

Machine Learning Engineer Nanodegree - Capstone Project

Xin Tao

Part I Definition

Project Overview

How well can a human classify a dog's breed. Let's try out [1]!



It might sound easy for humans to identify a dog breed, but we can't guarantee we have 100% accuracy of identifying them. Some dogs of different breeds look very similar just by their appearance. DNA test is the most accurate way to identify the breed. A professional can have an accuracy above 90%. For regular people, like me, the accuracy can range from 10%-80% depending on knowledge about dogs.

The problem we described above is a typical image classification problem. Image classification refers to the task of identifying or predicting the category of an image based on its visual content. Like other classification problems, it can be a binary, multi-class or multi-label classification. For example, given an image, an example of binary classification is to identify whether the image contains a dog or not. Among all the known breeds of dogs, classifying the breed of a dog is a multi-class classification problem. If a picture has multiple objects, detecting what objects it has is a multi-label task.

How can a machine solve the problem? Up to date, the most efficient and high performance technique is using Deep Learning algorithms. Specifically, Convolutional Neural Network (CNN) has been widely proven to have high performance in computer vision related tasks. CNN is a type of deep learning neural network, which has brought breakthroughs in image classifications in the past few years. Images have high dimensionalities as each pixel is considered a feature. As the number of layers grows, the total number of parameters in the neural network explodes rapidly, which makes it impossible to train a model with regular computational power. CNN can effectively reduce the number of parameters without compromising the quality of the model. The dimensions are reduced in CNN by a sliding window (called filter or kernel) with a size less than input image[2]. As the window sliding across the whole image matrix, it extracts the features such as edges and colors from the input image but with a reduction in its

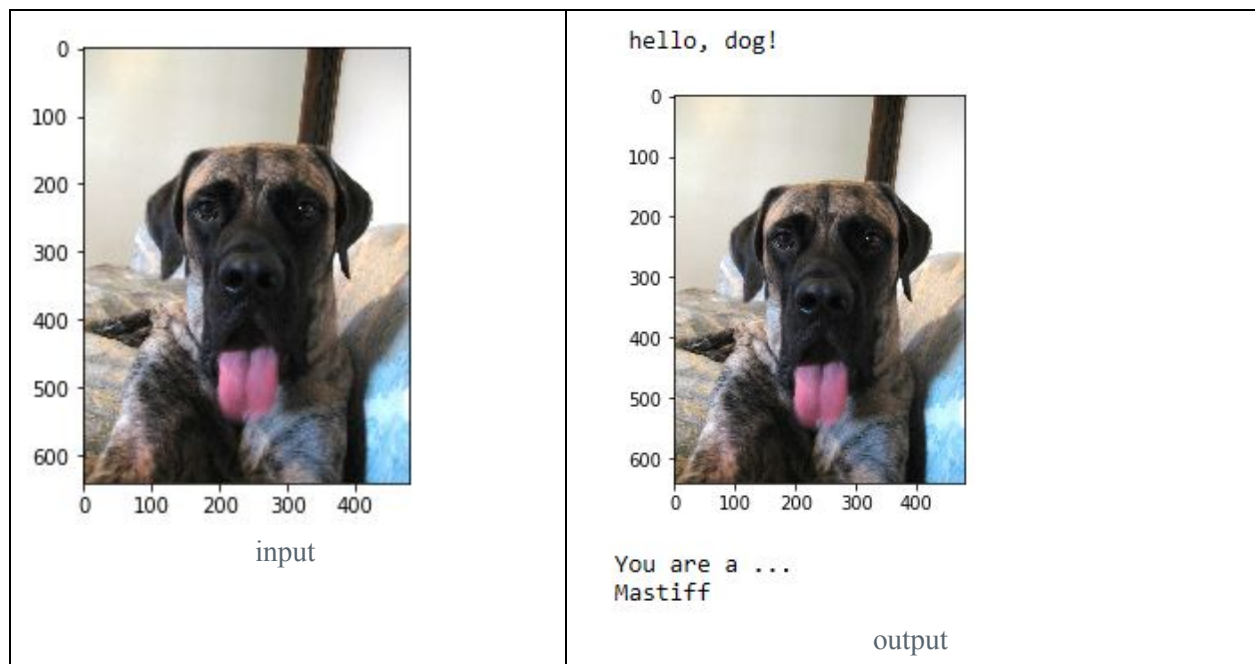
dimensionality. As the network becomes deep, it extracts features from low level to high level. For the above reason, CNN is very suitable for image classification problems.

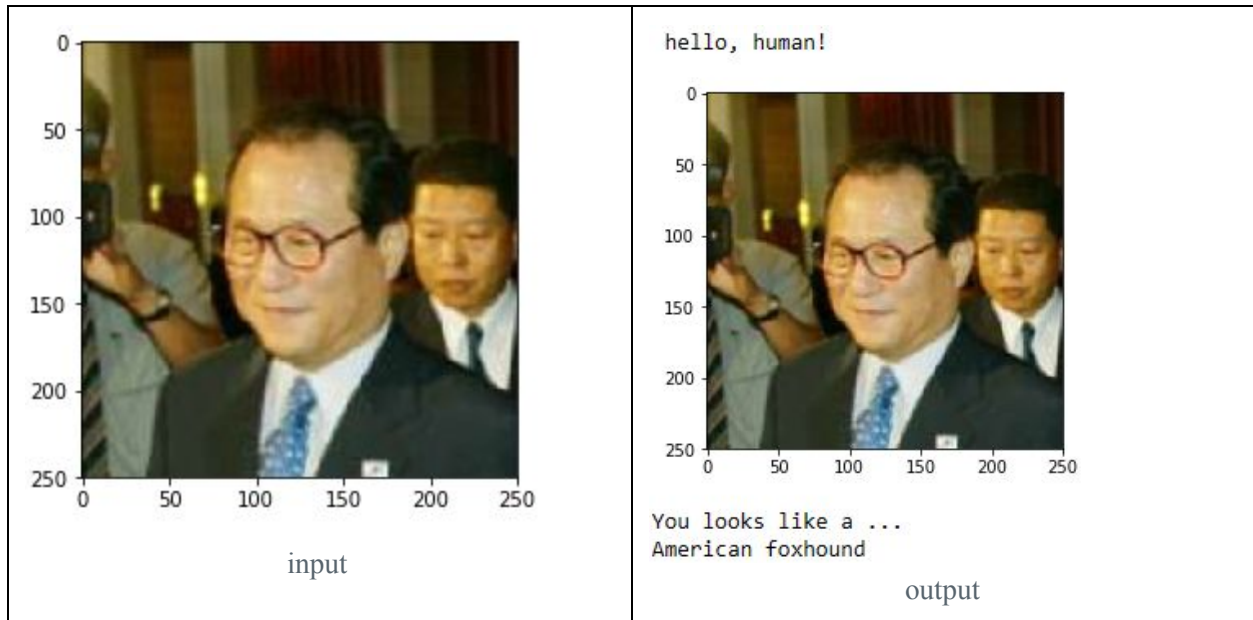
Problem Statement

In this project, I will be working on the “CNN Project: Dog Breed Classifier ” [3] provided by Udacity. This is a classic multi-class image classification problem. The goal of this project is to develop an app that will identify an estimate of the canine’s breed, given an image of a dog. If supplied an image of a human, the code will identify the resembling dog breed. To be successful in this project, I must build a machine learning model that can beat average human accuracy of identifying dog’s breed. The task has two major functionalities to achieve:

- 1) Tell if an image is a human or a dog. If it is a human, tells an estimate of the dog breed that is most resembling
- 2) If the image is a dog image, it will provide an estimate of the dog’s breed

Examples of input and output expected from this app is shown below.





Regarding the strategy for solving the problem and expected solution, Convolutional Neural Network (CNN) has been proven to achieve very high performance on image classification tasks. Specifically, I will use variants of **CNN models**, such as ResNet, which has very good performance to train deep networks and extract more high-level features from input data.

Metrics

I will primarily use **test accuracy** to evaluate the model. A good model will achieve a test accuracy above 60%. Also the Transfer model which trained based on larger amounts of data should outperform the benchmark model (created from scratch). The accuracy is calculated by the number of correctly labeled images divided by the total number of images in the test set.

Apart from test accuracy, another metric to look at is the **Cross-entropy loss**. The Cross-entropy loss combines LogSoftmax and NLLLoss in one single class [7], which is very reliable for multi-class problems, especially when you have an unbalanced training set. The loss increases as the predicted probability diverges from the actual label. The loss will not directly tell how many images are predicted correctly, but it will be very useful in the training stage. If we see the loss value is small and continue decreasing for both train and valid dataset, that will be a good sign that the model will succeed.

Part II Analysis

Data Exploration

The datasets are provided by Udacity, including two different folders containing human dataset and dog dataset. The dog datasets are further split into train, valid and test datasets for machine learning tasks. There are 13233 total human images and 8351 total dog images. Among the dog images, the train/valid/test ratio is 6680/836/836. There are 133 breeds in each dataset stored in subfolders.

```
import numpy as np
from glob import glob

# Load filenames for human and dog images
human_files = np.array(glob("/data/lfw/*/"))
dog_files = np.array(glob("/data/dog_images/*/"))

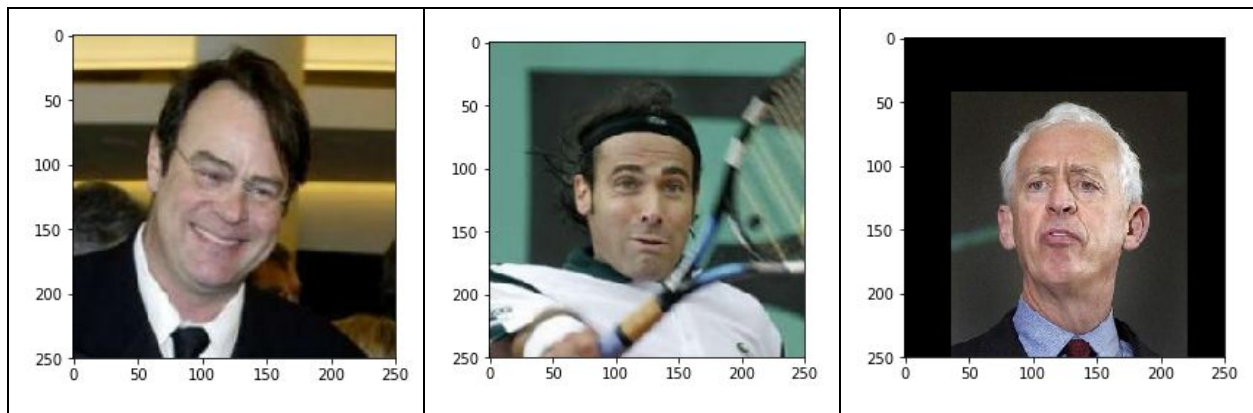
# print number of images in each dataset
print('There are %d total human images.' % len(human_files))
print('There are %d total dog images.' % len(dog_files))
```

```
There are 13233 total human images.
There are 8351 total dog images.
```

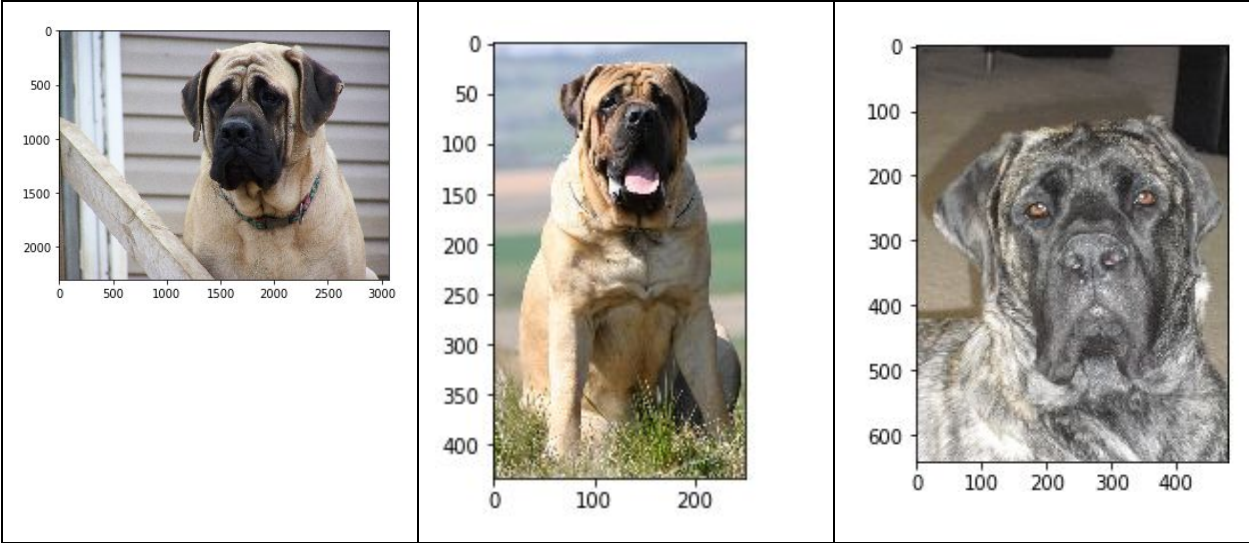
The input data types are all images, which will be later converted to tensors for training to feed to the model. The images have different size, color, resolutions, and angles, which need data preprocessing. The use of the input datasets is very appropriate for CNN image classification because they are raw images with 3 channels, perfect to feed into VGG and ResNet.

Exploratory Visualization

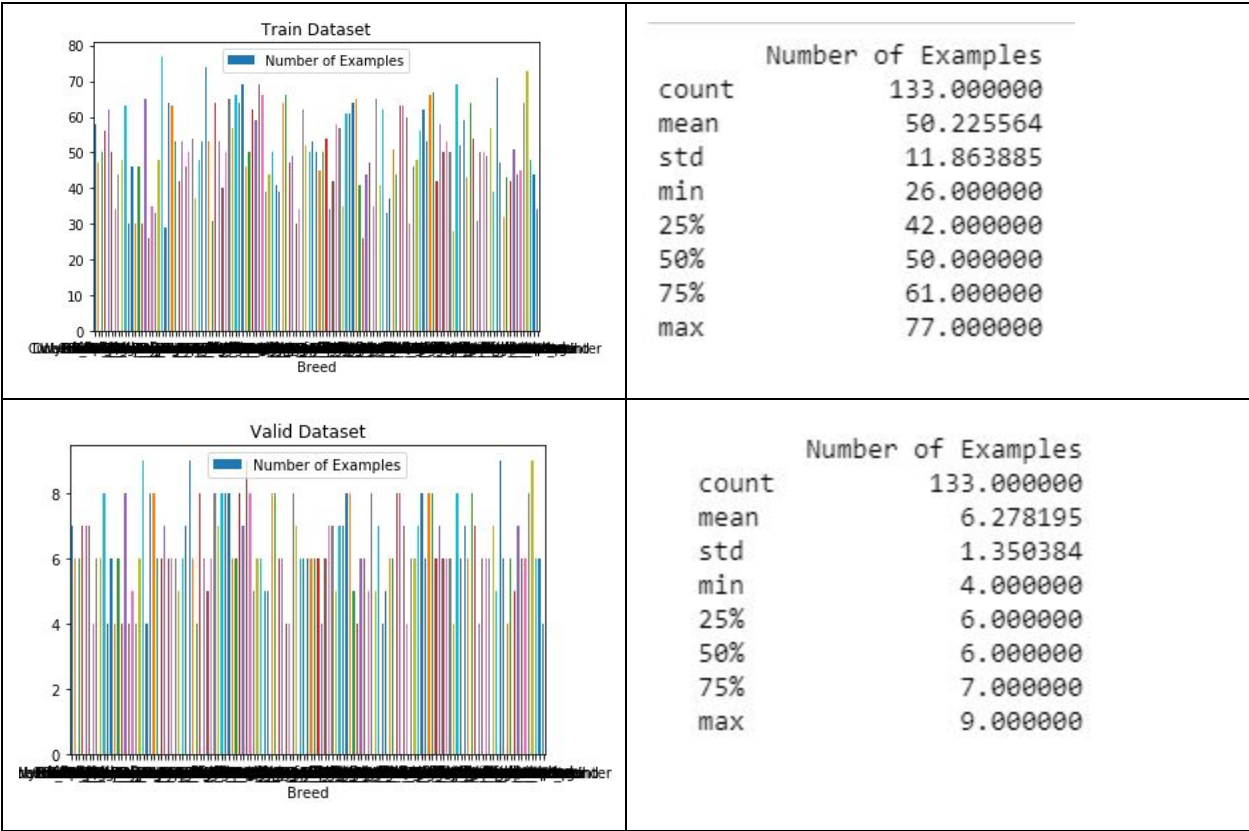
Below are a few examples of images from human and dog folders. The dataset of dogs is unbalanced because some breeds have more images than others. A distribution of the dog's breeds in train/valid/test dataset is also shown below.

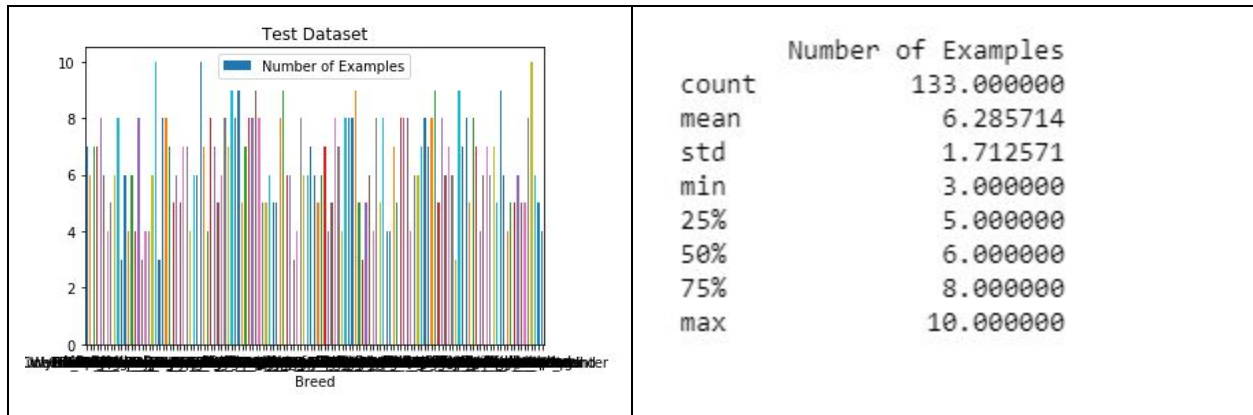


Human images



Dog images





Distribution of dog's breeds

Algorithms and Techniques

The solution will have two steps. First I will detect human faces in images using OpenCV's implementation of **Haar feature-based cascade classifiers** [4]. Second, I will use a pre-trained **VGG-16** model to detect dogs images. Third, I will develop the CNN model to predict the dog's breed both from scratch and from transfer learning.

Particularly, The CNN model from scratch will be built using a modified ResNet CNN Architecture, with parameters tuned to be appropriate to the input datasets. Deep Residual Network (ResNet) has been one of the most groundbreaking algorithms in the Deep Learning community in the past few years, especially in the field of object detection and face recognition. ResNet makes it possible to train hundreds or even thousands of layers with high compelling performance. [5]

For the transfer learning model, I will use the most recent pretrained **ResNet152** Architecture. I will modify the FC layer parameters to match the dimension of our case. ResNet152 is suitable for our problem because it has the deepest layers in all ResNET architecture so far. When the layer gets deeper, it can extract more relevant features (edges, shapes, colors, etc) and thus can make a better generalization/prediction. Though it is deep, each layer is relatively simple. ResNET152 is pre-trained on the ImageNet with a depth of 152 layers -- 8x deeper than VGG nets but still having lower complexity. An ensemble of these residual nets achieves 3.57% error on the ImageNet test set [6]. ResNet152 makes the solution qualifiable (parameter to tune, loss function to use), measureable (evaluation method to use) and replicable (reliable result).

Benchmark

The CNN model created from scratch is a good benchmark model. It will be trained on the datasets provided with limited available images (8351 total dog images). A random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%. For the model created from scratch, I will use 10% as a success criteria. Later the pre-trained transfer model, built up on a lot more training data, should perform much better than the benchmark model.

Part III Methodology

Data Preprocessing

As mentioned above, the data preprocessing is necessary to convert input image to the dimensionalities and data types that are compatible with the model. Also it serves to make the images more comparable to each other to facilitate training and reduce overfitting. Specifically, Four types of data preprocessing is used. A snippet of the code used for data preprocessing is also shown below.

- 1) **Convert input to Tensor** to be accepted into the model. Dimensionality change is not required in this case since all input images are 3 channel image files. However, if using my own images that have 4 channels or 1 channel, a dimension change is needed.
- 2) **Image resize and crop.** All input images are resized/cropped to (224,224,3) to make them consistent. All images need to be resized to the same fixed size before feeding to CNN. If resize is not done correctly, it can cause loss of information (e.g. inappropriate cropping) or deformation of features (too much scaling) [6].
- 3) **Normalize the images.** This is to ensure each input parameter (i.e. pixel) has a similar data distribution, which makes convergence faster while training the network [7].
- 4) **Input Augmentation.** Here, random rotation and random flip are used to expose the neural network to a wide variety of variations, which makes the CNN model more robust and reduces overfitting [7].

```
normalize=transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
transformed=transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.RandomRotation(15),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.ToTensor(),
    normalize])
```

Implementation

The implementation consist of 6 major steps, as listed below :

Step 0: Import Datasets: download the required human and dog datasets and store it in the project's home directory. The file paths are saved in the numpy arrays for further calls.

```
# Load filenames for human and dog images
human_files = np.array(glob("/data/lfw/*/"))
dog_files = np.array(glob("/data/dog_images/*/"))
```

Step 1: Detect Humans: use OpenCV's implementation to detect human faces in images. OpenCV provides pre-trained human face detectors. We will cv2.CascadeClassifier for this purpose.

```
# 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
```

Step 2: Detect Dogs: use a pre-trained VGG-16 model to detect dogs in images. It returns a prediction of the category of the object (in the image) out of 1000 possible categories in ImageNet.

```

### 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)
    result=True if (151<=pred<=268) else False

    return result # true/false

```

Step 3: Create a CNN to Classify Dog Breeds from Scratch. The goal is to achieve accuracy of more than 10%. First step is to write three separate data loaders for the training, validation and test datasets of dog images. Image Augmentations and transformations are performed as mentioned in the data preprocessing step. The result will be a dictionary of data loaders for train/valid/test datasets.

```

train_data=datasets.ImageFolder('/data/dog_images/train', transformed)
train_loader=torch.utils.data.DataLoader(train_data, batch_size=32, shuffle=True, num_workers=0)

valid_data=datasets.ImageFolder('/data/dog_images/valid', transformed)
valid_loader=torch.utils.data.DataLoader(valid_data, batch_size=32, shuffle=True, num_workers=0)

test_data=datasets.ImageFolder('/data/dog_images/test', transformed)
test_loader=torch.utils.data.DataLoader(test_data, batch_size=32, shuffle=True, num_workers=0)

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

```

Next step is to define the CNN model architecture from scratch. I referenced a ResNet CNN Architecture outlined in Pytorch git repo [8], and modified according to our case. First it's input undergoes a convolution layer, a batch normalization, a ReLU and a Max Pool. Then the output undergoes four layers, an avgpool layer and a FC layer. Its basicblock (in each layer) consists of convolution block 1 -> batch normalization 1 -> ReLU -> Convolution block 2 -> batch normalization 2. If input and output channels are different, it adds a downsample (Convolution block -> Batch normalization) in its basic blocks. The initial kernel size are chosen to be 7 with stride 2 and padding 3, in order to enable densely connections between feature maps and per-pixel classifiers, which enhances the capability to handle different transformations. Later, as each layer evolves, it uses more typical kernel size 3 used in image classification. If input and output channel number are different, it uses stride to speed up the process. If input and output are has same number of channels, use stride equal to 1 to capture the finer details of the input image. The evolution of the number of channels are 3->16->32->64->128->133, which is a common choice in image processing. The ResNet architecture is printed below.

```

print(model_scratch)

Net(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
  )
)

```



```

        (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(16, 32, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(32, 64, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=128, out_features=133, bias=True)
)

```

Next step is to specify the loss function and optimizer. As mentioned in the evaluation metrics, during training, we can use the Cross-entropy loss to get an idea of how the training goes through each epoch, and tune the model if the loss is too big or did not decrease as training more epochs. I tried both Adam and SGD optimizer and doesn't show too much difference here. I chose a learning rate of 0.001, a typical value for a scratch model, and a weight decay of 2e-3 to add L2 regulation and prevent overfitting.

```
### TODO: select loss function
criterion_scratch = nn.CrossEntropyLoss()

### TODO: select optimizer
optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=0.001, weight_decay=2e-3)
```

Now we can train the model. The training is through GPU with 30 epochs. The trained model is saved as 'model_scratch.pt'. The performance of the model is discussed in Part IV justification session.

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

The final step is to test the Model. The model has achieved a test accuracy of 24%, 205 out of 836 images have been classified correctly. It exceeds our success criteria for this task (10%) .

```
# Load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
# call test function
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

Test Loss: 2.899311

Test Accuracy: 24% (205/836)

Step 4: Create a CNN to Classify Dog Breeds using Transfer Learning. Similarly, this step requires to specify data loaders for the dog dataset, define model architecture, specify loss function and optimizer, train and test the dataset. Here, the transfer model I used is the most recent pretrained **ResNet152** model. I changed its FC layer parameter to (2048, 133) to match the dimension of our case. requires_grad is set to false because we don't need to store the gradient, which will cause more data storage and reduce speed. I think the ResNet152 is suitable for our problem because it has the most deepest layers in all ResNET architecture so far. When the layer got deeper, it can extract more relevant features (edges, shapes, colors, etc) and thus can make a better generalization/prediction. Though it is deep, each layer is relatively simple. ResNET152 is pre-trained on the ImageNet with a depth of 152 layers -- 8x deeper than VGG nets but still having lower complexity. An ensemble of these residual nets achieves 3.57% error on the ImageNet test set [9].

The test accuracy reached 85%, much higher than required pass criteria (60%). With this model, we can predict the dog breed given an input image. The performance of the model is discussed in Part IV justification session.

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

Test Loss: 1.105151

Test Accuracy: 85% (716/836)

Step 5: Write the Algorithm. Now we are ready to put them all together and write the dog breed detection app. Remember what we want to achieve: if a dog is detected, return the predicted breed; if a human is detected, return the resembling dog breed, if neither is detected, output an error.

```
In [41]: ### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

def run_app(img_path):
    ## handle cases for a human face, dog, and neither

    def show_image(img_path):
        # get bounding box for each detected face
        img = cv2.imread(img_path)
        # 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()

    if dog_detector(img_path):
        print('\n hello, dog!')
        show_image(img_path)
        pred=predict_breed_transfer(img_path)
        print('You are a ... \n' + pred)
    elif face_detector(img_path):
        print('\n hello, human!')
        show_image(img_path)
        pred=predict_breed_transfer(img_path)
        print('You looks like a ... \n' + pred)

    else:
        print('\nError: Nothing is detected\n')
```

Refinement

The model development is a try-and-error process. I have tuned many scenarios and obtained a optimized version, though not perfect, summarized as below

- 1) For the model from scratch, initially I used a standard CNN model (Conv2D->ReLU->Maxpool->Conv2D->ReLU->Maxpool....FC). However, no matter what hyperparameter I tune (# layers, dropout, learning rate, number of epochs, regularization, etc) or data preprocessing I use, it can't surpass 10%. So finally I chose a ResNET and easily achieved 24% without much tuning.

- 2) For the transferred model. First I modified the data augmentation method. I apply the random rotation and random flip only to the training set. Validation and Test set needs to have the same distribution. By removing the randomness, it reduces the variances during prediction, and thus improves accuracy. The second refinement I did is to change the optimizer from Adam to SGD. Surprisingly, the accuracy improves from 65% to 85% for 30 epochs. This is consistent with the finding that SGD has better generalization than Adam, even though Adam converges faster [10]. In addition, I tuned the learning rate and gave it a relatively large value of 0.01 (compared to model scratch) to let it converge faster since SGD optimizer usually converges slowly.

```
import torch.optim as optim
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr=0.01)
```

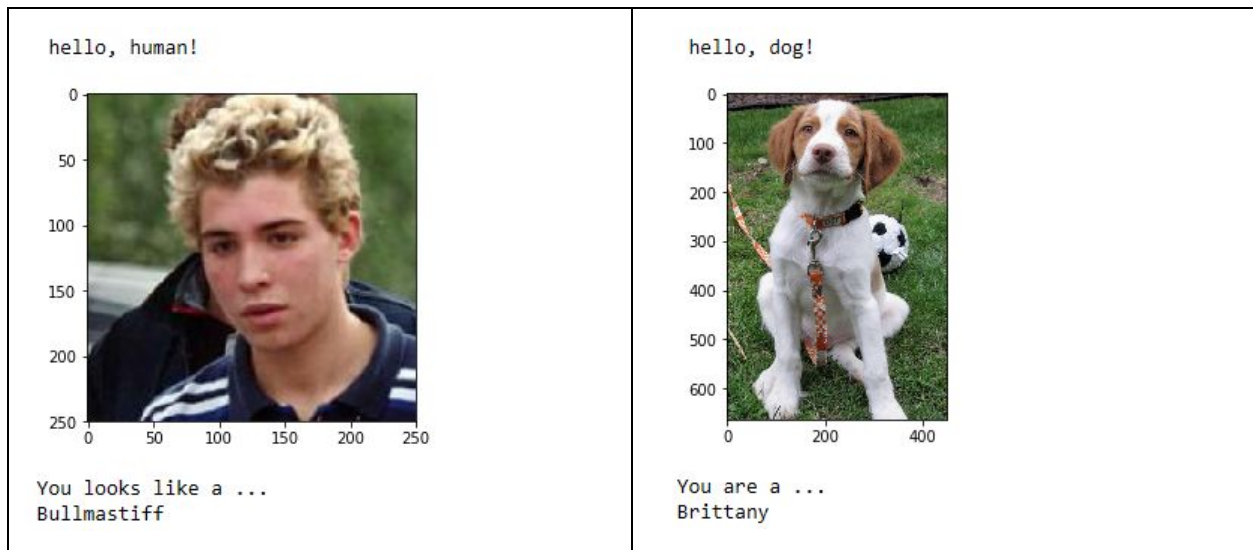
Part IV Results

Model Evaluation and Validation

With respect to the functionalities we planned to achieve, the dog app I developed performs pretty well. The final model's qualities are detailed below.

Detector Type	Evaluation Matrix	Accuracy	Sample size
Face detector	How well it detect a human	98%	100
Dog detector	How well it detect a dog	100%	100
Beed Classifier	How well it identify the dog's breed	85%	836

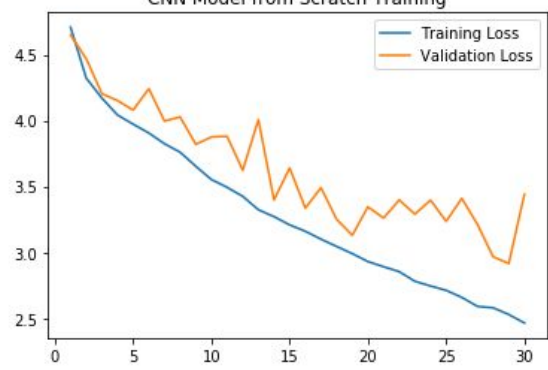
To validate the robustness of the model's solution. I have tested the app using several different groups of images (either from different sections of human files and dog files or images in the image folder). It successfully distinguished human vs dog. Also compared with the dog name in the folder, it correctly recognized most of the dogs.

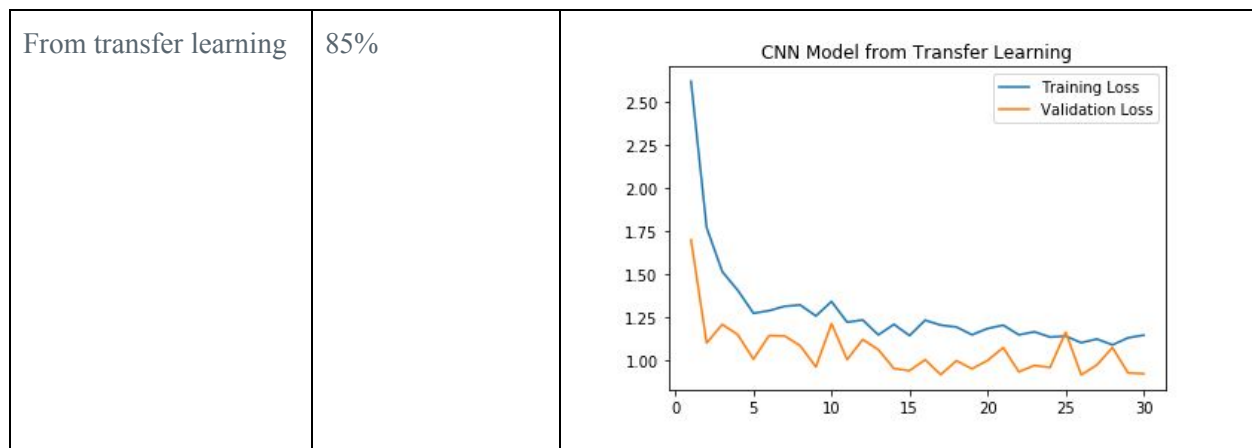


Examples of outputs

Justification

The CNN model developed from scratch is used as a benchmark. Below is a comparison of the final model (transferred model) with the benchmark.

CNN Model	Test Accuracy	Cross-entropy loss
Build from Scratch	24%	<p>CNN Model from Scratch Training</p> 



We can see that after 20 epochs, the model from scratch is starting to show signs of overfitting. On the other hand, the transferred model has much lower loss compared to the model created from scratch. Surprisingly, the validation loss is also lower than training loss. Though oscillating, it is still decreasing after 30 epochs, indicating not much overfitting.

The final result is better than expected. But it is not yet significant enough to adequately solve the problem. Sometimes it didn't predict the breed correctly especially when the two breeds are alike. I think people using this app will mostly be concerned about distinguishing dog breeds that are similar, so we need a more robust solution with even higher accuracy for these types of data. Possible improvement can be made as follows.

- The pre-trained model only ran 30 epochs. I didn't train more Epochs because my poor internet always disconnects and causes the trained model to have error saving the model. A practical training should run more than 100 epochs, which should give at least 90% accuracy.
- The training sample size is small. The distribution of provided training samples may be different than what is on ImageNet. If more images can be provided, it should certainly improve the accuracy.
- More hyperparameter tuning can be conducted, such as learning rate, dropout rate, weighted decay, minibatch size and regularization, etc. This can also improve the model
- Trying different types of pre-image processing to better locate the face or hairs. Maybe add a locator before doing the image augmentation
- I haven't tested much using images without either human or dog, or images with both human and dog to see how the app performs. One reason for that is the model always throws out error for incorrect dimensionality. I guess the images I have are all 4 channels. I was not able to convert them to 3 channels successfully. Or there may be other culprits behind this.

References

[1] Can You Tell These Dog Breed Look-Alikes Apart?

<https://www.akc.org/expert-advice/lifestyle/dog-breed-look-alikes/>

- [2] Why are Convolutional Neural Networks good for image classification?
<https://medium.com/datadriveninvestor/why-are-convolutional-neural-networks-good-for-image-classification-146ec6e865e8>
- [3] CNN Project: Dog Breed Classifier
<https://github.com/udacity/deep-learning-v2-pytorch/tree/master/project-dog-classification>
- [4] Face Detection using Haar Cascades
http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html
- [5] An Overview of ResNet and its Variants.
<https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035>
- [6] Enlarging smaller images before inputting into convolutional neural network: zero-padding vs. interpolation. <https://journalofbigdata.springeropen.com/articles/10.1186/s40537-019-0263-7>
- [7] Image Data Pre-Processing for Neural Networks.
<https://becominghuman.ai/image-data-pre-processing-for-neural-networks-498289068258>
- [8] ResNet example Reference
<https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py>
- [9] Reference of ResNet152. <https://arxiv.org/abs/1512.03385>
- [10] The Marginal Value of Adaptive Gradient Methods in Machine Learning.
<https://arxiv.org/abs/1705.08292>