Machine Learning Engineer Nanodegree

Capstone Project

Luiz Antônio Nonenmacher Júnior
April 2019

I. Definition

Project Overview

The idea for this project came from the combination of some tendencies that I see in the machine learning field with some skills that I would like to develop as a machine learning engineer.

The first tendency is that deep learning requires a lot of data to be successful [1][2], which is generally not available for small and medium size companies. The second tendency is that most of the job of machine learning practitioners are data cleaning and preparation [3]. Because of this, my project will be based on transfer learning using a small dataset collected and prepared by myself, so I can train those data processing skills and I can simulate an environment of a small company.

The domain where I will apply this project is computer vision, which is one the fields where deep learning has a much better performance than traditional models. To exemplify this, we could look at the ImageNet performance in Figure 1:

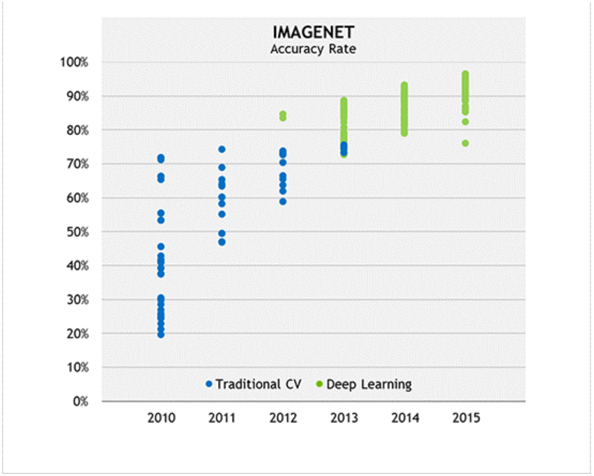


Figure 1: Performance on ImageNet. Extracted from [4]

In 2012 the ImageNet competition was won by a deep learning algorithm that performed better than handcrafted software written by computer visions experts. In 2015, another deep learning algorithm has performed better than humans on classifying those pictures [4].

There are several application of computer vision in different industries, like automotive, retail, financial services and healthcare [5], which make this field an important skill to master for machine learning practitioners.

Problem Statement

Given the background, I want to do a project that involved computer vision, transfer learning and the collecting and processing of data. To this end, I will use my two cats as subjects. My problem will be, given a photo of one of them, classify each cat it is. This problem *per se* is not useful for companies, but the same idea (given a photo classify into two or more categories) can be applied for different kind of problems.

To solve this problem, the first step will be to preprocess the data, transforming them from their original size (for example, 3088 x 2320 or 3724 x 2096) to a square format (331x331 and 299x299) by the models in the next steps. After it, I will create and train a CNN from scratch to serve as benchmark. After training this benchmark model, I will implement transfer learning, by importing three models (NasNet Large, Xception and InceptionResNetV2) and training only the last layer. Finally, the last step will consist of improving the results of this model by using transfer learning.

Metrics

The ImageNet dataset (where the models I will use for transfer learning are trained) have two main evaluation metrics: top-1 accuracy and top-5 accuracy. Top-1 accuracy is calculated by looking at the class that the model gives the biggest probability and to compare with the actual label. Top-5 accuracy is similar, but we look at the five classes with more probability and compare it with the real class.

In my problem, there is no top-5 accuracy because the model just predicts two probabilities, so I will use just the simple accuracy, by counting how many pictures did the model has correctly classified.

II. Analysis

Data Exploration

The dataset used in this project consists of 400 images taken by myself using a smartphone, half of them containing one cat and the other half the second cat (I took and selected the same number of pictures from each cat so the dataset would be balanced). Those pictures were taken in my home and the size and position of the cats vary. Above, there are two sample pictures from each of the cats (Gepetto and Kuki):

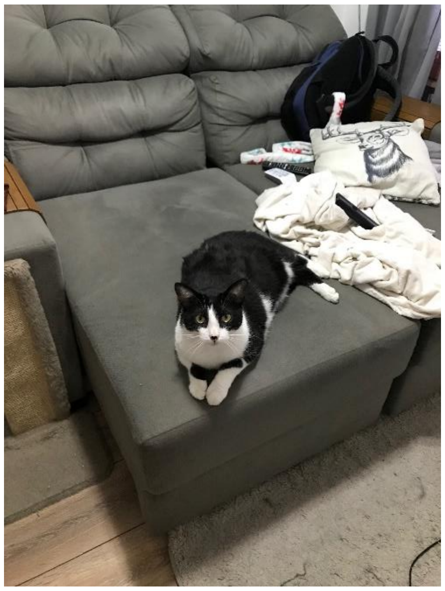


Figure 2: Sample picture 1 (Gepetto)

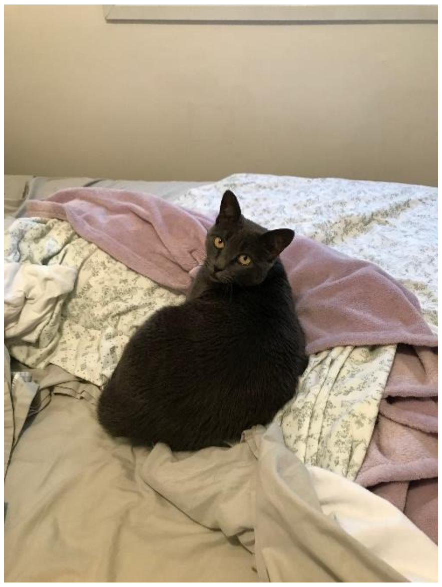


Figure 3: Sample picture 2 (Kuki)

The original pictures are in .JPEG and .JPG formats and all the pictures are colored. Most of the pictures has a width of 3024 and a height of 4032, but they are some pictures in a wide format (for example, 3088 x 2320 or 3724 x 2096). I will use 60 images (15% of the dataset), 30 from each cat, as test set and the remaining 340 pictures as training/validation set. When training, I will use 15% of the remaining 340 pictures as validation set.

Exploratory Visualization

The dataset consists entirely of images, so one important analysis to be done is to compare the images orientation (Landscape x Portrait) and resolution. Figure show that 83.8% (335 of 400) images are portrait-oriented, which is expected given that the majority of the pictures were taken by smartphones.

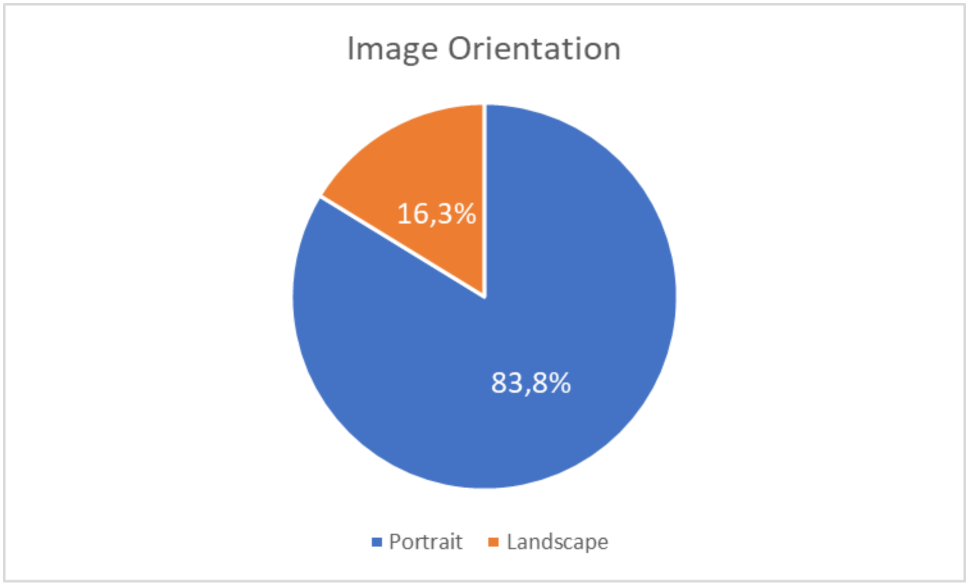


Figure 4: Image Orientation Distribution

In terms of resolution, there is 46 different resolutions in the dataset. Figure 5 show the percentage of each resolution in the dataset, grouping the least used resolutions (3 images or less) into an "Others" group. One conclusion that can be taken from this figure is that most of the pictures are concentrated in three resolutions, as 69.3% of the images comes from one of those three options (4032x3024, 2048x1536 and 3724x2096).

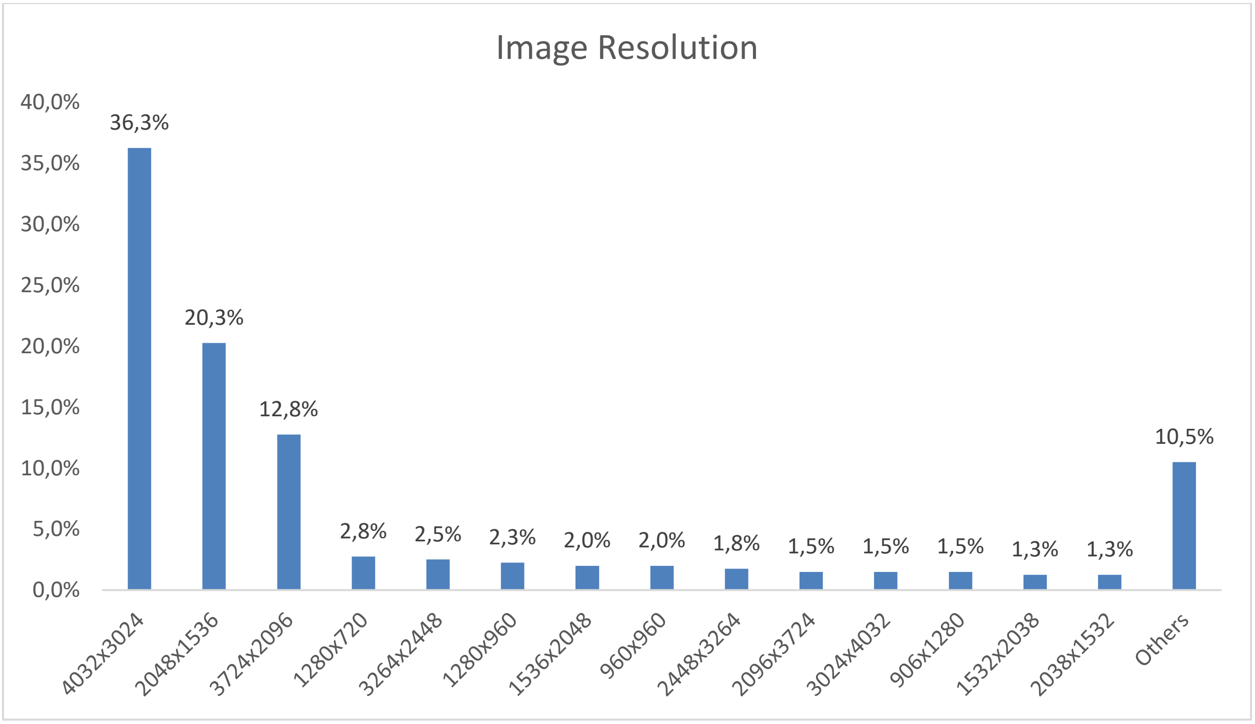
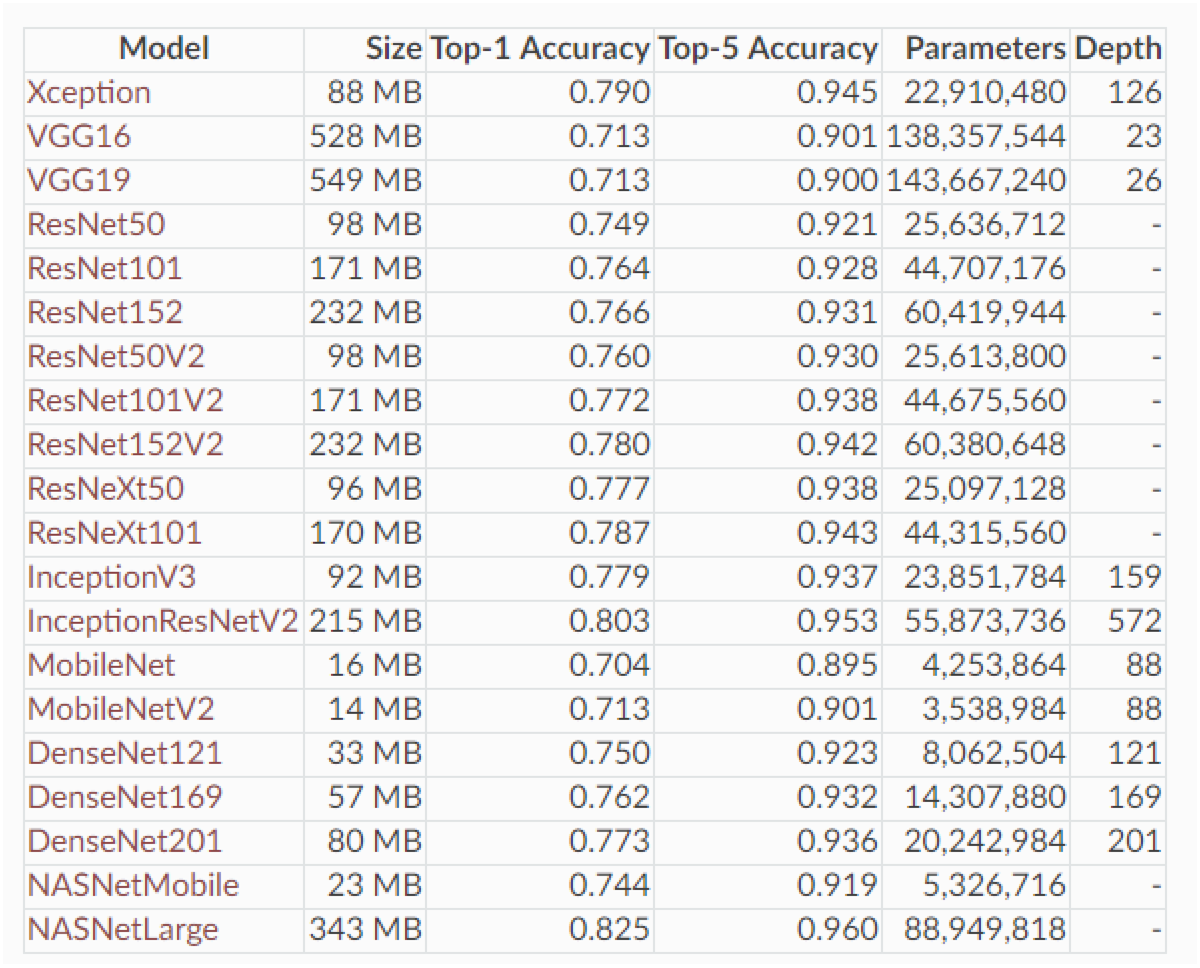


Figure 5: Image Resolution Distribution

Algorithms and Techniques

All the models used in this project are Convolutional Neural Networks, the state-of-the-art methods for tasks involving images. To select the three models I compared all pre-trained models available in Keras Applications [6] by their Top-1 Accuracy and selected the three methods with the best performance (NasNetLarge, InceptionResNetV2 and Xception). Figure 6 shows the table extracted from [6]:



Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	98 MB	0.749	0.921	25,636,712	-
ResNet101	171 MB	0.764	0.928	44,707,176	-
ResNet152	232 MB	0.766	0.931	60,419,944	-
ResNet50V2	98 MB	0.760	0.930	25,613,800	-
ResNet101V2	171 MB	0.772	0.938	44,675,560	-
ResNet152V2	232 MB	0.780	0.942	60,380,648	-
ResNeXt50	96 MB	0.777	0.938	25,097,128	-
ResNeXt101	170 MB	0.787	0.943	44,315,560	-
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201
NASNetMobile	23 MB	0.744	0.919	5,326,716	-
NASNetLarge	343 MB	0.825	0.960	88,949,818	-

Figure 6: Models accuracy on ImageNet (extracted from [6])

The NASNet Large (Neural Architecture Search Network) is a model developed by Google AI in 2018 [7] that was obtained by using the AutoML project, an approach that makes the model learn the best architecture for some task by using some building blocks (like convolution, average pooling and max pooling layers) [8][9]. This specific model architecture was built to solve the CIFAR-10 dataset and after was applied to the ImageNet dataset, obtaining a state-of-the-art performance at that time.

The InceptionResNetV2 [10] is a model developed by Google AI in 2016 that combines the earlier model version Inception V4 with Microsoft's ResNet models[11]. This combination is based on using the Inception blocks developed by the Inception models with the Residual connections used on the ResNet models. Below are a diagram of the model architecture (taken from [10]):

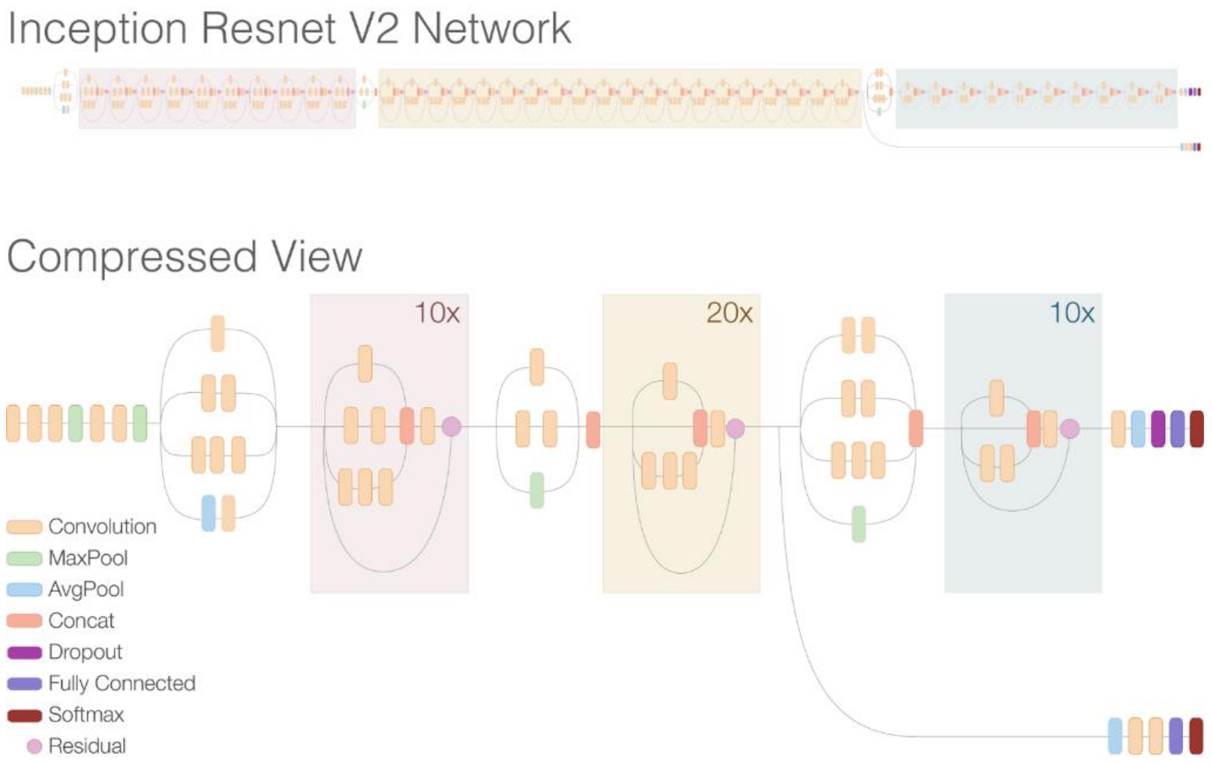


Figure 7: InceptionResNet V2

When the model had launched it had a state-of-the-art performance on the ILSVC 2012 image classification benchmark (that used the ImageNet database) [12].

The Xception model (Extreme Inception)[13] is a model developed by François Chollet, a AI Researcher at Google and one of the core developers of Keras [14], in 2017. The basic idea of the model is to create an architecture based entirely on depthwise separable convolution layers, which can be understand as a stronger version of the Inception hypothesis used on the Inception family of models (hence the name Extreme Inception). Figure 8 contains the model architecture, as found on [13].

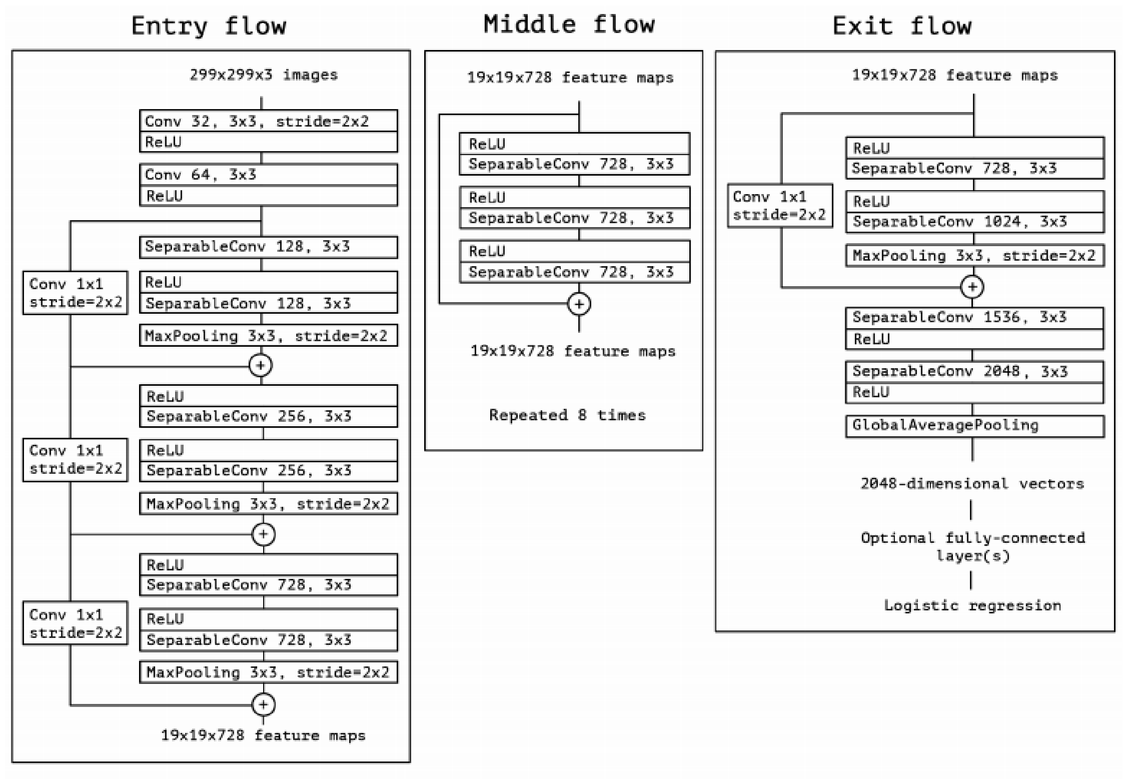


Figure 8: Xception Network

Benchmark

The results of the pre-trained models on ImageNet are not totally comparable to my results, because they are trying to classify images into several classes, where I will just classify of type of image (cats) into two categories. Also, one of the fundamentals of my project is that I will be using self-taken pictures, so there is no benchmark model on the internet that used the same dataset (especially because the quality of the pictures and the small size of the dataset).

To serve as a benchmark, I will create and train a CNN from scratch without using transfer learning, just the 400 pictures in my dataset. Because the amount of data is small, I will use a small model without ton of layers, to reduce overfitting.

I tried three alternative architectures, but all three takes 299x299x3 arrays as input as pass it through pairs of Convolution/MaxPooling Layers. After it, I used a Global Average Pooling (GAP) layer to flatten the result of the last layer and finally pass the results trough some fully connected layers to obtain the probability of each class. More details about each model architecture and training are located on the implementation segment.

III. Methodology

Data Preprocessing

The first part of the project was to take the original images and convert it to a format that can be passed to CNNs. The preprocessing was done using the opencv and the numpy libraries.

First, I created a function that given a path to a file and a desired format (331x331 or 299x299):

- Read the image into an numpy array;

- Converted it to RGB (the default import format for opencv is BGR [15]);

- Resized it to the desired format;

- Divided each pixel value by 255 (to increase the convergence speed of the CNNs, as stated on [16][17]);

- Output the final numpy array.

After defining this function, I looped trough all images and saved the results into two numpy arrays (one for each desired format, X_299 and X_331). I also created a y array that contains 0 for the first 200 pictures and 1 for the remaining 200 (the cat pictures were loaded in order).

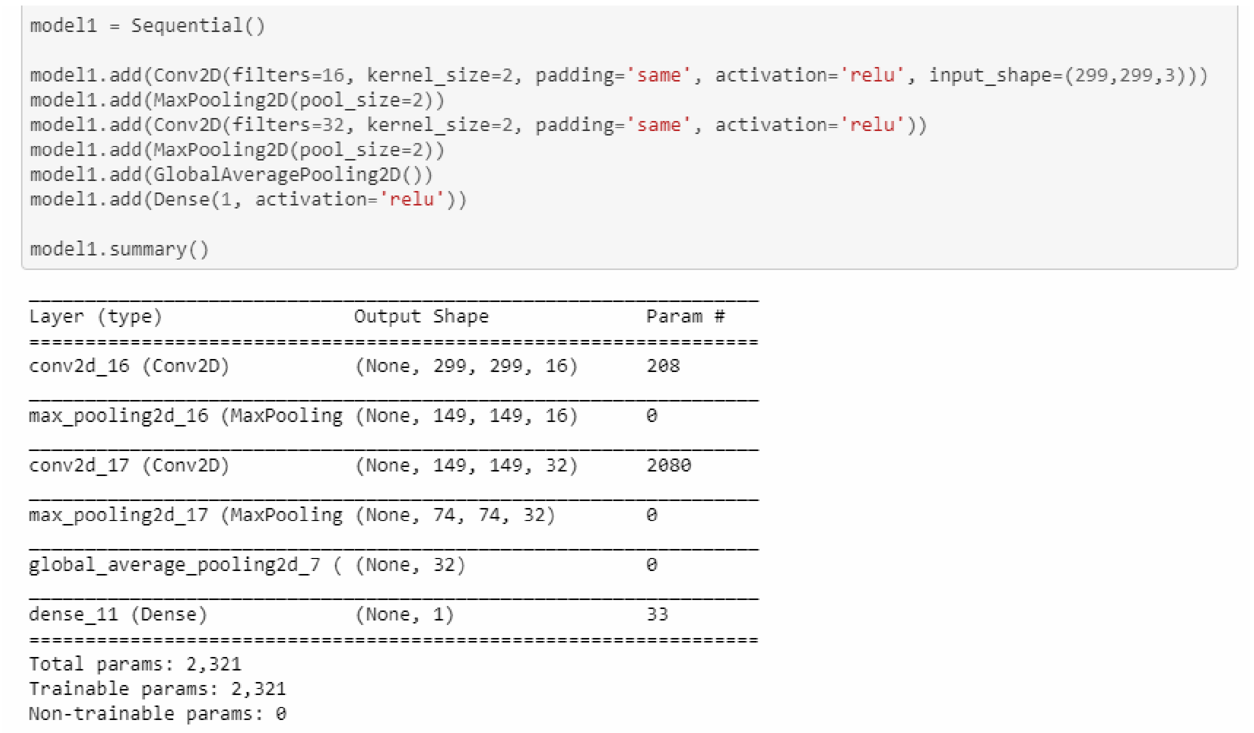
The next step after obtaining the processed images arrays was to obtain the bottleneck features. Those features are calculated for each of the three imported models (NasNet Large, Xception and InceptionResNetv2) by passing the processed images trough the networks (without the last layer) and saving the output (in files called nasnet_bottleneck, inception_bottleneck and xception_bottleneck).

Those outputs will be used in the next step to train the models to classify the cats. Obtaining the bottleneck features and training just the final layers has the same effect (in terms of classification performance) of training all the network but freezing the first layers, but using bottleneck features increases the training speed, because we just need to pass the images trough all the network once.

Implementation

Before using transfer learning, the first step was to create and train the benchmark models. I test three different architectures, but all followed the same structure of pairs of convolutional/pooling layers followed by a GAP layer and a final fully connected layer. Also, all models take as input a 299x299x3 array.

The first model (Figure 9) is the simplest model of the three and consists of two pairs of convolutional/max pooling layers, with the first convolutional containing 16 filters and the second 32. In both convolutional layers I used "same" padding and the "relu" activation function. After those layers I added a Gap layer (to flatten the output) and finally a 1-neuron fully connected layer that predict the probability of the cat being Kuki (given that she has the label 1).

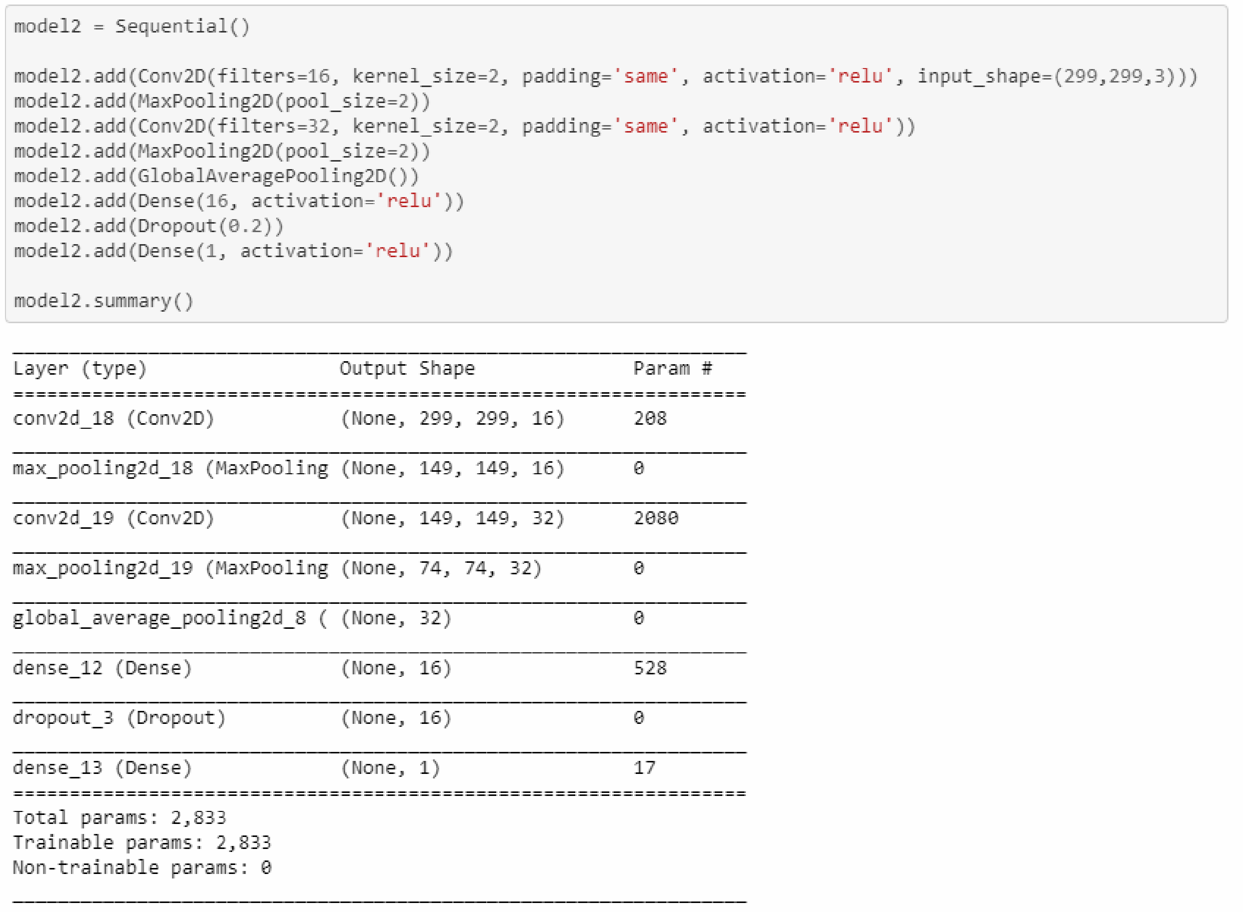


```
model1 = Sequential()

model1.add(Conv2D(filters=16, kernel_size=2, padding='same', activation='relu', input_shape=(299,299,3)))
model1.add(MaxPooling2D(pool_size=2))
model1.add(Conv2D(filters=32, kernel_size=2, padding='same', activation='relu'))
model1.add(MaxPooling2D(pool_size=2))
model1.add(GlobalAveragePooling2D())
model1.add(Dense(1, activation='relu'))

model1.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_16 (Conv2D)           (None, 299, 299, 16)      208       
_________________________________________________________________
max_pooling2d_16 (MaxPooling (None, 149, 149, 16)      0         
_________________________________________________________________
conv2d_17 (Conv2D)           (None, 149, 149, 32)      2080      
_________________________________________________________________
max_pooling2d_17 (MaxPooling (None, 74, 74, 32)        0         
_________________________________________________________________
global_average_pooling2d_7 ( (None, 32)                0         
_________________________________________________________________
dense_11 (Dense)             (None, 1)                 33        
=================================================================
Total params: 2,321
Trainable params: 2,321
Non-trainable params: 0
```

Figure 9: Benchmark Model 1

The second benchmark model is a bit more complex than the first one. The first layers are the same, but after the GAP layers I added a 16-neuron fully connected layer, a dropout layer with a drop rate of 0.2 and finally the 1-neuron fully connected layer that output the prediction. This model architecture is shown on Figure 10:



```
model2 = Sequential()

model2.add(Conv2D(filters=16, kernel_size=2, padding='same', activation='relu', input_shape=(299,299,3)))
model2.add(MaxPooling2D(pool_size=2))
model2.add(Conv2D(filters=32, kernel_size=2, padding='same', activation='relu'))
model2.add(MaxPooling2D(pool_size=2))
model2.add(GlobalAveragePooling2D())
model2.add(Dense(16, activation='relu'))
model2.add(Dropout(0.2))
model2.add(Dense(1, activation='relu'))

model2.summary()
```

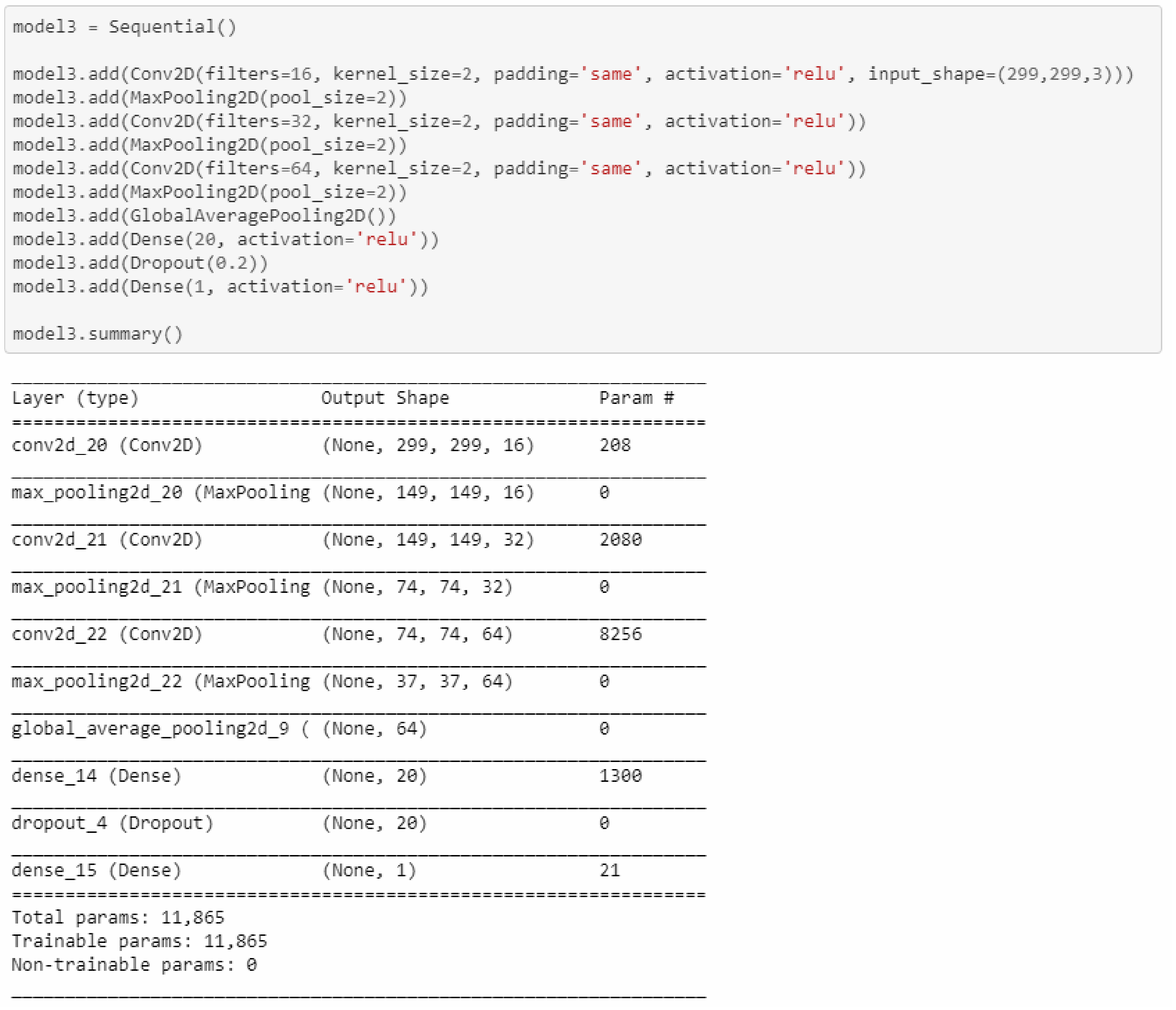
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_18 (Conv2D)           (None, 299, 299, 16)      208       
_________________________________________________________________
max_pooling2d_18 (MaxPooling (None, 149, 149, 16)      0         
_________________________________________________________________
conv2d_19 (Conv2D)           (None, 149, 149, 32)      2080      
_________________________________________________________________
max_pooling2d_19 (MaxPooling (None, 74, 74, 32)        0         
_________________________________________________________________
global_average_pooling2d_8 ( (None, 32)                0         
_________________________________________________________________
dense_12 (Dense)             (None, 16)                528       
_________________________________________________________________
dropout_3 (Dropout)          (None, 16)                0         
_________________________________________________________________
dense_13 (Dense)             (None, 1)                 17        
=================================================================
Total params: 2,833
Trainable params: 2,833
Non-trainable params: 0
_________________________________________________________________
```

Figure 10: Benchmark Model 2

The third and final benchmark model is the more complex model. It has the same structure as the second model but after before the GAP layer I added a 64-filter convolutional layer and a max pooling layer. This model is shown on Figure 11.

All the models were built and trained using Keras and the Tensorflow core. I used the adam optimizer [19][20] with the default Keras parameters [21], which are the same recommended by the original paper. As the loss function I used binary cross-entropy [22], which is a loss function suited for binary classification problems. I trained each model for 20 epochs using a checkpoint to save the best model given by the validation loss.

After training the three models, I selected the best one in terms of validation accuracy and used it to obtain a test accuracy, metric that was used as benchmark for the transfer learning.

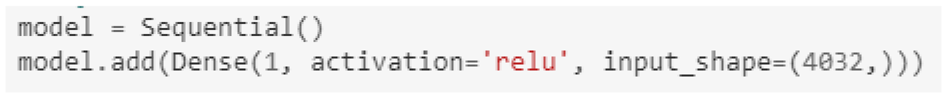


```
model3 = Sequential()

model3.add(Conv2D(filters=16, kernel_size=2, padding='same', activation='relu', input_shape=(299,299,3)))
model3.add(MaxPooling2D(pool_size=2))
model3.add(Conv2D(filters=32, kernel_size=2, padding='same', activation='relu'))
model3.add(MaxPooling2D(pool_size=2))
model3.add(Conv2D(filters=64, kernel_size=2, padding='same', activation='relu'))
model3.add(MaxPooling2D(pool_size=2))
model3.add(GlobalAveragePooling2D())
model3.add(Dense(20, activation='relu'))
model3.add(Dropout(0.2))
model3.add(Dense(1, activation='relu'))

model3.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_20 (Conv2D)           (None, 299, 299, 16)      208       
_________________________________________________________________
max_pooling2d_20 (MaxPooling (None, 149, 149, 16)      0         
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 149, 149, 32)      2080      
_________________________________________________________________
max_pooling2d_21 (MaxPooling (None, 74, 74, 32)        0         
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 74, 74, 64)        8256      
_________________________________________________________________
max_pooling2d_22 (MaxPooling (None, 37, 37, 64)        0         
_________________________________________________________________
global_average_pooling2d_9 ( (None, 64)                0         
_________________________________________________________________
dense_14 (Dense)             (None, 20)                1300      
_________________________________________________________________
dropout_4 (Dropout)          (None, 20)                0         
_________________________________________________________________
dense_15 (Dense)             (None, 1)                 21        
=================================================================
Total params: 11,865
Trainable params: 11,865
Non-trainable params: 0
_________________________________________________________________
```

Figure 11: Benchmark Model 3

After training the benchmark model, the next step was to use transfer learning to train the models using the bottleneck features. For each bottleneck feature I teste three different architectures. The first one was simply a 1-neuron fully connected layer that outputs the probability, as shown of Figure 12 (the code shown on picture is for the NasNet model, but the same can be applied for other models changing the input_shape):



```
model = Sequential()
model.add(Dense(1, activation='relu', input_shape=(4032,)))
```

Figure 12: Transfer Learning Model 1

The second alternative add a 10-neuron fully connected layer before the final layer, as shown on Figure 13:

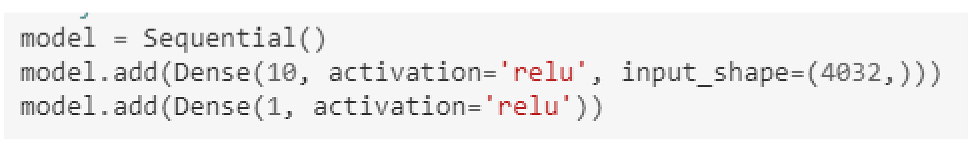

```
model = Sequential()
model.add(Dense(10, activation='relu', input_shape=(4032,)))
model.add(Dense(1, activation='relu'))
```

Figure 13: Transfer Learning Model 2

Finally, the third model adds a dropout layer with a drop rate of 0.2 (Figure 14). One important thing to notice is that all models are very simple, which was done to reduce overfitting given my small dataset.

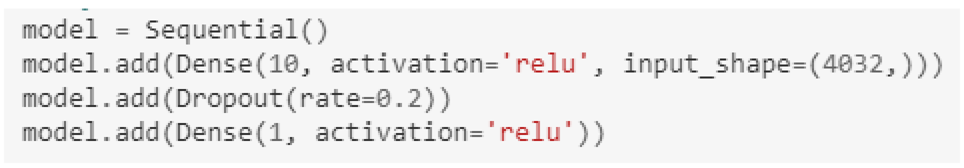

```
model = Sequential()
model.add(Dense(10, activation='relu', input_shape=(4032,)))
model.add(Dropout(rate=0.2))
model.add(Dense(1, activation='relu'))
```

Figure 14: Transfer Learning Model 3

All the models were trained with parameters similar to the benchmark models, using the adam optimizer, binary cross-entropy and using a checkpoint to save the best model in terms of validation loss.

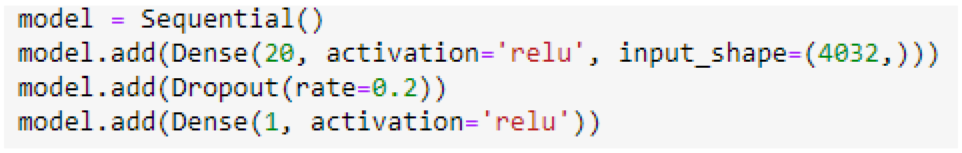
To select the best model, I run a 5-step loop that at each step split the data into training and validation set, trained the model and calculated the validation accuracy of the model. The final model was selected as the one with the best mean validation accuracy. After finding the best model, I calculated the test accuracy to compare with the benchmark model accuracy (for consistency, the test set were the same in both situation).

Refinement

After obtaining the best model, the next step to refine it and to increase its results was to use data augmentation. This is done by adding some minor changes (like rotation and translation) to the images in our dataset, so that we have more images to train and can reduce overfitting [23].

The first step is to import the ImageDataGenerator class from Keras preprocessing library [24]. Then I started a generator with the default parameters used by Keras on a example on [24] and created two batches of augmented images passing the 340 original images used for training and validation (the test set was not used so the training would not see those images). With this, I had now 1020 images to train and validate the model (340 original images + 640 augmented). After this, I passed all the images trough the model with the best results in the previous section to obtain the new bottleneck features.

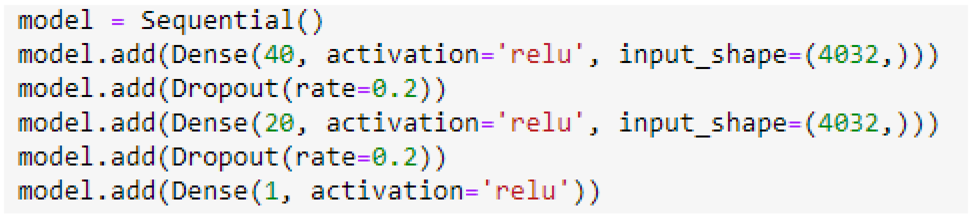
With the bottleneck features, the next step is to train the models using the new data. I trained three alternative architectures. The first alternative is the same as the Model 1 described in the Figure 12. The second alternative is similar to the Model 3 (Figure 14), but instead of 10-neurons in the fully connected layer I added 20-neurons (Figure 15).



```
model = Sequential()
model.add(Dense(20, activation='relu', input_shape=(4032,)))
model.add(Dropout(rate=0.2))
model.add(Dense(1, activation='relu'))
```

Figure 15: Data Augmentation Model 2

Lastly, the third alternative is bigger than the second model because before I added a 40-neuron fully connected layer and a dropout layer in the beginning of the model (Figure 16):



```
model = Sequential()
model.add(Dense(40, activation='relu', input_shape=(4032,)))
model.add(Dropout(rate=0.2))
model.add(Dense(20, activation='relu', input_shape=(4032,)))
model.add(Dropout(rate=0.2))
model.add(Dense(1, activation='relu'))
```

Figure 16: Data Augmentation Model 3

One important thing to notice is that the models are bigger than the models used on the Implementation segment because now I have more data to train. The training was similar to the training used in the previous segment and I choose the final model based on the mean validation accuracy over 5 loops. Finally, after selecting the best model, I calculated the test set accuracy to compare to the previous results.

IV. Results

Model Evaluation and Validation

The first result of this project was the three models trained as benchmark models (Table 1). The best accuracy obtained was 0.667, which is not a great result overall but good considering that I'm training a CNN from scratch with small quantity of data. Also, another thing to notice is that the simpler the model is (Model 1 is simpler than Model 2, that is simpler than Model 3) the better was the accuracy. The main reason for this is because bigger models have more space to overfit, which is a problem considering my dataset size.

Table 1: Benchmark Models Validation Accuracy

Model	Accuracy
Model 1	0.667
Model 2	0.647
Model 3	0.588

Model 1 was the best model in terms of validation accuracy, so I took it and calculate the accuracy on the test set. The result was 0.617, which is, as expected, smaller than the validation accuracy (even this simple model can overfit with so few data) but still a good result. This accuracy will be the benchmark for the next models.

The results of the transfer learning implementation are presented on Table 2. For each model and architecture (described in the Implementation segment) the table show the accuracy for each of the five loops (Acc 1, Acc 2, ...) and the Mean Accuracy (Mean Acc).

Table 2: Transfer Learning Validation Accuracy

Model	Architecture	Mean Acc	Acc 1	Acc 2	Acc 3	Acc 4	Acc 5
NasNet Large	1	0.9530	0.9216	0.9608	0.9412	0.9608	0.9804
NasNet Large	2	0.8588	0.5294	0.9412	0.9216	0.9216	0.9804
NasNet Large	3	0.9490	0.902	0.9608	0.9216	0.9804	0.9804
Inception ResNetv2	1	0.8392	0.9216	0.8824	0.9412	0.9608	0.4902
Inception ResNetv2	2	0.7882	0.5098	0.9804	0.9804	0.4902	0.9804
Inception ResNetv2	3	0.77645	0.9216	0.9804	0.9804	0.4902	0.5098
Xception	1	0.5020	0.5098	0.5098	0.5098	0.4902	0.4902
Xception	2	0.7373	0.7255	0.8824	0.8235	0.4902	0.7647
Xception	3	0.6314	0.8039	0.5098	0.8627	0.4902	0.4902

The first thing to notice is that NasNet Large is the best model of the three considering both the accuracy on the best architecture (0.9530) and the mean for the three architectures (0.9203 for the NasNet, 0.8013 for the Inception and 0.6235). In terms of ordination, those results are the same as the ImageNet results presented in Figure 6 (where NasNet is the best model followed by Inception), but the overall performance should not decrease so much between models (especially for the Xception).

One way to look at it is to compare the size of the output (bottleneck features) of each model. The NasNet has an output size of 4032, whereas Inception has 1536 and Xception 1000. In this problem we have few data to train the networks, so we need most of the job to be done by the transfer learning model, so is was expected that models with a bigger output had better results, which happened.

Looking at the architectures, we can see that the architecture 1 has the best results for the NasNet and the Inception networks. Architecture 1 is simplest one, just one neuron to predict the final probability, and its good performance can be understood because the quality of the transfer learning models output and the least amount of overfitting possible. On the other hand, Architecture 1 was bad on the Xception model, with the final accuracy being almost the same as a random guess. This could be caused because the 1000 bottleneck features were not able to correctly classify the images, needing more neurons to create a valid classification.

The second architecture is more complex than the first one by adding a 10-neuron layer before the final layer, so it could create more complex relations based on the bottleneck features but could lead to overfit. Looking at the NasNet model, we can see that the architecture 2 had a good performance except for the first run, where it probably overfit to the training set and could not get a valid accuracy (the result, 0,5294 is almost random guess). The same behavior happened in the first iteration of the Inception model. For the Xception the situation was opposite, because now the model had predictions above the random line, although it still overfit (iteration 2) and the final performance was worse than the other two models.

Finally, the third architecture was built to reduce the overfit from the second model, by adding a dropout layer between the two layers. In the first model it worked and it increased the performance, getting almost the same as the first architecture (and the second greatest performance of all combinations of models and architectures). For the Inception network the dropout layer had no effect, getting almost the same performance as the second architecture. For the Xception network the dropout layer reduced the accuracy. This can be understood in the light of the output size; the Xception has the smallest number of bottleneck features and by adding dropout we reduced the actual information used at each epoch (because we didn't use all neurons), so we had a worse final performance.

After those results, I selected the NasNet Large model with the first architecture to be our best transfer learning model. Using this model to predict the images in the test set, I obtained an accuracy of 0.9167, a result smaller than the validation accuracy of 0.9530 (which indicates that the model had overfit a bit) but a great result compared to our problem and the performance of the bottleneck model (0.617).

The next step was to take the NasNet Large model and train it using data augmentation. Table 3 shows, similar to Table 2, the validation accuracy for the three architectures tested. It is important to note that the three architectures are different than the ones used with the transfer learning model, as described in the Refinement section.

Table 3: Data Augmentation Validation Accuracy

Model	**Architecture**	**Mean Acc**	**Acc 1**	**Acc 2**	**Acc 3**	**Acc 4**	**Acc 5**
NasNet Large	1	0.50064	0.5033	0.4967	0.5098	0.4967	0.4967
NasNet Large	2	0.76864	0.7647	0.6928	0.7778	0.8105	0.7974
NasNet Large	3	0.73854	0.7451	0.7516	0.7908	0.7908	0.6144

The first architecture was the simplest one with just one neuron. Previously, it could get nice results, but now that we added augmented images it was not able to classify correctly the images, so I had to use more complex models to get a better accuracy. The architectures 2 and 3 had similar results but we select the architecture 2 as the final model because of the bigger mean accuracy and because the architecture is simpler than the third architecture, which could lead to less overfit in the test set.

Another thing to take notice is that the validation accuracy for all the three architectures is smaller than the ones obtained without data augmentation. This, however, doesn't mean that data augmentation didn't work and reduced the performance because we now are using a different and more difficult (given the noise) data set for the validation set, so it was expected that the validation could decrease. In the final test set we didn't apply the augmentation, so the final results would be comparable to the application without data augmentation.

Finally, taking the second architecture and applying to the test set, I obtained a test set accuracy 0.95, a great result and an improvement of the previous model. This accuracy is bigger than the validation accuracy, which proved my hypothesis about the augmented dataset being harder to predict.

Justification

In this project, I tested three different models (benchmark model, transfer learning and data augmentation) on the test set. Figure 17 show the results:

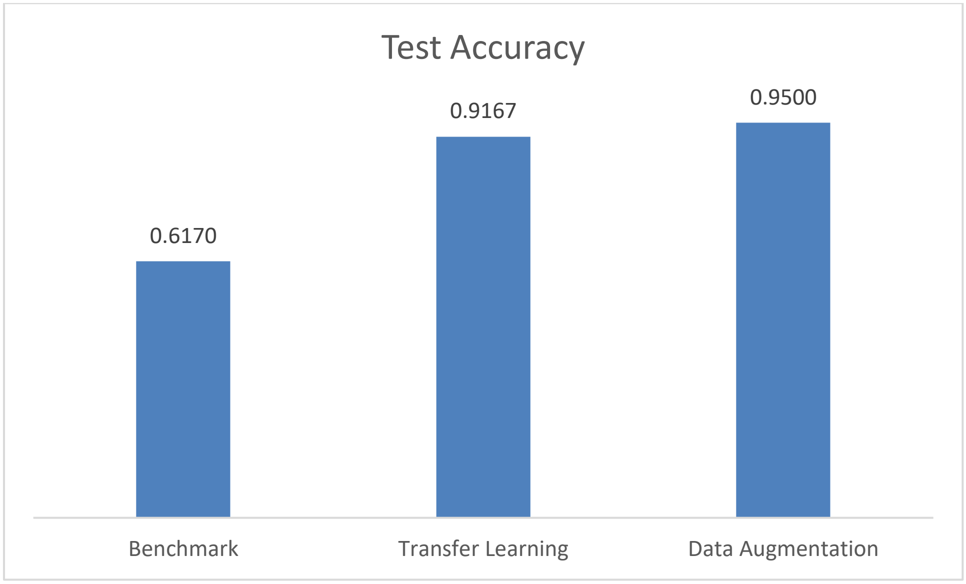


Figure 17: Test accuracy for each model

Looking at the Figure, we can see that we had a great improvement in accuracy when switching from a CNN trained from scratch (Benchmark) to use transfer learning, which was the main purpose of the project. Using data augmentation we could increase the accuracy even more, which is a good result in the context of this project.

V. Conclusion

Free-Form Visualization

The final model using Data Augmentation has obtained an accuracy of 95.00%, which corresponds to an error of 3 of the 60 images that composed the test set. As visualization, we will look at those 3 images to try to understand what could have caused the error.

Figure 18 represents the first error, as Kuki (the gray cat) is incorrectly classified as Gepetto (the black-and-white cat). The main reason I found for this misclassification is because of the background, because looking at the training dataset there was two other pictures of Gepetto in the same background but no one of Kuki, so maybe this caused to model to get confused and misclassify.

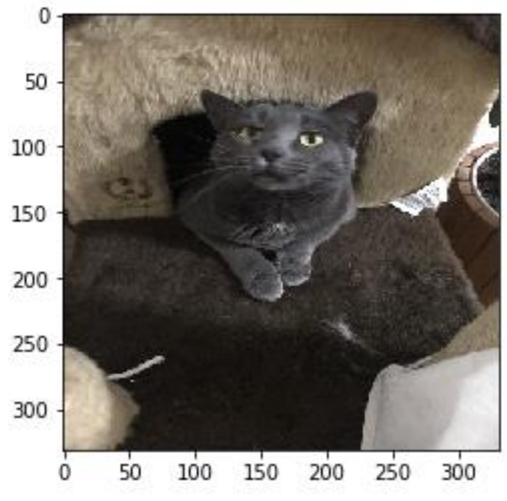


Figura 18: First misclassification

The second error is shown of Figure 19. This is also Kuki misclassified as Gepetto, but this error is more understandable, because the size of the cat in the image is very small, so the model has just little information to correctly classify the cat.

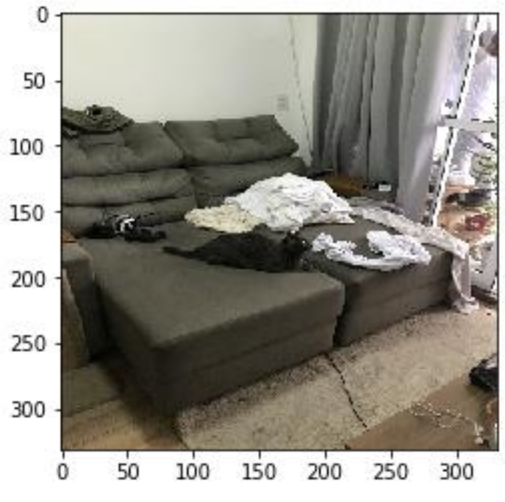


Figure 19: Second misclassification

Finally, Figure 20 shows the third and last error. In this situation Gepetto as classified as Kuki. This error could be caused by illumination problems (which can be seen in the cat face), just a little glimpse of the cat eyes and the black background (monitor), which the algorithm could mix with the black of the fur and have problems identifying the cat.

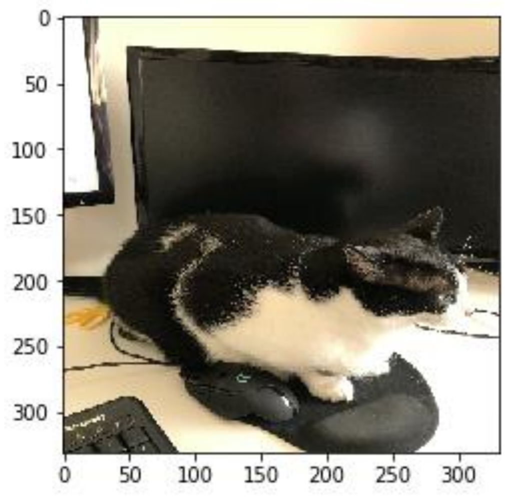


Figure 20: Third misclassification

Reflection

As stated in the introduction and in the project proposal, my intention with this project was to exercise data manipulation (a skill very important for almost all machine learning projects) and to use transfer learning with a small dataset, a situation that can be found in many companies.

Those objectives were fully reached by this project. The image manipulation part was hard at the beginning, as I had never worked directly with images before (just used some preprocessed dataset) and was a bit lost at the beginning, but after some research and trial I was able to create a process and use the correct tools to read, process and save the images. The training of the models and the importation of the pre-trained models was the easiest part of the project, mostly because I had already had some previous experiences with it. Also, confirming what has been said in several articles (for example, [25]), the most time-consuming part of the project was the data manipulation and processing and not the models training.

Another challenging part of the project was the data augmentation because it used several concepts that I was only familiar with the theory but not the practice, so I had to spend a time trying and researching to obtain the results. Also, most used of data augmentation generates the data in batches used directly in the training, but in my situation, I had to generate the bottleneck features first, so it was a different process to generate the augmented images and save them.

For me, the biggest and most important conclusion of the project is that transfer learning can be used as an amazing tool for situation where we have few data, especially with the images represents something that was represented in the dataset used for the transfer learning (in my situation, for example, I classified cats, which are several categories in the ImageNet dataset). Data augmentation also increased the performance and should be implemented in those situations, but in general, transfer learning is a very important technique and should be known and trained by all deep learning practitioners (even if you have lots of data, transfer learning can be used to initiate the models weight).

Improvement

There are several improvements and additions that can be done to this project. The first one was to use more data. The first possibility is to use more data for the same two cats: this can give us more confidence in the project results (as we would have more data for validation/test) and could improve the model performance, as we would have more data for the model to train.

The other option is to get more data from another cat and train a model to predict which cat was the correct one. This could be important to see how more difficult a 3-class problem is instead of a binary one and I'm curious to see how the performance would be using the same set of models and tools. The third option is to obtain more data from several other cats and train the model to recognize three options: Kuki, Gepetto or other cat. This also can be extended further to recognize cats from non-cats, so the model would have 4 outputs: not a cat, Kuki, Gepetto or other cats.

Another interesting option for further projects was to apply the same set of tools and model to another dataset, similar in size, but representing another category, like two dogs or two humans. In that case, it would be nice to see if the models had the same prevalence (NasNet better than Inception and Inception better than the Xception) or if one of those two models is better suited for dogs and another for humans.

Finally, another option to take this project further is to take the final model trained here and deploy it online, so we could send the picture to an API and receive as response what cat it is. There is several ways to do it, but one option could be using Flask, similar to what has been done on [26].

References

[1] - https://towardsdatascience.com/7-practical-deep-learning-tips-97a9f514100e

[2] - https://hackernoon.com/%EF%B8%8F-big-challenge-in-deep-learning-training-data-31a88b97b282

[3] - https://www.theverge.com/2017/11/1/16589246/machine-learning-data-science-dirty-data-kaggle-survey-2017

[4] - https://blogs.nvidia.com/blog/2016/01/12/accelerating-ai-artificial-intelligence-gpus/

[5] - https://indatalabs.com/blog/data-science/applications-computer-vision-across-industries

[6] - https://keras.io/applications/#usage-examples-for-image-classification-models

[7] - https://arxiv.org/abs/1707.07012

[8] - https://ai.googleblog.com/2017/11/automl-for-large-scale-image.html

[9] - https://towardsdatascience.com/everything-you-need-to-know-about-automl-and-neural-architecture-search-8db1863682bf

[10] - https://arxiv.org/abs/1602.07261

[11] - https://ai.googleblog.com/2016/08/improving-inception-and-image.html

[12] - http://image-net.org/challenges/LSVRC/2012/

[13] - https://arxiv.org/abs/1610.02357

[14] - https://hackernoon.com/interview-with-the-creator-of-keras-ai-researcher-fran%C3%A7ois-chollet-823cf1099b7c

[15] - https://stackoverflow.com/questions/54442808/python-cv2-imshow-loading-arrays-as-bgr

[16] - https://www.quora.com/Why-are-image-pixel-values-zero-centered-by-dividing-by-255-before-passing-them-as-input-to-feedforward-neural-network-or-convolutional-NN

[17] - http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

[18] - https://ai.stackexchange.com/questions/3089/what-are-bottleneck-features

[19] - https://arxiv.org/abs/1412.6980

[20] - https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[21] - https://keras.io/optimizers/

[22] - https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a

[23] - https://medium.com/nanonets/how-to-use-deep-learning-when-you-have-limited-data-part-2-data-augmentation-c26971dc8ced

[24] - https://keras.io/preprocessing/image/#imagedatagenerator-class

[25] - http://veekaybee.github.io/2019/02/13/data-science-is-different/

[26] - https://hackernoon.com/deploy-a-machine-learning-model-using-flask-da580f84e60c