Project Report Dog Breed Classifier

I. Definition

Project Overview:

Image classification is always an important topic in computer vision. The accuracy of the image classification itself is an essential metric to evaluate how good a neural network architecture is. Also, be able to do the image classification is an important part of artificial intelligence and can be applied to a lot of useful things. Also, there are a lot of useful pre-trained models in this area such as VGG16 which could be very useful for transferring learning. Thus, I choose the dog breed classifier as my capstone project.

To classify the correct dog breed is extremely useful and challenging. Even a human has problems distinguishing between two very similar dog breeds. Thus, if we can train a model to find the difference between all kinds of dogs, it would make categorizing dogs much easier and save a lot of human times.

Before starting this project, I want to briefly discuss two other image classification projects that I am familiar with. The first one is the Mnist dataset, this is a dataset that contains images of handwritten digits. The goal is to correctly classify the number between 0 to 9. This is a simple dataset that are often used to create a simple CNN project. People can use this dataset to learn fundamentals of CNN. The second one is the Dog vs Cat project: https://www.kaggle.com/c/dogs-vs-cats. This is a project that aim to classify whether an image is dog or cat. This project is similar to the dog breed classifier where both of the projects are dealing with animal images. The dog breed classifier is much harder as we have 133 labels. But this could be a good start point of my project.

The dataset that I used for this project is the provided data set in the Jupiter notebook that consists of 13k human images and 8k dog images. The human images are used for the first part of the project which is using an OpenCV function to detect human faces. The dog images are the core part of the project. The data for dog images are organized as follows. There are train, valid, and test folders inside the dog_images folder that will be used for training, validation, and testing. For each folder there are 133 subfolders and each subfolder represent a different breed to the dog and it contains all images of dogs that belong to this breed. These breeds are the target of the dog breed classification. I will use the training images as the input to train my dog breed classifier and test the result against the test set.

Dog: https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip

Human: https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip

Problem Statement:

The end goal for this problem is to complete a classifier that if given an image of a dog, should be able to output its breed.

The product of this problem should be a neural network, given an image or a batch of image as input, returns labels that indicate the breed of the dog.

For this purpose, I could either train a neural network from scratch or applying transfer learning to fine-tuned pre-trained models.

Relevant Metrics:

- As a classification problem, the nature metric is the accuracy, which is given the test images, how many of them are correctly labeled. Apart from that, to calculate the accuracy is also the default requirement mentioned in the jupyter notebook. Accuracy might not be a good choice when data is imbalanced, so we probably need to check the data distribution later. Another thing to notice is that the test data has less than 1000 samples while we have 133 different dog breeds to classify. If we use metric such as precision and recall for each class, the available data for each class is likely not enough. Thus overall, I think accuracy is the best suited metric for this problem.
- At the same time, cross entropy loss is very important for the back propagation of the neural network as well as evaluation of the validation data and test data. This is the metric to use during the training time.
- If in the later analysis we find accuracy is not a good metric, precision and recall are also worth trying. In our case, we have n different labels. Thus, we can get the precision and recall for each dog breed and check how well our model performs.

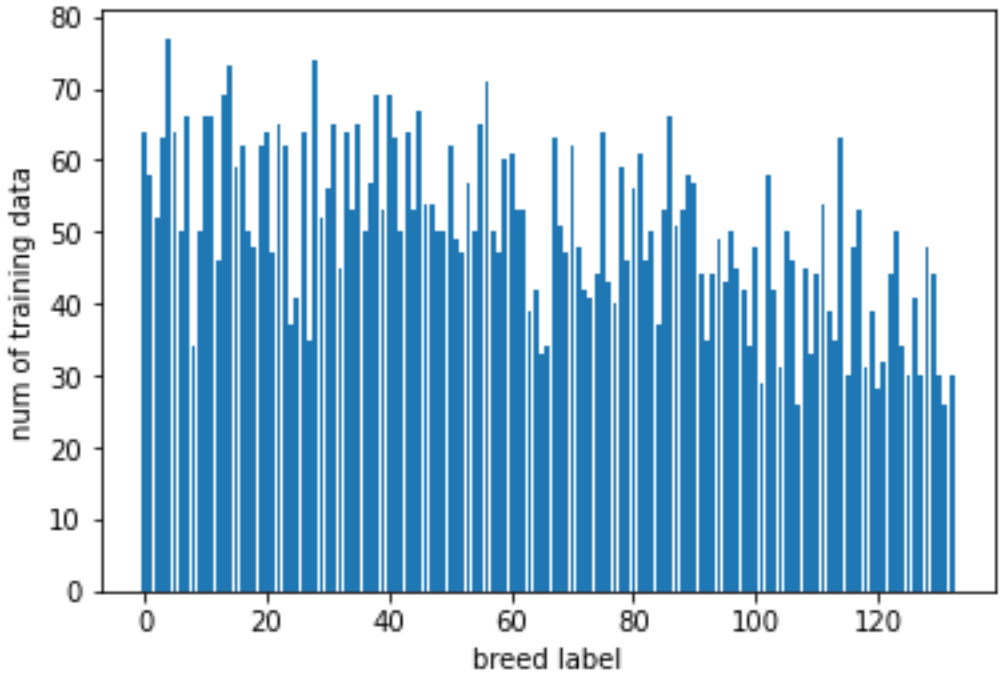
II. Analysis

Data Exploration and Visualization

We use the PIL to inspect some of the images first. Here we can have a close look on one of the images.

In our neural network each image will be represented as a matrix of [channel, height, weight]. Upon on inspection, we can find different image have different size. Thus, we will need to apply transformation to the input, either by resize or crop to make it suitable for the neural network we defined.

It is also worthwhile to discuss the data distributions, below we plot the training data for each breed in a bar chart. The x-axis represents the breed and the y-axis represents the num of training samples for each breed.



As can be seen, the data is relatively evenly distributed for different breeds. Thus, we do not need to deal with imbalanced data. At the same time, we can see that the training sample for each breed is less than 100. This means we do not have lots of data to work with. Thus, if we are going to train with our own neural network, the parameter space should not be too huge. And the transfer learning will probably give us better results. As existing neural networks weights are learned from all kinds of data, not restricted to the data that we have.

Algorithms and Techniques

Looking at how the data stored in the data folders. The ImageFolder class from Pytorch is good method for loading the data. Also, as mentioned in the above, we need to apply transform while using ImageFolder. Possible transformations are resize, image crop and normalization. After obtaining the dataset, we also need to use DataLoader class to generate the final data.

For the neural network part, we will use a convolutional neural network. Thus, we need to use Conv2d, Linear, MaxPool, Dropout from torch.nn module. These are essential parts in constructing the neural network.

In the training part, we also need to specify the criterion, which is the CrossEntropyLoss. For the optimizer, I will use optim.Adam as my choice.

Detailed explanation on some of the techniques:

Convolution: This is one of the basic techniques in CNN. By applying a kernel to the image, the nearby pixels in an image can be combined, this allows the neural the neural network to learn the relationship between pixels. After the convolution, the image becomes abstracted to a feature map with new number of channels, height, and width. Convolution is also a great way to share parameters which reduces the number of total parameters compared to a fully connected layer. Thus, CNN is powerful at dealing with images. For the convolutional layers, the hyper parameters to use are:

- Kernel size: The area of image that we want to compute as a single value.
- Stride: Controls the number of pixels the filter will move at a time. A stride big than 1 will reduce the image height and width.
- Output Channels: The parameters will affect the number of channels after a image pass through the convolutional layer

Pooling: Pooling layer reduce the dimensions of the data by combining the outputs of a cluster of images into a single value. The cluster size is typically 2 * 2.

Dropout: During the training time, the dropout will randomly shutdown neurons in the fully connected layers which is a great way to prevent overfitting. We can choose the percentage of the neurons to shutdown when setting up the dropout.

Fully connected layer: This is also a core part of the neural network; it uses neurons to connect all activations in the previous layer. This is often used in the last few layers of a CNN. Here we need to specify the hidden dimensions for the fully connected layers.

Cross Entropy: This is a common loss function used for multi-class classification.

Adam Optimizer: It is an extension to the well-know SGD optimizer. It is computationally efficient and well suited for problems that are large in terms of parameters.

Benchmark

The main metric to compare here is the accuracy. The instruction jupyter notebook clearly sets the benchmark for this project:

For a neural network form scratch, the accuracy should be more than 10%. This is justified because with the limit amount of data and a lot of labels, 10% accuracy is good enough and it is significantly better than guessing.

For a neural network use transfer learning, the accuracy should be at least 60%. This is also reasonable, because the pre-trained model often learned parameters from loads of data that are not included in our training set.

III. Methodology

Data Preprocessing

As mentioned in previous sections, the image data that we have are of different dimensions. So there need to be a data preprocessing. In my use case, I can use transforms and datasets from torchvision to achieve the purpose.

Here I defined two variables: image_size, cropsize. For training images, the images will first be resized to (3, image_size, image_size) and then cropped to (3, cropsize, cropsize) by random cropping. For testing images, they will be directly resized to (3, cropsize, cropsize). The reason behind cropping is to reduce the contribution of the background in CNN decisions.

Besides resizing, I also augmented training data through random horizontal flip to make sure my CNN could recognize mirror image.

Finally, normalizing need to be performed for both training and testing to make sure the inputs are on the same scale so that the convergence could be faster.

For CNN model from scratch, the input shape for neural network is (3, 64, 64) and for transfer learning CNN, the input shape is (3, 224, 224).

Implementation

- Model Architecture for CNN from scratch:

```
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.conv0 = nn.Conv2d(3, 3, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv1 = nn.Conv2d(3, 6, kernel_size=5, stride=1, padding=1)
        self.conv2 = nn.Conv2d(6, 12, kernel_size=5, stride=2, padding=1)
        self.conv3 = nn.Conv2d(12, 24, kernel_size=3, stride=2, padding=1)
        self.fc1 = nn.Linear(24 * 7 * 7, 200)
        self.fc2 = nn.Linear(200, 500)  #output layer
        self.fc3 = nn.Linear(500, 133)
        self.drop = nn.Dropout(0.3)

    def forward(self, x):
        ## Define forward behavior
        # print (x.shape)
        x = x.view(-1, 3, 64, 64)
        x = self.pool(F.relu(self.conv0(x)))
        # print(x.shape, "expect -1, 3, 32, 32")
        x = F.relu(self.conv1(x))
        # print(x.shape, "expect -1, 6, 30, 30")
        x = F.relu(self.conv2(x))
        # print(x.shape, "expect -1, 12, 14, 14")
        x = F.relu(self.conv3(x))
        # print(x.shape, "expect -1, 24, 7, 7")
        fc_input_dim = x.size(1) * x.size(2) * x.size(3)
        x = x.view(-1, fc_input_dim)
        x = F.relu(self.fc1(x))
        x = self.drop(x)
        x = F.relu(self.fc2(x))
        x = self.drop(x)
        x = self.fc3(x)
        return x
```

  - As showed in the above, I have 4 convolutional layers and 3 fully connected layers.
  - The input size for my CNN is (3, 64, 64). The first step is to use convolutional layers to reduce the image size while also increasing the number of channels. Here I use four convolution layers conv0-3 to transform the input into (24, 7, 7). This part can be seen in the forward function above. Pooling is applied in the above process. The activation function used is Relu.
  - Now the neural network will flatten the (24, 7, 7) tensor and start to construct the fully connected layer. Here I have three fully connected layers fc1-3. fc3 is the output layer that has 133 classes.
  - I also applied dropout in fully connected layers to prevent overfitting.

- Model Architecture for CNN with transfer learning

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

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

num_ftrs = model_transfer.classifier[6].in_features
model_transfer.classifier[6] = nn.Linear(num_ftrs, 133)
```

    - Here I used VGG16 as the pretrained model.
    - VGG16 has two parts, the first part is convolution layers. This part is mainly used to detect patterns. This part can be directly applied to my use case without updating the parameters. So, I have the parameters in this part as untrainable as can been seen in the image above.
    - The second part is fully connected layers which are the classifier part. I change the out features of the last layer to 133 to suit my use case. And set this part as trainable.
    - I think this architecture is suitable for my use case because VGG16 is a successful model in image classifications and it is also the main goal of my project. I could use the pre-trained weights of VGG16 efficiently with the last few layers as trainable.

- Training and testing:
    - Here I used the cross-entropy function as the loss function and Adam optimizer with learning rate = 0.001 as the optimizer.
    - The training algorithm goes through the training data via data loader. For each mini batch, it first sets model as train mode, forward computes the network, calculates the loss, and uses optimizer to update the parameters through the gradients. After the end of each epoch, it computes the validation loss. If the validation loss decrease compares to the minimum validation loss before, it saves the model. After the training method finishes, I test the model against the test datasets.

Refinement

For the pretrained CNN, the initial result is not very good. I could only reach about 4% accuracy on testing data. In order to solve that problem, I increase the number of channels in the convolutional layers. In the end of the convolutional layers, I have 24 channels instead of initially 12 channels. The accuracy increased to 7%.

The result is still not ideal. I found that the training process tend to overfitting. I added the dropout in the fully connected layer to prevent that from happening. At the same time, I applied random crop and random horizontal flip to the training data to increase the predictive power.

In the end, I found normalizing data helped a lot for fast convergence. It greatly increases the learning speed in the first few epochs.

IV. Results

Model Evaluation and Validation

Here is how the model selection is completed during the training process. The data is divided into three different parts, training, validation, and testing. The training data is used for computing the gradients and update the model parameters. Validation data is used for selecting the best model during the training process. At the end of every training epoch, the neural network will compute the validation loss. If it is smaller than the previous smallest validation loss, the current model will be saved locally. Thus, the model selection is based on the data that the neural network is not trained on which is reasonable.

In the end, the final model is tested on testing data, which it has not seen before.

Also, the training procedure include random cropping, dropout, and random flipping. These techniques ensure the small perturbation will not greatly affect the training results.

Finally, the existence of the testing model makes sure the final model is trustable.

Justification

In the end, the accuracy on testing data for CNN from scratch is 11% and for CNN use transfer learning is 72%. Both are better than the benchmark set before (also the requirement for this project as stated in the project jupyter notebook).

I have also run the final model on specific sample images, the results are promising.

In the end, I built a small algorithm. It combines the human face detector, dog detector and a dog breed classifier. The algorithm first tests the image against face detector, if it finds human faces, the image will be passed to the dog breed classifier and the output breed is the dog breed that resembles the human face. If human face is not detected, the algorithm tests the image against dog detector, if a dog is detected. It will use the dog breed classifier to classify the dog breeds. Finally, if the image is neither human nor dog, return a message that indicator no human or dog found. The dog breed classifier is the CNN with transfer learning.

I tested 10 human files and 10 dog files. For the human files, 9 out of 10 images are classified as human and outputs dog breed that resembles the face. For the dog files, 8 out of 10 images are classified as dogs and outputs the correct breed for 6 of them.

Now I can confidently state that the final solution is significant enough to solve the problem.

V. Conclusion

Visualization

Below is a very good example that our trained model is very robust and has strong predictive power.

```
files1 = 'dog_Images/test/130.Welsh_springer_spaniel/Welsh_springer_spaniel_08214.jpg'
print (predict_breed_transfer(files1))
im1 = Image.open(files1)
im1.resize((300, 200))
```

```
Welsh springer spaniel
```

```
files2 = 'dog_Images/test/037.Brittany/Brittany_02591.jpg'
print (predict_breed_transfer(files2))
im2 = Image.open(files2)
im2.resize((300, 200))
```

```
Brittany
```

Here we have two very similar dog images that are both in test dataset. Without seeing the test dataset before. The trained model was able to predict the correct breed type for these two breeds. This task is very hard even for human. This shows how powerful CNN can be at image classifications.

Reflection

The end to end procedure is import datasets -> analyze datasets and decide possible preprocessing -> create a neural network model -> training and testing the model -> create application using the trained model.

I think the part that is the most challenging is to optimize the CNN model from scratch. While the neural network algorithm is not super hard. It is extremely important to set an efficient model architecture and preprocessing the images properly. I spend a lot of time trying to make my CNN model better. And I also learned a lot during the project.

Improvement

Although I am satisfied with my results, I do think that there are a lot of areas that I can improve.

First, for the CNN from scratch. Maybe I could achieve better results with a larger input size than 64*64 and a more complex neural network to make the accuracy higher.

Second, this project is mainly for classification. I would like to implement an algorithm that also does the detection of the dog face, not only the breed.

Finally, I would also want to try other pre-trained net to see whether they perform better than VGG16.