Machine Learning Engineer Nanodegree

Capstone Project

Sebastian von Hoesslin
October 21st, 2019

I. Definition

Project Overview

In this section I will provide a high-level overview of the project. As my Machine Learning Engineer Nanodegree Capstone Project I chose the dog breed classifier challenge, which was once a Kaggle challenge (https://www.kaggle.com/c/dog-breed-identification).

In this challenge a strictly canine subset of the famous ImageNet Dataset is provided with the purpose to reach a fine-grained image categorization. This dataset is mixed with the Stanford Dogs Dataset (http://vision.stanford.edu/aditya86/ImageNetDogs/) with the aim to gain a large enough Dataset to make high accuracy predictions possible.

In my Capstone Project I implemented two models which can, given an image of a dog, identify an estimate of the canine's breed. If supplied an image of a human, the code will identify the resembling dog breed.

Along with exploring state-of-the-art CNN models for classification, I learned how to make important design decisions about the modeling of image classification algorithms. The goal of this project was to understand the challenges involved in piecing together a series of models designed to perform various tasks in a data processing pipeline. Each model has its strengths and weaknesses, and engineering a real-world application often involves solving many problems without a perfect answer.

Problem Statement

The problem to solve was as follows. I was given two datasets. One dataset with dogs of different breeds and one dataset containing images of human faces. The underlying problem is to define a model strong enough to recognize whether it is confronted with an image of a dog or with an image of a human. In case an image of a dog is provided, the model should be able to estimate its breed with an acceptable amount of confidence. In case an image of a human is provided, the model should

compare it with its breed classes and tell us which breed our human face resembles most.

I will use two different approaches, famous for their efficiency in image recognition and classification tasks. The first approach is to design a model from scratch. Mine is a model inspired by the VGG-16 Model of Oxford University, which itself once won a Kaggle competition for its efficiency on image recognition.

The second part of the project consists of the goal to use transfer-learning in order to attain an even higher accuracy. I, here, decided to choose the ResNet50 Model, which is famous for its efficiency on multiclass classification problems.

Metrics

The metrics or calculation I will use to measure performance of a model are clearly defined in the project outline. As I had two datasets of labeled pictures, I had to solve a supervised machine learning task.

So, the key metric I followed was the accuracy, that is, the percentage of the amount of correctly classified images. The first model, which I built from scratch had to reach an accuracy score of at least 10%, defined as the minimum threshold by Udacity. My model achieved an accuracy of 35% on this task. The low accuracy requirement is due to the relatively small dataset, which makes our model prone to overfit the training data and thus makes it harder for the model to generalize well on before unseen test date.

The second task should at least attain an accuracy score of at least 60%, again my model could attain a solid performance of 77% accuracy. As I could use a pre-trained model, the threshold of minimum accuracy was increased to 60%.

II. Analysis

Data Exploration

In this section, I will briefly analyze the data I was using to solve the problem. As I mentioned above, two datasets were provided, one with images of dogs of different breeds and one with images of human faces.

The first step to take, when provided with data, is to explore all given datasets in order to get a first impression on the possibilities one has in order to solve a problem. It turned out that in the dog dataset 8351 dog images were included, whereas the human dataset contains 13233 of human images. While this might sound like a reasonable number of images first, it is not too much if you want to implement a model from scratch, as it is prone to overfit the features of relatively small datasets.

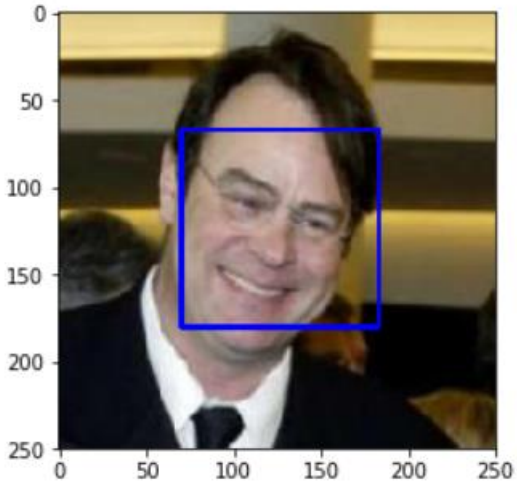
In contrast to the human dataset our dog dataset naturally comes with features that indicate their breed. These identification labels are provided in the form of class ids and breed labels. A first check on the dataset revealed, that there are 133 classes of different dog breeds in my dataset.

Exploratory Visualization

In this section, I will provide some form of visualization that summarizes or extracts a relevant characteristic or feature about the data. In a first step I had to set up a face detector, which is able to identify faces of humans from a picture provided. For this purpose, I used OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images. This is a pre-trained classifier, which follows the famous Paul Viola approach, first proposed in the paper, "Rapid Object Detection using a Boosted Cascade of Simple Features" in 2001. A link to a corresponding tutorial can be found here:
https://docs.opencv.org/trunk/db/d28/tutorial_cascade_classifier.html

To visualize the outcome of this algorithm used on a picture, I printed a screenshot of an analyzed picture below. The model will count the number of faces detected and draw a bounding box around them.

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

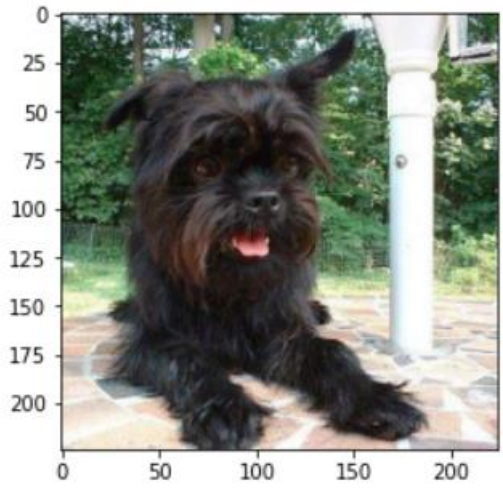


Next, I conducted a sanity check on our datasets to check whether or not this algorithm works fine on the provided data.

```
98.0% images of the first 100 human_files were detected as human face.
17.0% images of the first 100 dog_files were detected as human face.
```

It turned out that there is a considerable amount of dog images falsely classified as humans, which is not ideal, but still not too much of an issue for the purpose of my project.

In a next step, I used a pre-trained VVG-16 Model to create a dog detector. To use this model, our provided dated had to be cropped to the format 244x244 pixels in order to be accepted by the model. Used in our detector it gave the following result on a random sample image.




```
Predicted class id: 252
Label:affenpinscher, monkey pinscher, monkey dog
Predicted class for image is *** AFFENPINSCHER, MONKEY PINSCHER, MONKEY DOG ***
```

As we can see, it returns a predicted class id, a label of the breed and the top three breeds according to the calculated possibility in descending order.

A sanity check on the dog detector gave me the following result, which implies that there is a fairly small amount of humans classified as dogs, whereas all pictures of dogs seemed to be correctly recognized as such.

```
VGG-16 Prediction
The percentage of the detected dog - Humans: 1%
The percentage of the detected dog - Dogs: 100%
```

Algorithms and Techniques

In this section, I will discuss the algorithms and techniques I used for solving the problem. I will justify the use of each one based on the characteristics of the problem.

For the first task, to create a dog breed classifier from scratch, I decided to create a model inspired by the VGG-16 Model, as it is highly efficient, with a reasonable amount of computational resource use, even though, I would recommend to use a GPU-accelerated instance in the cloud to perform the training, which requires up to 3 hours on a relatively modern laptop. As I chose this kind of approach, I had to transform the data first in order to gain the required 224x224 data format. I normalized the data by center cropping it and also included some random horizontal flips and rotations to augment the data and help the model to generalize well on new, unseen images.

As a next step, as for any machine learning model which is to be trained manually, I had to split up the dataset. I created three datasets, the first one, *train*, for the training data, as second one, *valid*, for the cross validation and a third one, *test*, which would show us how good our model is.

For the model I used 14 convolutional layers with a kernel size of 3 and a padding of 1, I also implemented batch normalization layers as a technique for improving the speed, performance, and stability of my CNN. A max pooling layer, a ReLu function and a dropout keep the model slim and swift and keep the model, at least to some extent, from overfitting. At last I had several linear layers with a final output of 133 for our 133 classes of dog breeds.

To only take the best result into account I implemented a function which would save and later use the model with the lowest validation loss. As the loss function I implemented a cross entropy loss function which combines the perks of a logistic softmax and a log loss function and performs well on multiclass classification problems. (For reference, follow this link: http://wiki.fast.ai/index.php/Log_Loss#Multi-class_Classification).

For the second task, I decided to use a ResNet50 Model, which excels at the task of image classification. It has proven to be a powerful backbone model that is very frequently used in many computer vision tasks. The advantage of the ResNet Model is that in residual learning, instead of trying to learn some features, residuals will be learned, that means, a subtraction of features learned from input of a nth layer will be performed using shortcut connections by directly connecting input of nth layer to some (n+x)th layer. Using this method training of networks is easier than training simple deep convolutional neural networks and also the problem of degrading accuracy is resolved.

I used the same data loaders as for the VGG-16 Model in the form of a training, validation and test dataset. I then modified the model, so its output layer would conform to our needs. To use it for our purpose, we need to substitute the classifier (model_transfer.fc = nn.Linear(2048, 133, bias=True)).

On this model I again implemented the CrossEntropyLoss function, for the same reasons as mentioned for the first model above.

Finally, an algorithm will go over some selected input data and apply the model for us. Here, I implemented a run_app function which will identify a dog if the model has a confidence of at least 30% that it identified a and will then print the probabilities of the top three breeds the dog resembles. If the algorithm finds a human in the picture, it will solely output, "It's a human!", and if neither a dog nor a human can be identified the output will be, "I can't detect if it's a human or dog!"

Benchmark

A benchmark model was provided by Udacity. For the model we build from scratch an accuracy of correctly classified images with the value of 10% is given. While this may sound small in the beginning, it is in fact reasonable, as we must train parameters from scratch and are being given relatively small datasets. It is therefore very likely that our model will be prone to overfit the training data and will not be too good at generalizing well, when confronted with new, previously unseen data. My model achieved a test accuracy of 35% and classified 298 of 836 images correctly.

For the second task of transfer learning a benchmark of 60% in accuracy must be outperformed. Here, a relatively high benchmark makes sense, as we don´t have to train the parameters of our pre-trained model, but instead can simply modify the output layer according to our needs. Here, my model achieved a reasonable test accuracy of 77% and classified 652 of 836 images correctly.

III. Methodology

Data Preprocessing

In this section, all my preprocessing steps will be documented.

Implementation

To use our dog image dataset some preprocessing was required in advance. The pictures have been resized with the help of the transforms.Resize function, then they were cropped with the transforms.CenterCrop function to obtain the required 224x224 format for the VGG16-Model, the same format I later used for my ResNet50 Model. I then transformed the images to tensors and normalized them with the following means and standard deviations: transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])]). The code below gives an overview on these steps:

```
def image_to_tensor(img_path):
    '''
    As per Pytorch documentations: All pre-trained models expect input images normalized in the same way,
    i.e. mini-batches of 3-channel RGB images
    of shape (3 x H x W), where H and W are expected to be at least 224.
    The images have to be loaded in to a range of [0, 1] and
    then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225].
    You can use the following transform to normalize:
    '''
    img = Image.open(img_path).convert('RGB')
    transformations = transforms.Compose([transforms.Resize(size=224),
                                          transforms.CenterCrop((224,224)),
                                         transforms.ToTensor(),
                                         transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                              std=[0.229, 0.224, 0.225])])
    image_tensor = transformations(img)[:3,:,:].unsqueeze(0)
    return image_tensor


# helper function for un-normalizing an image  - from STYLE TRANSFER exercise
# and converting it from a Tensor image to a NumPy image for display
def im_convert(tensor):
    """ Display a tensor as an image. """

    image = tensor.to("cpu").clone().detach()
    image = image.numpy().squeeze()
    image = image.transpose(1,2,0)
    image = image * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.406))
    image = image.clip(0, 1)

    return image
```

I also used several transforms functions in order to augment the given dog data. I implemented random horizontal flips and random rotations to make the model more robust and keep it from overfitting certain structures of our training data. This can be seen in the code snippet below.

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


# how many samples per batch to load
batch_size = 16

# number of subprocesses to use for data loading
num_workers = 2

# convert data to a normalized torch.FloatTensor
transform = transforms.Compose([transforms.Resize(size=224),
                                transforms.CenterCrop((224,224)),
                                transforms.RandomHorizontalFlip(),
                                transforms.RandomRotation(10),
                                transforms.ToTensor(),
                                transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])

# define training, test and validation data directories
data_dir = '/data/dog_images/'

image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x), transform)
                  for x in ['train', 'valid', 'test']}
loaders_scratch = {
    x: torch.utils.data.DataLoader(image_datasets[x], shuffle=True, batch_size=batch_size, num_workers=num_workers)
    for x in ['train', 'valid', 'test']}
```

Refinement

In this section, I will discuss the process of improvement I made upon the algorithms and techniques I used in my implementation. There a several hyperparameters I will go over in this section, including the number of epochs, the learning rate and the use of a momentum hyperparameter.

For the first model I tried several hyperparameters and compared the results in order to attain the highest accuracy and the lowest loss on the test data. It turned out that a comparably small learning rate of 0.001, this means only very small steps will be performed on each backpropagation step. To make up for the small learning rate I used the momentum concept, which is designed to speed up convergence of first order optimization methods like gradient descent. This trick essentially works by adding what's called the momentum term to the backpropagation update formula for gradient descent. As a result, its natural "zigzagging behavior" will be neutralized to some extent, especially in long narrow valleys of a cost function. A relatively high momentum of 0.9 worked best on my VGG-16 inspired model. Lastly, I ran 10 epochs which proved to be enough, as the model seemed to overfit the training data over time and thereby an increase of the validation loss could be observed.

This wasn´t the case for the second, pre-trained model. Here, I chose to run 20 epochs, as the outcome of the model consistently improved over time, in other words, it was kept from overfitting the training data and generalized well on the test data. As for the other hyperparameters, again a small learning rate of 0.001 worked very well and the momentum parameter was omitted as it didn´t add the desired benefit to the result.

In a last step, I implemented a predict_breed_transfer function were I again loaded and augmented the images, transformed them into tensor and got an output in the form of the probability of breeds with the help of a one-dimensional softmax. The code implementation can be seen below.

```python
def predict_breed_transfer(img_path):
    # class names without number (Affenpinscher, Brussels griffon...)
    # class names with number (001.Affenpinscher', '038.Brussels_griffon')
    class_names_without_number = [item[4:].replace("_", " ") for item in image_datasets['train'].classes]
    class_names_with_number = image_datasets['train'].classes

    # Load image
    img = Image.open(img_path)

    # Image Preprocessing
    transform_predict = transforms.Compose([transforms.Resize(256),
                                            transforms.CenterCrop(224),
                                            transforms.ToTensor(),
                                            transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                                 std=[0.229, 0.224, 0.225])])
    # Get Tensor
    img_tensor = transform_predict(img)

    # without the code below, it occurs an error !
    # RuntimeError: expected stride to be a single integer value or a list of 1 values
    # to match the convolution dimensions, but got stride=[1, 1]
    img_tensor = img_tensor.unsqueeze_(0)

    # Send the tensor to the device(GPU or CPU)
    img_tensor = img_tensor.to(device)

    # PyTorch's models need inputs as a form of Variable.
    # Variable is a wrapper of Pytorch Tensor.
    img_var = Variable(img_tensor)

    # Get output
    output = model_transfer(img_var)

    # Get the probability of breeds
    softmax = nn.Softmax(dim=1)
    preds = softmax(output)

    # Get three breeds which has the highest probabilities.
    top_preds = torch.topk(preds, 3)

    # Get the names of breed for displaying
    labels_without_number = [class_names_without_number[i] for i in top_preds[1][0]]
    labels_with_number = [class_names_with_number[i] for i in top_preds[1][0]]

    # Get the probabilities as a form of Tensor
    probs = top_preds[0][0]

    return labels_without_number, labels_with_number, probs
```

IV. Results

Model Evaluation and Validation

In this section, the two final models and any supporting qualities will be evaluated in detail.

As explained before, the two models worked well, compared with the given benchmarks of 10% and 60% respectively. A VGG-16 style CNN was the most efficient and slim way to obtain a relatively good result when training a model from scratch. The relative speed and the strong recognition patterns of several 3x3 sized kernels outperform most other CNNs when confronted with this task. An even deeper VGG-19 model would probably attain even higher accuracy scores, but for the purpose of this challenge a VGG-16 model has proved to easily reach results of 35% accuracy or even higher. In respect of the relatively small dataset the final model generalized

good on unseen data. After I implemented some small perturbations in training data the result indeed improved again with an increase of about 10% in accuracy. While this result might still not be enough to trust the model entirely for our classification task, it was still a solid foundation for the training of such a model from scratch.

The second model can in contrary be trusted, as it achieved a very solid accuracy score of 77% (17% higher than the benchmark provided by Udacity). This pre-trained ResNet model generalized very well on unseen data and attained very small amount of training and validation losses with 1.60 and 1.35, respectively. The model is therefore robust enough to solve the underlying problem and can be used with a relatively high prediction confidence.

Justification

In this section, my model's final solution and its results will be compared to the benchmark Udacity established earlier.

It was very pleasing to find that my models outperformed the relevant benchmarks of 10% and 60% by 25% and 17%, respectively. To answer the question whether these results and the solution are significant enough to have solved the problem posed in the project, I must admit I need to answer with a yes and a no. While it is true that the first model outperformed the benchmark be a solid 25%, other, much lower accuracy scores can be attained with the same model. So, while it still is a solid and valid approach to create a multiclass classification CNN from scratch, further improvements will have to be found to make the model more robust and to keep it free from the dependence on good feature finding processes by the models obtained by apparently mere luck.

This isn´t the case for the second, transfer-learning model. Here we have a very solid model which continuously decreases test and validation losses with each epoch running. The model can therefore by called significant enough to have solved the underlying problem. This model has been optimized quite well by careful hyperparameter tuning.

V. Conclusion

Free-Form Visualization

In this section, I will provide some form of visualization that emphasizes an important quality about the project.

I want to share some impressions on the final output of the second model. As you can see below, the model is very good in identifying humans as such and gives as the resembled breeds and its probabilities as a fun feature.

```
It's a human!
```

```
Resembled breeds and its probabilities:

Chihuahua
2.60%
Chinese crested
2.38%
Dachshund
2.03%


It's a human!
```

```
Resembled breeds and its probabilities:

Glen of imaal terrier
2.40%
Bedlington terrier
2.20%
Dandie dinmont terrier
2.16%
```

It also works reasonably well on dog images, as you can see below.

```
It's a dog!
```

```
Predicted breeds and its probabilities:
```

```
Bernese mountain dog
63.59%
Entlebucher mountain dog
5.60%
Greater swiss mountain dog
4.68%


It's a dog!
```

```
Predicted breeds and its probabilities:

Norfolk terrier
37.15%
Norwich terrier
9.40%
Cairn terrier
9.19%
```

Still, I need to admit I was a bit disappointed that the model also failed several times when being fed with some new, handpicked images of dogs from the internet. It failed to classify the dogs and its breed completely, which is probably due to the unusual nature of the pictures I chose in order to challenge my model.

```
I can't detect if it's a human or dog!
```

Reflection

In this section, I will summarize the entire end-to-end problem solution and discuss particular aspects of the project I found interesting or difficult.

While I have a firm understanding of deep learning networks and the processes carried out during training that happen under the hood, it is always challenging for me to preprocess the data in order to make it usable for a model. The use of the

transform functions and the transformation of the pictures to tensors was therefore the most challenging part for me.

I found the tuning of the model the most interesting process as it significantly changed the outcome when I didn´t expect it. Most of my understanding about deep learning models I gained from deeplearning.ai´s Deep Learning Specialization, where famous instructor Andrew Ng also stated, that it is often very difficult or even impossible to predict how a very deep model will react to changes on its hyperparameters. The finetuning of these is therefore a very iterative and interesting process.

Altogether, I am very satisfied by the results of my models and can say that the outcomes meet my expectations.

Improvement

In this section, I will provide discussion as to how one aspect of the implementation I designed could be improved. During the reflection process, I identified the following issues, which leave room for improvement to achieve even better results:

- The number of pictures which couldn´t be classified indicate, that our model needs more training material. With a rather limited number of training data our model is prone to overfit the data and therefore seems to have trouble when it comes to generalize the extracted features when it is confronted with new data.
- Generally, our model can be improved by finetuning its hyperparameters. So, trying to implement different values of the learning rate for example could minimize our loss on the validation set and give us a higher accuracy scores.
- Another approach that comes to mind is to apply different models, optimizers and loss functions. By doing so we can learn more about which model is best for our task. Because CNNs are quite in transparent under the hood, this iterative approach is often necessary, as it is rather difficult to find out which of these factors is the best one by mere reasoning.
- Finally, the code could be cleaned up and rewritten in a more modular manner.