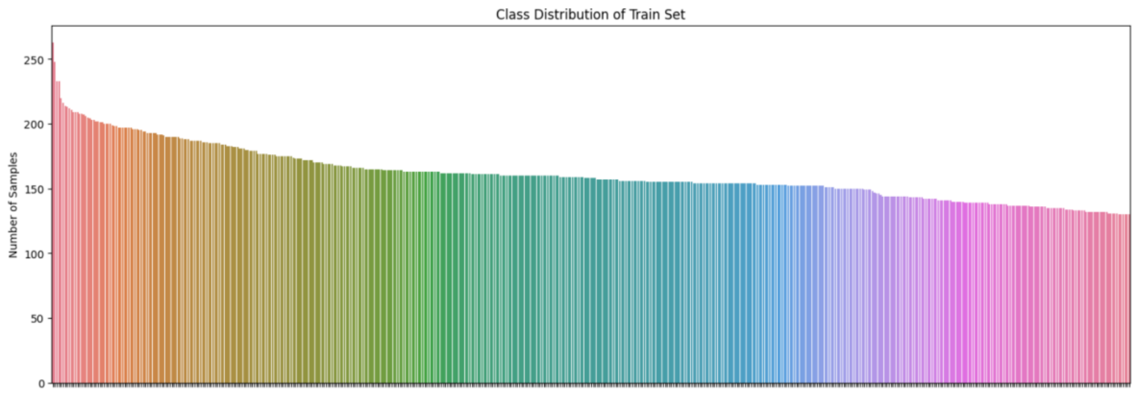
Bird Classification using CNN.

Our data consists of 89,885 Images of 525 different types of birds, most of the images have the following dimensions [3,224,224].

The dataset is divided into 84,635 samples for training, and 2625 samples each for both validation and testing, 5 images for each bird.

For each different bird we have an average of 161 images with a max of 263 and a min of 130 per class, the data is of very high quality, all of the images only contain 1 bird and in most of them the bird takes up around 50% of the pixels, also, the images have not been augmented and we kept it that way for multiple reasons the main reason being that our dataset is already very big and we didn't have enough computational power to process all of the original images as well as the augmented.

The data is relatively balanced, with an average of 161 per class, a max of 263 and a low of 130, looking at the graph below we can see that it is quite balanced:

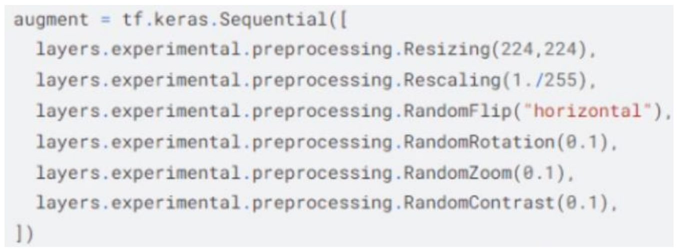


Here are some sample images:

As we can see, there are some birds that are relatively similar to one another (e.g., Fan Tailed Widow and the Fairy Bluebird) but there are also birds that are very distinct from the rest (e.g., Wild turkey and the Barn Owl).

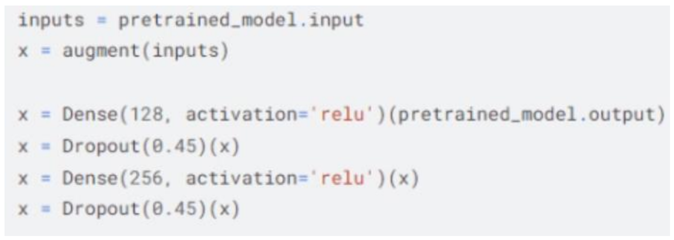
Benchmark:

We found a notebook that used the same dataset, they used weights from a pre-trained model (EfficientNetB0) and heavily augmented the data by flipping, rotating, zooming and changing the contrast



```
augment = tf.keras.Sequential([
  layers.experimental.preprocessing.Resizing(224,224),
  layers.experimental.preprocessing.Rescaling(1./255),
  layers.experimental.preprocessing.RandomFlip("horizontal"),
  layers.experimental.preprocessing.RandomRotation(0.1),
  layers.experimental.preprocessing.RandomZoom(0.1),
  layers.experimental.preprocessing.RandomContrast(0.1),
])
```

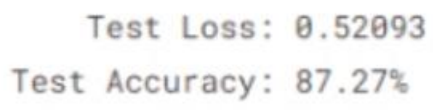
And this is their network structure:



```
inputs = pretrained_model.input
x = augment(inputs)

x = Dense(128, activation='relu')(pretrained_model.output)
x = Dropout(0.45)(x)
x = Dense(256, activation='relu')(x)
x = Dropout(0.45)(x)
```

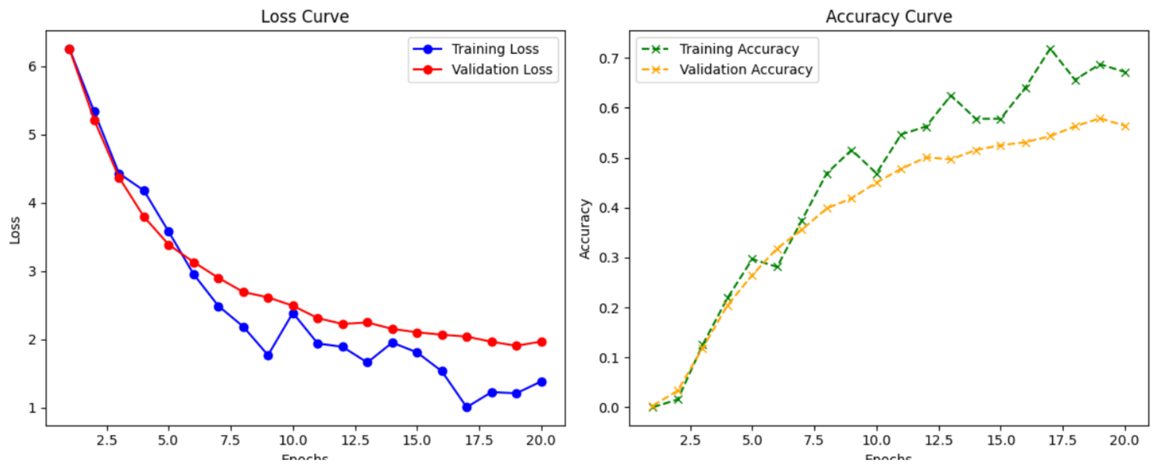
And these are their test results:



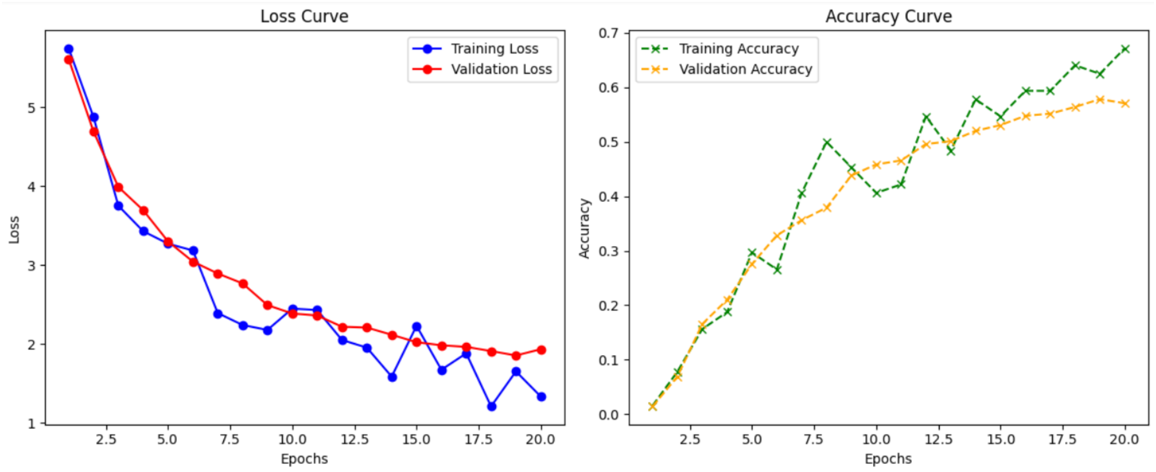
```
    Test Loss: 0.52093
Test Accuracy: 87.27%
```

After implementing our Convolutional Neural Network and training it on the train dataset with k-fold validation (k=5) these are our results:

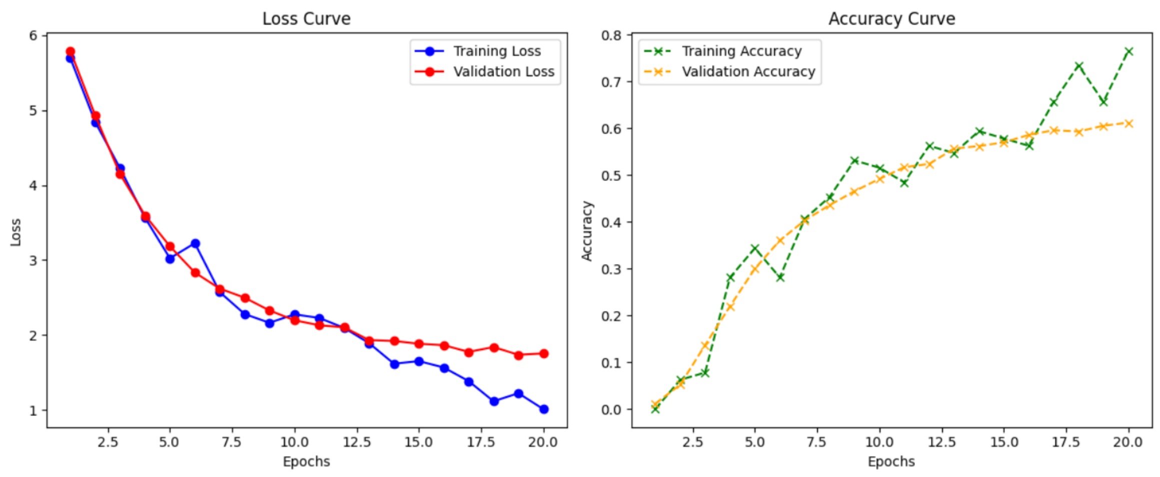
Fold 1:



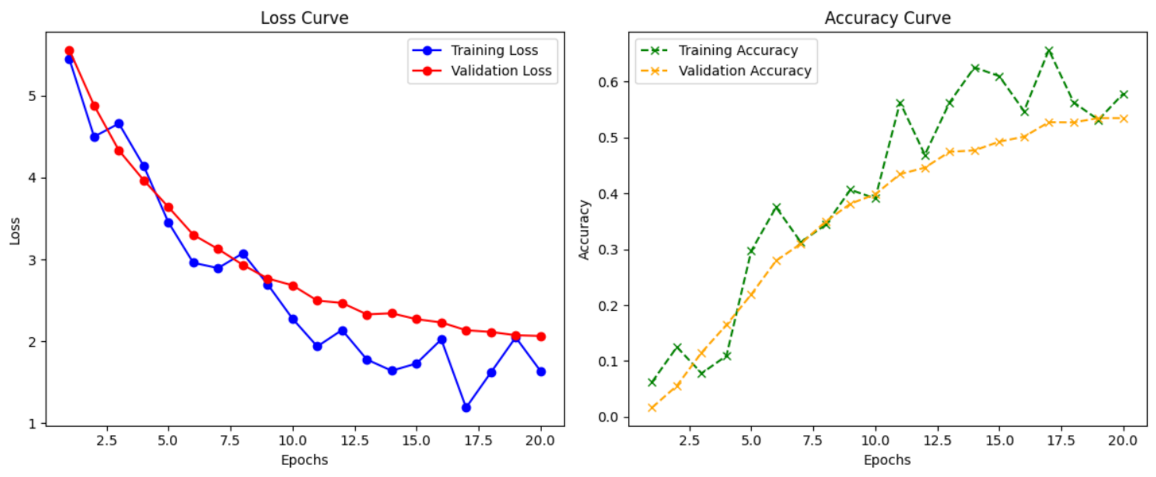
Fold 2:



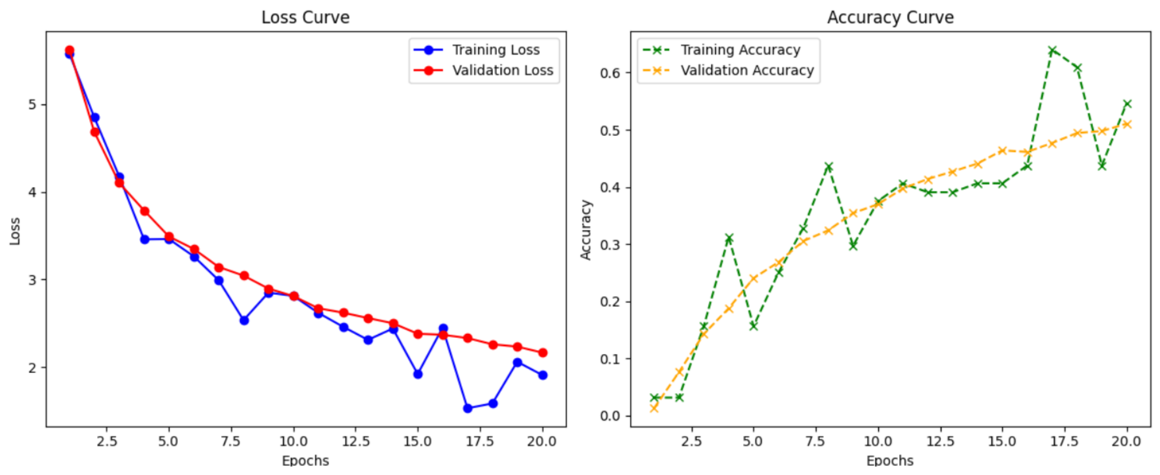
Fold 3:



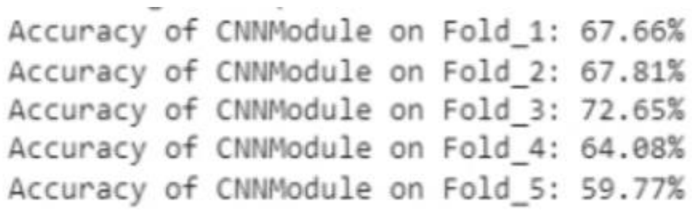
Fold 4:



Fold 5:



We ran all the folds on 20 epochs and got the following accuracies on the test dataset:



```
Accuracy of CNNModule on Fold_1: 67.66%
Accuracy of CNNModule on Fold_2: 67.81%
Accuracy of CNNModule on Fold_3: 72.65%
Accuracy of CNNModule on Fold_4: 64.08%
Accuracy of CNNModule on Fold_5: 59.77%
```

Average accuracy: 66.4%

Here are some good and bad classification examples:

Train images	Test images	Classification
		Correct
		Correct
		Incorrect

Our 3 suggestions:

1. Using batch normalization, since it helps in reducing internal covariate shift, thus making optimization more stable and accelerating training.
2. Learning rate scheduler, because an adaptive learning rate strategy can help the model converge faster and avoid overshooting or getting stuck in local minima.
3. Dropout, since it helps the model avoid overfitting by dropping out random neurons during training.

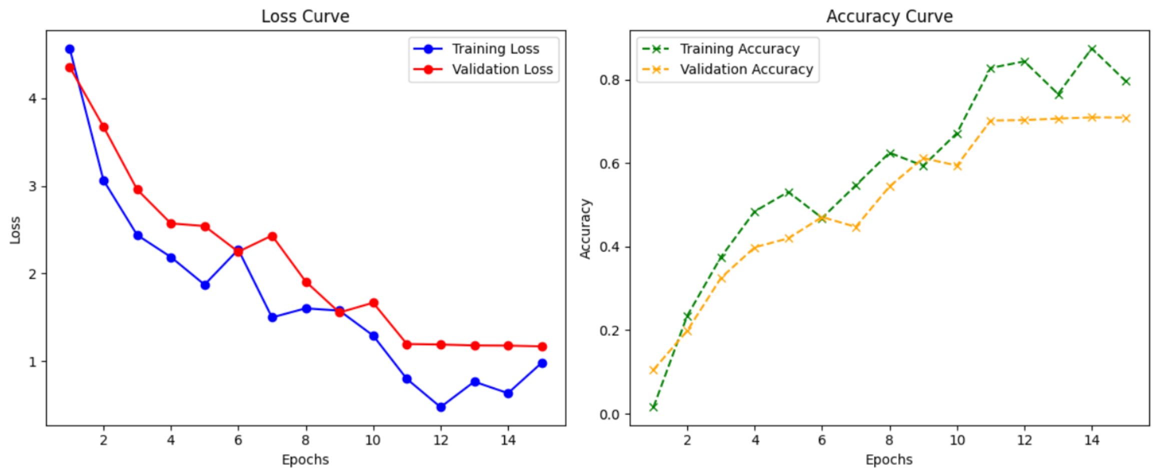
In the end, we chose to implement the batch norm and the learning rate scheduler and not dropout because by looking at the loss curve graphs from before, we saw that our model is not overfitting.

After the implementation, we suggest the following three:

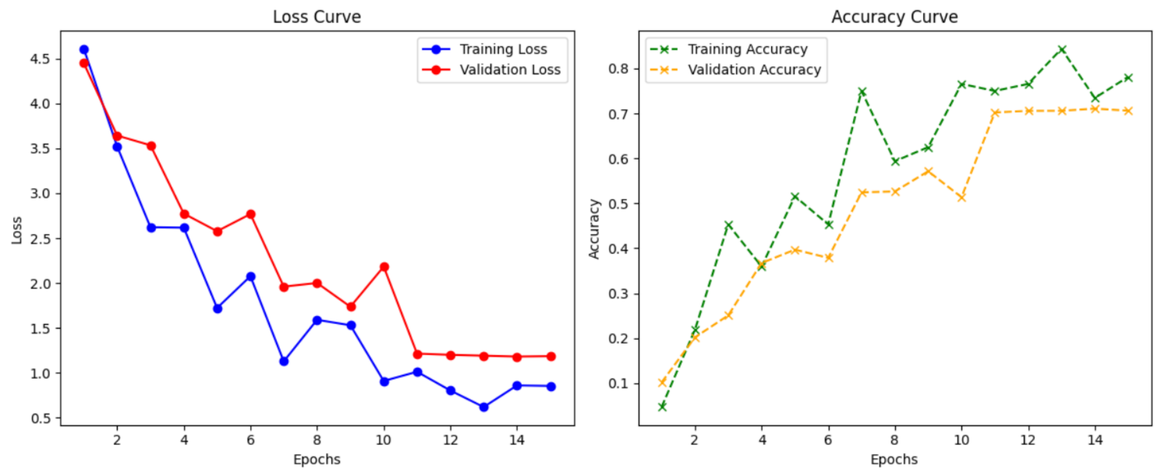
1) Deepen the network, by deepening the network our model can pick up even more details.
2) Initialize our weights with weights from pre-trained networks.
3) Dropout, after deepening the network, our model might overfit, so adding dropout now will be much more beneficial.

And here are the model performance after the improvements:

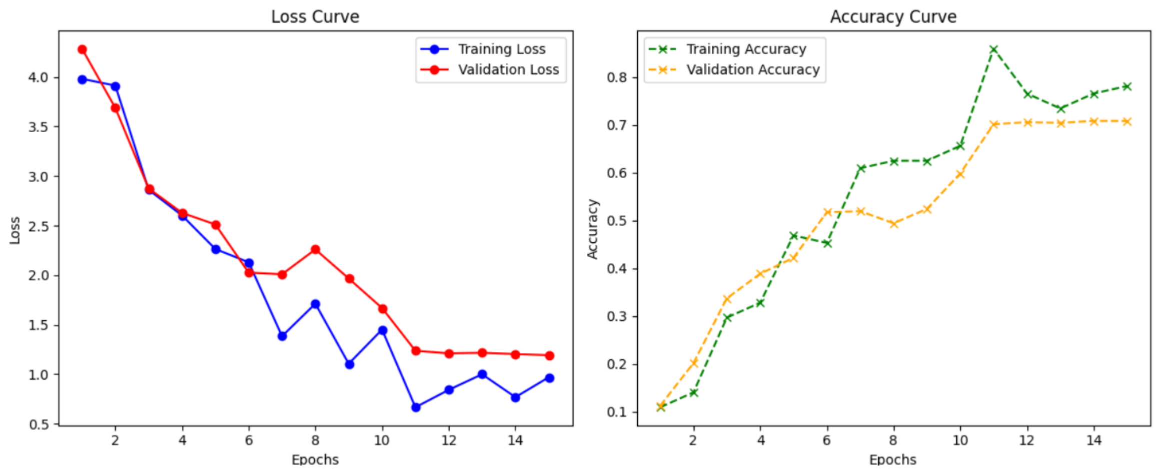
Fold 1:



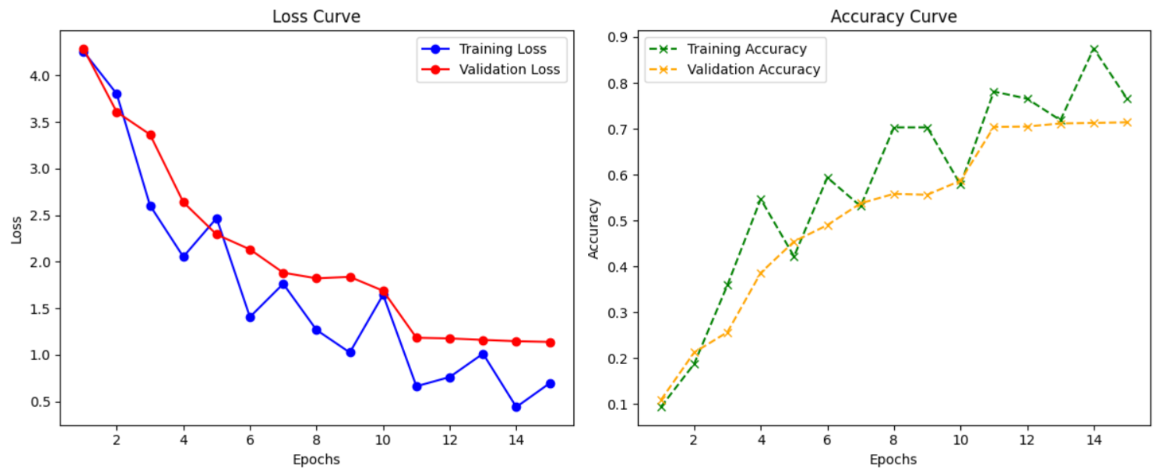
Fold 2:



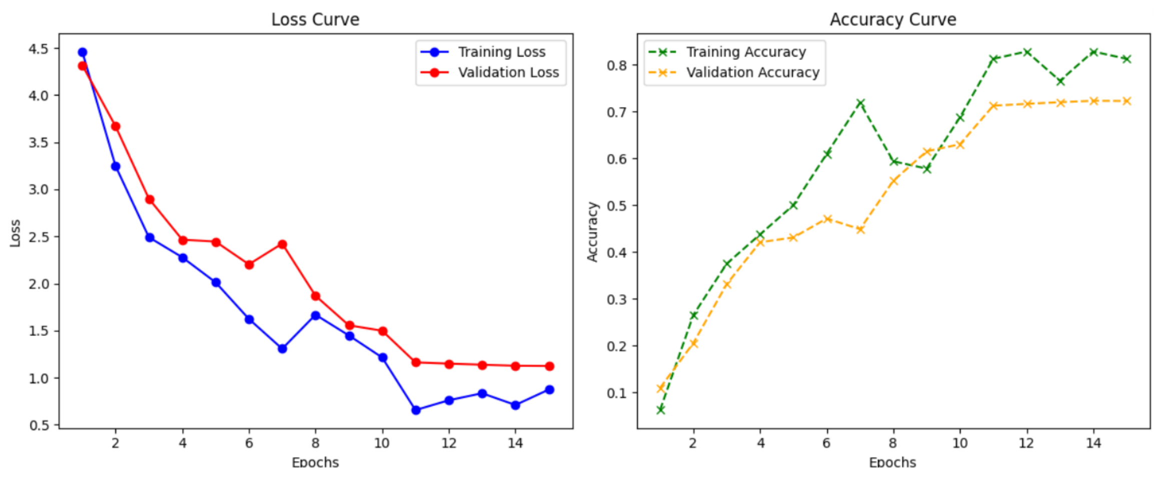
Fold 3:



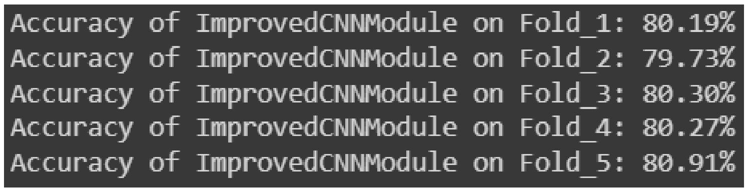
Fold 4:



Fold 5:



And here are the accuracies:

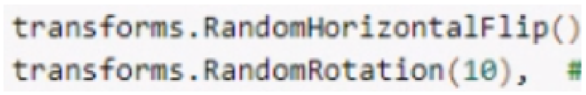


```
Accuracy of ImprovedCNNModule on Fold_1: 80.19%
Accuracy of ImprovedCNNModule on Fold_2: 79.73%
Accuracy of ImprovedCNNModule on Fold_3: 80.30%
Accuracy of ImprovedCNNModule on Fold_4: 80.27%
Accuracy of ImprovedCNNModule on Fold_5: 80.91%
```

Average accuracy: 80.28%

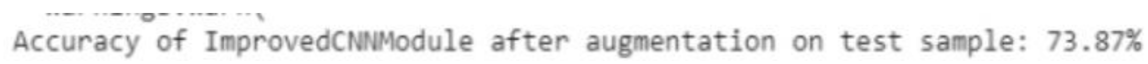
Note: here we only ran 15 epochs per model because we couldn't run them on 20 epochs like previously, however, we trained a model on 25 epochs and got a test accuracy of 90%.

here are the augmentations that we ran on the test data.



```
transforms.RandomHorizontalFlip()
transforms.RandomRotation(10),   #
```

And this is the resulting accuracy:



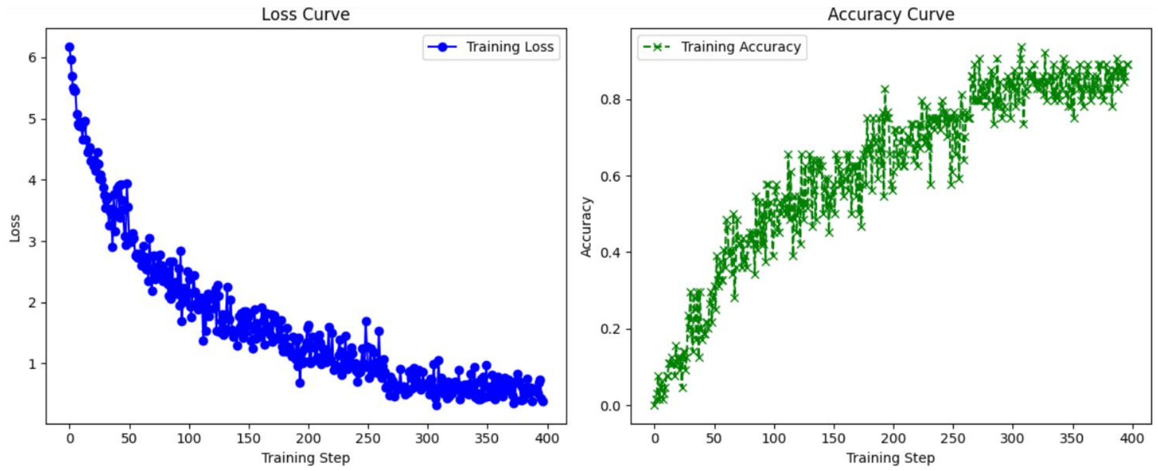
```
Accuracy of ImprovedCNNModule after augmentation on test sample: 73.87%
```

As you can see, the accuracy dropped by 6% because our training data was not augmented, so seeing a rotated bird or a flipped bird (not that kind of bird) was completely new to our model.

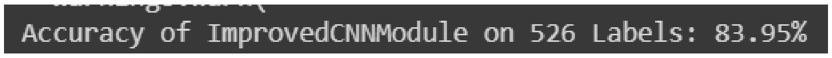
For this section, we added 200 images of a new bird called "Common Kingfisher" to the train data from a dataset called "Indian birds species image classification".

Here is the bird:

And here is the learning curve as well as the accuracy (only on train and not validation)

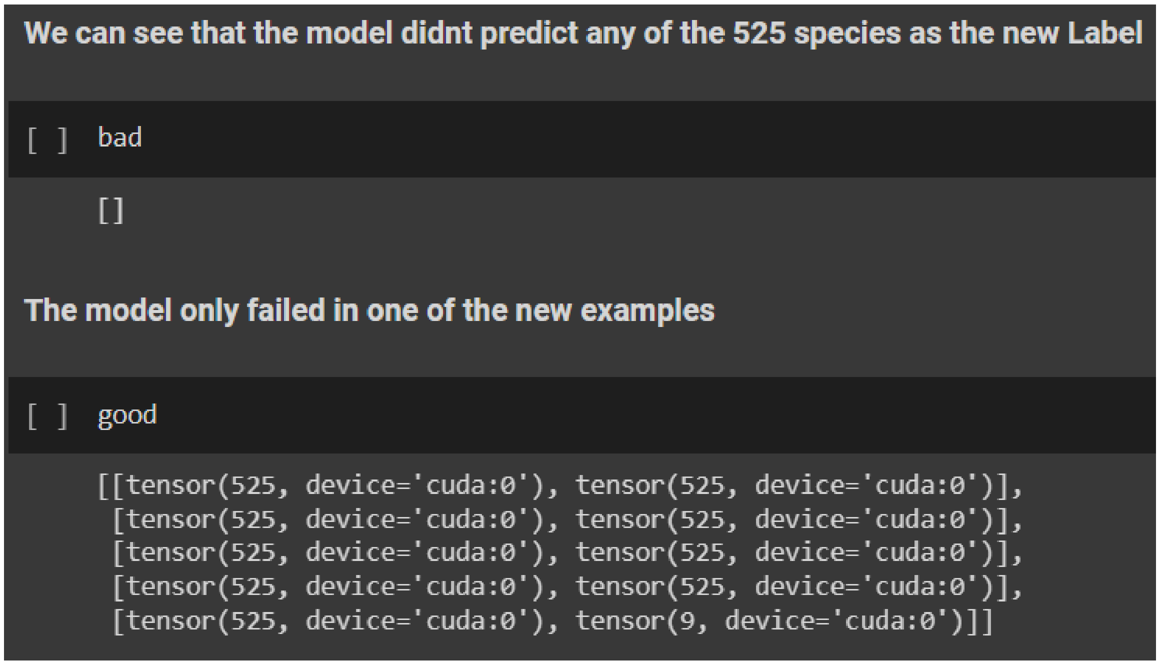


And this is the result on the test dataset with the added 5 images of the new kind



```
Accuracy of ImprovedCNNModule on 526 Labels: 83.95%
```

We can see that the model accuracy increased which is kind of surprising, but by digging deeper we found out the reason, and its because the model didn't misclassify another bird as the new one, and it only failed to classify 1 of the new birds.


We can see that the model didnt predict any of the 525 species as the new Label

```
[ ]  bad
```

```
[]
```

The model only failed in one of the new examples

```
[ ]  good
```

```
[[tensor(525, device='cuda:0'), tensor(525, device='cuda:0')],
 [tensor(525, device='cuda:0'), tensor(525, device='cuda:0')],
 [tensor(525, device='cuda:0'), tensor(525, device='cuda:0')],
 [tensor(525, device='cuda:0'), tensor(525, device='cuda:0')],
 [tensor(525, device='cuda:0'), tensor(9, device='cuda:0')]]
```

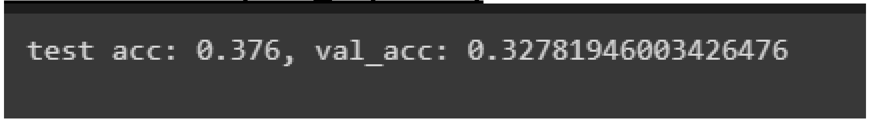
We ran each of the following models with epochs=10 & batch_size=64

Model Name	#parameters	Val loss	Val acc	Test loss	Test acc	# unique correct samples	# unique errors
1. resnet18 – v1	11689512	0.7155	0.9006	0.282	09209	4	141
2. AlexNet	59154765	2.3491	0.7573	1.012	0.7832	0	441
3.googleNet	6624904	0.8805	0.8907	0.3533	0.9078	5	156
4.resNet34	21797672	0.6467	0.907	0.2344	0.9362	6	108

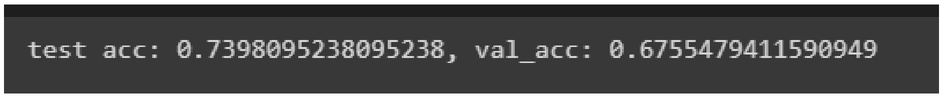
We chose two ML models: RandomForest and KNN. And chose googleNet neural network. So the googleNet sends 1024 features (one layer before the output) to the ML models as input.

Results:

Random Forest(max_depth=10)


```
test acc: 0.376, val_acc: 0.32781946003426476
```

KNN (n_neighbors=30):


```
test acc: 0.7398095238095238, val_acc: 0.6755479411590949
```

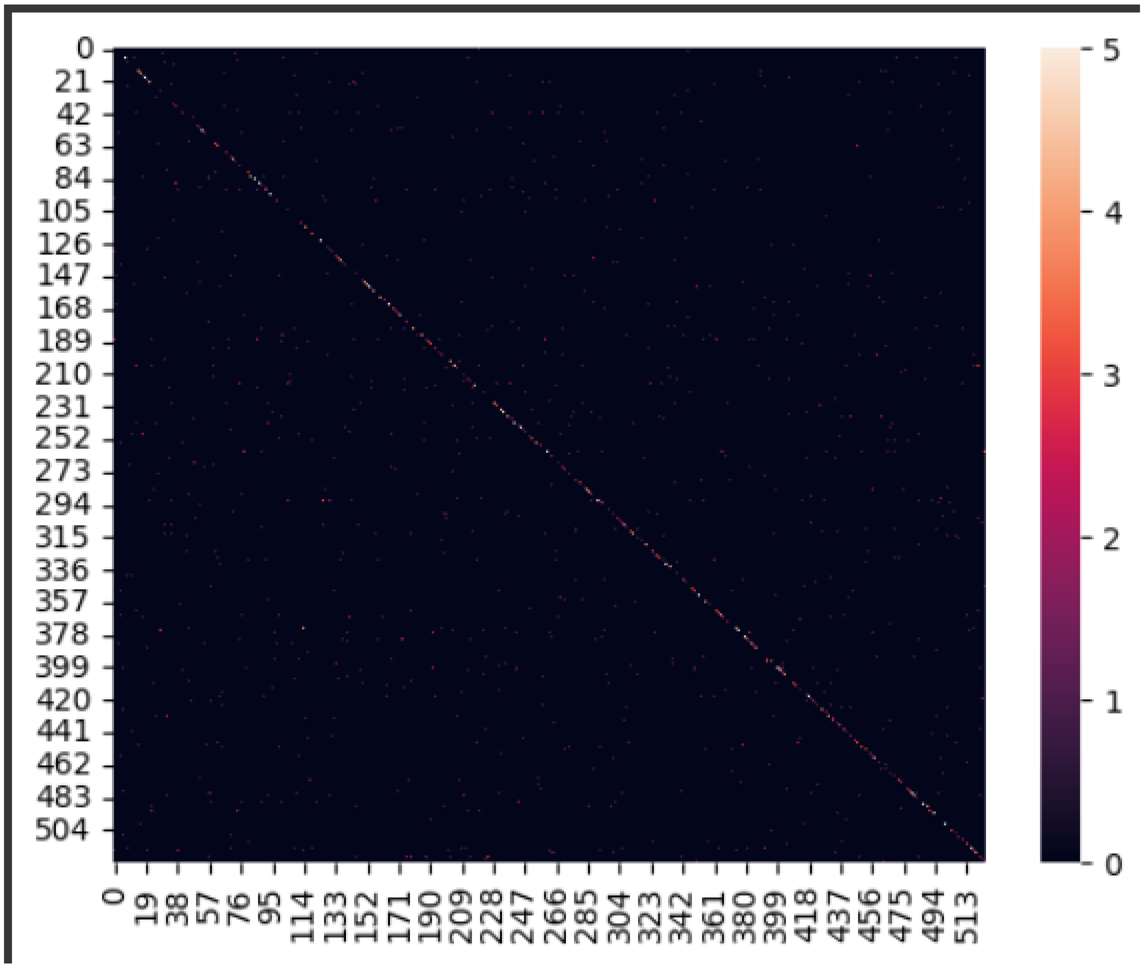
We can see huge decrease in the performance for test and validation, but we knew that KNN would perform better because it will look for images with “similar” features (features extracted from googleNet)

The only significant changes are changing the last layer, and manual hyperparameter tuning for the classic ML models. We tried to show confusion matrix of each model but they all looked the same because of the large number of classes.

Model Name	#parameters	Test loss	Test Acc	F1 score	Train Time
1. resnet18	11689512	0.282	09209	0.9222	59 mins
2. AlexNet	59154765	1.012	0.7832	0.7648	49 mins
3.googleNet	6624904	0.3533	0.9078	0.91	62 mins
4.resNet34	21797672	0.2344	0.9362	0.9348	64 mins
5.randomForest	0		0.376	0.2691	24 mins
6. KNN	0		0.7398	0.7	7 mins

All of the confusion matrix looked like this for 2 reasons:

1. Very large number of classes to show in a heatmap
2. The test set size (2k samples) is very small compared to 525X525 matrix


0
21
42
63
84
105
126
147
168
189
210
231
252
273
294
315
336
357
378
399
420
441
462
483
504
0
19
38
57
76
95
114
133
152
171
190
209
228
247
266
285
304
323
342
361
380
399
418
437
456
475
494
513
5
4
3
2
1
0