Dog Breed Classifier

Jose Vaides

August 2020

Abstract

In this paper, the capstone project report for the Machine Learning Engineer Nanodegree from Udacity is presented. In this project, an application to classify pictures of dogs into their corresponding breed was developed, using Pytorch to build two models: a model from scratch, and a pre-trained model, using convolutional neural networks in both cases. The accuracy for the generated models was calculated and compared to a similar benchmark model.

1. Definition

1.1. Overview

For most human beings, the ability to perceive objects using our eyes is present from the moment we are born. This sense, which was developed over millions of years of evolution, allows us to identify objects at a large distance as well as obtain vast information about the world around us.

However, for computers, it is not yet possible to process "vision" in the same way as human beings do. **Computer Vision**, is a field of artificial intelligence dedicated to studying algorithms that allow computers to process visual information contained in a digital form, such as images and videos, and was born out of the interest to enable computers to understand the visual world as humans do.

In this paper, we explore building an Image Classification model that can be trained using sample images, to teach it how to classify images of dogs into their corresponding breeds. To train the model, we use the dataset provided in the Machine Learning Engineer nanodegree from Udacity, which contains a collection of dog images organized into a folder structure with images grouped in folders for each breed.

The model is built using a neural network with convolutional layers used to extract features from each of the images, and a fully connected layer to perform the classification.

1.2. Problem Statement

The task of image classification is one of the many use cases where machine learning has been widely used in recent times. Generating a set of rules that can adapt to pictures of different dog breeds or the same breed at different angles, color, scales and positions, would be extremely complex. Approaching this problem with a hard coded or rules based system would be very difficult, so it's a great example of where an algorithm that learns from examples can be used.

In this project, the problem being addressed is that of finding out what is the breed of a dog, by providing an application with a picture of the dog. Although the use case might seem trivial, the underlying technology could be used for more significant use cases, such as processing medical images to detect diseases (i.e. classifying images into different disease categories).

1.3. Metrics

The benchmark metric to be used for comparison will be accuracy defined as:

$$\boldsymbol{Accuracy} = \left(\frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predicions}\right)$$

- *https://developers.google.com/machine-learning/crash-course/classification/accuracy*

since the benchmark model selected (described in the sections below) has provided this metric as evaluation of their results.

The metric **F1 Score**, defined as:

$$\boldsymbol{F1\ Score} = \left(2 * \frac{precission * recall}{precission + recall}\right)$$

- *Scikit-learn.org*

will also be calculated (although not used as a benchmark) to evaluate the performance of the generated model (only used for the final pre-trained model), since the system will be classifying multiple classes which have a different number of test images per class, and we are mainly interested in finding out how many of the images are classified correctly.

Other metrics such as recall, and precision will also be calculated for the final model generated, but will not be the focus when evaluating the model.

2. Analysis

2.1. Data Exploration

The dataset to be used in the development of the Machine Learning Model will be the set of images provided by Udacity as part of the Machine Learning Engineer nanodegree. This dataset contains more than 8000 images, which are organized into a folder structure, where all the images for each dog breed are included into individual folders.

The images in the dataset are of different sizes, but will be transformed and normalized when being used as inputs for the model. The data set also includes pictures of each dog breed with different backgrounds, and different positions of the dogs, as shown in the figure below:

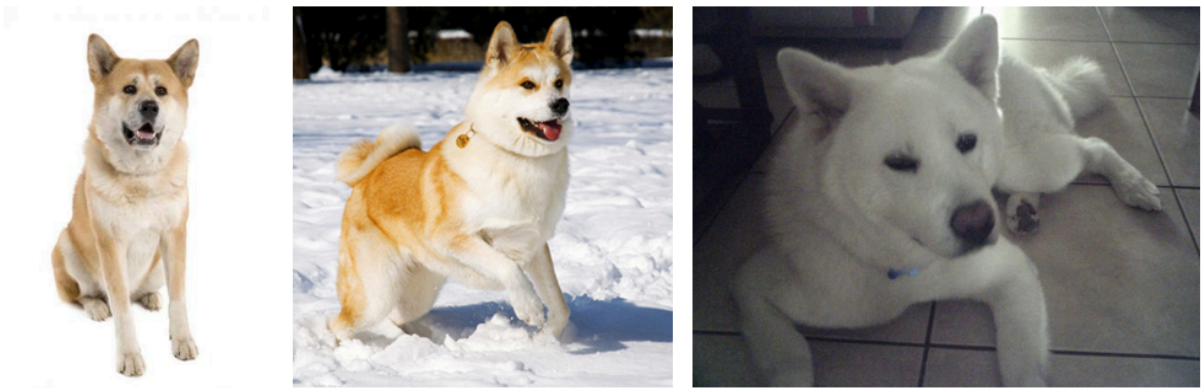
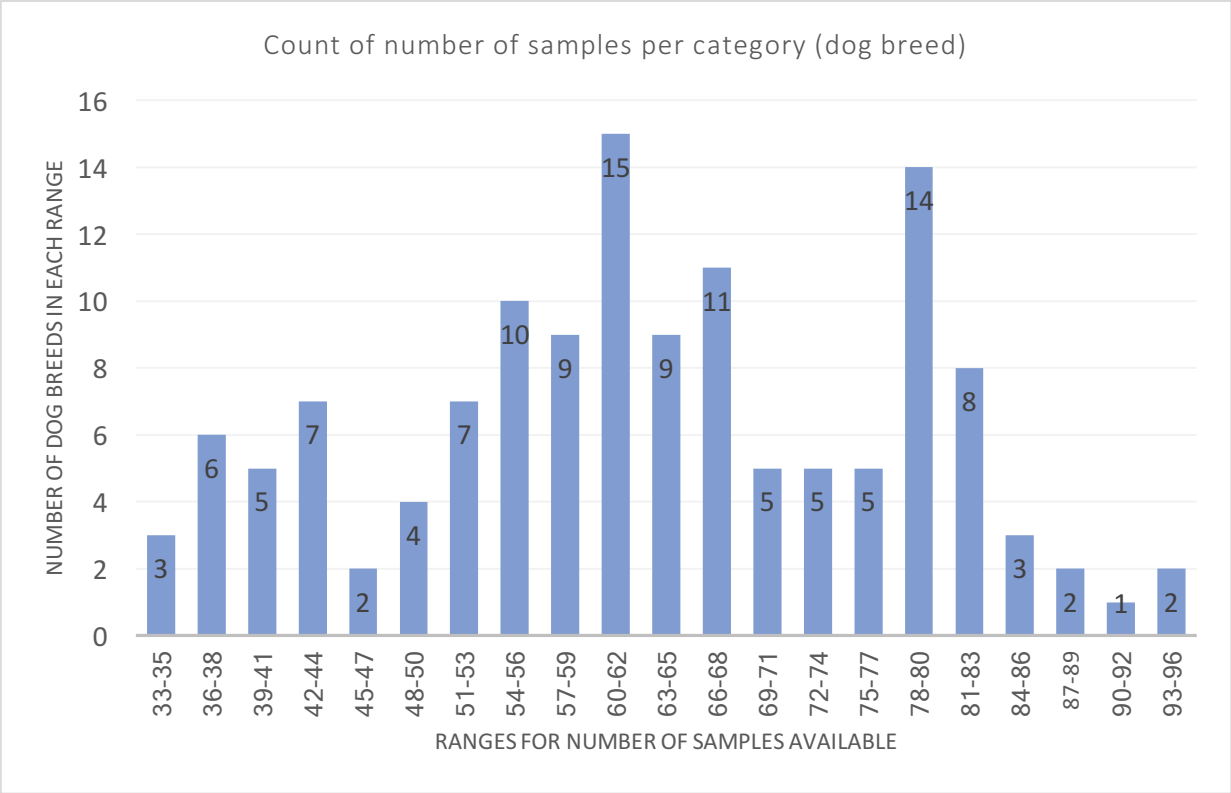


Figure 1. Akita_00255.jpg (left), Akita_00272(middle) and Akita_00279(right) - three sample images for the "Akita" breed, from the train dataset

The distribution for the training, validation and test datasets is the following:

Dataset	Number of samples	% from total
train	6680	80%
test	836	10%
valid	835	10%
Total	8351	

The full dataset contains 133 different breeds of dogs, all 3 sub-datasets (train, valid and test) contain samples for each of the breeds. The total distribution of the samples available by dog breed is the following:



The images in the dataset are of different sizes, therefore they needed to be cropped to a size of 224x224 pixels, and normalized. The Python library "torchvision.transforms" was used for this purpose, and the transforms used for the training dataset also included randomly rotating and randomly flipping the images to augment the dataset, by providing positional variations of the pictures to the model.

2.2.Algorithms and Techniques

As a part of this project, two different models were built. The first model was built "from scratch", meaning that a completely new model was developed by deciding the parameters for each part of the model. The second model built was a "pre-trained" model, which as it name implies, uses a part of the model which has been already trained, and exists as a model available within the torchvision library, and then can be customized to be adapted to the specific use case in this project (classification of dog breeds).

To build both models for the dog classifier, a convolutional neural network was trained using supervised learning.

Supervised learning is a machine learning technique, in which a model is trained by providing it multiple examples of input data, and their corresponding expected outputs (targets). In the case of the dog classifier, for each sample, the input is a 3-dimensional matrix containing the RGB values for each of the pixels in the image, and the output is the class to which the image belongs.

Convolutional neural networks are a type of artificial neural networks. They differentiate from standard multilayer perceptrons by including convolutional layers, in addition to the fully

connected layers present in standard multilayer perceptrons. Each of the convolutional layers contain "filters" which, after they have been trained, can detect patterns on an image, such as edges curves or textures.

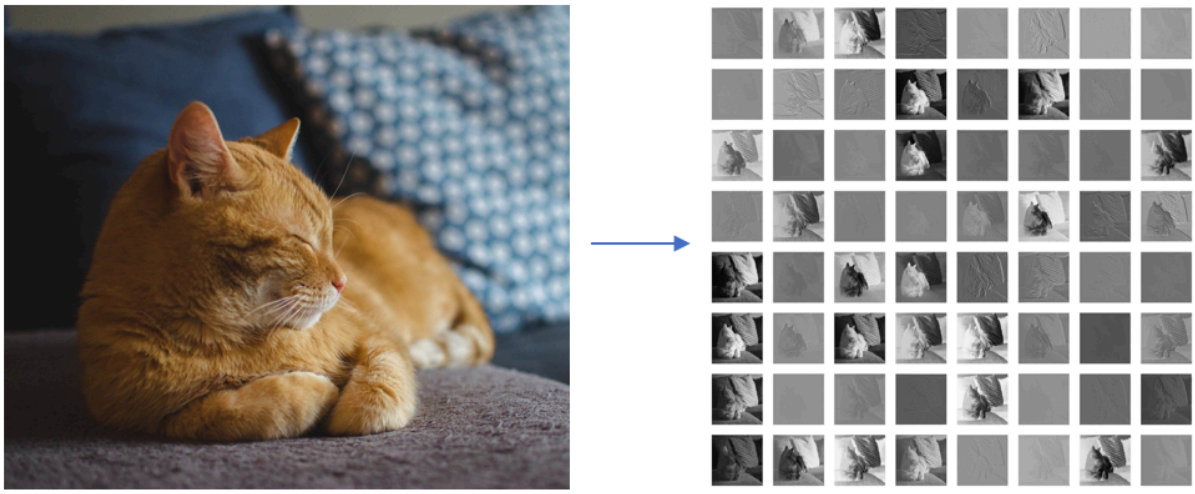


Figure 2 Filters in a convolutional layer applied to a picture of a cat

- *Visualizing Filters and Feature Maps in Convolutional Neural Networks using PyTorch*

The filters in the convolutional layers, which as mentioned above, can detect specific characteristics of an image, then become the inputs for the final fully connected layer, where classification takes place. In this sense, the convolutional layers act as "feature detectors" which generate the inputs (or features) for the fully connected layer (classifier).

The decision to use a convolutional neural network was made due to the success that this algorithm has recently achieved in image classification. While other types of neural networks, or completely different algorithms could potentially have been used, in recent times the models with the highest accuracy results have been models using convolutional neural networks.

2.3. Benchmark

The Benchmark Model used for this project was the one built by Paul Stancliffe, in his article "Udacity Dog Breed Classifier — Project Walkthrough" (Described in https://medium.com/@paul.stancliffe/udacity-dog-breed-classifier-project-walkthrough-e03c1baf5501) which was also trained with the dataset provided in the same Udacity project, and also classifies the same number of categories (dog breeds).

In the benchmark model, the accuracy reached was 8.6% on the model created from scratch, and 84.8086% on the pre-trained model.

3. Methodology

3.1. Data preprocessing

As described in previous sections, the data for the project exists as a folder structure with a folder corresponding to each dog breed, and images of the breed within the folder. The training and validation data were uploaded to a Bucket in Amazon SageMaker, since the SageMaker infrastructure was used to train and deploy the models.

The images in the dataset have different dimensions and different positions of the dogs. Since the model needs to have the same number of inputs for all training samples (in this case the same number of pixels), each of the images needs to be cropped so that they have the same size once they are entered in the model.

Additionally, the images are also randomly rotated and randomly flipped horizontally to enhance the data set, by providing multiple positional variations of the images in the dataset.

After resizing the images, and randomly adjusting their position, they are transformed into tensors and normalized, so that they can be used as inputs for the Pytorch model.

The resulting transformation sequence for the training data set is the following:

```
transforms.Compose([transforms.RandomRotation(30),
                    transforms.RandomResizedCrop(224),
                    transforms.RandomHorizontalFlip(),
                    transforms.ToTensor(),
                    transforms.Normalize([0.485, 0.456, 0.406],
                                         [0.229, 0.224, 0.225])])
```

Figure 3. Transformation sequence for the training dataset (using Torchvision's transforms library)

For the test and validation datasets the pre-processing sequence used is similar, but for these datasets the random repositioning of the images was not used. The pre-processing sequence for the test and validation datasets is the following:

```
transforms.Compose([transforms.Resize(256),
          transforms.CenterCrop(224),
          transforms.ToTensor(),
          transforms.Normalize([0.485, 0.456, 0.406],
                               [0.229, 0.224, 0.225])])
```

Figure 4. Validation and testing datasets pre-processing

3.2. Implementation

The implementation of the model is documented in detail in the Jupyter notebook submitted together with this report. However, in this section an overview is presented of how the solution was implemented.

For the model built from scratch, it was decided from the start to use convolutional neural networks to build the model. However, it was still necessary to decide the architecture of the model, as well as the hyper parameters used to train it.

To design the network, the general the idea was to have a set of convolutional layers to extract features and then a fully connected layer at the end to perform the classification. Pooling layers were used after each convolutional layer to down sample the features detected. Padding and stride were not used.

The steps used to get the final architecture were the following:

1. Attempt to build a model using a simple architecture, to have a starting point and a model to compare performance
2. Train the model with the full data set
3. Measure performance metrics (for the benchmark model only accuracy was measured)
4. In case the accuracy is not higher than 10% (this was a requirement for the project) adjust the hyper parameters (mainly the number of epochs and learning rate were adjusted).
5. If the accuracy is still not high enough after some attempts, adjust the model architecture.

3.2.1 Training the model using Amazon SageMaker

Amazon SageMaker was used during the training of the models, in order to use a GPU. SageMaker requires a director with name "train" to be created for the training scripts. Within this directory the following files were created for the model built from scratch:

- **model_scratch.py:** This file contains the architecture of the model, which defines each of the functions used in its layers, as well as it's forward function.
- **train_scratch.py:** This file contains methods specifying where to find the input data in Amazon S3, and methods indicating how the inputs should be pre-processed and the output that should be returned.

Similarly, for the pre-trained model, a train directory is required to define the methods used when training the model. In this case, the files "**model.py**" and "**train.py**" were created, with the same functions as with the model from scratch. The methods in these are also used when measuring the accuracy of the model from the Jupyter notebook.

3.2.2 Deploying pre-trained model to be used by web application

For the pre-trained model deployed to be used by a web application, an additional directory with name "serve" was required. Within this folder, the following files needed to be created:

- **model.py:** Like the file in the "train" directory, this file also specifies the model architecture and its methods.
- **Predict.py**: This file specifies how the data needs to be processed when the deployed web application sends a request to make a prediction for an image. While the scripts in the "train" directory process a whole folder of images, for the model used in the deployed web application, the input is the URL of a single image that gets uploaded by the web application, and which needs to be classified.

3.2.3 Deployed web application

The interface provided to the end users for obtaining classifications of images is a website deployed using Amazon's web services. The steps used for the website to generate a classification are the following:

1. The user selects an image from the local directory, using the "CHOOSE A PICTURE" button.
2. The user clicks on the button "UPLOAD FILE". This button will upload the picture selected by the user to a directory in Amazon S3. The URL of the file uploaded is stored in a hidden form element in the .html file. After the file is uploaded a success message is shown to the user.
3. After uploading the file, the user clicks on the "GET RESULTS" button, which sends the URL of the picture uploaded as input. Then the script deployed in SageMaker gets the file in the input URL, peforms the corresponding transforms on the image and uses it as input for the deployed neural network model in SageMaker.
4. The "predict" script returns a dictionary (later turned into json by the javascript code), which contains the value classified by the model (an integer, the index of the dog breed), the name of the breed classified, a URL for an image corresponding to the breed classified (to be displayed to the user), and the URL of the picture uploaded by the user (so that it can be also displayed to the user).

The following is a screenshot of the deployed application:

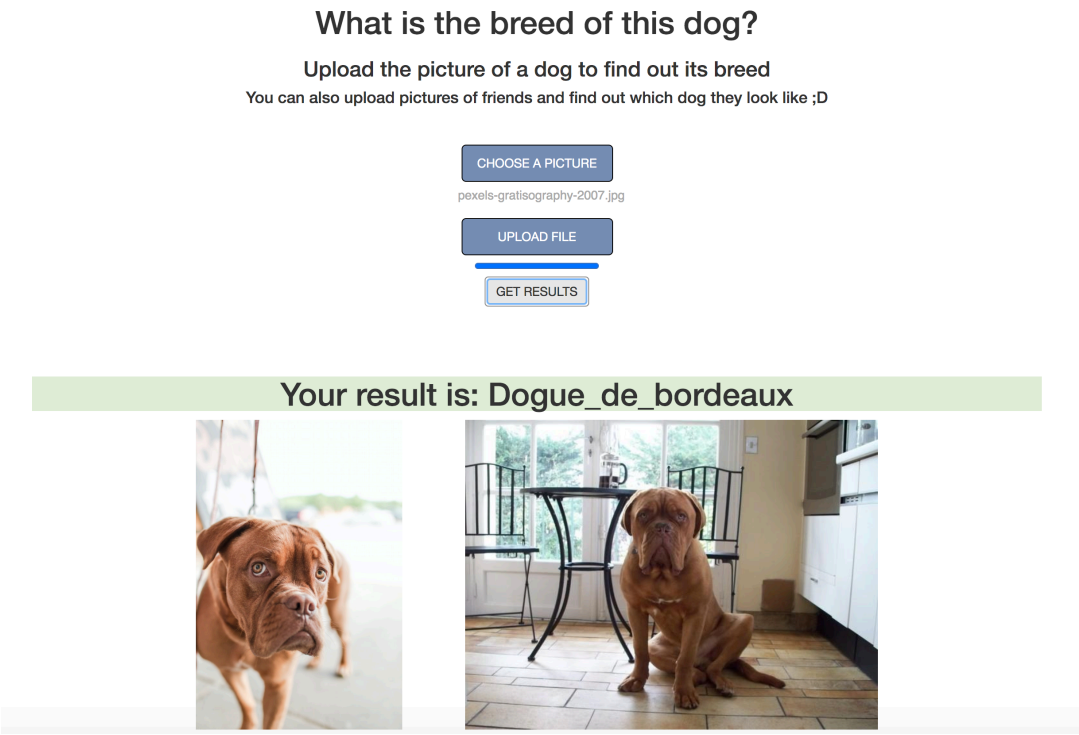


Figure 5. Screenshot of the deployed web application

The files used for the website are the following:

- **index.html:** A simple html file which contains the mark-up and styles for user interface with the buttons that allow the user to upload a picture and receive the resulting classification.

- **js/file_upload.js:** This file contains the code used to upload the picture file selected by the user to Amazon S3. It also contains the code which generates the URL of the file uploaded and saves it as a hidden form input to be submitted later.
- **js/submit_file.js:** This file contains the code which submits the URL of the file uploaded by the user to the SageMaker model, and which receives the response sent by the Sagemaker script to display the results it to the user.
- **js/custom_file_upload_button.js:** This file was used to customize the buttons for "CHOOSE A PICTURE" and "UPLOAD FILE".

The files were uploaded to an Amazon S3 bucket where they are hosted as a static website.

4. Results

4.1.1 Results for the models trained from scratch

The first design used had the following configuration:

```
* conv_layer1: conv2d(3,32, kernel_size=5) -> MaxPool(2) -> Relu
* conv_layer2: conv2d(32, 64, kernel_size=5) -> MaxPool(2) -> Relu
* Fully connected layer: linear -> log_softmax
```

Figure 6. Initial configuration for model built from scratch

This configuration was trained first with 10 epochs, where the accuracy achieved was less than 10%. On a second attempt, 20 epochs were used, when the accuracy reached was 11.36%. Athough this result fulfilled the requirements of the project, a third attempt was made to further improve the accuracy using another model, with the following architecture:

```
* conv_layer1: conv2d(3,32,kernel_size=5) -> MaxPool(2) -> Relu
* conv_layer2: conv2d(32,64,kernel_size=5) -> MaxPool(2) -> Relu
* conv_layer3: conv2d(64,128,kernel_size=5) -> MaxPool(2) -> Relu
* Fully connected layer: linear -> Relu -> Dropout -> linear -> log_softmax
```

Figure 7. Final architecture for model built from scratch

This model was trained with 40 epochs, and additionally, a scheduler was implemented to adjust the learning rate while running the training procedure. The scheduler started with a learning rate of 0.05, which was adjusted by reducing it by half whenever 2 consecutive epochs were executed without the training loss being reduced. This is the final configuration used for the model, and the one that reached the highest accuracy for the models trained from scratch.

4.2 Results for the pre-trained model

For the pre-trained model the architecture consisted on using the VGG16 model available in the torchvision library. The Convolutional layers form this model were used as feature extractors, and then a customized classifier was implemented on the fully connected layers of the model, to adjust it to the 133 dog categories in this project. The architecture is shown in the figure below:

- VGG-16 convolutional layers
- classifier layer 1: linear layer (inputs: 25088, outputs:5000, relu activation function)
- classifier layer 2: Dropout layer (dropout rate: 0.5)
- classifier layer 3: linear layer (inputs: 5000, outputs: 133)
- output layer: LogSoftmax

Figure 8. Architecture of the pre-trained model with VGG16

This model was trained using 25 epochs, and a scheduler for the learning rate was also used. After training with these parameters, the model achieved an accuracy of 84.21%. However, it could be noticed that the scheduler was only adjusted once in the 25 epochs, therefore the accuracy could potentially be improved by running this setting on a higher number of epochs. This was not done in this project since the 60% accuracy required in the project was considerably surpassed by the 84.21% accuracy achieved with this model. The results are also comparable to the 84.8086% achieved by the benchmark model.

In addition to accuracy, recall, precision and the F1-score were also calculated for the pre-trained model. The results are shown in the table below:

Metrics Report	
accuracy	84.2105%
Micro metrics	
Recall	84.2105%
Precision	84.2105%
F1 score	84.2105%
Macro metrics	
Recall	82.7148%
Precision	85.4121%
F1 score	82.3232%
Weighted metrics	
Recall	84.2105%
Precision	86.5531%
F1 score	83.8419%

5. Conclusions

During the development of the model built from scratch, on the first attempts, the main parameters being changed were the number of epochs and the learning rate. It was difficult to find a specific learning rate that could be trained "fast" enough. Using a larger learning rate was usually causing the loss of the model to quickly decrease initially, but then to increase again after a few epochs had been completed. This was solved after implementing a scheduler which adjusted the learning rate along with the training of the model, and illustrates that while the model architecture and hyper parameters are critical to obtain a model with good performance, the way in which the model is trained and the training script also have a very significant impact in the results.

Also on the models from scratch, it could be observed that the simpler models with less convolutional layers were intermittently increasing and decreasing the training loss on the last epochs of the training. When adding more convolutional layers, the models could reach a higher accuracy. However, a larger number of epochs were also required to reach this higher accuracy.

In this specific project, it was also possible to explore not only how to develop a machine learning model, but also how to deploy it to a production environment where it can be used by a web application. It was interesting to experience that the effort required to configure the files using Amazon's web services also require a considerable amount of time and effort, and that there are many possibilities that can help during the development of machine learning models (like the availability of GPUs for training).

6. Improvements

For the final model, the number of epochs used for training was 25, since this amount, in combination with the classifier architecture and the scheduler for the learning rate could achieve an accuracy that was above the level required for this project. Further attempts were not done, since training a complex model using SageMaker also generates a cost, and in this case the objective was already reached. However, the training loss was still decreasing steadily with this setup, therefore it could be that a higher accuracy could be achieved by training the model with a higher number of epochs.

Also in this project, some of the parameters of the convolutional layers, such as the dimensions of the pooling layers, or using stride and padding were not tuned to reach a higher accuracy. It could be that by modifying / including some of these parameters a higher accuracy could be reached.

7. References

SAS, Computer Vision and why it matters, URL: https://www.sas.com/en_us/insights/analytics/computer-vision.html#:~:text=Computer%20vision%20is%20a%20field,to%20what%20they%20%E2%80%9Csee.%E2%80%9D

Neurohive.io, VGG16 – Convolutional Network for Classification and Detection, URL: https://neurohive.io/en/popular-networks/vgg16/

Simonyan, K., Zisserman, A., Very Deep Convolutional Networks for Large-Scale Image Recognition, URL: https://arxiv.org/abs/1409.1556

scikit-learn developers (BSD License), sklearn.metrics.f1_score, URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html

Udacity, Machine Learning Engineer nanodegree – Dog Breed Classifier, URL: https://classroom.udacity.com/nanodegrees/nd009t/parts/2f120d8a-e90a-4bc0-9f4e-43c71c504879/modules/2c37ba18-d9dc-4a94-abb9-066216ccace1/lessons/4f0118c0-20fc-482a-81d6-b27507355985/concepts/65160313-7054-4ffb-8263-793e2a166d69

Marr, D. (2010). Vision: A computational investigation into the human representation and processing of visual information. Cambridge, MA: MIT Press.

Demush, R. (2019), A Brief History of Computer Vision (and Convolutional Neural Networks), URL: https://hackernoon.com/a-brief-history-of-computer-vision-and-convolutional-neural-networks-8fe8aacc79f3

Stancliffe, P. (2019) Udacity Dog Breed Classifier — Project Walkthrough, URL: https://medium.com/@paul.stancliffe/udacity-dog-breed-classifier-project-walkthrough-e03c1baf5501, github: https://github.com/paulstancliffe/Dog-Breed-Classifier

Ranjan S. (2020), Visualizing Filters and Feature Maps in Convolutional Neural Networks using PyTorch, URL: https://debuggercafe.com/visualizing-filters-and-feature-maps-in-convolutional-neural-networks-using-pytorch/

Udacity (2016), Machine Learning Engineer Nanodegree, Capstone Project, URL: https://github.com/udacity/machine-learning/blob/master/projects/capstone/capstone_report_template.md