To evaluate different preprocessing combinations, we began by building a CNN model from scratch (excluding transfer learning), using a modified version of Tang's architecture (Shin et al. 2016) adapted to our problem by increasing model depth and adjusting the classification head for higher-resolution inputs (**Appendix 4.1.1**). Additionally, we incorporated a Rescaling layer to normalize pixel values to the [0, 1] range along with augmentation layers Random Flip, Random Rotation, Random Contrast, and Random Translation based on the approach used in Ferreira et al.

2018. The callbacks used during training are detailed in **Appendix 4.1.1**, with emphasis on the LearningRateScheduler (exponential decay) and EarlyStopping (patience=3) to help mitigate overfitting. With our baseline model set, we first tested the impact of augmentation by comparing two models both with outliers and no oversampling, differing only in augmentation. Despite a slight F1-score improvement with augmentation (0.117 vs. 0.104), the gains were minimal and didn't justify the added computational complexity, so we chose to proceed with no augmentation for further steps. We then sequentially tested the remaining preprocessing steps (first outlier removal, followed by oversampling) but due to consistently poor results (all F1 macro scores in the test set were below 0.15) (**Appendix 4.1.2**), we shifted to a new CNN model using transfer learning (**Annex C**) with a pre-trained network to evaluate the preprocessing strategy.

The new model used VGG16 as its base due to its proven performance in image classification, suitability for transfer learning (Simonyan et all 2015) and its relevance to Lab 5. The classifier head, preprocessing, augmentation, and compilation of the model followed Ferreira et al. 2018 and Kassani et al. 2019. While Ferreira et al. proposes a hybrid transfer learning approach combining feature extraction (keeping all pre-trained layers frozen) and fine tuning (unfreezing some layers during training), we adopted only the feature extraction strategy as done in Kassani et al. (**Appendix 4.1.3**). Using the new model and the same callbacks as before, we sequentially evaluated all preprocessing steps (**Appendix 4.1.4**) and determined that the best strategy (going to be used from this point onward) was no augmentation, outlier removal, and application of oversampling, leading to a higher F1 macro score (0.284), improved test accuracy, and lower losses (**Appendix 4.1.5**).

4.2 Model Architecture Selection: Using the final preprocessing strategy and consistent model design, four base pre-trained architectures were explored and pre-trained using the ImageNet dataset: VGG19, InceptionV3, ResNet50V2, and Xception. VGG19 (Simonyan & Zisserman) is a deep but simple 19-layer architecture influential in early CNN design. InceptionV3 (Szegedy et al. 2015) improved InceptionV2 using extended factorized convolutions and regularization to boost performance with fewer parameters. ResNet50V2 (He et al. 2016) builds on ResNetV1 by introducing pre-activation residual units for better training stability in deep networks. Lastly, Xception (Chollet et al. 2017) replaces inception modules with depthwise separable convolutions, achieving superior performance on ImageNet. To select the best architecture, we evaluated the trade-off between test F1 macro score and overfitting. Among the top performers (ResNet50V2, InceptionV3, and Xception), InceptionV3 and Xception achieved the highest F1 test scores (0.596 vs. 0.65, **Appendix 4.2.1**). Although Xception showed more overfitting with early stopping at 9 epochs and a larger train-validation loss gap (**Appendix 4.2.2**), its higher score combined with the fact that InceptionV3 also registered overfitting lead us to select Xception as the final model.

4.3 Hyperparameter Tunning: After selecting the best architecture, we proceeded to apply hyperparameter tuning to optimize model performance (**Annex D**). Vo et al. 2023 compared Random Search and Bayesian Optimization, showing better results with the latter. Since we had already obtained a strong performing Xception model and aimed for efficient fine-tuning, the Bayesian Optimization class from Keras Tuner was selected due to its ability to focus the search on promising regions and efficiently optimize expensive black-box functions like deep learning models (Snoek et al. 2012). Based on Vo et al. 2023, the initial suggested hyperparameters and search space (**Appendix 4.3.1**) were considered but for computational efficiency a reduced version of it with the tuning limited to ten trials was defined (**Appendix 4.3.1**). For each trial, the best model was selected based on the lowest validation loss, and across the ten trials, the model with the highest F1 macro score on validation data was selected. Following this tuning process, the best model achieved the highest test F1 macro score so far (0.702) and the lowest validation loss (1.0998, **Appendix 4.3.2**) with the best hyperparameters defined in **Appendix 4.3.2**. However, the gain in performance came with the cost of overfitting starting after the third epoch (**Appendix 4.3.3**), with a significant gap between train and test F1 macro scores (0.953 vs 0.702).

4.4 Overfitting Reduction Approaches: To address the overfitting observed after tuning the Xception model, four new setups were created based on the tunned model with the following changes (**Appendix 4.4.1**): increasing the dropout rate, adding L2 regularization using the keras l2 function with different lambda values, and unfreezing the last ten layers of the base architecture. Increasing dropout helps by turning off more neurons during training, while L2 regularization penalizes large weights (stronger for larger lambdas). Unfreezing layers introduces a hybrid transfer learning approach by mixing frozen and trainable layers. Although it could increase overfitting, this was tested to help the model better adapt to our dataset. Setup 1 and Setup 2 included only the first change, Setup 3 included both the first and second changes, and Setup 4 included all changes suggested. After training, all setups except Setup 3 showed signs of overfitting with early plateauing of validation loss (**Appendix 4.4.2**). Setup 3 did not converge after fifty epochs, and although it did not overfit, it was discarded as retraining with more epochs would be needed to understand its full behaviour. Setup 4 achieved the highest F1 macro score on test dataset so far (0.733) (**Appendix 4.4.3**). Due to the failed attempt to reduce overfitting, two new setups (Setups 5 and 6) were defined by applying the min_delta parameter in EarlyStopping (to avoid negligible validation loss improvements that might lead to overfitting), a more aggressive exponential decay factor (to reduce learning rate faster during training), and a reduced learning rate from 0.1 to 0.01 to allow for more stable fine-tuning. These changes were applied to Setup 3 and Setup 4, resulting in Setups 5 and 6, respectively (**Appendix 4.4.4**). After training, both setups converged and showed a total reduction of overfitting (**Appendix 4.4.5**), with training and validation losses staying close together across epochs and no plateauing of validation loss. However, the F1 macro scores on the test set decreased (0.498 and 0.592, respectively). (**Appendix 4.4.6**).

4.5 Final Model Selection and Error Analysis: Setup 4 was selected as the final model for the project, achieving the highest test F1 macro score (0.733). Despite some remaining overfitting, its superior overall performance justified its selection. Analyzing the confusion matrix (**Appendix 4.5.1**) for the final model reveals that most predictions fall along the diagonal, indicating strong classification performance across families. Out of 1647 test samples, 1261 were correctly classified (76.56%), while 386 predictions (23.44%) were incorrect (**Appendix 4.5.2**). The top ten most misclassified classes (**Appendix 4.5.3**) were dominated by chordata families, including *Carcharhinidae* and *Salamandridae* (each with 13 misclassifications), followed by *Cercopithecidae* (8), *Bovidae* (7), *Laridae* (7), *Bufonidae* (7), *Anatidae* (6), *Bucerotidae* (6), and *Ambystomatidae* (6). Given that these classes had a relatively high number of training images (**Appendix 4.5.4**) compared to the dataset average (60 images), the main source of misclassification does not seem to be related to class imbalance. To better understand these errors, we examined each of the top 10 most misclassified classes and counted how often their incorrect predictions were assigned to other classes, with the analysis focused on the top two. (**Appendix 4.5.5**). For *Salamandridae*, most misclassifications occurred with other *salamander* families such as *Plethodontidae* and *Ambystomatidae*, likely due to similar body shapes and habitats. For *Carcharhinidae*, most of the misclassifications were with aquatic animals like *Sphyrnidae* (hammerhead sharks) and *Delphinidae* (dolphins), explained by underwater conditions where low resolution and lighting obscure distinguishing features. These findings suggest that the model struggles with families sharing subtle morphological differences and similar environmental contexts. To better illustrate these errors, two wrongly classified images for each of the top 10 misclassified classes were generated. Two examples involved *Carcharhinidae* images classified as *Sphyrnidae* and *Delphinidae*, respectively (**Appendix 4.5.6**): the first due to shared body structures, and the second due to the animal being less visible. Additional examples of misclassifications caused by background dominance, subtle morphological similarities, and environmental influence are shown in **Appendix 4.5.7 to 4.5.9**. It is important to highlight that all images were normalized to the [0, 1] range, while some pre-trained models like Xception were originally trained with inputs normalized to the [-1, 1] range, a mismatch that may have impacted model performance on visually similar classes.

5. Conclusion

This project consisted of the development of a CNN model capable of predicting the family of a given species based on a dataset of thousands of rare species images sourced from the Encyclopedia of Life. After initial data exploration, several problems were identified: family and phylum imbalances with many underrepresented families, existence of non-RGB images, and the presence of noise label images (images not correctly representative of the family). To tackle these issues, two different outlier removal strategies were tested. However, due to computational and time limitations, manual removal was selected instead of the confident learning approach proposed by Northcutt et al. 2022. To address class imbalance, the method suggested by Langenkämper et al. 2018 was adapted to our case. During the modelling stage, different preprocessing strategies and pre-trained architectures were explored, with the combination of no augmentation, outlier removal, and oversampling applied to the Xception architecture being the winning strategy. To further improve the best model obtained, we implemented Bayesian Optimization following the approach suggested by Vo et al. 2023. Despite performance improvements, the tuned model showed signs of overfitting, leading to the development of a total of six new setups. The first four setups were unsuccessful in reducing overfitting but resulted in the discovery of the best-performing model, using a hybrid transfer learning approach. The two additional setups tested were successful in reducing overfitting but achieved significantly lower performance.

As expected, the best-performing model (corresponding to Setup 4) was not a model built from scratch but one that utilized a pre-trained architecture on the ImageNet dataset combined with a hybrid transfer learning approach (following the method proposed by Ferreira et al. 2018). Some limitations of this project should be highlighted. First, no specific strategy was applied to deal with near-duplicate images (images showing the same animal with slight variations, captured in quick succession, **Appendix 5.1**). Although some near-duplicates were likely removed during the manual outlier removal process, others may have remained, possibly introducing bias by overrepresenting specific visual patterns. Second, the oversampling method used may have contributed to the model's overfitting, since augmentations could have been applied to near-duplicate images. Finally, the selection of manual outlier removal instead of the confident learning approach makes the process more prone to human error and less scalable for future applications to larger datasets.

6. Future work

Considering the project’s limitations, a few improvements could be explored to enhance model performance and scalability. First, by expanding the use of transfer learning methods: while feature extraction was the primary approach used throughout the project, the hybrid method was applied only once (yielding the best-performing model), and fine-tuning was never explored. Additionally, including a dedicated step to evaluate the best transfer learning approach during the modelling stage could be beneficial. Second, improving the oversampling method used to address class imbalance: introducing different transformations, such as background changes for underrepresented families, could help reduce overfitting. Finally, dedicating more time and computational resources to implement the confident learning approach. In this project, a simple base model using ResNet50 as the pre-trained architecture was trained to obtain out-of-sample predictions and self-confidence values, using only 12 epochs. Moreover, exploration with a more robust base model and an increased number of epochs would likely have resulted in more accurate and reliable self-confidence values. Applying this methodology to remove images flagged as low-confidence errors per class would allow the definition of a scalable strategy for outlier removal, making the process less dependent on the dataset size.

Bibliography

1. Chollet, François. *Xception: Deep Learning with Depthwise Separable Convolutions*. 4 Apr. 2017.
2. Ferreira, C. A., Melo, T., Sousa, P., Meyer, M. I., Shakibapour, E., Costa, P., & Campilho, A. (2018). *Classification of breast cancer histology images through transfer learning using a pre-trained Inception ResNet V2*. In A. Campilho, F. Karray, & R. R. Janaqi (Eds.), *Image Analysis and Recognition (ICIAR 2018), Lecture Notes in Computer Science* (Vol. 10882, pp. 763–770). Springer. https://doi.org/10.1007/978-3-319-93000-8_86.
3. He, Kaiming, et al. "Identity Mappings in Deep Residual Networks." *ArXiv:1603.05027 [Cs]*, 25 July 2016, arxiv.org/abs/1603.05027.
4. Hu, Junjie, et al. "Number of Epochs of Each Model and Hyperband's Classification Performance." *2021 2nd International Seminar on Artificial Intelligence, Networking and Information Technology (AINIT)*, Oct. 2021, pp. 500–503, ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9725112, https://doi.org/10.1109/ainit54228.2021.00102.
5. Langenkämper, Daniel, et al. "Strategies for Tackling the Class Imbalance Problem in Marine Image Classification." *Lecture Notes in Computer Science*, 19 Dec. 2018, pp. 26–36, https://doi.org/10.1007/978-3-030-05792-3_3.
6. Northcutt, Curtis G., et al. "Confident Learning: Estimating Uncertainty in Dataset Labels." *ArXiv:1911.00068 [Cs, Stat]*, 21 Aug. 2022, arxiv.org/abs/1911.00068.
7. Kassani, S. H., Kassani, P. H., Wesolowski, M. J., Schneider, K. A., & Deters, R. (2019). *Breast cancer diagnosis with transfer learning and global pooling*. In 2019 International Conference on Information and Communication Technology Convergence (ICTC) (pp. 1231–1236). IEEE. https://doi.org/10.1109/ICTC46691.2019.8939878
8. Shin, Minchul, et al. "Baseline CNN Structure Analysis for Facial Expression Recognition." *ArXiv (Cornell University)*, 1 Aug. 2016, pp. 724–729, ieeexplore.ieee.org/document/7745199, https://doi.org/10.1109/roman.2016.7745199.

9. Simonyan, Karen, and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition.” *ArXiv.org*, 10 Apr. 2015, arxiv.org/abs/1409.1556.

10. Snoek, Jasper, et al. “Practical Bayesian Optimization of Machine Learning Algorithms.” *ArXiv:1206.2944 [Cs, Stat]*, 29 Aug. 2012, arxiv.org/abs/1206.2944.

11. Szegedy, Christian, et al. “Rethinking the Inception Architecture for Computer Vision.” *ArXiv.org*, 2015, arxiv.org/abs/1512.00567.

12. Vo, Hoang-Tu, et al. “An Approach to Hyperparameter Tuning in Transfer Learning for Driver Drowsiness Detection Based on Bayesian Optimization and Random Search.” *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 14, no. 4, 2023, thesai.org/Publications/ViewPaper?Volume=14&Issue=4&Code=IJACSA&SerialNo=92, https://doi.org/10.14569/IJACSA.2023.0140492.

Appendix 1. Introduction

Appendix 1.1 How the notebooks developed complement each section of the report

- Folders/Notebooks inside the Notebooks Directory

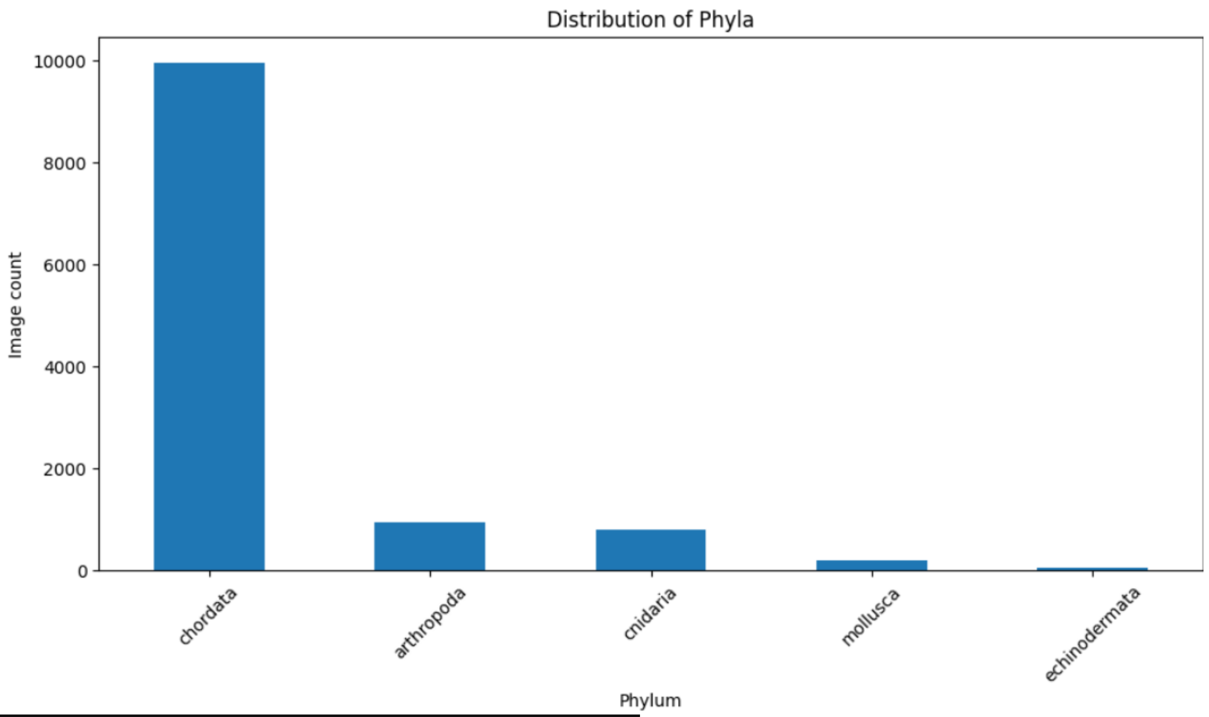
- How each folder/file inside the Notebooks Directory supports a section inside the report

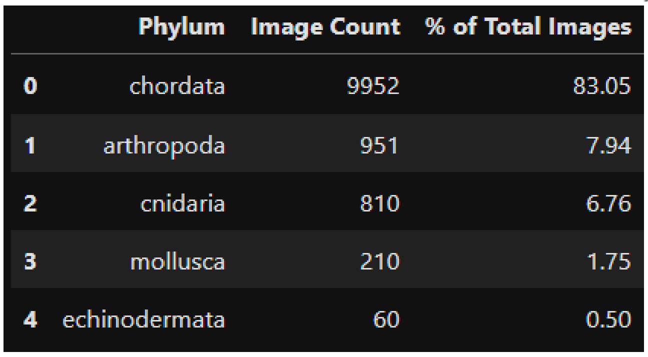
Report Section	**Corresponding Folders/Notebooks**
2. Exploratory Data Analysis and 3. Preprocessing	**EDA_Preprocessing.ipynb**
4.1 Data Preprocessing Strategy	**Baseline model** and **VGG16** folders with corresponding notebooks
4.2 Model Architecture Selection	**Other architectures** folder with corresponding notebooks
4.3 Hyperparameter Tunning	**DL_Xception_HyperparameterTuning_OverfittingAnalysis_FinalSelection.ipynb**
4.4 Overfitting Reduction Approaches	**DL_Xception_HyperparameterTuning_OverfittingAnalysis_FinalSelection.ipynb**
4.5 Final Model Selection and Error Analysis	**DL_Xception_HyperparameterTuning_OverfittingAnalysis_FinalSelection.ipynb**
Utilized throughout the report as supporting tools	**Library/utils.py**, **model_logs** and **Model Evaluation.ipynb**

Appendix 2. Exploratory Data Analysis

Appendix 2.1 Class imbalance in phylum

- **Image Distribution Across Phyla**

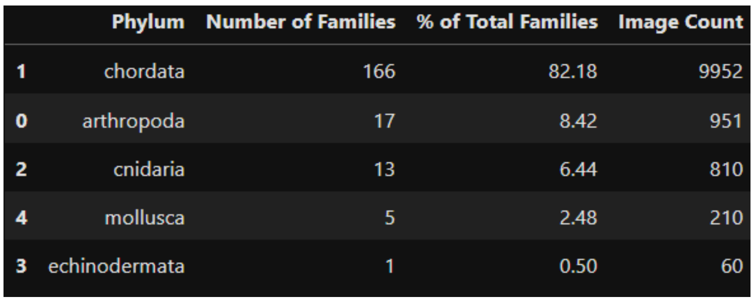




	Phylum	Image Count	% of Total Images
0	chordata	9952	83.05
1	arthropoda	951	7.94
2	cnidaria	810	6.76
3	mollusca	210	1.75
4	echinodermata	60	0.50

Echinodermata only has 60 images while chordata has 9.952 families.

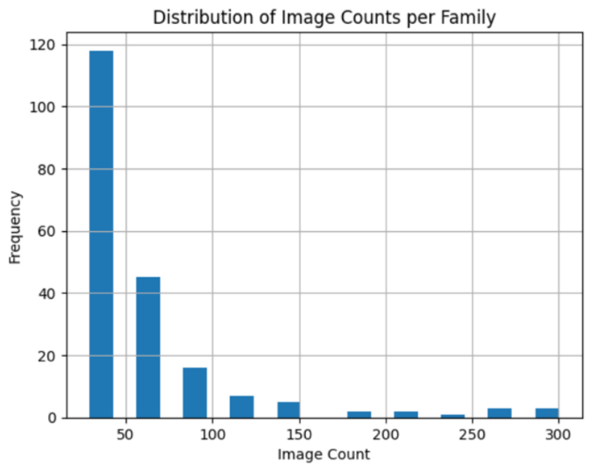
- **Family Distribution by Phylum**



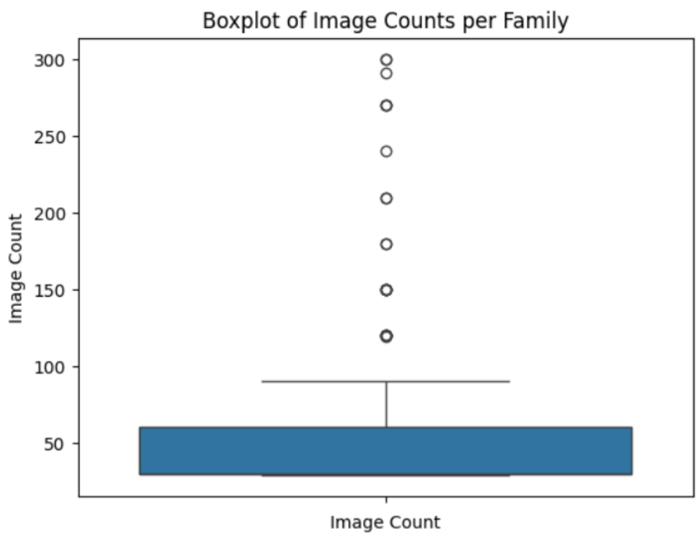
	Phylum	Number of Families	% of Total Families	Image Count
1	chordata	166	82.18	9952
0	arthropoda	17	8.42	951
2	cnidaria	13	6.44	810
4	mollusca	5	2.48	210
3	echinodermata	1	0.50	60

Echinodermata only has 1 family while chordata has 166 families.

APPENDIX 2.2. Class imbalance in family

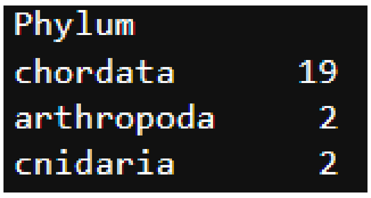


x-axis: how many images a family has
y-axis: how many families have that many images



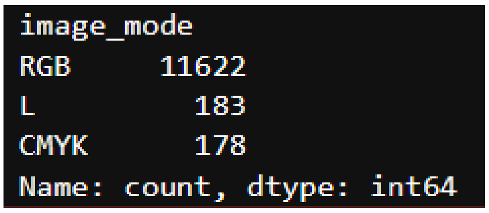
Family	Phylum	Image Count	% of Total Images	% within Phylum
cercopithecidae	chordata	300	2.50	3.01
dactyloidae	chordata	300	2.50	3.01
formicidae	arthropoda	291	2.43	30.60
plethodontidae	chordata	270	2.25	2.71
carcharhinidae	chordata	270	2.25	2.71
salamandridae	chordata	270	2.25	2.71
bovidae	chordata	240	2.00	2.41
bucerotidae	chordata	210	1.75	2.11
acroporidae	cnidaria	210	1.75	25.93
anatidae	chordata	180	1.50	1.81
diomedeidae	chordata	180	1.50	1.81
atelidae	chordata	150	1.25	1.51
laridae	chordata	150	1.25	1.51
apidae	arthropoda	150	1.25	15.77
bufonidae	chordata	150	1.25	1.51
dasyatidae	chordata	150	1.25	1.51

delphinidae	chordata	120	1.00	1.21
agariciidae	cnidaria	120	1.00	14.81
callitrichidae	chordata	120	1.00	1.21
psittacidae	chordata	120	1.00	1.21
spheniscidae	chordata	120	1.00	1.21
iguanidae	chordata	120	1.00	1.21
accipitridae	chordata	119	0.99	1.20



```
Phylum
chordata       19
arthropoda      2
cnidaria        2
```

APPENDIX 2.3. Color mode variation



```
image_mode
RGB     11622
L         183
CMYK      178
Name: count, dtype: int64
```

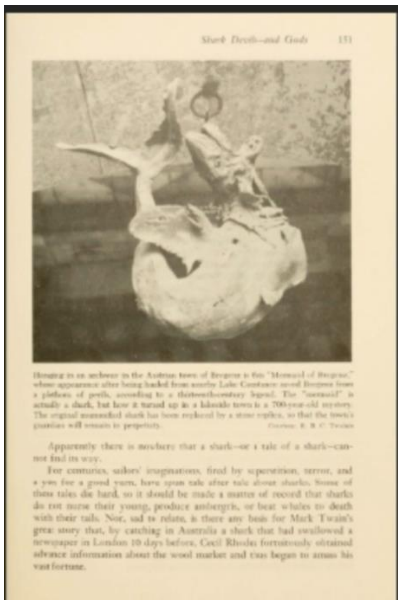
Appendix 3. Preprocessing

APPENDIX 3.1. Examples of Removed Outliers

- Skulls / Skeletons / Body Parts

Example found in multiple phylum_families

- Paper / Diagram where the animal is not clearly present



Shark Devils—and Gods 151

Hanging in an archway in the Austrian town of Bregenz is this "Mermaid of Bregenz," whose appearance after being hauled from nearby Lake Constance saved Bregenz from a plethora of perils, according to a thirteenth-century legend. The "mermaid" is actually a shark, but how it turned up in a lakeside town is a 700-year-old mystery. The original mummified shark has been replaced by a stone replica, so that the town's guardian will remain in perpetuity. Courtesy E. R. C. Twain

Apparently there is nowhere that a shark—or a tale of a shark—cannot find its way.

For centuries, sailors' imaginations, fired by superstition, terror, and a yen for a good yarn, have spun tale after tale about sharks. Some of these tales die hard, so it should be made a matter of record that sharks do not nurse their young, produce ambergris, or beat whales to death with their tails. Nor, sad to relate, is there any basis for Mark Twain's great story that, by catching in Australia a shark that had swallowed a newspaper in London 10 days before, Cecil Rhodes fortuitously obtained advance information about the wool market and thus began to amass his vast fortune.

Example found in *chordata_lamnidae*

- Unrelated content not showing animal

Example found in *chordata_leporidae*

- Animal not clearly visible / low resolution / extremely blurry / dark

Example found in *chordata_pangasiidae*

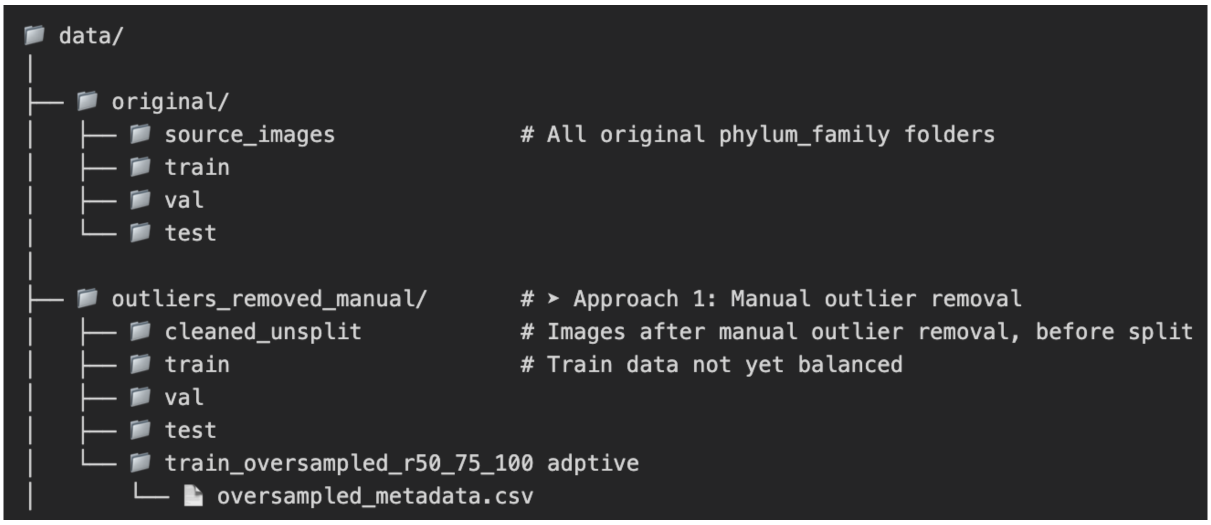
APPENDIX 3.2. Oversampling Formula from Langenkämper et al. (2019)

$$r_{50,75,100} : s(l) = \begin{cases} \frac{1}{2}s(l_{\mathrm{maj}}) & \text{if } s(l) \leq \frac{1}{4}s(l_{\mathrm{maj}}) \\ \frac{3}{4}s(l_{\mathrm{maj}}) & \text{if } \frac{1}{4}s(l_{\mathrm{maj}}) < M_k \leq \frac{1}{2}s(l_{\mathrm{maj}}) \\ s(l_{\mathrm{maj}}) & \text{else} \end{cases}$$

Please consider M_k= s(l)

Appendix 4. Modeling

Appendix 4.0.1 - Dataset structure used to support different preprocessing approaches



```
📁 data/
│
├── 📁 original/
│   ├── 📁 source_images            # All original phylum_family folders
│   ├── 📁 train
│   ├── 📁 val
│   └── 📁 test
│
├── 📁 outliers_removed_manual/      # ➤ Approach 1: Manual outlier removal
│   ├── 📁 cleaned_unsplit           # Images after manual outlier removal, before split
│   ├── 📁 train                     # Train data not yet balanced
│   ├── 📁 val
│   ├── 📁 test
│   └── 📁 train_oversampled_r50_75_100 adptive
│       └── 📄 oversampled_metadata.csv
```

Folder structure used to define the different datasets obtained. This structure allowed us to test different preprocessing techniques. For example, if we wanted to train our model after manual outlier removal with oversampling, we would need to use the following:

- **Train**: outliers_removed_manual/train_oversampled_r50_75_100_adaptive
- **Validation**: outliers_removed_manual/val
- **Test**: outliers_removed_manual/test

4. 1 Data Preprocessing strategy – baseline model

Appendix 4.1.1 – Baseline model architecture (based on Tang's architecture)

For our baseline model, we adopted a CNN architecture based on Tang's model, originally proposed for facial recognition. Tang's structure was highlighted in Shin et al. 2016 as one of the most efficient and lightweight convolutional networks, achieving competitive predictive power.

In our implementation, we retained the fundamental design principles of Tang's original architecture, including the small kernel sizes and alternating convolution–pooling structure. However, we made two significant adjustments to better accommodate our dataset of high-resolution species images. First, rather than flattening the convolutional output (an operation that would result in an excessively large number of parameters and likely cause overfitting) we introduced a GlobalAveragePooling2D layer. This layer condenses each feature map to a single representative value, reducing the number of trainable parameters while preserving the most salient spatial features.

Second, unlike the original Tang model, which uses a constant number of filters (32) across all convolutional layers, we progressively increased the number of filters at each convolutional stage (32 → 64 → 128 → 256). The deeper feature hierarchy allows the network to better capture inter-class differences in our species classification task, which would be less feasible with the limited representational power of the original architecture designed for 48×48 facial images.

Finally, for our classifier head, we introduced a dense layer with 512 neurons and a dropout layer with a rate of 0.5 to help mitigate overfitting. This approach was not originally employed in Tang's model, as his model ends with a single large dense layer (3072 units).

- Callbacks used (imported from Library.utils.py)

1) **ModelCheckpoint**:
Automatically saves the model weights corresponding to the lowest validation loss, ensuring the best-performing model is preserved regardless of later performance fluctuations.

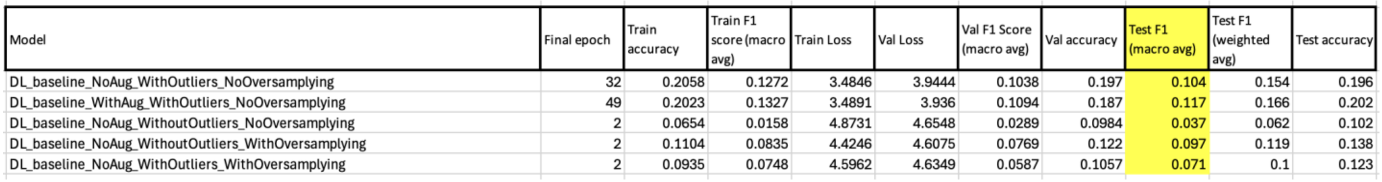
2) **CSVLogger**:
Records training metrics (e.g., loss and accuracy per epoch) into a CSV file.

3) **LearningRateScheduler**:
Applies an **exponential decay** to the learning rate across epochs. It allows:
    - Larger weight updates during the initial training phase to make learning faster.
    - Smaller adjustments in later epochs to support convergence and prevent overshooting.
    - The decay is controlled by a constant factor (e.g., 0.95), which multiplies the learning rate at each epoch.

4) **EarlyStopping**:
Monitors validation loss and stops training if no improvement is observed for a given consecutive number of epochs. It also restores the weights from the epoch with the best validation loss to ensure optimal performance.

All logs and checkpoints are saved in a directory with a timestamp for proper version tracking.

Appendix 4.1.2 – Performance Results of Baseline Model Across Preprocessing Combinations



Model	Final epoch	Train accuracy	Train F1 score (macro avg)	Train Loss	Val Loss	Val F1 Score (macro avg)	Val accuracy	Test F1 (macro avg)	Test F1 (weighted avg)	Test accuracy
DL_baseline_NoAug_WithOutliers_NoOversamplying	32	0.2058	0.1272	3.4846	3.9444	0.1038	0.197	0.104	0.154	0.196
DL_baseline_WithAug_WithOutliers_NoOversamplying	49	0.2023	0.1327	3.4891	3.936	0.1094	0.187	0.117	0.166	0.202
DL_baseline_NoAug_WithoutOutliers_NoOversamplying	2	0.0654	0.0158	4.8731	4.6548	0.0289	0.0984	0.037	0.062	0.102
DL_baseline_NoAug_WithoutOutliers_WithOversamplying	2	0.1104	0.0835	4.4246	4.6075	0.0769	0.122	0.097	0.119	0.138
DL_baseline_NoAug_WithOutliers_WithOversamplying	2	0.0935	0.0748	4.5962	4.6349	0.0587	0.1057	0.071	0.1	0.123

4. 1 Data Preprocessing strategy – VGG16 model

Appendix 4.1.3: Benchmark model architecture using VGG16 as pre trained architecture

To adapt VGG16 to our specific classification task, we excluded the original fully connected layers (include_top=False) and froze the pre-trained architecture layers. We then applied a custom classification head based on Ferreira et al. 2018 and Kassani et al. 2019.

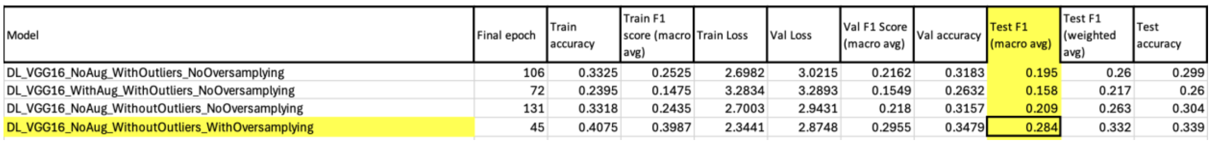
Following this rationale, we introduced a Global Average Pooling (GAP) layer immediately after the final convolutional block of VGG16. The GAP layer condenses each feature map to a single value, significantly reducing the number of trainable parameters while preserving the most significant information. The pooled features are then passed through two fully connected layers with 256 neurons each, activated with ReLU to introduce non-linearity.
Additionally, to prevent overfitting we included dropout layers with a rate of 0.4 after each dense layer. Finally, the network outputs class probabilities through a final dense layer using softmax activation.

Appendix 4.1.4 - Decision Table for Preprocessing Steps

Steps	Decision	Justification (based on Metrics Notebook and Appendix 4.1.5)
Applying Augmentation?	Between: DL_VGG16_**NoAug**_WithOutliers_NoOversamplying and DL_VGG16_**WithAug**_WithOutliers_NoOversamplying, We decided to select: DL_VGG16_**NoAug**_WithOutliers_NoOversamplying	The model without augmentation achieved a significantly higher test F1 macro score (0.195 vs. 0.158) and higher test accuracy (0.299 vs. 0.260). Moreover, it showed lower validation loss (3.02 vs. 3.29). While the val/train loss ratio is slightly higher (1.12 vs. 1), indicating a greater risk of overfitting, the performance gains justify the trade-off, which is why we select the model without augmentation.
Applying Outlier Removal?	Between: DL_VGG16_NoAug_**WithoutOutliers**_NoOversamplying and DL_VGG16_NoAug_**WithOutliers**_NoOversamplying We decided to select: DL_VGG16_NoAug_**WithoutOutliers**_NoOversamplying	Despite the slightly longer training time, removing outliers achieved a higher test F1 macro score (0.209 vs. 0.195), lower validation loss (2.94 vs. 3.021), higher test accuracy (0.304 vs 0.299), and a more favorable val/train loss ratio (1.09 vs. 1.12).
Applying Oversamplying?	Between: DL_VGG16_NoAug_WithoutOutliers_**NoOversamplying** and DL_VGG16_NoAug_WithOutliers_**WithOversamplying** We decided to select: DL_VGG16_NoAug_WithoutOutliers_**WithOversamplying**	The model with oversampling achieved a significantly higher test F1 macro score (0.284 vs. 0.209), and a higher test accuracy (0.339 vs. 0.304). Moreover, the model with oversampling also achieved a lower validation loss (2.87 vs. 2.94, respectively). While the model seems to starts to overfit after 9 epochs, the performance gains justify the trade-off, which is why we select the model with oversampling.

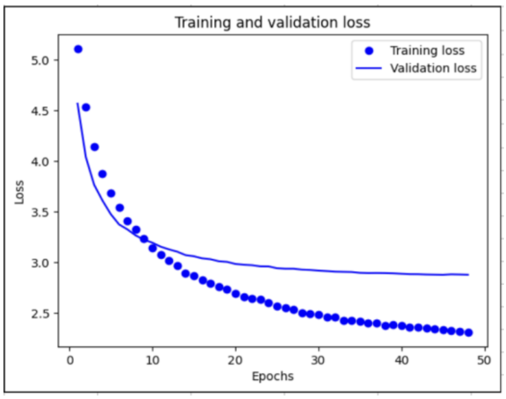
Appendix 4.1.5 - Model performance with selected preprocessing strategy (no augmentation, outlier removal and application of oversampling)

- Metrics comparison across different preprocessing strategies.



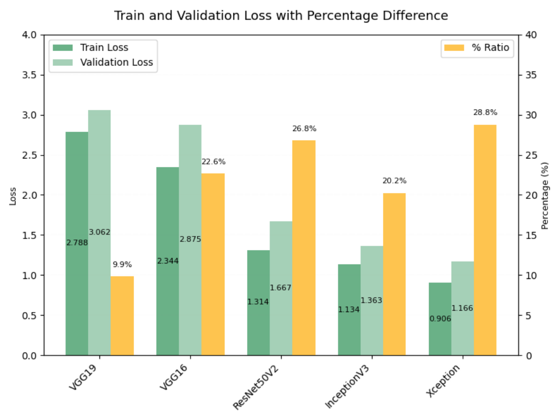
Model	Final epoch	Train accuracy	Train F1 score (macro avg)	Train Loss	Val Loss	Val F1 Score (macro avg)	Val accuracy	Test F1 (macro avg)	Test F1 (weighted avg)	Test accuracy
DL_VGG16_NoAug_WithOutliers_NoOversamplying	106	0.3325	0.2525	2.6982	3.0215	0.2162	0.3183	0.195	0.26	0.299
DL_VGG16_WithAug_WithOutliers_NoOversamplying	72	0.2395	0.1475	3.2834	3.2893	0.1549	0.2632	0.158	0.217	0.26
DL_VGG16_NoAug_WithoutOutliers_NoOversamplying	131	0.3318	0.2435	2.7003	2.9431	0.218	0.3157	0.209	0.263	0.304
DL_VGG16_NoAug_WithoutOutliers_WithOversamplying	45	0.4075	0.3987	2.3441	2.8748	0.2955	0.3479	0.284	0.332	0.339

- Train and validation loss plot for the selected preprocessing strategy

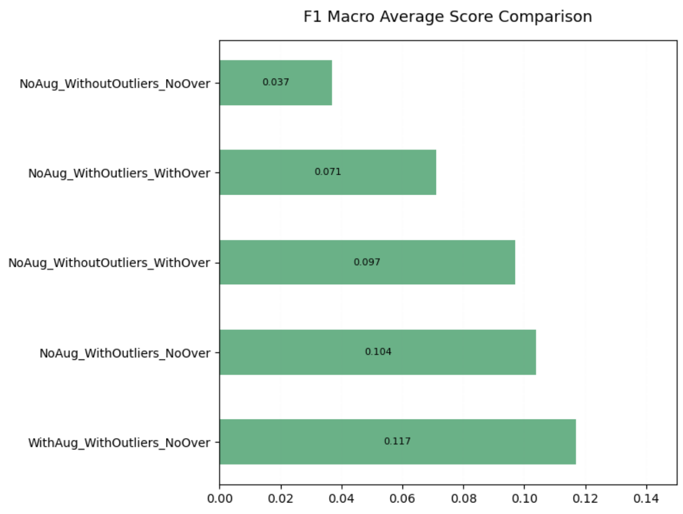


4. 2 Architecture selection – model comparison

Appendix 4.2.1 - Model Architecture Comparison Based on Test F1 Score and Overfitting

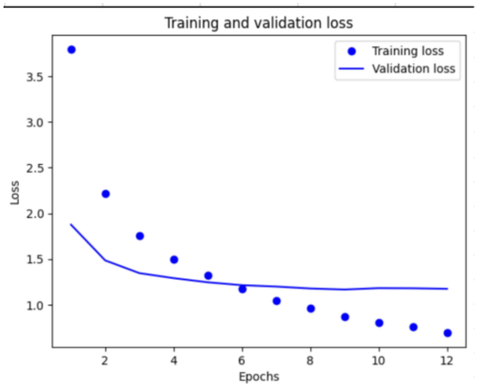


Model	Final epoch	Train accuracy	Train F1 score (macro avg)	Train Loss	Val Loss	Val F1 Score (macro avg)	Val accuracy	Test F1 (macro avg)	Test F1 (weighted avg)	Test accuracy
DL_VGG19_NoAug_WithoutOutliers_WithOversamplying	46	0.3247	0.3069	2.7877	3.0625	0.2553	0.3084	0.251	0.291	0.308
DL_InceptionV3_NoAug_WithoutOutliers_WithOversamplying	17	0.6683	0.6401	1.1336	1.363	0.6084	0.6527	0.596	0.632	0.645
DL_ResNet50V2_NoAug_WithoutOutliers_WithOversamplying	7	0.6451	0.624	1.3143	1.6666	0.5318	0.5811	0.556	0.581	0.591
DL_Xception_NoAug_WithoutOutliers_WithOversamplying	9	0.7392	0.7224	0.9059	1.1664	0.6201	0.6764	0.65	0.691	0.705

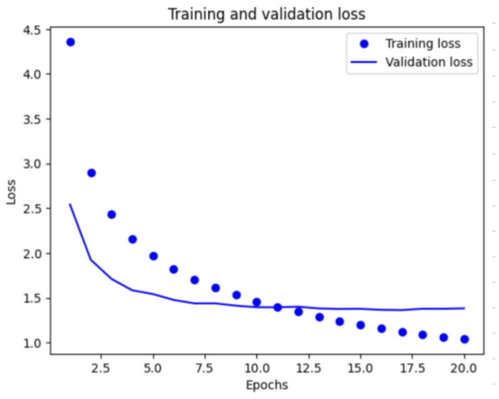


Appendix 4.2.2

1. Xception train and validation loss plot



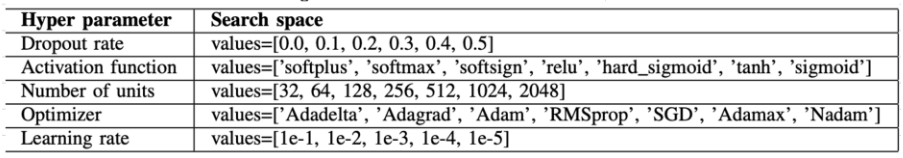
2. InceptionV3 train and validation loss plot



4.3 Hyperparameter Tunning Preprocessing strategy

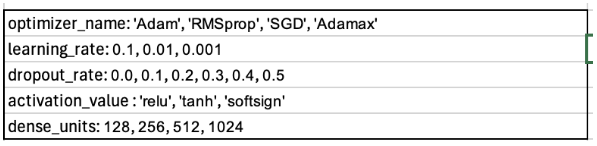
Appendix 4.3.1 – Hyperparameters and Search space used in the paper versus the ones used in our project

1. Suggested in the paper



Hyper parameter	Search space
Dropout rate	values=[0.0, 0.1, 0.2, 0.3, 0.4, 0.5]
Activation function	values=['softplus', 'softmax', 'softsign', 'relu', 'hard_sigmoid', 'tanh', 'sigmoid']
Number of units	values=[32, 64, 128, 256, 512, 1024, 2048]
Optimizer	values=['Adadelta', 'Adagrad', 'Adam', 'RMSprop', 'SGD', 'Adamax', 'Nadam']
Learning rate	values=[1e-1, 1e-2, 1e-3, 1e-4, 1e-5]

2. Used in the project



optimizer_name: 'Adam', 'RMSprop', 'SGD', 'Adamax'
learning_rate: 0.1, 0.01, 0.001
dropout_rate: 0.0, 0.1, 0.2, 0.3, 0.4, 0.5
activation_value : 'relu', 'tanh', 'softsign'
dense_units: 128, 256, 512, 1024

Appendix 4.3.2 – Best hypermeters values and model performance

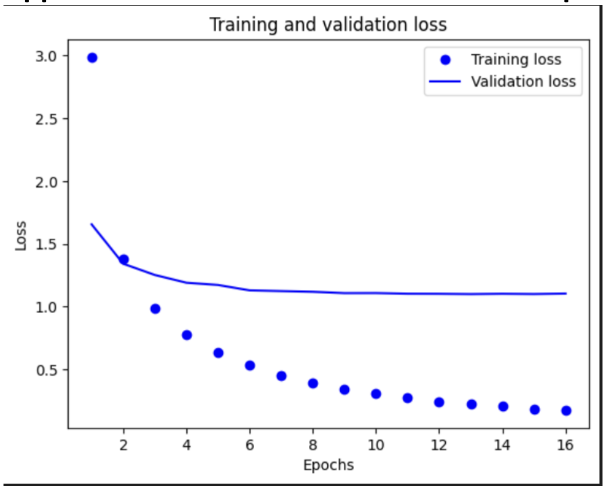
- Tunned model performance

Model	Final epoch	Train accuracy	Train F1 score (macro avg)	Train Loss	Val Loss	Val F1 Score (macro avg)	Val accuracy	Test F1 (macro avg)	Test F1 (weighted avg)	Test accuracy
DL_Xception_NoAug_WithoutOutliers_WithOversampling (Hyperparameter tun	13	0.9623	0.953	0.2251	1.0998	0.6907	0.7158	0.702	0.732	0.738

- Best hyperparameter values

Optimizer	SGD
Learning Rate	0.1
Dropout Rate	0.1
Activation Function	tanh
Dense Units	1024

Appendix 4.3.3 - Train and Validation loss plot for the tuned model



4.4 Overfitting Reduction Approaches

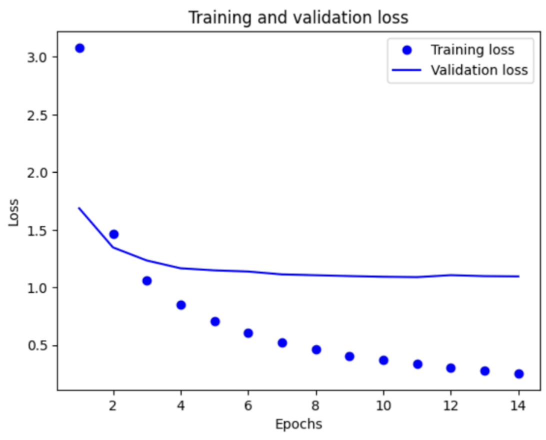
Appendix 4.4.1 – Four new setups to deal with overfitting

Setup	Dropout	L2 Reg	Unfreeze (last 10 layers)
1	0.3	None	No
2	0.4	None	No
3	0.4	1.00E-03	No
4	0.3	1.00E-04	Yes (last 10)

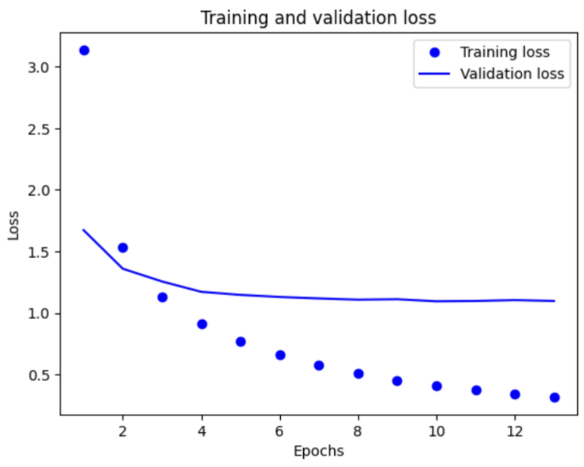
Setup 4 : Hybrid transfer learning + Fine tuning

Appendix 4.4.2 – Train and validation loss plots for the four new setups

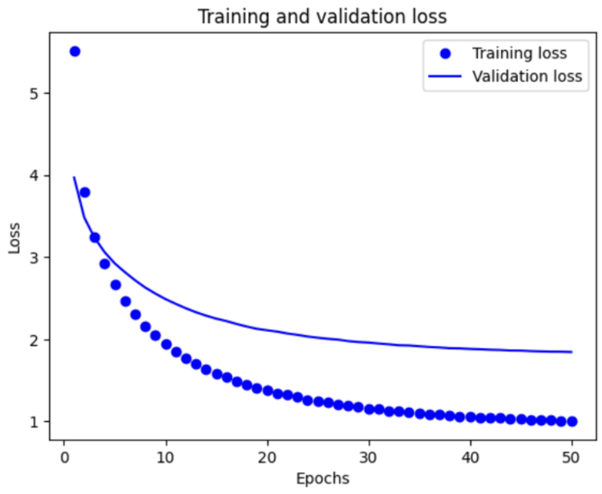
- Setup 1



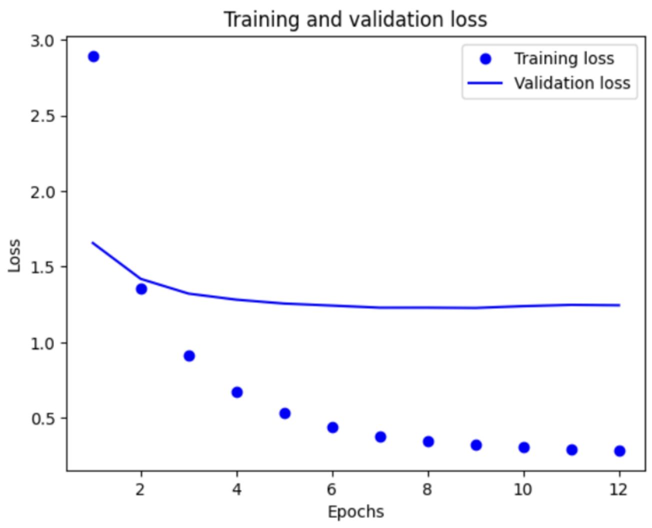
- Setup 2



- Setup 3



- Setup 4



Appendix 4.4.3 – Performance of four new setups to deal with overfitting

Model	Final epoch	Train accuracy	Train F1 score (macro avg)	Train Loss	Val Loss	Val F1 Score (macro avg)	Val accuracy	Test F1 (macro avg)	Test F1 (weighted avg)	Test accuracy
DL_Xception_ReduceOverfiting_Setup1	11	0.9232	0.9166	0.3413	1.0893	0.6975	0.7171	0.684	0.716	0.721
DL_Xception_ReduceOverfiting_Setup2	10	0.8989	0.893	0.4156	1.0956	0.6922	0.7152	0.688	0.72	0.727
DL_Xception_ReduceOverfiting_Setup3	50	0.9662	0.958	1.0055	1.8451	0.6886	0.7286	0.681	0.719	0.727
DL_Xception_ReduceOverfiting_Setup4	9	0.9931	0.983	0.326	1.2272	0.728	0.7523	0.733	0.761	0.766

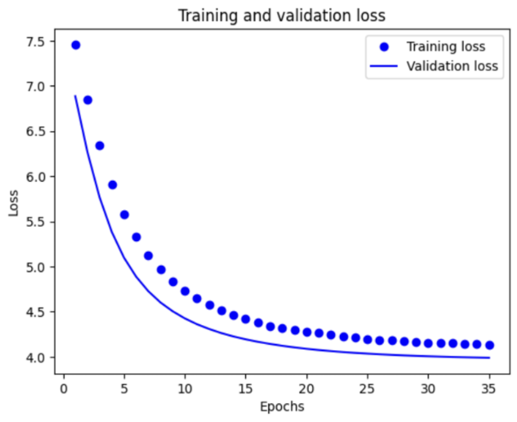
Appendix 4.4.4 – Last setups to deal with overfitting

Setup	Dropout	L2 Reg	Unfreeze (last 10 layers)	min_delta	Learning Rate	Exp_decay factor
5	0.4	1.00E-03	No	0.01	0.01	0.9
6	0.3	1.00E-04	Yes (last 10)	0.01	0.01	0.9

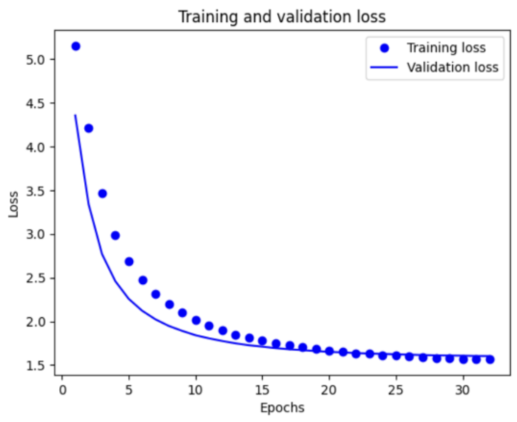
min_delta : 0.01 -> minimum treshold to what's considered an improvement

Appendix 4.4.5 - Train and validation loss plots for the setup 5 and 6

- Setup 5



- Setup 6

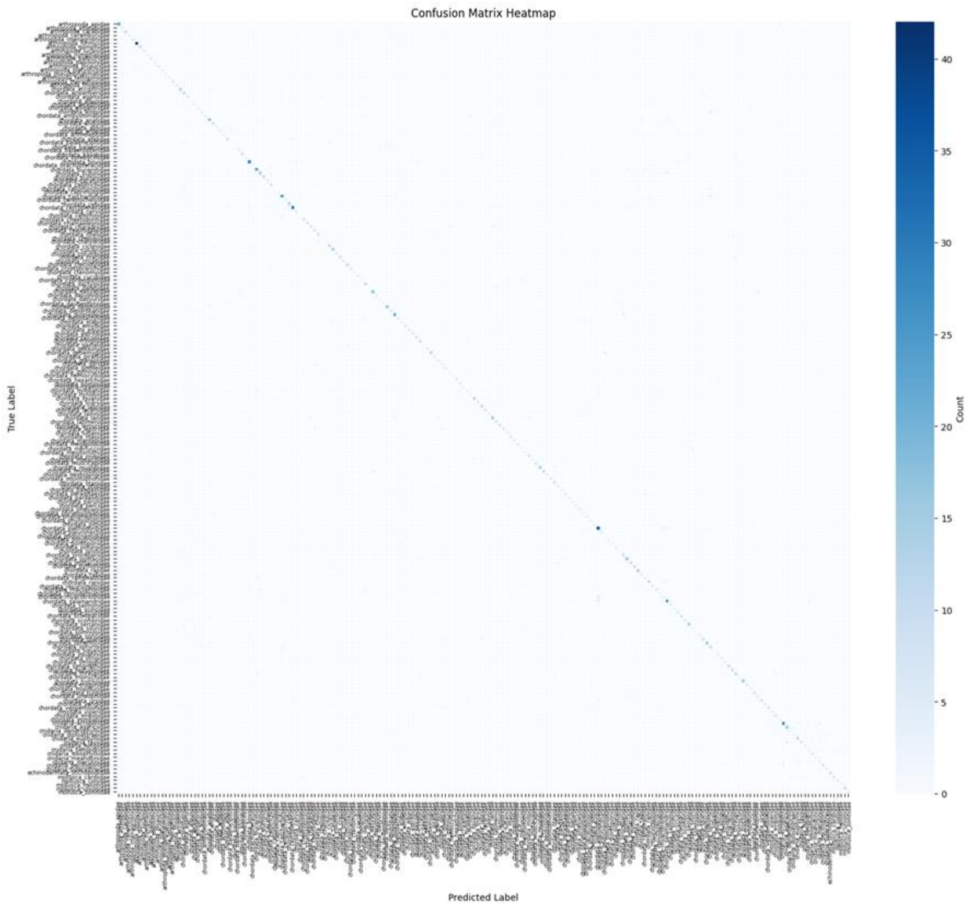


Appendix 4.4.6 – Performance of setups 5 and 6

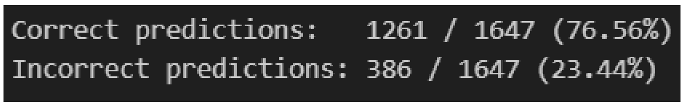
Model	Final epoch	Train accuracy	Train F1 score (macro avg)	Train Loss	Val Loss	Val F1 Score (macro avg)	Val accuracy	Test F1 (macro avg)	Test F1 (weighted avg)	Test accuracy
DL_Xception_ReduceOverfiting_Setup5	32	0.5733	0.5096	4.1543	3.9978	0.4825	0.5962	0.498	0.579	0.616
DL_Xception_ReduceOverfiting_Setup6	29	0.691	0.6571	1.5828	1.6083	0.5989	0.6642	0.592	0.639	0.659

4.5 Final Model Selection and Error Analysis

Appendix 4.5.1 - Confusion Matrix

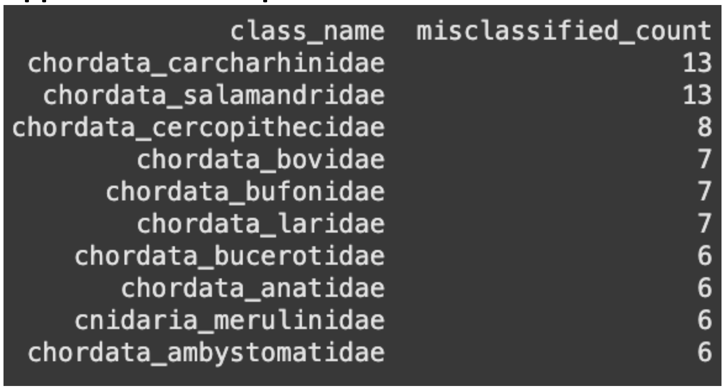


Appendix 4.5.2 - Correct and Incorrect Predictions



```
Correct predictions:   1261 / 1647 (76.56%)
Incorrect predictions: 386 / 1647 (23.44%)
```

Appendix 4.5.3 - Top 10 misclassified classes

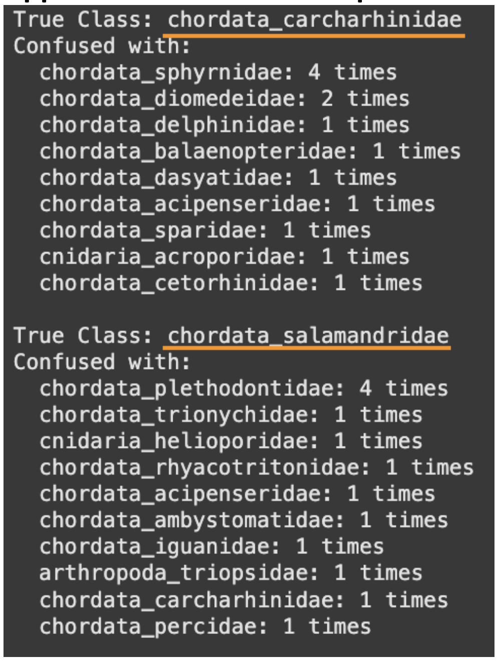


```
              class_name  misclassified_count
  chordata_carcharhinidae                  13
   chordata_salamandridae                  13
chordata_cercopithecidae                    8
         chordata_bovidae                   7
       chordata_bufonidae                   7
         chordata_laridae                   7
     chordata_bucerotidae                   6
        chordata_anatidae                   6
     cnidaria_merulinidae                   6
  chordata_ambystomatidae                   6
```

Appendix 4.5.4 - Count of the original images in the top 10 misclassified classes

chordata_carcharhinidae	252 images
chordata_salamandridae	262 images
chordata_cercopithecidae	256 images
chordata_bovidae	229 images
chordata_laridae	144 images
chordata_bufonidae	144 images
chordata_anatidae	156 images
cnidaria_merulinidae	59 images
chordata_bucerotidae	196 images
chordata_ambystomatidae	60 images

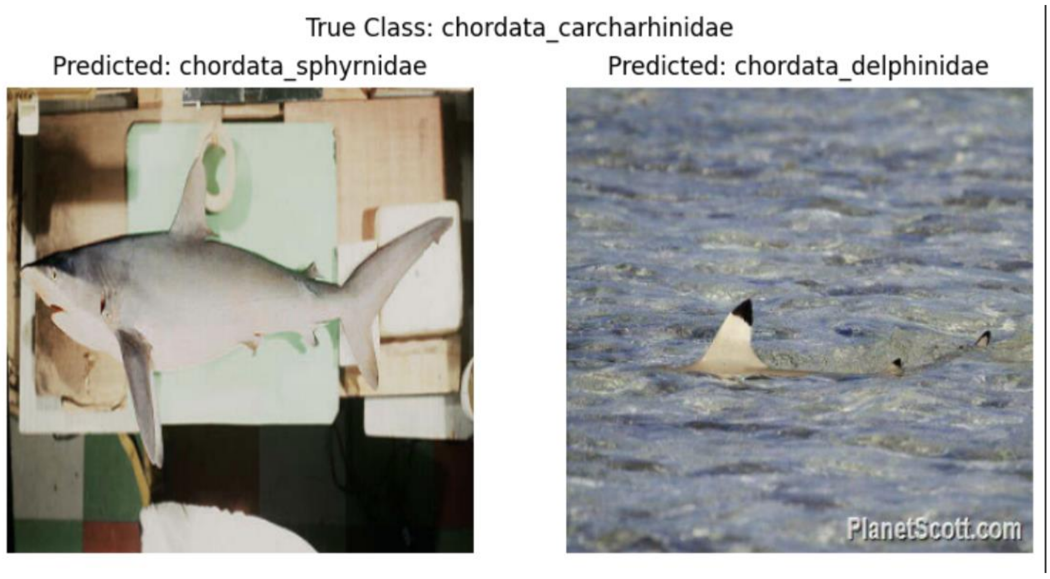
Appendix 4.5.5 - Most Frequent Misclassifications for Carcharhinidae and Salamandridae



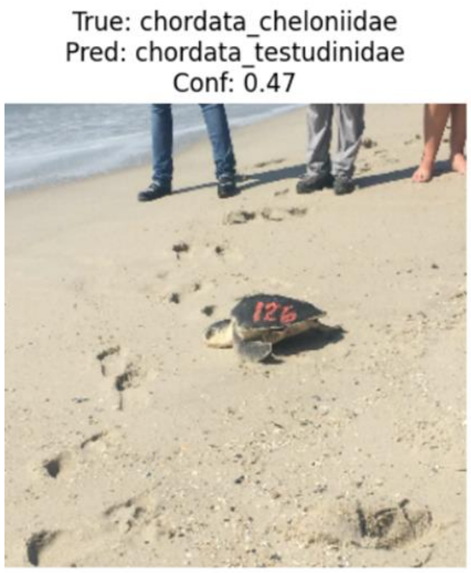
```
True Class: chordata_carcharhinidae
Confused with:
  chordata_sphyrnidae: 4 times
  chordata_diomedeidae: 2 times
  chordata_delphinidae: 1 times
  chordata_balaenopteridae: 1 times
  chordata_dasyatidae: 1 times
  chordata_acipenseridae: 1 times
  chordata_sparidae: 1 times
  cnidaria_acroporidae: 1 times
  chordata_cetorhinidae: 1 times

True Class: chordata_salamandridae
Confused with:
  chordata_plethodontidae: 4 times
  chordata_trionychidae: 1 times
  cnidaria_helioporidae: 1 times
  chordata_rhyacotritonidae: 1 times
  chordata_acipenseridae: 1 times
  chordata_ambystomatidae: 1 times
  chordata_iguanidae: 1 times
  arthropoda_triopsidae: 1 times
  chordata_carcharhinidae: 1 times
  chordata_percidae: 1 times
```

Appendix 4.5.6 - Missclassified images for *Chordata_ Carcharhinidae*


True Class: chordata_carcharhinidae
Predicted: chordata_sphyrnidae
Predicted: chordata_delphinidae
PlanetScott.com

Appendix 4.5.7 - Missclassified *Chordata_Cheloniidae*
Example of an image of a *Cheloniidae* (sea turtle) being classified as a *Testudinidae* (land tortoise), probably because both species share similar shell morphology, and the sandy beach setting made the aquatic context less evident.

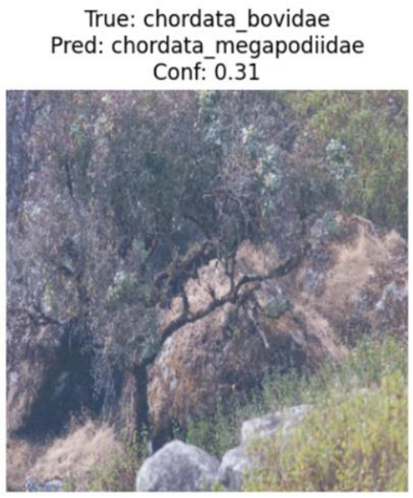


Appendix 4.5.8 - Missclassified *Chordata_Motacillidae*

Example of an image of a *Motacillidae* (wagtails and pipits) being predicted as *Iguanidae* (iguanas), likely due to the grassy environment making it difficult to distinguish small birds from reptiles

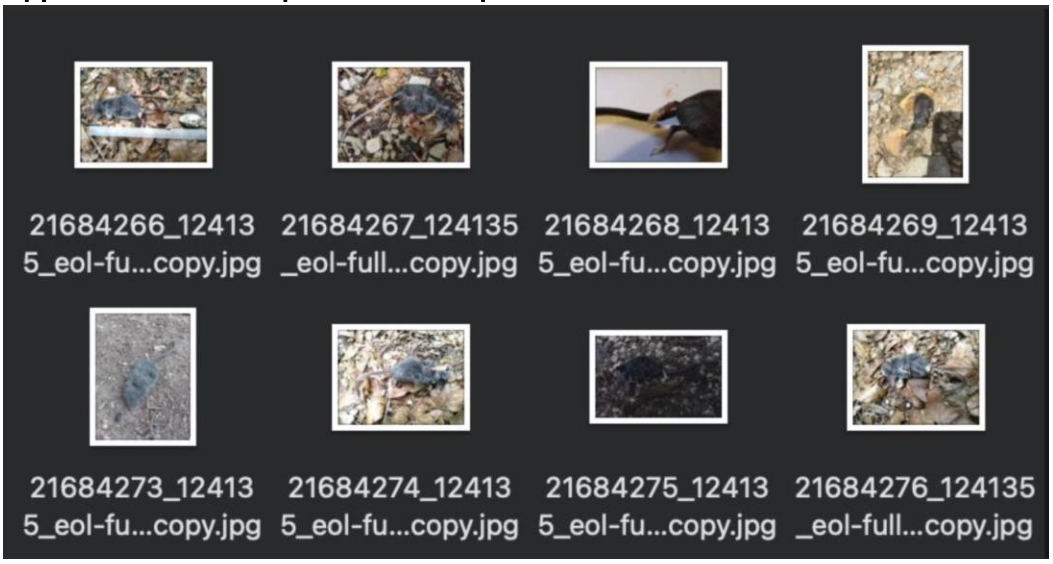
Appendix 4.5.9 - Missclassified ***Chordata_Bovidae***

Example of an image of *Bovidae* (e.g., cattle or antelopes) being misclassified as *Megapodiidae* (incubator birds). Here, dense vegetation and the small, less visible subject may have caused the model to focus more on background features than on the animal itself

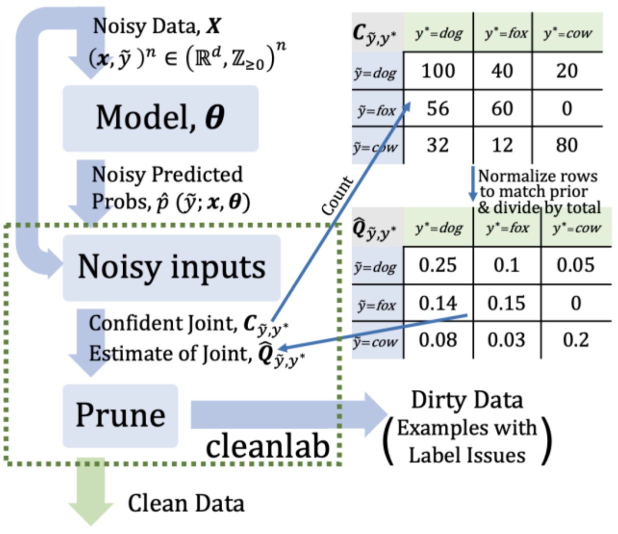


5. Conclusion

Appendix 5.1 – Examples of near duplicates


21684266_124135_eol-fu...copy.jpg
21684267_124135_eol-full...copy.jpg
21684268_124135_eol-fu...copy.jpg
21684269_124135_eol-fu...copy.jpg
21684273_124135_eol-fu...copy.jpg
21684274_124135_eol-fu...copy.jpg
21684275_124135_eol-fu...copy.jpg
21684276_124135_eol-full...copy.jpg

Annex A – Confident Learning approach


Noisy Data, $\boldsymbol{X}$
$(\boldsymbol{x}, \tilde{y})^n \in (\mathbb{R}^d, \mathbb{Z}_{\geq 0})^n$
Model, $\boldsymbol{\theta}$
Noisy Predicted Probs, $\hat{p}(\tilde{y}; \boldsymbol{x}, \boldsymbol{\theta})$
Noisy inputs
Confident Joint, $\boldsymbol{C}_{\tilde{y},y^*}$
Estimate of Joint, $\hat{\boldsymbol{Q}}_{\tilde{y},y^*}$
Prune
cleanlab
Clean Data
Dirty Data (Examples with Label Issues)
Count
Normalize rows to match prior & divide by total

$\boldsymbol{C}_{\tilde{y},y^*}$	y^*=dog	y^*=fox	y^*=cow
$\tilde{y}$=dog	100	40	20
$\tilde{y}$=fox	56	60	0
$\tilde{y}$=cow	32	12	80

$\hat{\boldsymbol{Q}}_{\tilde{y},y^*}$	y^*=dog	y^*=fox	y^*=cow
$\tilde{y}$=dog	0.25	0.1	0.05
$\tilde{y}$=fox	0.14	0.15	0
$\tilde{y}$=cow	0.08	0.03	0.2

Figure A1 – Confident Learning Approach.
[Adapted from (Northcutt et al. 2022)]

Contrary to other model-based approaches used to deal with noisy data, Northcutt et al. 2022 propose a data-based method: the confident learning methodology. As described in Northcutt et al. 2022, confident learning is a data-centric approach for identifying label errors by estimating the joint distribution between the noisy labels and the unknown true labels. The method follows a few key steps. First, a classifier that outputs a probability distribution is trained to generate predictions. A key requirement of this methodology is the usage of out-of-sample predicted probabilities, obtained through cross-validation, to ensure unbiased estimates of model uncertainty for each image. Second, the self-confidence of each image is computed as the predicted probability of the image belonging to its assigned label with low values pointing to possible errors. Third, the confident joint matrix is built using the predicted probabilities and the known labels. This matrix estimates how often each observed (noisy) label actually corresponds to each true class based on model predictions. Finally, likely label issues or outlier images are flagged using both the self-confidence and the confident joint matrix.

To integrate this methodology with Keras, the Cleanlab package was used, following the Cleanlab's documentation (Cleanlab Team, n.d.). Its compatibility with sklearn's API allowed us to apply sklearn-style cross-validation to generate out-of-sample predictions while using our Keras model, wrapped using Cleanlab's KerasWrapperModel class.

In our project, due to time and resource limitations, we tested a simplified version of this pipeline using a lightweight ResNet50 model pre-trained on ImageNet. The model was trained for only 12 epochs to generate out-of-sample predictions. Due to this limited training, the model performed poorly and showed very low confidence across most samples. As a result, it flagged nearly all images as potential label issues, since it was highly uncertain and unable to correctly distinguish between correct and incorrect labels.

Annex B– Google Colab Pro

Due to the lack of GPUs and overall computational power in the devices available to our group, we decided to use Google Colab Pro to run our models with GPU access. Because of slower hardware access, shorter session times, and limited GPU availability in the free tier, we purchased Google Colab Pro with an overall total cost of 45€ . This spending was justified as Google Colab Pro provided priority access to GPUs and allowed us to use the A100 GPU. The A100 was selected the most due to its superior memory, speed, and performance for deep learning workloads, with each training step during an epoch taking less than 140ms. However, using Google Colab Pro also presented some limitations:

- Execution order of cells were not present after session restarts.
- Risk of losing variables and runtime memory if the session disconnects or times out.
- Need to re-mount Google Drive frequently for saving/loading models.
- Sometimes GPU type would downgrade (e.g., from A100 to T4) depending on availability.

To tackle these issues we tried to save our models and training logs frequently inside the model_logs folder and used CSVLogger callback for metrics tracking.

Annex C – Types of Transfer Learning

1) Feature Extraction

In the feature extraction approach, the model configuration of a pre-trained network is used as a fixed feature extractor. All layers are frozen, meaning their weights remain unchanged during training, and only the newly added classification head is trained on the new dataset. Feature extraction is computationally efficient and works well when the dataset is small or similar to the source dataset the pre-trained architecture was trained on.

Reference paper where the approach is implemented: *Donahue, J., Jia, Y., Vinyals, O., Hoffman, J., Zhang, N., Tzeng, E., & Darrell, T. (2014). DeCAF: A deep convolutional activation feature for generic visual recognition. ICML.*

2) Fine Tunning

Fine-tuning involves unfreezing the entire network or a significant portion of it and continuing training on the training dataset. It allows the model to adjust the pre-trained features to the specific characteristics of our dataset. Fine-tuning generally requires more data and computational power but can result in superior performance if the domain shift between the training dataset and source dataset is significant.

Reference paper where the approach is implemented: *Yosinski, J., Clune, J., Bengio, Y., & Lipson, H. (2014). How transferable are features in deep neural networks? NeurIPS.*

3) Hybrid approach (Partial Fine-Tunning)

The hybrid approach combines elements of both feature extraction and fine-tuning. In this method, only a subset of the layers in the convolutional base are unfrozen and trained, while the earlier layers remain frozen.

Reference paper where the approach is implemented: Ferreira, C. A., Melo, T., Sousa, P., Meyer, M. I., Shakibapour, E., Costa, P., & Campilho, A. (2018). *Classification of breast cancer histology images through transfer learning using a pre-trained Inception ResNet V2*.

Annex D – Bayesian Optimization for Hyperparameter Tuning

Bayesian Optimization, as described by Snoek et al. 2012, distinguishes itself from other optimization methods by building a probabilistic model of the objective function and using this model to guide the search for optimal values while integrating out uncertainty.

Rather than relying on local gradient or Hessian approximations, it uses all available information from past evaluations to decide where to evaluate next. It involves making two main choices: selecting a prior over functions and defining an acquisition function. The first corresponds to the assumptions made about the function's shape and properties before we observe any training data. The second corresponds to the function that transforms the posterior distribution into an utility function that will determine the selection of next combination to experiment with. The posterior distribution is built after the model has learned from previous trials and builds a probabilistic view of where the objective function might be in the search space.