Machine Learning Nanodegree Capstone Project Report

Dog Breed Identification

Daniel Fernando García Rodríguez

Definition

Project Overview

Automatic image classification is one of the most important areas of machine learning, both in research and commercial applications. Since 2012 Convolutional neural networks (CNN) based image classification techniques dramatically improved the then current benchmarks setting CNN as SOTA for computer vision while propelling the hype of the so called “Deep Learning revolution”. The goal of this project is to explore the CNN-based approach for image classification using a dog breed dataset.

Problem Statement

Although former computer vision approaches were certainly capable of some image classification task, Deep Learning approaches just outperformed them. Among image classification problems animal breed differentiation is a challenging one because the differences can be subtle and complex; because of that applying deep learning could be an effective solution. The aim of this project is to develop a model capable of distinguishing between dog breeds when the user submits a dog image. This process implies two steps: recognizing if the picture is from an actual dog, and if it is then identifying its breed. This supervised learning approach requires a labeled dataset of dog pictures with its corresponding breed.

Metrics

The used metric is multiclass class log loss, which measures the performance of a multiple class classification model where the prediction score is a probability (between 0 and 1). In this project the output consists of a 133 sized probabilities vector assessing how likely the input image belongs to each one of the 133 breeds. Cross-entropy values increases if the predicted probability diverges from the actual label.

Analysis

Data Exploration

The provided dataset consists in a set of dog pictures labeled with its corresponding breed; it also includes a series of human pictures. The dataset is divided in two separated files, one for dogs and one for humans, both hosted by Udacity. The human set is required for the Haar cascade part of the project which consist in differentiating between whether there is a dog in a picture; then a neural network model will differentiate the breed of a dog picture supplied as input.

The dataset is divided into 3 segments: Train consisting in between 30 – 70 pictures per breed, Test segment with 6 – 10 pics per breed and Validation also 6 – 10 per breed; all containing the same 133 dog

breeds. The human dataset contains 5794 pictures of (you guess it!) humans. Not all the dog pics contains only dogs.

The images are RGB color space, meaning every picture is represented by a m*n*3 tensor. All human images feature a resolution of 250x250, while dogs ones presents various resolutions from ~200x~200 to ~1000x~1000

Algorithms and Techniques

The solution of this project will include a trained detector using CNN to detect dog breeds based on the supplied picture. CNN stands for convolutional neural network, a deep learning architecture which is widely used in image classifications.

A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets can learn these filters/characteristics.

The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlaps to cover the entire visual area.

A ConvNet can successfully capture the Spatial and Temporal dependencies in an image through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.

A complete CNN model also features hyperparameters such as number of epochs, batch size, pooling, number of layers, weight, and dropouts. Example explanations of some hyperparameters are listed below:

1. Number of epochs: the number of times the entire training set pass through the neural network.
2. Number of hidden layers: the number of convolutional and linear layers specified in the CNN model, more layers can result in higher computational cost, less layers can result in underfitting.
3. Dropout: a preferable regularization technique to avoid overfitting in deep neural networks. The method simply drops out units in neural network according to the desired probability.

Haar Cascades are also used for human – dog classification, this method was proposed by Paul Viola and Michael Jones in their paper Rapid Object Detection using a Boosted Cascade of Simple Features a machine learning-based approach where a lot of positive and negative images are used to train the classifier.

Benchmarks

The benchmark for the model can be referenced to the Kaggle leaderboard for dog breed identification competition. The target for this model is to reach a multiclass loss score less than 0.01, which is in the top 100 of the competition. The other benchmark will be 90% prediction accuracy, which will be used as the upper limit. The benchmark set by Udacity will be 60% prediction accuracy, which will be used as the lower limit. The final performance of the model will sit in between the two limits.

Methodology

Data Preprocessing

The training, test, validation images were resized, and center cropped into 224x224 pixels for normalizing input data format, then randomly flipped in the horizontal direction before transforming into tensors for data augmentation. Transformed data were organized into train, test, and validation directories, respectively.

```
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes

ImageFile.LOAD_TRUNCATED_IMAGES = True
#data_loader = torch.utils.data.DataLoader
batch_size = 20
workers = 0
dog_img = 'dogImages/'

img_transform = {'train': transforms.Compose([transforms.Resize(size=224),
                                      transforms.CenterCrop((224,224)),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.ToTensor(),
                                      transforms.Normalize(mean=[0.485,0.45
6,0.406],
                                                           std=[0.229,0.224
,0.225])]),
                 'test': transforms.Compose([transforms.Resize(size=224),
                                      transforms.CenterCrop((224,224)),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.ToTensor(),
                                      transforms.Normalize(mean=[0.485,0.45
6,0.406],
                                                           std=[0.229,0.224,
0.225])]),
                 'valid': transforms.Compose([transforms.Resize(size=224)
,
                                      transforms.CenterCrop((224,224)),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.ToTensor(),
                                      transforms.Normalize(mean=[0.485,0.45
6,0.406],
                                                           std=[0.229,0.224,
0.225])])}

# load and transform data using ImageFolder
train_data = datasets.ImageFolder(dog_img+'train', transform=img_transform
['train'])
test_data = datasets.ImageFolder(dog_img+'test', transform=img_transform['
test'])
valid_data = datasets.ImageFolder(dog_img+'valid', transform=img_transform
['valid'])

train_loader = torch.utils.data.DataLoader(train_data,
                                           batch_size=batch_size,
```

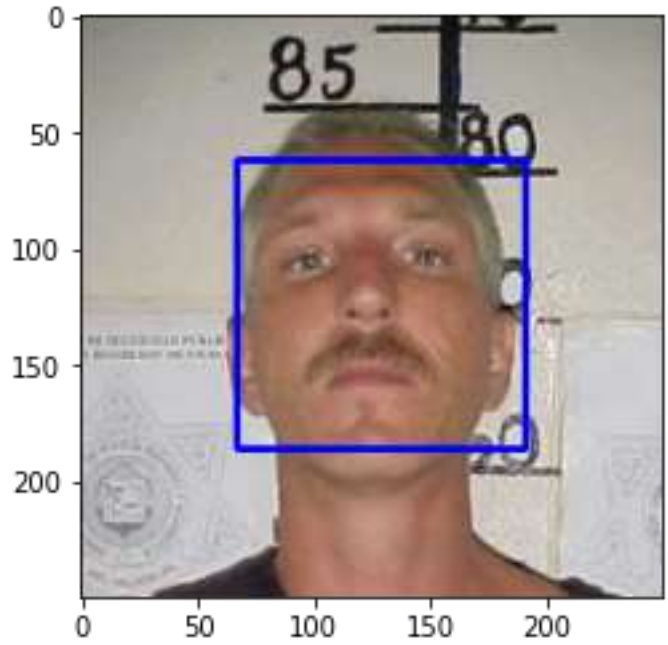
```
                                                    num_workers=workers,
                                                    shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data,
                                                    batch_size=batch_size,
                                                    num_workers=workers,
                                                    shuffle=False)
valid_loader = torch.utils.data.DataLoader(valid_data,
                                                    batch_size=batch_size,
                                                    num_workers=workers,
                                                    shuffle=False)

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

Implementation

For the human detector: The image is firstly feed to a human detector function to detect if a human face is presented. The pre-trained model haar-cascade classifiers is implemented to achieve this functionality.

```
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
```

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



Dog detector: If no human face is detected by the human detector, the image will be passed to the dog detector to see if a dog is presented. The dog detector is constructed using a VGG16 pre-trained model which can identify classes in the 1000 categories.

```
# Set PIL to be tolerant of image files that are truncated.
ImageFile.LOAD_TRUNCATED_IMAGES = True

def img2tensor(img_path):
    '''
    load an image and tranform it to tensor for VGG16 model input
    '''
    # load image
    img = Image.open(img_path).convert('RGB')
    # transform the image to tensor which can be accepted by the VGG model
    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_tensor = transform(img)[:3,:,:].unsqueeze(0)
    return img_tensor

def tensor2img(tensor):
    '''
    Convert tensor to img
    '''
    img = tensor.to("cpu").clone().detach()
    img = img.numpy().squeeze()
    img = img.transpose(1,2,0)
    img = img * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456,
0.406))
    img = img.clip(0, 1)
    return img

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
    '''

    ## 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

    img_tensor = img2tensor(img_path)

    # use CUDA if available
```

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

    # use VGG model to detect dog
    output = VGG16(img_tensor)

    return torch.max(output,1)[1].item() # predicted class index

def dog_detector(img_path):
    ## TODO: Complete the function.
    idx = VGG16_predict(img_path)
    is_dog = False
    if idx in range(151, 268):
      is_dog = True
    return is_dog # true/false

is_dog_in_humans = 0
for i in range(len(human_files_short)):
    if dog_detector(human_files_short[i]) is True:
        is_dog_in_humans += 1

is_dog_in_dogs = 0
for i in range(len(dog_files_short)):
    if dog_detector(dog_files_short[i]) is True:
        is_dog_in_dogs += 1

percent_dog_in_humans = is_dog_in_humans / len(human_files_short)
percent_dog_in_dogs = is_dog_in_dogs / len(dog_files_short)

print("Percent Dogs in Human Files: ", percent_dog_in_humans, "\nPercent D
ogs in Dog Files: ", percent_dog_in_dogs)
```

```
Percent Dogs in Human Files: 0.0

Percent Dogs in Dog Files: 0.97
```

Dog breed classifier: if a dog is detected by the dog detector, the image will be forwarded to the dog breed classifier. In this project, I first create a scratch CNN model for predicting the dog breed. The structure for the scratch CNN model is described below:

1. First convolution layer uses 32 filters, max pooling and stride reduced the image size to 56x56

2. Second convolution layer uses 64 filters, max pooling and stride reduced the image size to 14x14

3. Third convolution layer uses 128 filters and max pooling reduced the image size to 7x7

4. Two linear layers are assigned, with 133 as the output size. 5. Dropout is set to be 0.3 to avoid overfitting.
However, the scratch model is a simple CNN which has a high possibility of not achieving the accuracy metric (60%-90%) that is proposed in this project.

Refinement

Several structures have been tested with the scratch CNN model. Specifically, the number of filters (8 vs 16 in the first layer), pooling and striding (striding=0 vs striding=2). The scratch model is trained using GPU mode in Google Colab environment. The worst and best accuracy after 20 epochs is 8% and 1%, respectively. No drastic improvement is found using limited filters variations and pooling strategies.

That could be result of scratch CNN's being too basic for achieving higher accuracy, also possibly requiring more training epochs and more data for training. That results in that scratch CNN model is a far cry from the proposed accuracy (60%-90%) in this project. Therefore, using a more established pretrained model makes sense. In this project, ResNet50 pre-trained model is used to serve as the real dog breed classifier. The ResNet50 model is tuned to match the 133 class outputs for this project. The ResNet50 model should theoretically produce much higher accuracy.

Results

Model Evaluation and Validation

As mentioned in the refinement section, the scratch CNN model only features a poor prediction accuracy of 11% (Figure 9). Specifically, after 20 epochs of training, the training loss and validation loss are reduced to ~3.53 and ~4.06, respectively. However, these are still very poor performance values comparing to some of the best metrics on Kaggle platform, which features a loss score of 0.01. The result indicate that the scratch model is too simple in terms of architecture. Additionally, the size of training data can be a caveat to a fresh model like this.

However, by implementing the ResNet50 model, after 20 epochs of training. Training loss and validation loss are reduced to ~1.72 and ~1.53 (Figure 10), which is a significant improvement comparing to the scratch CNN model. The transferred model also shows a test loss of ~1.51 and an accuracy of 77%. This result suggests a significant advantage of using a pre-trained CNN model in terms of general image classification than building a CNN model from scratch.

```
# train the model
model_transfer = train(20, loaders_transfer, model_transfer, optimizer_transf
er, criterion_transfer, use_cuda, 'model_transfer.pt')

# load the model that got the best validation accuracy (uncomment the line be
low)
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

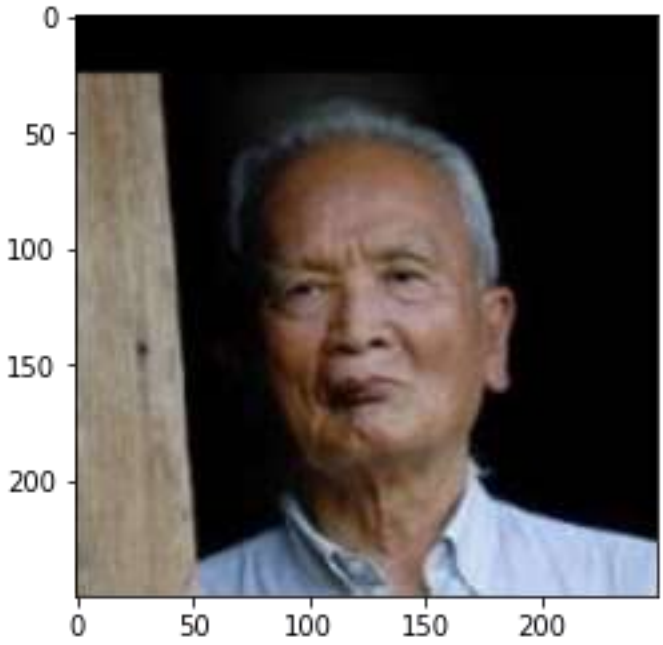
```
Epoch: 1        Training Loss: 4.772473        Validation Loss: 4.576998
Validation loss decreased (inf --> 4.576998).  Saving model ...
Epoch: 2        Training Loss: 4.474916        Validation Loss: 4.282395
Validation loss decreased (4.576998 --> 4.282395).  Saving model ...
Epoch: 3        Training Loss: 4.214767        Validation Loss: 4.009892
Validation loss decreased (4.282395 --> 4.009892).  Saving model ...
Epoch: 4        Training Loss: 3.963670        Validation Loss: 3.744841
Validation loss decreased (4.009892 --> 3.744841).  Saving model ...
Epoch: 5        Training Loss: 3.727339        Validation Loss: 3.497347
```

```
Validation loss decreased (3.744841 --> 3.497347).  Saving model ...
Epoch: 6       Training Loss: 3.510273       Validation Loss: 3.261775
Validation loss decreased (3.497347 --> 3.261775).  Saving model ...
Epoch: 7       Training Loss: 3.298942       Validation Loss: 3.059388
Validation loss decreased (3.261775 --> 3.059388).  Saving model ...
Epoch: 8       Training Loss: 3.114596       Validation Loss: 2.876079
Validation loss decreased (3.059388 --> 2.876079).  Saving model ...
Epoch: 9       Training Loss: 2.928063       Validation Loss: 2.674017
Validation loss decreased (2.876079 --> 2.674017).  Saving model ...
Epoch: 10      Training Loss: 2.768578       Validation Loss: 2.528656
Validation loss decreased (2.674017 --> 2.528656).  Saving model ...
Epoch: 11      Training Loss: 2.626950       Validation Loss: 2.382006
Validation loss decreased (2.528656 --> 2.382006).  Saving model ...
Epoch: 12      Training Loss: 2.482496       Validation Loss: 2.236146
Validation loss decreased (2.382006 --> 2.236146).  Saving model ...
Epoch: 13      Training Loss: 2.358243       Validation Loss: 2.113386
Validation loss decreased (2.236146 --> 2.113386).  Saving model ...
Epoch: 14      Training Loss: 2.243359       Validation Loss: 2.019116
Validation loss decreased (2.113386 --> 2.019116).  Saving model ...
Epoch: 15      Training Loss: 2.137082       Validation Loss: 1.920456
Validation loss decreased (2.019116 --> 1.920456).  Saving model ...
Epoch: 16      Training Loss: 2.035073       Validation Loss: 1.835387
Validation loss decreased (1.920456 --> 1.835387).  Saving model ...
Epoch: 17      Training Loss: 1.953583       Validation Loss: 1.724881
Validation loss decreased (1.835387 --> 1.724881).  Saving model ...
Epoch: 18      Training Loss: 1.871911       Validation Loss: 1.654590
Validation loss decreased (1.724881 --> 1.654590).  Saving model ...
Epoch: 19      Training Loss: 1.796170       Validation Loss: 1.588897
Validation loss decreased (1.654590 --> 1.588897).  Saving model ...
Epoch: 20      Training Loss: 1.725263       Validation Loss: 1.513451
Validation loss decreased (1.588897 --> 1.513451).  Saving model ...
```

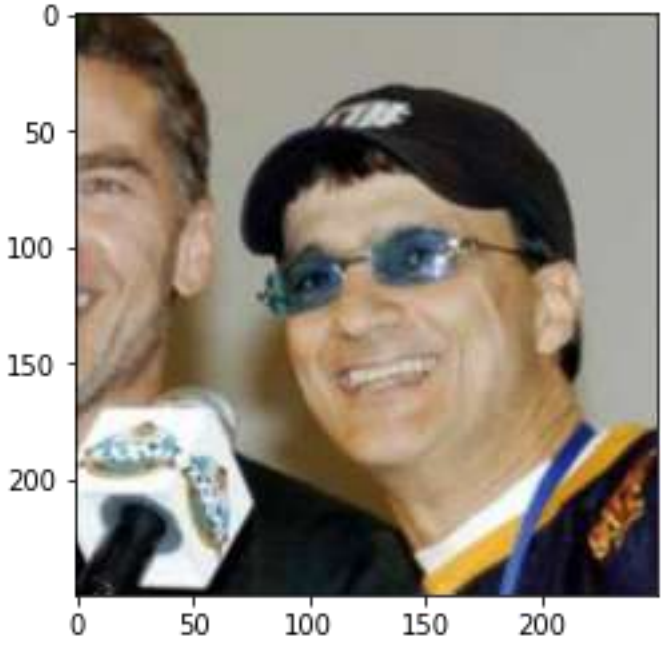
Out[42]:

```
<All keys matched successfully>
```

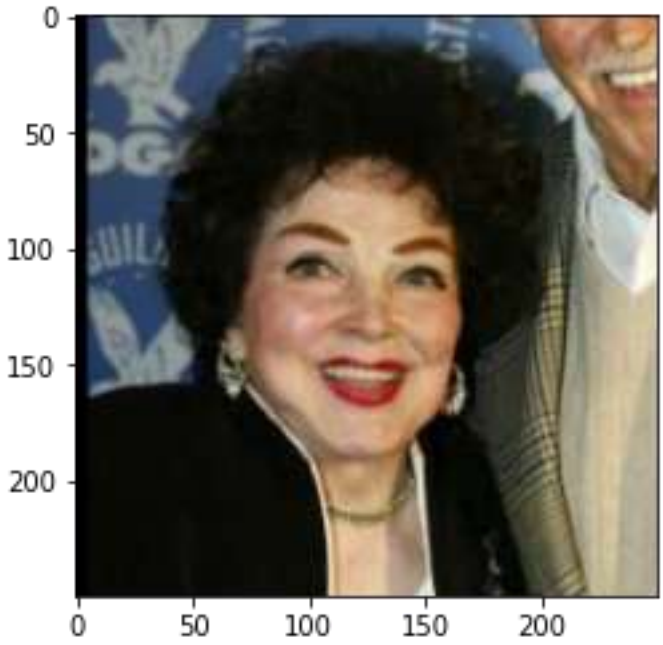
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.


%matplotlib inline
human_files_short1 = human_files[100:106]
dog_files_short1 = dog_files[100:106]

for file in np.hstack((human_files_short1, dog_files_short1)):
    run_app(file, train_data.classes)
```



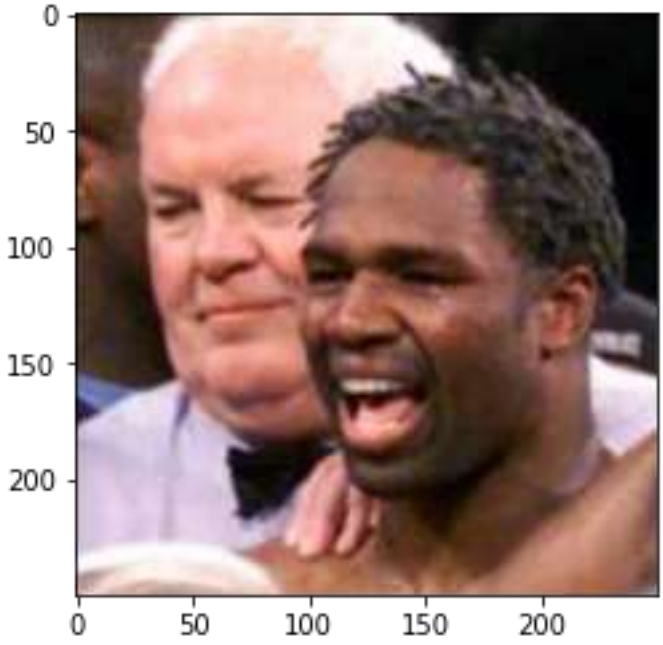

```
The image shows a human resembles 060.Dogue_de_bordeaux
```



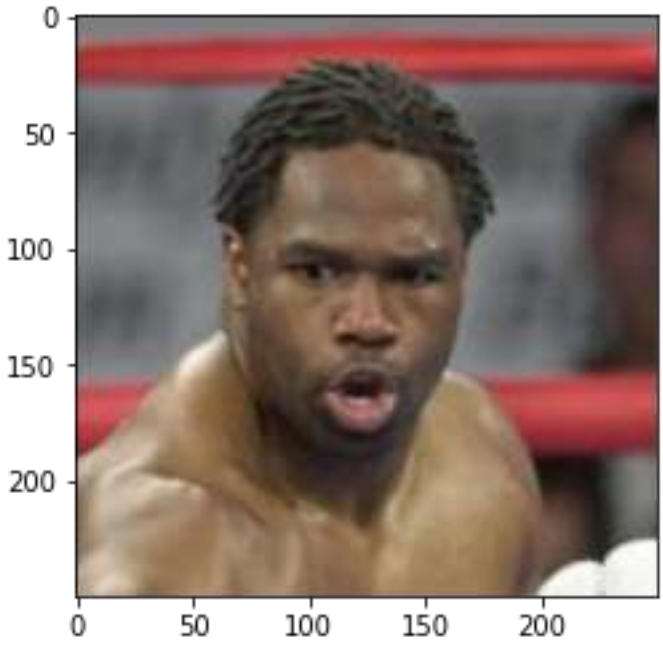

```
The image shows a human resembles 048.Chihuahua
```



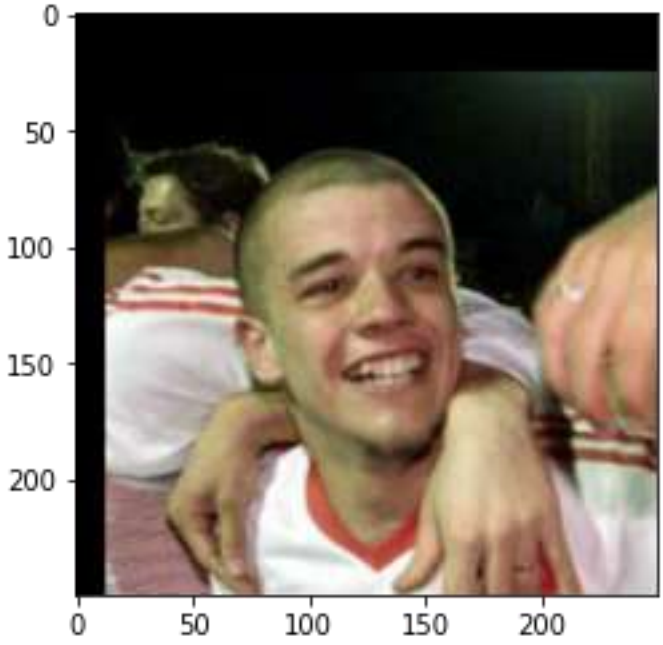

```
The image shows a human resembles 060.Dogue_de_bordeaux
```



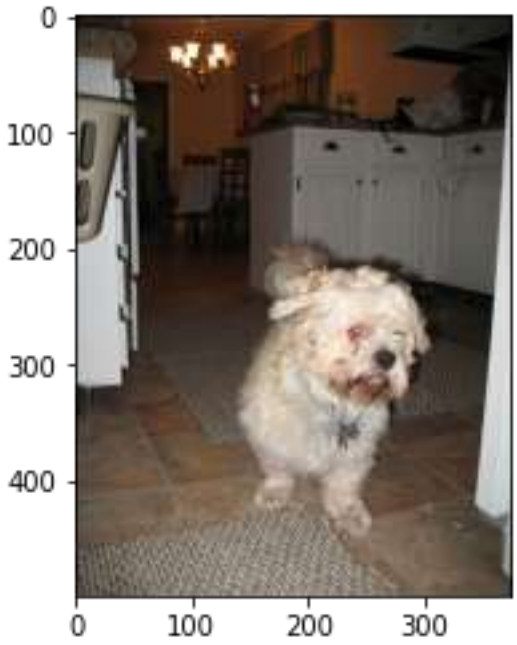

```
The image shows a human resembles 060.Dogue_de_bordeaux
```



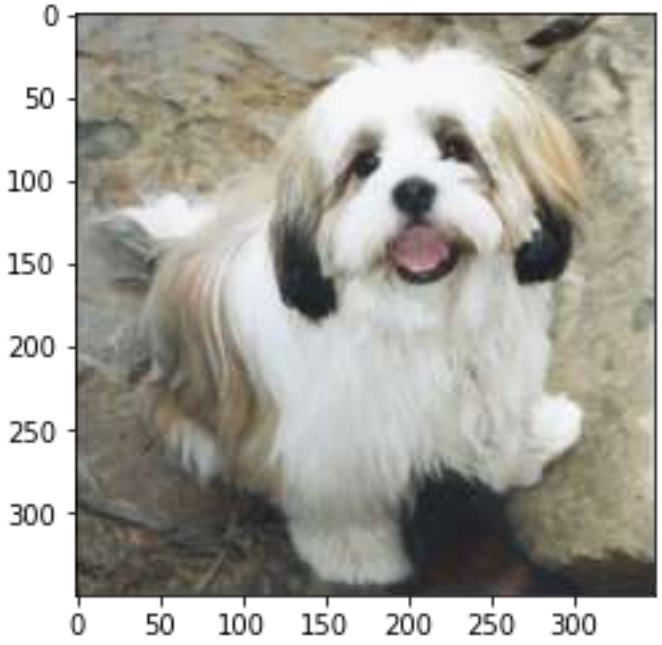

```
The image shows a human resembles 039.Bull_terrier
```



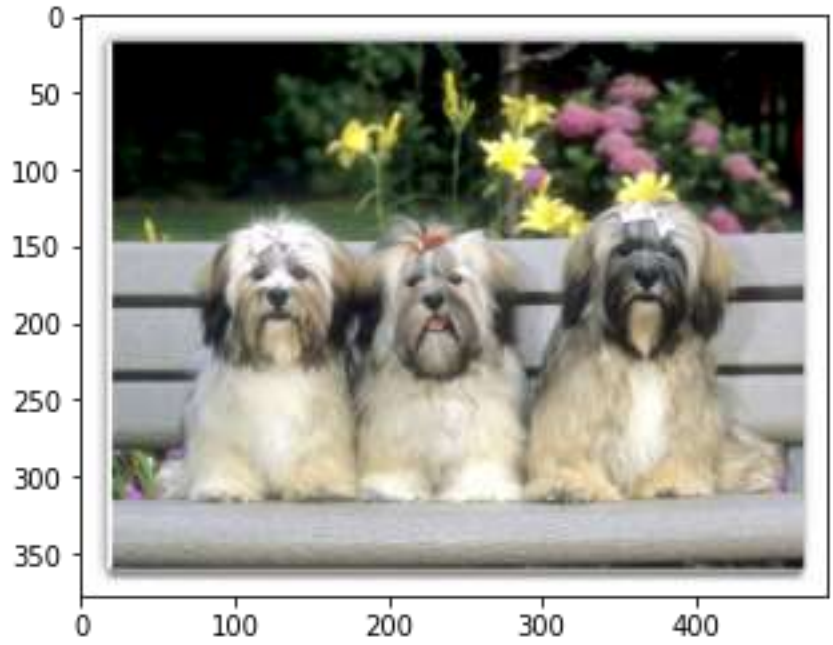

```
The image shows a human resembles 060.Dogue_de_bordeaux
```




```
The image shows a dog of 082.Havanese breed
```



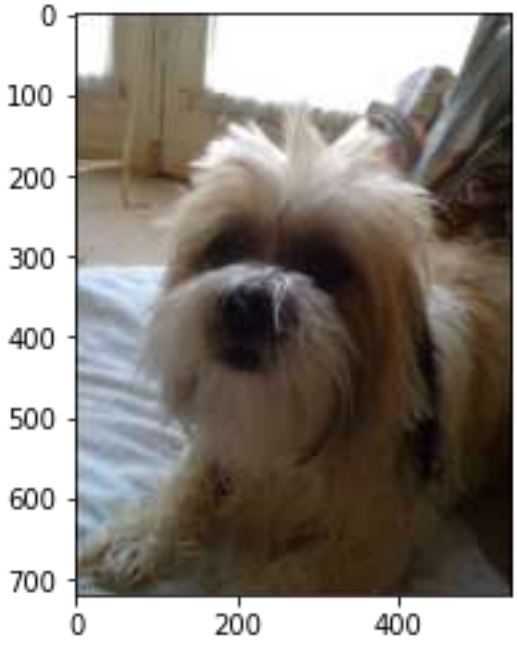

```
The image shows a dog of 082.Havanese breed
```




```
The image shows a dog of 017.Bearded_collie breed
```


```
The image shows a dog of 082.Havanese breed
```


```
The image shows a dog of 082.Havanese breed
```




```
The image shows a dog of 038.Brussels_griffon breed
```

Conclusion

The best prediction accuracy I have achieved is 77% and the best test loss score is ~1.51. These results are still a fair distance away from the proposed performance. I have concluded the following ways to potentially improve the accuracy of my implementation.

1. Given more training time and epochs, the accuracy can be improved.

2. Larger input dataset for training can improve the accuracy of the model.

3. Manipulation on the training images to make input dataset more robust.

4. Play with the layer structures and hyperparameter tuning.

5. Examine the relevance of a different pretrained model.

References

1. Paul Viola and Michael J. Jones. Robust real-time face detection. International Journal of Computer Vision, 57(2):137–154, 2004.

2. Rainer Lienhart and Jochen Maydt. An extended set of haar-like features for rapid object detection. In Image Processing. 2002. Proceedings. 2002 International Conference on, volume 1, pages I–900. IEEE, 2002.

3. ml-cheatsheet, https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html

4. VGG16 1000 categories, https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a

5. A Walkthrough of Convolutional Neural Network — Hyperparameter Tuning, https://towardsdatascience.com/a-walkthrough-of-convolutional-neural-network-7f474f91d7bd

6. Overview of convolutional neural network in image classification, https://analyticsindiamag.com/convolutional-neural-network-image-classification-overview/