Dog Breed Classifier Project Summary

by Bobby Mander

Feb 17, 2020

In this summary we will recap the work involved with our dog breed classifier project from start to finish.

Project Overview

We will be classifying images using machine learning in this project. Specifically we will be attempting to classify whether an image contains a dog and if it does, what breed of dog is present. If no dog is present, we will also attempt to detect if a human face is present and if it is, what kind of dog that face most resembles. We will solve this problem using convolutional neural networks (CNNs) trained for the purpose.

Problem Statement

Image classification is one of the oldest machine learning problem domains. This project will focus on examining images of dogs and human faces. For dogs we will attempt to determine the exact breed of dog in the picture. For humans we will find the most similar dog breed that their face resembles. At the conclusion of the project we will have a command line tool that will take in an image and tell the user whether a dog or human is there and the breed associated with them to the highest accuracy possible based on machine learning using CNNs and a well sized training, validation, and test data set.

We will be creating two CNNs in this project. The first CNN will be created from scratch with no pre-training. This will require constructing all the layers of the model, configuring them with appropriate parameters, and training on the training and validation data set from ground zero. The second CNN will be a pre-trained CNN that has already been extensively trained on everyday image data. We will use that pre-trained CNN by employing transfer learning to our problem of dog classification. We will compare the two CNNs and their performance once they are completed. We will then choose the best CNN for our prediction tool.

Metrics

The success of the project will be based entirely on prediction accuracy, specifically how many dog breeds can be predicted correctly versus the total number of predictions. We will evaluate this metric against both CNNs we create. Accuracy will determine which CNN is chosen to be used by our final prediction tool. Although not a major factor in the success of the project we

will keep a close eye on training time and hope to keep it within reasonable time limits so that we can perform hyperparameter tuning and experiment with many different settings.

Analysis

Let's take a closer look at the data we will be working with to train and test the CNNs developed.

Data Exploration

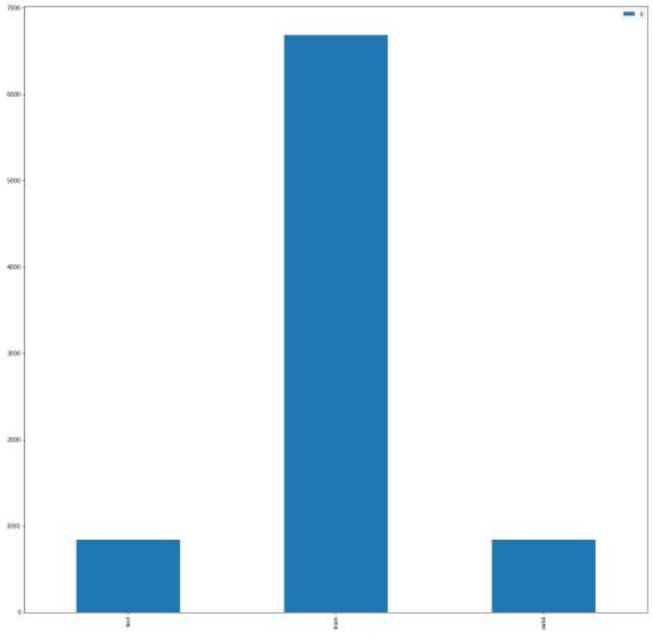
There are two datasets involved with this project. The first is a set of images of human faces for use with our human detection algorithm. We will not be training with this dataset, We will simply be testing with the data using a standard face detection algorithm. This data set has over 13,000 images of human faces of various famous people including John Lennon and Paul McCartney, for example. The images are all rgb color jpg type images of size 250x250. Some examples are shown below:

The second dataset is more interesting as it has a rich set of rgb color images of dogs of various breeds. There are over 8,000 images in this data set and the data set has been split into training, validation, and test subsets. There are 133 different breeds of dogs labeled in this dataset with a good number of images for each breed included. The images are of varying sizes ranging from approximately 105x113 to 4278x4003. Normalization of these images into a standard size will be important for training our CNN. Some examples are shown below:

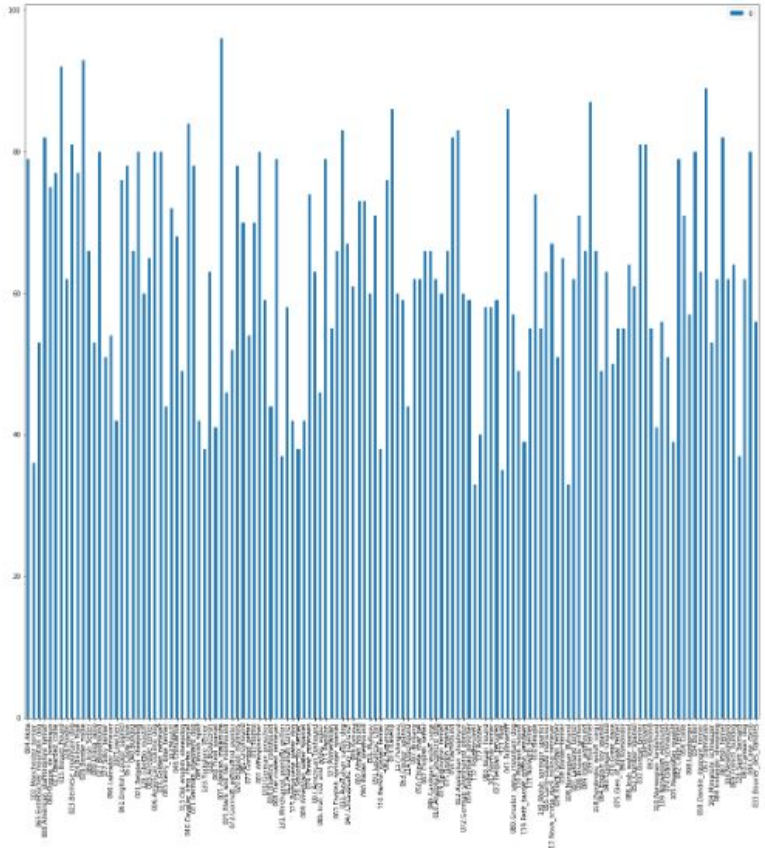
Exploratory Visualization

The histogram below shows a breakdown of the data into the training, validation, and test dataset. The second histogram below shows a breakdown of the data into the 133 different dog breeds. Clearly the majority of images belong to the training set as our CNN will need extensive training to be useful. The images are well distributed among the different breeds of dogs we have to learn.

Histogram 1: Images per test, training, and validation datasets



Histogram 2: Images per dog breed



Algorithms and Techniques

Many algorithms and techniques will be used for this project. Here we outline the most important ones:

1. Convolutional neural networks (CNNs). We will use CNNs to perform the breed prediction. CNNs are commonly applied to classification problems such as this and are well suited to handling image processing machine learning applications like this. The use of convolutions ensures multilayer filtering of the image looking for features from simple edges to more complex patterns. They are also computationally efficient since each layer in the neural network is not fully connected.
2. Data Transforms. Since input images can vary so widely in terms of size, quality, and image position, transforms are necessary to normalize the image. Various transforms will be tried and applied to our images in order to produce the best performing CNN. Examples of transforms are resizes, crops, pixel value normalization, horiztonal flips, and vertical flips.
3. Haar feature-based cascade classifiers. Haar classifier is a well known available face recognition algorithm that can be applied to images. We will use this classifier to detect human faces in our input images in those cases dogs are not detected.
4. VGG16. A very powerful CNN that performs well with everyday images is called VGG16. Named after a research group at Oxford University this CNN has 16 layers with many convolution layers and linear layers providing tremendous power for it to classify images. VGG16 has been trained extensively using the well known and vast ImageNet database, with training time in excess of several days. Because of this training VGG16 performs very well with image classification.
5. Transfer learning. Using the VGG16 pre-trained network and the transfer learning technique, we will be able to apply the power of VGG16 to our problem of dog breed classification. This will save us considerable time in training our CNN and also lead to greater accuracy since the ImageNet database of images and our dog images are similar input data sets so VGG16 will almost be directly applicable.
6. AWS. We will leverage the power of Amazon Web Services, in particular its SageMaker platform with Jupyter notebooks. This will also us to tap into the vast compute power AWS has including access to GPUs which speed up training immensely during hyperparameter tuning. CNNs involve many hyperparameters so tuning is an extensive process that requires hefty computing resources and time to perform.

Benchmark

In terms of benchmarks, random guessing will result in 0.8% chance of getting the dog breed correct since there are 133 different breeds. There are a variety of other well known CNNs available but these would have to be applied to the dog breed classification problem. We will be evaluating using a well known CNN and creating a CNN from scratch. We will attempt to achieve 10% or greater accuracy using a CNN from scratch.

The VGG16 CNN will be our benchmark model even though it has not been applied to this problem. We will adapt it using transfer learning to apply it to this domain to achieve superior results. We will attempt to achieve 60% or greater accuracy using the VGG16 with transfer learning. The VGG16 model will likely be chosen as the final benchmark due to its higher predicted accuracy.

Methodology

Here we will share details of the processing we performed in order to achieve our final system that predicts dog breeds. We will focus on our CNN from scratch as this model required the most construction, the most training, and the most hyperparameter tuning.

Data Preprocessing

We tested briefly with the human face data while we spent most of our time training and testing with the dog breed data set. This data set contained over 8,000 images of various sizes showing dogs in everyday poses. The dogs were oriented in random ways, facing left, right, or straight ahead. Some examples were shown earlier. The dogs were labeled with their breeds, making this a supervised learning problem. In order to normalize the training image dataset, we performed the following operations using PyTorch's normalization libraries:

- RandomRotation. We randomly rotated the image by 10 degrees in either direction. This is to ensure we can detect dogs that were slightly tilted in the images presented during testing. We only used a small angle as this is the limit of what we should see with standard everyday pictures.
- RandomResizeCrop. We set the input image dimensions to be 256x256 so we used this transform to randomly crop and resize the image appropriately. Since 256 is cleanly divisible by a power of 2, this will help with the CNN later as we continually shrink the dimensions of the input as we move further into the CNN layers. After this transform all input images have the same dimensions, perfect for feeding into our CNN.
- RandomHorizontalFlip. Since the dog orientation in the images should not affect the prediction we randomly flip the image horizontally only so we can see dogs facing left and right randomly in training. We don't vertically flip since that would have the dog upside down, not a realistic picture we should see.
- ToTensor. In order to feed into our CNN, we need to convert the input image to a tensor data type.
- Normalize. We take the image pixel RGB values and we normalize them to the range 0 to 1. We don't use 0.5 mean and 0.5 std deviation but close. Instead we use VGG16's optimally found mean and std deviations, leveraging what they already have about image processing and classification. These values were [0.485, 0.456, 0.406] for the mean and [0.229, 0.224, 0.225] for the std deviation.

Note we preprocessed the validation and test datasets slightly differently as shown below. This is because the purpose of training is to present all possibilities of images so the CNN can learn the most, while the purpose of validation and test is to predict using the already trained model:

- Resize. Instead of cropping randomly, we simply resize the image to the 256x256 dimensions. This ensures we have the relevant dog in the image for prediction purposes.
- ToTensor. We convert data types just as before.
- Normalize. We normalize just as before.

Implementation

Implementation of the two CNNs involved many of the same technologies. The common technologies and techniques used were:

- AWS. Using Amazon's cloud, we were able to leverage SageMaker and Jupyter notebooks directly without installation of any software. In particular we used the conda_pytorch_36 kernel and we used the ml.p2.xlarge notebook instance which provided access to one Nvidia K80 GPU to speed up training time by a factor of about 10.
- PyTorch. This powerful library in Python provides the ability to quickly construct neural networks including CNNs without requiring extensive programming. Once constructed PyTorch has the libraries to train the network and compute all the losses and perform back propagation as well. PyTorch also has a rich set of APIs for transforming data so that it can be fed into a neural network. Finally it contains APIs for loading the data itself using loaders and specifically image folder loaders which we'll use.
- VGG16. Within PyTorch there is an implementation of VGG16 which is already pretrained on ImageNet. This makes it straightforward to instantiate and use it directly.
- OpenCV. This Python library contains many features including the Haar cascades face detection algorithm which we will use straight out of the box.

The specific technique used for the application of VGG16 to our domain was transfer learning.

Minor complications did occur during the implementation. These included:

- Picking the proper notebook kernel. We started with the amazon_mxnet_36 kernel since it already contained the OpenCV libraries using for human face detection, but we ran into problems later when working in PyTorch even after installing it in the kernel. So we eventually switched over to the better conda_pytorch_36 kernel and installed OpenCV on top of that kernel which allowed us to run our CNNs without problem.
- Picking the proper notebook instance. We started with the AWS free tier instance ml.t2.medium but we quickly found that training our CNN was very time consuming. A single run with a 3 layer CNN took over 60min. We struggled this way for a while before finding and switching over to the ml.p2.xlarge GPU instance. The training time dropped to about 10min. This was important as we spent a considerable amount of time running different scenarios with different hyperparameter settings. Our hyperparameter tuning

was manual so the quicker the cycle the better. We did notice the cost was considerably different as with the free instance we were only spending about 50 cents while with the GPU the cost increased to $20.

- Picking the right hyperparameters. Creating and training a CNN involves numerous hyperparameters in order to classify properly. Setting these optimally will result in a more optimal and accurate CNN. We used the following hyperparameters in our training and varied them to see their effect on the resulting CNN prediction accuracy:
    - Convolutional layers. The number of layers in a CNN determine how sophisticated the CNN is and how many patterns it can discover along with the complexity of those patterns. Too few layers will mean detailed features will not be discovered. Too many layers will mean longer and longer training times and more difficulty in training the network properly.
    - Linear layers. The number of linear layers at the end of a CNN will determine how sophisticated the classification can be. If you have one linear layer, more sophisticated classifications will not be possible while having too many linear layers can cause training issues due to their fully connected nature.
    - First linear layer size. Considering the case of having 2 linear layers, the size of the first linear layer's output and the second linear layer's input can be chosen and varied to see its effect on training.
    - Kernel size. The convolution kernel size can be chosen from the classification task. Common sizes are 3x3 and 5x5.
    - Initial features size. The number of initial features the first convolution layer should produce can be varied. The more features, the more different types of convolution are applied. This comes at training effort cost as this choice ripples through all the CNNs layers.
    - Image size. Choosing the best normalized image input size is important to ensure you capture the details in the image you need to perform the classification while at the same time making sure you are not overwhelming the CNN with too much or too little detail. Too much detail will require more training while too little detail will result in loss of feature information.
    - Learning rate. The CNN learning rate is important to achieve proper training results and in efficient time as well. If the learning rate is too small, the learning will require more and more epochs while if the learning rate is too large, the training may not find the optimal loss due to overfitting.
    - Horizontal flip. Whether to perform a horizontal flip on the images was an option we considered turning on and off.
    - Normalization params. The actual normalization factors to apply to the rgb pixel values were varied from vanilla 0.5 mean and 0.5 std dev to the finely tuned VGG16 values.
    - Dropout factor. The size of the dropout factor was varied to control the amount of overfitting that may be occurring.
    - Transformation equality. Whether to transform the training data set and the validation and test data set equally versus differently was considered and tried.

- Batch norm layer. The use of batch norm layers after every convolutional and max pool layer was considered. We turned this layer on and off to observe the results.
- Loss function and optimizer. Although we did not vary them, these are CNN parameters that could be varied. We used CrossEntropyLoss for our loss function (standard for classification problems) and optim.SGD as our optimizer.

Refinement

As described, our set of hyperparameters was vast so many combinations had to be tried to find the best results. We started varying the convolution layers between 2-6 layers. For every convolution layer we had a max pooling layer in order to reduce the input size for the next layer. We fixed the max pooling layer to do a 2x2 maxpool. After several tests we found that 3 layers was a good compromise between training time and too simple a CNN. We also tried just a couple of kernel settings before setting on 3x3 as the best approach as it is well known and used to find edges and other higher level features.

The next hyperparameter was the number of linear layers. We varied this between 1-3 but found that 2 layers was also a good compromise for training time and complexity while retaining the power we needed to classify. After this was determined, we moved to the size of the first linear layer's output. We tried various values between 500-5,000 but found that 1,000 provided very good results and was a good intermediate figure to our final goal of 133 predicted classifications.

Next, we looked at how many features we wanted our first convolution layer to produce. We varied this between 16 and 64 before finding that 16 was just fine and a good figure considering that at each convolution layer this would double for the next layer.

We also experimented with many different input image sizes after normalization. We started with small images 100x100 and went up to 500x500. The larger images took quite a long while to train on the CNN as it exploded the size of the model while the smaller images would lose detail on the larger images in the data set. After many experiments varying this we decided on 256x256 as it was a perfect power of 2 and a good middle ground for the sizes of our input images.

The learning rate was varied with trial and error between 0.001 and 0.01. We found a good learning rate was 0.005 based on these experiments. Using this learning rate we were able to train our model within 20 epochs and within 15min on the GPU instance.

We then experimented with turning on horizontal flip transforms for training. We found that accuracy was definitely improved by doing this random flipping perhaps signifying that the input data used for training had a bias in one orientation or the other.

Next we experimented with the dropout rate and the transform numerics. We varied the dropout rate between 0.2 and 0.5 and found that 0.2 provided the best results in our training. For the rgb pixel value transform we started with 0.5 mean and 0.5 std deviation but then moved to the VGG16 selected numbers for the mean and standard deviation for the rgb pixel value components. These were [0.485, 0.456, 0.406] for the mean and [0.229, 0.224, 0.225] for the std deviation. We found that the VGG16 values worked best so we stuck with those.

Finally we experimented with using different transforms for the training versus the validation and test data sets. We found that having different transforms worked best. Extra transforms on the training set included a random crop and horizontal flip while for the validation and test data sets we did not do those transforms. Along with this we decided to turn on batch normalization layers after each max pool layer. It turns out that training is improved and the loss reduced by employing these layers. Without the layers the training loss would be ever so gradually reduced but after turning these layers on we found the training loss dropped significantly and more consistently. It appears that normalizing the input per layer is a big help in training and should be turned on to avoid problems as a complex CNN is trained.

Results

Let us now present and evaluate the results we obtained.

Model Evaluation and Validation

We trained and tested two models. Those separate evaluations are presented next.

CNN From Scratch

Using the hyperparameter settings described earlier, we were able to achieve 13% accuracy using the CNN from scratch model. This exceeded the project benchmark target of 10%. This was a good, not great result, which could have been improved with more hyperparameter tuning and more extensive training. A larger CNN with more layers would most certainly have been needed to get better results. Perhaps more training samples and/or augmented data sets could have been used. We could have doubled the data set size by horizontally flipping all images for instance. We could have downloaded many more pictures of dogs from the Internet. The only drawback is that we would have had to label those pictures ourselves or perform separate searches for each breed. Given the time and resources we had available, the CNN we constructed performed as designed and well enough for our purposes.

VGG16 With Transfer Learning

The training for the VGG16 CNN with transfer learning was much more straightforward and required far less hyperparameter tuning. We used the same transforms for all training data sets and had no difficulty achieving our target performance. We achieved 69% prediction accuracy once the transfer learning was complete. We used a 224x224 image size as that is what

VGG16 expects. We also converted the last VGG16 linear layer to have 133 outputs, matching the number of dog breeds we were trying to classify. We turned off all adjustment of the VGG16's weights except for the classifier layers and we used a low learning rate of 0.001. Within 10 epochs we achieved tremendous accuracy versus the CNN from scratch.

Justification

In both models we exceeded our target benchmark performance target. For the CNN from scratch we achieved 13%, exceeding our target 10% by 30%. For the VGG16 with transfer learning we achieved 69%, exceeding our target 60% by 15%. These were good results and the VGG16 model was put to use in our final system for predicting dog breed. That final system performed well and had some interesting insights for humans and what dog breeds they are similar to as well. We explored many details related to CNNs and how they perform which will be valuable to constructing CNNs for any future problem domain. Hyperparameter tuning and model construction, and data normalization are the more challenging aspects of creating a CNN. Significant time must be devoted to each of these tasks.

For our final system, see a few example runs output below:

```
Adam_Sandler_0002.jpg :  Human detected, resembling breed found to be  Dachshund
```

```
John_Lennon_0001.jpg :  Human detected, resembling breed found to be  English toy spaniel
```

```
Paul_McCartney_0006.jpg :  Human detected, resembling breed found to be  Silky terrier
```

```
Beagle_01197.jpg :  Dog detected, breed found to be  Beagle
```

```
Belgian_sheepdog_01540.jpg :  Dog detected, breed found to be  Belgian sheepdog
```

```
Boxer_02426.jpg :  Dog detected, breed found to be  Boxer
```

References

1. AWS Instance Types: https://aws.amazon.com/sagemaker/pricing/instance-types/
2. Base Jupyter Notebook: https://github.com/udacity/deep-learning-v2-pytorch/blob/master/project-dog-classification/dog_app.ipynb
3. Dog image dataset: https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip
4. “Face Detection Using Haar Cascades” https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_objdetect/py_face_detection/py_face_detection.html
5. Free Dog Images: https://www.pexels.com/search/dog/
6. Human face image dataset: https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip
7. ImageNet: http://www.image-net.org/
8. Introduction to OpenCV-Python Tutorials https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_setup/py_intro/py_intro.html#intro
9. Mander, Bobby “Dog Breed Classifier Proposal”.
10. Popular Networks “VGG16 – Convolutional Network for Classification and Detection” https://neurohive.io/en/popular-networks/vgg16/
11. PyTorch documentation: https://pytorch.org/docs/stable/nn.html
12. Quora “What is batch normalization?” https://www.quora.com/What-is-a-batch-norm-in-machine-learning
13. Shams S “Understanding TensorFlow” https://machinelearningmedium.com/2018/01/02/understanding-tensorflow/