Machine Learning Engineer Nanodegree

Project Report

Edson Romero
May 26, 2020

Definition

Project Overview

In this project I will use convolutional neural networks to identify dog breeds. Recent research in the past 5 years have made break throughs in image classification by using convolutional layers. These layers can extract features from images by using a kernel and applying convolutional operations. These features can be high and low level where high-level features signify edges and shapes of the semantic object and low-level features signify finer details such as eyes, nose, and mouth of a dog for example. Generally, the more convolutional layers and the more kernels per convolution we use the more features we can extract from images. This leads to big network architectures, commonly refer to as deep networks, which results in significant increases in training time, even when high-end computational resources are available. To reduce training time researchers have developed a technique known as transfer learning where a deep network has been trained on a large dataset, and to reuse this training we replace some layers towards the end of the network to fit our custom dataset and retrain. The idea is that this new network will have a better starting point during training and hence will achieve better performance than a similar network trained for the same number of epochs. To solve the dog breed classification problem, I will use transfer learning by exploiting the architecture of the VGG16 model pretrained on the ILSVRC-2014 dataset which is a subset of ImageNet containing 1.2 million images and 1000 categories.

Problem Statement

Our problem of dog breed classification is very specialized as the model would have to learn to distinguish the difference between similar looking animals. That is, every dog has eyes, a nose, ears, fur, four legs and some breeds can be very difficult to tell apart for the human eye. Thus, for application purposes, we desire our model to achieve superhuman performance. By using deep networks and transfer learning I hope to achieve this goal. As stated one possible solution can be to use the VGG16 model pre-trained on the ILSVRC-2014 dataset. This model was trained for 2-3 weeks using four NVIDIA Titan Black GPUs and achieved first place in the ImageNet classification task challenge in 2014 with an accuracy of 92.7% [1]. Luckily, it is packaged up in PyTorch and hence is readily available for transfer learning for our dog breed dataset. The model will be evaluated by its accuracy score on the classification task and will be compared to a benchmark model as a sanity check. The benchmark model is a network built from scratch using PyTorch with a similar architecture as VGG16 but much smaller in size. It will be trained for the same number of epochs and with the same optimizer as the pre-trained

model. The expectation is that the pre-trained model should easily outperformed the benchmark model.

Metrics

The evaluation metric of choice is accuracy for both the benchmark and pre-trained model. This metric is appropriate as it is used in evaluating models in the yearly classification challenge of ImageNet.

Analysis

Data Exploration

Since the dataset is already obtained and annotated from a reputable source, ImageNet, the workflow of the project begins with the transformation of the dog breed dataset in order to feed it into the models. Transformation refers to image resizing, scaling, normalization and later augmentation for improved learning. As a small aside, viewing a sample of the images in the training set we see some complications; presence of humans, multiple dogs, and dog clothing. This could make it harder for the model to learn dog features since there are now more semantic objects. However, since real world images may have these complications in them we will keep them with the hope to generalize better. Below are some examples of these complications,

Figure 1: Images in the training set with humans, multiple dogs, and/or dog clothing.

Exploratory Visualization

One of the techniques we will use is image augmentation. This is a random affine transformation applied to the image pixels. In this application, I applied rotations, translation, and horizontal flipping of the images to some scale with randomness. Rotation was applied randomly with a maximum of 20 degrees, that is a random number was chosen from 0 to 20 and used as the degree of rotation. Likewise, translation was applied with a maximum of 20% of the image size in both vertical and horizontal directions, and horizontal flipping was applied with a probability of 50%. This is done so that the model can generalize better and hence avoid overfitting. Examples of images after this affine transformation is shown below,

Figure 2: Examples of images from the training set after image augmentation.

Algorithms and Techniques

The main technique used in this project is transfer learning where a pre-trained model is retrained to a custom dataset and is expected to outperform a model trained from scratch. Transfer learning is a popular technique used for many applications as it significantly reduces training time and can eliminate the need for high-end computational resources. This is appealing since it essentially reduces the time and cost for training a model and still provides state-of-the-art performance. In this case, I will reuse the architecture of the VGG16 model trained for 2-3 weeks on the ILSVRC-2014 dataset which achieved an accuracy of 92.7%. So, I hope to get close to that result in just a few hours. More details of the implementation of the pre-trained model are given in the implementation section.

We want our two models, pre-trained and benchmark, to perform as best as possible. Thus, as is standard, we will scale image pixel values from 0 to 1 and then normalize them by subtracting the mean and dividing by the standard deviation per color channel. So, the mean and standard deviation are vectors of dimension 3. We do this so that pixel values are small and centered which helps in training.

Then I will try to improve the two models as much as possible by using image augmentation on the input and adding dropout layers. Both of these techniques help the model generalize better avoiding overfit. Image augmentation helps by making the model recognize the semantic object in different orientations and sizes. The dropout layer sets the output of neurons to zero with a probability *p*. This helps the model avoid overfitting by diminishing the correlation between neighboring neurons and by forcing the model to learn to use all its neurons for classification instead of just relying on a few. So, if a neuron outputs a large activation value for classifying a certain dog breed and dropout sets it to zero, the model would have to rely on other neurons for the classification.

For the sake of increasing training speed, I will use batch normalization layers. This layer helps to reduce internal covariate shift of the inputs to the following layer [3]. Intuitively this

helps the hidden layer in the same way as normalizing the image pixel values helps the first layer of the network. Note that the effects of batch normalization are an active area of research. It has also been stated that the effects are not due to reducing internal covariate shifts but rather to that of smoothing out the loss function such that gradient descent gives a better gradient for optimizing [4]. This in turn allows for higher learning rates for faster training, a known result of batch normalization. Also, I tried using modern optimizers for training such as Adam and RMSprop, however, PyTorch's Stochastic Gradient Descent (SGD) with a high learning rate proved to be faster for training in this application. Note that PyTorch's SGD also implements Nesterov momentum.

Benchmark

A benchmark model was created to compare the results of the pre-trained model. The benchmark model was built using PyTorch and trained for the same number of epochs and with the same optimizer as the pre-trained model. We expect the pre-trained model to easily outperform the benchmark model in accuracy. The architecture will be similar in design to that of VGG16 but smaller in number of layers and neurons since we don't have as powerful GPUs in the workspace and don't want to train for 2-3 weeks. The architecture of VGG16 is of 5 convolutional sets each followed by a max-pooling layer which is then followed by 3 dense layers. Each convolutional layer set increases the channel size of the image features going from 64 channels of dimensions 224 x 224 in the first convolution to 512 channels of dimensions 7 x 7 by the last. In a simpler design, the benchmark model will have 3 convolutional layers each followed by a max-pooling layer and increasing the channel size from 16 to 32 and then to 64. This is followed by 3 dense layers of sizes 1024, 512 and 113. This benchmark model turns out to be sophisticated enough to achieve a high accuracy of 97% on the training set as shown in the figure below,

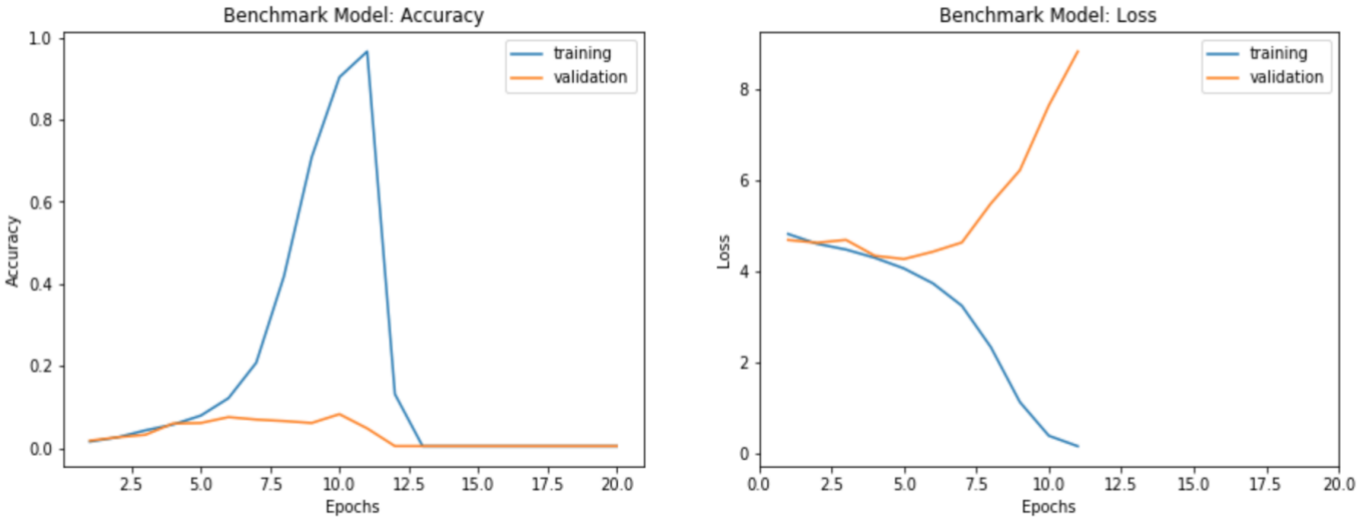

Figure 3: Training plots for accuracy and loss over 20 epochs for the benchmark model. The model is able to achieve a 97% accuracy on the training set at epoch 11. This is clearly the result of overfitting since the validation accuracy doesn't increase much. After epoch 11, the accuracy

dies off and the loss values become NaN (not a number). This could be the result of numerical instability from very large activation a common characteristic of overfitting.

Inspecting figure 3, we see that the model is able to fit the training data but is overfitting since the validation accuracy only achieves a high of 8%. Also, the accuracy on the test set was only 6%. This tells us that the model is complex enough to learn a mapping from image to dog breed but is unable to properly learn the data patterns that entail each dog breed. After epoch 11, the loss values become NaN (not a number) which could be the result of numerical instability from large activation values, a common characteristic of overfitting networks. In classical machine learning, this overfitting problem is due to the model being overparameterized and fitting the noise in the training data. A solution to this problem is to make the model less complex or smaller. However, empirical results in deep networks tell us that making the model more complex, such as adding more layers, can result in better performance. In this case adding supportive layers such as batch normalization and dropout helped improve the validation accuracy. The results of the improved benchmark model as well as its architecture are discussed later in the refinement section.

Methodology

Data Preprocessing

As mentioned before, since the dog dataset comes from a reputable source, ImageNet, no data cleaning was performed. The pixel values of images were scaled from 0 to 1 and then normalized per channel. The images were then fed into the models in batches of size 32. The images were also resized depending on the model. The pre-trained model of VGG16 expects images of size 224 x 224 x 3. The benchmark model takes images of size 128 x 128 x 3; the image size is smaller to speed up training time since this model is trained end-to-end. Later, I improved both models as much as possible by means of image augmentation and by additional means in the case of the benchmark. Augmentation was defined as a random affine transformation applied to the image pixels which include operations such as rotation, translation, and horizontal flipping of the images to some scale with randomness. This is described in more detail in the Exploratory Visualization section.

Implementation

Two models were trained on the dog dataset; the pre-trained model created from transfer learning using the VGG16 architecture and the benchmark model created from scratch with a similar design. Ideally, the benchmark would be the VGG16 model trained end-to-end but because training such a large model is expensive a simpler model was used instead. Note that VGG16 was trained for 2-3 weeks with more powerful GPUs than those available in the workspace.

To create the pre-trained model, we first consider how VGG16 was trained. The original model was trained on a large dataset ILSVRC-2014 containing 1.2 million images and 1000

categories while our dog dataset is small containing 8,351 images and 113 categories. Both of these datasets are somewhat similar as they contain four-legged animals. Therefore, the type of transfer learning we will use is that of replacing the last layer with a new layer of the same number of neurons as the number of dog breeds. Then all layer weights will remain fixed during training except for those of the last layer. This type of transfer learning is done to avoid overfitting to the dog dataset which is a problem with smaller datasets. Intuitively, since the datasets are similar, high end and low-level features extracted from convolutions should be transferable from one model to another and hence the weights from convolutions are fixed.

To create a benchmark model for comparison purposes, we design a similar architecture to that of VGG16 but smaller in size. That is, we have the same number of convolutional sets but with less convolutional layers and less kernels. The same is true for dense layers where we reduce the number of neurons per layer. The architecture of the benchmark model is described in more detailed in the benchmark section.

In order to compare the two models, they were trained for 20 epochs and used the same optimizer, Stochastic Gradient Descent (SGD) with momentum with a learning rate of 0.01. The training of the pre-trained model (Figure 4) looks more promising to that of benchmark model (Figure 3). It achieves 91% and 87% accuracy on the training and validation set, respectively,

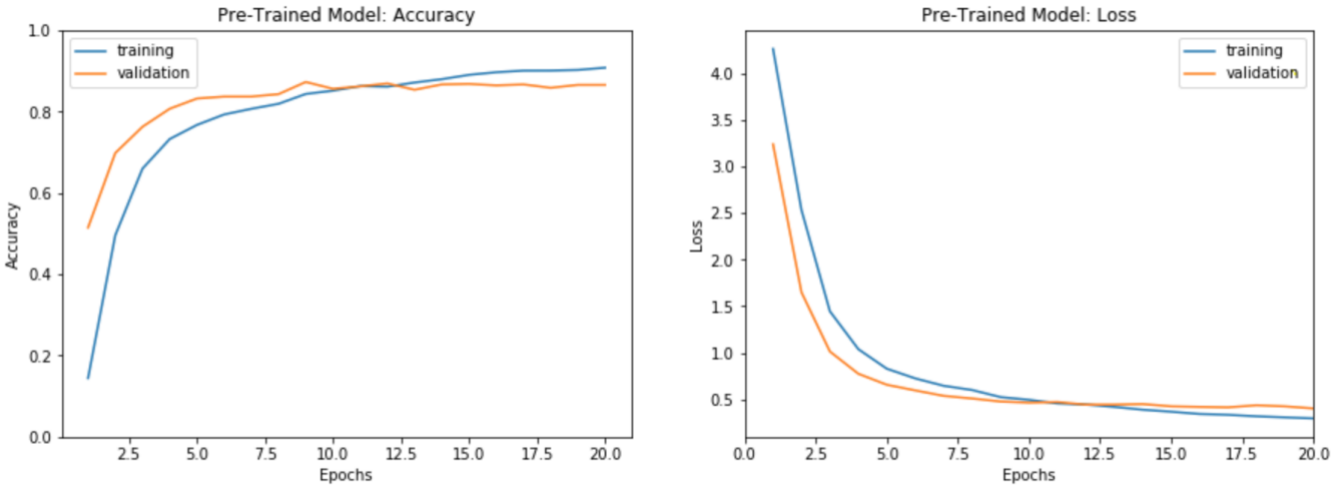

Figure 4: Training plots for the pre-trained model created from transfer learning via the VGG16 architecture. It achieves a 91% and 87% accuracy on the training and validation set, respectively. The model achieved 86% accuracy on the test set.

Although the training accuracy and training loss continue to improve after epoch 20, the validation metrics seem to have leveled off. In an attempt to improve this, I added image augmentation to the pre-processing step which is discussed in the refinement section. One last thing to note is that 20 epochs equated to a little less than an hour of training. This is pretty

significant given that the original model achieved a 92.7% accuracy on its test set by training for 2-3 weeks and we achieved 86% on our test set.

Refinement

To demonstrate the importance of transfer learning, it is of interest to improve the benchmark model as much as possible. We have seen that the pre-trained model easily out performs the benchmark but is the difference significant if we try to create the best benchmark possible given our GPU resources and learning techniques available? Likewise, in a real world setting the pre-trained model would be improved as well. So, in this section I improve both models.

To improve the pre-trained model, we only have the option to pre-process the data since by design the model is fixed and the last layer must output the scores for classification and hence cannot be altered. A common technique in pre-processing is image augmentation on the training set. This random affine transformation on the pixel values resizes and shifts the semantic object around the image. This forces the model to learn to identify the object in different orientations and sizes which helps to avoid overfitting. However, after many attempts trying out augmentation parameters, I could not get an improvement on accuracy, below are the training plots,

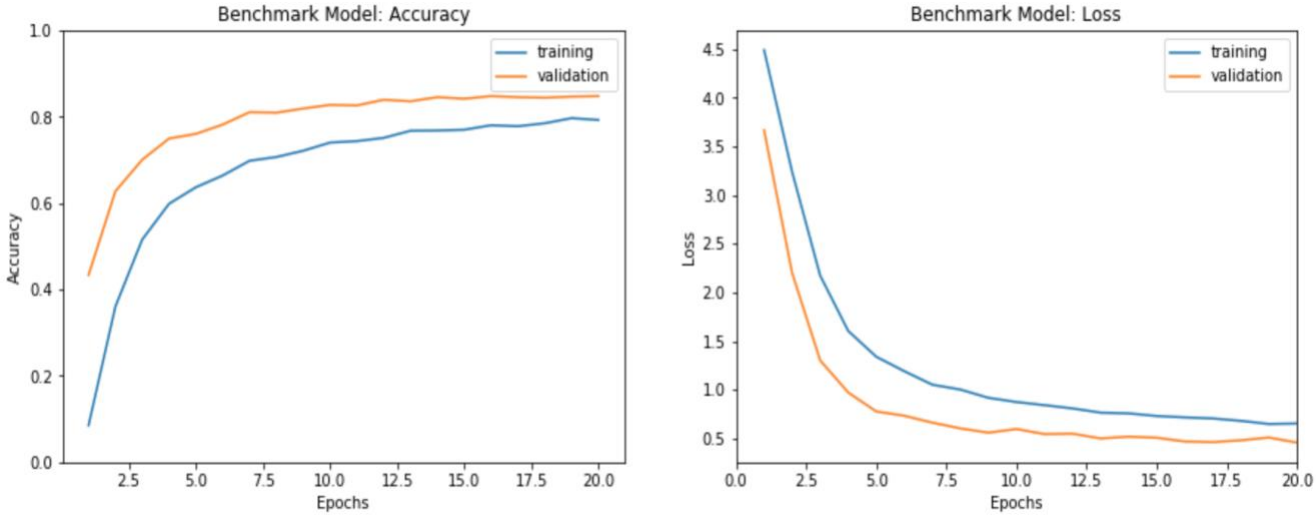

Figure 5: Training plots for the pre-trained model with image augmentation. Note that the model is having a harder time classifying the training set due to the affine transformations. The validation accuracy still levels off after 84%. The test set had an accuracy of 85%, which is not significantly different to the 86% accuracy on the test set without image augmentation.

To improve the benchmark, in addition from image augmentation, I added batch normalization and dropout layers. The one batch normalization layer was added after the third convolution but before its relu activation function. Then, after the third's relu, a dropout layer was added. In the dense layers, each one had a batch normalization added after it but before its

relu. No dropout layers were added in the dense layers. The design choice here was partly trial and error to see what worked best. Adding batch normalization layers to each convolution was expensive thus I only added it to the last one. Adding too many dropout layers reduces the networks ability to learn thus I only added it to the last convolution layer as well. Since the convolutional layers are responsible for extracting image features I figured it would be best to apply these supportive layers to the final output of the convolutions. The batch normalization on the dense layers were not expensive and improved training time. Below are the plots for the training of the improved benchmark,

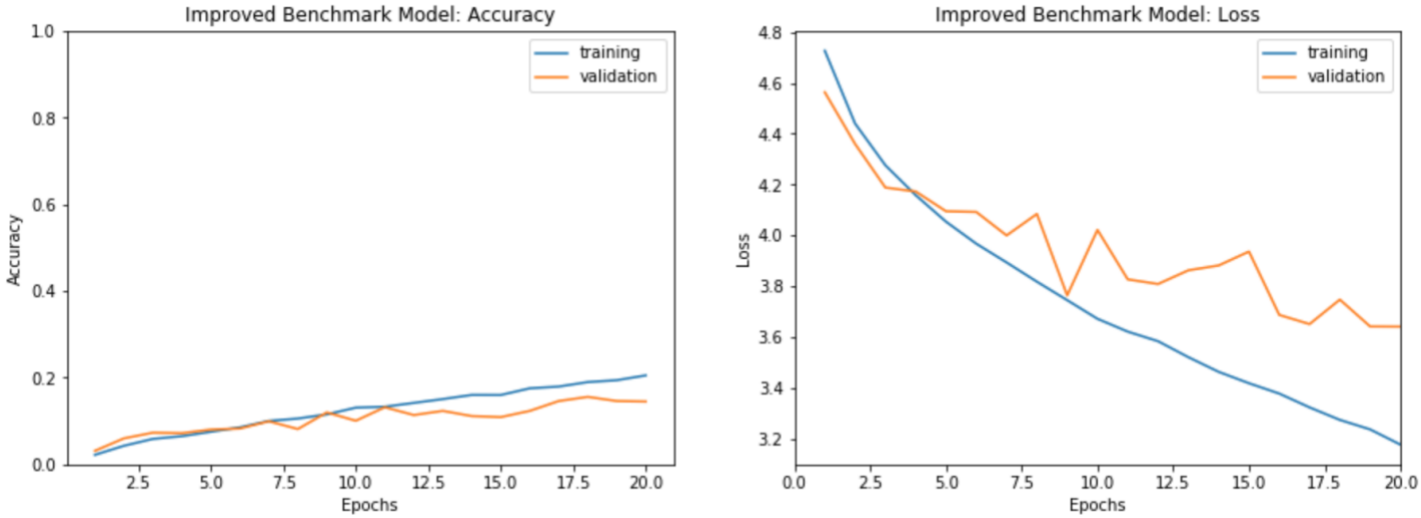

Figure 6: Training plots for the improved benchmark model with image augmentation, batch normalization and dropout. The accuracies after 20 epochs are 21% and 14% for training and validation sets. Both training and validation seem to continue improve after the 20th epoch. The test accuracy was 17%.

The test accuracy of the improved benchmark is 17% which is significantly higher than the original benchmark's 6%. Both training and validation have an increasing trend after the 20th epoch so it would be interesting to see how far it would go before leveling off. However, training per epoch is now much slower because of the additional layers and image augmentation.

Results

Model Evaluation and Validation

The best benchmark model could only achieve a 17% test accuracy after 20 epochs while the pre-trained model achieved an 86% test accuracy for the same number of epochs. This demonstrates the advantage and importance of transfer learning. Furthermore, this also simplifies the development of models for state-of-the-art performance. Within a few hours of training and modifications, the pre-trained model was ready for use. As oppose to the improved benchmark model which required trail-and-error layer modifications and some intuition behind

design choices. It is noteworthy that empirical results of deep networks are ahead of its theory and thus designing a network can be a time consuming task.

Justification

Not all developers have access to expensive GPU resources and a training window of weeks could be hard to schedule. Additionally, the training time per epoch can be smaller in transfer learning when not all layer weights are being updated. Hence this technique is invaluable in a rapid development environment where many iterations of a product are required in a short period. Furthermore, from a community perspective, this allows for sharing of models and ideas easily. A pre-trained architecture can be modified with new structural ideas and retrained to improve performance on an already state-of-the-art model.

Citations

1. https://arxiv.org/pdf/1409.1556.pdf
2. http://vision.stanford.edu/aditya86/ImageNetDogs/
3. https://arxiv.org/abs/1502.03167
4. https://arxiv.org/abs/1805.11604