DS 675-851 Mini Project -- Animals-10 Dataset

Kevin Kaplan & Travis Virgil

The following portion was written by Kevin Kaplan:

Full notebook with code: https://github.com/KEVKAP13/ds_675/blob/main/final.ipynb

Introduction

Dataset

The dataset selected for this project was the Animals-10 dataset on Kaggle. It contains 26179 images, each belonging to one of ten classes. The classes are as follows: dog, cat, horse, butterfly, squirrel, chicken, sheep, cow, elephant, and spider. The files are downloaded in a folder which contains a unique folder for each class's images. There is a slight class imbalance with this dataset, but it is robust enough that it should not have a major impact on the results.

Link to dataset: https://www.kaggle.com/datasets/alessiocorrado99/animals10
Link to video: https://youtu.be/JlLcns59nOE

Learning Objective and Summary

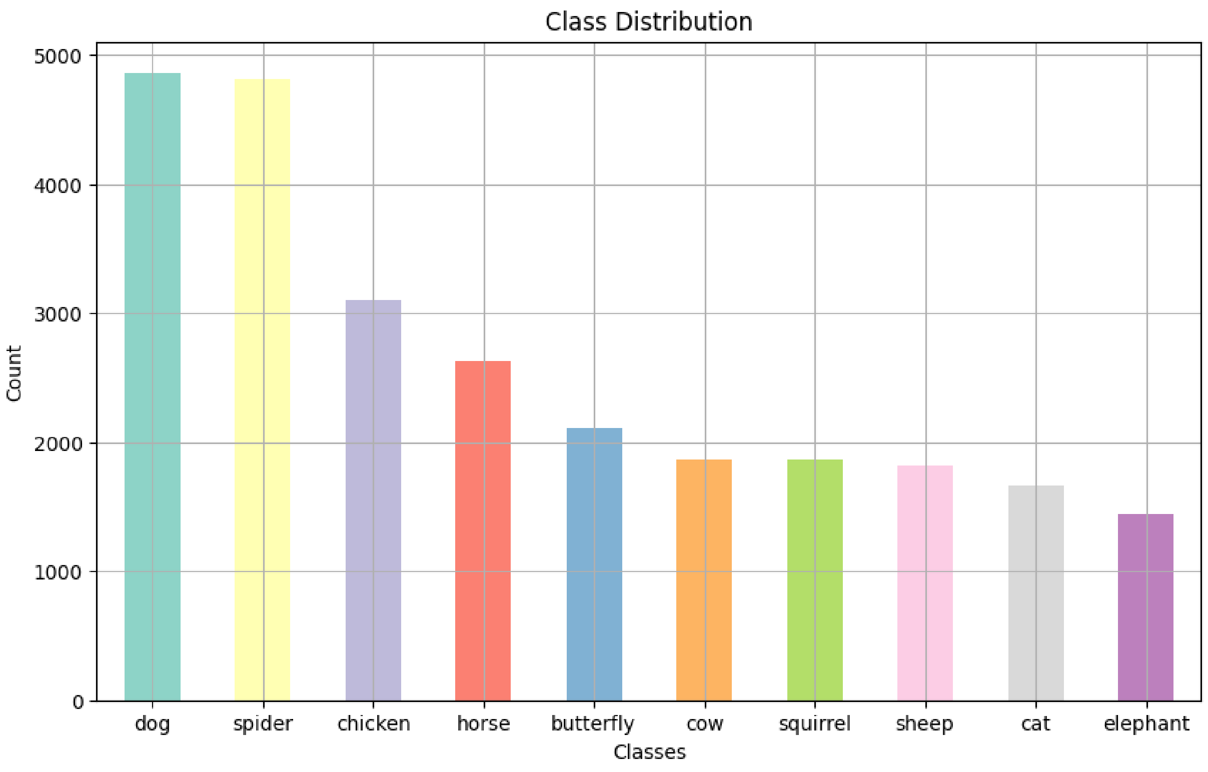
Our main objective in this mini project was to classify images of the Animals-10 dataset. To execute that task, we created Convolutional Neural Networks. Several homemade CNNs were trained and tested. For each model the results were analyzed, and then altered in an attempt to get better results. From there several CNN interpretative methods were used to gain deeper insights into the model, and ultimately refine the model with those insights. Additionally, pretrained CNNs such as VGG-16 and others will be tested and analyzed with the same methods, and then ensembled to test results.

Data

Prepare Data

To prepare the data for use, each file's image path and label is first loaded into a dataframe. The data is visualized by entries per class. It is then split into train, test, and validation sets. An ImageDataGenerator is used for data augmentation and preprocessing for training. Some examples of the augmentations used are random height and width shifts, flips along both axes, etc. Then the datagenerators for each set are prepared.

Visualize Data



Remarks
Regarding the class imbalance-- there are many methods to try to combat this, such as class weighting, oversampling, and several others. Some methods will be tried, but this is a potential area for improvement of the model in the future.

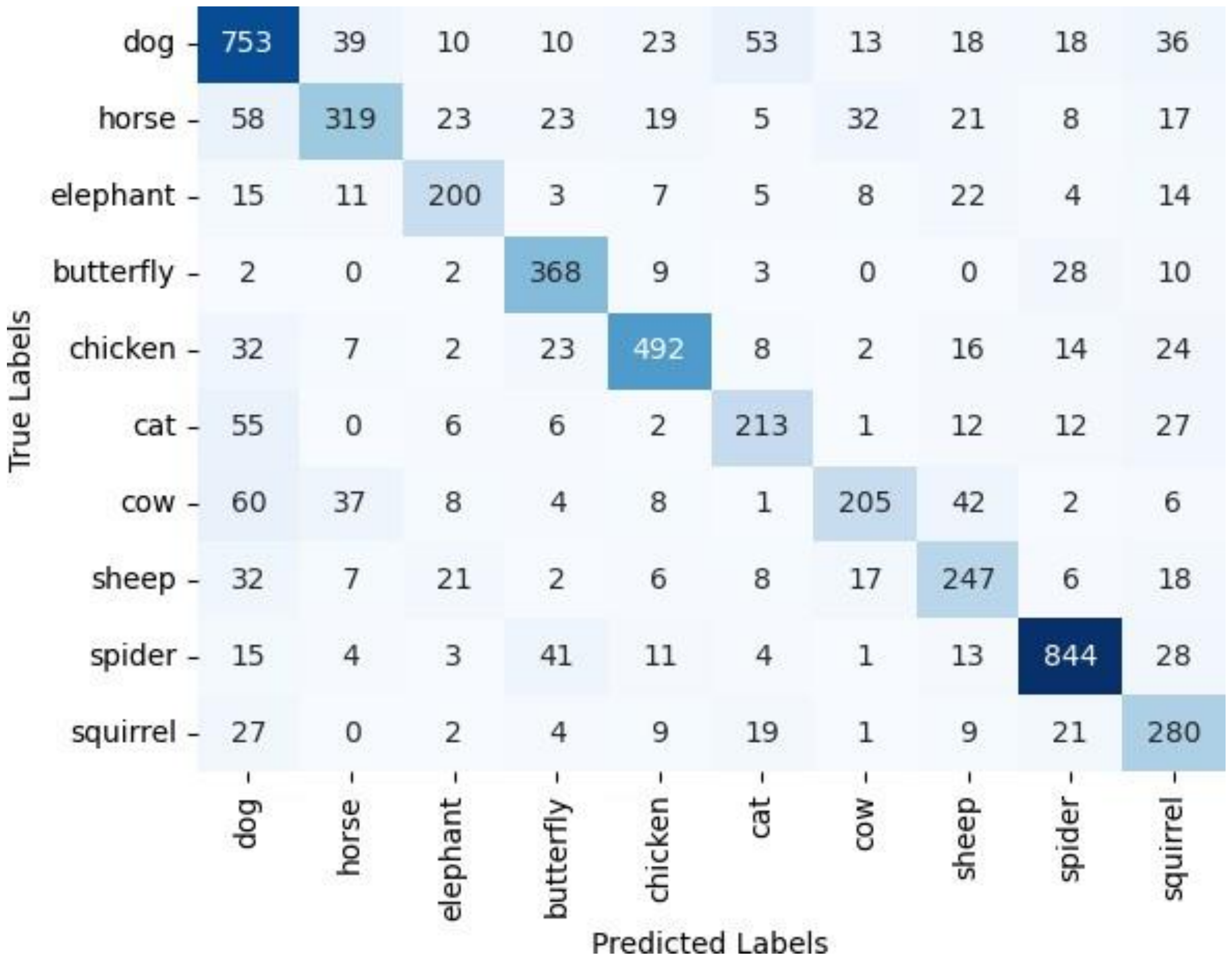
Homemade CNN

Model Construction

For this project fourteen models were initially trained, making adjustments and tuning hyperparameters in an attempt to get a better model. Shown below are the results for the initial model. The tuning and modification of the model, and those results will be discussed in detail at the end of the section.

Results for Initial Model

Accuracy: 0.7488540870893812
Precision: 0.7526712058084805
F1 Score: 0.7478009328361234
Recall: 0.7488540870893812



True Labels \ Predicted Labels	dog	horse	elephant	butterfly	chicken	cat	cow	sheep	spider	squirrel
dog	753	39	10	10	23	53	13	18	18	36
horse	58	319	23	23	19	5	32	21	8	17
elephant	15	11	200	3	7	5	8	22	4	14
butterfly	2	0	2	368	9	3	0	0	28	10
chicken	32	7	2	23	492	8	2	16	14	24
cat	55	0	6	6	2	213	1	12	12	27
cow	60	37	8	4	8	1	205	42	2	6
sheep	32	7	21	2	6	8	17	247	6	18
spider	15	4	3	41	11	4	1	13	844	28
squirrel	27	0	2	4	9	19	1	9	21	280

As a starting point, we have a decent model. There is certainly room for improvement though, and that is what we are going to do. We can begin to see from the confusion matrix some areas the model is struggling with. It is overpredicting and overclassifying the largest class, dog. There is also a lot of confusion between most of the classes of the larger mammals, as well as confusion between the insect classes. Additionally, there is a decent amount of confusion between chicken and butterfly as well. It is possible this is from the wings. Overall, it seems like we may want to try methods for class imbalance as well as attempting to capture different features, and hopefully nudge the model into a more productive direction.

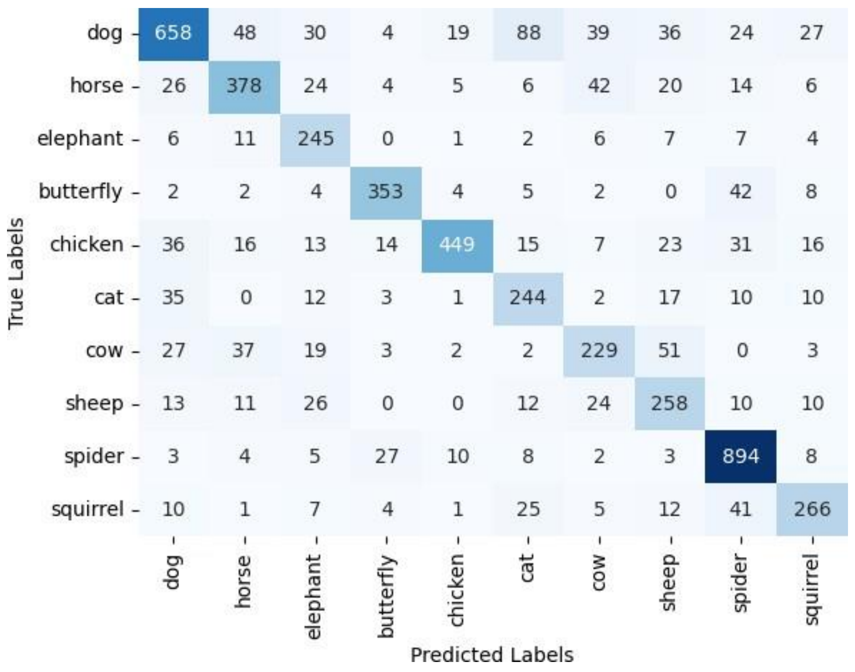
All Models

Results

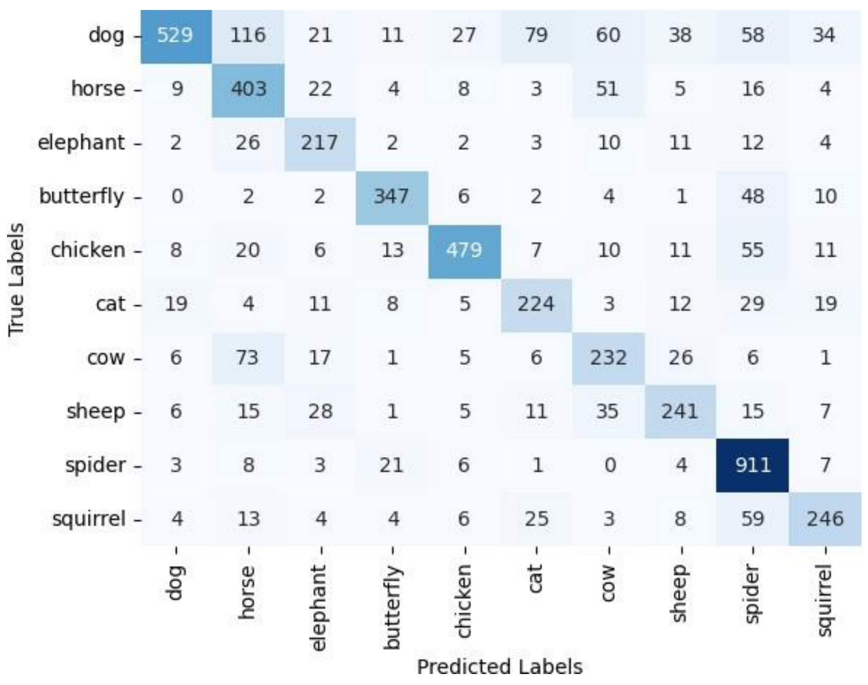
```
        Model Accuracy          Precision F1 Score  Recall
0       Model 1 0.741406        0.744405 0.740308 0.741406
1       Model 2 0.758594        0.767435 0.758835 0.758594
2       Model 3 0.733766        0.753361 0.731221 0.733766
3       Model 4 0.713331        0.734819 0.714899 0.713331
4       Model 5 0.728992        0.739722 0.730808 0.728992
5       Model 6 0.735676        0.751400 0.734740 0.735676
6       Model 7 0.803858        0.807871 0.803629 0.803858
7       Model 8 0.735294        0.750443 0.732624 0.735294
8       Model 9 0.732047        0.741438 0.730629 0.732047
9       Model 10 0.799465       0.801255 0.797968 0.799465
10      Model 11 0.809206       0.815901 0.809119 0.809206
11      Model 12 0.806914       0.816320 0.808881 0.806914
12      Model 13 0.750764       0.761182 0.749832 0.750764
13      Model 14 0.790107       0.793305 0.788868 0.790107
```

In the above output, each iteration of the model is shown, along with its performance metrics. Other starting points were tested, but are not included in the table shown due to poor performance. Many changes were made in an attempt to improve the original model. The use of class weights were implemented at a few different iterations, but interestingly, yielded minimal or no improvement. When examining the confusion matrix for a model with and without class weights (shown below), we can see that it is no longer overpredicting the largest class (dog), but it just leads to more misclassifications of that class. Additionally, it is now overpredicting the second largest class more than without weights (spider). This could be improved upon potentially by trying different weights possibly, however, the model without weights was proceeded with.

Without Class Weights:



With Class Weights:



	dog	horse	elephant	butterfly	chicken	cat	cow	sheep	spider	squirrel
dog	529	116	21	11	27	79	60	38	58	34
horse	9	403	22	4	8	3	51	5	16	4
elephant	2	26	217	2	2	3	10	11	12	4
butterfly	0	2	2	347	6	2	4	1	48	10
chicken	8	20	6	13	479	7	10	11	55	11
cat	19	4	11	8	5	224	3	12	29	19
cow	6	73	17	1	5	6	232	26	6	1
sheep	6	15	28	1	5	11	35	241	15	7
spider	3	8	3	21	6	1	0	4	911	7
squirrel	4	13	4	4	6	25	3	8	59	246

In addition to class weights, different activations were tested, including leaky relu. It did not seem to help, so the original and more common relu activation was kept for the remaining iterations. The batch size was both halved and doubled, and neither led to any improvement. The batch size of 64 for used for the rest of the models.

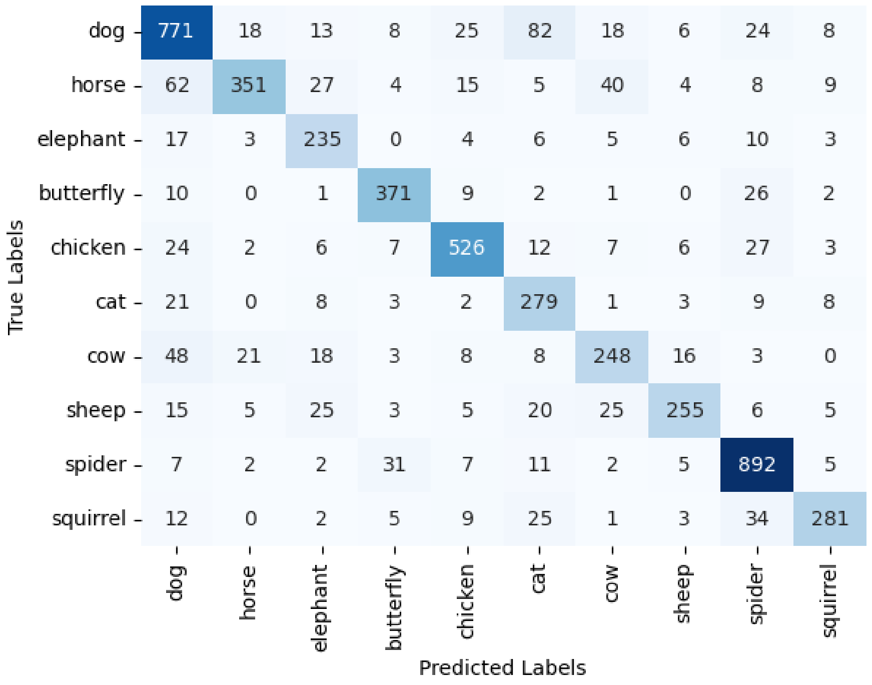
To help improve the training of the model, batch normalization was used. Batch normalization can also help a bit with generalization and overfitting by providing a regularization affect. However, through various iterations, it was evident that there was still some overfitting occurring when analyzing the loss curves for the train and validation data. To help mitigate this, various regularization methods were employed. L1 and L2 regularization were used, however they did not provide much help, or increased the computational cost too greatly to be a viable option. The one regularization method that did seem to have good results was including dropout layers. The dropout layers regularize by randomly deactivating a portion of the neurons at each step.
Additional impactful changes were increasing the number of filters and changing strides of the convolutional layer. By adding more filters and using different strides, the goal was to try to capture more and different features of the images. Adding and removing an additional dense layers, as well as the number of neurons in those layers were experimented with as well. However, finding a balance of these hyperparameters and architectures began to become a guessing game. Aside from finding a balance between having too many filters (causing overfitting), different layers, what strides to use, how many dropouts, where to put them, and what fraction to use, computation cost is another major consideration. On top of that, to this point, while alot has been done it is only scratching the surface of possible options to try. From here it is becoming even more clear that it would be invaluable to be able to have more information to make better decisions in tuning the model.

We can see from the results of the model training and tuning that while improvements were made, at a certain point it becomes more of a guessing game with the available information. Couple that with the fact that each model took an average of five and a half hours to train, and one misstep in tuning the model can easily cost you the better part of the day. This drives home the importance of informed decision making in tuning the model, and raises the question-- what can we do to improve our decision making?

After previous experience studying interpretable machine learning methods, I became interested in Grad-CAM and similar methods. These methods work to enable us to visualize what the model 'sees' when making a decision, giving us more insight to what the model is doing and how it is making predictions. From here, we can leverage these methods for a deeper understanding of our model, and hopefully make more informed decisions when modifying the model.

Analyzing the Best Model

Model 11 appears to be our best model, and here we will analyze it further:



True Labels \ Predicted Labels	dog	horse	elephant	butterfly	chicken	cat	cow	sheep	spider	squirrel
dog	771	18	13	8	25	82	18	6	24	8
horse	62	351	27	4	15	5	40	4	8	9
elephant	17	3	235	0	4	6	5	6	10	3
butterfly	10	0	1	371	9	2	1	0	26	2
chicken	24	2	6	7	526	12	7	6	27	3
cat	21	0	8	3	2	279	1	3	9	8
cow	48	21	18	3	8	8	248	16	3	0
sheep	15	5	25	3	5	20	25	255	6	5
spider	7	2	2	31	7	11	2	5	892	5
squirrel	12	0	2	5	9	25	1	3	34	281

Aside from the model overpredicting for the largest class, there are many insights we can gain from the confusion matrix. Similarly, to our original model, there is also a lot of confusion between most of the classes of the larger mammals, as well as confusion between the insect classes. Some of the areas where the previous model had a lot of confusion have been improved upon, like butterfly and chicken and the mammals in general. There are still some areas where the model is specifically struggling, like cat and dog or horse and cow. Overall, it still seems like we may want to try methods for class imbalance as well as attempting to capture different features, and hopefully nudge the model into a more productive direction. Next, we will look into some of these misclassifications in more depth.

Interpretability Methods

Interpretability methods for CNNs give us insights on how they make decisions. They provide visualizations enabling us to 'see' what the CNN sees when making its predictions-- shedding some light into the 'black box.' There are many different methods for interpreting CNNs, including perturbation-based methods like occlusion sensitivity or LIME ImageExplainer. There are activation-based methods like activation maximization and intermediate activations as well. In this project, the focus was on gradient based methods. The gradient based methods used in this project include Vanilla and SmoothGrad Saliency Maps, Grad-CAM, Grad-CAM++, and Path Integrated Gradients.

While all these methods will give us a deeper understanding of how the model is making decisions and further insights into how the images are being classified, the objective is to use information gained from these methods to help guide decisions on improving the model. By taking intuitions and insights gained from the original analysis of the model’s performance, the interpretation methods can help us look further into them. Specific misclassifications and classes can be looked at, hopefully enabling us to see key features the model is focusing on and ultimately identifying weaknesses in the model.

Using this information, modifications to the model and data can be made accordingly. Now instead of being at a point of guessing, we can hopefully make a more informed decision on what to do. In this notebook, the focus is on confusion between dog and cat as it is the most significant confusion we saw in the previous section. The demonstration is for showing these specific misclassified images; however, the code can be used to analyze any portion of the data desired.

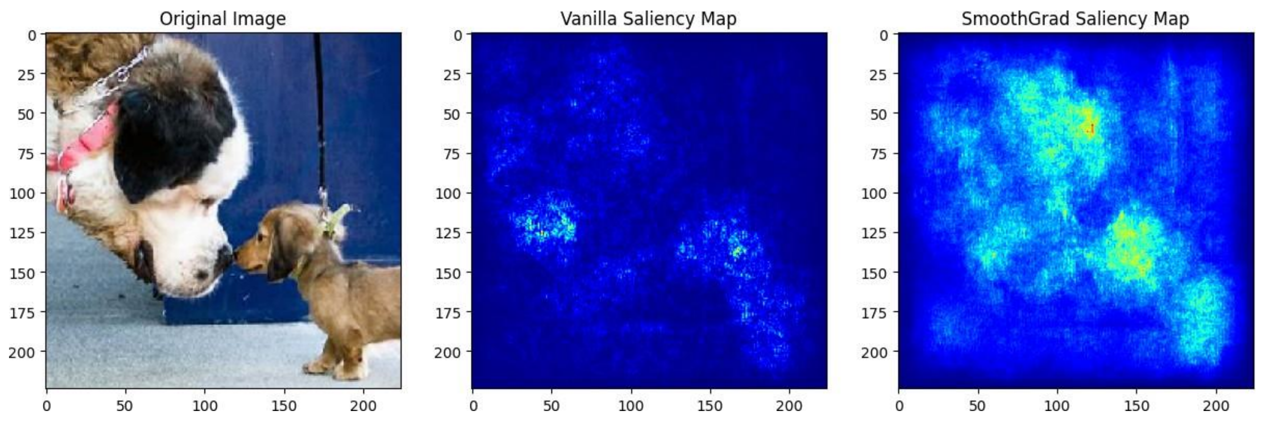
Gradient Based Attribution Methods

Gradient based methods calculate attribution maps for each datapoint through forward and backward passes through the network. To calculate the attribution maps, the gradients in the backward pass are used. We can then use the results to visualize how much each input feature contributes to the model’s prediction. Saliency Maps, both Vanilla & SmoothGrad, Grad-CAM, Grad-CAM++, and Path Integrated Gradients are all used in this section. Each method provides us with a different way of visualizing and understanding the model's predictions. This will provide us with valuable insights into which parts of the data are key to the model's decisions. For each of the models, a single example will be shown, and finally with Grad-CAM++, several misclassifications.

Saliency Maps

Saliency maps highlight critical regions of input, relying on gradients to determine which pixels impact the models' predictions most significantly. The original method, 'Vanilla' Saliency Maps, rely on the absolute value of the gradients. It attempts to find the pixels that can be perturbed the least to result in an output that changes the most. However, it doesn't conduct the perturbations and can't validate it due to the use of the absolute values. SmoothGrad on the other hand, creates small perturbations which introduce noise to the image, calculates the gradients of them, and then averages the results to give a 'smooth' resulting saliency map. Here both methods are shown.

Saliency Maps Results



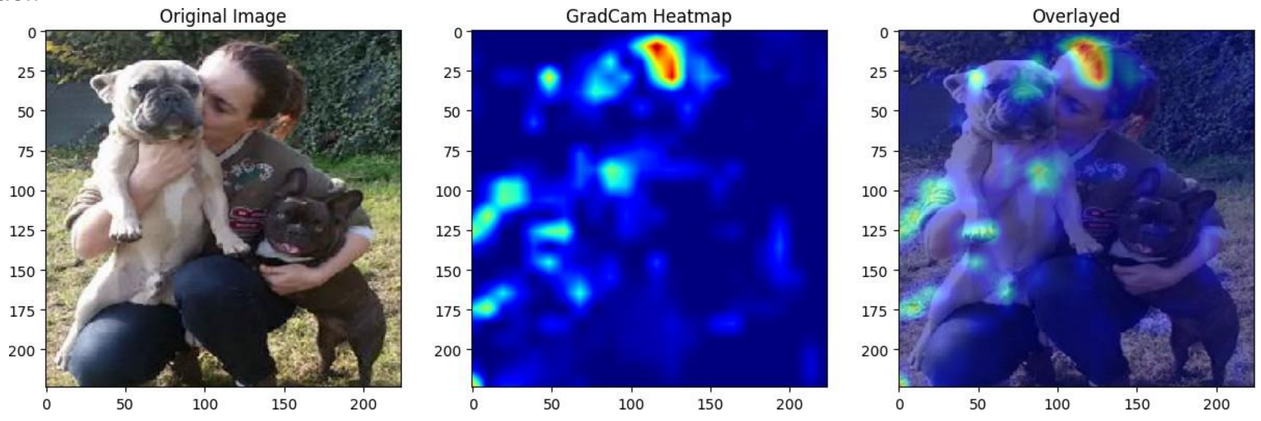
Interpretation

We can see from the maps here that while the model is not focusing on entirely irrelevant features, we still may need to capture more complexities to gain improvement in our model.

Grad CAM

Class Activation Maps (CAM) work by removing everything except for the last dense layers and replaces the MaxPooling layer with a Global Average Pooling layer. The idea is that the spatial details in a CNN are lost in the fully connected layers. When the last convolutional layer is weighted, the filters represent the most salient regions of the image. CAM is difficult to implement, so Grad-CAM uses gradients to get neuron importance weights. Activation maps are then produced and can be visualized for us to interpret. While Grad-CAM is an improvement over CAM, it does have some issues. It can fail in certain circumstances, and Grad-CAM++ was developed to improve upon this.

Grad CAM Result
Interpretation

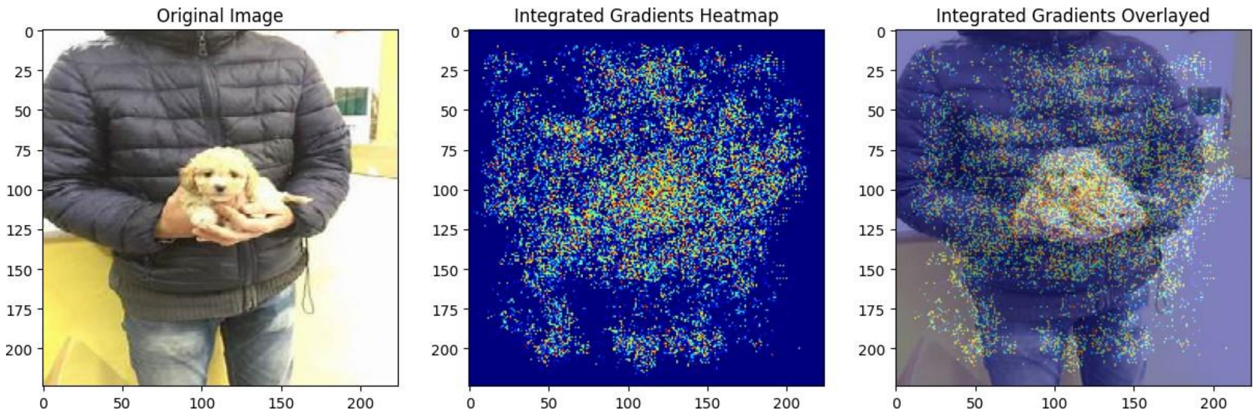


Here it looks like the background and person in the image may be confusing the model. The model is picking up certain features on the dog, however, most heavily focuses on the woman's hairline, and some other irrelevant features. Providing different data augmentation and attempting to capture more complex features can help to mitigate this.

Integrated Gradients

Integrated Gradients, or Path Integrated Gradients, work to calculate feature importance using the integral of the gradients along paths connecting a baseline to the input image. The baseline is an all black image in our case, as the image pixels are all set to 0. It progressively gets to the baseline image through a number of steps, puts all of these images through the CNN, computes the gradient, and then averages it. The integrated gradient is then calculated-- it is the dot product of the image and the average gradients. We can use the results to further confirm hypotheses from analyzing the model results.

Integrated Gradients Results



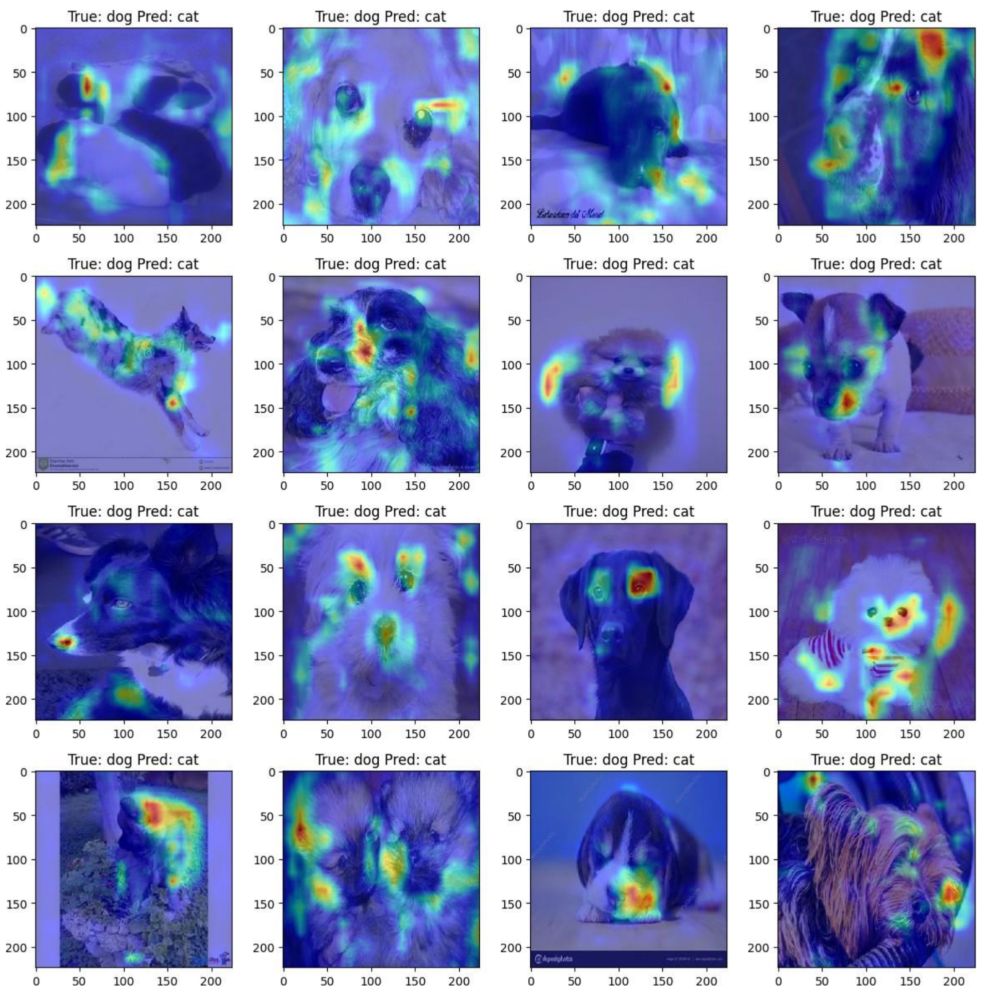
Interpretation

Again, the model is not focusing on relevant features and is confused by the background of the image.

Grad CAM++

Grad-CAM++ is an improved version of Grad-CAM, addressing the issues that Grad- CAM has. Here we can see how this version does in fact provide a better representation of the data.

Grad CAM++ Results



Interpretation

We can see the model is highlighting some irrelevant features in these cases; however it is focusing on the face eyes and fur of the dogs as distinguishing features. We can aim to improve the model to help it identify finer distinctions between these classes through different methods.

Modifying Model with New Insights

Implement Changes

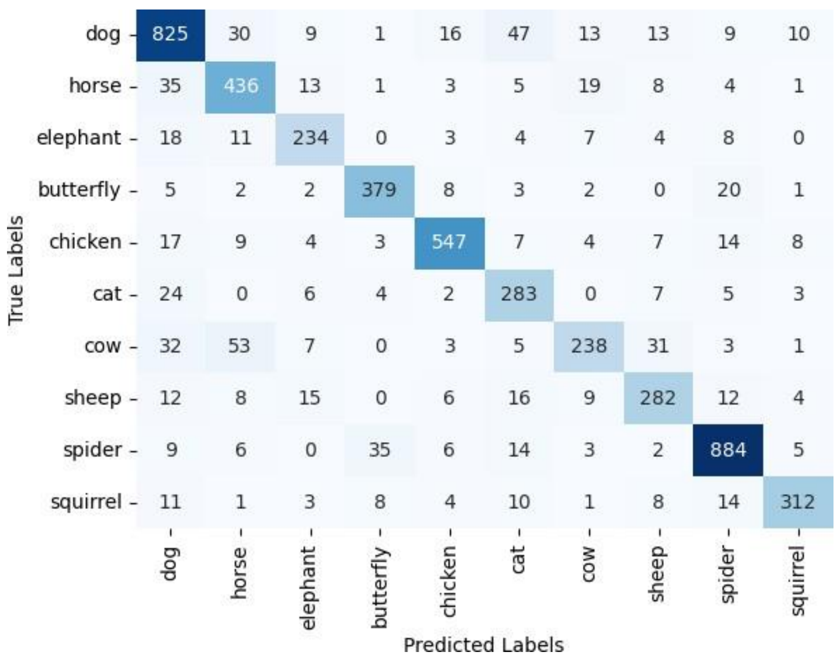
Through the analysis of the results, it is evident there are a few things to focus on. we need to get the model to focus on finer distinguishing features, eliminating or reducing confusion between classes, and try to get the model to stop focusing on irrelevant features and/or the background. This can be addressed in two ways initially. The first is to use more and more aggressive data augmentation techniques. This will provide the model with more diverse data to learn from. Hopefully allowing it to focus on the more important and generalized features and ignoring irrelevant ones that won't remain across the augmentations. While I was initially more cautious with this, trying to mitigate the chances of introducing bad data that would negatively impact learning, based on observations of the model this should help performance and address the issues of the model focusing on irrelevant features in certain cases like we observed in the previous section. Next, based on our observations, it seems the model is largely focusing on important areas of classification, but perhaps not picking up the finer details and nuances that distinguish these features. To address this, we can augment the model to capture more features. We need to be careful to not overfit it or create a model that is computationally unrealistic. First, three models were trained with that in mind. The results are displayed below.

Results

	Model	Accuracy	Precision	F1 Score	Recall
0	Model 15	0.784759	0.809150	0.789521	0.784759
1	Model 16	0.811306	0.815792	0.809882	0.811306
2	Model 17	0.798510	0.805131	0.800051	0.798510
3	Model 18	0.802712	0.802379	0.801421	0.802712
4	Model 19	0.816272	0.819758	0.816800	0.816272
5	Model 20	0.843583	0.846173	0.843230	0.843583

While we land on one model that is outperforming our previous models (model 16), the others did not improve and were showing signs of overfitting (by examining loss curves). Next, the model is altered in an attempt to focus on capturing more complex features, while still maintaining good overall performance. Additional convolutional layers were added, and the final dense layers were increased (model 19). All of the layers include batch normalization before the ReLU activation. Two dropout layers are used as well. To improve upon model 19, one of the dense layers were removed to help the model be more generalized and reduce potential for overfitting. The model summary is shown below.

Analyze



Here we see the confusion matrix for model 20, the top performing model. Aside from the general improvement of the model, we have achieved our goal of reducing the confusion between dog and cat, as well as reducing confusion between other classes. We are starting to see some of the other confusions decrease or go away all together as well.
However, there are still areas that could require attention.

While we were able to gain improvements, and do so more efficiently, this process could be repeated and continued for better model performance. In addition to these methods, we can try different data augmentation techniques and continue to make informed decisions on improving the model. There are many other things we can do from here to improve performance as well. One such example would be to ensemble models together in order to get a more diversified set of predictions. This could lead to better generalization, and a better performing model overall.

Traditional ML Methods

Additionally, data was prepared for and trained using traditional machine learning methods with the goal of comparison to the more modern and widely used CNNs for image classification used in this project. The two methods tested were Support Vector Machines and Random Forest. To prepare the data for training, it was first loaded into a dataframe, then split into train and test sets. The pixel values were then normalized, preparing it for feature extraction to use in the models. To extract features, custom Gabor Filters were used.

Several other filters were attempted as well, but the cost of running it was too great. The dataset was modified to use just a quarter of the data, and the filters were modified to use just a few iterations of the Gabor Filters. The other methods were eliminated. Still, with less data and significantly less features to extract, the compute time was still quite long. First, SVMs were trained. A GridSearchCV and RandomSearchCV were attempted to find optimal hyperparameters, however this process took an infeasible amount of time to run, and so a single SVM was trained. The time it took to prepare the data, train, and test a single SVM on a fraction of the dataset was nearly three times that of the average training time of the homemade CNNs from this project. After testing, the final results of the SVM gave an accuracy of 34% on the test data. The same methods were used to train a Random Forest model. While the train and test times for Random Forest were significantly lower than SVM, they were still considerably long. The final accuracy obtained for the Random Forest Model was close to that of the SVM, with 34.05%.
If I were to continue working on traditional machine learning methods for this dataset there are a few things I would do. The first would be to investigate more efficient methods of feature extraction. I think there is definitely a way to speed up that process and getting that sped up would make a huge difference in doing this. From there I would focus on Random Forest as it was significantly faster and produced essentially the same results. After that, gradient boosted tree methods like XGBoost could be tested as well. However, while there are certainly advantages to using traditional machine learning methods, in this instance, the best approach is using CNNs.

Conclusion

In summary, this project used Convolutional Neural Networks to classify the images of the Animals-10 dataset. Several models were trained and at each iteration of the model, augmented to improve it. The results were then analyzed with interpretable machine learning methods with the aim of using that information to guide decisions on model improvement.

These methods provided valuable insights into the model and helped us to understand it better. By affectively utilizing the information obtained through these methods ultimately improve the model with more informed decisions, leading to an increased performance that is not negligible.

Future Improvements

There are several things that I wanted to try for this project that I was not able to get to in the timeframe of this project. One thing that I think could be addressed further for this dataset is the slight class imbalance. Using different methods for class weights, and one thing I wanted to try but didn't have time for, using oversampling with SMOTE, are all possibilities for model improvement. Continuing to use the interpretation methods for improving the model is of course a possibility as well. Additionally, any of the methods from the vast array of other interpretability methods to try is something I would like to do. As mentioned in the Traditional Machine Learning section, there are other methods to explore there as well. There are many avenues to consider continuing work on this project, and provided the time I would have tried them.

Sources

Kaggle Notebooks:

- https://www.kaggle.com/code/quadeer15sh/grad-cam-what-do-cnns-see
- https://www.kaggle.com/code/abdallahwagih/animals-detection-efficientnetb3-acc- 97-6
- https://www.kaggle.com/code/min4tozaki/animal-classification
- https://www.kaggle.com/code/vencerlanz09/animal-image-classification-using- efficientnetb7
- https://www.kaggle.com/code/abdulbasitniazi/resnet50fromscratch-eda

Textbooks:

- Masís, S. (2021). Interpretable Machine Learning with Python: Learn to Build Interpretable High-performance Models with Hands-on Real-world Examples.

Libraries:

- https://keisen.github.io/tf-keras-vis-docs/index.html

Videos:

- https://www.youtube.com/watch?v=ywyomOyXpxg

Below portion written by Travis Virgil. The full notebook for all information below can be viewed here:

https://gist.github.com/virgilt/c9393cbeece08e8368a5937f8f6edf38

Background Pretrained Models

In this section the focus is on pre-training models and utilizing transfer learning to apply them to the animal 10 data set. The two pre-training models used are Xception and VGG16. One of the main reasons for choosing Xception in VGG16 is that they are not commonly used on other Kaggle projects. We wanted to perform transfer learning utilizing something other than what is most commonly used such as EfficientNet and ResNet. We thought this could lead to some interesting results that have not yet been observed.

Some background on the models that we chose for pre-training:

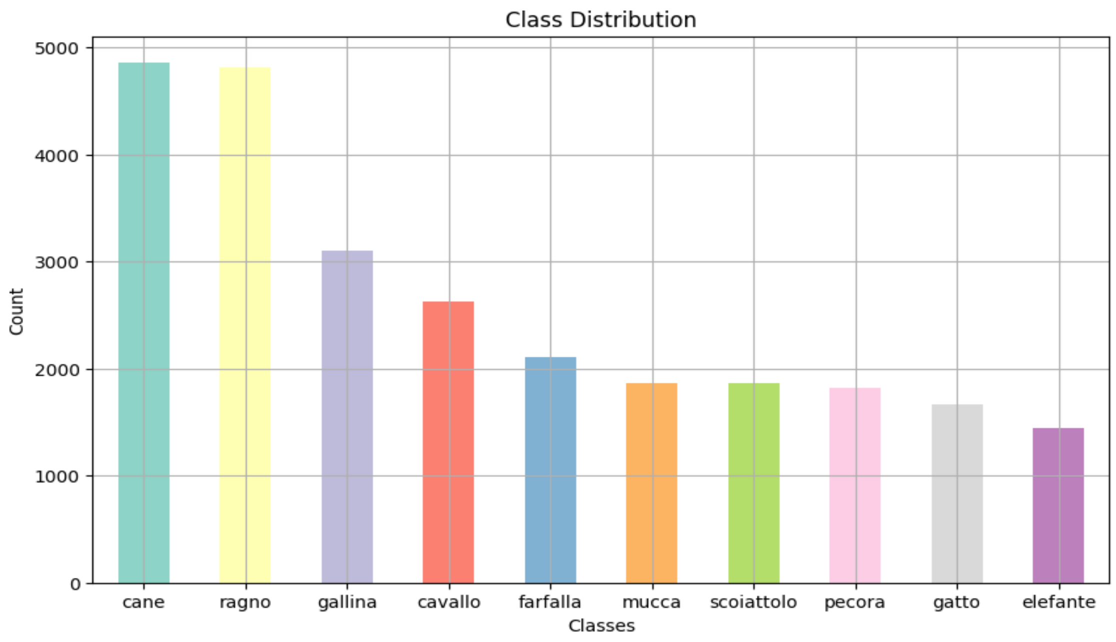
Xception uses depthwise convolutions followed by pointwise convolutions aiming to capture spatial and cross-channel correlations efficiently. This leads to a reduction in computational complexity while preserving representational power. So, the main reason to choose Xception is because of the lower computational cost. Xception is also known for having above average performance for computer vision tasks such as image classification, which is our intended use case.

VGG16 has a more straightforward architecture design consisting of repeated smaller convolutional filters 3x3 followed by max pooling layers. This model offers stable performance for various data sets and tasks however it has a high computational cost and memory usage due to its deeper architecture. It does however offer a variation to Xception in the image sizes that it needs for preprocessing. It uses 224, 224 instead of 299, 299. We thought this would offer some variation compared to Xception while still maintaining our desire to use a non-commonly used pre-trained model.

Data and Preparation

Throughout this report you will see the use of various notable libraries including tensorflow, keras, numpy, pandas, and matplotlib. We use tensorflow and keras to do most of the heavy lifting for training, validating, and testing the models while pandas and numpy are used for data preparation and matplotlib for displaying results.

The first few sections of the report involve importing the data set animals 10 and storing all of the information including the raw data images within a directory. To obtain the data set we used two different methods one was downloading the data set locally and the other one was using an API key and username to obtain the data set straight from kaggle. Next you'll see a list of class names and their various accounts of data points per each class name. This can be seen in the printing of the array and also in a bar chart titled class distribution. An interesting note for the class distribution are the amounts of data points and images for each class. You will notice that cane and ragno have significantly more data points than others such as gato and elefante. This may lead to the model favoring cane and ragno over other less common classes. We will discuss this class distribution further in the confusion matrix and accuracy scores.



Split Data

After reviewing the data set we split it into three sets: training validation and test. I split it into 75% training, 15% validation, and 10% test sets. In future splitting I may decide to have less data in the training ratio so as to compensate for overfitting. I did not experience any overfitting in this pre-training however it may be a way to improve the model in the future. I noticed when training less epochs this ratio did impact the validation set. I tested with 3 epochs initially and it seems the validation set did not have enough time to learn the model. The Lost values were increasing initially and then about halfway through the second epoch started to decrease, however this was not enough time for the validation set's loss to settle. I extended the epochs to 10 and the validation set had more time to level out with loss. After the training there is a plot comparing the model loss for training and validation and you can see that after only 3 epochs the model loss is fluctuating quite violently, but after around 5 epochs begins to level off.

```
Training set size: 20026
Validation set size: 3535
Test set size: 2618
```

Data Augmentation

As noted earlier this data set consists of more cane and ragno data than some of the other classes. In order to counteract and reduce the likelihood of overfitting we performed data augmentation. We utilized some of the Keras properties of RandomFlip, RandomRoation, and RandomContrast to randomly flip, rotate, and adjust the contrast of the input images. These augmentations introduced variations to the training data allowing the model to learn from a more diverse set of examples. By applying these transformations randomly the model becomes more robust and less sensitive to minor variations in the input images and leads to a better generalization of the model.

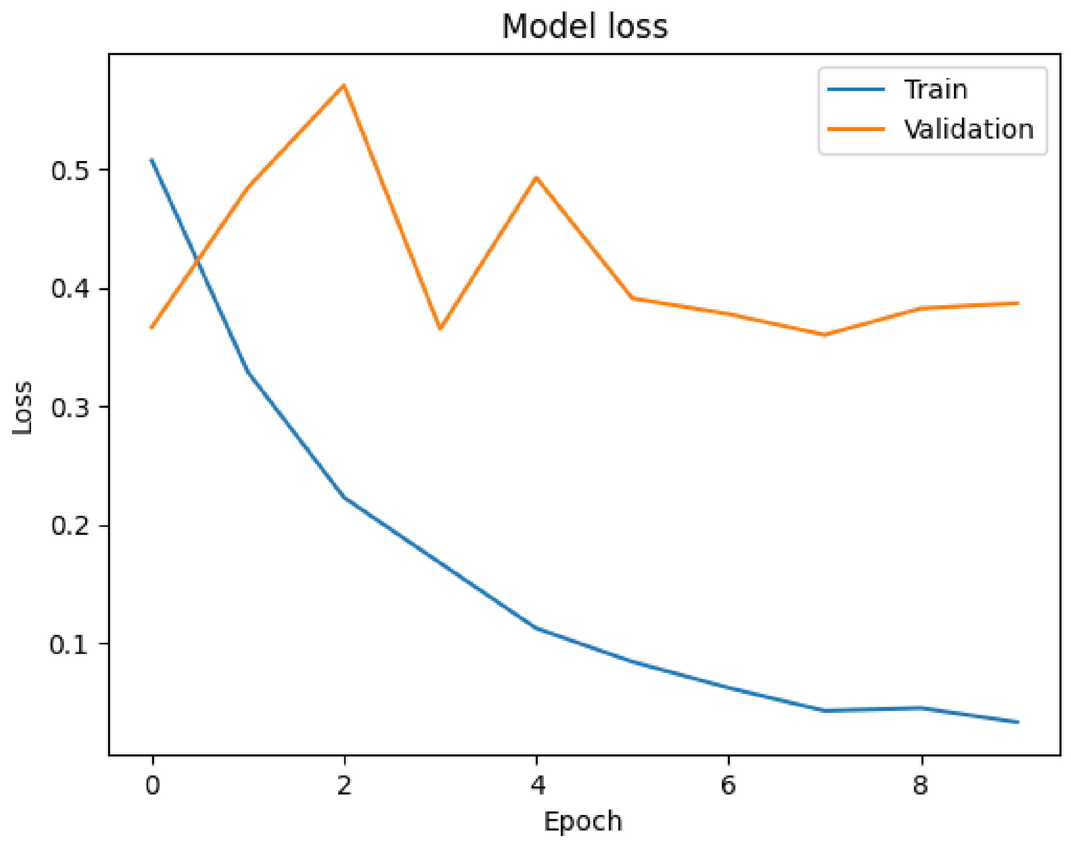
Fit Model to Test Data

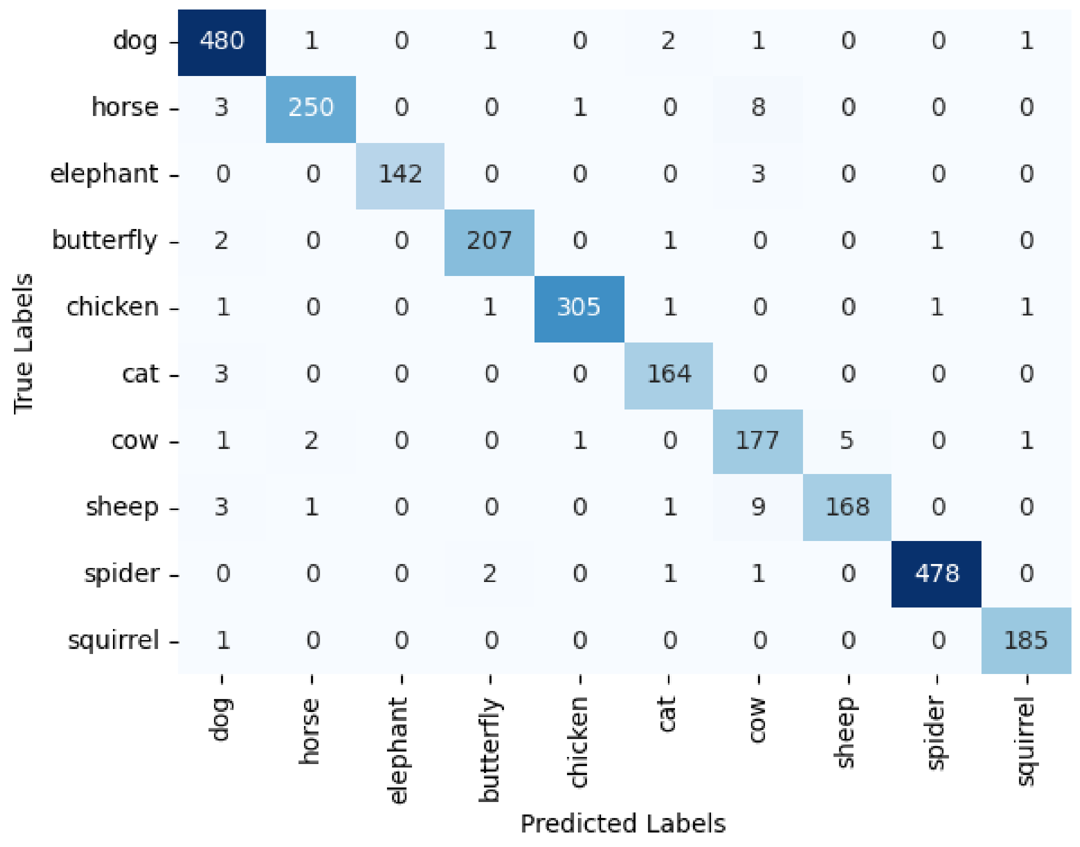
The next step in the process is fitting the model to the data. for exception the model performed very well leading to a 97.6% accuracy precision F1 score and recall. only slightly lower at 95.5%. We can also gather further Insight by looking at the confusion Matrix. You will note that these models perform very well, usually only confusing images for those animals that are similar such as cows and sheep, cows and horses, butterflies and spiders, etc. One more note is that most of the confusion happens between the dog class and other classes. Some of this may be because the dog class is the most common in the data set.

Evaluation Methods

Xception

```
82/82 [==============================] - 19s 229ms/step - loss: 0.5393 - accuracy: 0.9763 Test accuracy: 97.63%

Accuracy: 0.9763177998472116
Precision: 0.9768056431986176
F1 Score: 0.9764041199995079
Recall: 0.9763177998472116
```

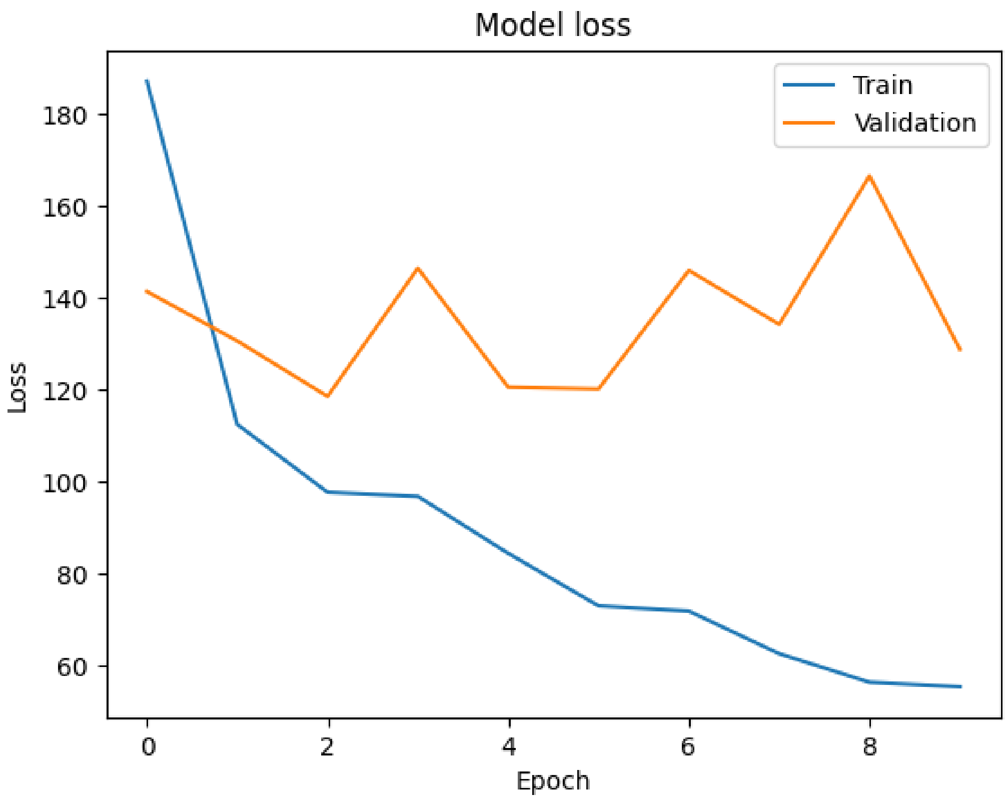


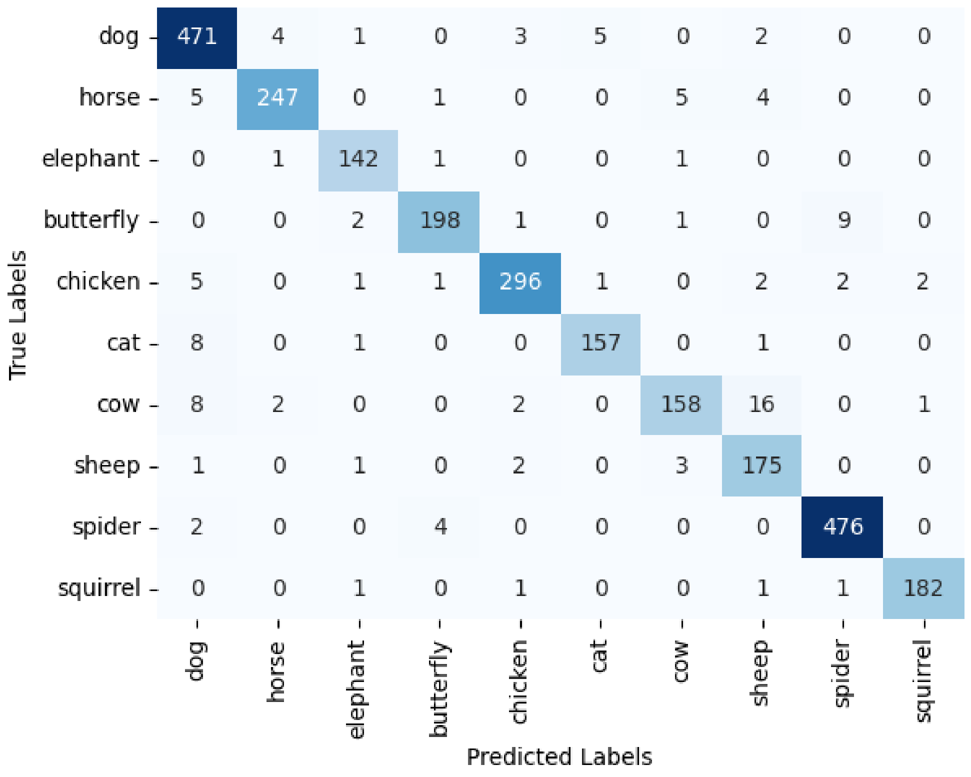

True Labels
Predicted Labels

	dog	horse	elephant	butterfly	chicken	cat	cow	sheep	spider	squirrel
dog	480	1	0	1	0	2	1	0	0	1
horse	3	250	0	0	1	0	8	0	0	0
elephant	0	0	142	0	0	0	3	0	0	0
butterfly	2	0	0	207	0	1	0	0	1	0
chicken	1	0	0	1	305	1	0	0	1	1
cat	3	0	0	0	0	164	0	0	0	0
cow	1	2	0	0	1	0	177	5	0	1
sheep	3	1	0	0	0	1	9	168	0	0
spider	0	0	0	2	0	1	1	0	478	0
squirrel	1	0	0	0	0	0	0	0	0	185

VGG16

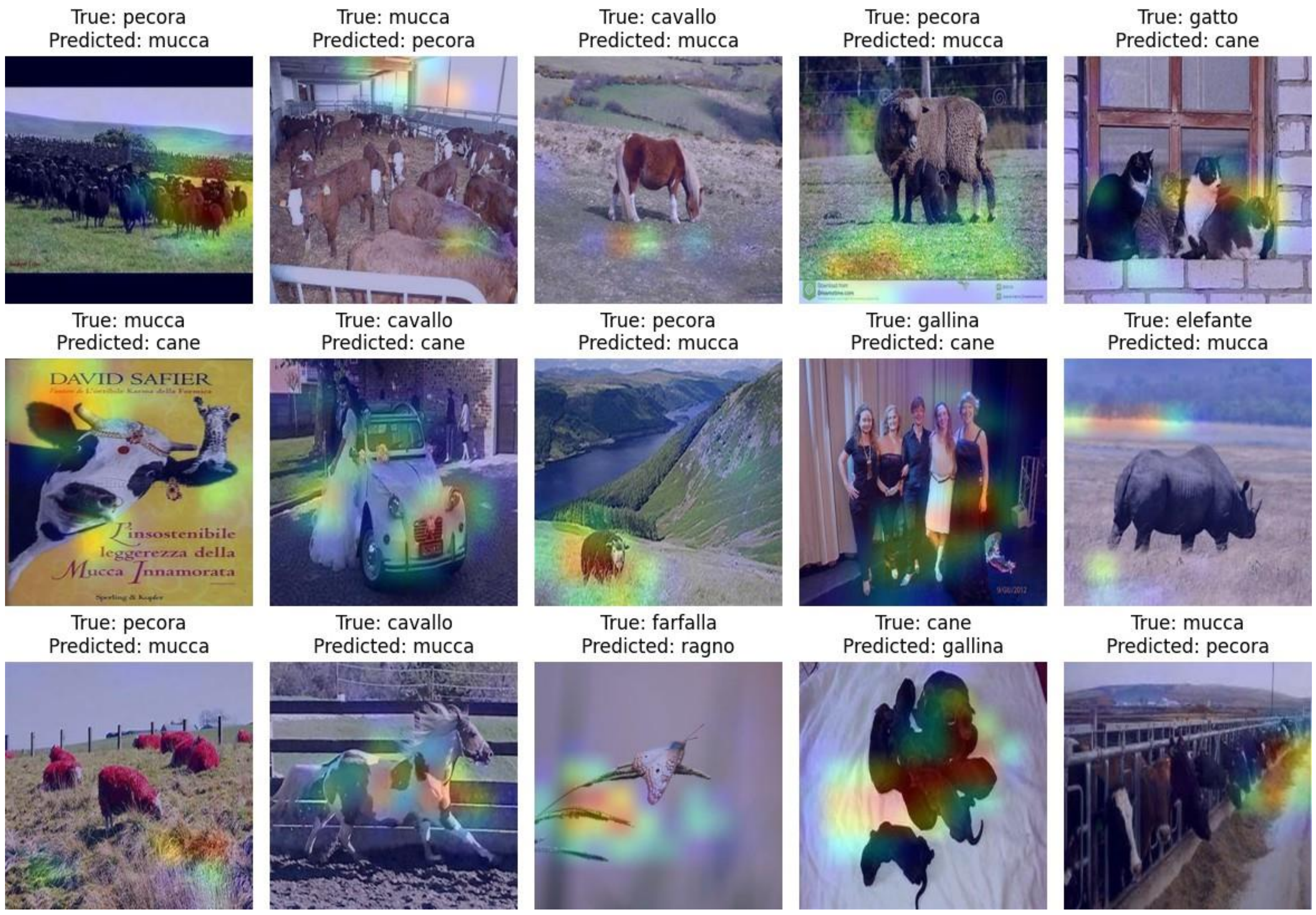
```
82/82 [==============================] - 13s 159ms/step - loss: 141.6045 - accuracy: 0.9557 Test accuracy: 95.57%

Accuracy: 0.9556913674560733
Precision: 0.9563551843220685
F1 Score: 0.9555753527223924
Recall: 0.9556913674560733
```





	dog	horse	elephant	butterfly	chicken	cat	cow	sheep	spider	squirrel
dog	471	4	1	0	3	5	0	2	0	0
horse	5	247	0	1	0	0	5	4	0	0
elephant	0	1	142	1	0	0	1	0	0	0
butterfly	0	0	2	198	1	0	1	0	9	0
chicken	5	0	1	1	296	1	0	2	2	2
cat	8	0	1	0	0	157	0	1	0	0
cow	8	2	0	0	2	0	158	16	0	1
sheep	1	0	1	0	2	0	3	175	0	0
spider	2	0	0	4	0	0	0	0	476	0
squirrel	0	0	1	0	1	0	0	1	1	182

Grad-CAM



Ensemble

Ensemble learning involves combining multiple individual models to create a stronger and more accurate predictive model. Instead of relying on a single model's predictions ensemble learning leverages the collective predictions of the models to improve the overall performance of the prediction. We employed two types of Ensemble learning: majority voting Ensemble and weighted ensemble.

Majority Voting

For majority voting we utilized the predictions across the three models and chose the prediction that came up the most times for each image, so ⅔ times in this instance. The results were 95.9% accuracy, precision, F1 score, and recall. This is less than Xception but more than the VGG16 and the Homemade CNN on their own. The results are less than Xception because the voting weights are equally distributed across the three models for each prediction and thus Xception can be outvoted by VGG16 and the Homemade CNN. These results led to the implementation of the weighted ensemble where we could assign weights to each model.

Accuracy: 0.959511077158136
Precision: 0.9631159794025888
F1 Score: 0.9598418813943396
Recall: 0.959511077158136

Weighted Ensemble

Weighted ensemble learning assigns varying weights to the predictions of different models based. The weights in this case are predetermined by us and therefore we can control which model gets more voting power for the prediction. Because Xception had a higher accuracy on its own we chose to give it more weight while still allowing it to be overridden if both VGG16 and the Homemade CNN had the same prediction. The accuracy of the weighted ensemble was still less than Xception alone at 97.4% but it was an improvement of 2% over the majority voting method.

Accuracy: 0.9744079449961803
Precision: 0.9749730668265956
F1 Score: 0.9745147607002961
Recall: 0.9744079449961803

Conclusion

In future iterations we might increase the weight of Xception further, but it may be more interesting to perform a weighted ensemble or majority voting ensemble with multiple iterations of the Homemade CNN. It seems that improving a very highly accurate model such as Xception is very difficult, however using the ensemble learning may improve models that perform poorly on their own. Also, it may be interesting to combine models that are accurate specifically on the images where Xception sometimes struggles. For example, we could construct a model that performs exceptionally well on cow and sheep differentiation or butterfly and spider differentiation, and allow it more weight when the prediction percentages are close between those two classes.

Sources

Kaggle Notebooks:

- https://www.kaggle.com/code/emreiekyurt/bird-species-classification-with-deep-learning
- https://www.kaggle.com/code/vencerlanz09/bird-classification-using-cnn-efficientnetb0

Textbooks:

- Masís, S. (2021). Interpretable Machine Learning with Python: Learn to Build Interpretable High-performance Models with Hands-on Real-world Examples.
- Geron, A. (2022). Hands-On Machine Learning with Scikit-Learn, Keras & Tensorflow: Concepts, Tools, and Techniques to Build Intelligent Systems.
- Raschka, S. Mirijalili, V. (2019). Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and Tensorflow.