Machine Learning Engineer Nanodegree

Capstone Project

Shivaadith Anbarasu
June 23rd, 2020

PROJECT OVERVIEW

Classification of animals is a highly important aspect of environmental conservation (as well as in the sciences involving animals). For centuries scientists and explorers have recorded information about the various species of animals and classified them based on their unique features. As more and more data are collected, the task of classification, and subsequent identification by others, becomes increasingly daunting. This growth is almost exponential. Even the most proficient scientists sometimes find it cumbersome to identify each different subspecies (main categories split even further) they might come across.

Figure 1: Four of the many subspecies of Rhinos

For the uninitiated, all four of these Rhinos on the left would appear the same, with varying horn sizes of course.

This holds true even for domesticated animals like dogs. All four dogs below are of different breeds!

Figure 2: Four different Dog breeds 1

Amongst the people I have come across, hardly a few can identify more than ten different breeds of dogs. There are over a hundred different breeds and many of them are indistinguishable from others. For the average person, knowing more than the basics is unnecessary, however being able to clearly distinguish between the different breeds is important for those in the science community – and for enthusiasts as well. With vast improvements in artificial intelligence technology, this mindboggling task of classification becomes far easier – provided that the models developed are reliant and accessible.

Dog breed classifier apps are quite famous these days, many of which produce nearly perfect results. A similar concept is used in the field of conservation and animal sciences. Hence, this project could serve as a starting point for developing more complex algorithms that are required for real world application in wildlife classification. This project makes use of deep learning frameworks to create a model that can predict breeds of dogs based on user input. The user inputs an image, the model first identifies if the image contains a dog or not. If so, it predicts the breed of the dog, but if the image is of a human then the model predicts a breed that closely resembles the person. In case the input is neither of those mentioned before, an error message is shown to the user.

There are identification and classification projects on Zooniverse, where people must manually tag animals in images for research purposes. These images are obtained from researches who make use of the online community in order to tag the different species of animals they come across in the field. The projects are open to all those who can contribute. The was the source of inspiration for my capstone project.

PROBLEM STATEMENT

In order to predict the breed of dog based on user input images, we make use of Neural Networks, specifically Convolutional Neural Networks or CNN for short. We use CNN since they are better adapted to image classification problems when compared to RNN (Recurrent Neural Networks).

There are two major sections to this problem. First is where the model needs to differentiate between human faces and dog faces in the input image. Then the model needs to predict the breed of dog based on the input. The first section is split into two subdivisions, each handling the identification of human and dog faces respectively.

For the detection of human faces, computer vision technology is required. OpenCV's Haar Cascade face classifier is used here for this purpose. Although there are few other classifiers that could produce better results, this will serve as a benchmark model which can be improved upon later.

```python
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

# load color (BGR) image
img = cv2.imread(human_files[0])
# convert BGR image to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# find faces in image
faces = face_cascade.detectMultiScale(gray)

# print number of faces detected in the image
print('Number of faces detected:', len(faces))

# get bounding box for each detected face
for (x,y,w,h) in faces:
    # add bounding box to color image
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

# convert BGR image to RGB for plotting
cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# display the image, along with bounding box
plt.imshow(cv_rgb)
plt.show()
```

```
Number of faces detected: 1
```

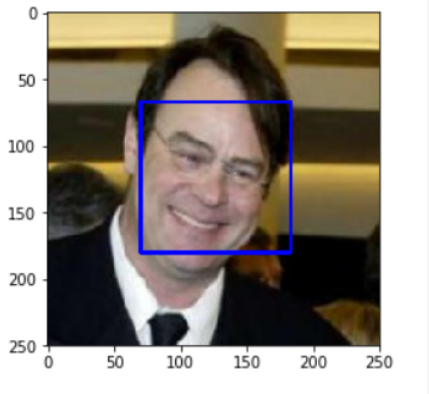


Figure 3: Using HaarCascade Classifier

In order to detect dog faces in the datasets, we make use of pre-trained models such as VGG16, ResNet50 etc These serve as valuable benchmark models as well. Initially, the VGG16 model is used to detect dogs in the images. The input image first undergoes resizing and normalizing in order to generalise the type of inputs to the model. It is also transformed into tensors.

```python
from PIL import Image
import torchvision.transforms as transforms

def VGG16_predict(img_path):
    '''
    Use pre-trained VGG-16 model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
        img_path: path to an image

    Returns:
        Index corresponding to VGG-16 model's prediction
    '''

    image = Image.open(img_path)

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


    image_tensor = transform(image).unsqueeze_(0)

    if use_cuda:
        image_tensor = image_tensor.cuda()

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image

    with torch.no_grad():
        output = VGG16(image_tensor)
        pred = torch.argmax(output).item()

    return pred
```

```python
import torch
import torchvision.models as models

# define VGG16 model
VGG16 = models.vgg16(pretrained=True)

# check if CUDA is available
use_cuda = torch.cuda.is_available()

# move model to GPU if CUDA is available
if use_cuda:
    VGG16 = VGG16.cuda()
```

Figure 4: Using a pretrained VGG16 model

The basic structure of the project is:

1. Import necessary libraries and perform pre-processing on the datasets
2. Identify human faces in the data sets
3. Use a standard pretrained CNN architecture to predict breed of dog in the input image.
4. Create a CNN from scratch in order to identify dog breeds.
5. Use Transfer Learning to create another CNN model to perform the same task, but with much better accuracy.
6. Write an algorithm to predict dog breeds based on inputs and test this algorithm.

After the development of the network and the prediction algorithms, the model is supplied with test data in order to visualise the results. If an image of a human is input, the model returns a predicted dog breed that closely resembles the person's face. If a dog is detected, then the out is simply the breed of the dog itself.

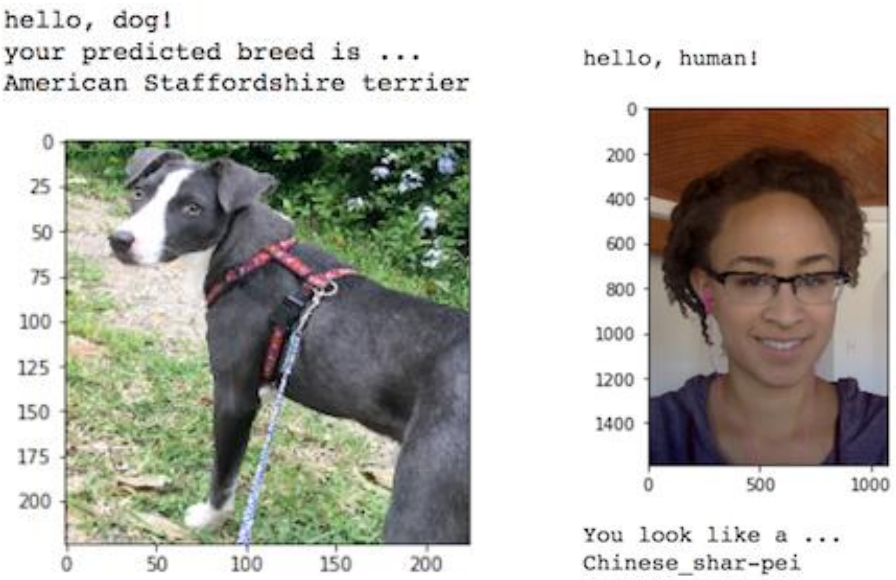


Figure 5: Sample inputs and predictions

In case the input consists of neither a human nor a dog (like a cat in this image), the model displays an error message to the user.

METRICS

Accuracy score is used as a basic evaluation method for the models in the project. This is simply the percentage of correct predictions over the total number of images being considered. This applies to the face detector, dog detector and the dog breed predictor models.

The most common metrics used in classification problems are the accuracy score, F1 score and confusion matrix. Although these measures provide solid evaluation for models, they might not perform as expected on datasets that have some degree of imbalance, as is the case with the current dataset. These metrics also tend to be overoptimistic in terms of producing results and we might end up with an evaluation that is far better than the actual score

Keeping that in mind, the main evaluation metric used for the final model of the project is the **Matthews Correlation Coefficient** or **MCC** for short. Although initially designed for Binary classification problems, it has been generalised to accommodate multiclass classification problems as well. It provides a far more reliable evaluation when compared to the previously mentioned metrics. The MCC score varies between -1 and +1, where the closer the score is to +1, the better the correlation (i.e. the more accurate/reliable are the predictions). This statistical metric produces a high score **only** if all four sections of the confusion matrix, namely true positives, false negative, true negatives and false positives, all produce good results. The score is proportional to the size of all the classes of a dataset. These properties set the MCC apart from F1 score and the confusion matrices.

$$\mathrm{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

The above formula is used to calculate the correlation coefficient. If any one of the summations in the denominator equal zero, the denominator is arbitrarily set to one. (in order to a avoid math error).

- TP - True Positives
- TN - True Negatives
- FP - False Positives
- FN – False Negatives

The formula above was designed for Binary classification problems, for the generalised version used in Multi-class classification problems such as in this project, we use

$$\mathrm{MCC} = \frac{\sum_k \sum_l \sum_m C_{kk} C_{lm} - C_{kl} C_{mk}}{\sqrt{\sum_k (\sum_l C_{kl})(\sum_{k' | k' \neq k} \sum_{l'} C_{k'l'})} \sqrt{\sum_k (\sum_l C_{lk})(\sum_{k' | k' \neq k} \sum_{l'} C_{l'k'})}}$$

Where K represents each of the different classes in the dataset. This generalisation was also derived from the Pearson coefficient that is widely used in statistics.

DATA EXPLORATION

The dataset provided consists of two folders split between 13,233 images of humans and 8351 images of various breeds of dogs. Within these folders, each of the unique categories are split into further sub-folders. There are 133 unique folders in the folder for dogs, each representing a different breed. These sub-folders are organised alphabetically in both the human and the dog directories. Roughly going over the images shows that there is a high imbalance between various classes. For example, some dog breeds contain over 60 images while other contain less than 40. Likewise, variations in number of images exist in the human folder as well, however this variation does not have a big impact on the output of the project model.

The images of dogs are contained within training, validation and testing datasets. This segregation method helps us train the model using certain parameters and configurations applied to the images whilst also allowing unseen/uncompromised data from the test set to be input to the model – which will serve as the evaluation technique. The validation set is used to validate/tune our model. This is where we find out the number of optimal hidden units and figure out the ideal stopping point for the backpropagation section of the algorithm. There are 6680 images in the training dataset, 835 images in the validation set and 836 images in the test set

Figure 6: Sample data from dog_images

Figure 7: Sample data from Human images

EXPLORATORY VISUALIZATION

The bar chart below represents various breeds of dogs in the dataset plotted against the number of images for each of these classes. I've chosen a bar graph representation since it provides a visually pleasing distribution of the dataset, and for its simplicity.

Matplotlib coupled with Seaborne was used to provide that bar plot output.

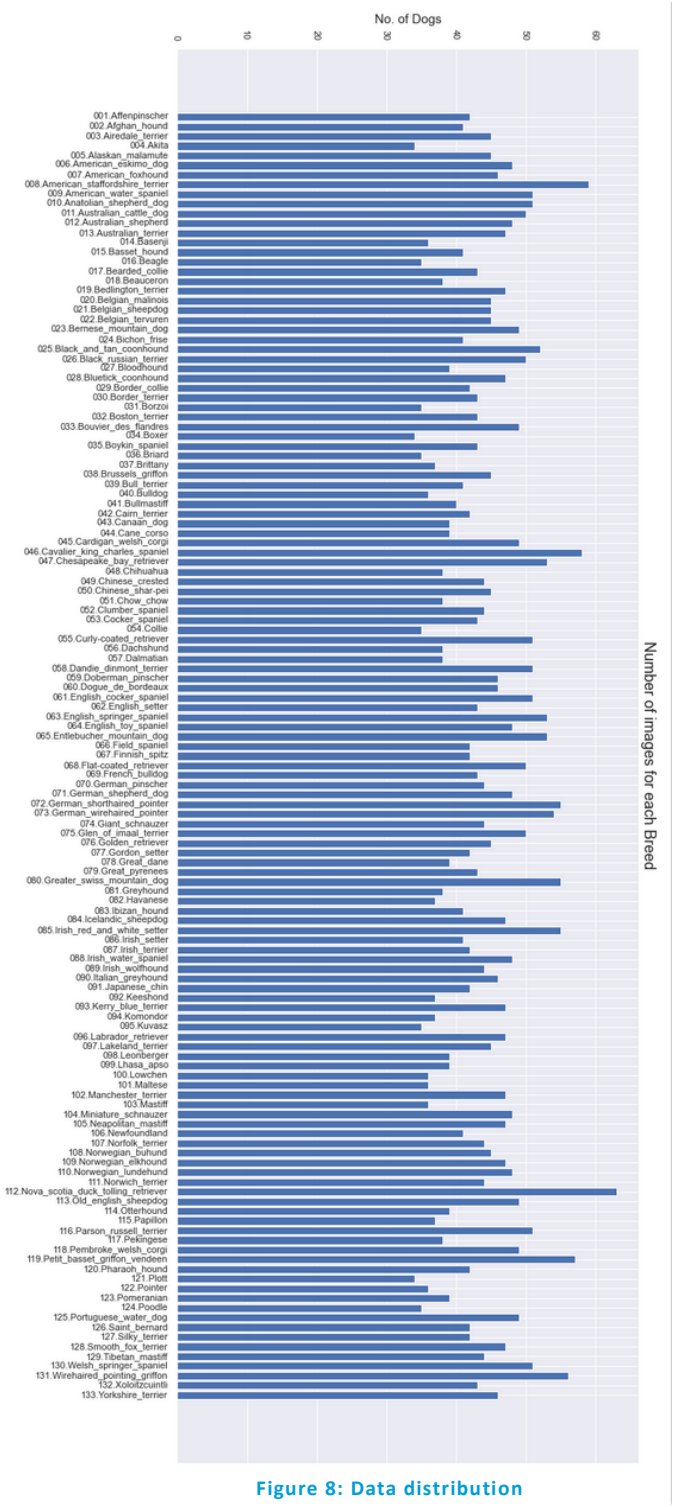


Figure 8: Data distribution

ALGORITHMS AND TECHNIQUES

For the face-detector: Haar Cascade

```
# returns "True" if face is detected in image stored at img_path
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
```

As previously stated in the Problem Statement section, the face detector uses OpenCV's Haar cascade classifier that was trained on facial patters. This is a machine learning algorithm that is widely used in identifying objects in images and videos.

An excellent video depicting its working mechanism:

https://www.youtube.com/watch?v=L0JkjIwz2II&feature=youtu.be

The accuracy obtained is as follows

```
98% of images in human_files were detected as human faces
17% of images in dog_files were detected as human faces
```

The algorithm employs a cascade function that is trained on large amounts of 'positive' and 'negative' images in order to detect useful objects when provided with new/unseen images. In its basic form, the main stages of the algorithm implementation involve feature selection, Adaboost training and using Cascading classifiers. During the training stage, the algorithm identifies and extracts all possible features in an image and through numerous 'learning' iterations, builds a linear combination of these features. Cascading classifiers are then built, consisting of multiple stages of these linear combinations. Boosting method is used to train these stages in order to select only the effective features, thereby speeding up the process. With each pass of the window over the images, the collected features are passed through the classifier where they are assigned positive or negative depending on whether an object is found or not. The main goal is to minimize the number of false negatives, which in turn ensures maximum reliability during regular usage.

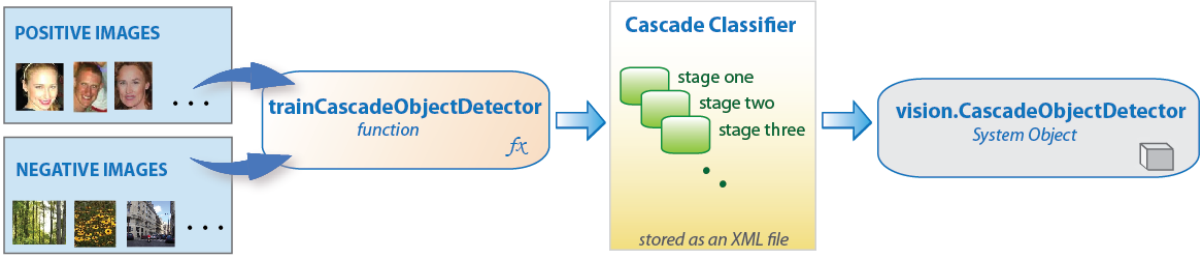


Figure 9: Haar Cascade Classifier

For the dog-detector:

1) VGG16

VGG-16 is a CNN architecture that was introduced in 2014 by Simonyan and Zisserman in the paper, *Very Deep Convolutional Networks for Large Scale Image Recognition*. (this paper is also the influence for the choices made in this project) The network is pretrained on over a million images from the database ImageNet. It is widely used in the field of computer vision and classification.

The image below visualises the layers involved.

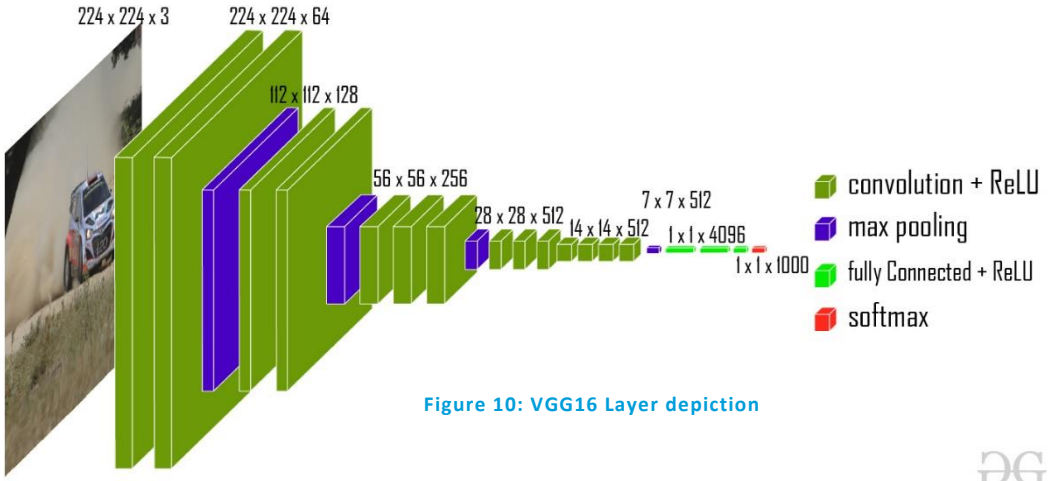


Figure 10: VGG16 Layer depiction

Although the network is capable of producing highly accurate results, owing to being trained on a very large variety of datasets, there are newer and more capable models which, as well as producing better results, also provide much quicker results. This VGG model will also serve as a benchmark for the project. Major drawback to VGG networks is their relatively slow speed of training and implementation.

The predictions from the model returned an accuracy score as shown in the image below.

```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    pred = VGG16_predict(img_path)

    if (pred >= 151 and pred <= 268):
        return True
    else:
        return False # true/false
```

```
0% of images in human_files were detected as dogs
99% of images in dog_files were detected as dogs
```

The values 151 and 268 are the ImageNet dictionary keys that correspond to images of dogs. Hence if the index returned from the pre-trained model function lies between these two values, the image contains a dog.

2) *ResNet-101*

The ResNet (Residual Network) architecture has the capability of extended over 100 layers deep. This network also eliminates another drawback in traditional/sequential networks, which is the vanishing or explosion of gradients as the network gets deeper. In these traditional networks, as the number of layers is increased beyond a certain level, the accuracy of the output diminishes owing to the problems stated above. Ideally, we would want a deeper network to produce much better results than shallow ones.

ResNet uses Residual Blocks which establish skip/identity connections between the input and output of the layers.

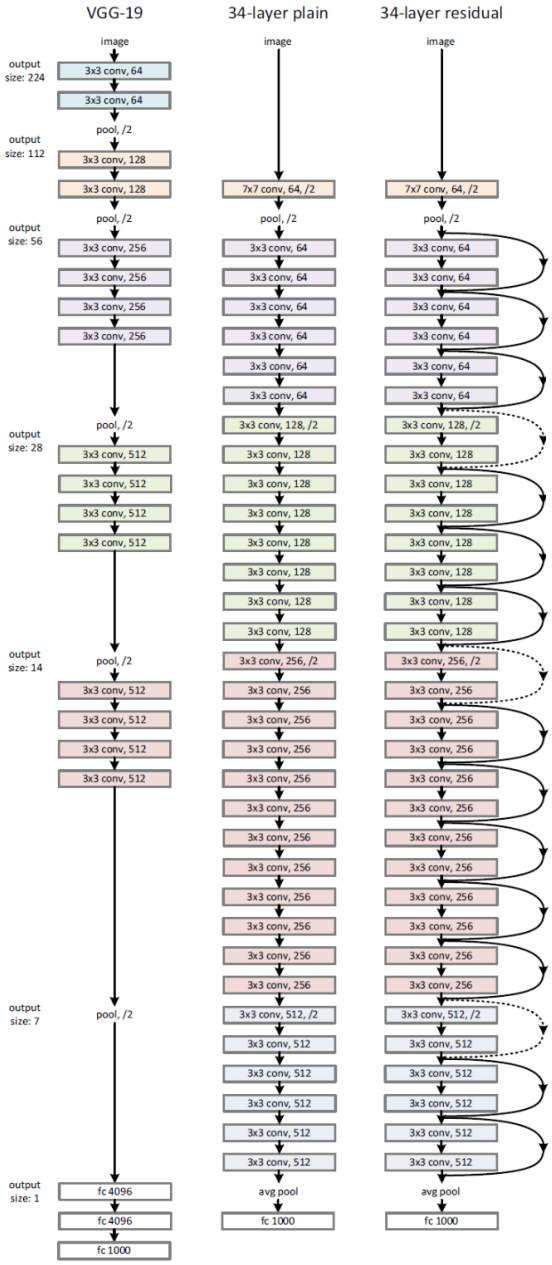


Figure 11: ResNet Layer depiction

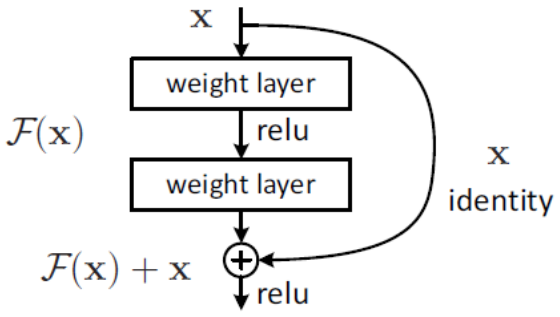
The image shows the differences in architecture between plain and residual networks. The residual learning technique makes it much easier to handle very deep networks without having to deal with the consequences of gradient vanishing/explosion.

	plain	ResNet
18 layers	27.94	27.88
34 layers	28.54	**25.03**

Table 2. Top-1 error (%, 10-crop testing) on ImageNet validation. Here the ResNets have no extra parameter compared to their plain counterparts. Fig. 4 shows the training procedures.

The difference in errors between traditional and residual networks as the depth increases can be seen here. (trained on the same datasets)

This formation also helps minimize the overfitting/underfitting of data.



Data Loaders

Data input to the network is required to be in a certain format. The process of handling, pre-processing and feeding data into the network is both very time consuming and requires large amounts of memory/computing resources. It can be painstakingly inefficient especially when the dataset is very large. To make the whole process much more efficient and less resource straining, dataloaders are used. The PyTorch framework provides and easy way of creating these dataloaders and providing methods to alter the data images into an acceptable format.

```
train_loader = torch.utils.data.DataLoader(images_data['train'], batch_size=20, num_workers=0, shuffle=True)

valid_loader = torch.utils.data.DataLoader(images_data['valid'], batch_size=20, num_workers=0, shuffle=True)

test_loader = torch.utils.data.DataLoader(images_data['test'], batch_size=20, num_workers=0, shuffle=False)

loaders_scratch = {
    'train': train_loader,
    'valid': valid_loader,
    'test': test_loader
}
```

The data augmentation precedes this step but will be discussed in the upcoming sections.

The data from each of the training, validation and test set are augmented and the loaded using the dataloaders functionality. The batch size used here is varied between the different models that were implemented, but the test dataloader remains at a batch size of 20 throughout.

Benchmark

The benchmark for the project is the CNN architecture that is created from scratch. This model produces the lowest accuracy score amongst all those implemented here, and since the model framework is trained and tested under similar conditions to the final implementation, it serves as a valuable baseline for comparison

```
Test Loss: 3.756675


Test Accuracy: 15% (126/836)
```

The architecture created from scratch produced the results above. Clearly an accuracy of 15% is far too low for practical purposes. Further improvements are made to the networks and hyperparameters are tuned well enough in order to produce much better results.

The minimum accuracy score requirements specified was 10% for the fresh network and 60% for the network created using transfer learning methods.

DATA PREPROCESSING

As stated in the previous section, before the dataloaders can be created, the datasets must undergo augmentation process.

```
# Data augmentation step: references -- PyTorch Tutorials and research papers
data_transforms = {
    'train': transforms.Compose([
        transforms.Resize(256),
        transforms.RandomRotation(10),
        transforms.RandomHorizontalFlip(),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'valid': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'test': transforms.Compose([
        transforms.Resize(size=(224,224)),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])
}
```

As can be seen from the image above, PyTorch provides easy methods to implement these processing steps on the datasets. This transforms method is applied to all three of the training, validation and testing sets

All the images are resized to 256 by 256. These resized images are then cropped around the centre to a 224 by 224 – roughly going through the images shows that most dogs are located at the centre of the frame.

For the training set alone, additional parameters are used, namely RandomRotation and RandomHorizontalFlip. This transformation process provides variations and additional formatting which is helpful in producing the best possible results. All these variations help mimic real input images provided by users thereby establishing a good amount of generalisation, since these inputs could be vastly different from what the model sees during training. Tensors are also created during this process

Test data undergoes only resizing in order to maintain its uncompromised state. This is the data set that will be used for the final evaluation hence it needs to be presented in its raw format.

Normalisation is applied to all three datasets in accordance with standard values used in similar projects on Kaggle (this applies to few other sections of the project as well).

IMPLEMENTATION

CNN from scratch:

The network architecture is defined as follows:

```
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(256, 512, kernel_size=3, padding=1)
        self.conv5 = nn.Conv2d(512, 512, kernel_size=3, padding=1)

        self.pool = nn.MaxPool2d(2,2)

        self.fc1 = nn.Linear(25088,512)
        self.fc2 = nn.Linear(512, 133)

        self.dropout = nn.Dropout(0.25)

        self.batch_norm = nn.BatchNorm1d(512)
```

With the paper by Simonyan and Zisserman as the primary reference throughout the process, I had decided to implement five convolutional layers. Initially however, there were only three convolutional layers and three fully connected layers, but this form achieved less than satisfactory results which led me to exhaustive research and experimentation before landing on the final form shown above. Since convolutional layers perform more efficiently at learning spatial eatures and are 'lighter' than fully connected(FC) layers, I had gone with the choise of only two FCs. This adds to a total number of weight layers equalling 7 (the format used in the paper is not follwed strictly of course, hence the diference)

Spatial pooling occurs over five MaxPool layers between the convoluted layers in order to simplify the network. The window size for max-pooling is set to a size of 2 by 2 pixels, with a stride of 2. The receptive field is restricted to a size of 3 by 3 (setting kernel size to 3) and padding of 1 pixel is applied. The stride is set to its defalt value of 1. The output from the final fully connected layer is fixed at 133, each of which corresponds to a unique class/breed of a dog in the dataset. Batch normalisation is applied in order to regularize the network – albeit somewhat unnecessary since the difference in results wasn't significant.

The dimensions of the original image stood at 224 by 224 while the input to the first fully connected layer stands at 7 by 7 coupled with a depth of 512. This major difference helps in performing better generalisation on images since more key features can now be identified.

The forward function:

```
    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 = self.pool(F.relu(self.conv4(x)))
        x = self.pool(F.relu(self.conv5(x)))


#         print(x.shape)


        x = x.view(x.size(0),-1)
        x = self.dropout(x)

        x = F.relu(self.batch_norm(self.fc1(x)))
        x = self.dropout(x)

        x = self.fc2(x)

        return x
```

The activation function used in this network is the ReLU, which is applied across all the layers. The loss function used is the CrossEntropyLoss coupled with the Adam optimizer – all standard practices picked up from numerous kaggle competition notebooks.

The ReLU function provides a simple and effective method for activation. For positive values, the function is a linear model and for negative values it stays at zero. This results in much less complicated computations thereby reducing training time effectively. The linearity also ensures that there is no vanishing gradients.

Cross Entropy Loss, also known as Log Loss, provides an output between 0 and 1 thereby making it easy to predict the outcome directly. As the prediction reaches the ground truth, the loss value decreases. This also means that incorrect predictions are penalised heavily, which improves final accuracy score and the confidence/reliability in the model.

Training the model:

The model was trained on 20 Epochs and the initial and final loss values stood at

```
Epoch: 1        Training Loss: 4.839165         Validation Loss: 4.671487
Validation loss decreased (inf ==> 4.671487). Saving model...
...
Epoch: 20       Training Loss: 2.783571         Validation Loss: 3.407300
Validation loss decreased (3.570901 ==> 3.407300). Saving model...
```

As stated in the Benchmark section, the testing accuracy of this network was 15%

CNN using Transfer Learning:

Using transfer learning, the accuracy of the model had improved significantly. A pre-trained VGG16 model was used in this section.

```
## TODO: Specify model architecture
model_transfer = models.vgg16(pretrained=True)

for param in model_transfer.parameters():
    param.requires_grad = False

model_clf = nn.Sequential(nn.Linear(25088, 2048),
                          nn.ReLU(),
                          nn.Dropout(0.5),
                          nn.Linear(2048, 512),
                          nn.ReLU(),
                          nn.Dropout(0.5),
                          nn.Linear(512, 133))

model_transfer.classifier = model_clf
```

After running into several errors and complications, I had decided to define my own classifier. A new set of dataloaders was also created but with the only difference from the previous loaders being the batch size. The batch sizes for training and validation sets for the new model are 30 each, whilst previously it was 20. In order to prevent backpropagation during feature extraction, the parameters are frozen. This new classifier consists of three fully connected layers and dropout value of 0.5 in order to minimize overfitting. The same activation function, loss function and optimizer from the previous section were used here as well.

Upon training the model (the number of epochs was reduced to 12), the difference in the initial and final loss values stood at:

```
Epoch: 1        Training Loss: 4.265953         Validation Loss: 2.231884
Validation loss decreased (inf ==> 2.231884). Saving model...

Epoch: 12       Training Loss: 2.037477         Validation Loss: 1.064958
Validation loss decreased (1.084981 ==> 1.064958). Saving model...
```

The test accuracy had clearly improved quite a bit:

```
Test Loss: 1.100100


Test Accuracy: 67% (564/836)
```

REFINEMENT

So far, the accuracy scores produced were 15% for the model created from scratch and 67% for the model created using transfer learning. Although the improvement is quite significant, from a practical point of view 67% is still quite not satisfying. This is especially true if the model were to be deployed to web or mobile applications (which I expect to do). In order to further improve the predictions, hyperparameters had been tuned and another model based on the ResNet-101 architecture was built. As stated in the previous section, the ResNet model can produce much better accuracies when compared to traditional networks. (however, based on personal experiments on similar datasets, the VGG model seemed to outperform both ResNet and InceptionV3 – I suppose a lack of experience in this field caused the differences)

Stage One - Using a tuned VGG model:

```
mod_v2 = models.vgg16(pretrained=True)

for param in mod_v2.parameters():
    param.requires_grad = False

# as defined in the research paper
mod_clf = nn.Sequential(nn.Linear(25088, 1024),
                        nn.ReLU(),
                        nn.Dropout(0.5),
                        nn.Linear(1024, 133))

mod_v2.classifier = mod_clf


if use_cuda:
    mod_v2 = mod_v2.cuda()
```

The only difference between this model and the one from the previous section (using a pre-trained vgg16 model), is that there are only two fully connected layers in this one. As mentioned before, quite a bit of experimenting was done on the models in order to optimise the final form. During this optimisation process, the accuracy of the previous model improved from around 50% to the current value of 67% (the original model). This optimisation was also done after declaring this current model (model v2). So, in the initial stages of refinement, the differences in accuracy stood at 17%, which is a significant amount. Although upon further experimenting, the difference was brought down to a mere 3%. The main factor behind this improvement seems to be in the reduction of fully connected layers.

The accuracy obtained from this refined VGG implementation is:

```
Test Loss: 1.162889


Test Accuracy: 70% (588/836)
```

Final Model – *Using ResNet-101:*

```
use_cuda = torch.cuda.is_available()
## TODO: Specify model architecture
mod_v3 = models.resnet101(pretrained=True)

mod_v3.fc.parameters = nn.Linear(2048, 133, bias=True)

if use_cuda:
    mod_v3 = mod_v3.cuda()
```

The concept of transfer learning is applied here as well. The output layer is modified to accommodate 133 different classes, corresponding to the breeds of dogs in the dataset. To accommodate for the use of a new metric for evaluation, the testing function was slightly modified.

After training the model (model v3) for 10 Epochs, the test accuracy obtained is:

```
Test Loss: 0.500795


Test Accuracy: 85% (714/836)
```

There has been a significant improvement when compared to the implementations using VGG16. Owing to this improvement, I had decided to apply the evaluation metric to this final model. The evaluation metric of choice is the **Matthews Correlation Coefficient (MCC).**

Scikit-learn provides a useful library which contains the MCC metric as well.

```
The Matthews correlation coefficient(MCC) is: 0.8530745743386203
```

The closer the MCC value is to 1, the more reliable the model is. A value of 0.85 shows that the model performs considerably well.

FINAL EVALUATION

Metrics	Model from Scratch	Model from transfer learning – VGG16	Refinement Stage one: VGG16	Final Model: ResNet101
Final Training Loss	2.78	4.27	1.26	**0.20**
Final Validation Loss	3.41	2.23	1.10	**0.46**
Test Loss	3.76	1.10	1.16	**0.50**
Test Accuracy	15 %	67 %	70 %	**85 %**

MCC	**0.85**

From the above table it is evident that there has been a significant improvement from the initial stage. The final model was trained for only 10 Epochs, at which point the validation loss was still falling. With more iterations and finer hyperparameter tuning, the accuracy and MCC scores could increase even further thereby providing a highly accurate and reliable model.

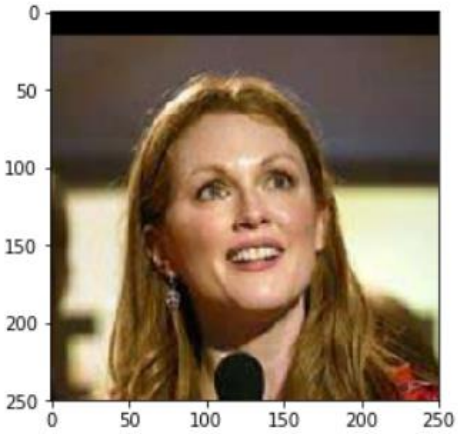
The prediction stage of the model follows the following protocol:

- The input image if first transformed using the same protocols that were used with the test dataset, with the additional stage of cropping.
- If a dog is detected, then the model predicts the breed of the dog and displays the prediction.
- If a human is detected, then the model predicts a breed that resembles the human face and displays the prediction
- If the image consists of neither of the above, then an error message is displayed to the user.

The use of simple for loops and if-statements is enough in building an algorithm for this process.

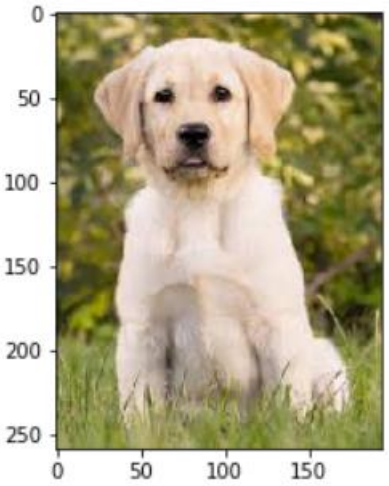
RESULTS

A Human???



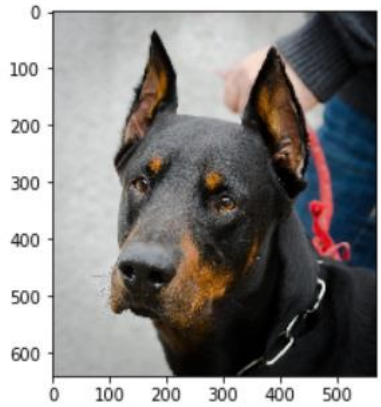
Ok, this person looks like a....
English toy spaniel

A DOG!!!



This is a....
Labrador retriever

A DOG!!!

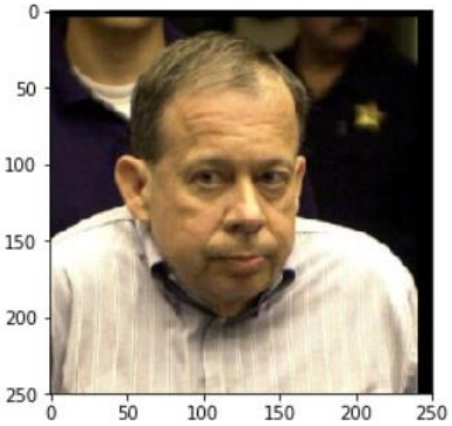


This is a....
Beauceron

A Human???

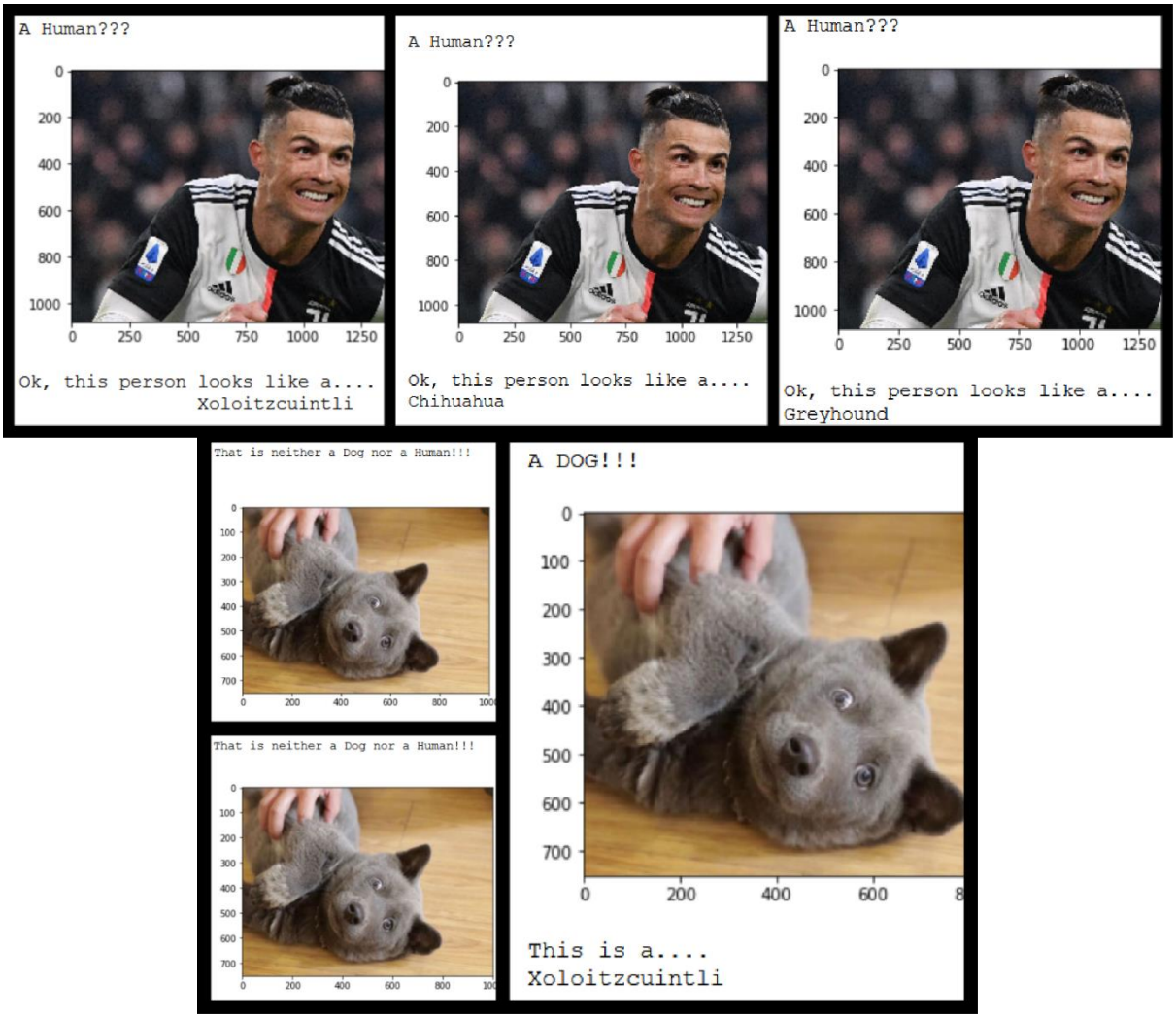
Ok, this person looks like a....
Greyhound

A Human???

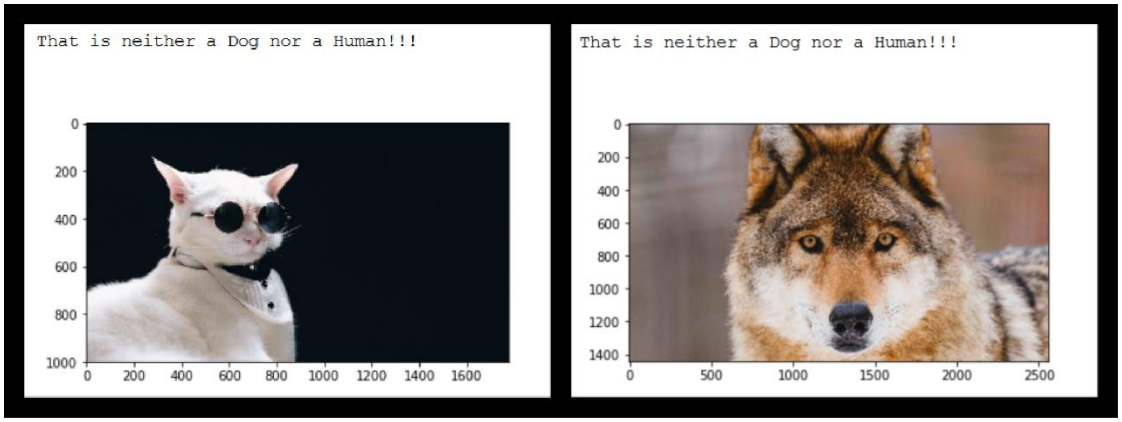


Ok, this person looks like a....
French bulldog

Differences between the Original, Tuned and the Final Model



Common results across all models



REFLECTION

This has been a very educational project. I had come across a lot of new methods of implementing known ideas. I lacked the experience in dealing with large scale projects on Convolutional Neural Networks, apart from those provided by Udacity in the lectures. This made the project especially challenging when I came across errors that I had never seen before. Extensive research was required in order to rectify those mistakes and produce a reasonably well modelled program. I am sure there are far better solutions to this problem, however, this being my first major project in an entirely new field, I believe I have exceeded my own expectations.

The most interesting aspect of the project was in the refinement section. Although I had followed well established guidelines from research papers during the project, the refinements were mostly experimental – sort of a trial and error method mixed with research work. This is where my creativity and logical implementation limits were pushed, this was also the best results were produced.

Looking back the entire process, it would have certainly been helpful if I had taken some refresher courses on certain topics such as CNN architectures and implementations. My expertise lies in research work and obtaining any/all useful information needed for a project, the only drawback is that this process takes a significant amount of time. With further personal development in this field of deep learning, I am confident that there will be greater fluidity in future projects.

IMPROVEMENT

One of the major improvements that could be applied to this model is a much larger and more varied dataset for training purposes. As I have mentioned before, by going through the images it was evident that there are insufficient variations between the images. They might not effectively model real-world user inputs. A larger number of Epochs would also be effective in improving the model.

Another key area is in the choice of algorithms themselves. With my limited experience, I had initially gone with the VGG model which turned out to be less than satisfactory – although hyperparameter tuning was not done effectively and the network itself needed better architecture. In the end it turned out that ResNet was much better suited for this problem that I had initially believed. Perhaps there are other networks that provide even better results. Also, there could be a few key elements of designing the model that I had missed out on. Further research is required in order to formulate a near perfect model and deploy it for real world use on web and/or mobile platforms.

REFERENCES

1) Very Deep Convolutional Networks For Large-Scale Image Recognition by Karen Simonyan & Andrew Zisserman, 2015, https://arxiv.org/abs/1409.1556v6
2) Chicco, Davide & Jurman, Giuseppe. (2020). The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. BMC Genomics. 21. 10.1186/s12864-019-6413-7.
3) Choice of metrics: https://www.researchgate.net/publication/338351315_The_advantages_of_the_Matthews_correlation_coefficient_MCC_over_F1_score_and_accuracy_in_binary_classification_evaluation
4) HaarCascadeClassifier: http://www.willberger.org/cascade-haar-explained/
5) Normalisation of images: https://machinelearningmastery.com/how-to-normalize-center-and-standardize-images-with-the-imagedatagenerator-in-keras/
6) ResNet explanation: https://medium.com/@14prakash/understanding-and-implementing-architectures-of-resnet-and-resnext-for-state-of-the-art-image-cf51669e1624
7) Deep Residual Learning for Image Recognition by Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun – Microsoft Research
8) Dropout: A Simple Way to Prevent Neural Networks from Overfitting, by Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, Journal of Machine Learning Research 15 (2014) 1929-1958, Submitted 11/13; Published 6/14
9) Sample images of different dog breeds: http://www.vetstreet.com/our-pet-experts/seeing-double-14-look-alike-dog-breeds-can-you-tell-them-apart
10) Images of Rhinos: https://www.savetherhino.org/rhino-info/rhino-species/