Machine Learning Engineer Nanodegree

Capstone Project

Jinesh Mehta
November 25th, 2019

I. **Definition**

Project Overview

A lot of work has been done in the field of image recognition over the past few years. In this project a specific object recognition problem was examined. The goal is to build a model capable of identifying to which breed a dog belongs, based only on its photo. To train and evaluate the model the custom dataset provided by udacity, containing 13233 total human images and 8351 total dog images, was used.

Problem Statement

There are several factors that make the problem of dogs categorization challenging. Firstly, there many different breeds of dogs exist, and all breeds on a high level look alike, i.e. there can be only subtle differences in appearance between some breeds. In fact, a dog's breed might be not obvious right away even for people who have an expertise in the domain. Therefore, determination of dog breeds provides an excellent domain for fine grained visual categorization experiments. Secondly, dog images are very rich in their variety, showing dogs of all shapes, sizes, and colors, under differing illumination, in innumerable poses, and in just about any location. The photos have different resolutions, backgrounds, and scales. On some images the dogs are are partially covered by other objects or wear clothes such as hats, scarfs, glasses. Thus, there is a lot of noise on many of the photos that makes the problem more challenging. Thirdly, our data set is small in terms of the number of photos per breed.

If humans want to distinguish a dog from a cat, or from some other object, then they need to look at things like ears, whiskers, tails, tongues, fur textures, and so forth. These are the features that we (humans) use to discriminate. But none of these features are available to a computer, which receive only a matrix of independent integers as an input. Models such as Convolutional Neural Networks (CNN) allow computers to automatically extract hierarchies of features from raw pixels. These techniques proved to be successful in a variety of visual analysis tasks. Using these techniques, we can create a dog-breed classifier and allay that to solve our problem.

The complete solution consists of the following steps:

- **Import Datasets** : Our first step will be to load the datasets that are divided into train, validation and test folders.

- **Detect Humans** : In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.
- **Detect Dogs** : In this section, we use a pre-trained VGG-16 model to detect dogs in images.
- **Create and train a CNN to classify Dog Breeds (from scratch)** : In this step, we will create a CNN-based model from scratch that classifies dog breeds. In the next step, we will use an approach based on transfer learning.
- **Train a CNN to Classify Dog Breeds (via transfer learning)** : We will now use transfer learning to create a CNN that can identify dog breed from images. We will reuse the data loaders that we created earlier.
- **Dog breed classification algorithm** :
  - if a dog is detected in the image, return the predicted breed.
  - if a human is detected in the image, return the resembling dog breed.
  - if neither is detected in the image, provide output as other thing.
- **Testing the algorithm** : Lastly, We will test our algorithm on six sample images to measure the performance of our algorithm.

Metrics

For this particular project, we are only going to focus on an accuracy score. The goal of what we're looking to do here is pretty simple: we want to see how well we can do at classifying breeds of dogs. Accuracy will be able to tell us in a simple and easy-to-understand way how well our deep classification model is performing in this regard.

II. **Analysis**

Data Exploration

Following are the required human and dog datasets:

- To download the dog dataset
- To download the human dataset

Note: If you are using a Windows machine, you are encouraged to use 7zip to extract the folder.

Above custom dataset is provided by Udacity. It contains 13233 total human images and 8351 total dog images.

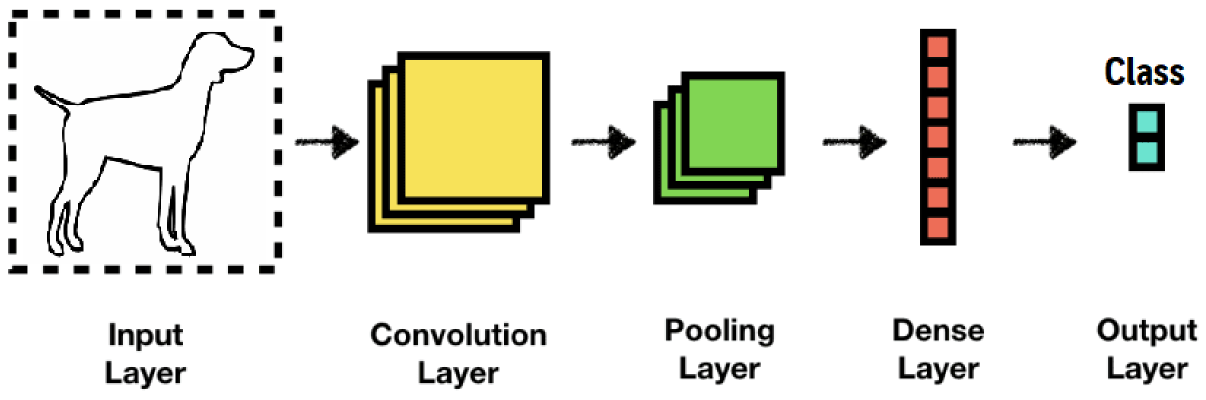
Algorithms and Techniques

In this section, we discuss the algorithms and techniques used for solving this problem.

Convolutional Neural Networks (CNNs)

A convolutional neural network is a class of deep, feed-forward artificial neural networks that has successfully been applied to analyzing visual imagery. CNNs take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. The layers of a

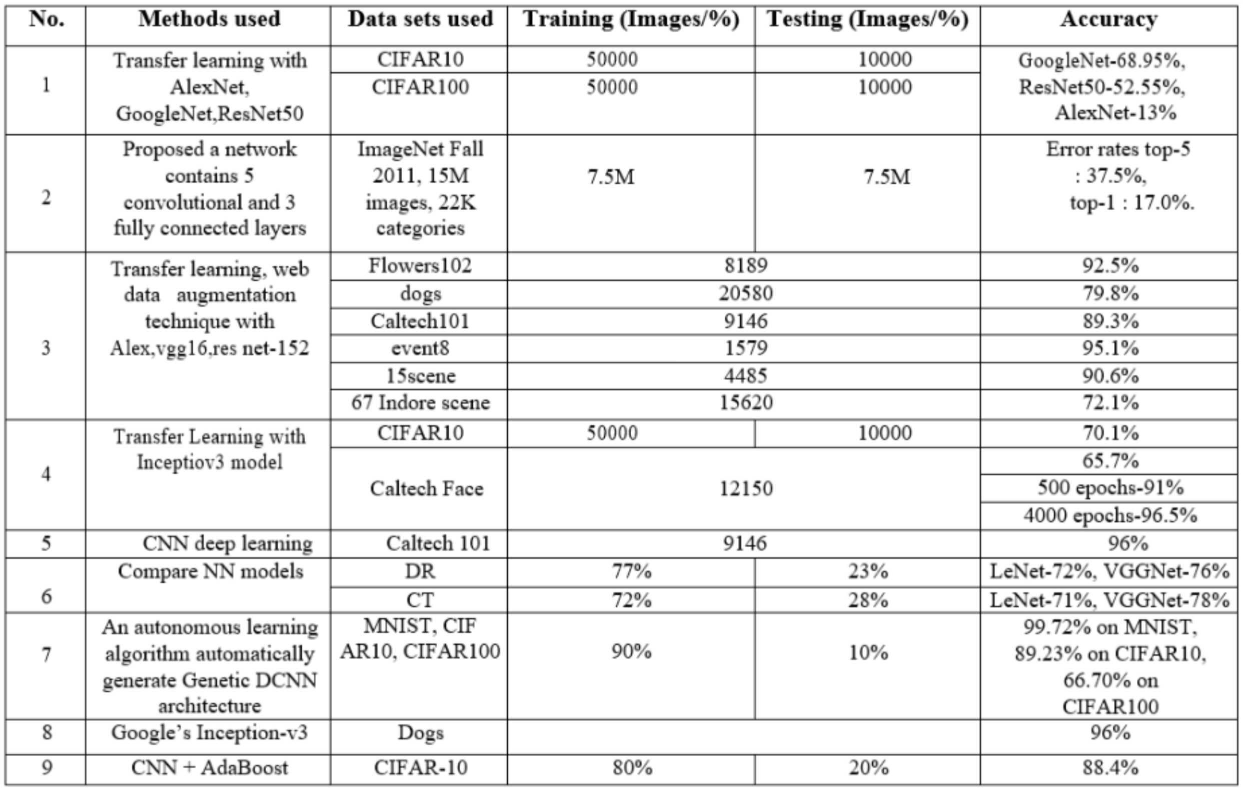
CNN have neurons arranged in 3 dimensions in so called convolution layers. In the convolution layer, the filters are passed across the input, row by row, and they activate when they see some type of visual feature. Now, rather than treating each of the raw pixel values in isolation, CNN's treat them in small local groups. These sliding filters are how the CNN can learn meaningful features and locate them in any part of the image.



These is only a high level overview of the CNN. There are also many other details related to other layers (pooling, relu, fully-connected, dropout), loss functions (cross-entropy, MSE, MAE, cosine similarity), optimizers (SGD, RMV, Adagrad), etc.

There is almost a linear relationship in the amount of data required and the size of the model. General idea is that the model should be large enough to capture relations in the data along with specifics of our problem. Early layers of the model capture high level relations between the different parts of the input (like edges and patterns), whereas later layers capture information that usually helps to discriminate between the desired outputs. Therefore if the complexity of the problem is high (like Image Classification) the number of parameters and the amount of data required is also large.

Benchmark



No.	Methods used	Data sets used	Training (Images/%)	Testing (Images/%)	Accuracy
1	Transfer learning with AlexNet, GoogleNet,ResNet50	CIFAR10	50000	10000	GoogleNet-68.95%, ResNet50-52.55%, AlexNet-13%
		CIFAR100	50000	10000	
2	Proposed a network contains 5 convolutional and 3 fully connected layers	ImageNet Fall 2011, 15M images, 22K categories	7.5M	7.5M	Error rates top-5 : 37.5%, top-1 : 17.0%.
3	Transfer learning, web data augmentation technique with Alex,vgg16,res net-152	Flowers102	8189		92.5%
		dogs	20580		79.8%
		Caltech101	9146		89.3%
		event8	1579		95.1%
		15scene	4485		90.6%
		67 Indore scene	15620		72.1%
4	Transfer Learning with Inceptiov3 model	CIFAR10	50000	10000	70.1%
		Caltech Face	12150		65.7%
					500 epochs-91%
					4000 epochs-96.5%
5	CNN deep learning	Caltech 101	9146		96%
6	Compare NN models	DR	77%	23%	LeNet-72%, VGGNet-76%
		CT	72%	28%	LeNet-71%, VGGNet-78%
7	An autonomous learning algorithm automatically generate Genetic DCNN architecture	MNIST, CIF AR10, CIFAR100	90%	10%	99.72% on MNIST, 89.23% on CIFAR10, 66.70% on CIFAR100
8	Google's Inception-v3	Dogs			96%
9	CNN + AdaBoost	CIFAR-10	80%	20%	88.4%

III. **Methodology**

Data Preprocessing

Before we can push our data through our algorithm, we'll have to pre-process it so that it appropriately conforms to the format we will need it in. Because we are using a TensorFlow-backed Keras CNN, this means that we will need to tranform the data appropriately. The images are resized by doing a crop of random size of the original size and a random aspect ratio of the original aspect ratio. This crop is finally resized to 224x224 pixels in order to be able to reuse the same data loaders for the transfer learning step.

Implementation

Architecture

```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
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.features = nn.Sequential(
            # 1st 2D convolution layer
            nn.Conv2d(3, 16, kernel_size=2, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Defining another 2D convolution layer
            nn.Conv2d(16, 32, kernel_size=2, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Defining another 2D convolution layer
            nn.Conv2d(32, 64, kernel_size=2, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Dropout(p=0.5),
            nn.AvgPool2d(kernel_size=2, stride=2)
        )

        self.classifier = nn.Sequential(
            nn.Linear(64 * 14 * 14, 133),
            nn.LogSoftmax(dim=1)
        )

    def forward(self, x):
        ## Define forward behavior
        out = self.features(x)
        out = out.view(-1, 64*14*14)
        out = self.classifier(out)

        return out


# instantiate the CNN
model_scratch = Net()

# move tensors to GPU if CUDA is available
if use_cuda:
    model_scratch.cuda()
```

The architecture is composed from a feature extractor and a classifier. The feature extractor is has 3 CNN layers in to extract features. Each CNN layer has a ReLU activation and a 2D max pooling layer in to reduce the amount of parameters and computation in the network. After the CNN layers we have a dropout layer with a probability of 0.5 in to prevent overfitting and an average pooling layer to calculate the average for each patch of the feature map. The classifier is a fully connected layer with an input shape of 64 x 14 x 14 (which matches the output from the average pooling layer) and 133 nodes, one for each class (there are 133 dog breeds). We add a softmax activation to get the probabilities for each class.

Training and Validation

For training and validating our model, we execute the following code snippet below :

```
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf

    for epoch in range(1, n_epochs+1):
        # initialize variables to monitor training and validation loss
        train_loss = 0.0
        valid_loss = 0.0

        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordingly
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data -

            #Set the parameter gradients to zero
            optimizer.zero_grad()

            #Forward pass, backward pass, optimize
            outputs = model(data)
            loss = criterion(outputs, target)
            loss.backward()
            optimizer.step()

            train_loss += ((1 / (batch_idx + 1)) * (loss.data - train_loss))

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            val_outputs = model(data)
            val_loss = criterion(val_outputs, target)
            valid_loss += ((1 / (batch_idx + 1)) * (val_loss.data - valid_loss


        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.fo
            epoch,
            train_loss,
            valid_loss
            ))
        ## TODO: save the model if validation loss has decreased
        if(valid_loss < valid_loss_min):
            print('Saving model: Validation Loss: {:.6f} decreased \tOld Valida
            valid_loss_min = valid_loss
            torch.save(model.state_dict(), save_path)

    # return trained model
    return model


# train the model
model_scratch = train(50, loaders_scratch, model_scratch, optimizer_scratch, cr

# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

Above Snippet resulted in the following output for 50 Epochs:

```
Epoch: 1     Training Loss: 4.804974     Validation Loss: 4.615381
Saving model: Validation Loss: 4.615381 decreased   Old Validation Loss: inf
Epoch: 2     Training Loss: 4.602738     Validation Loss: 4.448377
Saving model: Validation Loss: 4.448377 decreased   Old Validation Loss: 4.615
Epoch: 3     Training Loss: 4.491667     Validation Loss: 4.389207
Saving model: Validation Loss: 4.389207 decreased   Old Validation Loss: 4.448
Epoch: 4     Training Loss: 4.400146     Validation Loss: 4.396991
Epoch: 5     Training Loss: 4.352615     Validation Loss: 4.280646
Saving model: Validation Loss: 4.280646 decreased   Old Validation Loss: 4.389
Epoch: 6     Training Loss: 4.306862     Validation Loss: 4.252617
Saving model: Validation Loss: 4.252617 decreased   Old Validation Loss: 4.280
Epoch: 7     Training Loss: 4.269252     Validation Loss: 4.190484
Saving model: Validation Loss: 4.190484 decreased   Old Validation Loss: 4.252
Epoch: 8     Training Loss: 4.211070     Validation Loss: 4.275696
Epoch: 9     Training Loss: 4.174200     Validation Loss: 4.127232
Saving model: Validation Loss: 4.127232 decreased   Old Validation Loss: 4.190
Epoch: 10    Training Loss: 4.168478     Validation Loss: 4.148942
Epoch: 11    Training Loss: 4.141559     Validation Loss: 4.127430
Epoch: 12    Training Loss: 4.104400     Validation Loss: 4.143802
Epoch: 13    Training Loss: 4.087573     Validation Loss: 4.095686
Saving model: Validation Loss: 4.095686 decreased   Old Validation Loss: 4.127
Epoch: 14    Training Loss: 4.057649     Validation Loss: 4.024379
Saving model: Validation Loss: 4.024379 decreased   Old Validation Loss: 4.095
Epoch: 15    Training Loss: 4.028504     Validation Loss: 4.043189
Epoch: 16    Training Loss: 3.991086     Validation Loss: 3.963461
Saving model: Validation Loss: 3.963461 decreased   Old Validation Loss: 4.024
Epoch: 17    Training Loss: 3.983787     Validation Loss: 4.086076
Epoch: 18    Training Loss: 3.954470     Validation Loss: 4.004037
Epoch: 19    Training Loss: 3.912849     Validation Loss: 3.975426
Epoch: 20    Training Loss: 3.898360     Validation Loss: 4.016487
Epoch: 21    Training Loss: 3.867416     Validation Loss: 4.000227
Epoch: 22    Training Loss: 3.868152     Validation Loss: 3.974565
Epoch: 23    Training Loss: 3.823937     Validation Loss: 3.960703
```

```
Saving model: Validation Loss: 3.960703 decreased   Old Validation Loss: 3.963
Epoch: 24   Training Loss: 3.837263     Validation Loss: 3.966237
Epoch: 25   Training Loss: 3.807810     Validation Loss: 3.894585
Saving model: Validation Loss: 3.894585 decreased   Old Validation Loss: 3.960
Epoch: 26   Training Loss: 3.792684     Validation Loss: 3.880291
Saving model: Validation Loss: 3.880291 decreased   Old Validation Loss: 3.894
Epoch: 27   Training Loss: 3.792550     Validation Loss: 3.912843
Epoch: 28   Training Loss: 3.755440     Validation Loss: 3.997811
Epoch: 29   Training Loss: 3.753458     Validation Loss: 3.870937
Saving model: Validation Loss: 3.870937 decreased   Old Validation Loss: 3.880
Epoch: 30   Training Loss: 3.726488     Validation Loss: 3.869637
Saving model: Validation Loss: 3.869637 decreased   Old Validation Loss: 3.870
Epoch: 31   Training Loss: 3.712027     Validation Loss: 3.860836
Saving model: Validation Loss: 3.860836 decreased   Old Validation Loss: 3.869
Epoch: 32   Training Loss: 3.728802     Validation Loss: 3.894773
Epoch: 33   Training Loss: 3.674767     Validation Loss: 3.964670
Epoch: 34   Training Loss: 3.669846     Validation Loss: 3.895880
Epoch: 35   Training Loss: 3.638517     Validation Loss: 3.862057
Epoch: 36   Training Loss: 3.658729     Validation Loss: 3.877197
Epoch: 37   Training Loss: 3.623693     Validation Loss: 4.010448
Epoch: 38   Training Loss: 3.600378     Validation Loss: 3.906092
Epoch: 39   Training Loss: 3.628024     Validation Loss: 3.942724
Epoch: 40   Training Loss: 3.582008     Validation Loss: 3.789247
Saving model: Validation Loss: 3.789247 decreased   Old Validation Loss: 3.860
Epoch: 41   Training Loss: 3.595245     Validation Loss: 3.810999
Epoch: 42   Training Loss: 3.592308     Validation Loss: 3.847783
Epoch: 43   Training Loss: 3.580878     Validation Loss: 3.771963
Saving model: Validation Loss: 3.771963 decreased   Old Validation Loss: 3.789
Epoch: 44   Training Loss: 3.567358     Validation Loss: 3.806099
Epoch: 45   Training Loss: 3.548939     Validation Loss: 3.923195
Epoch: 46   Training Loss: 3.555041     Validation Loss: 3.810476
Epoch: 47   Training Loss: 3.537040     Validation Loss: 3.901146
Epoch: 48   Training Loss: 3.551056     Validation Loss: 3.740501
Saving model: Validation Loss: 3.740501 decreased   Old Validation Loss: 3.771
Epoch: 49   Training Loss: 3.520532     Validation Loss: 3.980445
Epoch: 50   Training Loss: 3.512998     Validation Loss: 4.055543
```

Testing

Now we try out our model on the test dataset of dog images. Use the code snippet below to calculate the test loss and accuracy.

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

    # monitor test loss and accuracy
    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 mode
            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().
            total += data.size(0)

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

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

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

Above code provides the following output which indicates a loss of 3.65 and an accuracy of 15% which is very less and hence a refinement is needed.

```
Test Loss: 3.654766

Test Accuracy: 15% (133/836)
```

Refinement

Since creating a CNN from scratch didn't do too great, we'll leverage one of the architectures packaged up nicely for us from the options between VGG-16, VGG-19, ResNet-50, InceptionV3, and Xception. We use the VGG16 network that was pretrained on the ImageNet dataset. The ImageNet dataset containes similar images with our own dataset so we can keep the feature extractor part. Because we have a small dataset and we don't need to identify dogs but dog breeds we replace the classifier with a fully connected layer with 1024 nodes, with ReLU and a dropout with a 0.4 probability, connected to an output layer with 133 nodes (same as the number of classes). We then train the model but we freeze the parameters for the feature extractor so only the classifier paramters get backpropagated.

Architecture

We use transfer learning to create a CNN to classify dog breed by executing the following code snippet below:

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

model_transfer = models.vgg16(pretrained=True)

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

classifier = nn.Sequential(
    nn.Linear(25088, 1024),
    nn.ReLU(inplace=True),
    nn.Dropout(p=0.4),
    nn.Linear(1024, 133),
    nn.LogSoftmax(dim=1)
)

model_transfer.classifier = classifier

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

Training and Validation

For training and validating our model, we execute the following code snippet below :

```
# train the model
model_transfer = train(10, loaders_transfer, model_transfer, optimizer_transfer

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

Above Snippet resulted in the following output for 10 Epochs:

```
Epoch: 1    Training Loss: 4.126252     Validation Loss: 1.948809
Saving model: Validation Loss: 1.948809 decreased   Old Validation Loss: inf
Epoch: 2    Training Loss: 2.914006     Validation Loss: 1.389281
Saving model: Validation Loss: 1.389281 decreased   Old Validation Loss: 1.9488
Epoch: 3    Training Loss: 2.662741     Validation Loss: 1.194920
Saving model: Validation Loss: 1.194920 decreased   Old Validation Loss: 1.3892
Epoch: 4    Training Loss: 2.601928     Validation Loss: 1.202487
Epoch: 5    Training Loss: 2.523065     Validation Loss: 1.224847
Epoch: 6    Training Loss: 2.514232     Validation Loss: 1.044022
Saving model: Validation Loss: 1.044022 decreased   Old Validation Loss: 1.1949
Epoch: 7    Training Loss: 2.434076     Validation Loss: 1.065839
Epoch: 8    Training Loss: 2.478743     Validation Loss: 1.210468
Epoch: 9    Training Loss: 2.429160     Validation Loss: 1.097689
Epoch: 10   Training Loss: 2.413055     Validation Loss: 1.134307
```

Testing

Now we again try out our model on the test dataset of dog images. Use the code snippet below to calculate the test loss and accuracy :

```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

Above code provides the following output which indicates a loss of 1.08 and an accuracy of 66% which is much better than our previous approach.

```
Test Loss: 1.081452

Test Accuracy: 66% (560/836)
```

IV.**Experiments and Results**

Model Evaluation and Validation

Now that we've created a model architecture using transfer learning, I'll test out my model on both some dogs as well as other random images to see how it necessarily performed. For that, I have created an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then,

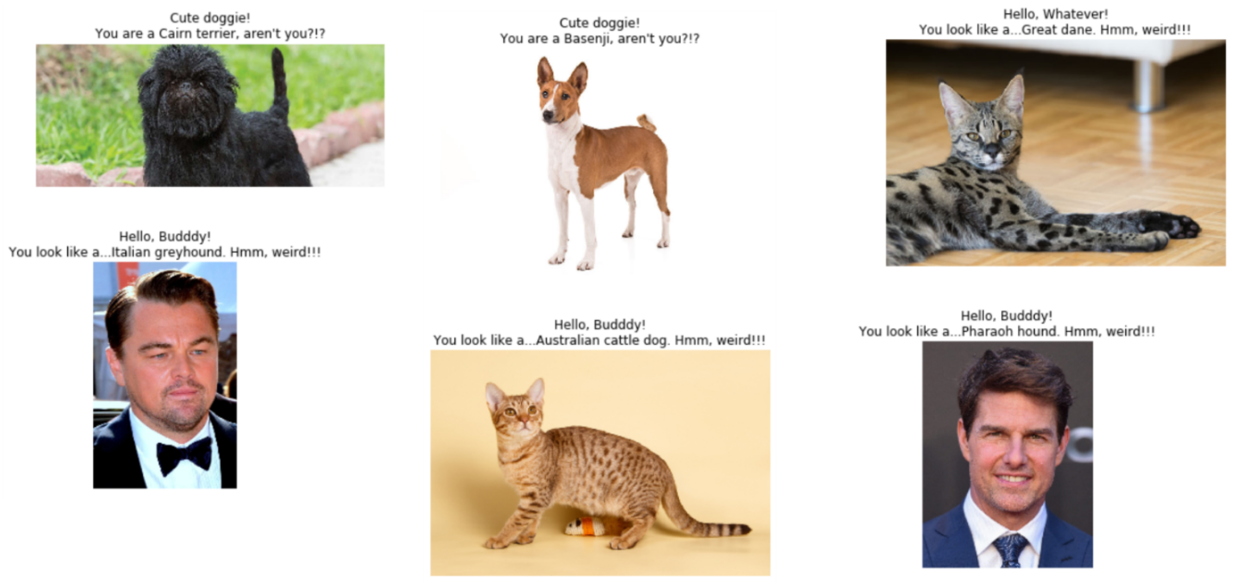
- if a dog is detected in the image, return the predicted breed.
- if a human is detected in the image, return the resembling dog breed.
- if neither is detected in the image, provide output that indicates an error.

The code snippet for the above mentioned process is as follows :

```
def process_image(image):
    ''' Scales, crops, and normalizes a PIL image for a PyTorch model,
        returns an Numpy array
    '''

    # scale
    scale_size = 256,256
    image.thumbnail(scale_size, Image.LANCZOS)

    # crop
    crop_size = 224
    width, height = image.size   # Get dimensions

    left = (width - crop_size)/2
    top = (height - crop_size)/2
    right = (width + crop_size)/2
```

```
        bottom = (height + crop_size)/2

        image = image.crop((left, top, right, bottom))

        # normalize
        mean = np.array([0.485, 0.456, 0.406])
        std = np.array([0.229, 0.224, 0.225])
        image_array = np.array(image) / 255

        image = (image_array - mean) / std

        # reorder dimensions
        image = image.transpose((2,0,1))

        return torch.from_numpy(image)

    def predict_breed_transfer(img_path):
        # load the image and return the predicted breed
            # open image
        image = Image.open(img_path)
        image = process_image(image)
        image = image.unsqueeze_(0)
        if use_cuda:
            image = image.cuda().float()

        model_transfer.eval()

        with torch.no_grad():
            output = model_transfer(image)
            # convert output probabilities to predicted class
            pred = output.data.max(1, keepdim=True)[1]

        return class_names[pred]
    def run_app(img_path):
        ## handle cases for a human face, dog, and neither
        is_dog = False
        if dog_detector(img_path):
            is_dog = True
            title = 'Cute doggie!'
        elif face_detector(img_path):
            title = "Hello, Budddy!"
        else:
            title = "Hello, Whatever!"


        breed = predict_breed_transfer(img_path)

        if is_dog:
            sub_title = f'You are a {breed}, aren\'t you?!?'
        else:
            sub_title = f'You look like a...{breed}. Hmm, weird!!!'

        image = Image.open(img_path)
        plt.imshow(image, interpolation='nearest')
        plt.axis('off')
        plt.title(f'{title}\n{sub_title}')
        plt.show()
        plt.close()

    images = [
        'images/test/dog_affenpinscher.jpg',
        'images/test/Leonardo_Dicaprio.jpg',
        'images/test/Basenji.jpg',
        'images/test/ocicat.jpg',
        'images/test/savannah.jpg',
        'images/test/tom_cruise.jpg']
    for file in images:
        run_app(file)
```

As shown above in the code snippet, we use 6 images (combination of cat, dog and human images) for testing the whole algorithm.


Cute doggie!
You are a Cairn terrier, aren't you?!?
Cute doggie!
You are a Basenji, aren't you?!?
Hello, Whatever!
You look like a...Great dane. Hmm, weird!!!
Hello, Budddy!
You look like a...Italian greyhound. Hmm, weird!!!
Hello, Budddy!
You look like a...Australian cattle dog. Hmm, weird!!!
Hello, Budddy!
You look like a...Pharaoh hound. Hmm, weird!!!

Test results from the proposed algorithm

When we got to testing out human images, that's when things started to get interesting. The classifier did correctly predict that every human image I tested out was a human image, but the type of dog breed it guessed the human to be produced some interesting results. However, there are certain classification where the algorithm failed things to guess which can be seen in test results.

V. **Conclusion**

In this work we tried to tackle the dog breed identification problem using a very small dataset with only a few dozens of images per breed. First, a small convolutional neural network was built and trained on the dataset from scratch, and the accuracy of 15% was achieved on the testing dataset. Then this result was further improved by applying VGG-16 using transfer learning. The testing accuracy thereafter improved to 66%. This result is very impressive considering the complexity of the problem, and the limited amount of data.

Even though at first glance the dog breed identification seems to be a very challenging problem, it was shown that a powerful and highly accurate CNN-based image classification model can be built with help of transfer learning.

Improvement

This project took one relatively simple approach to creating a CNN, but there a number of ways in which we could have improved the effort. Here are three things we could have altered along the way:

- **More training data**: Relatively speaking, we trained our model on a very low number of images. Most of these highly complex architectures are trained on vastly more images than our training dataset, so additional training data would almost certainly help to reinforce the outcomes we would expect.
- **Altering the CNN's architecture**: This one is harder to know, but given that the architectures like VGG or the Inception one used here contained many layers, it's feasible to think that adding more layers / adjusting the existing layers would produce better outputs.
- **Adjusting the transfer learning architecture**: For this project, I arbitrarily selected VGG16 as the architecture for our transfer learning. I didn't try any of the others like Xception, and it could have happened that they would have been a better fit for our algorithm.

Reflection

I've already stated this above, but clearly transfer learning is the way to go if an option. It helped our model to perform much better, and it was more efficient computationally. Computational efficiency is starting to emerge as a key factor in environmental friendliness. Also, another point to consider is the amount of energy it takes to run these deep learning models is astoundingly high. If we all try creating deep learning models from scratch, we contribute to this problem. Transfer learning helps to cut down these environmental concerns by sharing knowledge and cutting down energy consumption.

References

- For Problem Statement Brief
- For Template & Structure