Using AI to classify phenotypes in Australian parakeets: *Melopsittacus undulatus* (Shaw, 1805)

Professor:
Dr. Wei Ding
CS470 - CS670

TAs:
Haowen Guan Haowen.guan001@umb.edu
Mahsa Geshvadi Mahsa.geshvadi001@umb.edu

Team 8
Team leader: Nicole Strounine: nicole.strounine001@umb.edu; Naomi Adebo-Young: naomi.adeboyoung001@umb.edu; Emilia Morgan: emilia.morgan001@umb.edu; Billy Bakalis: william.bakalis001@umb.edu; Viviana Romero Alarcon: viviana.romeroalarcon@umb.edu; Raymond Dugas: raymond.dugas001@umb.edu; William Ding: william.ding001@umb.edu; Svetozar Draganitchki: s.draganitchki001@umb.edu

CONTENT

Abstract

Budgerigars are an Australian-domestic species of parakeets with a variety of discrete phenotypes, which is a relevant advantage to studying image classification in artificial intelligence. In nature, those parakeets exhibit a plumage green with yellow maculate (wild phenotype). Still, the exhaustive inbreeding and domestication process has expressed other phenotypes such as yellow, blue, and white during the last 100 years. Thus, our study explores the application of machine learning techniques to classify the phenotypes of different types of budgerigars based on their unique characteristic traits. To reach our objectives, we used the fastup library to clean up the data set by filtering duplicates, invalid/broken files, outliers, and problematic images that could lead to misleading classifications. Subsequently, images were stored in an hdf5 file for efficient data retrieval and processing when working with the datasets.

We normalized the dataset by scaling numerical data to a standard range from 0 to 1. Then, augmentation consisted of increasing the model robustness by reshaping images by 224 * 224 in the three RGB channels, rotating 60 degrees, flipping horizontally, and changing the width and height. Given that some classes had more elements than others, we balanced the dataset by defining the smaller class size and using that size to sample without replacing the others. Finally, in the preprocessing phase, we split the dataset into train and test sets, considering a ratio of 70:30, respectively. During the project's second phase, we used the Hugging-Face platform to obtain pre-training models that performed tasks in image classification. We selected four models: RestNet18 [1], MobileNet V2 [2], Resnet 152 [3], and the Vision Transformer (ViT) model [11]. All of them were selected following three main criteria: 1) Accuracy, 2) efficiency in small datasets, 3) good performance with computational limitations (CPU), and 3) ease of fitting the code and fine-tuning. As a result, we found that accuracy in our models was strongly influenced by micro decisions such as including zoom-in in the augmentation or checking for mislabeled images using automatic tools or doing it manually. Likewise, balancing our dataset was crucial in our training and validation process because due to that, we achieved an accuracy higher than 85%. Demonstrating success in pre-processing, model selection, and fine-tuning.

Understanding the Dataset

The budgerigars (*Melopsittacus undulatus*) are a species of the parrot family that have been extensively bred because of their plumage traits. It is a domestic species currently but is also considered an exotic and invasive species outside Oceania. The domestication and inbreeding have produced several colorful phenotypes in this species, including an albino phenotype, because of the loss of the psittacofulvin pigments. The genetic base behind those phenotypes is described by a Mendelian heritage, which generates mainly four different patterns such as wild-type, recessive blue-type, yellow-type, and Albin[4]. For this exercise, we will use those phenotypes to perform a supervised classification to detect the phenotype type depending on the pigments expressed on the feather. Thus, the used classes will correspond to the most common phenotypes.

- Wild Budgies: This category comprises images depicting budgerigars with the characteristic green plumage, yellow face and abdomen, black wing markings, and blue cheek patches, representing the natural coloration of wild budgerigars.
- Blue Budgies: In this category, you will find images showcasing budgerigars with a blue color mutation, where the typical green plumage is replaced by varying shades of blue, often retaining the signature black wing markings and blue cheek patches.
- Yellow Budgies: This category features budgerigars with vibrant yellow plumage resulting from specific genetic mutations inhibiting melanin production. These birds exhibit a range of shades from pale yellow to deep golden, adding a bright and cheerful presence to the dataset.
- White Budgies: The white budgie category includes images of budgerigars with pure white plumage resulting from genetic mutations suppressing pigment production. These birds may have dark or pink eyes, contributing to their unique appearance within the dataset.
- Others: In this category, two or more budgies of different colorations exist in a single image.

Pre-processing: Phase I

See Phase1_FinalColab.ipynb for related code

Cleaning Data

Our initial dataset consisted of 384 images distributed in five classes, with 135 in the blue class, 80 in the wild, 53 in white, 65 in yellow, and 51 in the others. Given the evident unbalance among classes (See below Figure 3A), we decided to include images from the INaturalist database [5]. After that, we got a dataset composed of 138 in the blue class, 151 in the wild, 77 in white, 83 in yellow, and 70 in the others (See below Figure 4A). To process that final database, we focused on finding anomalous images that could bias the training and the model validation. It was a semi-automatic task, in which, first of all, we did a manual cleaning to discard images with objects different from budgerigars, discard pictures smaller than 9 Kb, and relocate or

retag images. We got 309 images in total after the manual checking. Secondly, we used the fastDUP library [6] as an automatic process to delete corrupt files and detect anomalies such as dark, bright, and blurred pictures. This library also allowed us to detect images with many extra objects that could cause problems, considering them outliers. Finally, using the nearest neighbor search, we detected duplicate photos. Thus, duplicates were the most problematic factor during the cleaning process, and outliers were the second. In contrast to over or underexposure, which is the most common noise in the databases, we did not find it as a part of our problems (Figure. 1). Finally, we got a clean data set with 264 images.

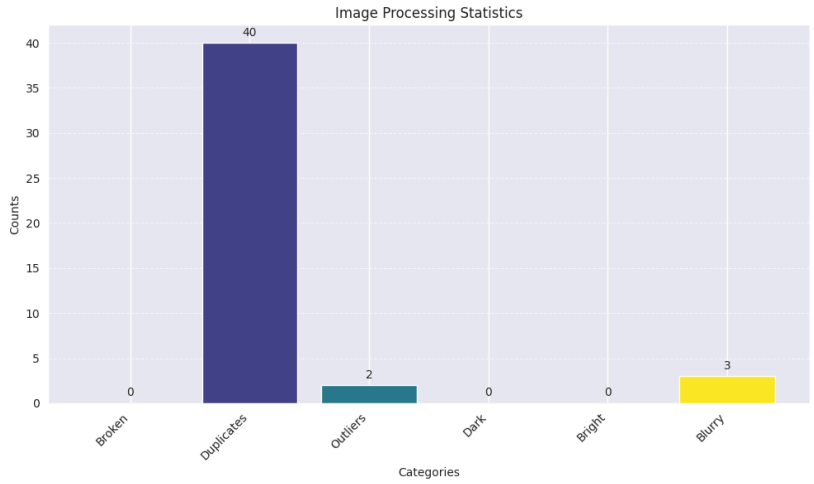


Figure 1. Distribution of problematic factors in the cleaning process.

Data Normalization and Resizing

In this task, we decided to resize the images to a target of 224 * 224, always preserving the aspect ratio. We converted the RGB format to an alpha channel to posteriorly normalize pixels in a range of 0 to 1. Then, we stored images in an hdf5 file to allow for efficient data retrieval and processing when working with the datasets. All those tasks use OpenCV, Numpy, and H5py libraries [7,8,9].

Data Visualization

Visualization was a transversal objective in every step of our pre-processing workflow. For instance, we created galleries to check duplicates and dark or anomalous images using the tools offered by the FastdUP library [10]. Also, we used basic Matplotlib functions [11] to create bar

plots (i.e. Figures 1 and 3) to check distributions or make decisions about thresholds. Likewise, we used the same library to plot images with their labels to check the right label assignment. For example, for Figure 2, we used the library under “Process Visualization” in the Colab (before we call it later). It visualizes the first five images but can be set to visualize more or less based on what n is set to.

Figure 2. Checking label assignation and classes

Data Augmentation

We played around with various augmentation techniques based on discussions about replicating photos that someone might take of a bird (Box 1). We figured that rotating the images is a fair representation of possible photos someone might take, exactly with flipping the images. We initially had the zoom range set, given that someone might take a zoomed-in photo. Still, after visualizing some of the zoomed-in images, we decided that these were not relevant for pictures that did not necessarily show a budgie clearly, so ultimately, we went with just small width and height ranges (especially since we had to ensure all our photos were the same size by filling out the various images to 256x256 in the first phase, and then switching to 224x224 for phase 2 based on what our models necessitated.

```
datagen = ImageDataGenerator(
  rotation_range=60,
  width_shift_range=0.2,
  height_shift_range=0.2,
  #zoom_range=0.01,          # maybe get rid of zoom
  horizontal_flip=True,
  fill_mode='nearest'
```

```
)
```

Box 1. Script used for augmentation

Data Splitting

During the curation process, we kept the order of the data as described above, conserving the initial categories without splitting them into subsets for training, testing, or validation. Thus, for the splitting task, we initially only distributed the categories into training (70%) and testing (30%) subsets but realized that the way we were calling our training epochs required an additional set for validation. We split our dataset into three subsets: the testing set consisted of 30% of our original images, and the validation set consisted of 20% of the 70% of the initial training images. We kept 70% of our images for training because we had so few images to begin with. We augmented the training data images to increase the number of images to around 1000 and then split this data set (with augmented and unaugmented images) into training and validation.

Initially our dataset had imbalanced classes which resulted in lower accuracy in our models, since blue budgies were over represented, while other classes were underrepresented. We balanced the classes before splitting by finding the minimum class size and then randomly selecting the minimum amount of images for each class so they would be equal in size. Then we split the data. So training had around 400 photos, and testing had around 200, with about as equal class representation as possible. This greatly improved accuracy as is detailed in the ResNet 18 documentation.

Reading Out and Saving the Data:

Given we were using one long Colab for the data preprocessing, we wanted to save the images in files that could be easily shared so people didn't have to rerun the Colab every time they wanted to access the data. Initially, we tried doing this with CSV files, but they were too big to download and send on our computers. We spent a long time trying to get this done and then learned about H5 files, which are a way to save data with less memory. We quickly turned to

this to download the image data sets and label data sets. The image data sets were still large, so we zipped them after saving them as .h5 files. To open them in the other Colabs, we unzipped the x_test and x_train files and then read all four files in as .h5 files to Numpy arrays to turn them into tensors like we had done previously for homework to use for our models.

Model Selection: Phase II

This phase consisted of three steps: research and selection of the model, modification of the code, and fine-tuning of the model. We took advantage of pre-trained models from the Hugging-Face platform and used transfer learning to achieve our task of classifying Budgerigars. For that objective, we fine-tuned three models: ResNet18 [1], MobileNet V2 [2], and ResNet 152 [3]. We explored several other models as well, such as EfficientNet and Google VIT, however there were issues with these models so we decided to focus on the two ResNet models and the MobileNet model.

ResNet18 Model

See: ResNet18.ipynb for code

Model Explanation

Introduced in 2015, ResNet18 is an 18-layered, convolutional neural network that performs image classification. It is pre-trained on ImageNet to classify images into 1000 classes. Using our training dataset, we trained this model to classify the five different categories of budgerigars.

Why we chose ResNet18

When choosing a model, we researched what the most popular image classification models were. One that appeared at the top of many lists was ResNet 18. It is a relatively small model, with only 18 layers. This makes it efficient and suitable for small datasets such as our own since the small size and few parameters reduces the risk of overfitting. Another reason for choosing ResNet 18 was computational resources. This model's size makes it much less computationally expensive than other larger models [Colab AI]. When working with Colab, there is a limited

amount of GPU available to the user, so if we were limited to using CPU, many of the larger models simply wouldn't run. ResNet 18, however, would run using CPU, albeit slowly at two minutes per training epoch. This made it one of the more efficient models to use.

Training dataset

We used the image dataset that underwent preprocessing in Phase 1. This dataset was composed of various original and augmented images of the five categories of budgies, with each image resized and normalized to a standard of 224 by 224.

Fine Tuning and Results

When fine-tuning this model, we based our code on the fine-tuning example that was shown in the lab session for Phase II. Most of our effort was spent modifying our data preprocessing to first fit the model, and then to improve accuracy. One of the first adjustments we made was resizing. Originally we had chosen to resize our images to 256 x 256, but upon further investigation, it seemed most models needed data to be 224 x 224. We also had to get our y_train and y_test data, which contained the labels, to be in the right format. Originally in our preprocessing, we had them as strings but we decided it would be simpler to use numerical data. Using an encoder, we transformed the string labels into integers 0-4 to represent the five classes.

Once our data was properly formatted and loaded into test and train data loaders, we loaded the ResNet18 model. We adjusted the final layer of the model to have five output classes. This was important because the original model was trained to classify images into 1000 classes. However, by changing the final layer to match the number of classes, we could benefit from the pretraining while using the model for our task.

One major aspect affecting our accuracy was class imbalance. Going back into preprocessing, we were able to see the class counts for all of our classes at the end of preprocessing. The blue budgies category was overrepresented (Figure 3A), affecting the accuracy of the training (Figure

3B). Because of this, our model ended up skewed towards blue budgies. After we made adjustments to the learning rate, tried different optimizers, and fine-tuned in other ways, the highest accuracy we could achieve with this imbalanced data was ~78%.

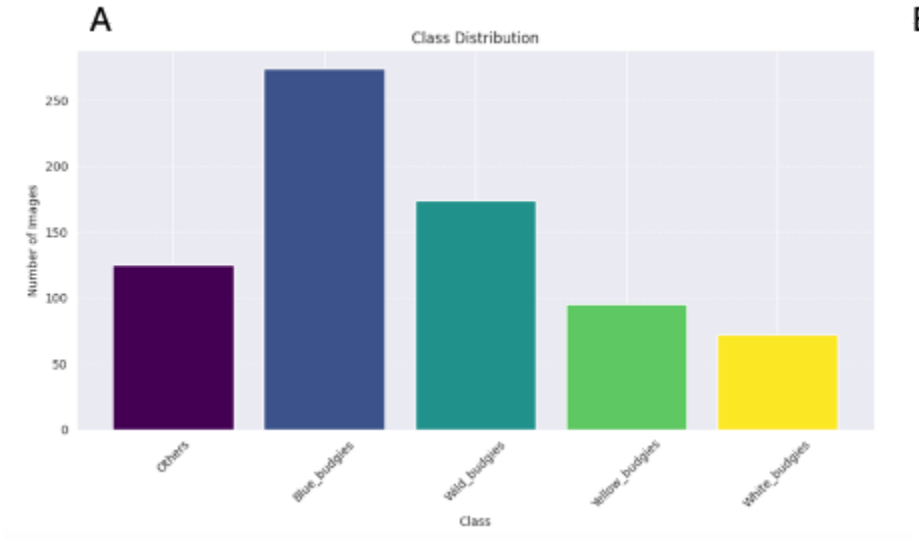


Figure 3. Unbalanced Dataset. **A** Distribution of the initial dataset without including INaturalist or balancing it. **B** Confusion matrix shows a low accuracy and precision after the first training.

Since the data was imbalanced, we needed to balance it. We did the balancing after data augmentation and before data splitting so that each class had roughly an equal amount of images. Once the classes were balanced (Figure 4A) and we ran the model, we could see from the output that the accuracy improved to ~85% (Figure 4B).

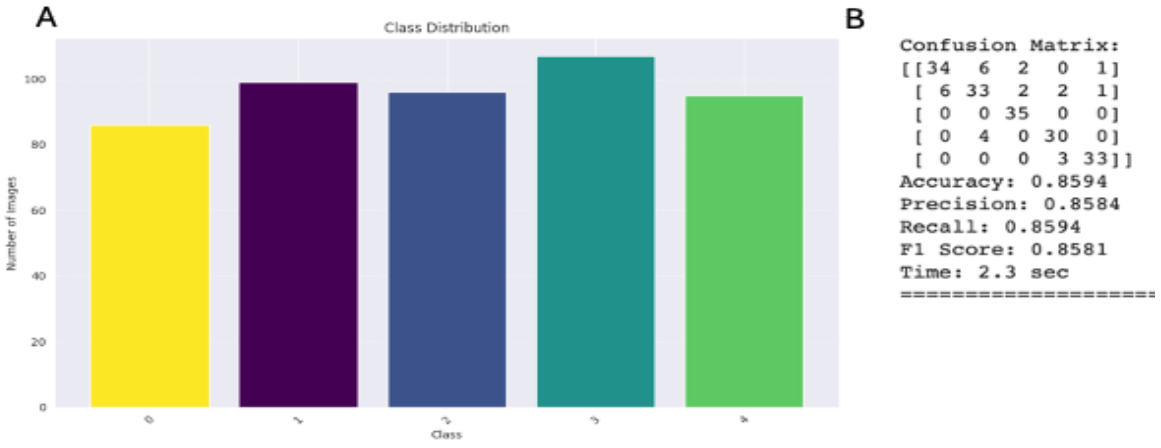


Figure 4. Balanced Dataset. **A** Distribution of the initial dataset, including INaturalist and balance processing. **B** Confusion matrix shows higher accuracy and precision after using a balanced dataset for training.

Inspecting the confusion matrix further, we were able to see that the “others” class was being misclassified the most often. In order to understand why this was happening, we manually inspected our dataset to see if we had missed something in our data cleaning. Upon inspecting the images in the “others” folder, we found several images that contained just a single bird, when the “others” category should only contain images with more than one type of budgie. To solve this we had to go back into the data cleaning to make adjustments so that the incorrect images were either deleted or moved to the correct folders. Now, when we ran our model with the updated cleaning, we saw the accuracy go up to ~94% on some training runs (Figure 5).

Figure 5. Visualization and training summary. The confusion matrix shows accuracy and precision over 94%.

Another thing we experimented with was different epochs, different size batches, different learning rates, and different optimizers. We found that after about 15 epochs, the model’s accuracy improved no further, in fact it occasionally began to decrease past 15 epochs (Figure 6A, B). With batches, the smaller the batch size, the longer it took to run, however when connected to GPU even small batch sizes ran fairly quickly. Our dataset is relatively small, so we

decided to stick with a small batch size of 16. With a small batch size the model is able to update the weights more frequently and reduce overfitting. For optimizers, we tried Adam and RMSProp. Adam was by far the best, giving an accuracy of 95% (Figure 6C). There is a trade off of computation time, as with Adam each epoch took ~4 seconds while SGD and RMSProp both took ~2 seconds. If time is not a limiting factor for your task, then the difference is not a deal breaker. However, if speed is required, settling for the lower accuracy may be sufficient.

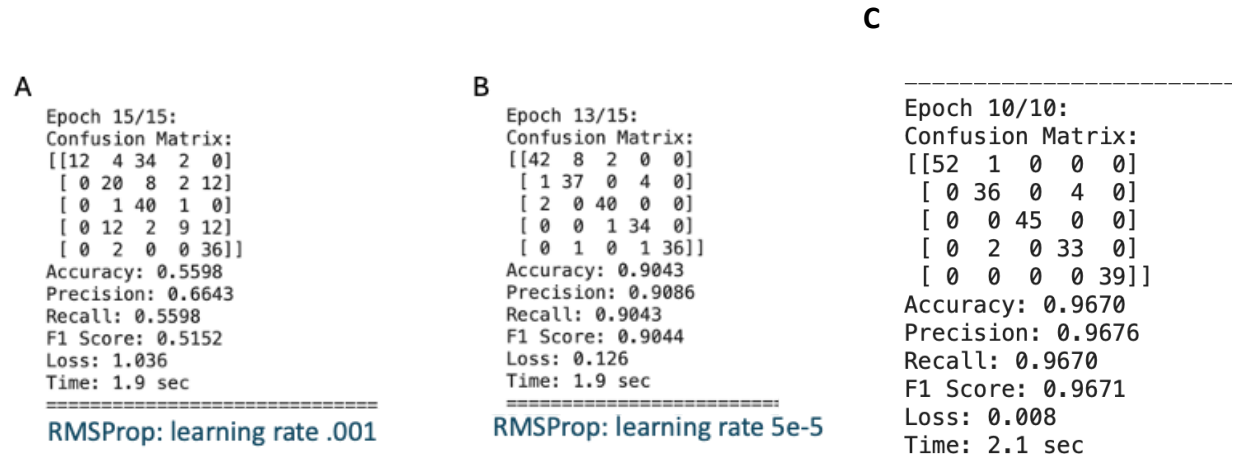


Figure 6. Improved accuracy, given the epoch number and optimization type. **A** RMSProp: learning rate .001, **B** RMSProp: learning rate 5e-5, **C** Adam: learning rate 5e-5.

Potential Improvements

There is an element of randomness in our preprocessing, during data augmentation and data splitting. Each time the Phase 1 colab is run, the final dataset generated is not the same as previous iterations of that dataset. Every iteration contains differently augmented images, and those images are split between test and train sets differently and randomly every time. This affects the accuracy of the model. An improvement that could be done is doing the class balancing after data augmentation and data splitting, ensuring that the classes all have exactly the same amount of images instead of roughly the same, removing that specific element of randomness from the preprocessing.

Conclusions

A major takeaway from fine tuning ResNet 18 was the importance of good data preprocessing. Ensuring the dataset contained correct and clear images as well as ensuring that each class of images was equally represented in the training set made the biggest difference in accuracy. Even small things, such as a couple of mislabeled images in the incorrect category would affect the training, causing the model to incorrectly predict the labels.

Something we noticed as well was that the “Others” category was the one most often misclassified. Keeping the rest of the classes equal but increasing the representation of the “others” class could potentially help the model learn to classify it correctly. This is not an adjustment we had time to implement, but it would be interesting to see how it would affect the output.

MobileNet V2 Model

see MobileNet V2.ipynb for code

Model Explanation

Developed by Google in 2018, MobileNet V2 is a lightweight model designed for mobile and embedded devices. It is trained on ImageNet to classify images into 1000 classes. Its lightweight architecture helps reduce overfitting. [Colab AI]

Why we chose MobileNet V2

We chose this model for similar reasons as ResNet 18, it is a small and efficient model that is good for small datasets and has a low computational cost. Also, we were able to very easily modify the code from our ResNet18 model to fit the MobileNet V2 model.

Training dataset

We used the image dataset that underwent preprocessing in Phase 1. This dataset was composed of various original and augmented images of the five categories of budgies, with each image resized and normalized to a standard of 224 by 224.

Fine Tuning + Results

Since we used the same code as our ResNet18 model, we didn't have to focus on adjusting the preprocessing. Instead we focused on trying out different optimizers, batch sizes, and learning rates. With SGD optimizer, and a learning rate of 5-e5, we got an accuracy of 89% which was already very good, but we wanted to see if we could do better.

SGD, batch size 16, lr 5e-5:

```
Confusion Matrix:
[[45  1  6  0  0]
 [ 4 30  1  7  0]
 [ 0  0 42  0  0]
 [ 0  1  0 33  1]
 [ 0  0  1  2 35]]
Accuracy: 0.8852
Precision: 0.8940
Recall: 0.8852
F1 Score: 0.8837
Loss: 0.138
Time:4.4 sec
----------------------
```

SGD, batch size 64, lr 5e-5

```
Confusion Matrix:
[[43  4  5  0  0]
 [ 3 33  1  5  0]
 [ 0  0 42  0  0]
 [ 0  0  0 35  0]
 [ 0  1  0  2 35]]
Accuracy: 0.8995
Precision: 0.9043
Recall: 0.8995
F1 Score: 0.8983
Loss: 0.081
Time:3.4 sec
```

Next, we tried using a different optimizer, Adam, with batch size 16 and different learning rates. With this optimizer, the accuracy was still around 90%.

Adam, batch size 16, lr 5e-5

```
Confusion Matrix:
[[43  1  6  2  0]
 [ 5 33  0  3  1]
 [ 0  0 42  0  0]
 [ 0  1  0 32  2]
 [ 0  0  0  1 37]]
Accuracy: 0.8947
Precision: 0.8974
Recall: 0.8947
F1 Score: 0.8931
Loss: 0.032
Time:2.9 sec
----------------------
```

Adam, batch size 16, lr .001

```
Confusion Matrix:
[[49  0  3  0  0]
 [ 5 32  1  4  0]
 [ 0  0 41  0  1]
 [ 0  1  1 31  2]
 [ 0  0  0  0 38]]
Accuracy: 0.9139
Precision: 0.9166
Recall: 0.9139
F1 Score: 0.9120
Loss: 0.004
Time:2.7 sec
```

The final optimizer we tried was AdamW, which is similar to Adam but there is weight decay which can be adjusted. Weight decay is a technique used to prevent overfitting by adding a penalty term to the loss function, encouraging smaller weights and discouraging the model from learning undesirable behaviors. [Colab AI]

AdamW, lr .001, weight decay .01

```
Confusion Matrix:
[[51  0  1  0  0]
 [ 1 41  0  0  0]
 [ 1  0 41  0  0]
 [ 0  0  0 35  0]
 [ 0  0  0  1 37]]
Accuracy: 0.9809
Precision: 0.9812
Recall: 0.9809
F1 Score: 0.9809
Loss: 0.000
Time:2.7 sec
```

Adam, lr .001, weight decay .05

```
Confusion Matrix:
[[46  2  3  1  0]
 [ 0 39  0  3  0]
 [ 1  0 41  0  0]
 [ 0  1  0 34  0]
 [ 0  0  0  2 36]]
Accuracy: 0.9378
Precision: 0.9415
Recall: 0.9378
F1 Score: 0.9382
Loss: 0.012
Time:2.8 sec
```

The final best configuration of MobileNet V2 was using the AdamW Optimizer with a batch size of 64, a learning rate of .001 and a weight decay of .01. Although lower learning rates usually improve accuracy, learning rates that are too low such as .00005 can cause overfitting and won't achieve good results. Increasing the weight decay also lowered the accuracy. We also found that batch size didn't change the accuracy that much with SGD and Adam, but a larger batch size of 64 gave the best results for AdamW.

Potential Improvements

We spent most of our time working with the optimizer to find the best fit, and stopped once we got a high accuracy of 98%. However, we did not look into different loss functions. It would have been good to see how changing the loss function affected the model.

Conclusion

A major takeaway from fine tuning MobileNet V2 was the importance of finding a good optimizer and using the correct learning rate. Although SGD gave us good accuracy, if we hadn't

tried Adam, we wouldn't have been able to improve our accuracy up to 98%. We then went back to our ResNet 18 model, to try out different optimizers. However, MobileNet V2 with AdamW was by far the best model.

Resnet152 Model

See Resnet152.ipynb for code

Model Explanation

This updated version of ResNet18 can be used for the classification of images. The upgrades from Resnet 18 were achieved by avoiding downsampling which makes the model slightly more accurate. In the original ResNet v1, the downsampling is accomplished with a stride of 2 in the first 1x1 convolution, whereas in ResNet v1.5, the stride is implemented in the 3x3 convolution. Additionally, ResNet 152 has the advantage of having an existing implementation that utilizes image binaries instead of opening images directly. However, this caused issues, particularly with large batch sizes leading to exponential label increases. This problem was successfully addressed, resulting in significant RAM savings.

Why We Chose Resnet 152

We chose Resnet 152 because it's supposed to be more accurate than the regular ResNet. We had success with regular ResNet and figured it would be interesting to explore an updated version of our classification problem.

Training DataSet

We used the image dataset that underwent preprocessing in Phase 1. This dataset was composed of various original and augmented images of the five categories of budgies, with each image resized and normalized to a standard of 224 by 224.

Fine-Tuning the Model and Results

Similar to ResNet18, we had to change the final layer of our model to classify our images into 5 classes as opposed to 1000 classes which the model was originally trained to do. We tested 7

different optimizers, Adam, SGD, RMSprop, Adagrad, Adadelta, Adamw and Rprop. Adam and SGD showed far better results (we trained each with 15 epochs and 2 accumulation steps for comparison), if we had had more time we could have tried multiple epochs/accumulation steps as well as changing the various variables for each optimizer to determine more clearly with optimizer is better. RMSprop had accuracy of around 30%, Adagrad and Adadelta had an accuracy of about 40%, Adamw had an accuracy of around 50%, and Rprop had an accuracy of around 60%. With Adam our accuracy (which is only one metric) we got to 87%, and with SGD accuracy was 85% with 15 epochs and 3 accumulation steps, and momentum set to .9. Below we walk through the steps of fine-tuning with ResNet 152.

Our initial optimizer was set to Adam. With this we explored accumulation steps and epochs. We found that more epochs/fewer accumulation steps had higher statistics. We first trained it on 5 epochs, with 4 accumulation steps, this took a very long time - in part because we had used up all our collab units for the day. For 5 epochs and 4 accumulation steps the accuracy, precision, recall all were around 75% [see image below]

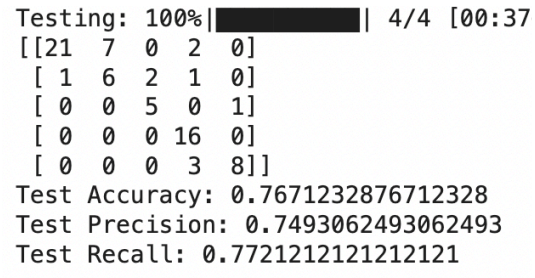


Image to the Left: 5 epochs, 4 accumulation steps, Validation set 20%, Optimizer: Adam, Original dataset that is not balanced

Our statistics improved with higher epochs, fewer accumulation steps:

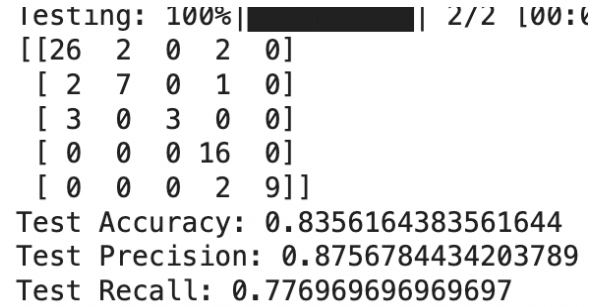


Image to the left: 10 epochs, 1 accumulation step, Validation 20%, Optimizer: Adam, Original dataset that is not balanced

But at 25 epochs, precision and accuracy started to drop, while recall was slightly higher :

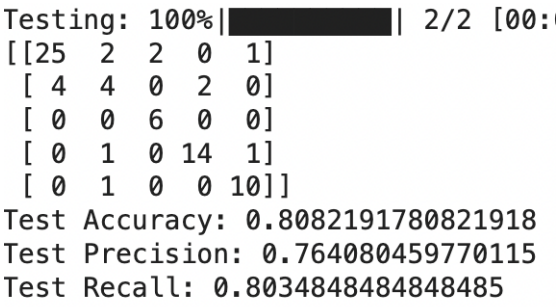


Image to the left: 25 epochs, 1 accumulation step, Validation 20%, Optimizer: Adam, original dataset that is not balanced (slightly higher recall, lower precision and accuracy)

Given precision and accuracy being slightly lower, we decided to keep our epochs at 15, while exploring how changing the accumulation steps affected the results.

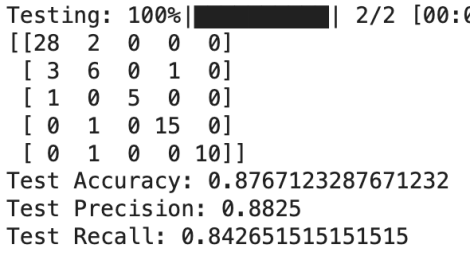


Image to the left: 15 epochs, 2 accumulation step, Validation 20%, Optimizer: Adam, original dataset that is not balanced (higher recall, precision and accuracy than any of the other attempts)

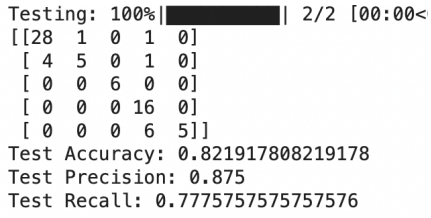
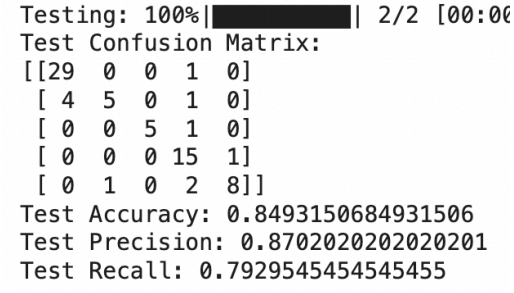


Image to the left: 15 epochs, 3 accumulation step, Validation 20%, Optimizer: Adam, original dataset that is not balanced

As we can see, 3 accumulation steps lowered the accuracy by about 5% points, the precision by 1% point, and the recall by 7% points. So for Adam optimizer, around 15 epochs with 2 accumulation steps seemed to produce the best result. 15 epochs makes sense in that the more epochs with an unbalanced data set can lead to overfitting. It would be interesting to analyze with the balanced data set how many epochs/steps produce the best results.

Image to the Right: 15 epochs, 3 accumulation step, Validation 20%, Optimizer: SGD, original dataset that is not balanced



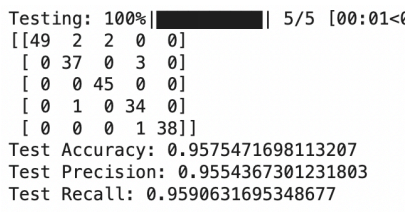


Image to the Left: Balanced data set with Adam as optimizer, 15 epochs, 2 accumulation steps

It is clear that with the balanced data set we get the best results, it would definitely be interesting to play around with the optimizers, accumulation steps, and epochs with a balanced data set to see if we get the same results for highest accuracy.

Potential Improvements

There are many more variables that we could spend time adjusting and trying for every different optimizer. Since we spent so much of our time focusing on the optimizer, we did not change the loss function much.

Conclusion

ResNet 152 is indeed a larger and more computationally complex version of ResNet 18, and can get a very high accuracy. Although we didn't get the accuracy to be significantly higher than our ResNet 18 model. This is because for our relatively small dataset, ResNet 152's complexity can actually cause more overfitting than ResNet 18. Ultimately the choosing one over the other would depend on computational resources and time versus the need for high accuracy. For applications such as mobile or embedded systems with low computational resources and a small dataset, ResNet 18 would be sufficient, and in fact perform better than ResNet 152. However, if we had a large dataset with thousands of images, and more computational resources, ResNet 152 could produce better results.

Vision Transformer (ViT) Model

See VIT-based_withconfusionmatrix.ipynb for code.

Model Explanation

The Vision Transformation (ViT) model we chose is actually a fine tuned version of Google's ViT(base-sized model) by gungbgs for a bird species image classifier. The ViT model uses the transformers architecture, which was initially designed for natural language processing, in other words, a sequence of words in a sentence. The idea behind Vision Transformation is to treat an image as a sequence of image patches. Then, given these patches, flatten them into a sequence feedable to the transformer encoder, which then captures spatial relationships and dependencies among the patches. This approach's main advantage is its ability to scale to larger datasets with state of the art performance as shown by Google, originally pre-training the base model on ImageNet-21k(14 million images, 12,843 classes) at resolution 224x224.

Why We Chose finetuned ViT model

Being that our dataset is all budgies (birds), we figured that a bird specific image classifier could give high accuracy. The fine tuned model was trained on the birds_species_dataset and uses a different architecture for computer vision from convolutional neural networks (CNN). Exploring another approach for this task is the main idea since the previous models use CNN.

Training DataSet

From Phase 1, we utilized the preprocessed image dataset, which contained various original and augmented images with 5 classes of budgies. Each image was resized and normalized to adhere to a standard dimension of 224x224.

Fine-Tuning the Model + Results

Upon first running the code, the model would classify most, if not all, images under one label. This is called mode collapse or label collapse. This occurs when the model fails to learn meaningful representations from the data and instead converges on a trivial solution by predicting the same label for all inputs. Overfitting could be a cause for this result since the

largest class was much larger than the rest, potentially influencing the training. Some techniques to mitigate this event are regularization, adjusting dropout rate, and increasing the size or diversity of the training data, data augmentation.

Below is a confusion matrix of the predicted (columns) and correct (rows) labels for one of our first runs.

5 epochs, 4 accumulation steps, 20% validation data, Optimizer: Adam

```
Testing: 100%|██████████| 4/4 [00:43<(
[[30  0  0  0  0]
 [10  0  0  0  0]
 [ 6  0  0  0  0]
 [15  0  0  0  1]
 [10  0  0  0  1]]
Test Accuracy: 0.4246575342465753
Test Precision: 0.18450704225352113
Test Recall: 0.21818181818181817
```

Although the model was 42.5% accurate, as you can see, one column holds most of the labels, meaning the model predicted label_0 97.3% of the time. This is not accurate and not what we want. Essentially, the model's accuracy will be, at best, the percentage of correctly predicted labels in label_0. In other words, the percentage of label_0 in the test dataset.

Since our starting dataset was small, we already increased its size and diversity with augmentation in Phase 1. So, it was thought that maybe the encoder is somehow messing with the data to cause the overfitting because transformer encoders are a difference in data processing between ViT and the CNN based classifiers.

Here are our results with OneHotEncoder()

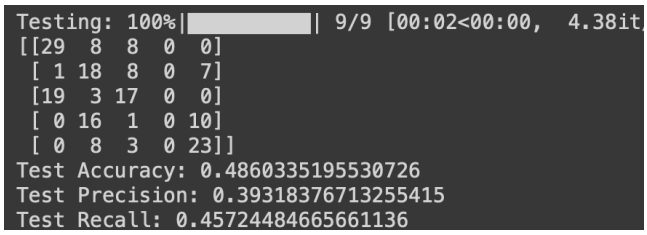
7 epochs, 4 accumulation steps, 20% validation data, Optimizer: Adam

```
Testing: 100%|██████████| 9/9 [00:01<00:00,  4.56it/s]T
[[37  8  0  0  0]
 [11 20  0  0  3]
 [36  3  0  0  0]
 [ 3 19  0  0  5]
 [ 1 15  0  0 18]]
Test Accuracy: 0.41899441340782123
Test Precision: 0.28409090909090906
Test Recall: 0.38797385620915026
```

For this run, the number of epochs was changed from 5 to 7. It was found that 7 epochs was a 'sweet' spot for the model. The classifier improved, predicting more than 1 label, however, only achieved 41.9% accuracy. Notice that test precision and test recall both increased from the first result, by 9.95% and 16.98% respectively. This result was found by only increasing the epochs.

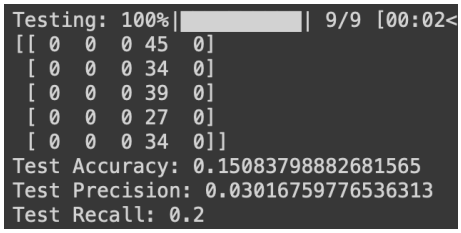
Here are our results without OneHotEncoder()

7 epochs, 4 accumulation steps, 20% validation data, Optimizer: Adam



To avoid using the encoder and the encoded data, the model and the datasets had to be initialized first before doing the test run. The model's accuracy increased by 6.71%. Test precision and test recall also increased, 10.91% and 6.93% respectively. This time the model predicted 4 of the 5 labels, which is also an improvement although still not desirable. This means with a balanced test set, by only predicting 4 of 5 labels the model can at best achieve ~80% accuracy if it predicts perfectly for the 4 of 5 labels. So, the strategy of avoiding OneHotEncoder and adjusting epochs was successful at mitigating mode collapse, however it did not solve it completely. The model is unable to successfully classify all labels, a limiting factor to accuracy.

14 epochs, accumulation steps 4, 20% validation data, Optimizer: Adam



Additionally we found that for this model, mode collapse became a major problem when epochs were <=6 and >=8. More training does not mean more performance but actually less. Here, accuracy, precision, and recall, all decreased dramatically. Also, the model failed to predict more than one label.

At best, the accuracy hovers around 40-49%. This is because the model fails to predict all 5 labels. Failing to predict 1 label, drops the potential maximum accuracy by 20% and in some runs the model fails to predict 2, 3, and 4 labels.

Conclusion

The ViT model was unable to achieve an accuracy greater than 50% in all test runs. The model was also unable to predict all 5 labels for our dataset. These facts together made ViT not a good choice. It did not perform as well as our other models and in theory is unable to surpass 80% as it runs. The strategy of using a confusion matrix to observe how the model is classifying the images was a major breakthrough for understanding the ViT model's performance and behavior. Also, the fine tuning strategies were successful in mitigating mode collapse because the number of predicted labels increased with each change although still inaccurate. This could be due to the architecture transformers.

Conclusions

When creating an image classification model, good training data is essential. Although our data preprocessing was fairly thorough initially, there were some things we overlooked, such as the class imbalance and mislabeled images in folders, some images labeled as "others" had only a single bird in the photo, we did not realize this the first time around. Furthermore, the initial data set had a serious overrepresentation of blue budgies. For Phase 2, we saw overfitting in various models, with models classifying a large portion of other classifications as Blue Budgies, given this we decided to go back, and balance the data set, and see what kind of improvement that would have with both training and testing our models.

Another very important part of finetuning was selecting a good optimizer. Although the base optimizer we used (in some cases SGD, in other cases Adam) did produce decent results of up to 89% in some cases (with the unbalanced data set), if we had stopped there and not tried different optimizers with different learning rates and other parameters, we would not have gotten such high accuracy across 3 of our 4 models.

One oversight was that we did not experiment with loss functions. We managed to get high accuracies across all our models despite this. However, it could have been beneficial to see how different loss functions affected the results.

There are a lot of variables that can be changed when training models, and these can have a huge impact on accuracy, precision, recall, etc. While we specifically highlighted accuracy

as a measurement given our multiple classes, and the kind of problem we were looking at, it was interesting to analyze the finetuning across all three statistics.

References

[1] “Resnet18 — Torchvision Main Documentation,” accessed May 8, 2024, https://pytorch.org/vision/main/models/generated/torchvision.models.resnet18.html.

[2] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L.-C. (2018). MobileNetV2: Inverted Residuals and Linear Bottlenecks. https://doi.org/10.48550/ARXIV.1801.04381

[3] “Resnet152 — Torchvision Main Documentation,” accessed May 8, 2024, https://pytorch.org/vision/main/models/generated/torchvision.models.resnet152.html.

[4] Cooke, Thomas F., et al. "Genetic mapping and biochemical basis of yellow feather pigmentation in budgerigars." Cell 171.2 (2017): 427-439.

[5] iNaturalist. Available from https://www.inaturalist.org. Accessed May 8, 2024

[6] FastFUP library. Visual Layer, accessed May 8, 2024, https://visual-layer.readme.io/.

[7] Bradski, G. (2000). The OpenCV Library. Dr. Dobb's Journal of Software Tools.

[8] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2

[9] Collette, A. (2013). Python and HDF5. O'Reilly.

[10] Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. Computing in Science & Engineering, 9(3), 90–95.

[11] Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, et al. “An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale” (2020). Accessed May 8, 2024. https://arxiv.org/abs/2010.11929.