DEEP LEARNING

Master in Data Science and Advanced Analytics

NOVA Information Management School

Universidade Nova de Lisboa

Classification of Rare Species Using CNNs and BioCLIP Data

Group 5

Inês Major, 20240486

Luís Semedo, 20240852

Pedro Santos, 20240295

Rafael Bernardo, 20240510

Rodrigo Miranda, 20240490

Fall/Spring Semester 2024-2025

ABSTRACT

This project tackles the challenge of classifying rare biological species at the family level using deep learning models and the *BioCLIP* dataset. The dataset contains highly imbalanced and visually diverse images sourced from the Encyclopedia of Life, making this a complex multi-class classification task. We evaluated both custom convolutional neural networks and state-of-the-art pre-trained models, including *InceptionResNetV2*, *DenseNet121*, and *ConvNeXtBase*. Our pipeline involved architecture-specific preprocessing, careful train-validation-test splitting, and regularization strategies such as data augmentation and dropout. Among all approaches, the *ConvNeXtFull* model achieved the best performance, reaching a macro F1 score of 82% on a 90/10 split. While fine-tuning showed limited benefits and overfitting remained a concern, our results highlight the importance of appropriate preprocessing, balanced augmentation, and model selection in achieving strong generalization performance. This work demonstrates the potential of transfer learning in biodiversity research and opens opportunities for further optimization.

Keywords: Convolutional Neural Networks, Image Classification, Transfer Learning, Deep Learning, Data Augmentation, Keras, Regularization

TABLE OF CONTENTS

1. INTRODUCTION

The remarkable advancements in computer vision over the years made possible new scientific discoveries across a wide range of fields. Among them, biology has greatly benefited from the application of deep learning models, especially in areas where the human eye can be faulty, such as species identification. In this project, we explore this challenge through deep learning models, leveraging the BioCLIP dataset: a collection of curated images of rare species sourced from the Encyclopedia of Life (EOL) aiming to develop a model capable of classifying specimens at the *family* level. This work can support biologists by improving the speed and accuracy of species classification, aiding in biodiversity monitoring and conservation efforts. By automating a traditionally manual process, our model contributes to more scalable and data-driven approaches in ecological research.

The goal of this work is to design and implement a deep learning pipeline that can learn visual representations from a highly imbalanced, complex and diverse dataset. Each image is associated with hierarchical metadata (kingdom, phylum and family), but our focus is predicting the family of the given specimen solely from its image. The difficulty of this task is amplified not only by the rarity and diversity of the species involved but also by the visual similarities across different taxonomic families. These conditions make it a good benchmark to test the capabilities of both traditional convolutional neural networks and newer pre-trained models adapted to image classification.

2. PREPROCESSING

After analyzing the class distribution, we identified a clear imbalance between categories. To account for this, we selected evaluation metrics that better reflect performance across all classes, not just the most frequent ones. Specifically, we used accuracy, macro F1-score, weighted F1-score, and top-k accuracy, allowing for a more balanced and meaningful assessment of the model's performance.

To ensure a balanced representation of classes across the dataset, we performed a stratified split using *train_test_split* from scikit-learn. First, the data was split into 70% for *train_files* and 30% for a temporary subset (*temp_files*). Then, *temp_files* was further divided equally into the validation and test sets, assigning 15% of the original data to *val_files* and 15% to *test_files*. This approach allowed us to maintain consistent class distributions across all subsets.

2.1. Data Loading

To load the image data efficiently, we used *image_dataset_from_directory* from *Keras* for all three dataset splits. For the training set, we enabled shuffling to ensure that each batch contains a diverse mix of classes, which helps prevent overfitting. Images were loaded in RGB format, with bilinear interpolation applied during resizing. The *label_mode* was set to *"categorical"* to produce one-hot encoded labels, suitable for multi-class classification. A batch size of 32 was chosen to balance computational efficiency and model convergence. For the validation and test sets, we kept shuffling disabled to preserve the order of samples, which is useful during evaluation and error analysis. Consistent preprocessing across all datasets ensures that the model is trained and evaluated under the same conditions.

2.2. Resizing

Initially, we normalized all input images by scaling their pixel values to the [0,1] range, following a standard practice in deep learning. However, when experimenting with pre-trained models from *keras.applications (e.g., DenseNet*, *InceptionResNetV2*, and *EfficientNet*), we identified an important detail: each architecture requires a specific preprocessing pipeline that matches its original ImageNet training.

Keras provides a *preprocess_input* function tailored to each model, which performs architecture-specific transformations. Examples of this include scaling to [-1,1] (Inception based models) or applying mean subtraction (DenseNet). Using the wrong preprocessing strategy can lead to suboptimal performance, as the input distribution would no longer match what the model expects.

To address this, we revised our preprocessing pipeline to dynamically apply the correct *preprocess_input* function based on the selected architecture. This ensured better alignment between the input data and the pre-trained model's internal representations, ultimately leading to more stable training and improved classification results.

3. INITIAL TESTS

3.1. Sequential Model

As part of our initial approach, we implemented a simple CNN, using the Sequential API, composed of three stacked *Conv2D* layers. This basic architecture helped us quickly validate that, as expected, the model suffered from severe overfitting primarily due to its depth, lack of regularization techniques, and the absence of a data augmentation pipeline. No further optimization efforts were made on this baseline.

3.2. CNN

As a second approach, we developed a custom convolutional neural network (CNN) from scratch, structured into two main convolutional blocks with batch normalization and max pooling layers, followed by a dense classification head. This model was implemented as a subclass of the *keras.Model* class, giving us greater flexibility and control over the architecture. We used callbacks such as early stopping and model checkpointing to manage training stability. Despite this setup being technically sound, we quickly realized that it was computationally infeasible, as it took over 10 hours to complete 4 epochs. The time and resources required made this approach very inefficient, leading us to pivot towards a transfer learning strategy, leveraging pre-trained models to drastically reduce training time.

4. TRANSFER LEARNING

In the initial phase, we tested several pretrained models, including EfficientNetB0/B3, Resnet50, DenseNet121, MobileNetV2, inceptionV3, NASNetMobile, ConvNeXtTiny, VGG16, InceptionResNetV2 and ConvNextBase. However, we decided to proceed with InceptionResNetV2, DenseNet121 and ConvNextBase due to the promising preliminary results.

4.1. InceptionResNetV2

4.1.1. Preliminary Experiments: Sampled Dataset and Baseline Model

Using the InceptionResNetV2 architecture as a feature extractor (*include_top=False*, frozen weights), we created a baseline model. To perform multi-class classification across 202 species families, we wrapped the model in a custom *Keras* class with a flattening layer and a *softmax* activation as output. We trained on a sampled dataset (1500 training and 500 validation images) to ensure quick experimentation and decrease computational costs. To evaluate the model's baseline capacity without added complexity, regularization was limited to a *Dropout* layer (rate = 0.2), and no data augmentation was applied. As anticipated given the lack of augmentation and the relatively high learning rate (0.001) for a pre-trained model, training produced roughly fifty percent validation accuracy after 10 epochs (Figure *1*) but also showed signs of overfitting.

We developed a versatile data augmentation pipeline, integrated directly into the model, to support more reliable experiments. To preserve image integrity (Figure *5*), the original augmentation configuration was used to ensure light transformations *(*Figure *8)*. We then retrained the model on the same dataset using the augmentation pipeline, increased regularization, and a reduced learning rate (1e-5). As evidenced by better validation performance, these modifications minimized overfitting and promoted more stable learning (Figure *2*).

4.1.2. Scaling Experiments to the Full Dataset

Insights from initial experiments emphasized the need to scale testing to the full dataset. Using the flexible augmentation function, several pipelines were evaluated. Stronger augmentation strategies (e.g., larger rotations, contrast, sharpness, and saturation changes; Figure *7*) improved regularization but led to degraded validation performance. In turn, a more moderate augmentation configuration (horizontal flipping, minor rotations and zooms, slight contrast and brightness adjustments; Figure *6*; Table *2*) yielded an approximate

10% improvement in validation accuracy as well as F1 Macro scores (Figure *3*). Training dynamics with the custom augmentation pipeline (Table *1*) showed steady improvement across metrics without signs of severe overfitting. Validation curves closely tracked training curves, and the stability of the learning rate (1e-5) contributed to controlled weight adaptation. These results undermine the importance of balancing augmentation strength to optimize generalization while maintaining learning effectiveness, refer to *Table 1 f*or a performance overview across our several experiments.

4.1.2. Fine Tuning

To further optimize performance, we unfroze the top 50 layers of the *InceptionResNetV2* backbone to further maximize performance. Validation gains in accuracy, F1-score, and loss were modest, indicating limited generalization improvement, while training accuracy increased significantly over 10 epochs (Figure *4)*. Additional tuning was not carried out as other models performed better. For improved control of overfitting, future investigations could focus on unfreezing fewer layers, tweaking dropout rates, or changing early stopping rounds.

4.2 DenseNet121

The *DenseNet121* model was used with pre-trained ImageNet weights, adapted for a multi-class classification task involving 202 categories. The architecture was extended with a *GlobalAveragePooling2D* layer, *Dropout* (0.5), and a final *Dense* layer with *softmax* activation. The model was compiled using the *AdamW* optimizer (*lr=5e-4, weight_decay=3e-4*), and performance was evaluated using *accuracy*, *top-5 accuracy*, *F1 macro*, and *F1 weighted*, as shown in Figure *9***.**

4.2.1. Baseline Model

In the baseline phase, only the top layers were trained, with the *base_model* frozen. After 51 epochs, a consistent improvement was observed in the training and validation curves (*Accuracy*, *F1*), along with convergence in *Loss*. Validation *accuracy* stabilized at 67.95%, with an *F1 macro* of 65.42%. On the test set, the results were strong: 70.52% accuracy, 68.45% F1 macro, and 89.21% top-5 accuracy (see

4.2.2. Fine Tuning

During fine-tuning, the last 50 layers of the backbone were unfrozen and trained with a reduced *learning rate* (1e-5). Although training metrics improved, the validation curves showed an early plateau and slight divergence (*Figure 10*), with validation *accuracy* and *F1 macro* both settling at 65.95%. On the test set, the performance dropped slightly to 68.91% accuracy and 66.52% F1 macro, with *top-5 accuracy* at 88.48%.

4.2.3. Model Comparison and Future work

Overall, the baseline model outperformed the fine-tuned version in terms of generalization, showing higher test accuracy and macro F1 score. The fine-tuning stage, although promising during training, did not translate into better validation or test results. As future work, improvements may include using *class_weights* to address class imbalance, experimenting with *Dropout* rates, fine-tuning fewer or more layers selectively, and applying more diverse data augmentation to enhance robustness.

4.3. ConvNeXtBase

Given the promising performance of the *ConvNeXtBase* architecture and the limitations imposed by our access to GPU resources on Google Colab, we focused most of our Colab-based experimentation on this architecture. This decision allowed us to thoroughly explore its potential using different configurations while maintaining a consistent and fair evaluation framework across experiments. As a result, a dedicated notebook named “ConvNext Notebook” was created specifically for these experiments and it is stored in the project folder “ConvNext”.

Unlike other models such as *DenseNet* or *Inception*, which require specific preprocessing transformations (e.g., mean subtraction or scaling to [-1, 1]), *ConvNeXtBase* was pre-trained on raw ImageNet images with pixel values in the [0, 255] range. As such, the *preprocess_input* function provided by *Keras* for *ConvNeXt* models is essentially a no-op: it leaves the image data unchanged. All images were fed into the model in their raw RGB format as loaded by *image_dataset_from_directory*, resized to (224, 224), and passed directly to the network.

4.3.1. Training Results Without Augmentation

Although a comprehensive data augmentation pipeline was initially implemented (including flipping, rotation, contrast, zoom, brightness and translation), empirical results showed no significant improvement in validation performance. On the contrary, augmentation sometimes led to instability and therefore we chose to remove it from the final models.

4.3.2. Model Architecture

We developed and tested four different model variants, all built upon the pre-trained *ConvNeXtBase* backbone. These variants differed only in their classification heads, as detailed in *Table 4*: *MiniConvNext*, *ConvNextDropout*, *ConvNextMid*, *ConvNextFull* in where the complexity of the model increases respectively. We compiled the model using the *AdamW* optimizer with a learning rate of 1e-4 and weight decay 3e-4, and trained the classification head for only 13 epochs since after some experiments all the models started overfitting after that. Early stopping and model checkpoint callbacks were used to prevent overfitting and preserve the best weights.

4.3.3. Training Results, Batch Size Comparison and Final Model Selection

All four *ConvNeXtBase* variants (*MiniConvNext*, *ConvNextDropout*, *ConvNextMid*, and *ConvNextFull*) were evaluated using batch sizes of 8, 16, 32, and 64. The objective was to assess whether batch size had a measurable impact on performance, training stability, and runtime. Interestingly, in our Google Colab setup, the runtime per epoch remained virtually identical across batch sizes, allowing us to freely explore larger batches without time constraints. As such, training duration did not influence the choice of model or configuration.

We summarize the results of all tested combinations in Table *5*, the best overall performance was obtained with *ConvNextFull* + batch size 32, which provided an optimal trade-off between learning capacity, generalization, and training stability since we aimed to not surpass a 5 points difference between train and test (history plots in *Figure 11*).

To better understand the limitations of the final model, we also analyzed its class-wise performance. *Figure 12* presents the 10 classes with the lowest F1-scores on the validation set. Most of these belong to underrepresented families, such as *chordata_trionychidae* and *chordata_rhinodermatidae*, where the model struggled to correctly classify samples. This is likely due to data imbalance and visual similarity to other classes.

4.3.4 Fine-Tuning attempts

We conducted several fine-tuning experiments by unfreezing the *ConvNeXtBase* backbone and lowering the learning rate to 1e-5. Despite adjusting multiple regularization strategies, learning schedules, and callbacks, the models consistently overfitted: training accuracy improved while validation metrics either plateaued or declined. Due to this behavior, fine-tuning was ultimately discarded and we retained the frozen backbone configuration, which proved more stable, consistent, and effective for this task.

4.3.5 Random Search Tuning

To further improve performance and validate our design choices, we explored hyperparameter tuning using the *Keras Tuner* library with *Random Search*. Our objective was to assess whether a more optimized combination of dense units, regularization, dropout, and learning parameters could outperform our manually built *ConvNeXtFull* architecture.

We defined a *HyperModel* class built on the frozen *ConvNeXtBase* backbone, allowing the tuner to sample configurations for the number of dense units, dropout rates, L2 regularization coefficients, learning rate, and weight decay. We used val_f1_macro as the optimization metric, with a total of 20 trials (1 execution per trial). The best model achieved a validation F1-macro score of 0.79498, which is comparable to our manually designed models, as we can see in Figure *13* which contains all the experimentations done in *random search.*

Despite this result, we observed that most of the best-performing configurations exhibited significant overfitting. This behavior was not fully captured in the final validation metrics stored in the tuning logs but was consistently observed during training monitoring. Therefore, we opted to continue with the *ConvNeXtFull* model, which we manually designed and had previously identified as our most stable and effective configuration.

5. Final Evaluation on a 90/10 Split

To further improve performance, we decided to increase the volume of training data, operating under the assumption that a larger dataset would allow the model to learn more representative features. We expanded the dataset and performed a 90/10 train-validation split, using our best-performing model from section 4.3, and trained it on a total of 10,784 images. The results (*Figure 14*) were encouraging: the model achieved an F1 Macro score of approximately 82% on the validation set and the gap between training and validation metrics remained within a 5% margin, a threshold we considered a reasonable indicator of good generalization.

Compared to our previous F1 Macro score of 78% on the test set, this approach clearly demonstrates that increasing the training data had a more substantial impact on performance than any other technique we applied. That said, a closer look at the model's performance per class reveals persistent challenges. Notably, only one class (*cnidaria_helioporidae*) was not predicted correctly at all (Figure *15*). These underperforming classes correspond to those with very limited representation in the dataset. While the overall model benefits from more data, classes with few examples may suffer due to this imbalance, highlighting the importance of addressing class distribution in future iterations.

6. Conclusion

Throughout the project, we successfully designed and evaluated a deep learning pipeline for classifying 202 distinct species at the family level using the BioCLIP dataset. The results highlighted the critical role of architecture-specific preprocessing when applying transfer learning, as well as the varying effects of data augmentation, batch size, architectural choice, and regularization depending on the model. *ConvNeXtFull* was the top-performing model, as was covered in Section 5, with a remarkable F1 Macro score of 82% on the validation set. The 90/10 training-validation split was an important contributor, since it gave the model access to a broader and more complete dataset. Improved generalization was also ensured by this tactic, as shown by the validation performance staying within 5% of the training outcomes.

Despite these developments, issues like class disparity continued to be a significant barrier, especially affecting the prediction accuracy for families with fewer images. Furthermore, it was challenging to manage overfitting during the fine-tuning stages, especially in deeper architectures like *ConvNeXtBase* and *InceptionResNetV2*. The need for model-specific data strategies was further reinforced by the fact that, although data augmentation was beneficial in some situations (most notably with *InceptionResNetV2*), it tended to negatively impact performance in others. More refined fine-tuning methods, like unfreezing fewer layers or thoroughly modifying learning rates and dropout values, should be investigated in future research. Furthermore, a *GridSearch* for hyperparameter tuning across architectures may help optimize performance, however, the computational cost of these experiments needs to be carefully considered.

References

Dogo, E. M., Afolabi, O. J., & Twala, B. (2022). *On the relative impact of optimizers on convolutional neural networks with varying depth and width for image classification*. **Applied Sciences, 12**(23), 11976. https://doi.org/10.3390/app122311976

TensorFlow. (n.d.). *tf.keras.activations.gelu*. TensorFlow. https://www.tensorflow.org/api_docs/python/tf/keras/activations/gelu lu.

Hugging Face. (n.d.). *ConvNeXtV2*. Hugging Face. https://huggingface.co/docs/transformers/model_doc/convnextv2

Chollet, F., & Keras team. (n.d.). *Keras documentation*. Keras.io. https://keras.io/api/

APPENDIX

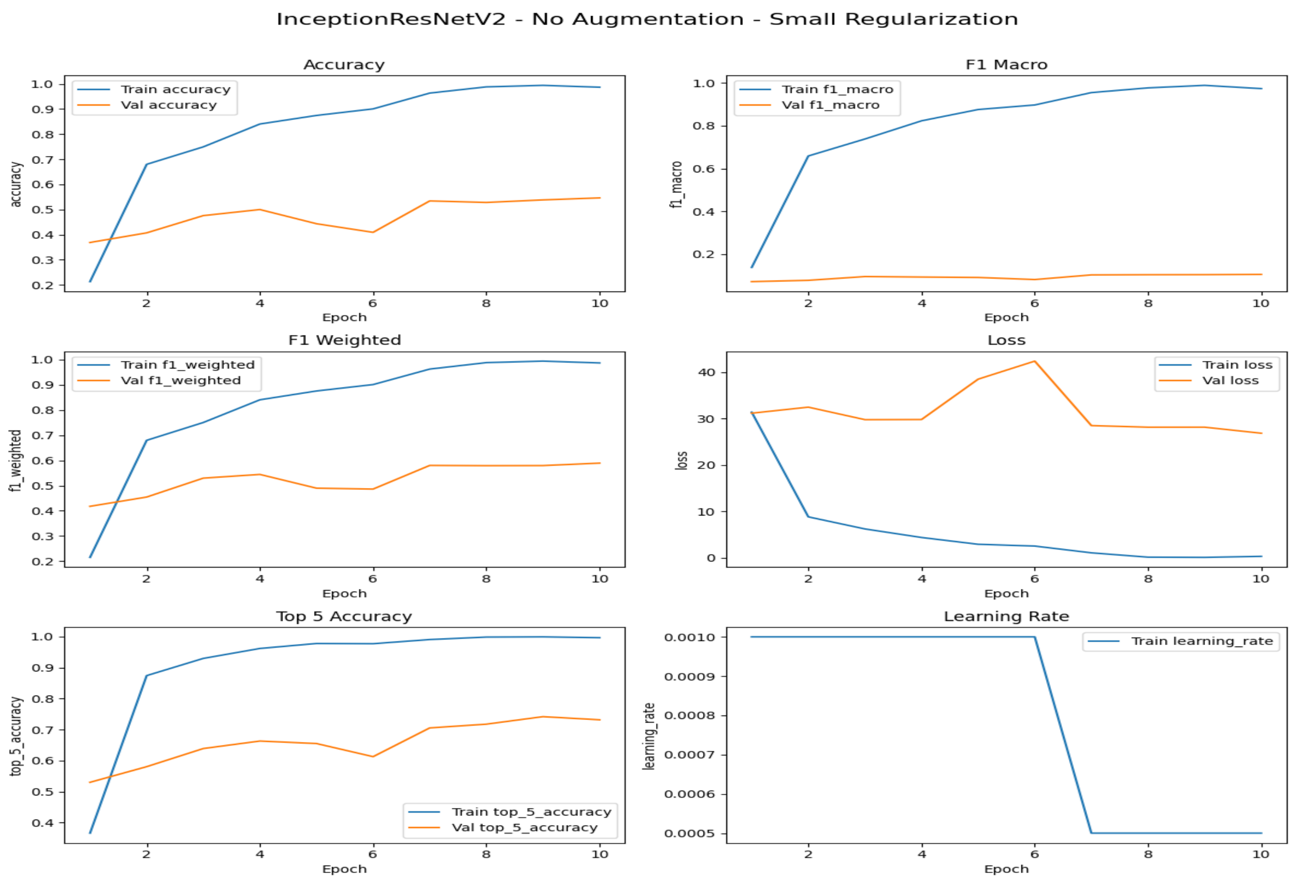


Figure 1: Metrics for First Sampled Run

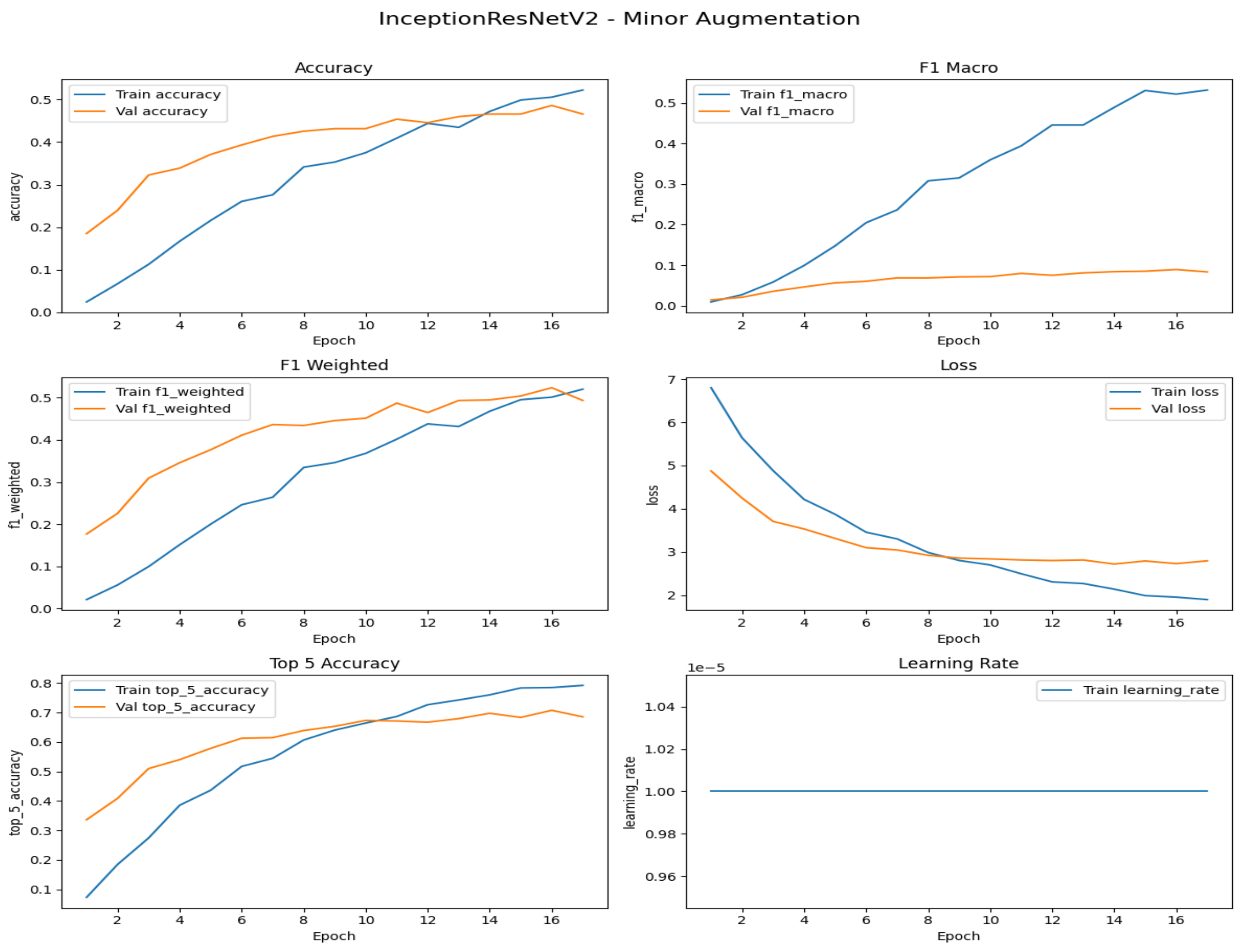


Figure 2: Metrics for Sampled Run with Minor Augmentation

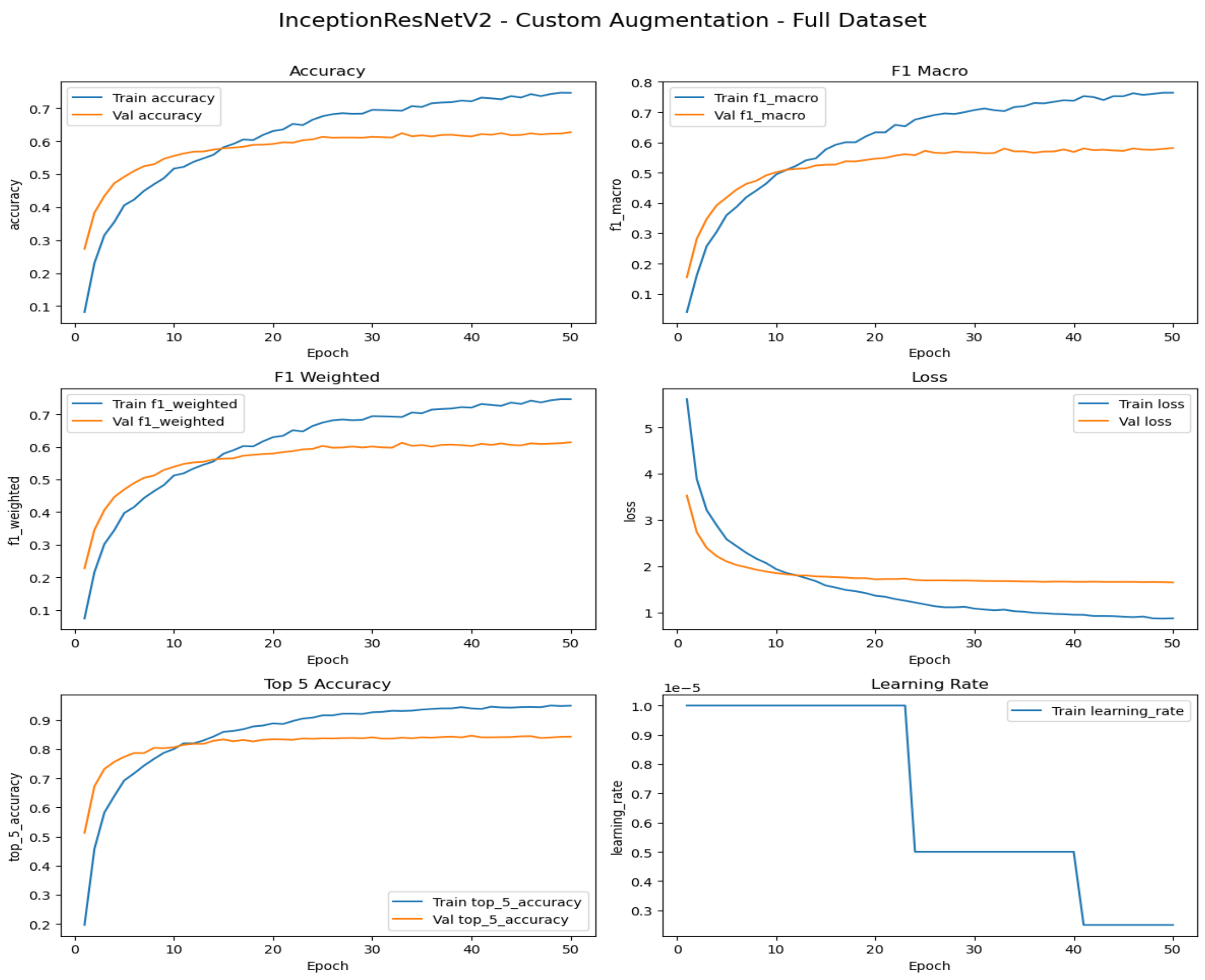


Figure 3: Metrics for Full Dataset Run with Custom Augmentation Pipeline

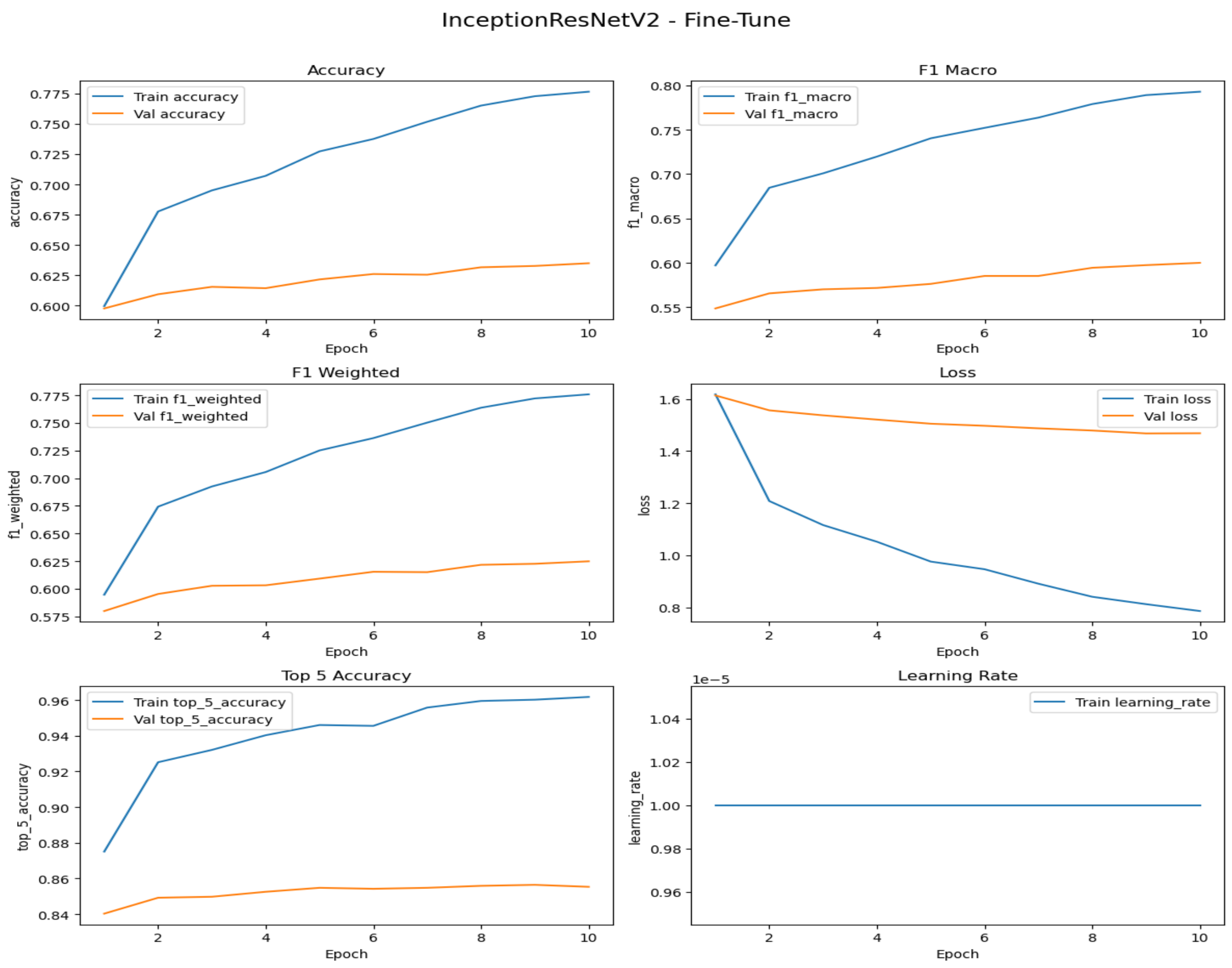


Figure 4: Metrics for Fine-Tuning Experiment

	Accuracy			F1 Macro			F1 Weighted			Top 5 accuracy		
	Train	Val	Test	Train	Val	Test	Train	Val	Test	Train	Val	Test
Sampled Run - No Augmentation	0.9866	0.5464	0.4127	0.9735	0.1055	0.3025	0.9866	0.5891	0.3904	0.9967	0.7319	0.6429
Sampled Run - Minor Augmentation	0.5221	0.4657	0.3348	0.5320	0.0834	0.2194	0.5199	0.4933	0.2890	0.7921	0.6855	0.6123
Full Dataset - Custom Augmentation	0.7465	0.6272	0.6274	0.7641	0.5817	0.5859	0.7459	0.6138	0.6168	0.9489	0.8425	0.8621
Fine-Tune	0.7765	0.6349	0.6440	0.7927	0.6000	0.5972	0.7759	0.6248	0.6351	0.9617	0.8553	0.8732

Table 1: Performance Overview

Runs	RandomFlip	RandomZoom	RandomContrast	RandomSharpness	RandomBrightness
Original Image	-	-	-	-	-
Augmentation 1 - (Original Implementation)	Horizontal	0.35	0.15	0.2	0.3
Augmentation 2 - (Old Pipeline)	Horizontal	0.4	0.2	0.25	0.15
Augmentation 3 - (Custom Augmentation)	Horizontal	0.2	0.1	0.075	0.075

Table 2: Data Augmentation Pipelines

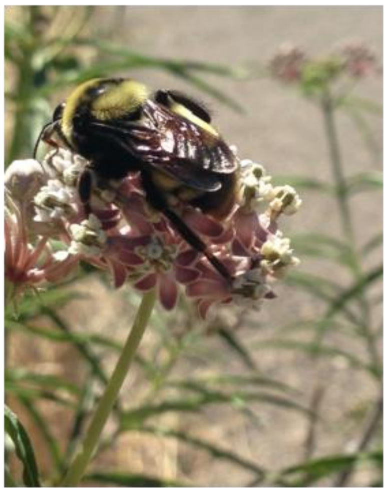


Figure 5: Original Image

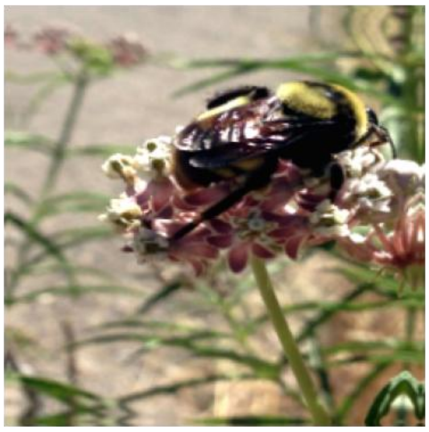


Figure 8: Augmentation 1

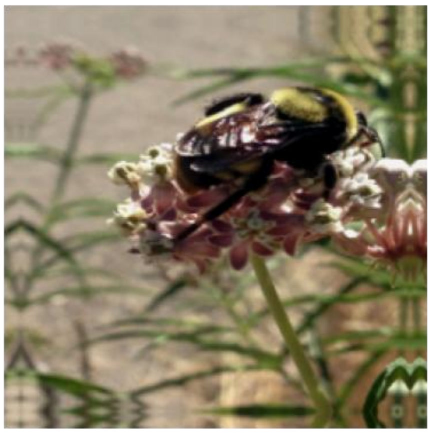


Figure 7: Augmentation 2

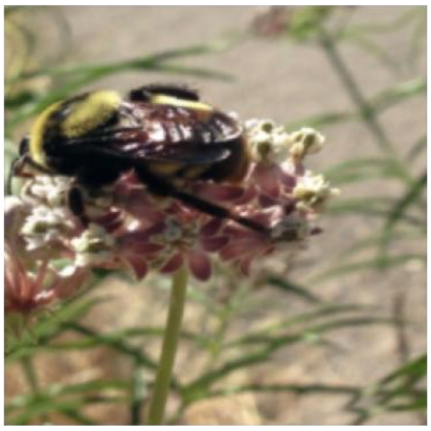


Figure 6: Augmentation 3

	Accuracy			F1 Macro			F1 Weighted			Top 5 accuracy			
	Train	Val	Test	Train	Val	Test	Train	Val	Test	Train	Val	Test	Epochs
Baseline	0,6982	0,6795	0,7052	0,6875	0,6542	0,6845	0,6959	0,6685	0,6958	0,9132	0,8792	0,8921	51/60
Fine-Tune	0,7144	0,6595	-	0,6987	0,6595	-	0,7122	0,6771	-	0,9286	0,8848	-	30/30

Table 3: Performance Overview (DenseNet121)

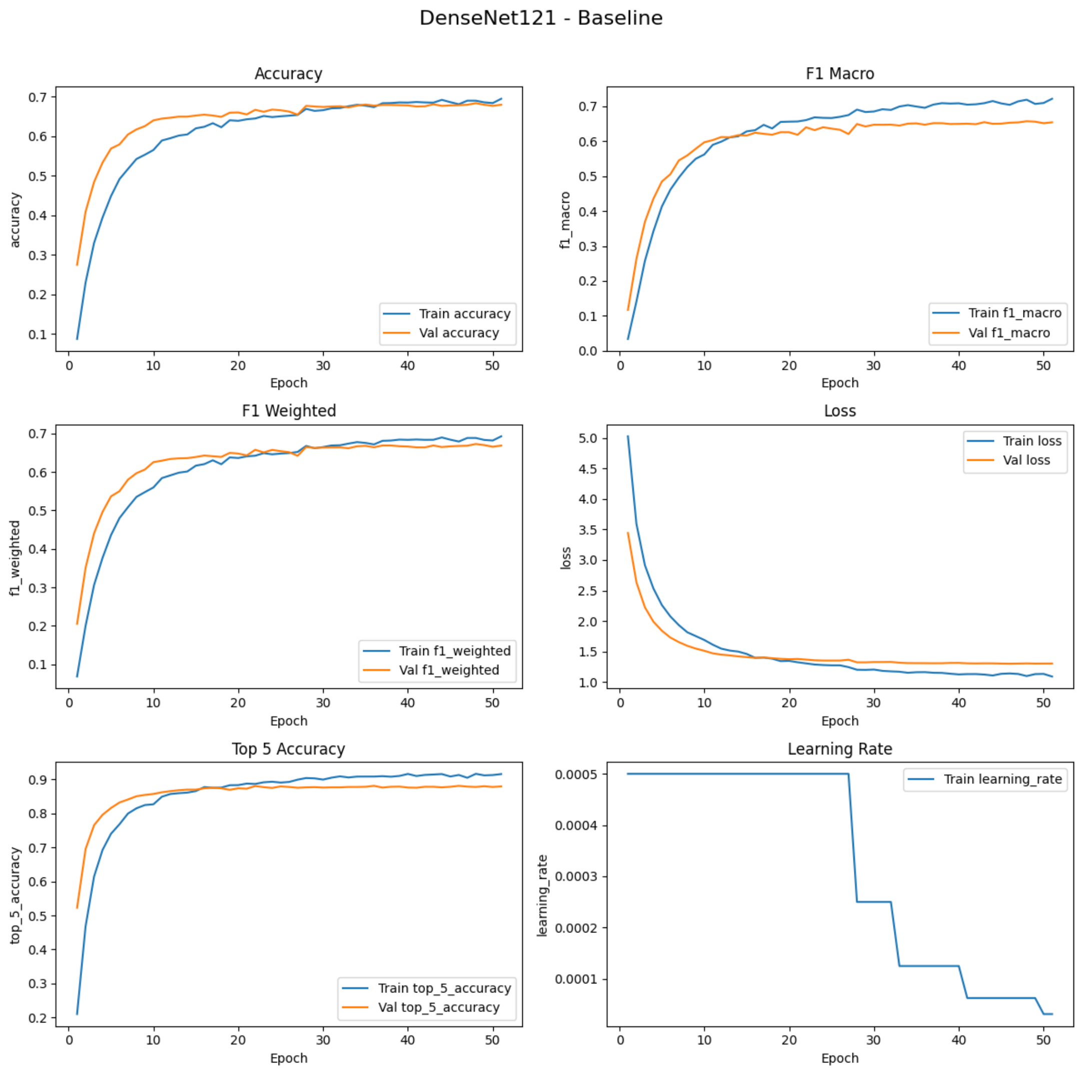


Figure 9: Metrics for the Baseline Model

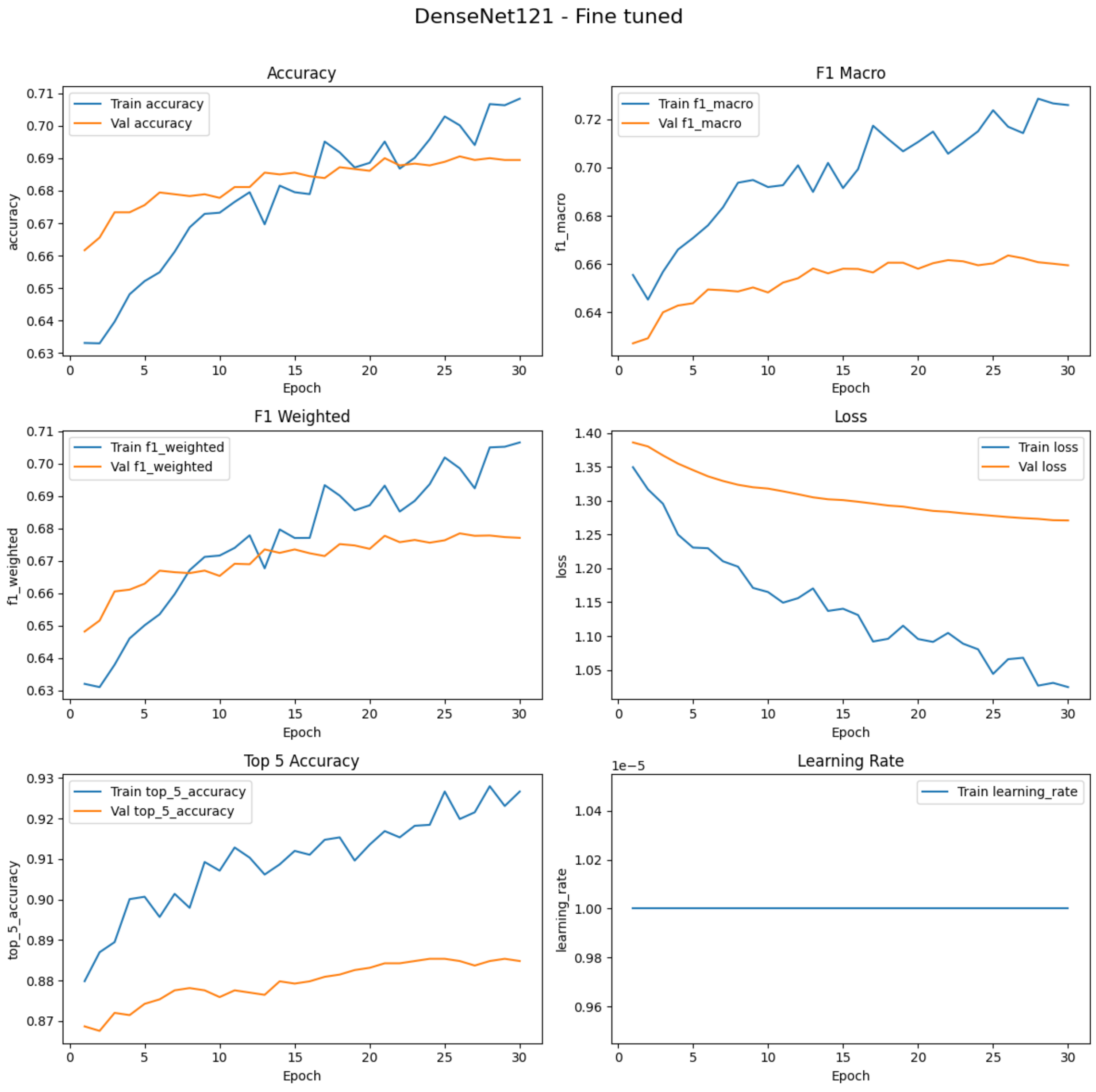


Figure 10: Metrics for the Fine Tuning Model

Models	Architecture (backbone + head)
MiniConvNext	ConvNeXtBase + GlobalAvgPool + Dense(n_classes)
ConvNextDropout	ConvNeXtBase + GlobalAvgPool + Dropout(0.3) + Dense(n_classes)
ConvNextMid	ConvNeXtBase + GlobalAvgPool + Dense(256, ReLU) + BatchNorm + Dropout(0.4) + Dense(n_classes)
ConvNextFull	ConvNeXtBase + GlobalAvgPool + Dense(1024, GELU, L2) + BN + Dropout(0.5) + Dense(512, GELU, L2) + BN + Dropout(0.5) + Dense(n_classes)

Table 4: The four models' architecture that we built

Models	Batch	Accuracy			F1 Macro			F1 Weighted			Top 5 accuracy		
		Train	Val	Test	Train	Val	Test	Train	Val	Test	Train	Val	Test
MiniConvNext	8	0.94	0.81	0.82	0.91	0.80	0.80	0.95	0.81	0.81	0.99	0.95	0.94
	16	0.90	0.80	0.80	0.88	0.81	0.78	0.9	0.80	0.79	0.98	0.94	0.94
	32	0.86	0.78	0.77	0.83	0.76	0.75	0.86	0.77	0.77	0.97	0.93	0.93
	64	0.80	0.74	0.74	0.77	0.72	0.71	0.80	0.73	0.73	0.96	0.92	0.92
ConvNextDropout	8	0.91	0.81	0.81	0.88	0.79	0.80	0.90	0.80	0.80	0.99	0.94	0.94
	16	0.86	0.79	0.79	0.84	0.77	0.77	0.86	0.78	0.78	0.98	0.94	0.93
	32	0.81	0.77	0.76	0.77	0.73	0.73	0.81	0.76	0.75	0.96	0.93	0.93
	64	0.75	0.72	0.72	0.70	0.68	0.68	0.75	0.7	0.70	0.94	0.92	0.92
ConvNextMid	8	0.91	0.81	0.80	0.88	0.79	0.77	0.92	0.81	0.79	0.99	0.93	0.93
	16	0.90	0.79	0.80	0.86	0.77	0.78	0.90	0.79	0.80	0.99	0.93	0.94
	32	0.86	0.78	0.78	0.81	0.75	0.74	0.85	0.78	0.77	0..98	0.93	0.93
	64	0.80	0.76	0.75	0.74	0.71	0.7	0.80	0.74	0.74	0.95	0.92	0.91
ConvNextFull	8	0.87	0.82	0.81	0.83	0.80	0.79	0.87	0.81	0.81	0.98	0.94	0.94
	16	0.88	0.82	0.81	0.84	0.80	0.79	0.88	0.82	0.81	0.98	0.94	0.94
	32	0.84	0.81	0.81	0.80	0.79	0.79	0.85	0.81	0.80	0.97	0.94	0.94
	64	0.88	0.82	0.81	0.84	0.80	0.79	0.88	0.81	0.80	0.98	0.94	0.94

Table 5: Results of all the combinations tested in ConvNext

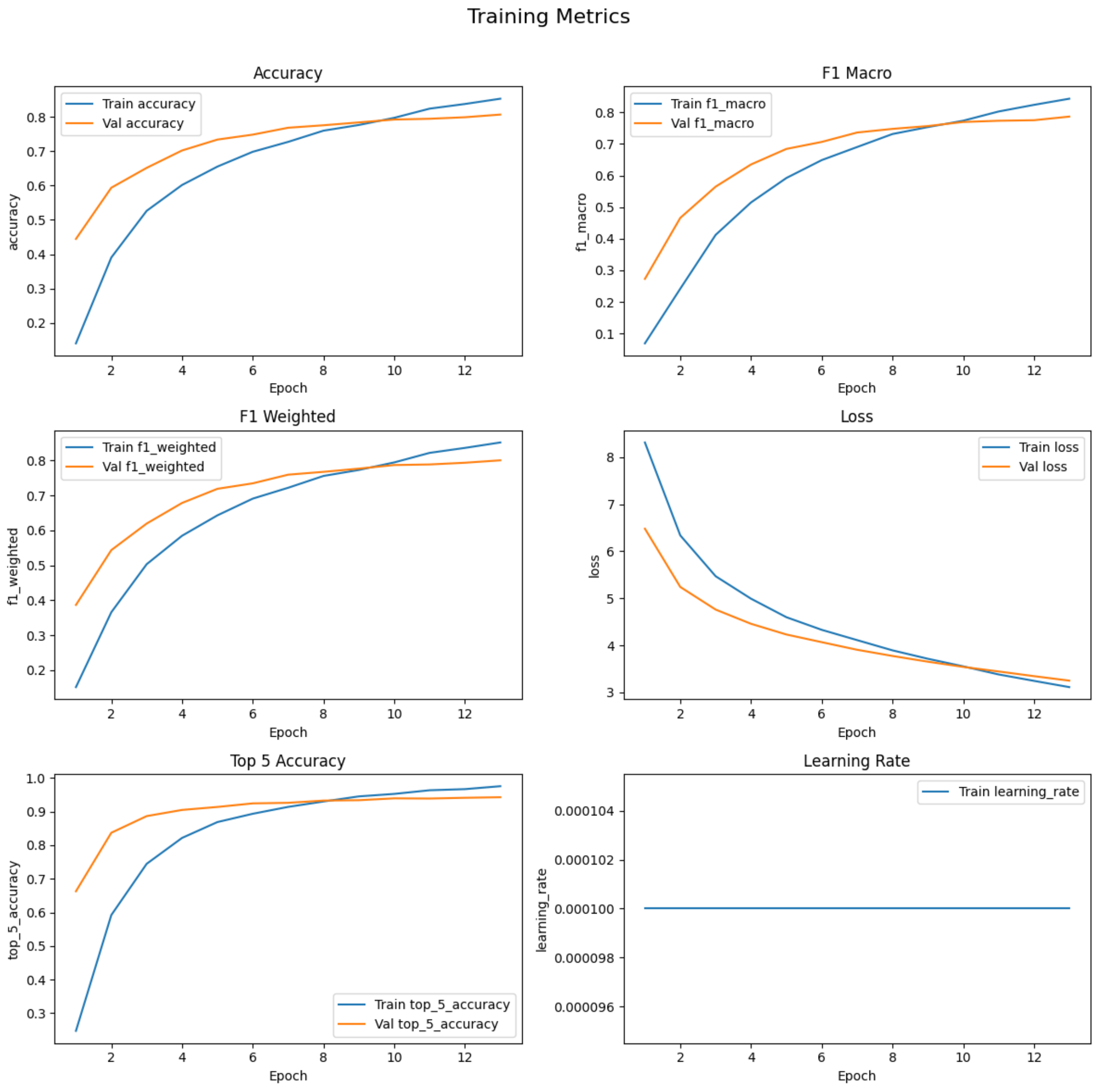


Figure 11: ConvNextFull – Batch 32 Results

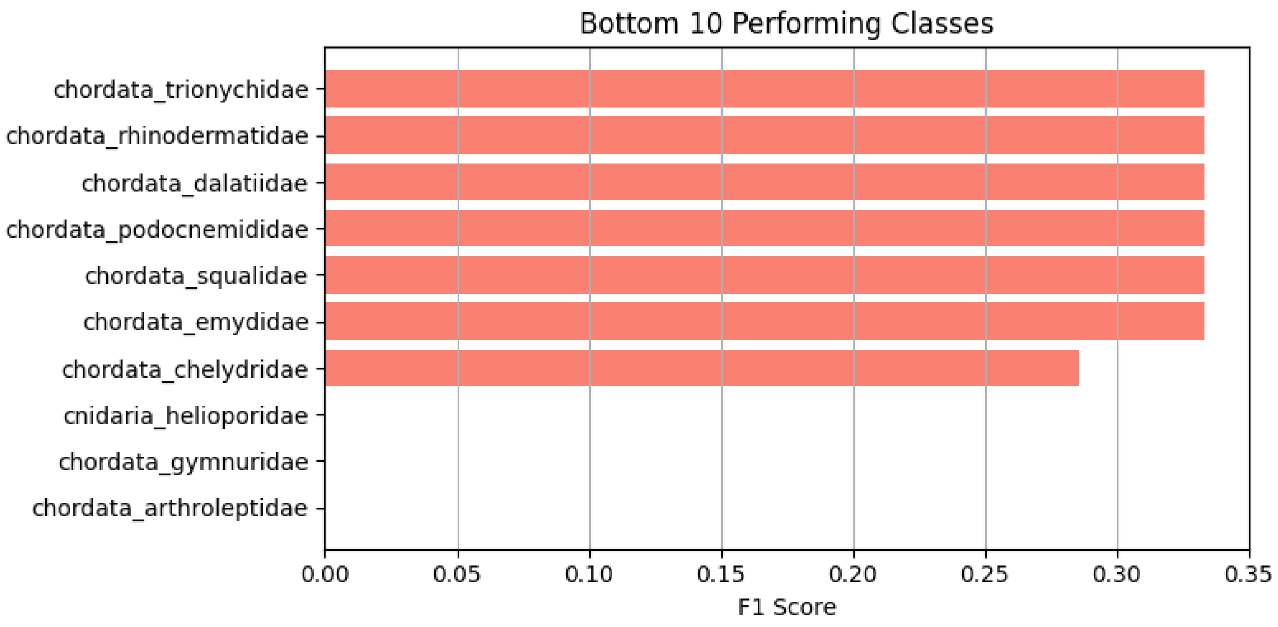
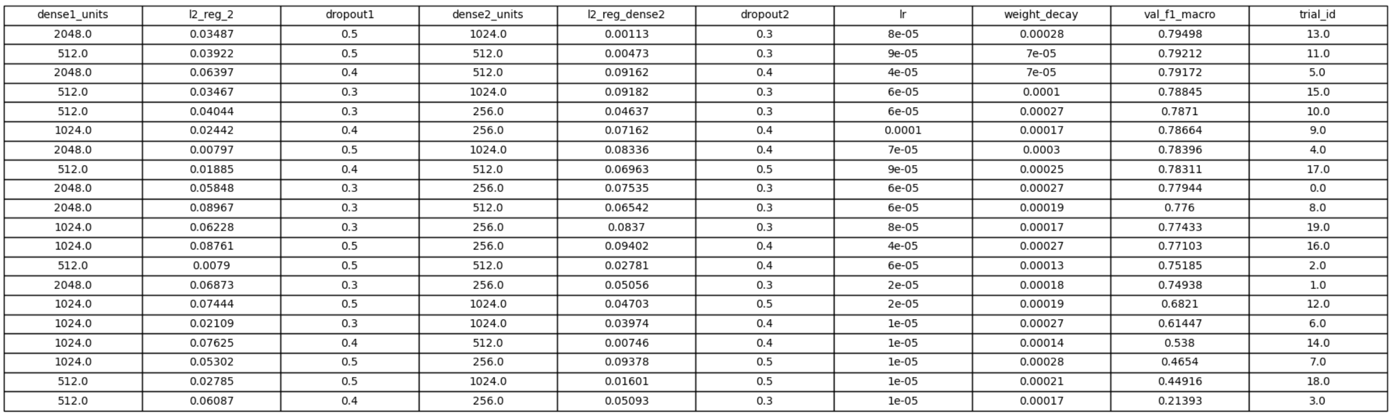


Figure 12: Bottom 10 Perfoming Classes - ConvNextFull with Batch 32



dense1_units	l2_reg_2	dropout1	dense2_units	l2_reg_dense2	dropout2	lr	weight_decay	val_f1_macro	trial_id
2048.0	0.03487	0.5	1024.0	0.00113	0.3	8e-05	0.00028	0.79498	13.0
512.0	0.03922	0.5	512.0	0.00473	0.3	9e-05	7e-05	0.79212	11.0
2048.0	0.06397	0.4	512.0	0.09162	0.4	4e-05	7e-05	0.79172	5.0
512.0	0.03467	0.3	1024.0	0.09182	0.3	6e-05	0.0001	0.78845	15.0
512.0	0.04044	0.3	256.0	0.04637	0.3	6e-05	0.00027	0.7871	10.0
1024.0	0.02442	0.4	256.0	0.07162	0.4	0.0001	0.00017	0.78664	9.0
2048.0	0.00797	0.5	1024.0	0.08336	0.4	7e-05	0.0003	0.78396	4.0
512.0	0.01885	0.4	512.0	0.06963	0.5	9e-05	0.00025	0.78311	17.0
2048.0	0.05848	0.3	256.0	0.07535	0.3	6e-05	0.00027	0.77944	0.0
2048.0	0.08967	0.3	512.0	0.06542	0.3	6e-05	0.00019	0.776	8.0
1024.0	0.06228	0.3	256.0	0.0837	0.3	8e-05	0.00017	0.77433	19.0
1024.0	0.08761	0.5	256.0	0.09402	0.4	4e-05	0.00027	0.77103	16.0
512.0	0.0079	0.5	512.0	0.02781	0.4	6e-05	0.00013	0.75185	2.0
2048.0	0.06873	0.3	256.0	0.05056	0.3	2e-05	0.00018	0.74938	1.0
1024.0	0.07444	0.5	1024.0	0.04703	0.5	2e-05	0.00019	0.6821	12.0
1024.0	0.02109	0.3	1024.0	0.03974	0.4	1e-05	0.00027	0.61447	6.0
1024.0	0.07625	0.4	512.0	0.00746	0.4	1e-05	0.00014	0.538	14.0
1024.0	0.05302	0.5	256.0	0.09378	0.5	1e-05	0.00028	0.4654	7.0
512.0	0.02785	0.5	1024.0	0.01601	0.5	1e-05	0.00021	0.44916	18.0
512.0	0.06087	0.4	256.0	0.05093	0.3	1e-05	0.00017	0.21393	3.0

Figure 13: Random Search Results

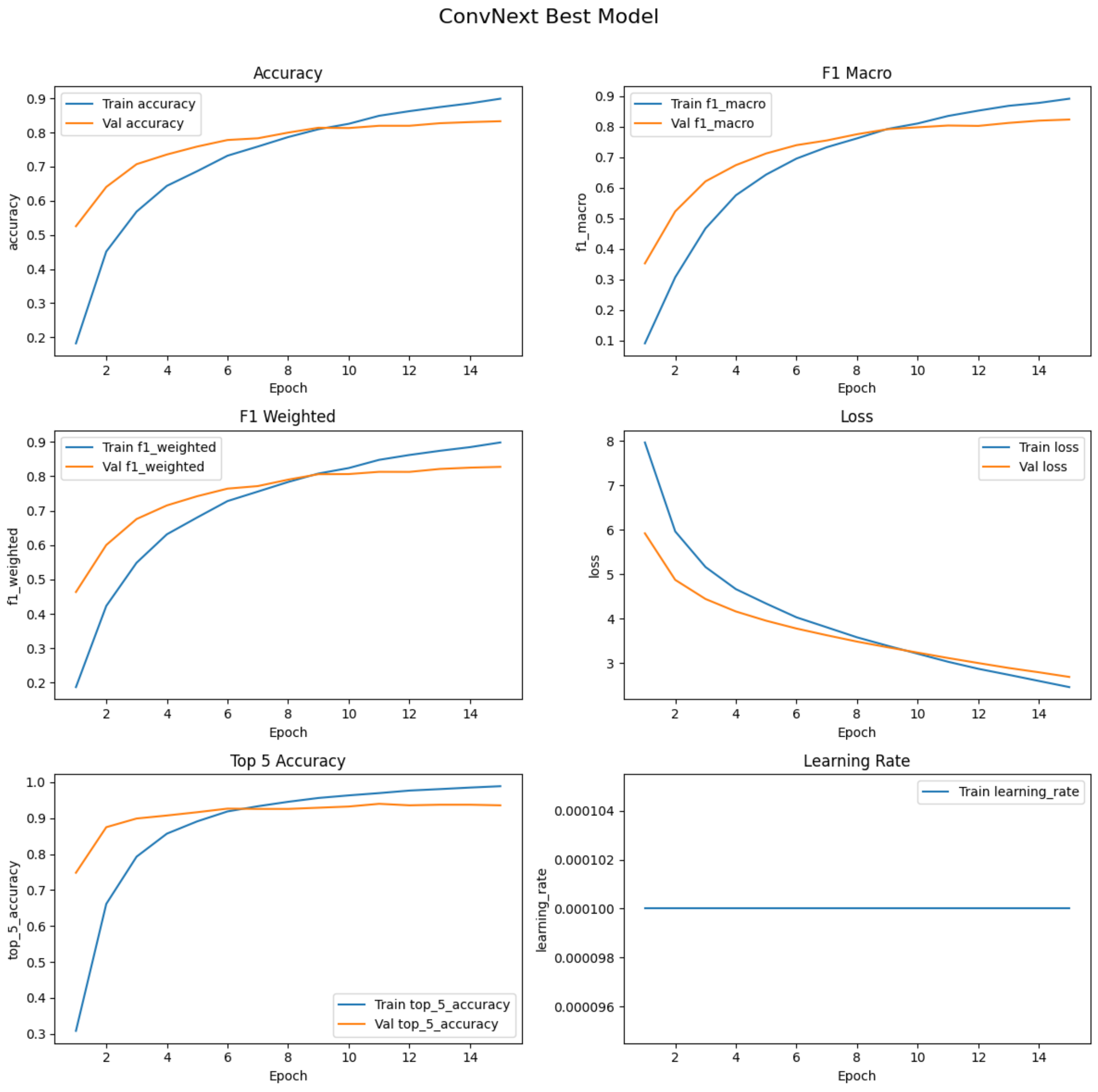


Figure 14: ConvNextFull – Batch 32 Results on the 90/10 Split

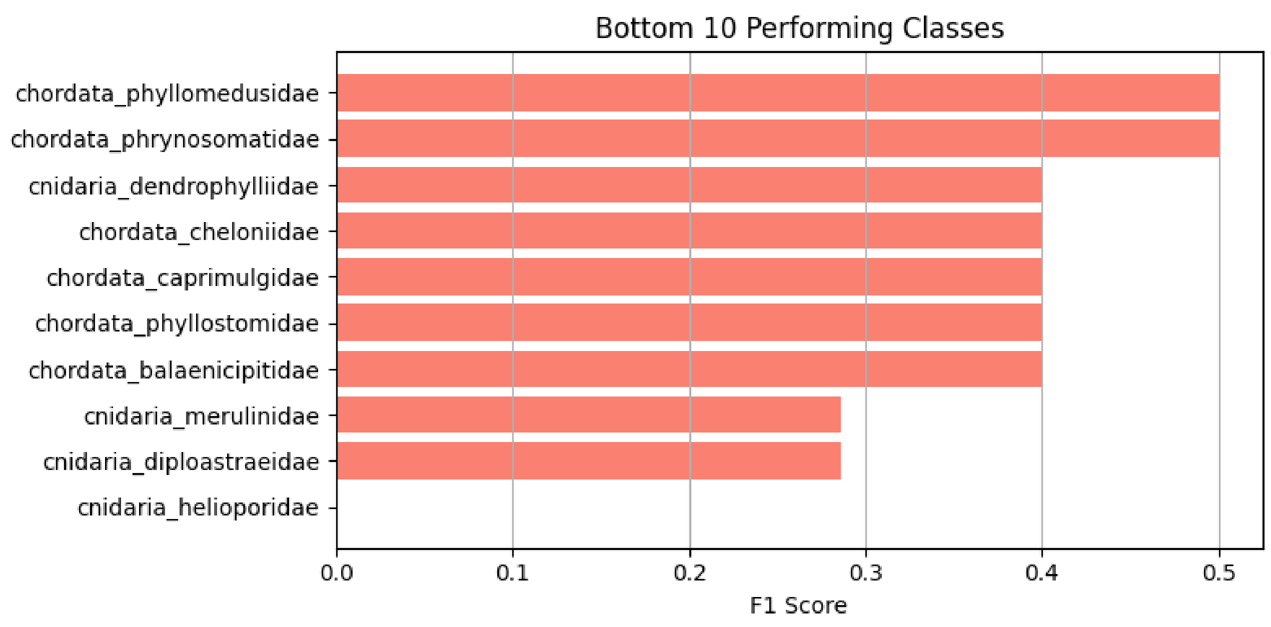


Figure 15: Bottom 10 Perfoming Classes – 90/10 Split