

# Machine Learning Engineer Nanodegree

## Capstone Project

Jorge Ramírez Carrasco

### Dog Breed Classifier (CNN)

#### Definition

This report contains the capstone project of the Machine Learning Engineer Nanodegree about creating a dog breed classifier using a Convolutional Neural Network.

#### Project Overview

This capstone project has the objective of creating an application able to categorize dogs estimating the closer canine's breed using AWS capabilities. This problem is one of the proposed ones from Udacity's proposals. Although the project consists on building the whole application, the core component is to develop a classifier able to categorize dog's breed using a dog dataset containing different dog's breed images.

The project is composed on different components that will be developed in the following steps:

- Model selection: Analysis for selecting the best model to perform the classification task.
- Model deployment & testing: Training the selected model from the benchmark analysis using AWS SageMaker capabilities and deploying it to include the model in a complete application.
- Inference deployment: Set up an endpoint using the model for inference.
- App deployment: Final step to create the whole application including the web app, the API gateway and the lambda function.

#### Problem Statement

The problem of the proposal is composed by two parts:

- First, the idea is to detect if in an input image there is a dog or a human and perform an image estimation about the closer canine's breed. Given an image of a dog, the algorithm will identify an estimate of the canine's breed and if supplied an image of a human, the code will identify the resembling dog breed.

From the initial dataset a subset of images will be stored for testing in order to measure the estimations' quality and with the training subset the different models will be trained and tested.

- Second part of the project once that we have selected the best candidate is to set up a whole SageMaker environment to train, deploy and create the whole application.

## Metrics

Model's performance will be compared using a classical metric for multiclass image classifier like accuracy. It is also the proposed metric in the project documentation. Accuracy's formula is:

$$\text{Accuracy} = \frac{\text{Number of Correct predictions}}{\text{Total number of predictions made}}$$

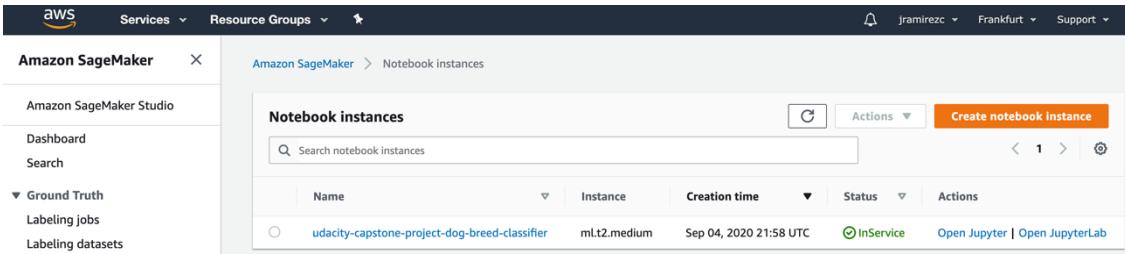
It works well only in cases where the classes are not unbalance where precision or recall will be a better metric.

## Analysis

In the following section we will take a deeper look about the data and the models that have been used during the project.

### Data Exploration

The dataset for this problem is composed of different images in order to start working first we need to set up an environment. Using Amazon SageMaker Notebook instances, we can access to a JupyterLab with preconfigured environments ready with many useful libraries for Machine Learning.



The screenshot shows the AWS SageMaker console. On the left, there is a sidebar with options: 'Amazon SageMaker Studio', 'Dashboard', 'Search', 'Ground Truth', 'Labeling jobs', and 'Labeling datasets'. The main area is titled 'Notebook instances' and shows a table with one row. The table has columns for 'Name', 'Instance', 'Creation time', 'Status', and 'Actions'. The single entry is 'udacity-capstone-project-dog-breed-classifier' (ml.t2.medium, Sep 04, 2020 21:58 UTC, InService, Open Jupyter | Open JupyterLab).

In this case I have selected pytorch\_p36 as environment as most of my models will use pytorch library.

Regarding the datasets to use, a detailed description about the two datasets is:

- Dog Dataset: It contains 8351 dog images structured in 133 folders by dog's breed class. The dataset is well distributed between the 133 classes although some dog breeds contain more samples than others. The sizes are different with a standard resolution. Downloaded from Udacity AWS S3. The purpose of the dataset is to train and test a convolutional neural network that classifies dog's breed.

- Human Dataset: It contains 13233 human images structured in folders by celebrities' name. The sizes are different with a standard resolution. Downloaded from Udacity AWS S3. The purpose of the dataset is to detect human faces and test the dog's breed convolutional neural network.

The required steps to download the datasets into the environment are shown in these code lines:

```
%mkdir ..data
!wget -O ..data/dogImages.zip https://s3-us-west-1.amazonaws.com/udacity-aind/dog-
project/dogImages.zip
!wget -O ..data/lfw.tgz http://vis-www.cs.umass.edu/lfw/lfw.tgz
!tar -zxf ..data/lfw.tgz -C ..data/
!unzip ..data/dogImages.zip -d ..data/
```

## Exploratory Visualization

The exploratory analysis in this case has been performed checking the images of the different dogs:



It has been observed that the images have different sizes and features so the preprocessing will be a very important step in our model training phase.

## Algorithms and Techniques

The initial idea to solve the problem is to split the classification problem in two phases.

1. A detector for checking if there is a human or a dog in the image.
2. A classifier for categorize the closer dog's breed in the image.

Regarding the detectors we will need a dog and a human detector.

The human detector has been developed using OpenCV's implementation of Haar feature-based cascade classifiers. OpenCV provides many pre-trained face detectors, stored as XML files the detector have been downloaded from OpenCV git repository and stored it in the haarcascades directory. The final detector code including a test is shown in the following snippet:

```

● ● ●

def face_detector(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    return len(faces) > 0

human_files_short = human_files[:100]
dog_files_short = dog_files[:100]

print("Detecting human faces in human_files_short")
human_results_short = [ face_detector(img_path) for img_path in human_files_short]
print(f" Detected {sum(human_results_short)} human faces")
print("Detecting human faces in dog_files_short")
dog_results_short = [ face_detector(img_path) for img_path in dog_files_short ]
print(f" Detected {sum(dog_results_short)} human faces")

-----
Output:

Detecting human faces in human_files_short
Detected 99 human faces
Detecting human faces in dog_files_short
Detected 5 human faces

```

With respect to the dog detector the idea is to use a pretrained version of VGG-16 model. We will apply the weights of a training using ImageNet, a very large datasets with 1000 categories, selecting just the corresponding for dogs for detecting if there is dog in the image.

For completing the function, we need to apply some transformations to the image like resize for what VGG-16 model is expecting and applying the normalization to the three channels.

```

● ● ●

from PIL import Image, ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
import torch
import torchvision.models as models
import torchvision.transforms as transforms

VGG16 = models.vgg16(pretrained=True)

def VGG16_predict(img_path):

    transform = transforms.Compose([
        transforms.Resize(size=(256,256)),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])

    img = Image.open(img_path)
    output = VGG16(transform(img).unsqueeze(0))
    return torch.max(output,1)[1].item()

### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    label = VGG16_predict(img_path)
    if label > 150 and label < 269:
        return True
    else:
        return False

human_results_short = [ dog_detector(img_path) for img_path in human_files_short]
dog_results_short = [ dog_detector(img_path) for img_path in dog_files_short]
-----
Output:

Detecting human faces in human_files_short
Detected 0 dogs
Detecting human faces in dog_files_short
Detected 99 dogs

```

Related to the classifier the idea is to test different algorithms like a CNN trained from scratch, a CNN using transfer learning, Support Vector Machines classifier and Logistic Regression.

In the case of the CNN we need to set up some transformers for the data loaders with the images to feed the networks. With respect to the SVM and Logistic Regression the approach will be to transform the images into pandas DataFrames to train and test the models.

## Benchmark

The convolutional neural networks and the classical classification methods like logistic regression and support vector machines will be compared using a common test dataset using the metric of the accuracy to determine the best model for solving the problem.

The project requirements have set up a minimum value of 10% accuracy for the CNN build from scratch.

## **Methodology**

In the following section we will go deeper into the methodology of the project and the different steps.

### Data Preprocessing

The convolutional neural network needs different preprocessing steps for the three subsets of the dataset: train, validation and test. For each of the data loaders a different transformer has been applied, there is one common preprocessing for all subsets and some specific steps just for training. In more detail the preprocessing steps are:

1. The image is resized to 224x244 as VGG16 initial layer.
2. In the training transformations a RandomHorizontalFlip, RandomVerticalFlip and RandomRotation are applied to make data augmentation. These steps will add randomness and decrease overfitting while training the model.
3. The image is converted into a tensor
4. The values are normalized following custom datasets pytorch documentation.

```
● ● ●

import os
from torchvision import datasets

data_dir = '/data/dog_images/'
train_dir = os.path.join(data_dir, 'train/')
valid_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')

transform_train = transforms.Compose(
    [transforms.Resize(size=(224,224)), transforms.RandomHorizontalFlip(),
     transforms.RandomVerticalFlip(), transforms.RandomRotation(10),
     transforms.ToTensor(), transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

transform_valid = transforms.Compose(
    [transforms.Resize(size=(224,224)), transforms.ToTensor(), transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

transform_test = transforms.Compose(
    [transforms.Resize(size=(224,224)), transforms.ToTensor(), transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

train_set = datasets.ImageFolder(train_dir, transform=transform_train)
valid_set = datasets.ImageFolder(valid_dir, transform=transform_valid)
test_set = datasets.ImageFolder(test_dir, transform=transform_test)

train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size,
                                           shuffle=True, num_workers=num_workers)

valid_loader = torch.utils.data.DataLoader(valid_set, batch_size=batch_size,
                                           shuffle=False, num_workers=num_workers)

test_loader = torch.utils.data.DataLoader(test_set, batch_size=batch_size,
                                         shuffle=False, num_workers=num_workers)
```

Regarding the classical classification models like SVM and Logistic Regression we need to include some preprocessing in order to modify our dataset to work from a pandas DataFrame instead of images. The idea is to create a DataFrame with the pixel information in the image on each column.

```
● ● ●

from sklearn import datasets, svm, metrics
import pandas as pd

def create_df_from_dir(ds_dir):
    images = []
    data = []
    transform = transforms.Compose(
        [transforms.Resize(size=(64, 64)),
         transforms.Grayscale(num_output_channels = 1),
         transforms.ToTensor(),
         transforms.Normalize(mean=[0.5],
                             std=[0.5]))]
    for path, subdirs, files in os.walk(ds_dir):
        files = [path+"/"+file for file in files]
        images += files
    for img in images:
        img = Image.open(img)
        data.append(transform(img).data.numpy()[0,:,:].flatten() )
    labels = [int(path.split("/")[-2].split(".")[0]) for path in images]
    df_dict = {'data':data, 'target':labels}
    return pd.DataFrame(df_dict)

train_df = create_df_from_dir(train_dir)
val_df = create_df_from_dir(valid_dir)
test_df = create_df_from_dir(test_dir)

X_train = train_df.data.apply(pd.Series)
y_train = train_df['target']
X_test = test_df.data.apply(pd.Series)
y_test = test_df['target']
```

## Implementation

Once that we have executed the preprocessing steps, we can start working with the model development. We will start implementing the CNN from Scratch, the idea was to replicate VGG-16 architecture in a simplified way removing convolution layers and keeping the 3 layers for the fully connected.

The convolution layers will increment the number of features using a kernel size of 5 and padding 1. After each convolution there is a relu and maxpool layer with a kernel of size 2 and stride 2 reducing to the half.

After the convolution layers there is a flatten layer that creates a vector of size  $256 \times 26 \times 26 = 173056$  that links with the fully connected of 3 layers reducing the number of features until the final number of classes and applying dropout to reduce overfitting between each of them.

```

● ● ●

import torch.nn.functional as F

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        num_classes = len(train_set.classes)
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=5, padding=1)
        self.conv2 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=5, padding=1)
        self.conv3 = nn.Conv2d(in_channels=128, out_channels=256, kernel_size=5, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        # [batch = batch_size, channel=256, height=26, width=26]
        self.fc1 = nn.Linear(in_features=256 * 26 * 26, out_features=512)
        self.fc2 = nn.Linear(in_features=512, out_features=256)
        self.fc3 = nn.Linear(in_features=256, out_features=num_classes)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 256 * 26 * 26)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return x

# instantiate the CNN
model_scratch = Net()

```

Once that the data loaders and the model architecture are defined, we have to implement the training and testing methods.

```

● ● ●

def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    valid_loss_min = np.Inf

    for epoch in range(1, n_epochs+1):
        train_loss = 0.0
        valid_loss = 0.0
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            optimizer.zero_grad()
            outputs = model(data)
            loss = criterion(outputs, target)
            loss.backward()
            optimizer.step()
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))

        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            if use_cuda:
                data, target = data.cuda(), target.cuda()
                with torch.no_grad():
                    outputs = model(data)
                    loss = criterion(outputs, target)
                    valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

    return model

model_scratch = train(30, loaders_scratch, model_scratch, optimizer_scratch,
                     criterion_scratch, use_cuda, 'model_scratch.pt')

```

In the train method we apply backpropagation on each batch using the optimizer and the criterion specified. In this case I have used CrossEntropy and Stochastic Gradient Descent respectively.

In the test method we change the model to evaluation mode and just calculate the difference with unseen data to compute the final accuracy.

```

● ● ●

def test(loaders, model, criterion, use_cuda):

    test_loss = 0.
    correct = 0.
    total = 0.

    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['test']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
        total += data.size(0)

    print('Test Loss: {:.6f}\n'.format(test_loss))

    print('\nTest Accuracy: {}% ({}/{})'.format(
        100. * correct / total, correct, total))

# call test function
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)

```

Second, I have implemented another CNN, in this case, using Transfer Learning. The idea is to download a pretrained model, freeze the parameters already trained and add some layers at the end to match with our purpose and just retrain those layers.

I have chosen the model resnet50 for the transfer learning and I have added 3 linear layers to solve the dog's breed. I have reused the train and test method and I have used the same optimizer and criterion from scratch model.

```

● ● ●

import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture
model_transfer = models.resnet50(pretrained=True)

# Freeze parameters of the model to avoid back propagation
for param in model_transfer.parameters():
    param.requires_grad = False

num_classes = len(train_set.classes)

model_transfer.fc = nn.Sequential(nn.Linear(in_features=2048, out_features=512),
                                  nn.Linear(in_features=512, out_features=256),
                                  nn.Linear(in_features=256, out_features=num_classes))

if use_cuda:
    model_transfer = model_transfer.cuda()

```

The performance for this model has improved a lot the previous values for accuracy and loss in the test set.

As last step of the implementation I have tried as well with classical machine learning classification algorithms like Support Vector Machines and Logistic Regression.

```

● ● ●

from sklearn import svm, metrics
# Create a classifier: a support vector machine classifier
classifier = svm.SVC(gamma=0.001)
classifier.fit(X_train,y_train)
y_pred = classifier.predict(X_test)

from sklearn.linear_model import LogisticRegression
# Create a classifier: a support vector machine classifier
classifier = LogisticRegression()
classifier.fit(X_train,y_train)
y_pred = classifier.predict(X_test)

```

## Refinement

Most of the improvement process for the algorithms have been adjust the transformers in the preprocessing steps and modify the model architecture and optimizer method.

I found some problems to see improvements in the network during training while using lower values in the learning rate, it could take 3 or 5 epochs to see some improvement in the validation set and the loss was not reducing enough. The idea of changing to the Stochastic Gradient Descent and the value of 0.05 for the learning rate was a good combination to start seeing significant improvements after each training epoch.

Then, I found some issues in the accuracy of the network. Using data augmentation in the transformer of the training data loader was a good addition to improve the model performance.

## **Results**

In the following section we will analyze the results of the implementation process.

### Model Evaluation and Validation

Regarding the detectors the dog and the human detectors have shown a performance of 99/100 detections in the desired label.

With respect the classifiers, we will compare the different models' performance to select the best candidate.

	<b>Test Loss</b>	<b>Test Accuracy</b>
<b>Scratch CNN</b>	3.49	16%
<b>Transfer Learning CNN</b>	0.68	78%
<b>SVM</b>		6.10%
<b>Logistic Regression</b>		2.87%

We can conclude that the model with best performance is the CNN using Transfer Learning because in the test subset composed of 836 unseen images of dogs it has achieved a 78% of accuracy. This model is the selected one for next steps mainly

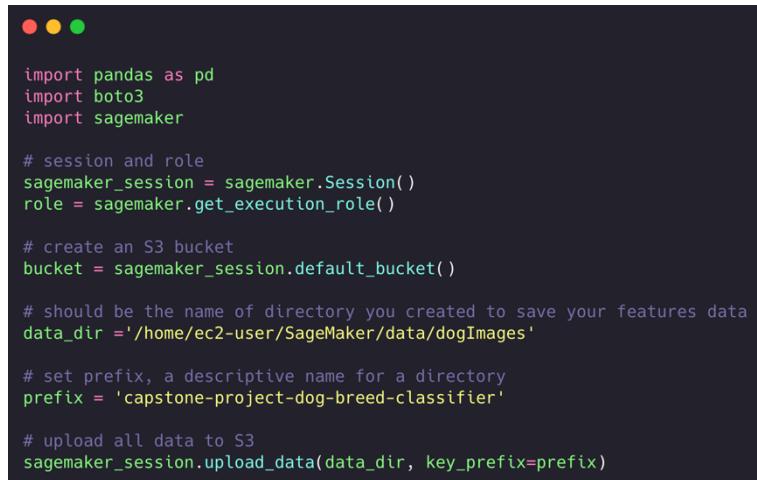
because the performance is very far from the other candidates and it has shown a good performance with raw images as well. Moreover, we will just select this algorithm but we will use SageMaker capabilities to make a new training trying to improve even this performance.

### Justification

After the training with AWS the accuracy for the CNN with transfer learning has improved until 81% for the testing dataset showing a good evolution from my initial approaches.

In order to obtain this result it was required to train the selected model from the models analysis using AWS SageMaker training capabilities.

The first step to work with the model in AWS is to upload the data to Amazon S3 so it is easy to read from SageMaker training capabilities.



```
● ● ●

import pandas as pd
import boto3
import sagemaker

# session and role
sagemaker_session = sagemaker.Session()
role = sagemaker.get_execution_role()

# create an S3 bucket
bucket = sagemaker_session.default_bucket()

# should be the name of directory you created to save your features data
data_dir = '/home/ec2-user/SageMaker/data/dogImages'

# set prefix, a descriptive name for a directory
prefix = 'capstone-project-dog-breed-classifier'

# upload all data to S3
sagemaker_session.upload_data(data_dir, key_prefix=prefix)
```

Using a SageMaker session we can upload the data and write down the bucket for future usage.

Then, we just have to set up our model code in a directory including a requirements.txt with the libraries and organizing the code to have a main method that trains the model and a model\_fn() method that will load the model.

Once that the PyTorch estimator is set up we can call the fit method adding the S3 bucket with the dataset and SageMaker will create a training job.

```

from sagemaker.pytorch import PyTorch

estimator = PyTorch(entry_point="train.py",
                    source_dir="models/cnn_pytorch",
                    role=role,
                    framework_version='0.4.0',
                    train_instance_count=1,
                    train_instance_type='ml.c4.4xlarge',
                    hyperparameters={
                        'epochs': 15
                    })

# Train your estimator on S3 training data
estimator.fit({'train': 's3://sagemaker-eu-central-1-411771656960/capstone-project-dog-breed-classifier'})

```

We can follow the training process checking the logs of the training job.

Name	Creation time	Duration	Status
sagemaker-pytorch-2020-09-11-09-11-53-314	Sep 11, 2020 09:11 UTC	5 hours	Completed
sagemaker-pytorch-2020-09-10-17-15-08-053	Sep 10, 2020 17:15 UTC	2 hours	Completed

Once that the model has been trained, we can see it in the SageMaker web portal.

Name	ARN	Creation time
sagemaker-pytorch-2020-09-11-09-11-53-314	arn:aws:sagemaker:eu-central-1:411771656960:model/sagemaker-pytorch-2020-09-11-09-11-53-314	Sep 11, 2020 14:17 UTC

In this case, in order to see our model performance, we need to recreate the test method to use the model endpoint instead of a local trained model.

```

def test_sagemaker_endpoint(loader, criterion, use_cuda):

    # monitor test loss and accuracy
    test_loss = 0.
    correct = 0.
    total = 0.

    for batch_idx, (data, target) in enumerate(loader):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = torch.from_numpy(predictor.predict(data))
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
        total += data.size(0)

    print('Test Loss: {:.6f}\n'.format(test_loss))

    print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
        100. * correct / total, correct, total))

    # call test function
    test_sagemaker_endpoint(test_loader, nn.CrossEntropyLoss(), False)

```

As we have trained the model with more epochs in this case the performance has improved.

	Test Loss	Test Accuracy
Transfer Learning CNN	0.65	81%

## **Conclusions**

With this final section I conclude this capstone project report.

### Free-Form Visualization

The visualization in this project is a task from the web app. I have created a method that use the deployed endpoint just adding as input an image as we will need something similar for the final application.

```
● ● ●

# list of class names by index, i.e. a name can be accessed like class_names[0]
class_names = [item[4:].replace("_", " ") for item in test_loader.dataset.classes]

def predict_breed_sagemaker_transfer(img_path):
    # load the image and return the predicted breed
    transform = transforms.Compose(
        [transforms.Resize(size=(224,224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                            std=[0.229, 0.224, 0.225])))

    img = Image.open(img_path)
    output = torch.from_numpy(predictor.predict(transform(img).unsqueeze(0)))
    # position 0 return value, position 1 return indices
    output_index = torch.max(output,1)[1].item()
    return class_names[output_index]

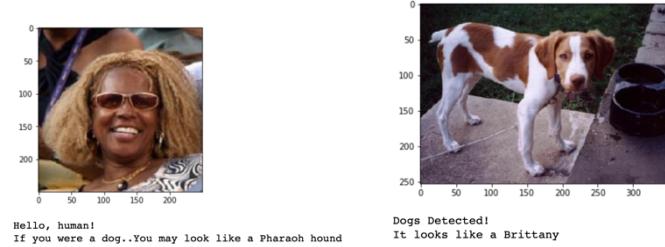
predict_breed_sagemaker_transfer('images/Labrador_retriever_06457.jpg')
```

With the previous method that we have created using the SageMaker endpoint with the deployed model we are ready to create a method that simulates the application workflow.

```
● ● ●

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()
    if dog_detector(img_path) is True:
        prediction = predict_breed_sagemaker_transfer(img_path)
        print("Dogs Detected!\nIt looks like a {}".format(prediction))
    elif face_detector(img_path) > 0:
        prediction = predict_breed_sagemaker_transfer(img_path)
        print("Hello, human!\nIf you were a dog..You may look like a {}".format(prediction))
    else:
        print("Error! Can't detect anything...")
```

In the following images we can see a test of the outputs that the run\_app method returns to several images:



In order to move to the real web app and be able to obtain a full application we need to perform some additional steps. As described, the standard procedure to use the model is using the `model_fn()` method that we defined while training but this usually requires to have a similar input as the model while training.

In order to deploy our application, we want to use the model but adding some additional code that preprocess our input before using the model in the same endpoint action, we can perform this using a SageMaker predictor.

We need to specify the estimator that we have trained and a directory with a python code containing a `predict_fn()` method that we will execute when calling the endpoint and two additional methods: `input_fn()` to preprocess the input and `output_fn()` to preprocess the output.

With the previous method that we have created I have completed the code in the `predict` file and I have created the inference endpoint.

```
● ● ●
from sagemaker.predictor import RealTimePredictor
from sagemaker.pytorch import PyTorchModel

class StringPredictor(RealTimePredictor):
    def __init__(self, endpoint_name, sagemaker_session):
        super(StringPredictor, self).__init__(endpoint_name, sagemaker_session,
                                              content_type='text/plain')

    estimator = estimator.attach(training_job_name='sagemaker-pytorch-2020-09-11-09-11-53-314')

    model = PyTorchModel(model_data=estimator.model_data,
                         role=role,
                         framework_version='0.4.0',
                         entry_point='predict.py',
                         source_dir='serve',
                         predictor_cls=StringPredictor)
    predictor = model.deploy(initial_instance_count=1, instance_type='ml.m4.xlarge')
```

Once that the endpoint is deployed we can test it using the `runtime.invoke_endpoint` function specifying the endpoint name, the content type and the body of the POST request having a json with the image in base64 format. We will take this snippet code as the template for our lambda function.

```
● ● ●
import boto3

runtime = boto3.Session().client('sagemaker-runtime')

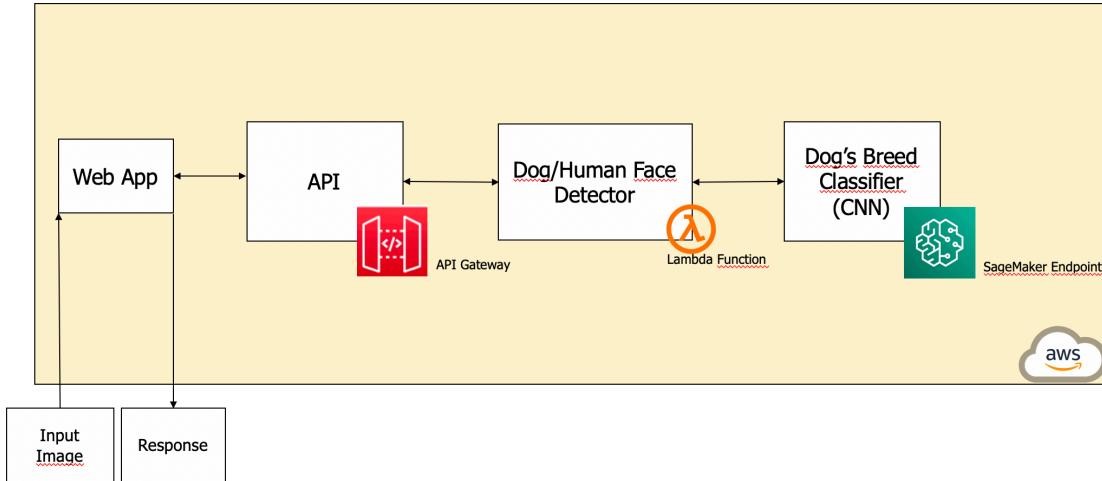
print("Sending to endpoint")
# Now we use the SageMaker runtime to invoke our endpoint, sending the review we were given
response = runtime.invoke_endpoint(EndpointName = 'sagemaker-pytorch-2020-09-12-20-32-128',      # The
                                    ContentType = 'application/json',          # The data format
                                    that is expected
                                    Body =
                                    '{"file": "data:image/jpeg;base64,/9j/4AAQSkZJRgABAQAAAQABAAAD//gA7Q1JFQVRPUjogZ2QtanBlZyB2MS4wIChlc2lu
BJSkcg5lBFRyb2NjIpLC8xdWF...k="}')
print("Endpoint request done")
print(response['Body'].read().decode('utf-8'))
```

The code response for that test is:

```
{"result": "Dogs Detected!\nIt looks like a Labrador retriever"}
```

## Reflection

During the whole capstone project development, the idea is to replicate this architecture:



In previous steps we have already deployed the SageMaker Endpoint with the best model and it is ready to receive POST requests with an image inside a json using base64 format.

On the opposite side the web app will be a local html file with a form ready to send the request with the image and wait a response from the API.

We just need to set up the two intermediate steps. First, let's create the lambda function with the code to call the endpoint.

At this point I have changed a bit the workflow from the capstone proposal as it has been easier to process the detector also in the endpoint instead of dealing with requirements and external data in the lambda function where I will just call the endpoint sending the same received image.

First step is to create a role to perform SageMaker invokations from the lambda function.

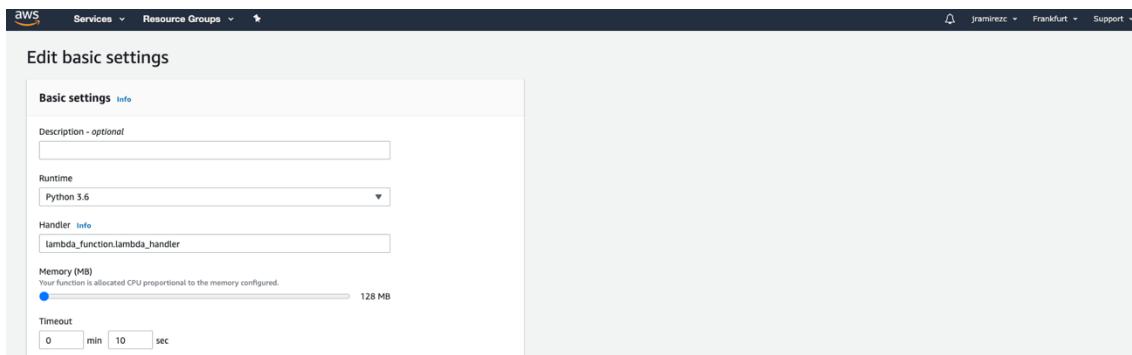
The screenshot shows the 'Create role' wizard in the AWS IAM console. The top navigation bar includes 'Services', 'Resource Groups', and user information. A progress bar at the top right indicates 'Step 1 of 4'. The main section is titled 'Select type of trusted entity'. It contains four options: 'AWS service' (selected), 'Another AWS account', 'Web identity', and 'SAML 2.0 federation'. Below each option is a brief description and a note about its usage. The 'AWS service' option is described as allowing AWS services to perform actions on behalf of the user. The 'Another AWS account' option is described as belonging to the user or a third party. The 'Web identity' option is described as using Cognito or OpenID providers. The 'SAML 2.0 federation' option is described as using corporate directories. A note at the bottom of the section states: 'Allows AWS services to perform actions on your behalf. [Learn more](#)'. Below this, a section titled 'Choose a use case' lists 'Common use cases': 'EC2' (allows EC2 instances to call AWS services) and 'Lambda' (allows Lambda functions to call AWS services). The 'Lambda' option is highlighted with a blue background.

Then, we can create the lambda function attaching the created role and including the desired Runtime, in this case Python 3.6

As mentioned initially I tried to add a lot of code inside the lambda function like installing the VGG16 model and other functions but I found a lot of problems and I decided to move that logic inside the inference endpoint and keep as simple as possible the lambda function.

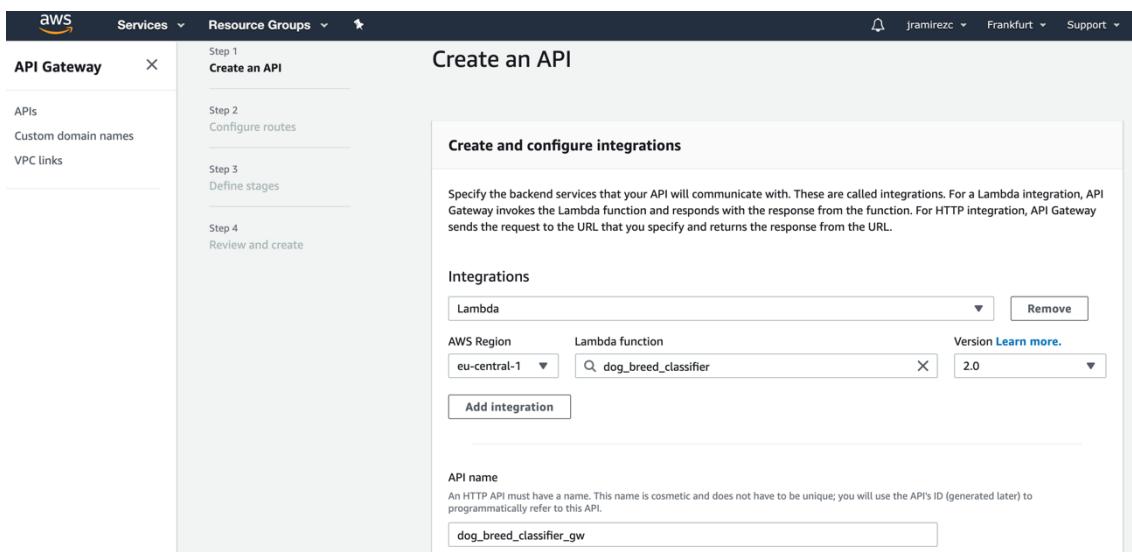
I have copied a similar code to the one used for the endpoint testing and I have included in the lambda function an environment variable with the endpoint name.

A very important step in my case was to change the timeout to a value bigger than 5-6 seconds. The default values are 3 seconds and my endpoint takes like 5 seconds to make the response. I have spent more time than expected dealing with this issue because I thought the problem was in the endpoint and it is a slow process to debug while waiting to redeploy.



With these steps our lambda function is ready and we are just missing the API gateway.

Moving to AWS API Gateway section we can create one api and configure the integration with our lambda function.



Then it is required to create a route, in this case I defined it as "/dog\_breed\_classifier" and using POST method.

The screenshot shows the AWS API Gateway console. On the left, there's a sidebar with navigation links for Services, Resource Groups, and various deployment stages. Under the 'Integrations' section, a specific integration named 'dog\_breed\_classifier' is selected. The main panel displays 'Routes for dog\_breed\_classifier\_gw' and details for the 'POST /dog\_breed\_classifier' route. It shows the Lambda function 'dog\_breed\_classifier' (eu-central-1) is used, with a payload format version of 2.0 (interpreted response format). The invoke permissions section includes a note about giving API Gateway permission to invoke the Lambda function, with a link to an example policy statement. A timeout of 30000 milliseconds is set.

Clicking on deploy we have the API gateway ready and it will work with Postman sending the expected input. In this case I have added the content-type as application/json and in the body a json with an image encoded in base64 format.

The screenshot shows a Postman workspace titled 'Untitled Request'. A POST request is being made to the URL [https://f35i442n34.execute-api.eu-central-1.amazonaws.com/prod/dog\\_breed\\_classifier](https://f35i442n34.execute-api.eu-central-1.amazonaws.com/prod/dog_breed_classifier). The 'Body' tab is selected, showing a JSON payload with a single key 'file' containing a long base64 encoded string. The 'Send' button is highlighted. Below the request, the response status is shown as 200 OK, with a time of 3.58 s and a size of 257 B. The response body is displayed as a JSON object: {"result": "Dogs Detected!\nIt looks like a Labrador retriever"}

As we can see the response took more than 3 seconds and it contains the answer from the system with the dog's breed.

As we want to make it interactive using a web app we need to configure as well from API gateway Cross-Origin Resource Sharing section some values. In this case, it is required to assign Access-Control-Allow-Origin and Access-Control-Allow-Methods so our API will reply the front with that information and we will avoid Cross-Origin errors.

I have used "\*" values as I was using just in my local the html but it is a better practice specify the front-end address in the origin.

The screenshot shows the AWS API Gateway CORS configuration page. The left sidebar has sections for APIs, Custom domain names, VPC links, Develop (Routes, Authorizations, Integrations, CORS), Deploy (Stages), Protect (Throttling), Monitor (Metrics, Logging), and a main API named 'dog\_breed\_class\_...'. The main content area shows a success message: 'Successfully updated "prod" stage with a new deployment ID (h4xp8e)'. Below it is the 'Cross-Origin Resource Sharing' section with a 'Configure' button. The 'Configure CORS' panel lists several CORS headers with their respective values:

- Access-Control-Allow-Origin: content-type
- Access-Control-Allow-Methods: GET, POST
- Access-Control-Expose-Headers: No Expose Headers are allowed
- Access-Control-Max-Age: 0 Seconds
- Access-Control-Allow-Credentials: No

With those changes we are ready to open our local index.html file that points to our API gateway.

The JavaScript code in the local web app is able to open a file explorer so the user can select an image, the image will be read and converted to base64 format and a json file will be included with the base64 image in the body of a POST request to the gateway.

The screenshot shows a browser window at localhost:8000. On the left, there's a form with the question 'Which is your dog breed?' and a file input field containing 'Labrador\_retr...er\_06455.jpg'. A 'Submit' button is visible. On the right, the browser's developer tools Network tab is open, showing network activity. One specific request to 'data:image/jpeg' is highlighted, showing a status of 200, a size of 0 B, and a duration of 0 ms. The Network tab also lists other files like 'localhost', 'jquery.min.js', and 'bootstrap.min.js'.

Once the gateway calls the lambda, and the lambda function calls the endpoint we will receive from the API gateway another json file with the result and our web app will show us the message with the dog's breed that the model has predicted closing our application workflow.

In my opinion, testing manually the results are good enough to set up a web app providing a positive user experience.

localhost:8000

Which is your dog breed?

Upload an image of a dog to find out...

Select image:  Labrador\_retr...er\_06457.jpg

Submit



Dogs Detected!  
It looks like a Labrador retriever

From the personal point of view, it has been a very challenging and enriching experience to work in a whole process and create a complete application that uses a CNN developed with PyTorch and trained with AWS Sagemaker.

Which is your dog breed?

Upload an image of a dog to find out...

Select image:  chow-chow.jpg

Submit



Dogs Detected!  
It looks like a Chow chow

## Improvement

I have identified some possible improvements for future work like:

- Train the model using more breeds to cover more cases.
- Test with different model configurations like alternatives pretrained models and different configurations for the fully connected or the optimizers.
- Apply different data preprocessing and data augmentation to the images.