527 Final project

Andy Chen, Coco Luo, Yiling Chen

June 4, 2022

Multiclass classification on Stanford Dog Breeds dataset

Github Repository: https://github.com/yilingchenn/stat527_final

Background

Deep learning provides many successful machine learning results to solve real world problems. Among all those problems, image classification is a crucial part in computer vision. It can be implemented using both unsupervised learning methods such as hierarchical clustering, k-means clustering, maximum likelihood algorithms, which tries to find patterns in a dataset without labels. And supervised learning methods, which learns to map a function from the labeled training data to an output.

In our project, we built convolutional neural network-based image classifiers to identify and separate images of different breeds of dogs. We first defined a model that defeats the baseline and then applied 3 different pretrained models: EfficientNet, VGG16 and Xception to better classify the images and make predictions using the test data. Those pre-trained neural networks were useful for transfer learning, but the training process could take expensive computation power. The objects of the project were to show how to build a real-world convolutional neural network, and to compare the pretrained model based on their performance on the prediction task.

Method and Analysis

The Stanford Dogs dataset contained 120 breeds of dog images from around the world, where there are a total of 20580 unbalanced number of image data for each class, ranging from about 150 to 200. It was originally collected for fine-grained image categorization, a challenging problem as certain dog breeds have near identical features or differ in color and age. Since we have limited access to an effective GPU, we decided to train only 30 classes.

The model structure we used is a convolutional neural network, which took as input tensors of shape (image_height, image_width, image_channels). In our case, that would be (224, 224, 3). This input were directly passed to the first convolution layer. And the last output tensor was directly fed into a densely connected classifier network, which processes 1D vectors. We therefore needed to turn the 3D output from the convolution layer into 1D vector via a flatten function.

A first convolution layer learns small local patterns, a second convolution layer learns larger patterns made of the features of the first layers, and so on, allowing convnets to efficiently learn increasingly complex and abstract visual concepts. Convolutions operate over feature maps with width, height and depth (channels). Because we have colored images, our depth is 3 (red, blue and green). The convolution operation extracts patches from the input feature map and applies the same transformation to all of the patches, introducing an output 3D feature map. The different channels in the depth axis of the output then stands for filters, which encode specific aspects of the input. We also have maxpooling layers followed behind each convolution layer. Max pooling consists of extracting windows from the input feature maps and outputting the max value of each channel. It is conducive to apply those layers as they can not only mitigate severe overfitting problems by cutting down the total number of parameters to be learned by the model,

but also enlarge the size of the filer so that we can focus on a bigger part of our interest (Chollet 123).

We looked for accuracy scores and areas under the curve to evaluate the goodness of our model. As the dataset was downloaded from Kaggle, we preprocessed the data through the Keras preprocessing packages. We used a 7:1.5:1.5 ratio to split the images into a training, a testing and a validation dataset, and then compiled a model that defeated the baseline. The baseline model uses an Adam optimizer to minimize the categorical cross entropy loss. We defined the batch size to be 20 and 10 epochs.

We further tried different transfer learning models, ResNet50, EfficientNet, and VGG16, hoping that one of the models would give a better performance. ResNet first introduced the concept of skip connection, where we stack convolution layers by adding the original input to the output of the convolution block. Skip connection can mitigate vanishing gradient by allowing this alternate shortcut path for gradient to flow through. And they allow the model to learn an identity function which ensures that the higher layer will perform at least as good as the lower layer, and not worse (He, et al).

EfficientNet provided a family of models (B0 to B7) that represents a good combination of efficiency and accuracy on a variety of scales through introducing a heuristic way to scale the model (Tan and Le, 2019). With such scaling heuristics, the efficiency-oriented base model (B0) is able to surpass models at every scale, while avoiding extensive grid-search of hyperparameters. The model takes input images of shape (224, 224, 3). Normalization is included as part of the model.

Visual Geometry Group (VGG) is a simple and widely used convnet architecture for ImageNet that supports up to 19 layers, and addresses the depth of CNN. VGG16 consists of 16

weight layers including thirteen convolutional layers with filter size of 3 x 3 with stride 1, and fully-connected layers with filter size of 3 x 3. It takes in a 224 x 224 pixel RGB image, and could also incorporate 1 x 1 convolutional layers to make the decision function more nonlinear without changing the receptive fields. It always uses the same padding and maxpooling layer of 2 x 2 with stride 2, and follows this arrangement throughout the whole architecture.

Using the convolution base of each of the pretrained model, we proceeded by adding the same dense layers on top and running the whole thing end-to-end on the input. We adjusted the size of our hyperparameters accordingly and tried multiple regularization methods such as data augmentation, batch normalization and dropout to mitigate overfitting. Data augmentation could generate more training data from existing training samples, by "augmenting" the samples via a number of random transformations that yield believable-looking images. Unfortunately, we were unable to perform those on a mini-batch before feeding it to our model as we don't have a powerful GPU. Thus, we were determined to train the models without data augmentation at first, and see if we needed it later.

Dropout is another effective and most commonly used regularization technique for neural networks. Dropout consists of randomly setting a number of output features of the layer to 0 during training. At test time, the layer's output values are scaled down by a factor equal to the dropout rate, to balance for the fact that more units are active than at training time. We tried different dropout rates at first. Since it seems that different dropout rates did not change much to the performance of our models and the overfitting problem, we selected 0.5 as the dropout rate for our network. We also used batch normalization to speed up training of different combinations of hyperparameters needed to optimize the use of dropout layers (He et al.). This technique also helped our models converge faster with additional regularization effects.

Results & Discussion

For the first part of the project, we successfully defeated the baseline line of 0.03 accuracy from a small model with two convolution layers (with 16, and 64 filters respectively), two maxpoling layers, one hidden layer with 64 nodes and one output layer with 30 nodes.

It took us a great amount of time to do transfer learning since we encountered multiple issues with our GPU. Moreover, we initially added too many transformations during data augmentation and trained for too many epochs (we used 100 initially) which caused the model to run extremely slow. As a result, we restricted the input to only 30 classes and ran the data again with only dropout and batch normalization. We also reduced the number of epochs to 10 and slowly increased it so that we could gain some insight on the general performance of our model.

To build better models, we tried the ResNet50 as the convolution base first. It seems that no matter how we tuned the hyperparameters and applied regularization techniques, the validation accuracy kept extremely low. From the figure, we could see that the training accuracy increased a lot while validation accuracy stayed flat, showing a severe overfitting (fig 1). We then decreased the number of layers and nodes and adjusted the regularization methods, but overfitting did not improve much.

ResNet50 is so deep that it took a long time to run, even on the local GPU. We then decided to try EfficientNetB0, but it also ran very slow, even slower then ResNet50. There are also a couple of limitations presented in EfficientNet, for one thing, resolutions were not divisible by 8, 16, etc. cause zero-padding near boundaries of some layers which wastes computational resources. This especially applied to smaller variants of the model. As a result, the input resolution for B0 and B1 were chosen as 224 or 240 (We chose 224). Depth and width limitations required the building blocks of EfficientNet demand channel size to be multiples of 8.

Moreover, memory limitation might bottleneck resolution when depth and width can still increase.

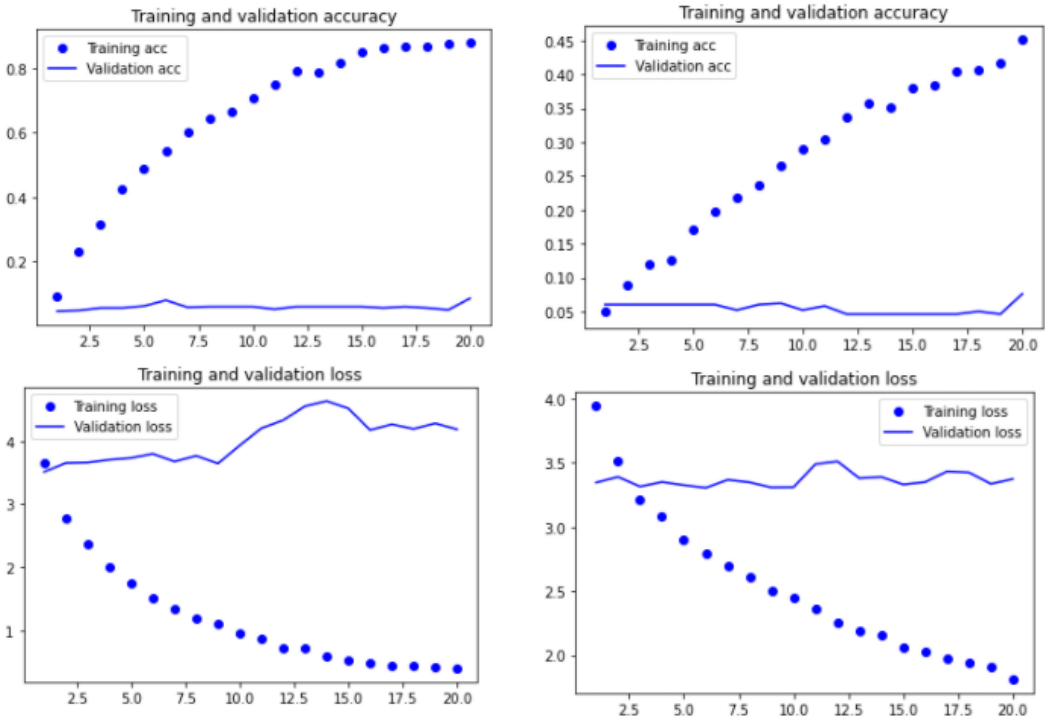

Figure 1: Training/ validation loss and accuracy for ResNet50 with 2 dense layers and 3 dense layers.

We finally trained VGG16, and experimented using different learning rates (0.001 and 0.05). Fig.2 showed that when we use 0.001 learning rate, there is a decrease in validation loss and an increase in validation accuracy. However, overfitting still exists since the training accuracy kept increasing after 20 epochs while validation accuracy became stable. Then we used a larger learning rate to facilitate the training, but it led to a fluctuated increase in validation accuracy, and an abnormal peak in validation loss.

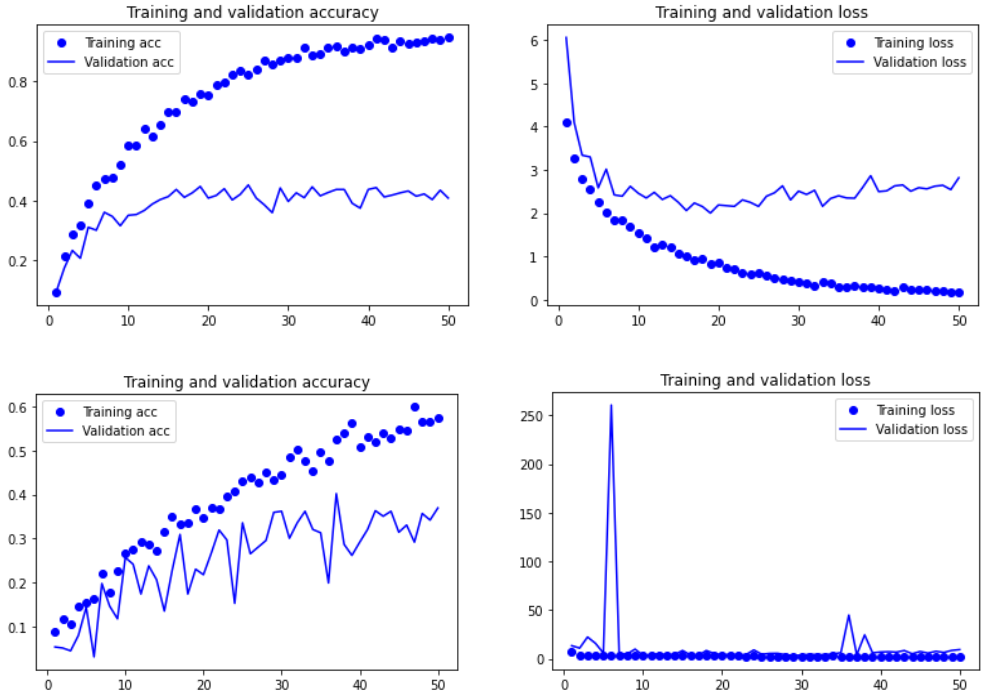

Figure2: Training/ validation loss and accuracy for VGG16 with RMSprop optimizer of 0.001 and 0.05 learning rate

Although we were not able to adjust the pretrained models as we want due to the ineffective GPU and limited local memory, we found that they provided much higher training accuracy (about 90%) compared to the simple baseline. The best training accuracy we got is 94% from VGG 16 with a test accuracy of 45% when doing 50 epochs. Although we could only run 20 epochs for ResNet50, we have already got a training accuracy as high as 88%, and we also found that the training accuracy of ResNet is a lot higher than VGG16 when 20 epochs is used, Thus, we thought that ResNet might be a better model with additional benefit on solving the vanishing gradient problem. The higher training accuracy of ResNet50 made sense. Although the size of the models are similar, ResNet is much deeper. Moreover, even though we got a very flat validation curve (a bad validation accuracy) for ResNet, we believed the severe overfitting

problem could be fixed or improved a lot when a combination of effective regularization methods were applied. In addition, EfficientNetB0 seems to provide an even higher training accuracy than ResNet50, and it is able to fuse multi-scale features effectively. We would not choose it for our dog breeds classification task. For one thing, the time and computation costs were too high (it took much time to train for even 1 epoch). For another thing, there were too many limitations presented in the application of EfficientNet, which were mentioned previously in our analysis. In reality, as we do not have effective local GPUs, and we were not aiming at gaining a near-perfect prediction score for this project, we didn't consider it as a better model for us to use compared to ResNet50 and VGG16.

References

Bengio, Yoshua., et al. “Dropout vs. Batch Normalization: an Empirical Study of Their Impact to Deep Learning.” *Multimedia Tools and Applications*, Springer US, 1 Jan. 1970, link.springer.com/article/10.1007/s11042-019-08453-9.

Brock, Andrew, et al. “FreezeOut: Accelerate Training by Progressively Freezing Layers.” *ArXiv.org*, 18 June 2017, arxiv.org/abs/1706.04983.

“Chapter 5.” *Deep Learning with Python*, by Chollet François, Manning Publications Co., 2018.

He, Kaiming, et al. “Deep Residual Learning for Image Recognition.” *ArXiv.org*, 10 Dec. 2015, arxiv.org/abs/1512.03385.

S. Zagoruyko and N. Komodakis. Wide residual networks. arXiv Preprint arXiv: 1605.07146, 2016.

Tan, Mingxing, and Quoc V. Le. “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks.” *ArXiv.org*, 11 Sept. 2020, arxiv.org/abs/1905.11946.

Team, Keras. “Keras Documentation: Image Classification via Fine-Tuning with EfficientNet.” *Keras*, keras.io/examples/vision/image_classification_efficientnet_fine_tuning/.