DEEP LEARNING PROJECT

Master in Data Science and Advanced Analytics

NOVA Information Management School

Universidade Nova de Lisboa

Rare Species Classification Based on Images

Group 17

André Sousa, 20240517

Francisco Pontes, 20211583

Isabella Costa, 20240685

Jéssica Cristas, 20240488

Tiago Castilho, 20240489

Spring Semester 2024-2025

TABLE OF CONTENTS

1. INTRODUCTION

The following report describes the process of building a deep learning model, with the goal of accurately classifying the family of rare species based on given images. Our dataset origins from the Encyclopedia of Life, and is composed of 11983 images, and a metadata file describing the kingdom, phylum and family that each image belongs to [Figure 1, Figure 2].
We will walk through every step of this project, from the preprocessing to the modelling efforts, evaluation of results and, finally, a reflection on the work done.

2. PREPROCESSING

2.1. Cleaning images from classes

Before doing any transformations or model training began, we realized that some images in the training set didn't accurately represent the species in question which led us to perform manual cleaning of the dataset to remove irrelevant or misleading images. Some examples of images we removed were maps, plain text images, random people or unrelated objects where the species in question were either absent or undistinguishable. This resulted in the removal of 382 images, only 3.2% of the data, which we deemed acceptable to not affect drastically the essence of the data, while improving quality. The images removed can be found at the following link.

2.2. Resizing and Batch Size

All images were resized initially to 224x224 and adapted according to evolving model needs. Despite reducing computational load and speeding up training, it has a drawback: image quality loss. This approach may lead to the loss of some detail in the images. However, this compromise was necessary to make training within the hardware and within batch memory feasible. A batch is the quantity of samples which are processed before the model's weights are updated. This way, we decided to use a batch size of 32, which is a typical trade-off between memory usage and training speed.

2.3. Data Augmentation

To increase the size of the training data and help the model generalize better, we did data augmentation on the images. We had to do this because the dataset was imbalanced and lacked diversity in some classes. Our augmentation pipeline included the following random transformations: horizontal and vertical flipping, rotation up to 20 degrees, zooming into images up to 10% and adjustments in brightness and contrast to a maximum factor of 20%.

These augmentations add real-world variability and avoid overfitting to the training data. We also shuffled the dataset during training which prevents the model from learning the order of the data and this way reduces overfitting. The randomness ensures that each batch is diverse which leads to better gradient updates. We used a shuffle buffer size of 1000 to ensure a higher degree of randomness.

After augmentation, we normalized pixel values using a rescaling layer, which scales pixel values from [0,255] to the range of [0,1]. This helps the neurol networks converge faster since it standardizes the input scale.

2.4. Tensor Flow Records

Since we had many samples per class, one of the main challenges for this project was the efficiency of the code, one of the ways to solve this was Tensor Flow Records. This is a binary data serialization format optimized for TensorFlow pipelines. By doing this, we gained efficiency in storage and loading of the data, since while using TensorFlow related functions, reading from these types of files is significantly faster than reading from images files like jpeg or png.

3. MODELLING

3.1. Class Weights

Our dataset contains 202 classes with significant imbalances. This way, besides doing data augmentation, we also applied class weights, assigning higher weights to underrepresented classes. This ensures that the model doesn't favor more dominant classes but also pays more attention to relevant features from the minority classes.

3.2. Model Training Setup and Data Preparation

Given the unbalanced nature of the dataset, we opted to focus on F1 Score with macro average as our main metric, to be considered the same when calculating F1 Score, since dominant classes may hide classification problems on minority classes when using metrics like accuracy. Because of this choice due to compatibility, the loss was made to Categorical Crossentropy.

3.3. Callbacks

During training, we used a set of callbacks API from Keras, such as **EarlyStopping**, **ModelCheckpoint.**

The **EarlyStopping** callback was used to stop the training process with different values during the development of the project. During the assessment of multiple models in the same runtime we decided to stop the model when validation loss was no longer decreasing.

The **ModelCheckpoint** callback was used to save the model with the highest validation macro F1-score to assess test data in case the model stops running beyond the best epoch.

3.4. Base Performance (simple CNN)

Our initial approach was to build a model from scratch, using Keras' sequential layers. We built a simple convolutional neural network, with two blocks of convolution, batch normalization, ReLU activation, and max pooling. This allowed the model to progressively extract and condense features from the input images. Unfortunately, the results did not coincide with our expectations, having only achieved 2.8% Accuracy, and 0.01% Macro F1-Score on the validation set. [**Table 3**]

3.5. Pre-Trained Model (base)

3.5.1. Transfer Learning

After seeing the disappointing results from the CNN built from scratch, another approach was implemented. We decided to analyze the performance of pre-trained models such as **MobileNetV2**, **DenseNet121** and **InceptionV3**, and the results were better. More info about the models in **Table 5**.

As we can see in **Table 3**, the model with the best result was **InceptionV3**. Knowing this, we opted to use it as our final model.

3.5.2. InceptionV3

The model with the best base performance was InceptionV3, from Keras API. This iteration of the architecture was introduced by Szegedy et al. (2015), with improvements from its predecessors on computational efficiency and model accuracy [**Figure 3**].

The overall architecture has an input layer that accepts images of size 299x299x3. It begins with convolution and pooling layers to extract low-level features, followed by specialized modules that process information at multiple scales. The network concludes with global average pooling and dense layer for classification.

3.6. Hyper-Parameter Tuning

For Hyper-Parameter Tuning we used Optuna (n.d.), that is a framework for automatic hyperparameter optimization. We gave 4 parameters to Optuna, learning rate, dropout rate, dense units and trainable layers, and coded him so that it would do twenty different configurations test and would compare the validation accuracy to determine which parameters were better for our dataset. After that we just needed to test the models with the parameters given by Optuna and after that fine tune it for our needs [**Table 6**].

3.7. Fine-Tuning

Fine-tuning was performed using InceptionV3 pre-trained on ImageNet. Initially, all layers in the convolutional base were frozen to preserve the general visual features learned from large-scale image classification. Then, the last 40 layers were unfrozen (excluding Batch Normalization layers) to allow the model to adapt its high-level representations to the specific characteristics of the dataset.

We added new layers on top of the model, including a global average pooling layer, dropout layers (to prevent overfitting), a dense layer with L2 regularization (to control weight sizes) and a final SoftMax layer for classification.

We trained the model using Cosine decay to gradually decrease the learning rate in a smooth pattern, helping the model learn without losing the useful features it already knew. For info about the parameters and techniques used in this section see **Table 4**.

3.8. Oversampling F1 score

In the minority classes that got an F1-score below 30% on the validation set, additional training examples were generated and added to those classes to oversample them. For each image belonging to these underperforming classes, two augmented copies were created using the previously defined data augmentation techniques. These augmented samples were then added to the training dataset, increasing the representation of the underperforming classes. The class weights were recalculated to ensure that all classes were balanced after this step.

3.9. Final Model Results

After the early stop we and partial oversample we recalculated the class weights to reflect the new dataset and ran the model once more with early stopping focused on the stop of validation loss reduction with a higher patience of 5. On validation we had a final validation macro F1 Score of 88.15% **Table 1**.

Final Model	Train Accuracy	Train Loss	Train Macro F1	Val Accuracy	Val Loss	Val Macro F1
InceptionV3	77.86%	1.03	86.67%	88.57%	0.86	88.15%

Table 1 - Final Results for Training and Validation

Lastly, we evaluated test data to see how our model performed on unseen data, the results can be seen at **Table 2**.

Final Model	Test Accuracy	Test Loss	Test Macro F1
InceptionV3	89.25%	0.75	88.13%

Table 2 - Test Results

4. CONCLUSION

In this project we addressed the task of classifying rare species images into their respective families using deep learning models. Our initial approach involved building a CNN model from scratch using Keras. This model served as our baseline but quickly showed us the main problem with our models: overfitting.

While trying to improve the results, we opted to use transfer learning on pre-trained models, choosing InceptionV3 over MobileNetV2 and DenseNet121 after a quick baseline assessment between the 3. After experimentation, fine tuning and hyperparameter tuning we arrived at an F1-Score macro of 88.13% on test data, by applying a strategy of slight oversampling and a class weight re-balance after an early stop.

The final model presented itself as balanced and less overfitted leading to an overall better generalization of unseen data. This result showed the improvement of our final solution for this classification task.

BIBLIOGRAPHICAL REFERENCES

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2015). *Rethinking the Inception Architecture for Computer Vision*. arXiv. https://arxiv.org/abs/1512.00567

Optuna. (n.d.). Key features. https://optuna.org/#key_features

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L.-C. (2019). *MobileNetV2: Inverted residuals and linear bottlenecks*. arXiv. https://arxiv.org/abs/1801.04381

Huang, G., Liu, Z., van der Maaten, L., & Weinberger, K. Q. (2018). Densely connected convolutional networks (arXiv:1608.06993). arXiv. https://arxiv.org/abs/1608.06993

APPENDIX A (TABLES)

Models	Train Accuracy	Train Loss	Train Macro F1	Val Accuracy	Val Loss	Val Macro F1
Simple CNN	90.67%	0.25	93.94%	2.8%	40.01	0.01%
MobileNetV2	2.74%	5.23	0.02%	2.43%	5.22	0.08%
DenseNet121	2.56%	5.07	0.12%	2.55%	5.06	0.11%
InceptionV3	6.81%	4.67	1.24%	7.42%	4.80	1.01%

Table 3 - Tested Models

Parameters	Description
Global Average Pooling Layer	Used to reduce the spatial dimensions of the feature maps output by the InceptionV3 backbone. This layer computes the average of each feature map, resulting in a single value per channel. It significantly reduces the number of parameters and helps prevent overfitting, while retaining essential global information from the image.
Dropout (rate = 0.5)	The Dropout layer randomly sets input units to 0 with a frequency defined by the rate parameter (in our case, 50%) at each training step. This helps prevent overfitting by discouraging the network from relying too heavily on specific neurons. The remaining active units are scaled by a factor of 1 / (1 - rate) (in this case, 2) so that the expected sum of the input remains unchanged during training.
Dense Layer (dense_units = 256)	A fully connected network layer that transforms input features into class-specific representations, enabling the model to learn complex patterns. The 256 dense units refer to the number of neurons in a fully connected layer.
L2 Regularization (kernel_regularization = l2(1e-3))	It's a parameter used in the Dense Layer that enables the model to add a penalty to the loss function proportional to the squared magnitude of the weights. The L2 regularization factor was set to 0.001 to provide a moderate penalty on large weights in the Dense layer.
Cosine Decay	Cosine decay is a learning rate scheduling technique where the learning rate decreases smoothly following a cosine curve as training progresses. This smooth decay allows for faster initial learning while fine-tuning the model more carefully in later epochs, leading to improved convergence and model performance.

Table 4 - New layers on top of the InceptionV3

Model	Description
Simple CNN	A Simple Convolutional Neural Network (CNN) designed for image classification and recognition tasks. It extracts hierarchical visual features by simulating the way the human brain processes images [**Error! Reference source not found.**].
MobileNetV2	This model is a lightweight CNN architecture, appropriate for efficiency [Figure 5]. It is built upon MobileNetV1, and it was introduced in 2018 by Google. Its key features rely on the fact that, contrary to traditional residual blocks, this structure expands narrow layers to higher dimensions, and due to using a linear activation instead of *RelU* (Sandler et al., 2019).
DenseNet121	This model has an architecture designed for image classification [**Error! Reference source not found.**], appropriate for the scope of this project, uses a method that each layer output is fed into the following layers, this connectivity improves the reutilization of features. And because of this, it also needs less parameters since it receives information from the previous layers. It also has bottleneck layers, that reduce computational costs and improve efficiency (Huang et al., 2018).

Table 5 - Models Description

Modelling Hyperparameter Tuning	Brief Explanation	Advantages and Disadvantages
Optuna	Open-source optimization framework that helps to find the best hyperparameters to improve models' performance. Automates the process of searching for the best hyperparameters (such as learning rate, number of layers, etc.) by utilizing advanced optimization algorithms. It works by running trials, evaluating different sets of hyperparameters and pruning poorly performing ones early to save resources, this is done many times until the set of best hyperparameters is found.	This allows to find the best hyperparameters without much trial and error and it supports parallel execution. It also reduces computational costs by stopping poorly performing trials early. However, it can lead to overfitting and can be computationally expensive for complex models, even though it uses pruning. Besides that, the quality of the optimization is highly dependent on the objective function.

Table 6 - Optuna Description

Model	Train Accuracy	Train Loss	Train Macro F1	Val Accuracy	Val Loss	Val Macro F1
InceptionV3	70.76%	1.63	62.58%	60.18%	2.26	52.45%

Table 7 - InceptionV3 before Oversampling

APPENDIX B (IMAGES)

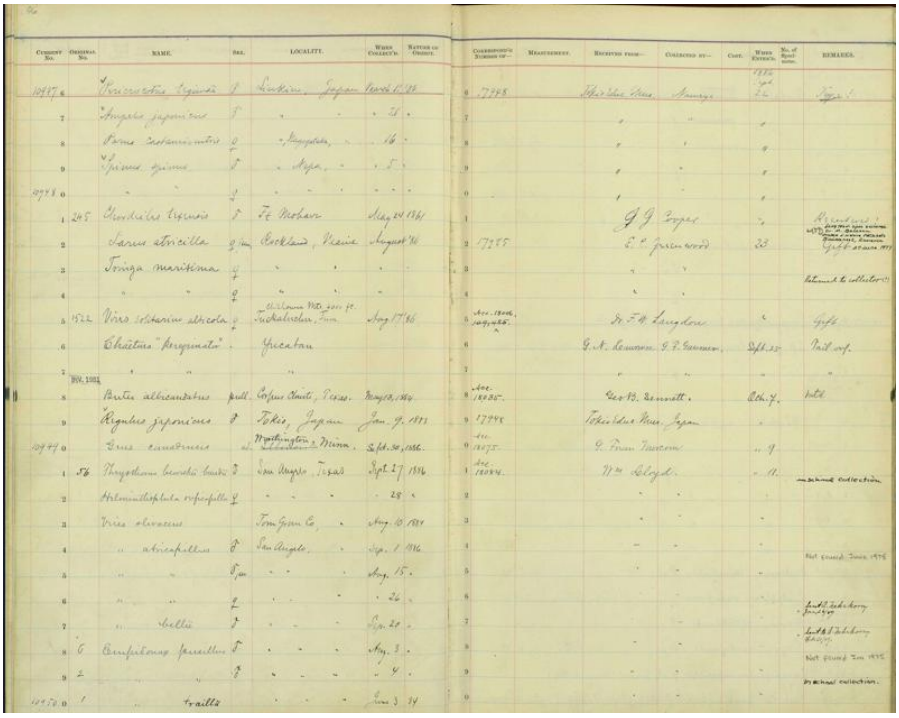

Figure 1 – Removed Image Example

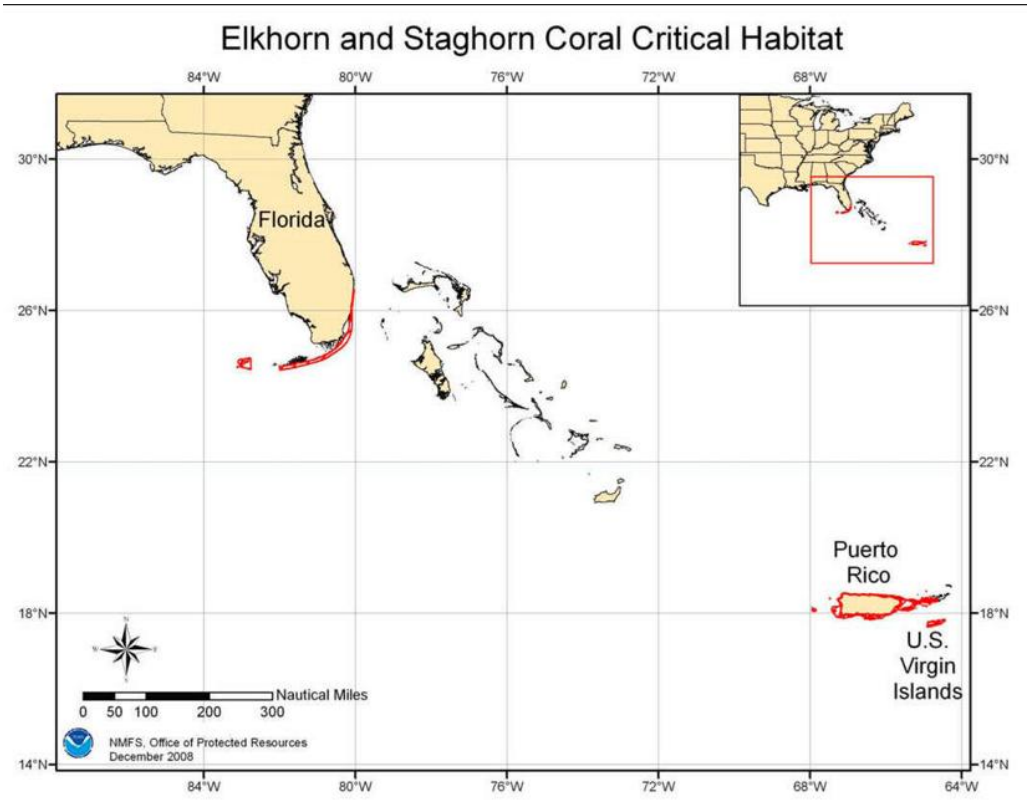

Figure 2 – Removed image example 2

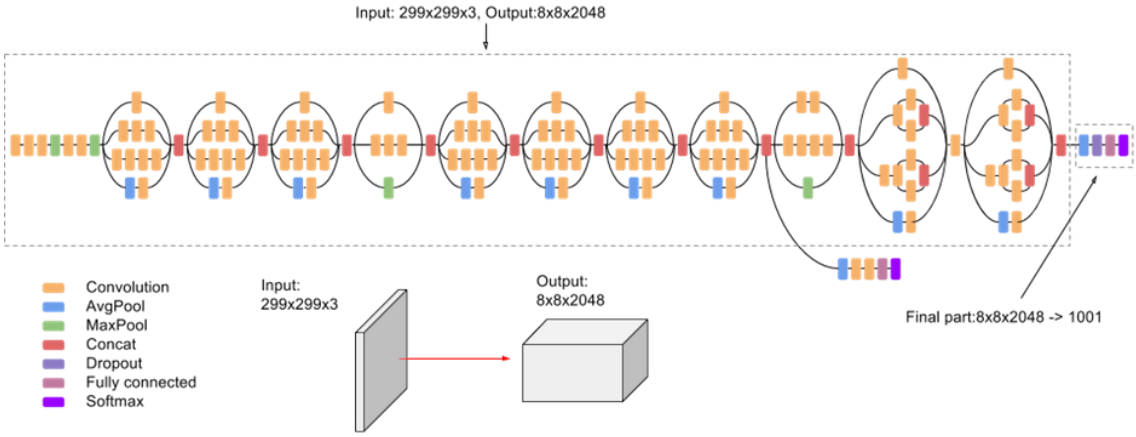

Figure 3 – InceptionV3 Scheme

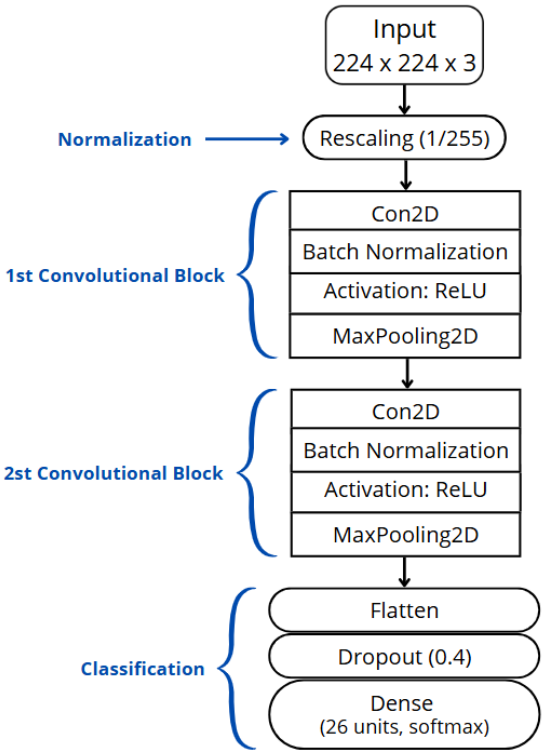

Figure 4 - Custom CNN Scheme

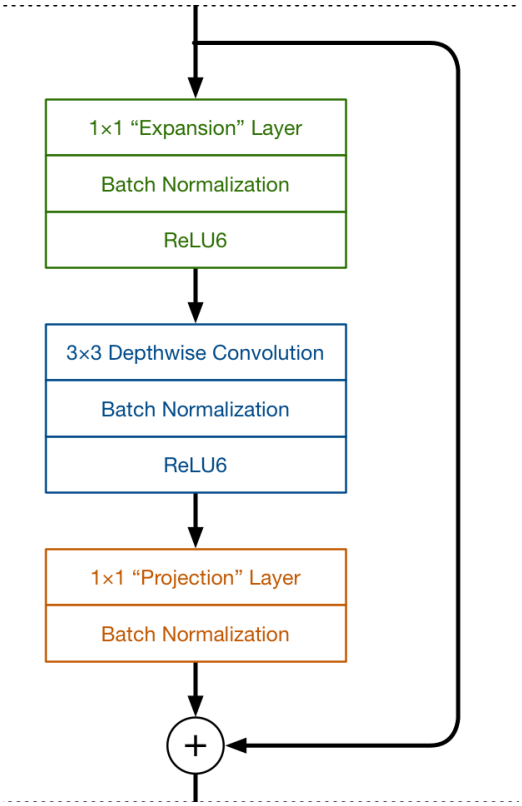

Figure 5 - MobileNetV2 Scheme

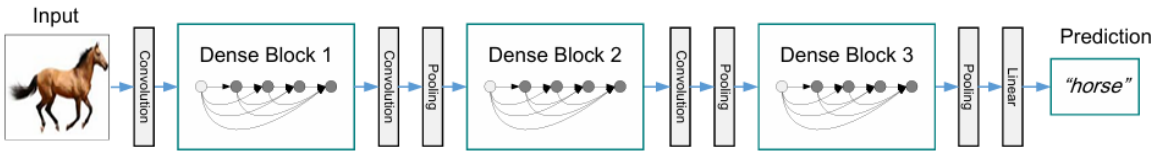

Figure 6 - DenseNet121 Scheme

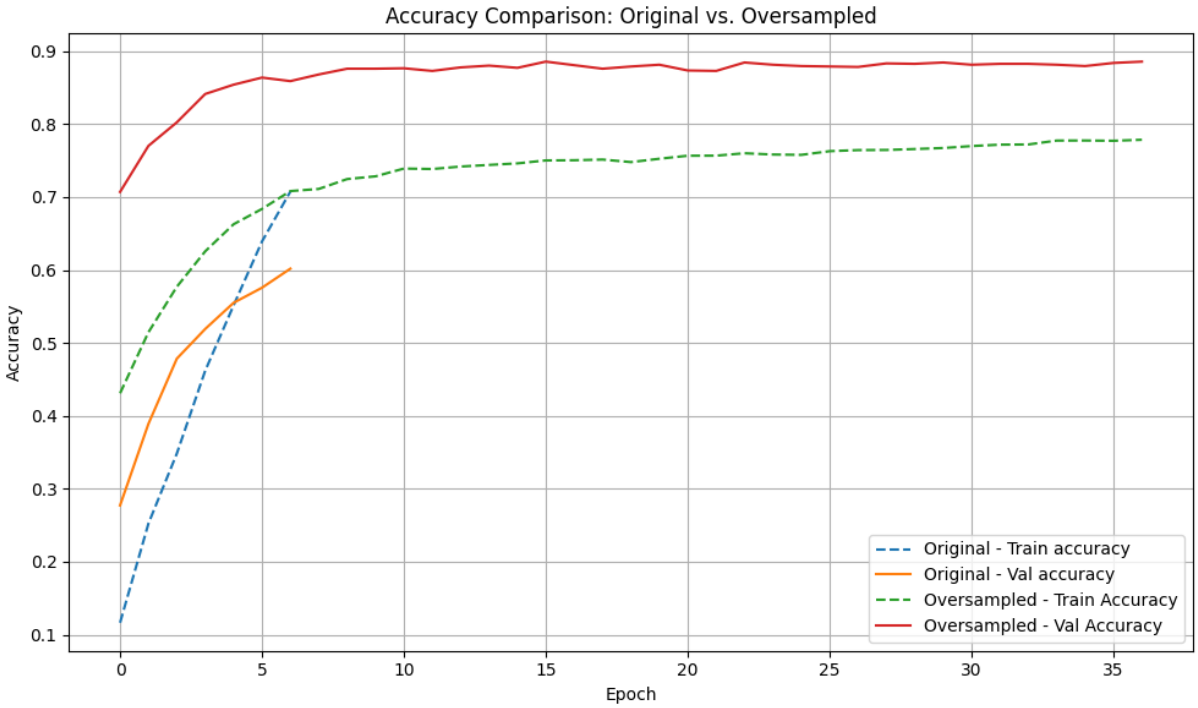

Figure 7 - Accuracy Comparison

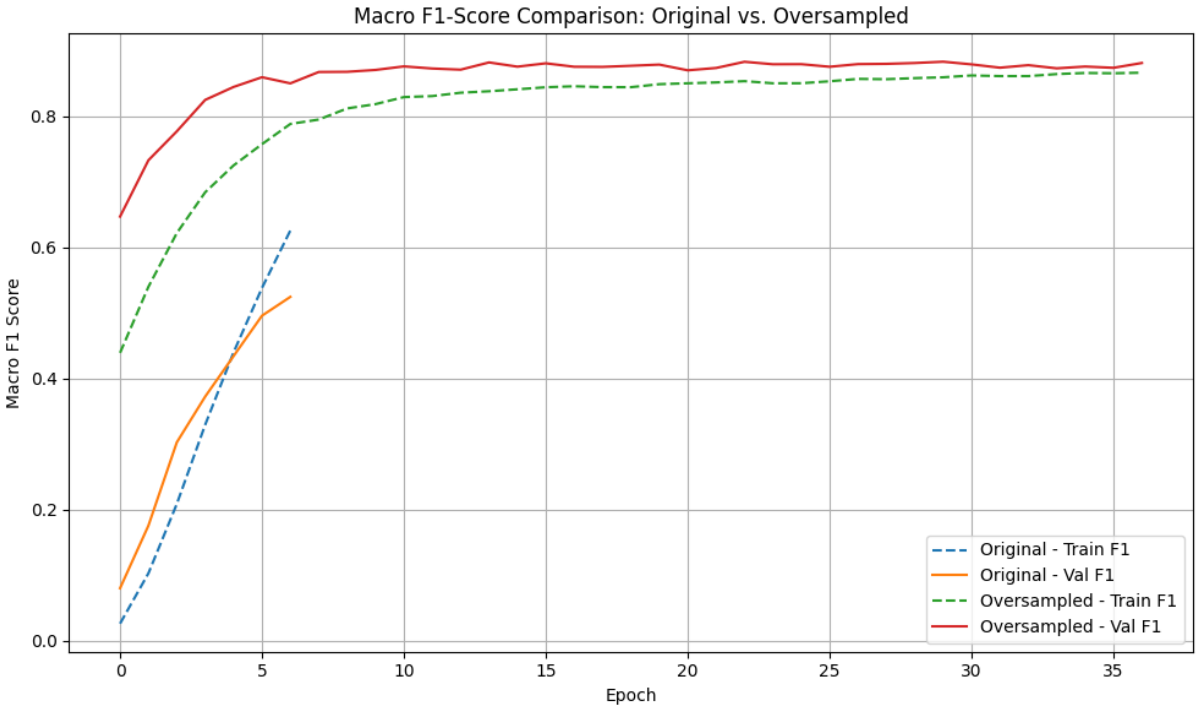

Figure 8 - Macro F1-Score Comparison